15-640 Distributed System - Lab 3
(Map/Reduce Framework & Distributed File System)

Abhishek Sharma (asharma3)
Douglas Rew (dkrew)

A. System Architecture and Design Decisions

Map/Reduce Systemic Architecture Diagram

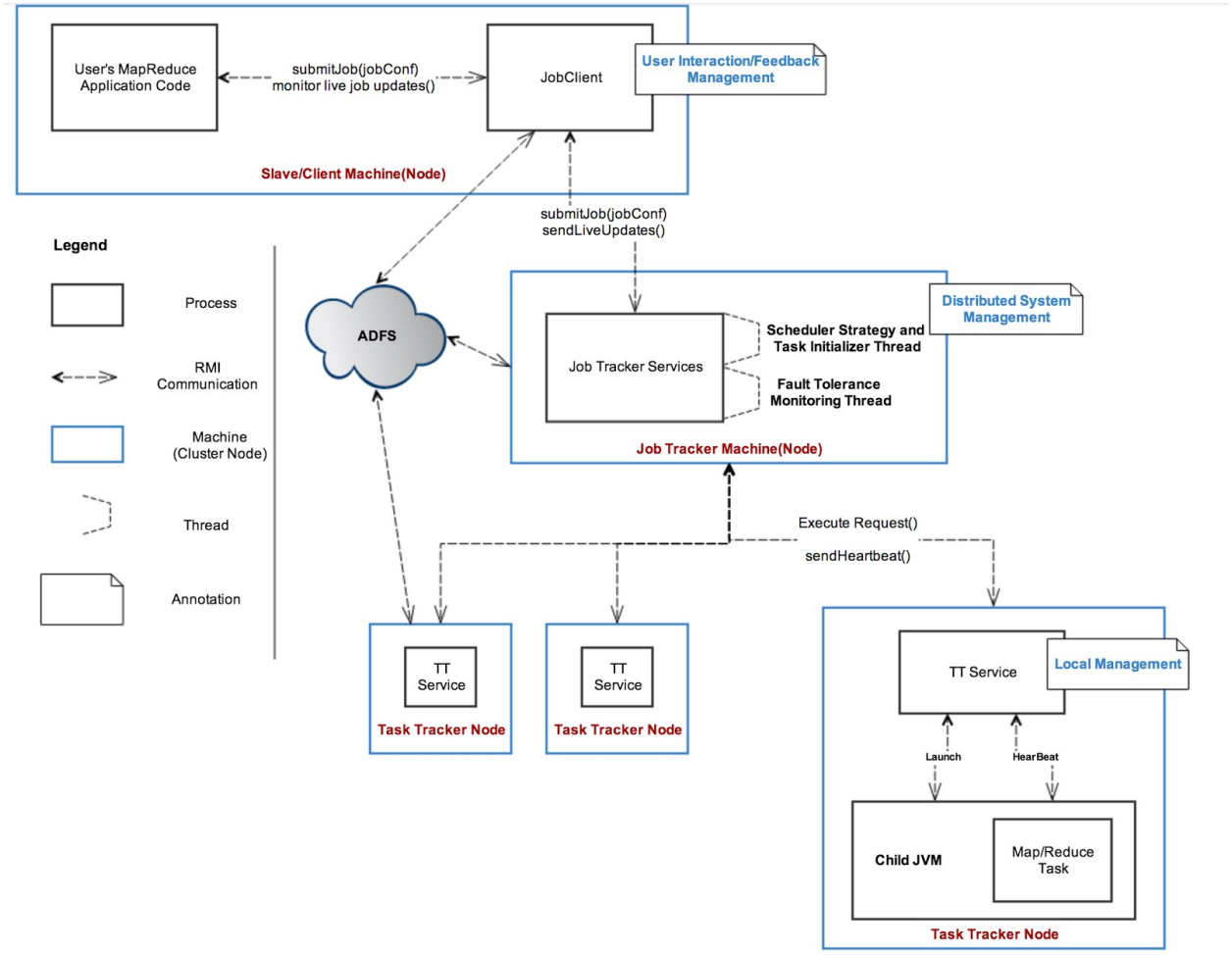

Figure 1: Our Map-Reduce Framework Architecture
(Inspired from actual map-reduce structure we have emulated the same structure)

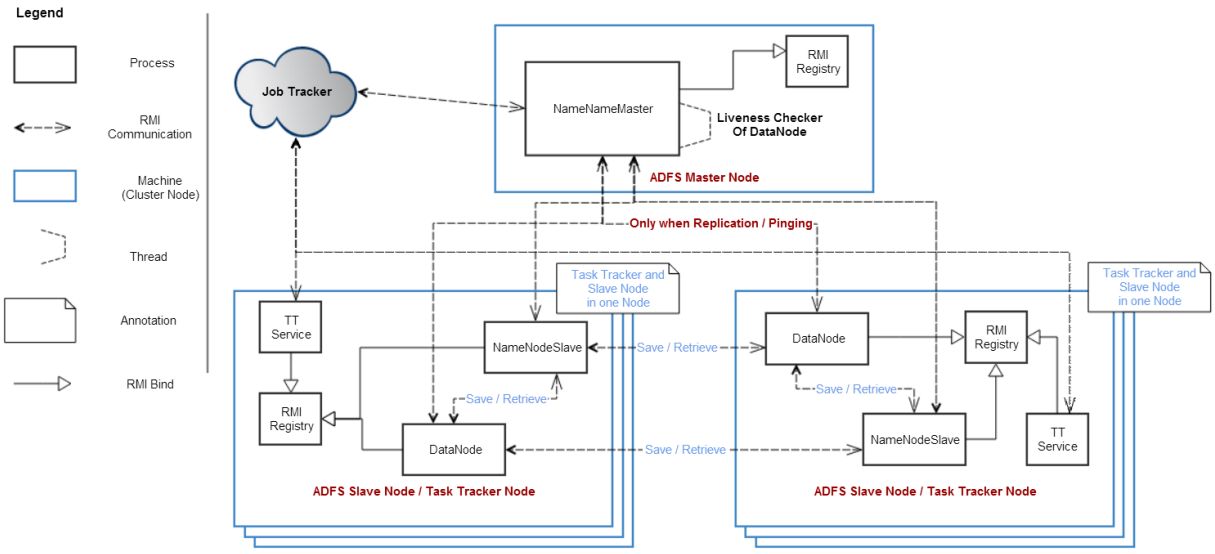

Figure 2: Our ADFS Framework Architecture

1. Map/Reduce Framework

2.1 Anatomy of Map-Reduce Workflow

As seen in the Figure 1 above we have strongly emulated the actual hadoop architecture and actually implemented each of the section shown above. From a high-level structure the system can be observed and understood by looking into three main parts:

a. User Interaction and Feedback Management - Application programmer writes his code and uses the Hadoop (in our case Local Manager) commands provided to submit his/her job. The Job-Client actually packages everything and submits the job to the map-reduce framework (i.e Job Tracker). The Job Client on slave-side also responsible for printing out feedback to the user as he prints out his job information.

b. Distributed System Management - This implies management of the complete cluster handled primarily by the Job Tracker. All the things about managing user' requests, creates appropriate tasks, the scheduler strategy and fault-tolerance is handled here. For the MR framework this is the central co-ordinator and plays the primary role in making all the magic happen.

c. Local System Management - This comprises of management at the node/machine level where the Task Tracker (TT) is in-charge. The TT dispatches each of the map or reduce tasks as requested by the Job Tracker. Also the TT reports the status to the job-tracker by sending a continuous heart-beat packaging all its local information.

The detailed-user perspective workflow is as follows:

1. User writes his application logic using the APIs (described further below) and creates a jar file.

2. User then uses the commands provided by our agent viz. Manager
(just like /bin/hadoop --commands) to actually initiated the map reduce job. Here we assume that the input file is in the adfs_files directory (we make this as the root directory for our distributed file system). The default adfs root directory is created once the adfs services are fired up, explained in the distributed file system section. The output path could be anything that the user wants on his local machine. (Commands described in Section B of the report).

3. The JobClient is instantiated by the user's Main function who packages all the job configuration as requested by the user and send's it to the Job Tracker. Before doing this the JobClient makes sure that the file is broken and replicated across the cluster and that the NameNode Acks this fact. If the file is not split then we first ask the ADFS to do it and then submit job to the job tracker

4. JobTracker accepts the requests and Gets the file chunk meta-data information from the NameNode. Based on this information the Job Tracker creates TASKS out of the Job. Both the map and reduce tasks requests are created and dumped onto a queue (logical structure). For each task a maintain a priority queue with min heap strategy such that when its time to actually execute a task we always have a Slave that has the most optimum slots available. This acts as a natural and optimum load-balancing mechanism which is a key highlight of our project. Also we can look-up what tasks can be scheduling in O(1) time due the data structure created.

5. A scheduler strategy service runs periodically on the Job Tracker and then checks if there are any pending tasks that needs to be scheduled. If they are any tasks and corresponding if the priority queue's task tracker has any slots available we dispatch the task to the corresponding task tracker. The job-tracker also runs a health checker thread that monitors the health of the cluster and re-schedules tasks if a slave fails.

6. At the task-tracker level we are at local management. TaskTracker actually deploys a Mapper or Reduce Field Agent so actually does the work. We decided to span a new JVM just like hadoop does since we do not want application programmer's faulty code to crash our task tracker. The task tracker also sends a heart-beat of the job tracker to indicate the progress of all the tasks running (which is later transferred to the user to show % done). TT also sends out the available slot information so that more work can be scheduled on it.
Note: In our system the number of mappers and reducers that can be run on each machine is pre-determined using a configuration file

Our system aims to provide maximum extension points and flexibility for the application programmer. From the hadoop guidelines and application programmer perspective we provide facility for adding:

a. Mapper Logic (Actual Map function)
b. Reducer Logic (Actual Reduce function)
c. Partition Logic (user can provide how to partition mapper's ouput)
d. Input Format (user can how to read an input file)
e. Output Format (user can how to write the final output of the reducer)
f. Local Combiner (Not implemented) - not implemented combiner as hadoop offers

2.2 Job Management and Scheduling Strategy

1. Job Tracker runs as a process and makes its services available to theoutside world via RMI and using a JobTrackerServiceProvider which is an external facing entity. We made an explicit decision to do this since we do not want to expose all the Job Tracker internals via RMI. Just things that other people need to call upon or use

2. Job Tracker breaks up the job into the tasks and adds to a data-structure so that they could be scheduled for execution. We also maintain another permanent list of all the map and reduce tasks ever being created on the Job Tracker so that if anything fails we have a way to re-schedule it.

3. The breaking up of the tasks are done based on information from the name-node so that we schedule exactly those tasks which have the potential to run on the appropriate slaves.

4. The scheduling happens via a separate scheduler make strategy thread that runs every 10 seconds under the Job Tracker Process. This thread looks if there are any new tasks(work) that needs to be done and tries to assign work to the TTs. If it can assign work it takes the task off the queue so. This applies to both map and reducer tasks.

5. Job Tracker then accepts heart-beats from the slaves (Task Trackers) every 2 seconds and then updates its data structures to update the progress of each of the scheduled tasks. This provides a mechanism to maintain almost real time status of the whole distributed system and job tracker acts as the true co-ordinator.

6. The progress of the various tasks is packaged into a Job-Info object and send across the wire to the jobclient so that we could display the almost real-time progress to the user who submitted the job.

2.3 Fault-Tolerance

1. Job Tracker also runs a TaskTrackerFaultTolerance thread that checks if the particular slaves(task trackers) are alive or not. Since we have a fixed heart-beat duration if a task tracker doesn't respond in that duration we assume it to be dead.

2. As mentioned before we still have a data structure that keeps tracks of all the tasks ever created. We extract that info and put it back on the queue. The strategy will again find this task to be executed and schedule it on the right available cluster node.

2.4 Task Execution and Management

1. Tasks execution is managed by the Task Tracker (TT) which runs as a process on each of the slave node.

2.Upon receiving a request to execute a task the TT recieves all the taskmetadata from jobtracker. At this point the necessary files to operate on and the user's code is already available. Thanks to the distributed file system.

3. Using Proces Builder the TT spans the each of the map/reduce task into its own JVM. the motivation to run it as separate process is to avoid the crashing of the Mapper/Reducer to crash the TT and also to isolate the Mapper and Reducer Field Agents who are actually doing the work. The field agents spit out their output as they are running into text log files to later viewing purposes if needed.

4. As the field agents spit out their output, this information is piggybacked in the heartbeat to the job tracker such that is available for moving final output to the reducer or the final chosen location of the user of this system.

5. The TT runs threads to monitor the status of jobs it has spanned and also send the heartbeat periodically to the Job Tracker.

6. Note: We have created a default adfs_files directory (can be changed via config file). All the intermediate output and the mapper and reducer results are written on-to this directory. Folders are created for each Job based on the job ID and the results are written on there.

2.5 Network Communication Model

We use the Java Remote Method Invocation (RMI) to do message passing and communication for this model. The original idea was to apply what we have just learned in previous project into practice. We could focus more on the MR and DFS rather that worrying about sockets.

2.6 Monitoring and Feedback for Application Programmer

1. This responsibility is delegated to the JobClient on the client node from where the user originally ran the map-reduce job.

2. The job client spawns a live status checker thread that periodically pings the job-tracker to get the TaskProgress of each of the tasks that ran under the job. We send the latest heart-beat information to this thread. By this we could print out meaningful information for the user.

3. Finally when all the relevant mappers and reducers are done we move the file to the requested directory on the local machine by the user.

Distributed File-System(ADFS) Systemic Architecture Diagram

2. Distributed File System (ADFS)

2.1 ADFS Bird's Eye View Workflow

There will be three part in our ADFS system. It will be the NameNodeMaster, NameNodeSlave and the DataNode. However for detail information of each part will be explained further down. But to highlight some points here all the interaction with the users will go through the NameNodeSlave with will communicate with the NameNodeMaster and also handle the interaction with the DataNode. So there will one NameNodeMaster and for each node there will be a NameNodeSlave paired DataNode. The commands will come through the NameNodeSlave and passed along either to the DataNode or the NameNodeMaster. But from user perspective the interaction of the ADFS will be through the Manager(from the Map/Reduce). And the user will use the Manager to specify the input file and the jar file that will be used in.

2.1.1 Workflow for the submission of a Map/Reduce Job related to the ADFS

a. Manager will ask the NameNodeSlave to distribute the JAR file that is specified by the user to all DataNodes. After distributing the nodes, ADFS will extract the Jar file in the specified folder which is configurable in the configuration file. By doing to it will ensure that all the DataNodes will be able to run the users Map/Reduce code without any problems.

b. After the JAR file is distribute, the manager will ask the NameNodeSlave again to partition the files and send it through out the DataNodes. It will be the NameNodeSlave responsibility to split the files into chunks, so it will look into the configuration file and determine the file size and distribute the partition files with the replications.

c. After the NameNodeSlave partitions the file and distributes it, it will create a InputFileInfo which will contain all the information about the input file. This will contain information about how many partitions there are and on which DataNodes there are living on. And it will register this InputFileInfo into the NameNodeMaster so that this could be used in the future. (Note that the information in this InputFileInfo will be used to determine where to run the Map/Reduce jobs because it will contain all the location of the partition files.)

d. After confirming that the input file is distributed, the manager will execute the users code to start the Map/Reduce process.

e. As the Map/Reduce process starts, it will be also the NameNodeSlave responsibility to handle the file saving and retrieving. For exmaple, the Mapper will produce an intermediate file for the reducer. The Mapper will use its local NameNodeSlave to save the information into the local DataNode and send out the information of the NameNodeSlave to the Reducer. The Reducer will use this information and look up the NameNodeSlave and ask for the data of the Mapper. And at the end the Reducer will access the NameNodeSlave on the users node and save the data on that location. At the end that data will be saved in the location where the user provided.

2.1.2 Workflow for the dumping the input file. (Partitioning a file in the ADFS)

a. The user will still use the manager in the Map/Reducer to handle this interaction with the ADFS. The user will specify that he or she wants to partition an input file the ADFS.

b. The NameNodeSlave will take the input file and partition it and send it out to all of the DataNodes using the replication configuration. To mention a bit more on the replication, if there are four dataNode up and registered to the NameNodeMaster. The NameNodeSlave will ask the NameNodeMaster for all the live dataNode that is registered in the sytem. By using the information the local NameNodeSlave will send out the partition files to all the dataNodes keeping track of the replication. And at the end all the information will be captured in the InputFileInfo which will be later on registered into the NameNodeMaster.

2.1.3 Workflow for the removing the partitioned input file.

a. The user will use the manager to handle the file removal in the ADFS. When the user specified that it want to remove the a file from the ADFS. The manager will call upon the NameNodeSlave which will trigger a command to the NameNodeMaster.

b. The NameNodeMaster will use the specified input file name and look up the InpuFileInfo which will contain will the information of where the partitioned file is.

c. After locating all the file the NameNodeMaster will ask the NameNodeSlave on each node to remove the partitioned files and return to the Manger.

d. Note that this will only remove the partitioned file, it will not remove the intermediate file and the Jar files which lives in the ADFS. (Future enhancement point)

2.1.4 WorkFlow for looking up local files in the DataNode

a. The user will ask the manager to look up the local files that are registered in the local ADFS. This means that we are looking up registered file on the local DataNode.

d. So the manager will ask the local NameNodeSlave and the NameNodeSlave will ask the DataNode to give all the information about the registered file.

c. Note that these registered file does not contain the jar files. It will contain the partitioned input file and the intermediate files which is related to the Map/Reduce jobs.

2.2 NameNodeMaster

The NameNodeMaster will be mainly responsible for three things. One will be to keep track of all of the partitioned files and keep track where it is located. Second it will keep track of the liveness of the DataNode in the system. Last responsibility will be the fault tolerance this mean that if one DataNode goes down the NameNodeMaster will trigger the replication process. By doing so we can keep the partition of the input file distributed through dataNodes. Only in the replication process and the pinging there will be direct access from the NameNodeMaster to the DataNodes.

Due to the fact that it is the NameNodeMaster that keeps all the location of the partitioned files and liveness of the DataNode(which accesses the data), the Job tracker from the Map/Reduce will ask the NameNodeMaster for this information. This will have all the necessary information that could help the Job Tracker to start the Map/Reduce. Addition to the information it will also provide some extra functionality to help determine whether the input file is exist in the ADFS and whether it is valid to use.

2.3 NameNodeSlave

The NameNodeSlave will handle the interaction between the DataNode, Manager and the NameNodeMaster. All workflow will mostly go through the NameNodeSlave and it will be paired up with one DataNode. So for each node it there is a NameNodeSlave there will also be a DataNode on that node.

The main functionality of the NameNodeSlave besides the passing along information along others will be the partitioning part. The NameNodeSlave will partition the input file and distribute to all DataNodes in the system using complete list of DataNodes which can be be retrieved from the NameNodeMaster. The partitioning size will be configurable in the configure file along with the replication number of the files.

2.4 DataNode

The DataNode will be in charge of handling the saving/retrieving the of the files which could be the input file or the jar file. If it is a jar file it is saving, it will also extract the jar file in the necessary directory.

Additionally when it is being instantiated, it will look into the config file to figure out the configured directory of the ADFS and the JAR folder. So it will write and retieve the files in a known location. By doing so we could easily maintain the files and look upon it when it is needed.

After the DataNode gets instantiated from the NameNodeSlave the NameNodeSlave will register it up into the NameNodeMaster so that all live DataNodes could be maintained up there for future use.

2.5 Fault-Tolerance

The fault-tolerance will be driven by the NameNodeMaster like it is mentioned before. The NameNodeMaster will spawn a thread that will check upon liveness of all registered DataNode. If there is detection of a DeadNode, the NameNodeMaster will start the replication process.

WorkFlow for Replication
a. The NameNodeMaster will iterate through the registered DataNodes and ping each one of them. If one returns an exception it will be considered as the DataNode to be dead.
b. After the detecting a dead DataNode the NameNodeMaster will start looking into all of the registered InputFileInfo and check whether they are affected input partitioned files in the system.
c. If there they are affected input files it will change the validation status of the InputFileInfo blocking the future use of this input file. After identifying the affected inputs by the dead DeadNodes, we will start building a list of partitioned files that needs to be replicated.
d. After locating the file list for replication, locate other live DataNodes that conatins the partitioned file and start distributing the partition files that we retrieved. After we do the replication

process we will update the InputFileInfo which the updated location of the partition file and perform a validation process.
e. The validation process will simple check whether all partition files is present in dataNodes and matches the orginial partition count of the input file. If the validation, it will change validated status to TRUE so that it could used in the future.

2.6 Network Communication Model RMI

All the communication in our system is driven by the RMI. So for ADFS each node will have its own rmiregistry running and pass along the NameNodeMasters local IP address and the port number for others in the config file which will be the NameNodeSlave and the DataNodes.

It has been mentioned above that the NameNodeSlave and the DataNode will be paired, this mean that in one node if there is a NameNodeSlave there will be a DataNode there. There two will be registered in thire own rmiregistry but will use the config file to locate the NameNodeMaster and gain the remote reference of it. To make the look up easy, we are appending the local host IP address on the bind name and using the default port.

NameNodeMaster : “NAMENODE-SERVICE”
NameNodeSlave : “NAMENODE-SLAVE-SERVICE_xxx.xxx.xxx.xxx”
DataNode : “DataNode_xxx.xxx.xxx.xxx”

Note that the NameNodeMaster and the NameNodeSlave look up name can be configured in the config file. The dataNode name is constructed and registered on the NameNodeMaster so there is no need to build up the dataNode name and do a look up. If there are any interaction with the dataNode it will always go through the NameNodeSlave except in the replication process in the NameNodeMaster where it will have the registered dataNode name.

B. Cluster Setup and System Deployment/Management

Configuration File

Configuration Parameter	Description and Sample Value
NAMENODE.REGISTRY.HOST	127.0.0.1 (Changeable)
NAMENODE.REGISTRY.POR	1099 (This is a fixed value)
JOBTRACKER.REGISTRY.HOST	127.0.0.1 (Changeable)

JOBTRACKER.REGISTRY.PORT	**1099 (This is a fixed value)**
JOBTRACKER.SERVICE	**jobtracker-service(Changeable)**
FILE.PARTITION.SIZE	**1 (Proportion to 1MB, so 5 will be 5MB partition file size)**
REPLICATION	**2 (Replication number of the partitioned file minimum value should be 2)**
NAMENODE.SERVICE	**NAMENODE-SERVICE (Changeable)**
NAMENODE.SLAVE.SERVICE	**NAMENODE-SLAVE-SERVICE (Changeable**
ADFS.DIRECTORY	**adfs_files (Changeable)**
JAR.DIRECTORY	**jar (Changeable)**
MAX.MAPPER.SLOTS	**4 (Numbers of max mapper on a task tracker)**
MAX.REDUCER.SLOTS= 4	**4 (Numbers of max reducer on a task tracker)**
CLIENT.POLICY	**client.policy (Policy file name for the RMI permission)**

=> Managers Usage Command (Manager Acts like /bin/hadoop so its user facing utility for the user to publish commanfs)

Manager --jar <JarFileName> <Execution Class Name> <InputFileName> <OutputLocation>

This will distribute the jar and the input file then execute the class name that the user has provided.

Manager --dump <Input File Name>

This will be the command to the ADFS to partition the input file and distribute it.

Manager --remove <File Name>

This will be the command to the ADFS to remove a particular file name. This will only work with files that has been dumped or entered in the ADFS by the jar command.

Manager --ls

This will print all the file which is registered into the local DataNode. Registered file could be the partitioned file and the intermediate files from the Map Reduce.

(Note: Detailed instructions about the HOW TO run and deploy are at the end of Document)

C. Application Programmer Reference Manual (API Reference)

1. Mapper

The defines the mapping functionality of the data the mappers read. The map has be to written by the user. We do not restrict the user to datatypes, he may choose the data-type of his own to do the mapping. The Mapper abstract class is defined in the Map Reduce Frame-work as follows:

```
/**
package abhi.mapreduce;

import java.io.IOException;

/**
 * @author abhisheksharma
 *
 * This class is the abstract implementation of the Mapper Class which defines the policy of using the Map Reduce framework provided
 * When a programmer uses the this Map-Reduce Framework he will have to extend this Mapper class
 * and implement the respective required methods
 *
 * Application Programmer will provide the precise implementations for these based on the KIND of Job
 * @param <VIN>
 * @param <KIN>
 * @param <KOUT>
 * @param <VOUT>
 */

public abstract class Mapper<KIN, VIN, KOUT, VOUT> implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;


    //Called once at the beginning to setup the Map Task or perform some preprocessing as per Application Need
    protected void setup() throws IOException, InterruptedException {};

    //Called once at the end (Possibly for some cleanup and housekeeping work)
    protected void cleanUp() throws IOException, InterruptedException{};


    public abstract void map (KIN key, VIN value, OutputCollector<KOUT, VOUT> outputCollector)
            throws IOException, InterruptedException ;
}
```

The map method has to be necessarily implemented by the application programmer's code. The Map method is invoked by the MapperFieldAgent. When the mapper is spawned in a cluster machine this mapperfieldagent actually delegates control to this method. The value is the value read-in as specified by the input-format (described later) by the user. The map function must take the data split it into tokens. The setup() and cleanup() will be called ONLY once before and after the map() function respectively. These functions could be used to perform some house-keeping or pre-processing work.

Following is an image from the Sample WordCount example provided with the source code which actually implements the map method as shown below. This would be typical implementation of the Map function.

```
/**
package abhi.wordcount;

import java.io.IOException;

/**
 * @author abhishekshrma
 *
 */
public class WordCountMapper extends Mapper<String, String, String, String>
{

    private static final long serialVersionUID = 1L;

    @Override
    public void map(String key, String value, OutputCollector<String, String> outputCollector)
            throws IOException, InterruptedException {

        String data = value.toString(); //Converted the Data Into String
        String[] wordsArray = data.split(" "); //User is saying that the file is space de-limited

        for(String word : wordsArray)
        {
            outputCollector.collect(word, Long.toString(1));
        }

    }

}
```

2. Reducer

The reducer abstract class as defined in our map-reduce framework is shown in diagram below. The setup() and cleanup() methods are called by the ReducerFieldAgent once before and after running the reduce job respectively. The reduce() method has to be implemented by the application programmer while the other two are optional. When the ReduceFieldAgent runs it will be delegate control to this reduce method.

```java
/**
package abhi.mapreduce;

import java.io.IOException;
import java.io.Serializable;
import java.util.Iterator;

/**
 * @author abhisheksharma
 *
 *This class is the abstract implementation of the Reduce Class which defines policy of usage
 *The Application Programmer must provide the appropriate implementation for Reduce based on the kind of job
 *
 * * Application Programmer will provide the precise implementations for these based on the KIND of Job
 * @param <V2>
 * @param <K2>
 * @param <K3>
 * @param <V3>
 */
public abstract class Reducer<K2,V2,K3,V3> implements Serializable{

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    //Called once at the beginning to setup the Reduce Task or perform some pre-processing as per Application Need
    protected void setup() throws IOException, InterruptedException {};

    //Called once at the end (Possibly for some cleanup and housekeeping work)
    protected void cleanUp() throws IOException, InterruptedException{};

    // The Reduce functionality that needs to be overwritten by the Application Programmer
    // Provide the reduce functionality
    public abstract void reduce(K2 key, Iterator<V2> values, OutputCollector<K3, V3> output)
            throws IOException;
}
```

This is a sample reducer code from our word count example. This would be typical implementation of the reducer on the application programmer side:

```java
/**
package abhi.wordcount;

import java.io.IOException;
import java.util.Iterator;

import abhi.mapreduce.OutputCollector;
import abhi.mapreduce.Reducer;

/**
 * @author abhishekksharma
 *
 */
public class WordCountReducer extends Reducer<String, String, String, String> {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Override
    public void reduce(String key, Iterator<String> values,
            OutputCollector<String, String> output) throws IOException {

        long addition = 0;  //Final summation of the word-count

        while(values.hasNext()) //While we have similar tokens
        {
            addition += Long.parseLong(values.next());
        }

        output.collect(key, Long.toString(addition));
    }

}
```

3. Partitioner

The partitioner interface in mapreduce framework is shown below. The number of partitions produced by the Mapper is driven by this function. The getPartition() function is called by the MapperOutputCollector of the Map-Reduce framework. When mappers are spitting out their intermediate files they use this strategy to do it.

```
/**
package abhi.mapreduce;

import java.io.Serializable;

/**
 * @author abhisheksharma
 *
 *This interface defines the methods to be implemented by the Partitioner
 *The partitioning policy will be provided by the App user
 *Generally in Hadoop this is just a HashFucntion of the key based on the number of reducers
 *choose for the job.
 */
public interface Partitioner<KOUT> extends Serializable{

    //The Paritioner of the App. Programmer Side will implement this method.
    public int getPartition(KOUT key, int numofReducers);
}
```

Following is a sample implementation of the partition function

```
/**
package abhi.wordcount;

import abhi.mapreduce.Partitioner;

/**
 * @author abhisheksharma
 *
 */
public class WordCountPartitioner implements Partitioner<String> {

    /**
     *
     */
    private static final long serialVersionUID = 1L;


    @Override
    public int getPartition(String key, int numofReducers) {
        return Math.abs(key.hashCode()) % numofReducers;
    }

}
```

4. OutputFormat

The outputformat abstract class is defined the following way. The user extends the outputformat class from the MR framework and then overrides the format function. This is given so that the user has the flexibility to specify the output format of his choosing. The method gets called when the reducers are spitting out there result file.

```
/**
package abhi.mapreduce;

import java.io.Serializable;

/**
 * @author abhisheksharma
 *
 * OutputFormat describes the output-specification for a Map-Reduce job.
 *
 * Two main functions:
 * 1. Provided with the key and value this method should return a String that must be written to the Output of the Map-Reduce Job
 * 2. Check if the OUTPUT directory already exists
 */
public abstract class OutputFormat<KOUT, VOUT>  implements Serializable{

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public abstract String format(KOUT key, VOUT value);


}
```

Following is a sample implementation from the wordcount example in our framework.

```java
/**
package abhi.wordcount;

import abhi.mapreduce.OutputFormat;

/**
 * @author abhisheksharma
 *
 */
public class WordCountOutputFormat extends OutputFormat<String, String> {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Override
    public String format(String key, String value) {

        return key + "\t" + value + "\n";
        //We write key values on new lines separated by tabs
    }


}
```

4. InputFormat

The InputFormat in our system is an Iterator. The programmer needs to implement the hasNext() and next() functions. The InputFormat provides the file operator to manipulate the I/O and basically defines how we should produce a record. These functions are called upon by the MapperFieldAgent who actually runs the map task:

```java
/**
package abhi.mapreduce;

import java.io.IOException;


/**
 * @author abhisheksharma
 *
 */
public abstract class InputFormat implements Iterator<KeyValueConstruct> {

	protected String filename;

	protected long offset;

	protected long fileSize;

	protected RandomAccessFile raf;


	protected InputFormat(String filename) throws IOException {
	  this.filename = filename;
	  this.offset = 0;
	  this.raf = new RandomAccessFile(filename, "r");
	  this.fileSize = this.raf.length();

	  this.raf.seek(offset);
	}

	protected boolean hasByte() throws IOException {
	  if (this.raf.getFilePointer() < (this.offset + this.fileSize))
	    return true;
	  return false;
	}
    }
```

A sample inputformat implementation from our wordcount examples is shown below:

```java
import java.io.IOException;

/**
 * @author abhisheksharma
 *
 */
public class WordCountInputFormat extends InputFormat
{
    public WordCountInputFormat(String filename) throws IOException {
        super(filename);
    }

    @Override
    public boolean hasNext() {
        try {
            return this.hasByte();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return false;
    }

    @Override //Function to construct the KeyValue
    public KeyValueConstruct next() {

        try
        {
            String line = this.raf.readLine();
            String key = Integer.toString(line.length());
            String value = line;

            return new KeyValueConstruct(key, value);
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }

    @Override
    public void remove() {

    }
}
```

5. The Main Class (Entry Point for Application Programmer's Code)

This is entry point of the application programmer's code. Following is a sample implementation of the word-count example. The central idea here is to create the job configuration object. After the jobclient object is created and then the jobConf object is passed onto it. The JobClient will spwan a thread that will print out the current status of the job.

```
import java.io.IOException;

/**
 * @author abhisheksharma
 *
 */
public class WordCount implements Serializable {

    private static final long serialVersionUID = 1L;

    public WordCount(){}

    public static void main(String[] args) {

        if(args.length !=2)
        {
            System.out.println("Invalid Usage. Please pass the input and output path");
        }

        JobConf jobConf = new JobConf();
        jobConf.setJobName("WordCount");

        jobConf.setInputPath(args[0]);
        jobConf.setOutputPath(args[1]);

        jobConf.setInputFormatClassName("abhi.wordcount.WordCountInputFormat");
        jobConf.setPartitionerClassName("abhi.wordcount.WordCountPartitioner");
        jobConf.setOutputFormatClassName("abhi.wordcount.WordCountOutputFormat");

        jobConf.setMapperClassName("abhi.wordcount.WordCountMapper");
        jobConf.setReducerClassName("abhi.wordcount.WordCountReducer");

        jobConf.setReducerNum(2);

        JobClient jClient = new JobClient();
        try
        {
            boolean result = jClient.submitJob(jobConf);

        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
```

D. Two Samples Jobs

1. Classic Word Count Example

This example is used to describe the Application Programmer Interface above. Counts the number of unique words in provided input. We have provided a sample file text file called: **WordCount.txt** that contents from one of the team members blog.

Final Output :

hello	14
world	23
life	5

2. Anagram Example

This examples reads a list of words from a file and tries to find out other words in the file that are its anagrams. We have provided a file for this called **Anagram.txt**.

Final Output:

aabelluv	valuable,valuable,valuable,
Waddeensy	Wednesday,Wednesday,Wednesday,
aacilpt	capital,capital,capital,

E. System Highlights and "Cool" Features

- In the Map/Reduce framework we are managing the scheduling strategy using a min heap data structure that we created. So the inspiration behind this is was that once the system up and running many people would map and reduce jobs. We want our cluster (with whatever capacity) to do maximum work meaning we need to utilize the resources efficiently accomodating the network and code delays. There could be a point the system when many tasks might be queued up. We want to schedule a task as soon as we know some slot in some machine is available via heartbeat. We used a min heap queue and keep a list of possible nodes where taks could run ready. Due to this the moment we get a slot we can get the task in O(1) time and schedule it immediately.

- Fault tolerance
    - In the ADFS, the NameNodeMaster is checking on all dataNodes for the liveness through pinging. If there is a dead deadNode detected it will start the replication process and replicate the existing partition files on that dead dataNode. After the replication the InputFileInfo will be updated.

    - In the Map/Reduce framework, if the Task Tracker dies it will be detected by the Jab Tracker via heartbeat. When a failure is detected the Job tracker will try to assign the failed task to the available next location. The task is built with the information of where all the replication is living on . So by that information if one task tracker fails, the job track sends the task to the next location. *We have implemented all of this but this has some bugs we are in process to resolve.*

- Input file distribution handling : Knowing that the input file might be huge we wanted a way to handle the partitioning of the file and the sending of the file in one location. So we are using the NameNodeSlave to handle this. When we get a request of an input partition, we go up to the NameNodeMaster and get the most updated list of the DataNode. While we go through the file and partitioning it we will send file to dataNodes keeping in mind the dataNode replication. And to handle management of these file, we are using the class called the InputFileInfo and carry this over.

F. Limitations and Future Enhancements

1. The status of the job while running is not in real-time or say 100% accurate. We just do a cumulative percent average based on the completions %age of all the tasks belonging to that particular job.

2. Currently the MAX number of mappers or reducers that can be run on each system is configured by the configuration file. An ideal implementation would be to specify only the % of the resources we want to use. In that case the framework would find out Memory and CPU usage on each of the cluster nodes and dynamically tune itself to schedule tasks. We currently do not do this. Our task scheduling is based on the how many more slots are available on each of the node.

3. We have not built any monitoring or management or logistics service on the JobTracker side to print what's going on the system or produce any reports. A really necessary functionality to be build in future for real world usage would be to provide for this so that administrator can produce reports and do some fine tuning for the system.

4. Hadoop provides implementation for RecordReader that can be used to help the MR Framework read a record. We currently do that do that and assume some fixed formats that need to be used.

5. Hadoop has robustness to deal with bad records and also re-priortize dynamically the tasks that are running to sort of have a fair execution. We currently do not have priority on the job. Each job is scheduled as it comes through. Also, if the Job-Tracker dies there is no way to recover.

6. Although our distributed file-system provides for all the services that map reduce framework we have not gone to the lengths to implement every single command or functionality that HDFS offers.

7. The current ADFS only remove files that has been partitioned by our system. The intermediate files that gets created by the Mappers/Reducers only gets tracked. There is no functionality to remove it now. This also goes same with the Jar files, we only move to Jar files location and extract it, there is no removal of these file. We decided to push this out because the input file will be the most large file chucks in the system. So we built in functionality for that however left out the other minor file tracking.

G. How to run and informational guide

1. Extract the zip file and open a command line and go to the extraction folder.
    a. There will three folder “bin, Example_MapReduce_Code, and src” with some additional files.
    b. Type : make
    c. This will build all the code in the “bin” folder.

2. Go to the bin folder.
    a. There will be multiple files in the bin.
    b. anagram.jar, Anagram.txt, client.policy, Configuration.properties, wordcount.jar, WordCount.txt

3. Modify the configuration file for the NameNodeMaster and the Job Tracker (Note make sure that the port 1099 is free in the machine) (Machine 1)
    a. NAMENODE.REGISTRY.HOST=128.237.205.17 (example)
    b. NAMENODE.REGISTRY.PORT=1099
    c. JOBTRACKER.REGISTRY.HOST=128.237.205.17
    d. JOBTRACKER.REGISTRY.PORT=1099
    e. If there are others thing to change please refer above for more details.

4. We are in the bin folder now and we are going to start to run the Map/Reduce Services

5. Start the RMI (Windown : start rmiregistry, Unix: rmiregistry) (Machine 1)

6. Open another command line go to bin folder (Machine 1)
    a. Type “java abhi.adfs.NameNodeMasterImpl”

7. Open another command line go to bin folder (Machine 1)
    a. Type “java -Djava.security.policy=client.policy abhi.adfs.NameNodeSlaveImpl”

8. Open another command line go to bin folder (Machine 1)
    a. Type “java abhi.mapreduce.JobTracker”

9. Open another command line go to bin folder (Machine 1)
    a. Type “java abhi.mapreduce.TaskTracker”

10. Now open another machine and unzip the file and go to the extraction folder (Machine 2)
    a. Type “make”
    b. Go to the Bin folder and update the config file with the correct IP address and the host.
    c. Start the rmiregistry

11. Open another command line go to bin folder (Machine 2)
    a. Type “java -Djava.security.policy=client.policy abhi.adfs.NameNodeSlaveImpl”

12. Open another command line go to bin folder (Machine 2)
    a. Type “java abhi.mapreduce.TaskTracker”

13. Now open another machine and unzip the file and go to the extraction folder (Machine 3)
    a. Type “make”
    b. Go to the Bin folder and update the config file with the correct IP address and the host.
    c. Start the rmiregistry

14. Open another command line go to bin folder (Machine 3)
    a. Type “java -Djava.security.policy=client.policy abhi.adfs.NameNodeSlaveImpl”

15. Open another command line go to bin folder (Machine 3)
    a. Type “java abhi.mapreduce.TaskTracker”

16. Open another command line go to bin folder in any machine we are going to start a Map/Reduce example code - Word Count

a. The example code has already been compiled and packaged a zip file that is the *.jar files we see in the bin folder
b. Type “java -Djava.security.policy=client.policy abhi.mapreduce.Manager --jar wordcount.jar abhi.wordcount.WordCount WordCount.txt RESULT”
c. To run the job again update the output path

17. Open another command line go to bin folder in any machine we are going to start a Map/Reduce example code - Anagram
    a. The example code has already been compiled and packaged a zip file that is the *.jar files we see in the bin folder
    b. Type “java -Djava.security.policy=client.policy abhi.mapreduce.Manager --jar anagram.jar abhi.anagram.AnagramMain Anagram.txt RESULT1”
    c. To run the job again update the output path

H. How to drive the Fault Tolerance in ADFS

1. To testing the fault tolerance of the ADFS, start one NameNodeMaster with three NameNodeSlaves running on two different machine.(Follow the instructions above)
2. After starting three Slaves
    a. Type “java -Djava.security.policy=client.policy abhi.mapreduce.Manager --dump Anagram.txt”
    b. This will partition the file in to three location.
    c. Now kill one Slaves and notice that the NameNodeMaster will detect that one node has been killed and starts the replication process.
    d. If the replication process has succeeded it will print out information on the NameNodeMaster.

    e. Type “java -Djava.security.policy=client.policy abhi.mapreduce.Manager --remove Anagram.txt”
    f. Type “java -Djava.security.policy=client.policy abhi.mapreduce.Manager --ls”
    g. See that there are no files in the DataNode

I. How to ensure the system is running and see the intermediate output