MapReduce Technical Report 1.0

Wenda Li

1. MapReduce Architecture Overview

1.0 Golang Implementation. Our implementation is in Golang; for specs, see https://go.dev/doc/. The RPCs will be implemented in the built in RPC library of Go, specifically, our handler of RPC will be

```
func call(rpcname string, args interface{}, reply interface{}) bool
```

which returns false if RPC fails and throw exceptions accordingly.
The worker and coordinator processes we spawned will be implemented as goroutine; see: https://golangdocs.com/goroutines-in-golang. We chose to use the abstraction of goroutine because:

a. Goroutine natively supports safe channel communication;
b. Light costs. Spawning a goroutine only costs ~2KB.

For Applications utilizing the MR library, we load the applications as plugins, see https://pkg.go.dev/plugin. ***Note that till now Golang plugin only supports Linux, MacOS and FreeBSD***. Go plugin supports the independent publication of new features and applications.
To build the plugin, for instance, we wish to run word count applications by MR library, we first enter src/main and run

```
>> go build -race -buildmode=plugin ../mrapps/wc.go
```

This will give you a wc.so file, and we can use this wc.so file to build the MR package:
First, we run the master node…

```
>> go run -race mrcoordinator.go pg-*.txt
```

Then we can spawn the worker process…

```
>> go run -race mrworker.go wc.so pg-*.txt
```

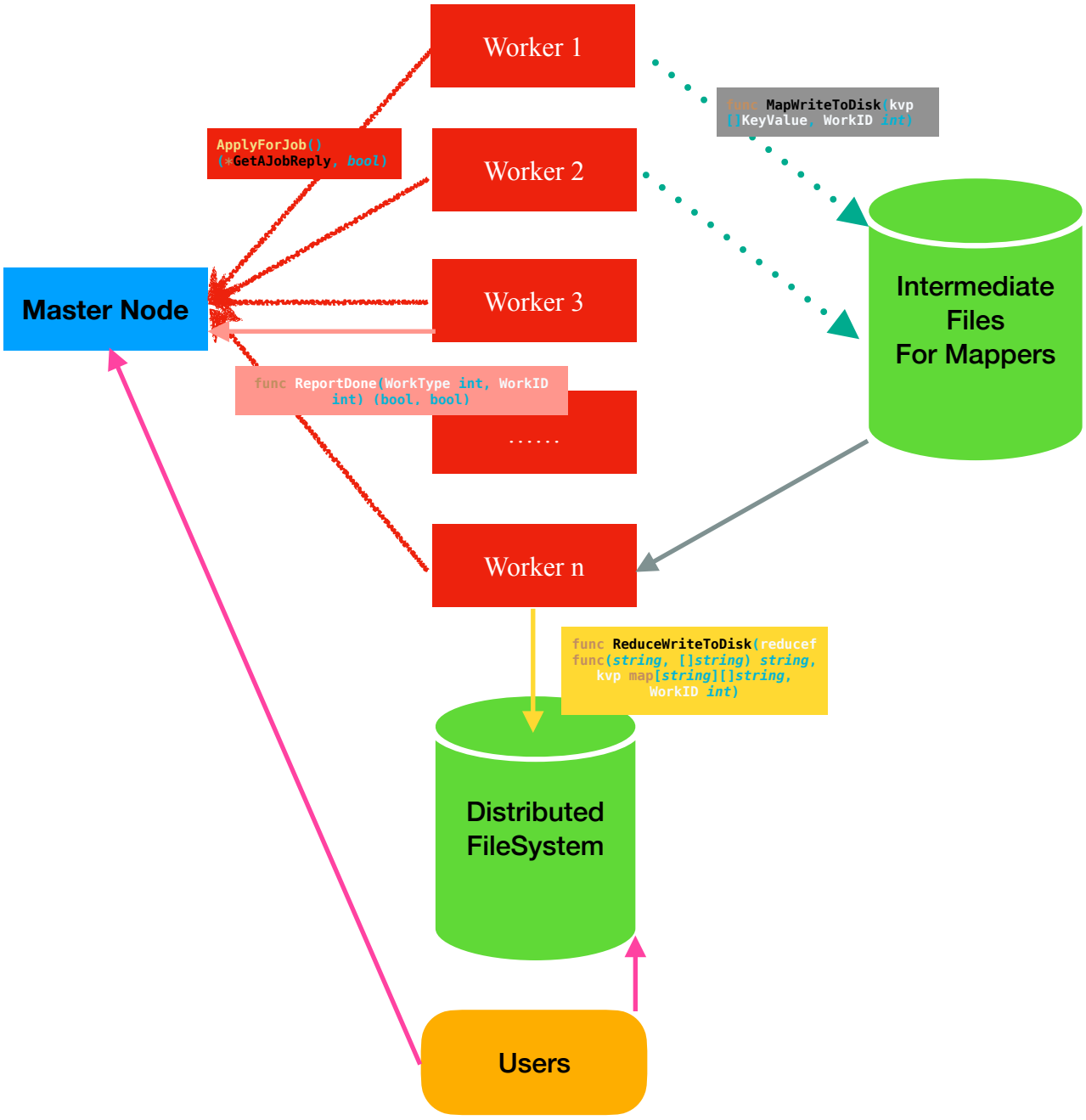
We restrict the limit size of possible data block distributed to every workers to be 20MB.
The current version largely borrows the designs from [2], Lab1.

1.0.0 ⚡ Warning. For the current implementation, there's a minor issue that our file system is still not truly distributed yet. Therefore the current implementation, as of the default form, will use the current single OS filesystem. However, if one uses the commercial or open source distributed system, e.g., KVRaft or Memcached, the current MR will be a working distributed MR demo. Fine tuning for work environment before final deployment is still needed. We will add a ⚡ whenever a truly distributed filesystem could potentially show difference so the current design should be read with a grain of salt.

1.1 A Sketch of Architecture. For a detailed account, see [1]. Our implementations of MR system prompts the spawned workers to message coordinator to apply for works to do and report the work, if the work is done. More specifically, worker processes whenever spawned, will prompt the master node to distribute jobs by periodically RPCalling `func ApplyForJob() (*GetAJobReply, bool)`, with WorkerID within Args and {WorkID, WorkType, File}, where WorkType could be Map, Reduce or Idle or Terminating, WorkID is the ID of the assigned piece of work, and finally File is the path of file, or file descriptor ⚡.

On the coordinator's side, whenever is `func ApplyForJob() (*GetAJobReply, bool)` called, RPC handler will try to contact `func (*Coordinator).GetAJob(Args *GetAJobArgs, Reply *GetAJobReply) error`, and the coordinator will then spawn a goroutine to wait for the dispatched worker. However, if over 25s the work is still not done, the work will be assigned to another worker and the current worker will be back to idle; heuristically, this should be able to reduce the dragging workers, see [1].

The general assembly of processes will be following.



1.2 Miscellaneous Points in Implementation of Reducers and Mappers. Order guarantee in 4.2, [1] is implemented in `func ReduceWriteToDisk(reducef func(string, []string) string, kvp map[string][]string, WorkID int)` where it tries to sort the keys in dictionary order before flushing to disk.

2. Communication Protocols and Data Structures

Communication Protocols are mostly declared in rpc.go.

2.1 Types for Work. Every piece of work is of the form {WorkerID, TaskID, WorkType, WorkStatus, File}, which characterizes the ID of worker who is assigned for this work, the ID of the work, the type of the work, the status of the work and the the file path associated to the work. The types of the work are: Map, Reduce, Idle, Kill, while the status of the work are: NotAssigned, Running, Done.

```
type Work struct {
    WorkerID   int64
    TaskID     int64
    WorkType   int
    WorkStatus int
    File       string
}
```

2.2 Types for Master Nodes. Master node is responsible for coordinating workflow, therefore to handle concurrency, we will include sync.Mute variable mu to lock the memory in use when necessary. Meanwhile, Master node will bookkeep a list(as well as the number of undone jobs) for Map works and Reduce works. The declaration is following.

```
type Coordinator struct {
    mu           sync.Mutex
    Map          []Work
    nTotalMap    int
    Reduce       []Work
    nTotalReduce int
}
```

2.3 Communication Protocols for Applying Jobs and Reporting. All RPC calls will be the following form:

```
func (t *T) MethodName(argType T1, replyType *Reply) error
```

Therefore when the workers try to apply for jobs, they will pass through RPC call a message of type GetAJobArgs, which contains a int variable recording the pid of worker process; when the coordinator try to respond, the reply will be of type GetAJobReply whose member variables WorkID contains the ID of the assigned task, WorkType describing the type of the work as specified in 2.1, and finally the file path associated to work.

For a worker process, it reports the status of the job by sending in a message of type ReportArgs, with specified worker's ID, Work's type, and work's ID, while the coordinator sends back to tell if the work process should terminate, should all the works are done. Moreover, we use partition function `ihash(key string) int` to hash the distribution of Reduce works, see 4.1, [1].

2.4 Socket Domain Name Protocols. In `func coordinatorSock() string`, we cook up a unique socket domain name in /var/tmp for the Master node. The naming fashion goes like: /var/tmp/mr-pid, where MakeCoordinator() will set pid to be the pid of Master node.

2.5 Intermediate File and Storage Name. We set the intermediate storage file name as TmpStore, for the use of intermediate storage, see Section 1.1. The naming convention for output files is mr-*.

2.6 User Interfaces. A simple exemplary implementation of user interface is implemented in mrcoordinator.go and mrworker.go.

3. Fault Tolerance

[To be done]

4. Examples and Tests

To test, run in `src/main$: bash test.sh` This generates all the tests listed in following. 4.1 and 4.2 check the correctness of the MR library, while 4.4 checks for parallelism. Finally 4.3 checks how the MR library deals with crash.

4.1 Word Count Implementation is in src/mrapps.

4.2 Indexer Implementation in src/mrapps

4.3 Crash and Delay Tests Implementation in src/test, crash.go. This gives a Mapper and Reducer that sometimes crash or delay for a long time.

4.4 Sanity Check. Mappers and Reduces pqarallelisms are checked in mtiming.go and rtiming.go.

References

1. *MapReduce: Simplified Data Processing on Large Clusters*, OSDI 2004, J. Dean and S. Ghemawat, *Google Inc.*
2. MIT 6.824 2022: https://pdos.csail.mit.edu/6.824/