15-440 Project 3: Building a Map-Reduce Facility
Spring 2013
Aditya Joshi (adityajo), Vishalsai Daswani (vhd)

Project Write-up

**System Requirements:**
Unix system
Java 1.6SE or higher

**Configuration steps:**
Edit the config.txt file that specifies all the masters and slaves in the system.

The file is in format:
Ipaddress \t Role \t Portnum\n
for each line, where Ipaddress is the ip of that machine, the role is either "MASTER" or "SLAVE", indicating whether it is a master or slave in this system, and Portnum is to refer to the port number that you would like to connect to for this system. You can only have one master in the system, but any number of slaves. Just make sure that no two systems are running on the same ipaddress and portnum.

Now to run the system, start on the master and run the hadoop.Start class file. This will run a master instance, and if necessary slave instances on the same machine.

After this, go onto every slave, and run hadoop.Start class file. This will instantiate the slaves, and establish a connection with the master.

Finally, once the system is up and running, you can send a user job request the map-reduce system. To do this, modify the /src/RequestMapReduce.java file, and specify all the necessary details including your input source file, your output source file, the type of input you are using (whether it is key-value type, or text type). You must also specify the details of the Map Reduce classes, and their respective locations.

Once these configurations are set up for the job you wish to execute, simply run this. It will print out the name of the job you have been assigned, and will execute the job on the system.

There are data structure present on the master and scheduler that keep track of all the jobs, but there is no user interface for the user to be able to actually detect what is running and what is not.

**Documentation for Application Programmer:**

The overall structure of the map-reduce library is basically a master that starts running, and every slave that wants to join the system can simply start it up, and dynamically connect to the master. That way, this can constantly be modified and updated if more slaves are ever needed.

The fileIO library was created to handle several different things involving reading records in a particular format, reading information from the configuration file as desired using the ConfigReader. We have implemented a RecordReader, which is designed to aid reading recordings in any input file. The DirectoryHandler was very important in compressing a lot of the mapped files into simpler key files that could be used when reducing. It would collect files of the same key in the different directories into a single file, which was then processed accordingly in the reduce part.

The hadoop library primarily contains all the functionality for the masters and slaves connections, and actually the framework to call the map and reduce classes.

MessageProtocol package was simply developed to create standardized messages that will happen between different systems, and would appropriately convey the necessary information to handle a specific message.

**Two examples:**

```
ArrayList<String> allFiles = DirectoryHandler.getAllFiles(filepath, ".txt")
```

This is an example of how you can get all the files that exists in filepath, that has the ".txt" as a file extension.

```
String filePath = ConfigReader.getResultsfiles() + j.getJobName() + "/";
```

This provides the location for where to store the Results files for a particular job.

Our project met the overall functionality of operating various map reduce jobs on a given system. It didn't meet the requirements of recovery from a failure on a given system. As of now, it assumes that each job request continues without a problem. Furthermore, the reduce functionality was applied to a single machine, rather than being distributed between multiple machine, as the map job was performed.

With more time, we would definitely have performed a much greater code clean up, as well as distribute the reduce job onto different systems in a similar way to how we performed the map job we already did. With the standardized message protocol, it wouldn't be a problem to simply modify this and allow a more distributed reducing stage.

We would also make sure that there were more stages the clearly indicated the state of the Job and the Slave. At the moment, it is not as specific as it could potentially be to monitor the system. It still functions as need be, but this code be helpful for extending the functionality of the project, as well as better monitoring of the state of the system.

Another potential thing to do would be to have a better method of sharing the job and slave information besides simply passing in references to each of the individual threads. At the moment, it's a little bit messy, and perhaps creating a singleton class with all the data store information would be more appropriate.

In terms of recovery, we planned to catch io exceptions, which indicated whether a socket connection broke, and notify the scheduler to basically move all the partitions for that slave to other slaves. This is something we would definitely implement given more time. In order to do this, we would need to send to the scheduler what slavewrapper it was that failed, and that way, the scheduler would remove that slavewrapper out of the slaves hashmap, and reallocate everything it had to other slaves. We would also handle this similarly if the MapResult returned a false for whether the map job was successful.

Lastly, when slaves receive Map requests, we thought of spawning a new thread for each one it received. That way, it wouldn't block waiting for one map job to be performed before the rest. That would allow for more concurrency when performing multiple map reduces for multiple jobs.

There are several things we would implement with additional time, but it performs the basic functionality of operating a map reduce job, on a system, as shown by the example that we created.