Map- Reduce FrameWork
REPORT – LAB 3
Distributed Systems – 15640
Summer 2014

Team Members

Pranav Saxena

Vaibhav Suresh Kumar

Design and Features

The aim of the project is to implement a Map-Reduce Facility similar to Hadoop which has the capability of dispatching parallel maps and reduces across multiple hosts, as well as recover from worker failure.

Understanding the existing Hadoop Map -Reduce Framework -

For developing a clear understanding of the Map-Reduce framework before we could implement it, we first decided to go through the open source Apache Hadoop map-reduce implementation which can help us take elegant design decisions. This process of brainstorming different ways of building the Map-Reduce framework helped us evaluate various approaches and finally adopt the most feasible approach.

PACKAGES IMPLEMENTED -

We have implemented 4 packages as follows-

1) Client(user program - **WordCount.java, SameSizeWords.java**)

2) Master - Central controller of the map-reduce framework

3) Distributed File System (distributedFS) - An abstraction of the file system on top of andrew file system. This has a main master node called the Name Node

4) Generics -Consists of different interfaces, abstraction of a fake distributed file, and abstraction of a chunk of a file transferred from one machine to another.

5) Worker - Consists of the worker machines which get registered to the master node as soon as they are launched by sending heartbeat.

OVERVIEW & KEY TERMS

1) **Master Node** (**Map-Reduce**) & **Name Node (DFS)**
2) **Worker Machine** (Slave Mode)
3) **Communication** Mechanism between any two machines - **JAVA RMI** (Remote Method Invocation)
4) **Heartbeat** between Worker machines To Master Node and Name Node to Master Node via Sockets
5) **Parallelism**
Multi-Threading approach - 4 mappers per machine and 1 reducer per machine with each mapper being run as individual thread.
6) **Chunks** - Blocks of File split on the basis of number of lines

Another degree of parallelism in transferring chunks of file from one machine to group of machines in multi-threaded way with each thread making a RMI call to the destination worker machine.

c) **NAME NODE as a separate process** - The Name Node runs on a separate process and sends heartbeat to the Master Node of the map reduce framework. If the Name Node is not available due to network latency or any other failure, the Master Node will know about it and inform the worker about it. The Name Node keeps on sending the map information about the chunks and the machine holding the chunks to the Master. Further, if a request to process

KEY DESIGN DECISIONS & ARCHITECTURE

1) Distributed File System - We decided to have a separate master node for our own distributed file system called the Name Node. This is more or less on the standard HADOOP map reduce framework (HDFS) where the Name Node is responsible for splitting a file into chunks, transferring the chunks from one worker machine to another or group of other worker machines i.e creating replicas of chunks for a more robust system. The key features of our DFS are -

a) **The Chunk Size** & **the Splitting Strategy**- We define a custom split strategy where the split of a input file happens on per line basis in order to create the chunk. This is customizable and can be increased or decreased depending upon the size of the input data being used. We tested our code by using the input data of magnitude 300 MB where each chunk was made to be of the size of 25000 lines , resulting in approximately 16 chunks. Thus, these chunks are not stores in one place but distributed throughout the layer of abstraction i.e the file system we create on top of the underlying file system provided by the cluster of machines we run our program upon. Further, if we try to run the job again on the same input file, the name node will return that this file has already been split and launch the mappers straight away.

b) **Chunk Transfer on Demand** - Each slave node needs to access chuks stored locally on their own system within the distributed file system accessible to them. But sometimes, the worker machine might need a chunk from another worker machine to run map or reduce jobs in parallel as they might have capacity to run more mappers or reducers. In this case, in order to read the contents of other chunks, the worker machine makes a RMI call to the NameNode and asks for the host addresses and ID of another chunk. The Name Node maintains a concurrent hash map which can return the ID of the chunk and IDs of worker machines that contain a given chunk, and then the worker machine can make another RMI call to fetch that chunk from one of the other worker machines.

c) **Abstraction of the Original File in terms of a Fake Distributed File -** To represent the abstraction of a distributed file, we define an object which essentially represents a chunk and encapsulates its properties. Example - Which all machine are holding this chunk, what is its ID, what is its fully qualified path if it needs to be be accessed, how many replicas of this particular chunk exist across the entire distributed system.

d) **Concurrent Hash Maps of Chunks to Array List of Slave Machines, File to Array List of objects of fake distributed file Chunks** - Name Node maintains two concurrent hash maps which contains mapping of each chunk being formed and which slave machine (IP address) is holding it in the distributed file system. Further, to improve algorithmic performance, we maintain another hash map where we can get the list of chunks in O(1) time by directly searching the table on the basis of the filename as the key.

2) MAP- REDUCE FRAMEWORK

Our Map-Reduce framework consists of the central Master Node and a cluster of worker machines connected over network.

a) **MASTER NODE** - The master node of our Map-Reduce framework serves as the central point of the distributed system. It is responsible for tracking each job, scheduling the jobs, tracking its status and forwarding the responsibility to the next available nodes in case a running node fails. It communicates with the Name-Node of the distributed file system and each worker machine. The Master Node is the first thing which needs to be started to bring the system up and running. As soon as the worker machines are started, they send a heartbeat signal to the Master Node and in return each of them gets an ID from the Master Node. HeartBeat signals are sent to the master node every 5 seconds which helps the Master Node to monitor which worker machines are up and running and thus take timely action if a fault occurs thus making the system fault tolerant as well.

b) **WORKER MACHINE** - The worker machine is responsible for running the map-reduce jobs locally. The worker machine runs a task tracker which communicates with the master and updates it with the status of the running mappers and reduce jobs. The worker machine maintains its local map of mapper and reducers, which chunk ID they are acting on, what is its status - “Running, Completed, Available”. If the status is ”Available” or “Completed”, the mapper can take up next chunk and starts its task on it. Once a mapper, completes a task, its status is updated to “Completed” in the local map which is sent through the heartbeat to the Master Node to update its global map of all the mappers & reducers running on all the machines.

c) **Mappers & Reducers as Independent Threads -** Mappers and Reducers run as threads within a single worker machine which is a separate process. Currently, we support running 4 mappers in parallel per machine and 1 per machine and a total of 2 reducers. Thus, as far as the availability of the system is concerned, it is highly available unless the worker node itself goes down as independent threads within a process cannot be killed individually. If the entire worker machine goes down, the master node stops receiving heartbeat from the worker and allocates the same job(initially assigned to the mappers in the worker machine) to the next available worker machine.

d) **REPLICATION OF CHUNKS -** We have a partitioning policy once a file is split into chunks. Depending on the number of worker machines in the cluster, we find the the ratio of total number of chunks divided by total number of machines in the system. Assuming the total number of chunks will be far greater than the number of machines, we transfer the group of chunks to each worker machine in the system in a round robin manner.

e) **Multi-Threaded way of transferring chunks across machines** - When we need to transfer the chunks from one machine to a group of machines, instead of transferring them one machine by one machine, we follow a multi-threaded approach where each thread is responsible for interacting with one machine and internally makes a RMI call to transfer the chunk of data. This not only promotes parallelism but also improves the performance of the system by a huge amount. Imagine if the chunk size is large enough and we follow a serialized approach of transferring the chunks, how bad the performance of the system would be.

f) **INTERMEDIATE FILES GENERATED ON THE DFS** - Once the user submits a map - reduce job by placing his file in the Input folder in the DFS, the file is divided into chunks and spread across different machines. The mappers start their work on the different chunks and output intermediate files in each machine's respective "Intermediate folder" within the DFS folder. We

f) **PARTITIONER & THE SHUFFLE AND SORT PHASE** - The mappers are internally responsible for hashing the keys and generate a hash value based on the number of reducers set in the configuration file. This allows to redirect the keys to the appropriate reducer machine by dividing the intermediate file produced by the mapper into two parts, if there were 2 reducers in the system. These intermediate files are sorted locally and is then picked up by the reducers for starting the reducing phase. By following this approach, it allows us to get rid of multiple reduce phases to get one final output as the hash function guarantees that the keys with same hash value are redirected to the same reducer.

g) **JOB SCHEDULER (JOB TRACKER)** - The Master Node is responsible for scheduling all the map-reduce jobs and maintains a concurrent hash map to schedule the jobs in a round robin way. All the mappers are initialized to AVAILABLE mode and are assigned the task (chunk of file to act upon) in a round robin way. Eventually all the mappers are assigned responsibilities and run in parallel on all the machines where their status is updated to “RUNNING”. The only issues with this approach is that if the chunk size assigned to the mapper is not very large while some of the mappers have a larger chunk size to act upon, the mapper with smaller chunk size would finish its job really fast and it can sometimes lead to a situation where the second mapper is not assigned any responsibility and hence it just starves. But this is not guaranteed to happen, if we are acting on a significant amount of big data. We experimented with 100 kb file, 300 Mb file and could observe the above mentioned scenarios carefully. Further, the system supports executing parallel jobs and queues up the tasks at the mappers, if the reducers are already busy with their tasks for the previously executing tasks. All these tasks are coordinated by the master node which communicates with the task tracker on every worker machine. Further the reduce tasks are not scheduled unless all the map tasks associated with that map-reduce request have been completed. The only drawback with this approach is that we have to iterate over the map again and again and check which mappers are free to be assigned any tasks. We thought about improving this design by moving from a round-robin approach to a priority-queue (MAX heap) scheduling approach where each node at the top of the heap is allowed to run the task where the node at the top of the heap would have highest number of available free mappers to execute a task. But due to time constraints we couldn’t implements this approach completely and hence resorted to a simple and naive round-robin scheduling of tasks.

h) Handling **RACE Conditions checks** at the Master and Worker Level Nodes - Due to client-server network latency, sometimes the maps updated at the master level take some time to be updated by the data sent by the heartbeat from the worker machines. This gap can allow additional threads to be spawned unless blocked by the worker machine. This level of double locking at the master and the worker level ensures the execution of already spawned threads and that the number of the mappers at any given point of time remains fixed. This helps in keeping track of all running jobs on a worker machine and renders the system more predictable.

i) **Fault Tolerance** - We tried to develop as much fault tolerance within the system but the time constraints did not allow us to build a fully fault tolerant system.
a) HeartBeat message from all the workers to the master keeps the master informed about each worker machine’s status. If one slave machine needs some chunk of file from another slave machine, it can query the master and check if it can fetch the chunk from the worker machine. In case that worker machine stops sending heartbeat messages, then the master

would query based on the chunk ID and see the next available machine which has that chunk available and relay back the message to the slave in which slave can make a RMI call directly to that slave machine and get that chunk of data.

j) **Communication Mechanism via JAVA'S REMOTE METHOD INVOCATION and HEARTBEAT using Sockets -** Java's RMI framework made the communication mechanism between master <-> workers, workers < ->Name Node, Name Node< - > MasterNode really easy to send any information. Invoking a function remotely using standard JAVA RMI framework was the key strategy in our framework implementation.

h) **Multiple Output/Intermediate Files with Multiple Reducers** - We create a folder structure hierarchy within our DFS to store the output at various stages of the map-reduce program. You can just navigate to the DFS folder and check for any file you might want to see. We tested our code with 1 and 2 reducers in the system. Ideally we would suggest to set 2 reducers in the configuration.

II. Describe the portions of the design that are correctly implemented, that have bugs, and that remain unimplemented.

Portions Correctly Implemented

1) Distributed File system
2) Map-Reduce Basic framework
3) Partitioner (Shuffle & Sort)
4) Parallel execution of Mappers as threads
5) Parallel execution of Reducers as threads
6) Creation of intermediate files and writing and sorting data
7) Basic Fault tolerance as explained earlier
8) Job scheduler, tracker , network communication over RMI
9) Configuration in the form of User Console instead of a static configuration file
10) Error handling and basic validation at the user level, exception handling on few boundary cases which throw NULL pointer exceptions
11) Multi-Threading approach of transferring files using byte arrays

Portion Not Implemented

1) Management tool on the user side to monitor which jobs are queued up/ running. Our framework allows one to monitor all the information related to the jobs on the master node's console but we didn't get enough time to capture all this information and bring it out to the end user in the user console.

2) Elegant Fault Tolerance - If the Name Node goes down, then the information of the chunks will be lost and we won't be able to cater to any chunk transfer requests. We have fault tolerance at the job scheduling and chunk transfer level only.

III. Tell us how to cleanly build, deploy, and run your project

NOTE - The code has been tested on ASF machines and all the screenshots shown below would be that of our ASF machine.

First Login (SSH) to your asf(andrew) machines and to the following-

1) Navigation and Compilation of the Source Code

Compile all the packages inside the src folder

javac worker/*.java distributedFS/*.java master/*.java generics/*.java client/*.java

2) Start the Master , then the worker machines , then the Name Node and then finally User Job Console.

NOTE - The ordering is important here.

To start the master -

java master/Master

To start any worker machine -

java worker/Worker < Master Node IP Address>

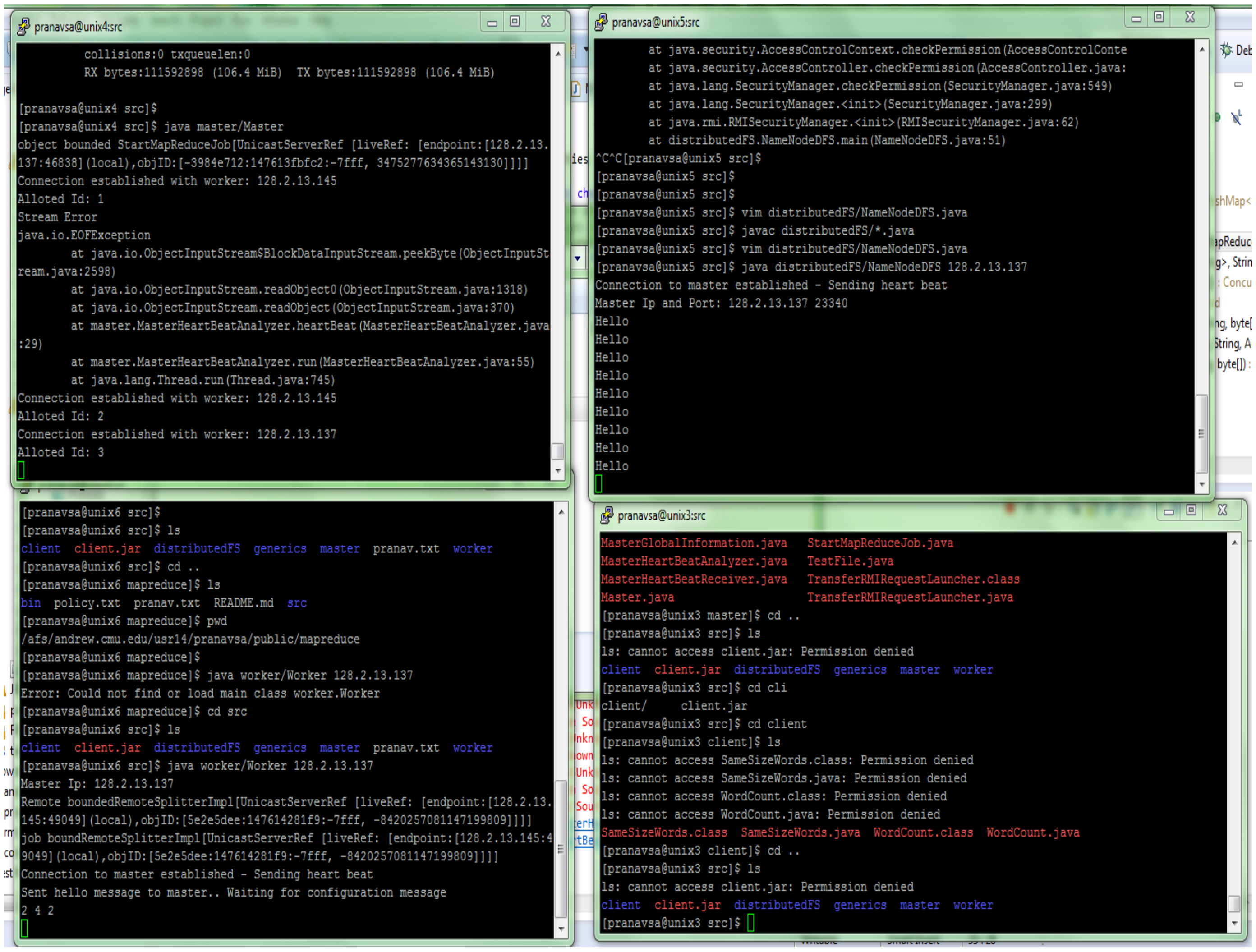
To start NameNode

java distributedFS/NameNodeDFS <Master Node IP Address>

To start User Job Launch Console

java worker/UserJobLaunchConsole <Master Node IP Address>

AFS SNAPSHOT



pranavsa@unix4:src

```
        collisions:0 txqueuelen:0
        RX bytes:111592898 (106.4 MiB)  TX bytes:111592898 (106.4 MiB)

[pranavsa@unix4 src]$
[pranavsa@unix4 src]$ java master/Master
object bounded StartMapReduceJob[UnicastServerRef [liveRef: [endpoint:[128.2.13.
137:46838](local),objID:[-3984e712:147613fbfc2:-7fff, 3475277634365143130]]]]
Connection established with worker: 128.2.13.145
Alloted Id: 1
Stream Error
java.io.EOFException
        at java.io.ObjectInputStream$BlockDataInputStream.peekByte(ObjectInputSt
ream.java:2598)
        at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1318)
        at java.io.ObjectInputStream.readObject(ObjectInputStream.java:370)
        at master.MasterHeartBeatAnalyzer.heartBeat(MasterHeartBeatAnalyzer.java
:29)
        at master.MasterHeartBeatAnalyzer.run(MasterHeartBeatAnalyzer.java:55)
        at java.lang.Thread.run(Thread.java:745)
Connection established with worker: 128.2.13.145
Alloted Id: 2
Connection established with worker: 128.2.13.137
Alloted Id: 3
```

pranavsa@unix5:src

```
        at java.security.AccessControlContext.checkPermission(AccessControlConte
        at java.security.AccessController.checkPermission(AccessController.java:
        at java.lang.SecurityManager.checkPermission(SecurityManager.java:549)
        at java.lang.SecurityManager.<init>(SecurityManager.java:299)
        at java.rmi.RMISecurityManager.<init>(RMISecurityManager.java:62)
        at distributedFS.NameNodeDFS.main(NameNodeDFS.java:51)
^C^C[pranavsa@unix5 src]$
[pranavsa@unix5 src]$
[pranavsa@unix5 src]$
[pranavsa@unix5 src]$ vim distributedFS/NameNodeDFS.java
[pranavsa@unix5 src]$ javac distributedFS/*.java
[pranavsa@unix5 src]$ vim distributedFS/NameNodeDFS.java
[pranavsa@unix5 src]$ java distributedFS/NameNodeDFS 128.2.13.137
Connection to master established - Sending heart beat
Master Ip and Port: 128.2.13.137 23340
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```

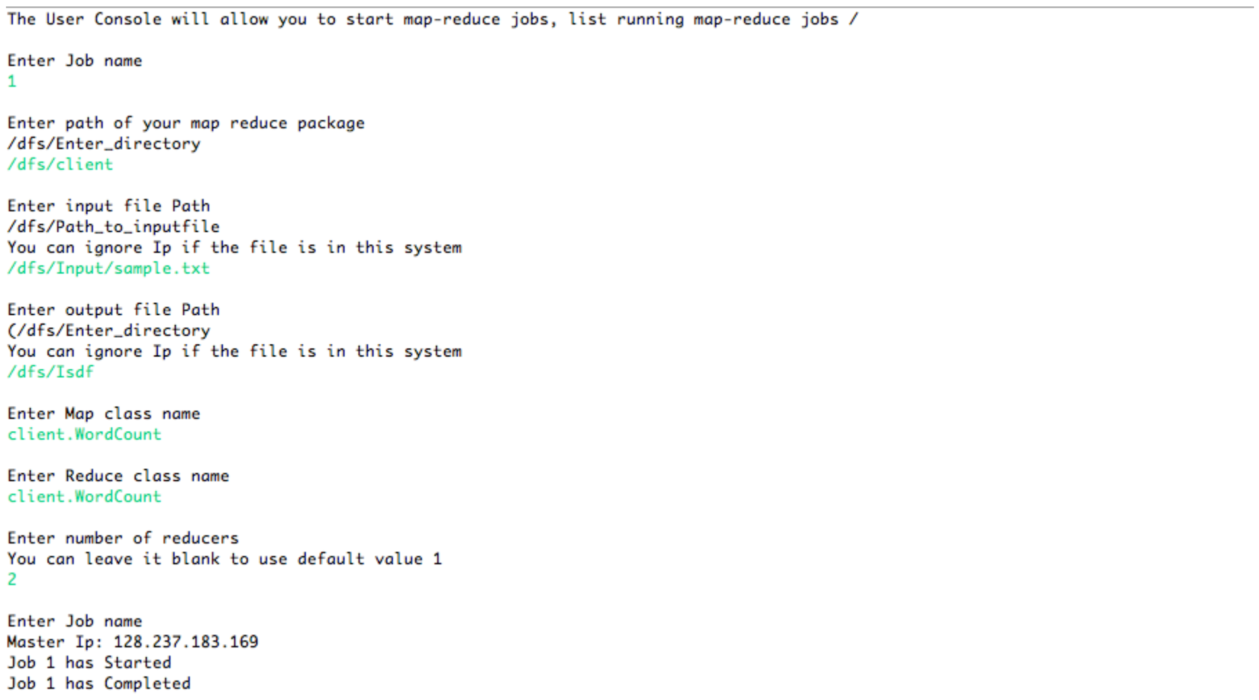
```
[pranavsa@unix6 src]$
[pranavsa@unix6 src]$ ls
client  client.jar  distributedFS  generics  master  pranav.txt  worker
[pranavsa@unix6 src]$ cd ..
[pranavsa@unix6 mapreduce]$ ls
bin  policy.txt  pranav.txt  README.md  src
[pranavsa@unix6 mapreduce]$ pwd
/afs/andrew.cmu.edu/usr14/pranavsa/public/mapreduce
[pranavsa@unix6 mapreduce]$
[pranavsa@unix6 mapreduce]$ java worker/Worker 128.2.13.137
Error: Could not find or load main class worker.Worker
[pranavsa@unix6 mapreduce]$ cd src
[pranavsa@unix6 src]$ ls
client  client.jar  distributedFS  generics  master  pranav.txt  worker
[pranavsa@unix6 src]$ java worker/Worker 128.2.13.137
Master Ip: 128.2.13.137
Remote boundedRemoteSplitterImpl[UnicastServerRef [liveRef: [endpoint:[128.2.13.
145:49049](local),objID:[5e2e5dee:147614281f9:-7fff, -8420257081147199809]]]]
job boundRemoteSplitterImpl[UnicastServerRef [liveRef: [endpoint:[128.2.13.145:4
9049](local),objID:[5e2e5dee:147614281f9:-7fff, -8420257081147199809]]]]
Connection to master established - Sending heart beat
Sent hello message to master.. Waiting for configuration message
2 4 2
```

pranavsa@unix3:src

```
MasterGlobalInformation.java   StartMapReduceJob.java
MasterHeartBeatAnalyzer.java   TestFile.java
MasterHeartBeatReceiver.java   TransferRMIRequestLauncher.class
Master.java                    TransferRMIRequestLauncher.java
[pranavsa@unix3 master]$ cd ..
[pranavsa@unix3 src]$ ls
ls: cannot access client.jar: Permission denied
client  client.jar  distributedFS  generics  master  worker
[pranavsa@unix3 src]$ cd cli
client/     client.jar
[pranavsa@unix3 src]$ cd client
[pranavsa@unix3 client]$ ls
ls: cannot access SameSizeWords.class: Permission denied
ls: cannot access SameSizeWords.java: Permission denied
ls: cannot access WordCount.class: Permission denied
ls: cannot access WordCount.java: Permission denied
SameSizeWords.class  SameSizeWords.java  WordCount.class  WordCount.java
[pranavsa@unix3 client]$ cd ..
[pranavsa@unix3 src]$ ls
ls: cannot access client.jar: Permission denied
client  client.jar  distributedFS  generics  master  worker
[pranavsa@unix3 src]$
```

STEP 2 -
Now you just need to enter the details in the UserConsole

Check the screenshot on the next page



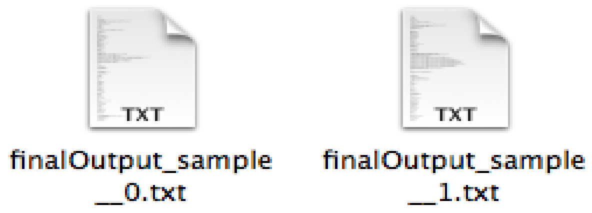
```
The User Console will allow you to start map-reduce jobs, list running map-reduce jobs /

Enter Job name
1

Enter path of your map reduce package
/dfs/Enter_directory
/dfs/client

Enter input file Path
/dfs/Path_to_inputfile
You can ignore Ip if the file is in this system
/dfs/Input/sample.txt

Enter output file Path
(/dfs/Enter_directory
You can ignore Ip if the file is in this system
/dfs/Isdf

Enter Map class name
client.WordCount

Enter Reduce class name
client.WordCount

Enter number of reducers
You can leave it blank to use default value 1
2

Enter Job name
Master Ip: 128.237.183.169
Job 1 has Started
Job 1 has Completed
```

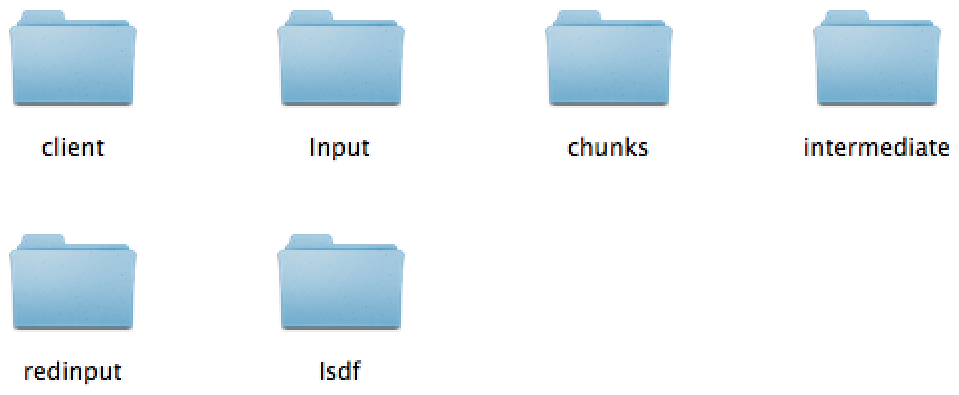
NOTE - To run the map reduce job, you need to put the file into the input folder within the DFS directory. Here we have sample.txt within the input folder.

STEP 3

You should see the output reducer files created in the Output directory you specified in the User Job Console program.

You can also notice the intermediate directories created in the DFS directory in the screenshot shown below . The final output directory contains the two output files as we ran 2 reducers and it generated two output files which have distinct keys.





IV. Highlight any dependencies and software or system requirements.

1)There are no specific dependencies but to compile the java code, you would need **jdk 1.7 .**

2) **Firewall Issues** - Further, please make sure that there are no firewall issues with the network on which you are running the system. This can possibly lead to failure of socket connection being established.

3) **PORT NUMBERS FOR RUNNING RMI REGISTRIES** - We have used the following ports in our code. SO please make sure, you have these port numbers opened -

1) 9876

2) 23333

3) 23390

4) 23340

5) **RMI POLICY.TXT FILE** **- For ensuring successful network communication over RMI, we need to have the policy.txt file stored on each node which grants permission to all users for enabling RMI server connection.**

V) **HOW TO USE OUR FRAMEWORK WITH TWO EXAMPLES -**

In order to use our framework, all one needs to do is extend the MapReduce class and provide the functionalities of map() and reduce() functions. Ofcourse, one needs to compile the code and supply it's .class files in the client package which has been explained in the instructions earlier. We have provided two examples -

1) WordCount.java - Counts the frequency of each word in the file and gives a sorted output
2) SameSizeWords.java - Outputs a set of words for a given length

Below is the implementation of WordCount.java

/**

EXAMPLE 1 - WORDCOUNT.JAVA

```
package client;

import java.util.ArrayList;

import worker.MapReduce;    // Import our framework's MapReduce class


public class WordCount extends MapReduce {   // Extend the MapReduce class


private static final long serialVersionUID = 1L;

public void map(String key, String value, MapReduce mapper)  {   // Define 3 parameters
        String frequency = "1";                        // key , value and an object of MapReduce Class

// Logic of the map() program
        String[] words = value.split("\\s+");
        for (String k : words) {
            k = k.trim();
            if (k.length() > 0) {
                mapper.writeToFile(k, frequency);
            }
        }
```

```
    }

public void reduce(String key, ArrayList<String> values, MapReduce reducerResult) {
        Integer sum = 0;                                   // Define  3 parameters
                                        // Key , ArrayList of values, object of  MapReduce class

// Logic of the reduce()  program
        for (String value : values){
            sum += Integer.parseInt(value);
        }
        reducerResult.writeReducerOutput(key, sum.toString());
     }



}

****************************************************************************************************/
```

EXAMPLE 2 - SAMESIZEWORDS.JAVA

```
package client;

import java.util.ArrayList;

import worker.MapReduce;    // Import our framework's MapReduce class


public class WordCount extends MapReduce {   // Extend the MapReduce class


private static final long serialVersionUID = 1L;

public void map(String key, String value, MapReduce mapper)  {   // Define 3 parameters
// Logic of the map() program

String[] words = value.split("\\s+");
        for (String k : words) {
          k = k.trim();
           if (k.length() > 0) {
              mapper.writeToFile(k.length()+"" , k);
           }
        }

    }

public void reduce(String key, ArrayList<String> values, MapReduce reducerResult) {
                                                // Define  3 parameters
                                               // Key , ArrayList of values, object of  MapReduce class

// Logic of the reduce()  program

         String output= "";
         for(String k:values){
              output = output + k + " , ";
            }
         reducerResult.writeReducerOutput(key, "[" + output + "]");
     }


}
```