MapReduce Framework
Final Report

Derek Tzeng
Carnegie Mellon University
Pittsburgh, PA 15213
dtzeng@andrew.cmu.edu

Yiming Zong
Carnegie Mellon University
Pittsburgh, PA 15213
yzong@cmu.edu

Abstract

This report reflects our final progress on MapReduce Framework project for 15-440, Fall 2014. We will first give an overview of the framework and discuss its components one by one. Then, we will describe the lifecycle of a MapReduce job and demonstrate the communication protocol between MapReduce components. Also, we will outline how to build and test the current framework from source, and include a Developer's Guide for developing MapReduce Applications based on the framework. Eventually, we will survey some additional unimplemented features that would be desirable for a commercial package.

1 Overview of MapReduce Framework

MapReduce is a highly scalable and parallel computing model that allows application programmers to specify *Map* and *Reduce* methods on a distributed data set in order to complete computational tasks. According to our design, nodes on the framework have one of the three following roles: *Master*, *Worker*, and the *Users*.

Essentially, *Master* is the coordinating node that does not store any data or perform any computational tasks but merely work on administrative tasks like maintaining a global Job Queue and a DFS Lookup Table. On the other hand, a *Worker* receives and performs tasks pushed from *Master*, and also holds partitions of data on the Distributed File System. Last but not least, a *User* node is a designated client of the MapReduce framework, so where an end-user is able to push data on to the DFS and execute a MapReduce job based on the data.

Next section contains more detailed specifications of the three components.

2 Framework Components

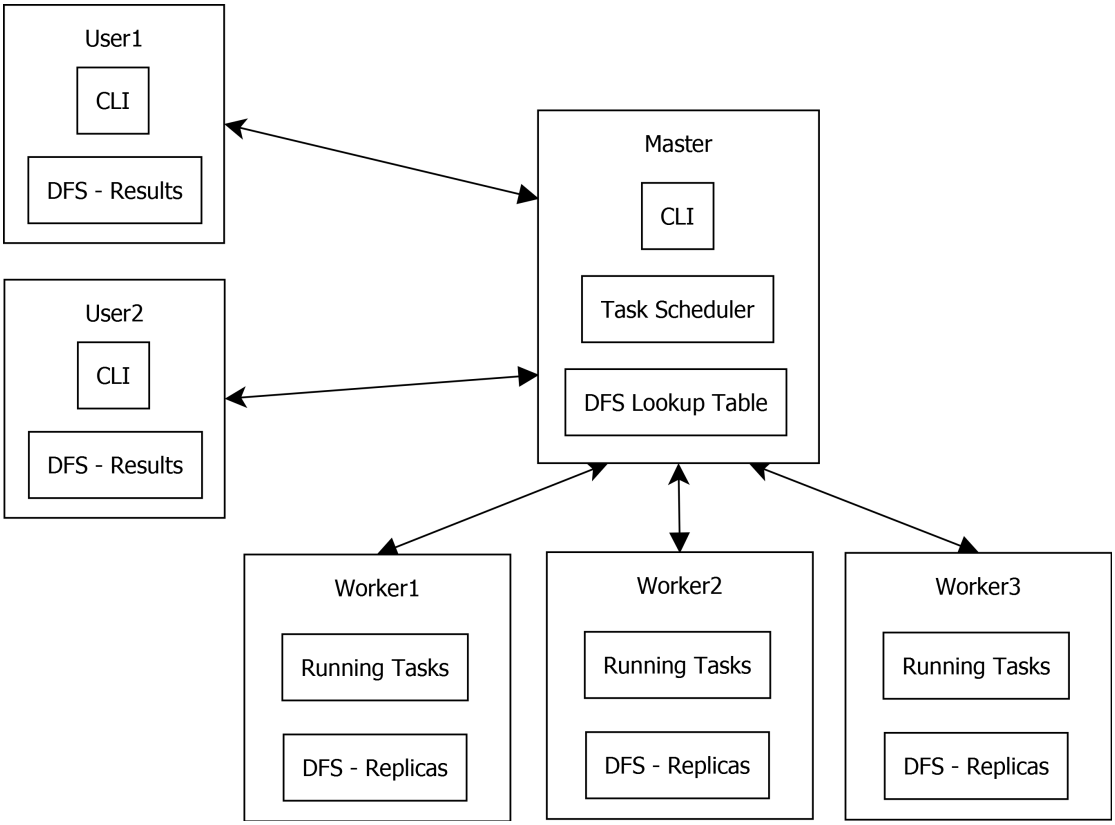
First of all, we will show how the components of MapReduce framework relate to each other in the diagram as follows. In the digram on the next page, there is one Master node, three Worker nodes, and two User nodes.

2.1 MapReduce/DFS Master

The Master node, as the name suggests, is the coordinator of the frameworks. Throughout the lifecycle of a MapReduce job, it talks to both workers and users of the framework, and its main purposes include:

- Keep track of all jobs and tasks on the framework;
- Schedule pending tasks to available slots on each;

Figure 1: Sample MapReduce Framework Diagram



- Maintain look-up table of files on DFS;
- Send heartbeat request to all workers periodically; and
- Accept client (user) requests to add file or execute MapReduce job.

Meanwhile, to make the life of the system administrator a lot easier, we have included an administrative shell in our Master node, which supports the following commands:

- `jobs`: Displays the list of running jobs;
- `tasks`: Displays the list of running tasks;
- `files`: Shows the list of files on the DFS;
- `workers`: Displays the list of workers in the facility;
- `users`: Shows the list of current users of facility;
- `shutdown`: Gracefully shuts down the MapReduce facility.

Nonetheless, it is important to notice that our Master is a *single point of failure* in our framework, because once it fails, all information related to Jobs, Tasks, and Distributed Files are lost, and recovery is impossible because we also don't know which partial results should be sent to which user client.

2.2 MapReduce/DFS Workers

The Worker nodes are the nodes in the framework that actually perform the map/reduce tasks and store partitions of files of DFS into their working directories. They are *hidden* from the users, so only the Master node is able to contact with them. Since the nature of the Worker node is more *daemonized* (i.e. background process that processes incoming requests from the Master), there is no command-line interface for the workers, and once we start them we leave them running, until the Master node disconnects from the network.

2.3 MapReduce Users

In our design, MapReduce users are specifically designated nodes that are allowed to interact with the MapReduce cluster by contacting the Master node. Especially, they should be able to push a local file onto the DFS for use as the input file to a MapReduce job. Similar to the Master node, we also included a user shell in each of the user nodes, which supports the following commands:

- `jobs`: Displays the list of running jobs;
- `files`: Shows the list of files on the DFS;
- `upload <filename>`: Uploads a certain local file onto DFS;
- `<JobName> <InputFile> <StartRecord> <EndRecord> <OutputFile> <Args>`: Executes a MapReduce job with job name `JobName`, input file `InputFile`, output file `OutputFile`, input record range `StartRecord` to `EndRecord`, and user-defined extra arguments `Args`;
- `quit`: Gracefully exits the User node.

Traditionally, only the Master node and Worker nodes are viewed as part of the MapReduce cluster infrastructue. In this case, we are also consider the users as a crucial component because the system administrator is able to change the list of users that are allowed to interact with a certain MapReduce infrastructue. Also, when the *reduce* jobs are done, the output files are explicitly pushed to the user node. Therefore, for our framework we consider the users as a *component of* the cluster, instead of as merely a client.

3 Lifecycle of MapReduce Job

Here is what happens when a MapReduce job is executed:

Step One: User Loads File onto DFS

Given that the MapReduce framework is correctly initialized, the user would call `upload` command on user CLI, which sends the byte contents of the local file to the Master. Master node takes the data and partitions it into blocks based on the `PartitionSize` parameter specified in the configuration. For each block, the Master finds a random subset of workers (equal to `ReplicationFactor` in config), and send the partition to each of the subset of workers. If the operations are successful, Master updates its file trackers (look-up table) and informs user that operation has succeeded.

Step Two: User Starts Job from User CLI

When a user want to execute a MapReduce job with some file on DFS as input, the user uses the `<JobName> <InputFile> <StartRecord> <EndRecord> <OutputFile> <Args>` command on user CLI, as specified in *Section 2.3*. When the parameters reach Master node, it splits the user job into *map*, *sort*, and *reduce* tasks and adds the metadata of tasks into the scheduler. The scheduler will keep track of all the task dependencies and push the appropriate tasks to the worker nodes when they are ready to be run.

Step Three: Coordination Between Master and Workers to Complete Job/Tasks

Master and workers must coordinate to make sure that task dependencies are reinforced while keeping track of remaining tasks for a job. When a *map* task is completed on a worker, it notifies the master, so the Master knows that its corresponding *sort* task is ready to be executed anytime on the intermediate file. Similarly, when a *sort* task is completed on a worker, it also notifies the master, but the *reduce* task is only ready to be launched when all *sort* tasks on that specific worker are done. Eventually, when a *reduce* task is done, the final result output file has been produced, so the worker simply sends back the result file to the master, which in turn will relay the file back to the user that created the job. Therefore, the result file will end up in the user's DFS working directory.

4 Failure Handling

Due to the inherent unreliability of a network, it is not unlikely that some request packages fail to go through temporarily. Also, because potentially a large number of machines can be involved in the framework, it is fairly possible that some nodes go down while running a task. Without proper failure handling, that faulty node can be a *single point of failure (SPOF)*, which is what we want to avoid in a distributed system. Therefore, we adopted the following three approaches to handle failures:

- Periodic Heartbeat: The master node sends periodic `ping` requests to all workers. If after a certain amount of time the worker still has not responded, we mark the worker as "faulty", and migrate the tasks already assigned to it to other nodes (see *Task Migration* below);
- Task Retry: When a task fails on a node for some reason, we allow the task to be re-routed to another node for re-try before marking the task as failed. With this measure, we avoid the case when a task and a worker are not compatible – for example, when the underlying file system of that worker node has a bad sector;
- Task Migration: After we have determined that a node is faulty, we re-distribute all tasks running on that node to other available nodes. In this case, we do not need to restart the entire Job only because of one faulty worker node.

5 Special Considerations

Following are special features that the authors think are useful for application programmers and are beyond the request of the project specification:

5.1 Command Line Utility at RMI Servers and Users:

Different from Hadoop's approach, our MapReduce framework comes with an interactive shell for both *user* and *master* nodes. The shells support most functions as currently supported by Hadoop, and the advantage is that once the shell is up, running any command is only up to the communication time within the framework. In Hadoop's approach, however, a JVM needs to be started for each MapReduce/DFS operation, e.g. `hdfs dfs -ls foobar`. We are not certain why Hadoop did not choose this approach, but our alternative approach is more light-weight and gives lower latency.

5.2 Elegant Error Catching:

Even the best application developers in the world cannot guarantee that their first attempt at writing a complex MapReduce job will work. Therefore, when an application developer feeds us buggy code that causes exceptions in mapper, sorter, or reducer, it is important that the worker node and the Master do not abort. For our framework, when a task causes an exception, the corresponding worker thread would catch it, and send `taskFailed` to the Master. Although this handling approach can still be improved (see *Section 8.3*), it does guarantee that the availability of our framework is not affected by bugs in user code.

5.3 MapReduce Elegant Shutdown:

When the Master node in the MapReduce framework shuts down, its Workers also automatically shut down after cleaning up the working directory for DFS. Meanwhile, the *User* (client) nodes are kept on, but with a message `DISCONNECTED` attached to the prompt. Also, if a *User* node fails to contact Master after a period of time (defined by user), its state automatically becomes `DISCONNECTED`, and the user is also able to *re-connect* with the Master. This allows the user nodes to not lose their states due to temporary connection problem to the Master node.

5.4 Worker Failure Checking with Heartbeats:

In our implementation, there is a background heartbeat thread at Master node that keeps polling all workers nodes for their statuses. Should any node fail to respond, we give the Master a certain number of chances to retry the heartbeat (defined in user configuration file). If the worker eventually fails to respond, Master re-schedules all tasks related to the node without failing the entire MapReduce job. This makes our MapReduce framework extra resilient to worker failures.

5.5 Designated User Nodes:

In our implementation, the user nodes are specifically listed in the configuration file, i.e. the system administrator has full control over which nodes are allowed to communicate with Master node and interact with DFS or execute MapReduce jobs. This adds an extra layer of security for the framework.

6 Developer's Guide – Using MapReduce Framework

Since the MapReduce Framework is to be used by application developers, a pre-compiled Javadoc of the project can be found in the `doc/` directory under the project root. Users can simply open `doc/index.html` and access the documentation of the entire code base including the API usage guides. The mechanism of MapReduce Framework has been explained in previous sections.

In order to fit a Java Object into the MapReduce Framework, say, `WordCount`, the application programmer should create three classes: `WordCountMapper` that extends the base `Mapper` class, and similarly `WordCountSorter` and `WordCountReducer`. Then, the developer should put the classes under `mapr/examples` directory, such that MapReduce users are able to execute job based on our definition of *Mapper*, *Sorter*, and *Reducer*.

In general, application developers are strongly recommended to look at the sample applications in the Package `mapr.examples`, and follow the existing code. We have provided the sample code for WordCount and Grep, and other MapReduce applications can be built similarly.

7 Dependencies, Building, and Testing

The `dtzeng.yzong` handin directory will contain two sub-directories, i.e. `reports` and `RMIFramework`. In the former directory you can find this report file, and in the latter directory is the clean source code for the project.

7.1 Dependencies

This project does not have any external dependencies, and can be compiled with standard Java libraries. To build the project from scratch without using IDE, see the following section.

7.2 Building Instructions

To build the project from scratch, use the `build.sh` script provided in the handin directory at `/afs/andrew/course/15/440-f14/handin/lab3/dtzeng-yzong.x`.

```
gkesden@ghc11 dtzeng-yzong.x$ chmod +x build.sh
gkesden@ghc11 dtzeng-yzong.x$ ./build.sh
```

This will create a jar file called `MapReduce.jar`. You may copy this jar to any desired directory for testing below.

7.3 Testing Instructions

The following test routine is designed to survey most functionalities of the MapReduce framework. Feel free to experiment with different commands in MapReduce Master and MapReduce User command-line interfaces. In our example, our facility has 1 master and 3 workers, with 2 users connected. First, change the hostnames in `mapr.properties` to reflect the desired Gates machines for testing. Let `GHC_M` be the host of the master process, `GHC_W1/2/3` the workers, and `GHC_U1/2` the users.

To start up the facility, do the following:

- Start up master

```
gkesden@GHC_M dtzeng-yzong.x$ java -cp MapReduce.jar mapr.master.MasterCoordinator mapr.properties
```

- On each of the workers

```
gkesden@GHC_W? dtzeng-yzong.x$ java -cp MapReduce.jar mapr.worker.WorkerCoordinator mapr.properties worker?
```

  - Now that the facility has started up, the command line on the master should appear. Make sure to connect at least 1 worker to the master before the specified timeout, otherwise the facility will shutdown automatically. With the facility started up, you can now start the 2 users.

- On each of the users

```
gkesden@GHC_U? dtzeng-yzong.x$ java -cp MapReduce.jar mapr.user.UserCoordinator mapr.properties user?
```

So, now we have everything up and running. On the master, you can use the `workers` and `users` commands to see the status of the workers and users. Now, let's upload our example files. You can do this on any user. This demonstrates on `user1`.

```
user1@9876 > upload large_count.log
File replicated successfully.
user1@9876 > upload large_grep.log
File replicated successfully.
```

Now, when you run the `files` command on either user or master, you can see the information of the files on the DFS, and where the replications are located. Let's now start a new WordCount and Grep job.

```
user1@9876 > count large_count.log 0 99999 ctr_out
Job started with ID 0
user1@9876 > grep large_grep.log 0 99999 grep_out tzeng
Job started with ID 1
```

You can now use the `jobs` command on either user or master, to see the status of the job. To see the individual tasks, use `tasks` on the master machine only. When everything is finished, you can see the results of each job in the user's specified DFS directory, or in our case `worker1-dfs-root`.

Now let's see what happens when a worker fails. Start another job, and kill a worker.

```
user1@9876 > count large_count.log 0 99999 ctr_out
Job started with ID 2
        *** Kill a worker, in this example worker 1 ***
gkesden@GHC_W1 dtzeng-yzong.x$ java -cp MapReduce.jar mapr.worker.WorkerCoordinator mapr.properties worker1
Successful connection to facility master established.
^C gkesden@GHC_W1 dtzeng-yzong.x$
```

When you run the `jobs` command again, you will see that the status for job 2, becomes `RESTARTED AS JOB 3`. Job 3 will now finish what job 2 requested. If you try the "files" command, you can see that the replicas on the killed worker (worker1) are removed.

Lastly, to shut down the facility, use the `shutdown` command on the master CLI. The users are not included in the facility, so another interesting feature is the "reconnect" command on the user. If the facility is started up again (masters and workers), the user can simply try to reconnect, and it will automatically be added back into the master's state.

8 Further Work & Enhancements

Due to time constraints, although the final product is fully functional, it still lacks many features that are desired in commercial packages (like *Hadoop*), and their implementation difficulties also vary. This section will mention a selected few and discuss the difficulty for implementing them.

8.1 MapReduce Job Priority

Currently, the MapReduce jobs in our framework are executed in a first-come-first-served basis. However, it would be nice if we could support priority-based job/task scheduling, because on a production cluster we might have a MapReduce job that takes 10 hours to complete, yet in the mean time we might want to add a job that takes 1 hour to complete. In the current framework, we will not get our result for the shorter job untill after 11 hours. However, if job priority is supported, then we can set the long-running job with priority value `NORMAL`, and let the short job have priority `HIGH`. In this case, the short job will be able to run even before the long job finishes, so we can get our results faster.

8.2 Pipelining MapReduce Phases

In reality, sometimes application developers might want to chain many MapReduce jobs together, i.e. feed the output of a MapReduce job directly to another MapReduce job. However, our current framework only supports sending a single MapReduce job at a time based on existing input, so in this case we need someone to wait for the reduce phase of the first MapReduce to complete before assigning the second MapReduce phase. However, this manual labor can be avoided if we can enhance our Application Developer API such that it supports pipelining many MapReduce jobs together.

8.3 Global Logging

Despite the failure recovery techniques we implemented as in *Section 4*, when an exception occurs the MapReduce job will more likely survive, but it does *not* help the Application Developer to debug the application because there is no persistent logging of our framework. A helpful enhancement would be for each node in the cluster to periodically log its past activities, exceptions, and then store the logs on DFS so that the application developer (User node) is able to examine the file and debug the application.