Map Reduce Framework Report

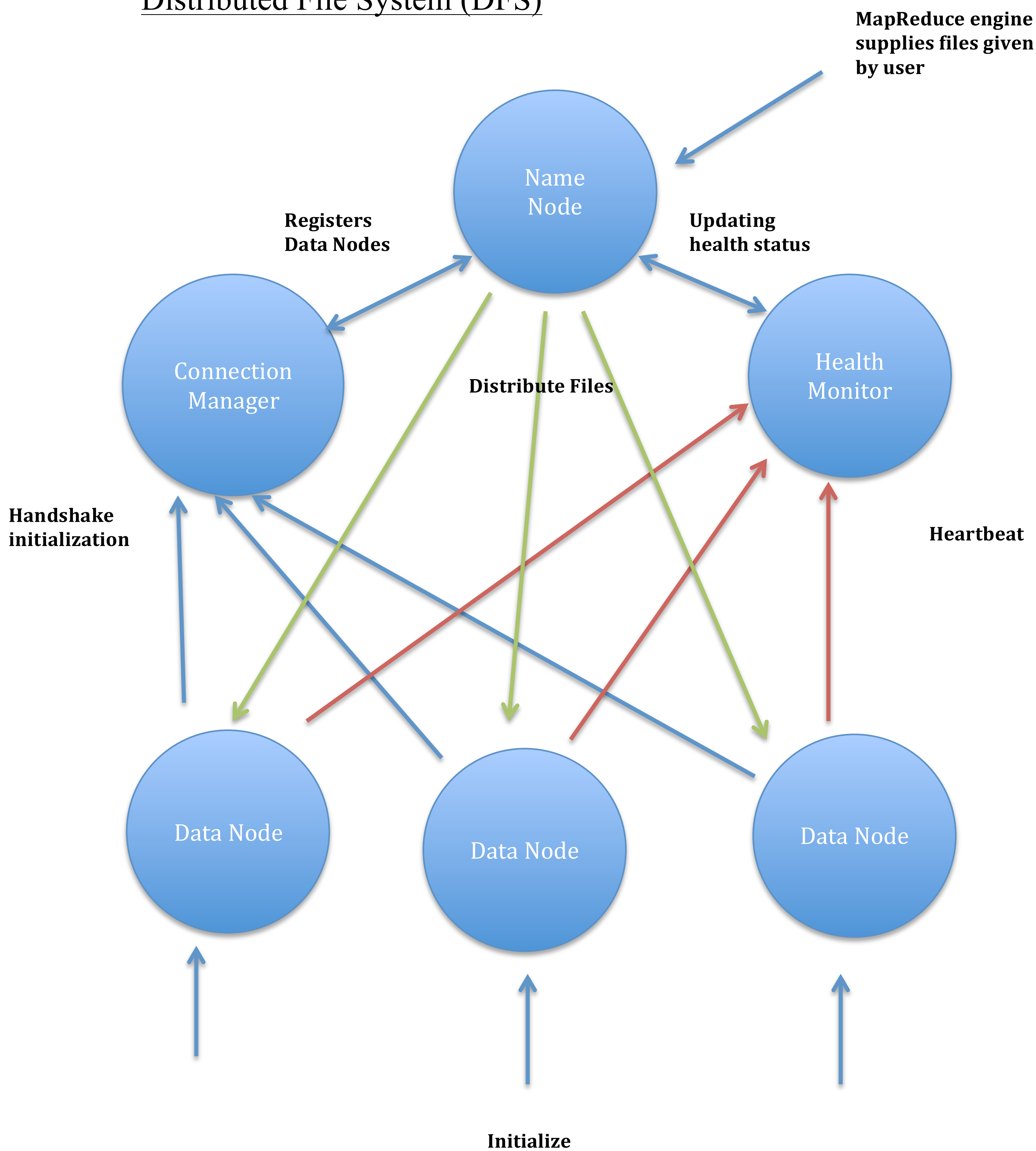
Arjun Puri & Karan Sharma

23rd November 2014

Contents

Overview

Distributed File System (DFS)



The DFS is the backbone of the MapReduce Framework. It provides a distributed data storage facility for all participants of the framework. The DFS consists of a few central components: The DFSNameNode, DFSDataNode, DFSConnectionManager, and the DFSHealthMonitor.

The DFS is initialized with the NameNode being created upon the creation of a MapReduceMaster host. This consequently instantiates a DFSConnectionManager and DFSHealthMonitor in separate threads to aid the facility of the DFSNameNode. Their functionality is explained further on. The DataNodes are created on separate machines (NOTE: must be separate from the NameNode). These DataNodes are created upon instantiation of the MapReduce Slaves.

Here is a better description of the respective parts of the DFS:

DFSNameNode

The DFSNameNode must be assigned before the assignment of any MapReduce jobs. The purpose of the DFSNameNode is to facilitate the partitioning and transfer of files to the DFSDataNodes that it receives from the MapReducer client. It also manages the status of all participants with the help of the DFSHealthMonitor. It provides RMI Services for other DFSDataNodes to query for information about where files are located across the system. In the case of a DFSDataNode failure, the DFSNameNode will ensure in maintaining the replication factor by transferring those files to another DFSDataNode, while also balancing the load appropriately. This is done with aid of FIFO queue.

DFSDataNode

The DFSDataNode facilitates the storage of file block replicas. To clarify- on the DFSNameNode, any given file is split up into chunks and then replicated across the file system. The DFSDataNode maintains the availability of space and also provides it's own RMI services for the name node to transfer file blocks. The data node is also responsible for sending a heartbeat at a fixed interval defined in an internal configuration class. This heartbeat is sent to the DFSHealthMonitor.

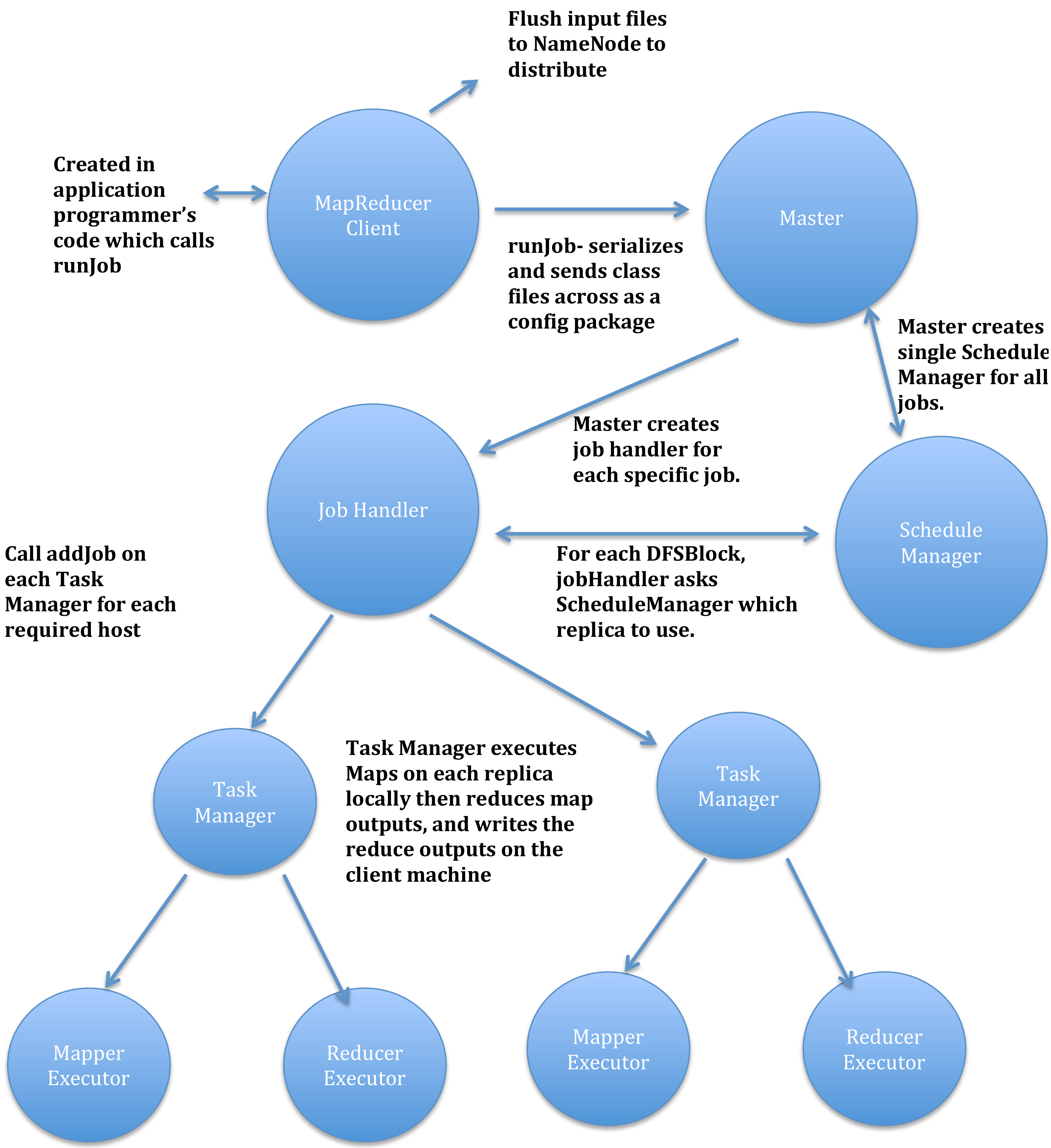
DFSConnectionManager

The DFSConnectionManager runs in a separate thread from the DFSNameNode. It exists to handle all incoming connection requests from any new DFSDataNode. On receiving a handshake from a data node, the DFSConnectionManager will register this DFSDataNode with the DFSNameNode, and the DFSHealthMonitor. One this is done, the DFSDataNode is officially part of the active system.

DFSHealthMonitor

The DFSHealthMonitor is responsible for ensuring that all participating DFSDataNodes are fully responsive. The DFSHealthMonitor stores an initial 100 health for all DFSDataNodes that are initiated into the system. When a data node skips a heartbeat, it loses a certain amount of health until it's health reaches 0 at which point it is deemed inactive/dead. The reason this is done is to allow a data node to go offline for a momentary period then join back with losing all it's files to other data nodes.

MapReduce



Flush input files to NameNode to distribute
Created in application programmer's code which calls runJob
MapReducer Client
Master
runJob- serializes and sends class files across as a config package
Master creates single Schedule Manager for all jobs.
Master creates job handler for each specific job.
Job Handler
Schedule Manager
Call addJob on each Task Manager for each required host
For each DFSBlock, jobHandler asks ScheduleManager which replica to use.
Task Manager
Task Manager executes Maps on each replica locally then reduces map outputs, and writes the reduce outputs on the client machine
Task Manager
Mapper Executor
Reducer Executor
Mapper Executor
Reducer Executor

The MapReduce framework provides a platform for application programmers to perform MapReduce jobs in an efficient and succinct manner. The first point of interaction is in the application programmer's code, which creates a MapReduceClient object to run a job. The programmer provides the input files, a Map class and a Reduce class to the framework in the form of a configuration abstraction called the MapReduceConfig. It must be noted that the DFS and the MapReducerMaster and Slaves must be running prior to running the programmer's code. The MapReduceClient will send all the require information to the MapReducerMaster which will start, schedule and monitor the job across the slaves.

It is extremely important to note that a MapReduceClient MUST run on a machine that is already hosting a DFSDataNode. This allows the framework to maximize efficiency through a certain amount locality.

Below is a more detailed description of the components of the MapReduce Framework.

MapReducerClient

The MapReducerClient is where the application programmer can interact with the framework. In the code that the programmer writes, he must create an instance of this, and then must call the runJob function. The MapReducerClient will take in a configuration that the programmer will write that contains the class files, the input files etc. that the programmer has customized. The MapReducerClient will then locate the input files and will flush it to the DFS by handing byte arrays to the DFSNameNode for partitioning and distribution. The MapReducerClient will also forward the

Mapper and Reducer Class as byte arrays to the Master. The MapReducerClient must handshake with the Master before this is possible.

Master

The Master on the MapReduce framework provides the central RMI facility to communicate the DFSNameNode and all other it's respective components such as the ScheduleManager etc. The Master on construction will start a DFSNameNode and a ScheduleManager. The MapReducerClient will handshake with the Master, which puts the client in a position to call the createJob method. This will construct the necessary components for a job-jobID, add the class byte arrays to the Master, and a JobHandler for that job. Starting the job will forward execution to the JobHandler. The Master also has some interactive capabilities in the terminal.

TaskManager

The TaskManager is local to the particular slave that it is working on. It runs as a Thread while working with the local files on the corresponding DFSDataNode. The TaskManager establishes a Thread pool based on the number of cores the current machine, which is determined at run time. The TaskManager is added to the registry of the DataNode. On adding a job to the TaskManager the task manager will get the required Mapper and Reducer class from the Master using RMI, and will then instantiate an instance of it. Each map operation is submitted to the thread pool, which will run the task when a thread opens up. At the end of each map task, the thread will check with the TaskManager to see if it was the last map task for the given job. If it was, then a reduce is performed which the last map idles, which allows us to maintain the number of active threads on the machine at all times in order to maximize parallelism. Also, if any of the map tasks fail, they report the failure to the TaskManager, which resubmits the tasks and logs the a failure occurred. Once the failure threshold is reached, the TaskManager notifies the Master (which tells the JobHandler) that the job has failed on that host and all remaining map tasks in the thread pool will exist as soon as they are run. If the local reduce operation succeeds, then the output is copied to a byte array and is sent to the JobHandler via the Master. The last responsibility of the TaskManager is to write all of the reduce output files to a local directory if it is on the same host as the client.

JobHandler

The JobHandler runs on the Master host and is created for each job created by the MapReducerClient. When it is run, the Master provides it with the unique file identifiers (the fileIDs given by the DFS upon upload), and it then becomes fully responsible for the job and must setup and run all tasks the job requires. In order to do this, it first gets the DFSBlocks for each of the input files using the NameNode's RMI and then asks the ScheduleManager which replica to use for each block. It then partitions the given replicas according to which host they reside on and adds the job to the TaskManager running tasks on each of these hosts. The JobHandler is eventually notified by the TaskManager via the Master of either the failure or success of the job on that host. Once all successes are reported, the JobHandler forwards the byte arrays of each local reduce output to the TaskManager of the initial client, which then writes the output locally on that machine. At any given point during execution, the JobHandler can be inquired about the status of the job, at which point it checks the status of each of the TaskManagers and returns a string with the full state description.

ScheduleManager

The ScheduleManager runs on the Master host and is in charge of selecting a replica of a block for a given map operation. The scheduling is based on locality is availability, as only hosts with the replica locally stored are considered and the host with the smallest task load is chosen and returned to the JobHandler. The load is acquired by called the load function on each relevant TaskManager.

MRCollector

The MRCollector is a wrapper around a ArrayList, which is provided to the client for simplifying the provided Mapper and Reducer classes. Essentially the provided Mappers and Reducers simply need to add whatever output is necessary to the MRCollector and the collector does the rest. The MRCollector creates a KeyValuePair of the given output and inserts the KeyValuePair to the private ArrayList. The KeyValuePair is constructed such that it implements Comparable and maintains order by the ordering of the KeyType (assumed to be Comparable itself). Thus when the Executers request the Arraylist after going through the revalent file, the returned ArraryList is easily sortable.

MapExecutor

The MapExecuter, submitted as a thread to a threadpool on a TaskManager, is in charge of executing a map operation on a local replica of a given block.

It first checks to make sure the job is still valid (yet to fail) and then reads the inputFile line by line, at each step calling the client provided Mapper function and collecting the output using a MRCollector. Then the MRCollector returns the hidden ArrayList, which is sorted and written to an intermediate file. The path of this intermediate file is provided to the TaskManager by way of a call to checkReduce(), which executes a reduce if it the appropriate time to do so. Otherwise, it updates the job state on the TaskManager and the thread ends.

ReduceExecutor

The ReduceExecuter is launched upon competition of the final MapExecuter for a given job on a TaskManager. It is launched within the synchronized function checkReduce on the TaskManager, so the number of cores running is conserved and the reduce task is thread safe. The RedueceExecture takes in each of the intermediate files and adds each KeyValuePair to a TreeMap containing the entire map output for the job in a sorted structure. Then each KeyValuePair in this TreeMap is given to the client provided Reducer and the output of this Reduces is collected within the MRCollector. After this, the ArrayList of MRCollector is acquired (already sorted) and written to a local reduce output file. Lastly, this filename is sent to the TaskManager, which will forward it to the JobHandler. It is important to note that if a reduce operation has a failure, it is not tolerated and the job will not be resubmitted. It is up to the client whether to resubmit the entire job to the master or accept the partial output of the Map Reduce framework.

Limitations

- The framework assumes that no other instance of this MapReducer Framework is running on the same host. Only one framework can run on a host which will ensures that when multiple jobs are being executed on an instance of the framework they don't confuse other instances of the framework.
- The Master is assumed to run perfectly fine all the time. No health checking is done for the Master or the NameNode.
- The number of block replicas of the files cannot be greater than the number of DFSDataNodes in the system. This will cause certain file replicas to be on the same Data Node which is redundant
- The MapReducerClient must be run on the same host as some existing DataNode. This is extremely important as it allows the framework to maximize efficiency through locality.
- The number of file replicas and their corresponding total size must be less than the maximum space capacity of the DataNode in order to make room for additional intermediate files to be created.
- The user has to provide compiled class files to the framework.

System Requirements

- The system must run on a Linux or Unix System only.
- The machine must have an appropriate amount of memory left according to user's job requirements.
- All classes must be compiled before running the code.
- The disk must have enough space for the file blocks to be uploaded.

Improvements

Distributed File System (DFS)

- NameNode failure recovery with the use of checkpoints and log files to recover file maps, locations and other data that the NameNode stores.
- File flush to data node in parallel. Currently our framework has to loop through all the files that the user provides, and with the help of a queue flushes the file replicas to the DataNode. This could be improved by doing this job in parallel to improve the efficiency of the system.
- Better Configuration set up for application programmers. In order to configure the settings of the framework, currently the programmer has to navigate into the src/Config folder of the source code and then change the fields of the .java file to suit his/her needs. This wasn't changed in the interest of time and for future versions our framework should work with .conf files that allow better customization.

MapReduce

- JAR file capabilities for the application programmer. With the current design, the application programmer has to compile his code and provide Class files to the framework. This can be changed to allow the programmer to create a separate JAR file to the framework with all code packaged.
- Provide recovery failure to the MapReduce Master node through the use of checkpoints. The Master may fail at any point due to network inconsistencies. At this point our framework does not support the recovery of the Master. With the use of checkpoints and log files, the Master node could potentially be recovered.
- Send .Class files as chunks to allow for consistency checking. Right now the framework simply sends the entire class as a byte[]. This doesn't account for the fact that there might be loss in data. Sending the .Class file in chunks would allow for the framework to check that the .Class file is still in tact.
- Provide a greater variety of job analytics to the system.

Framework Features

Distributed File System (DFS)

- **File Upload-** Files get uploaded to DFS on creating a MapReduceClient to NameNode. This gets stored in a buffer to allow the user to provide multiple files. The split is done consistently but assumes the files have a constant line length.
- **DataNode Handshake**- DataNode handshakes with NameNode on creation to register DataNode with system and add NodeId to maps.
- **Health Monitoring with heartbeat-** DataNodes send heartbeats to DFSHealthMonitor to maintain status and to ensure system is functional
- **Recovery Failure**- If DataNode has not been gone for two long (health > 0), on coming back all the files will be restored to the DataNode. If the health goes below 0, the DataNode is deemed as dead.
- **Replication Factor Maintenance**- If a DataNode dies, the blocks that were on it's system are moved to other DataNodes to ensure replication factor is maintained.

MapReduce

- **Thread Pool –** Utilized on each TaskManager. Jobs can be asynchronously be submitted to thread pool and it will ensure only a fixed number (equal to the number of cores) of task are executed concurrently
- **Load Balancing** – The ScheduleManager always selects the replica host with the smallest load, thereby balancing the load of the system.
- **Locality** – The ScheduleManager only selects hosts that have a replica of the given block stored locally. Thus all tasks take advantage of locality and avoid remote reads.
- **MapReduce Job failure handling** – Node failure is already handled by the DFS so the handling focuses on individual tasks. Tasks are resubmitted on the same host until the failure threshold is reached at

which point the job on that host is considered failed. It is up to the client to resubmit the job or use the partial result given at the end.

- **Return of reduce output to client** – When all nodes return reduce output byte arrays to the JobHandler, the jobHandler forwards them all to the TaskManager on the same host as the client, which writes all the output to a local directory specified by the client.
- **Interactive environment for Master and Client** – At any point the Master and Client can be queried for information about the state of a job, node, or system.
- **Allows multiple MapReduce jobs on the framework** – Multiple jobs can run concurrently, because each is organized by a unique JobHandler and the other components of the system group based on jobID (which the exception of the DFS which is agnostic).
- **Allows clients to provide multiple files for input** – Multiple files can be provided for a given job and can be uploaded to the DFS without multiple calls.

Building and Deploying

Configuration

Before the framework can be run, certain configurations must be made to the internal configuration settings of the framework. This can be done by simply navigating into the src/Config/ of the MapReduceFramework. There is a file named ConfigSettings.java files that can be modified to tailor the framework to a programmer’s needs. The following is a description of the fields present:

```
public static int replication_factor = 1;
public static int split_size= 2;
public static int heartbeat_frequency= 3;
```

The *replication_factor* is simply how many different block replicas the programmer wants to distribute over the DFS.
The *split_size* is the number of lines each block replica should contain
The *heartbeat_frequency* is how frequent the DFSDataNode should send heartbeats, and how frequently the DFSHealthMonitor should check.

Compile

There is one shell script located in the /MapReduceFramework folder called *bundle.sh*. The following describes how this is used:
Run the command *bash bundle.sh* to compile all the code.

```
[arjunpur@unix3 MapReduceFramework]$ bash bundle.sh
```

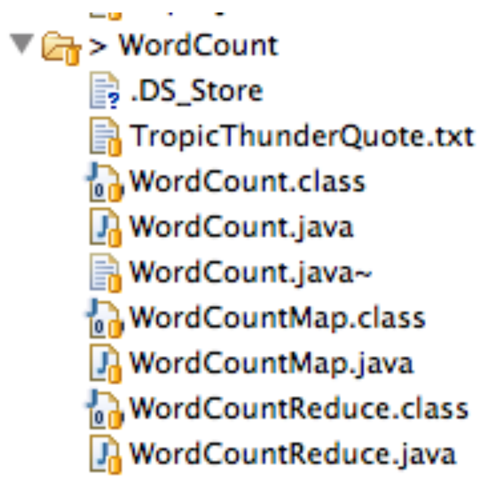
The use of this shell script is completely optional since the *run.sh* script already runs the compile script. *bundle.sh* should only be used if the user only wants to compile the code

Introduction to Environment & Things to Keep in Mind

To run a MapReduce job cleanly, the following few points must be kept in mind:

- The Master server must be run first. This creates a NameNode and starts off the basic center of the DFS.
- The DataNodes must then be created.
- Once these are done, the MapReduceJob can be run.
- The application programmer must create a package folder for the job he/she wants to run. This package folder must be placed in the src/ of the framework. The package folder has the following requirements:
    - It must contain a .Class file that has a *public static void main(String[] args);* function. The details of this .Class file will be explained later on.
    - It must contain a .Class file that implements the framework's *Mapper* interface.
    - It must contain a .Class file that implements the framework's *Reducer* interface.
    - It must contain all the required input files that the programmer wishes to provide to the framework.

The following is an example of what a sample application programmer's directory should look like:



Running the Master Server

Any execution that needs to be done with this framework comes only from the *run.sh* shell script. The following command deploys the Master Server:

```
bash run.sh –m <port_number>
```

Where *<port_number>* is the port number that the user wants to deploy the master on. For example:

```
[arjunpur@unix3 MapReduceFramework]$ bash run.sh -m 8080
NameNode being initiated.
Starting NameNode...
Connection Manager started...
Starting Master...
Master ->
```

Once the Master Server has been deployed, an interactive environment will launch indicated by the *"Master -> "/"*. The environment supports the following commands:

- `data_nodes?` – Displays all active/inactive data nodes
- `file_blocks?` – Displays all file blocks allocated to data nodes
- `datanode_health?` –Displays the health of all data nodes
- `host?` –Displays the host of Master Server

The following is a sample of how the interactive environment runs:

```
Master -> data_nodes?
DataNodes Present:
----------------
ACTIVE: Node2 unix5.andrew.cmu.edu
ACTIVE: Node1 unix4.andrew.cmu.edu
----------------
Master -> file_blocks?
File Blocks Present:
----------------
TropicThunderQuote.txt : 1
TropicThunderQuote.txt : 2
TropicThunderQuote.txt : 0
----------------
Master -> datanode_health?
----------------
Node2: 220
Node1: 220
----------------
Master -> host?
----------------
unix3.andrew.cmu.edu
----------------
Master -> 
```

NOTE: DataNodes normally have a health of 100. Here the 220 is only for testing purposes.

Running the TaskManager & DataNode

The TaskManager acts as a slave machine to the MapReduce Framework. It works in unison with a DataNode that it is linked to. Note that this is solely to maximize the efficiency of the framework with a certain amount of locality. Other DataNodes are also used by the MapReduce Job for Load Balancing.

To start a TaskManager and DataNode, the following command is used:

```
bash run.sh –d <node_id> <port> <master_host>
```

Where <*node_id*> must be a **UNIQUE** id for the node being launched. Uniqueness is up to the user to ensure, and can be guaranteed by checking the current nodes running on the Master Server with the Interactive Environment provided. <*port*> is the port of the Master Server, and <*master_host*> is the host of the Master Server. This is an example:

```
[arjunpur@unix4 MapReduceFramework]$ bash run.sh -d Node1 8080 unix3.andrew.cmu.edu
Creating Data Node...
DFSDataNode ID Node1 is starting...
DFSDataNode ID Node1 binding with NameNode Registry
Heartbeat Started
```

Running the MapReduce Job

Now, the final step to run the MapReduce job. After the programmer has followed the instructions about setting up his job package folder with the required .Class files (an API tutorial is provided later), he can run the job with the following command:

```
bash run.sh <JobPackage/JobClass>
```

Where *<JobPackage/JobClass>* is the package of the job to be executed and the class file with the main function to be executed. An example is as follows:

```
WordcountClient -> ^C[arjunpur@unix4 MapReduceFramework]$ bash run.sh WordCount/WordCount
Starting job: JobID = WordcountClientMapReduce.MapReducerConfig@53628ee
Starting WordcountClient interface...
WordcountClient ->
```

Once this command is run, a JobID is outputted to the terminal, and another interactive environment is initiated. This environment allows the programmer to monitor the state of the job. This is done with the following command in the environment

```
job_details <job_id>
```

An example of this command is the following:

```
WordcountClient -> job_details WordcountClientMapReduce.MapReducerConfig@53628ee
All tasks completed on Node2
All tasks completed on Node1

WordcountClient ->
```

Using the Framework (API) & Testing with Examples

Explaining the API

The WordCount job is a sample application that we wrote to demonstrate how our framework's API works. There are 3 primary .java files that the application programmer would have to write, and this is best demonstrated by looking through the source of the WordCount job.

There is WordCount package folder with 3 .class files in it in addition to the input .txt file:

- WordCount.class
- WordCountMap.class
- WordCountReduce.class

Looking at WordCount.java first we see the following:

```
public class WordCount {

    public static void main(String[] args) throws IOException{
        MapReducerConfig config = new MapReducerConfig();
        config.setMapperClass(WordCountMap.class);
        config.setReducerClass(WordCountReduce.class);
        config.setOutputFilePath(InternalConfig.DFS_STORAGE_PATH);
        File tropicThunder = new File("./src/WordCount/TropicThunderQuote.txt");
        File[] files = new File[1];
        files[0] = tropicThunder;
        MapReducerClient map_reducer = null;
        try {
            map_reducer = new MapReducerClient("WordcountClient",new Host("unix3.andrew.cmu.edu",8080));
        } catch (Exception e1) {
            e1.printStackTrace();
        }
        try {
            map_reducer.runJob(config, files);
            map_reducer.startInterface();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

A *MapReducerConfig* is a layer of abstraction for certain customizable settings that the programmer would have to provide. Namely the Mapper class, the Reducer class and the OutputFileDirectory.

To provide the input files to the framework, the programmer creates a List of the files he wants to operate the map reduce job on, and calls the *runJob(MapReducerConfig config, File[] files)* on it.

Before doing this, the application programmer must create an instance of the MapReducerClient. The programmer must provide a unique name to the client, and must also provide the hostname and port on which the Master Server is running. The programmer may use this instance of the MapReducerClient to run multiple MapReduce jobs.

On executing the *runJob* function, the client will send all the required files to the Master which will do all the scheduling and task assignment. Once this is done, the application programmer may choose to run the *startInterface()* function will start an interactive environment for the application programmer to monitor the progress of the job.

In addition to these two files, the application programmer must also write a Map class that implements a Mapper Interface, and a Reduce class that implements the Reducer Interface. This is seen in the WordCount example as below:

```
public class WordCountMap implements Mapper {

    @Override
    public void map(String line, MRCollector mapperOutputCollector) {
        String[] words = line.split(" ");
        for(String word : words) {
            mapperOutputCollector.addOutput(word, 1);
        }
    }

}
```

```
public class WordCountReduce implements Reducer{

    @Override
    public void reduce(String key, ArrayList<String> value,
            MRCollector reducerCollector) {
        reducerCollector.addOutput(key, value.size());
    }
}
```

In the above code, the MRCollector is a convenient collector for a map or reduce task that sorts the key into its correct place on insertion. This allows the framework to efficiently run through all the map/reduce tasks while collecting the output in a sorted manner.

Running the WordCount job

To run the WordCount job the following command needs to be run:

```
bash run.sh WordCount/WordCount
```

The following output is expected (ignoring compilation information):

Shell where job was executed:

```
Job ID: WordcountClientMapReduce.MapReducerConfig@6514af16
Sending File: ./src/WordCount/TropicThunderQuote.txt
Starting WordcountClient interface...
WordcountClient ->
```

Shell where local data node was executed

```
Creating Data Node...
DFSDataNode ID Node1 is starting...
DFSDataNode ID Node1 binding with NameNode Registry
Heartbeat Started
DFSFile_Id: WordcountClientMapReduce.MapReducerConfig@6514af16TropicThunderQuote.txt file block: ./dfs_storage/TropicThunderQuote_1.txtrecieved
Job WordcountClientMapReduce.MapReducerConfig@6514af16 has 1maps
Mapping File: ./dfs_storage/Node1/TropicThunderQuote_1.txt
Map Completed and written to ./dfs_storage/Node1/TropicThunderQuote_1out.txt; 0 Maps Left
Reducing Locally
Writing reduce output back to client
./dfs_storage/Node1/WordcountClientMapReduce.MapReducerConfig@6514af16Node2Output.txt
./dfs_storage/Node1/WordcountClientMapReduce.MapReducerConfig@6514af16Node1Output.txt
Reduce output files have been written
```

The lines before “Reduce output files have been written” specify where on the local machine the Map Reduce output can be found.

Running the SumOfSquares job

To run the WordCount job the following command needs to be run:

```
bash run.sh SumOfSquares/SumOfSquares
```

The following output is expected (ignoring compilation information):

Shell where job was executed:

```
Job ID: SumOfSquaresClientMapReduce.MapReducerConfig@1795f1cc
Sending File: ./src/SumOfSquares/squares1.txt
Sending File: ./src/SumOfSquares/squares2.txt
Starting SumOfSquaresClient interface...
SumOfSquaresClient ->
```

Shell where local data node was executed

```
DFSFile_Id: SumOfSquaresClientMapReduce.MapReducerConfig@1795f1ccsquares1.txt file block: ./dfs_storage/squares1_1.txtrecieved
DFSFile_Id: SumOfSquaresClientMapReduce.MapReducerConfig@1795f1ccsquares2.txt file block: ./dfs_storage/squares2_0.txtrecieved
Job SumOfSquaresClientMapReduce.MapReducerConfig@1795f1cc has 2maps
Mapping File: ./dfs_storage/Node1/squares1_1.txt
Mapping File: ./dfs_storage/Node1/squares2_0.txt
Map Completed and written to ./dfs_storage/Node1/squares1_1out.txt; 1 Maps Left
Map Completed and written to ./dfs_storage/Node1/squares2_0out.txt; 0 Maps Left
Reducing Locally
Writing reduce output back to client
./dfs_storage/Node1/SumOfSquaresClientMapReduce.MapReducerConfig@1795f1ccNode2Output.txt
./dfs_storage/Node1/SumOfSquaresClientMapReduce.MapReducerConfig@1795f1ccNode1Output.txt
```

The lines before “Reduce output files have been written” specify where on the local machine the Map Reduce output can be found.