

## 15640 Project 3 MapReduce Facility

Name: Yusi Zhang    Andrew ID: yusiz

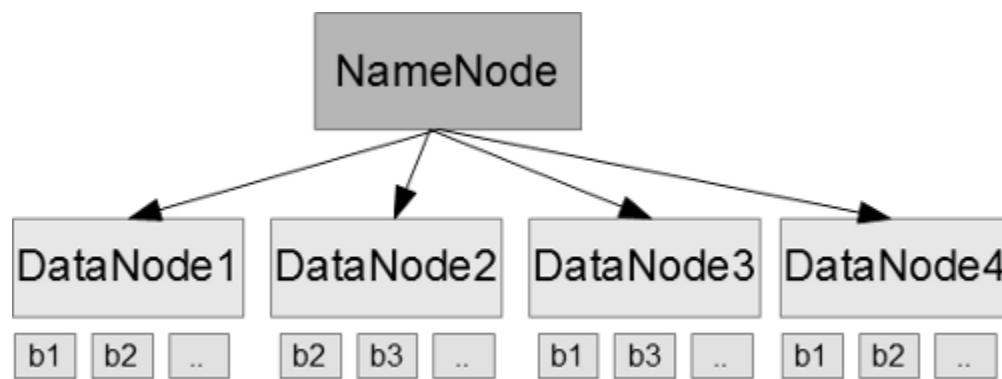
Name: Yue Zhuo    Andrew ID: yuezhuo

### Part 1. Framework

#### DFS

Our implementation of the MapReduce Facility contains two main parts of frameworks, the Distributed File System(DFS) and the MapReduce Infrastructure.

Our Distributed File System is implemented with the master/slave architecture.



The DFS consists of a single NameNode, which is a master server, and a number of DataNodes.

The NameNode is responsible for:

1. Representing the DFS to communicate with the client side to respond to the client's request of uploading and downloading files
2. Communicating with the slaves, managing the slave nodes by tracking their status
3. Storing and managing the metadata of the file splits

The DataNode is responsible for:

1. Serving the read and write requests sent from the client side
2. Storing the actual file splits.

The DFS can store very large files across the DataNodes reliably, because the blocks of a file are replicated for fault tolerance. The file replication mechanism is introduced in part2.

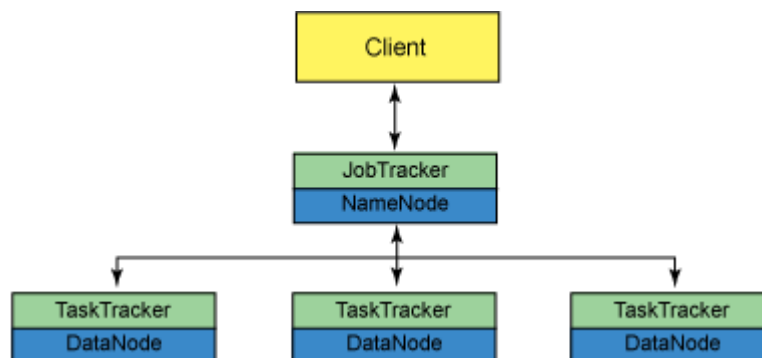
## MapReduce

Our MapReduce infrastructure adopts the same centralized architecture as Hadoop MapReduce. There is a single Master node runs as the master node, which is responsible for:

1. Managing the Slave nodes and their statuses
2. Listening to the client's request to run a new job
3. Divide the job into several different Mapper and Reducer tasks, and assign the tasks to the Slave nodes
4. Response to the client with the job result and output files

For each job submitted from the client to the Master Node, a Job Tracker is generated for managing and tracking the job. The Job Tracker is responsible for:

1. Divide the job into several map tasks and reduce tasks.
2. Assign the tasks to the Slave Nodes.
3. Deal with the slave node failure.



## Part 2. Design Features and Functions

### Communication

The client side, the Master Node and the Slave Nodes communicate with each other via socket connection. They send Serializable messages with different flags and specific contents to each other.

### DFS

The client can input command into console to upload and

The client split files into small blocks before it puts the file onto the DFS. Every time the NameNode of DFS receives the file put request from the client, the Master Node tells the client to put which file onto which DataNode. To reduce the workload of the NameNode, the client establish communication with the DataNode directly and upload the file.

As mentioned before, our DFS replicates the file blocks for fault tolerance. The block size and replication factor can be configured in the Configuration File. The files in DFS are write-once, read-many. Thus, the NameNode also tells the client to write the replicas to which nodes.

To guarantee that the original file block and its replicas are stored in different DataNodes, we first randomly choose an active DataNode, and then choose the next n DataNodes to store the next n replicas. At the same time, the NameNode manages a table to maintain which file block is stored on which DataNodes.

The detail sequence of the uploading file from client to the DFS is described below:

1. The user inputs the “put fileName” in the console
2. The client split the file into several blocks based on the chunk size
3. The client sends the “FILE\_PUT\_REQ\_TO\_MASTER” to Master
4. The Master responds the “AVAIL\_SLAVES” to the clients with a list of Slave Nodes
5. The client connects the Slave Nodes, send a message “FILE\_PUT\_REQ\_TO\_SLAVE” to ask them to open an active port to receive the file
6. The Slave Nodes return the active port numbers and listen to the port
7. The Client sends the “FILE\_PUT\_START\_TO\_SLAVE” message with the file in the content of the message
8. If the Slave receives and stores the file successfully, it returns a “FILE\_PUT\_DONE” message to the Master Node
9. The Master Node put the file block and the list of the Slave Nodes into its table

## MapReduce

As soon as the Master Node starts, it registers itself to be the Master Node.

There is a Scheduler running on the Master Node as a separate thread to listen to all the messages sent to Master Node’s main port and handle the messages according to the message type and content. Similarly, there is a Slave Scheduler thread running on every

Slave node to listen to the messages sent to the Slave Node's main port and handle the messages.

When a Slave Node starts, it sends a "REG\_NEW\_SLAVE" to the Master Node to register itself as a Slave Node, and the Master Node will put it into the SlavePool.

As soon as the client runs a new Job, a Job object is generated for maintaining the Job information and tracking the status of the Job accomplishment. The Job object sends itself to the Master Node. After the Master Node receives the new Job, it first checks which file blocks are needed for running this job and where they are stored. Then the Master Node generates several Tasks based on the number of file blocks, and assign them to the Slave Nodes that stores the relative file blocks locally.

In our system, the number of the map tasks is decided by the number of file splits. First, the master looks input file, and checks which file block is stored on which Slave. Then, the Master assigns the map task that reads and manipulates a specific file block to the Slave that stores the file block. This assignment method is chosen because:

1. Every file split except the last one is in the same size, so the running time of every map task is theoretically same with each other
2. Every map reads and manipulates the file locally, so there is no file transfer at this step. This not only avoids the time cost and error appearance the file transfer, but also saves the memory.
3. This is same as the assignment method used in Hadoop MapReduce. In the Hadoop MapReduce, the number of maps is usually driven by the total size of the inputs, that is, the total number of blocks of the input files.

The task is assigned to the Slave Node via a "NEW\_MAPPER" or "NEW\_SLAVE" message with the task information in the message content.

For every map and reduce task, the Slave Node runs a separate MapTracker and ReduceTracker thread for tracking the status of the task.

For the MapTracker:

1. Find the input file
2. Get the Mapper class using reflection
3. Run the mapper
4. Shuffle the Mapper output file based on the number of the Reducer

5. Generate the output file and inform the Mapper task accomplishment to the Master
6. If all the Mapper tasks accomplish successfully, the Master will assign the Reducer tasks
7. Send the shuffled Mapper output files to the Reducers

In our system, the Reducer number can be decided by the user by modifying the configuration file. Similar with choosing the Mapper Slave Nodes, we choose the Reducer Slave Nodes randomly, and the Master Node sends the Slave information of the Reducers with the task.

After finishing the Mapper tasks, the output file is split into blocks based on the number of reducers.

We mod the hashcode of the key of the mapper output data to decide which piece of file this context should be written into.

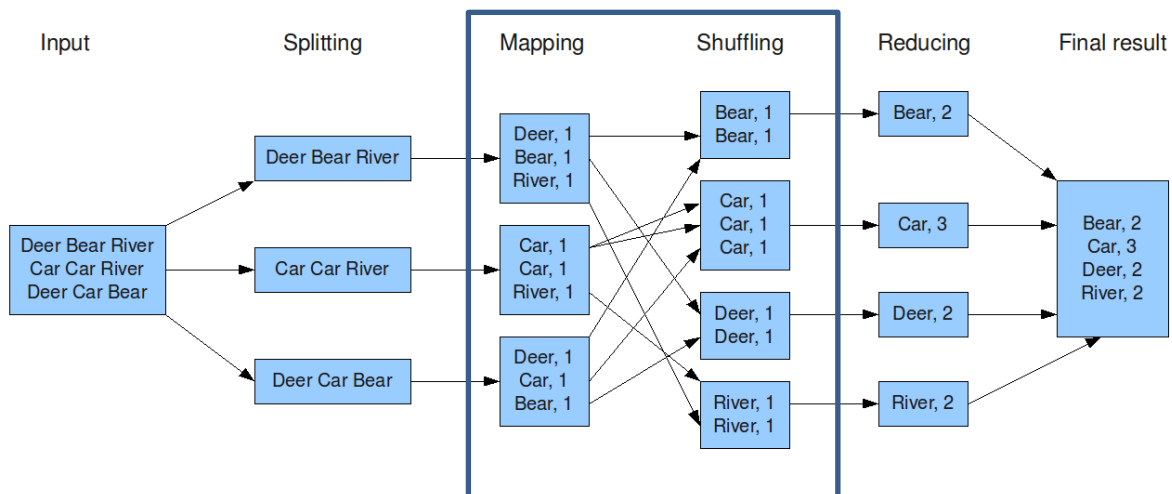
$$\text{File\_ID} = \text{Mapper\_output\_key's\_hashCode}() \% \text{number\_of\_Reducers}$$

Then the files with the same File ID will be sent to the same Reducer based on the order of Reducer List.

This algorithm is chosen because:

1. We can make sure that the context with the same key in different Mappers goes to the same Reducer
2. We can split the output files evenly
3. No duplicate file or extra space is needed, and every file split only need to be transferred once

Look into the word count as an example:



For the ReduceTracker:

1. Downloads the Mapper output files from different Mappers
2. Sort the key-value-pairs based on the key's hashCode
3. Merge the values of the same key value to a list of the values
4. Run the Reducer
5. Inform the Master the Reducer is successfully accomplished and the output file name
6. If all the Reducers are successfully accomplished, the Master Node informs the Client side of the successful accomplishment of the Job and the Reducer output files
7. The Client can download the Reducer output files based on the Reducer output directory that Master Node gives him

## Concurrency

Our systems supports multi-job and multi-task. This can be realized because:

1. Every Scheduler, MapTracker and ReduceTracker, file\_Upload and file\_Download is a thread that can run separately and concurrently.
2. The slave pool, job pool and the tables for the task and file management are Concurrent Hashmaps.
3. The Master Node and Slave Nodes listen to the main port to receive messages. If other manipulations are required regarding this message, the node will open an active port, so the connection of the main port cannot be influenced, and the main port can continuously receive messages.
4. When running multiple jobs concurrently, every job's socket listens to a different client main port, which is set up in the Configuration file, so that the jobs will not influence with each other. This will be introduced in the "Test and run" part.

## Robustness

Fault tolerance is taken into consideration in both the DFS and the MapReduce infrastructure.

For the DFS, there are several kinds of failure that we need to handle:

1. DataNode failure
2. The network partition between the NameNode and the DataNode
3. NameNode failure

For the MapReduce, there are several kinds of failure that we need to handle:

1. Task failure
2. Slave Node failure
3. Job failure
4. Master Node failure

We have two mechanisms for fault tolerance: file replication and heartbeat.

The file replication is introduced in the DFS part.

The Heartbeat mechanism is designed for fault tolerance. Each NameNode sends a Heartbeat message to the DataNode periodically, and the DataNodes reply to the Heartbeat as soon as it receives the message. It might be due to the DataNode failure or the network partition that the DataNode fails to send the message. As soon as the NameNode detects this condition by the absence of a Heartbeat message, the NameNode does the following operations:

1. Marks the DataNodes without recent Heartbeats as “fail” and does not forward any new IO requests to them.
2. Checks the meta-data files, and move the file blocks on the failed DataNode to other alive DataNodes.
3. Checks the list of running jobs to see the failed Slave Node influences which jobs.
4. Kills other running tasks of the same job, and sends a message to the Job Tracker to updates the status of failure.
5. As soon as the Job Tracker notices the failure, it sends “NEW\_JOB” message to Master Node again to restart the job.

Currently in our system a DataNode failure means the job failure, the running tasks of the job will be killed by the master, and the job will be to restarted. We implemented the recovery mechanism this way because of the limit of time.

For the next step, we will first distinguish the Slave Node failure, the task failure and the job failure, and second, restart only the failed tasks instead of the whole job.

## Configuration

The following parameters can be configured by the user in the configuration file “ConfigFile.txt”:

MasterIP: the client and slaves must know the Master's IP address

MasterMainPort

SlaveMainPort

ClientMainPort: if run multiple jobs at the same time, every job should have a different Port

SlaveHeartBeatPort

StartPort: the nodes assign active ports from the start port to the end port

EndPort: the active port number will not be reused

ClientIP

ClientPort

ChunkSize: the chunk size of file splits

HearBeatFreq: the frequency of heartbeat in ms

Replica: the number of copies of every file split

ReducerTaskSplits: the number of Reducer Nodes

## **Part 3. Implemented, Unimplemented Functions and Potential Improvements**

### **Implemented:**

We have implemented all the functions mentioned above.

1. DFS framework
2. Client can put files to DFS and get files from DFS
3. MapReduce infrastructure
4. the MapReduce job can be run on the MapReduce
5. Heartbeat and Recovery mechanism

### **Potential Improvements:**

1. In our system, the ClientMain class represents the client side of the DFS. For the MapReduce infrastructure, there should also be a Client side that can communicate with the Master Node, run different Jobs, receive and manages the Job outputs. Currently we run the Job directly by running the main class of the Job, ex. "WordCountExample.java".



2. In our system, a “Job” object is generated right after the Client launches a Job. We can separate the “Job” object and the JobTracker.
3. In our system, the Scheduler runs on the Master Node for handling different messages, scheduling different tasks, maintaining all the status of Slaves, Jobs and Tasks. We can separate the “Job Scheduler” from the Master’s Scheduler to schedule different jobs running on the MapReduce. Similarly, we can separate the Task Scheduler from the Slave Scheduler.
4. The MapTracker class and ReduceTracker class can be integrated as the TaskTracker.

## Part 4. Two test examples

1. Word count

WordCount example is the most classic example for the MapReduce program. The Mapper gets all the words of the input line and output “word - one” pair to the Mapper Output file. In the Reduce function, we iterate through the values for a single key and increase the counter. After process all the values for a single key, we output the “key - counter” pair to the output file.

2. N-Gram language model generation

An  $n$ -gram is a contiguous sequence of  $n$  items from a given sequence of text or speech. (<http://en.wikipedia.org/wiki/N-gram>)

Input: a text file.

Output: the phrases with the length from one to five and their appearances.

## Part 5. Test and run our system

### Test DFS:

1. Modify the “ConfigFile.txt”: clarify the “MasterIP” and “ChunkSize”
2. Run the MasterMain.java on the Master Node with the configuration:  
src/ConfigFile.txt
3. Run the SlaveMain.java on the Slave Nodes with the configuration:  
src/ConfigFile.txt
4. Run the ClientMain.java on the Client Node

## MasterMain:

```
start master                ← run MasterMain.java
start the scheduler on 15640
start the heart beat thread on master
Receiving REG_NEW_SLAVE / I am a new slave from /128.2.246.36:59539 ← run SlaveMain.java
Receives a REG_NEW_SLAVE message: I am a new slave
Register slave 0 with the IP: /128.2.246.36                ↓run ClientMain.java and input command
Receiving FILE_PUT_REQ_TO_MASTER / communication.WriteFileMsg@5b7ee8b8 from /128.2.246.37:60211
Receives a FILE_PUT_REQ_TO_MASTER message: communication.WriteFileMsg@5b7ee8b8
Write File: file.txt blk:0
ram next is 0
id set size is 1
Sending AVAIL_SLAVES / [node.SlaveInfo@1f5b0afd] to /128.2.246.37:60211
Send the slave list from the master to the client
filename: file.txt_blk0 0 ← file upload success. You can see the file.txt_blk0 on Slave Node
```

file.txt\_blk0 means the first block of the file, and it is stored on the Slave with SlaveId 0.

## SlaveMain:

```
start slave                ← run SlaveMain
slave is listening 0.0.0.0/0.0.0.0:1100011000
Sending REG_NEW_SLAVE / I am a new slave to /128.2.246.37:15640
send msg from slave I am a new slave                ← send message to master to register itself
start the heart beat thread on slave                ↓ receive msg from client
Receiving FILE_PUT_REQ_TO_SLAVE / I will upload a file to the slave from /128.2.246.37:60212
the scheduler receives a FILE_PUT_REQ_TO_SLAVE message
Message received from /128.2.246.37:60212 type: FILE_PUT_REQ_TO_SLAVE; content: I will upload
Sending PORT / 9001 to /128.2.246.37:60212                ← send active port number to client
send the reply to client 9001
send the port number from the slave to the client 9001
new socket from client/128.2.246.37:60213
Receiving FILE_PUT_START_TO_SLAVE / file.txt_blk0 from /128.2.246.37:60213
slave is listening 0.0.0.0/0.0.0.0:1100011000
File Downloading.....
file.txt_blk0                ← receive file from client
Download succeed!file.txt_blk0
```

You can see the file named “file.txt\_blk0” on the Slave Node

## ClientMain:

1. Input the command “put” in the console
  2. Input the filename “file.txt” in the console
- \* We assume the file is under the project directory

```
Enter your command:
put
Please input filename:
file.txt
Sending FILE_PUT_REQ_TO_MASTER / communication.WriteFileMsg@7bc9a690 to /128.2.246.37:15640
Receiving AVAIL_SLAVES / [node.SlaveInfo@24a4e2e3] from /128.2.246.37:15640
the client receives a message from the master
the client will connect:/128.2.246.36 11000
send the msg: I will upload a file to the slave
Sending FILE_PUT_REQ_TO_SLAVE / I will upload a file to the slave to /128.2.246.36:11000
client send the msg to slave
Receiving PORT / 9001 from /128.2.246.36:11000
the client receives a message from the slave
client connect to slave via assigned port 9001
Sending FILE_PUT_START_TO_SLAVE / file.txt_blk0 to /128.2.246.36:9001
Uploaded file.txt_blk1
File Uploading...
Upload succeed!file.txt_blk0
```

Right after you upload the file from the client to the DFS, you can delete the local file on the Client node, and test the function of download the file from the DFS.

1. Keep the MasterMain node and SlaveMain nodes alive
2. run the ClientMain.java again
3. Enter "get"
4. Enter file name
5. Enter the slave id that the file is on
6. The file is downloaded successfully, you can see the file downloaded to the client

```

Enter your command:
get
Please enter file name
file.txt
Please enter slave id
0
Sending GETFILE / node.FileInfo@73d6776d to /128.2.246.37:15640
Receiving null / node.SlaveInfo@26fd68b1 from /128.2.246.37:15640
file : file.txt is on slave id: 0IP: /128.2.246.36
Sending FILEDOWNLOAD / file.txt to /128.2.246.36:11000
File Downloading.....
Here are first 5 lines of filefile.txt
  GitHub Help
  Contact Support
  Return to GitHub
  Site Policy   GitHub Terms of Service
  SearchArticle last updated on 09 Jul 14
  GitHub Terms of Service
  file.txt
Download succeed!file.txt

```

### Test MapReduce:

1. Modify the "ConfigFile.txt"
2. Run the MasterMain.java on the Master Node with the configuration: src/ConfigFile.txt
3. Run the SlaveMain.java on the Slave Nodes with the configuration:  
src/ConfigFile.txt
4. Run the "WordCountExample.java" on the Client side with the configuration:  
src/ConfigFile.txt    inputFileNames    outputFileNames
5. Map tasks finish successfully and generate Map output files  
The files with names "JobID\_TaskID\_MapResult\_ReducerID" are generated on Map Nodes, for instance, "0\_0\_MapResult0".
6. The Reduce Nodes receive the shuffled Map output file and rename the file as "JobID\_TaskID\_Reduce", such as "0\_0\_Reduce".
7. Reduce tasks finish successfully and generate Reduce output files  
The files with names "JobID\_TaskID\_reduceResult\_ReduceID"
8. Every Reduce Node tells the Master which Reduce output file is generated
9. The Master returns the Reduce output file directory to the Client:  
1\_1\_reduceResult0 is on slave id: 0
10. The client can download the result file

## Master side:

---

```
start master
start the scheduler on 15640
start the heart beat thread on master
Receiving REG_NEW_SLAVE / I am a new slave from /128.2.246.36:59677
Receives a REG_NEW_SLAVE message: I am a new slave
Register slave 0 with the IP: /128.2.246.36
Receiving FILE_PUT_REQ_TO_MASTER / communication.WriteFileMsg@5b7ee8b8 from /128.2.246.37:60462
Receives a FILE_PUT_REQ_TO_MASTER message: communication.WriteFileMsg@5b7ee8b8
Write File: file.txt blk:0
1
ram next is 0
id set size is 1
Sending AVAIL_SLAVES / [node.SlaveInfo@1f5b0afd] to /128.2.246.37:60462
Send the slave list from the master to the client
Receiving NEW_JOB / mapred.Job@7e80fa6f from /128.2.246.37:60465
Receives a NEW_JOB message: mapred.Job@7e80fa6f
submitting a job from the master to mapper nodes!
Start the map job!
Job Mapper class is WordCountMapper
file name from jobfile.txt
file name from dfsfile.txt_blk0
file.txt_blk0 is on 0
mapper class in submit is WordCountMapper
Sending NEW_MAPPER / mapred.Task@348bdcd2 to cyh100-b4.cyert.andrew.cmu.edu/128.2.246.36:11000


Receiving MAPPER_DONE / mapred.Task@6262937c from /128.2.246.36:59679
Receives a MAPPER_DONE message: mapred.Task@6262937c
All the mapper task of wordcount are done!
Start the reduce job!
Sending NEW_REDUCER / mapred.Task@35c0e45a to cyh100-b4.cyert.andrew.cmu.edu/128.2.246.36:11000
finish submitReduceJob!
filename: file.txt_blk0 0
Sending KEEP_ALIVE / to cyh100-b4.cyert.andrew.cmu.edu/128.2.246.36:11001
```

## Slave side:

---

```
Sending FILE_PUT_REQ_TO_MASTER / communication.WriteFileMsg@5ca46701 to /128.2.246.37:15640
Receiving AVAIL_SLAVES / [node.SlaveInfo@5ca352a5] from /128.2.246.37:15640
the client receives a message from the master
the client will connect:cyh100-b4.cyert.andrew.cmu.edu/128.2.246.36 11000
send the msg: I will upload a file to the slave
Sending FILE_PUT_REQ_TO_SLAVE / I will upload a file to the slave to cyh100-b4.cyert.andrew.cmu.edu/128.2.246.36:11000
client send the msg to slave
Receiving PORT / 9002 from cyh100-b4.cyert.andrew.cmu.edu/128.2.246.36:11000
the client receives a message from the slave
client connect to slave via assigned port 9002
Sending FILE_PUT_START_TO_SLAVE / file.txt_blk0 to cyh100-b4.cyert.andrew.cmu.edu/128.2.246.36:9002
Uploaded file.txt_blk1
File Uploading...
Upload succeed!file.txt_blk0
submitting ...WordCountMapper
Sending NEW_JOB / mapred.Job@43b09468 to /128.2.246.37:15640
listening... 15641
Receiving JOB_COMP / [node.FileInfo@3a21b220] from /128.2.246.37:60540
1_1_reduceResult0 is on slave id: 0
Job wordcountcompleted sucessfully!
```

## The files on a Slave Node:

 0\_0\_MapResult0  
0\_0\_Reduce  
0\_1\_reduceResult0  
> file.txt  
file.txt\_blk0

**If you want to run two or more Jobs concurrently:**

1. You need one specific Configuration file for every Job, and make sure the ClientMainPort in every Configuration file is different from each other.  
For instance, put "ConfigFile1.txt" and "ConfigFile2.txt" under src directory
2. Run MasterMain.java with configuration: src/ConfigFile1.txt
3. Run SlaveMain.java on Slave Nodes with the configuration: src/ConfigFile1.txt
4. Run the test example classes with the configurations on the client side:

Run "WordCountExample" with the configuration:

src/ConfigFile1.txt    inputFileName1    outputFileName1

Run "NGramGenerationExample" with the configuration:

src/ConfigFile2.txt    inputFileName2    outputFileName2

You can see the "Job wordcount completed successfully!" and "Job ngramgeneration completed successfully" on the client side.

- \* We let the Reducer class sleep for a little while to see clearly the tasks are running concurrently.

**Test recovery:**

1. Modify the "ConfigFile.txt"
  2. Run the MasterMain.java on the Master Node with configuration: src/ConfigFile.txt
  3. Run the SlaveMain.java on the Slave Nodes with configuration: src/ConfigFile.txt
  4. Run the "WordCountExample.java" on the Client side with configuration:  
src/ConfigFile.txt    inputFileName    outputFileName
  5. During a Slave node is running a task, terminate it
  6. You can see the job is restarted
  7. The Client receives the job result files
- \* Limitation: recovery can only be done when there are two more slaves.

## Part 6. Document for Application Programmers

To implement MapReduce program on our MapReduce facility is really simple. You need to implement the Mapper, Reducer and Main class.

For your mapper class, you need to extend the Mapper class.

```
public class WordCountMapper extends Mapper{}
```

Then you need to overwrite the map function with your own one, for example,

```
public void map(LongWritable key, Text value, Context context){}
```

The output key-value pairs should be passed-into the Context format.

The way to implement your own reducer is almost the same as implementing the mapper class.

```
public class WordCountReducer extends Reducer {  
    public void reduce(Text key, Iterable<FixValue> values, Context  
context) {}  
}
```

You also need a class that contains the main method to generate a job and send it to the Master.

The main method should include the following lines:

```
//generate a new job  
Job job = new Job("jobName");  
  
//set the mapper and reducer class  
job.setMapperClass("MapperClass");  
job.setReducerClass("ReducerClass");  
  
//set the input and output file name  
job.setInputFileName(args[1]);  
job.setOutputFileName(args[2]);  
  
//the number of reducers is set in the configuration file  
job.setReducerTaskSplits(ParseConfig.ReducerTaskSplits);  
  
//connect to the master node  
Socket socket;  
socket = new Socket(ParseConfig.MasterIP, ParseConfig.MasterMainPort);  
  
//call the putFileHandler to upload the input file  
ClientMain.putFileHandler(socket, job.getInputFileName());  
  
//args[0] is the configuration file  
job.waitForCompletion(args[0]);
```