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

2.1 MapReduce/DFS Master

Meanwhile, 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 lose the routing rule of partial results to the requesting client.

2.2 MapReduce/DFS Workers

They carry out all the work.

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.

3 Lifecycle of MapReduce Job

Here is what happens when a MapReduce job is executed:

Step One: User Load Data onto DFS

Step Two: User Execute Task from User CLI

Step Three: Mappers Feed Results to Sorters

Step Four: Sorters Feed Results to Reducers

Step Five: Reducers Return Results Back to Client

4 Communication Protocols

As seen in the previous section, the key of the MapReduce framework is its management of concurrent requests, especially at the Master node. The following two sub-section will survey the network interactions between MapReduce Master node and two other components.

4.1 Communication Between MapReduce User & MapReduce Master

By discussion in *Section One*, (What master does). Following are the communication details of each method:

4.2 Communication Protocol: MapReduce Master & MapReduce Workers

4.3 Failure Handling

Due to the inherent irreliability 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 goes 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 addopted 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 work don't work well – 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. This this case, we do not need to re-start the entire Job only becaues of one faulty worker node.

5 Special Considerations

Following are speical 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 *Uesr* (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 RMI 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. Also, to build the project from scratch without using IDE, blah.....

7.2 Building Instructions

To build the project from scratch, follow the steps below:

```
gkesden@ghc11 yzong$ cd RMIFramework
gkesden@ghc11 yzong/RMIFramework$ ls
    build.xml  doc  lib  src
gkesden@ghc11 yzong/RMIFramework$ ant clean build jars
    (Output omitted)
gkesden@ghc11 yzong/RMIFramework$ ls
    bin         lib                       RMIServer.jar          src
    build.xml   RMICalculatorClient.jar   RMITestRegistry.jar
    doc         RMIRegistry.jar           RMIZipCodeClient.jar
```

To clean the project directory, run the following command:

```
gkesden@ghc11 yzong/RMIFramework$ ant clean
    (Output omitted)
gkesden@ghc11 yzong/RMIFramework$ ls
    build.xml  doc  lib  src
```

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. The machines and ports in use in the sample are `ghc12:6060` (RMI Registry), `ghc15:41052` (RMI Server), and `ghc16` (client-side application). Other combinations will also work, but be sure to substitute with the correct parameters below.

```
# Starts up RMI Registry
gkesden@ghc12 yzong/RMIFramework$ java -jar RMIRegistry.jar -p 6060
       *** Stdout tails RMI Registry activity log ***

# Sanity check for RMI Registry (-ea flag is necessary!)
gkesden@ghc15 yzong/RMIFramework$ java -jar -ea RMITestRegistry.jar
Connecting to RMI Registry Server...
Registry Hostname: ghc12.ghc.andrew.cmu.edu
Registry Port Number: 6060
Testing PING...
Testing BIND...
Testing LOOKUP
Testing REBIND
Testing LIST
Testing UNBIND
All tests passed!

# Start an RMI Server
gkesden@ghc15 yzong/RMIFramework$ java -jar RMIServer.jar -h ghc12.ghc.andrew.cmu.edu -p 6060
```

```
INFO -- RMI Server started at ghc15.ghc.andrew.cmu.edu:41052.
        Master RMI Registry at ghc12.ghc.andrew.cmu.edu:6060.
INFO -- Connection to RMI Server established. CLI started.
RMI Server @ 41052 > bind service1 com.yzong.dsf14.RMIFramework.examples.CalculatorServerImpl
Successfully registered service service1!

RMI Server @ 41052 > bind service9 com.yzong.dsf14.RMIFramework.examples.ZipCodeServerImpl
Successfully registered service service9!

RMI Server @ 41052 > list
Following are the entries of Remote Object Reference table on local RMI Server:
Object Key: 0
Remote Interface Name: com.yzong.dsf14.RMIFramework.examples.CalculatorServer
Object Key: 1
Remote Interface Name: com.yzong.dsf14.RMIFramework.examples.ZipCodeServer

# Test the RMI ZipCode Client
gkesden@ghc16 yzong/RMIFramework$ java -jar RMIZipCodeClient.jar
Registry Hostname: ghc12.ghc.andrew.cmu.edu
Registry Port Number: 6060
Registry Service Name for ZipCodeServer: service9
Data File Path: src/com/yzong/dsf14/RMIFramework/examples/ZipCodeData.txt
        *** Expected test output ***

# Test the RMI Calculator Client
gkesden@ghc16 yzong/RMIFramework$ java -jar RMICalculatorClient.jar
Registry Hostname: ghc12.ghc.andrew.cmu.edu
Registry Port Number: 6060
Registry Service Name for CalculatorServer: service1
Setting the name of Calculator Object as: CalcFooBar
        *** Expected test output ***
Registry Service Name for ZipCodeServer: service9
Data File Path: src/com/yzong/dsf14/RMIFramework/examples/ZipCodeData.txt
Initializing the ZipCodeSever...
        *** Expected test output ***

# Exit RMI Server; Clean-up Services on RMI Registry
RMI Server @ 41052 > exit
Unbound service service1 from RMI Registry.
Unbound service service9 from RMI Registry.
Done. Goodbye!
```

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 4.3*, 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.