MapReduce Facility Report

Chen Sun, Tao Yu {chens1,taoyu}@andrew.cmu.edu

1. Design Overview

There are mainly two parts in MapReduce Facility: Distributed File System and MapReduce Framework.

Through Distributed File System, users could upload files, mapper and reduce class files. Files would be divided into pieces of blocks with replication numbers of user's choice. Through MapReduce framework, users could override the mapper and reducer functions, submit mapreduce job(s) to the mapreduce framework, monitor Job status, and terminate running job as they want.

To enable this working we design the system as follows:

1. Distributed File System, mainly including Data Node and Name Node. Also, to enable file uploading and file downloading, we provide File/Class Uploader and Downloader. We provide DFSStatus to check status and DFSTerminator to terminate the distributed file system.
2. For mapreduce framework, we provide JobTracker to track all the jobs submitted and TaskTracker to track all the tasks that are part of the jobs. Also, we provide Message to pass command between nodes. Finally, we provide basic Mapper and Reducer class to be extended as the base for MapReduce framework.

All details mentioned above will be covered below.

2. Distributed File System

The distributed file system mainly contains two components: Name Node and Data Node.

2.1 Name Node

Name Node is the central component of distributed file system. It accepts connection from data node, register data node stub here. Also, it manages the distributed file system, which has an overview of the whole file system, including the mappings from files to their corresponding data blocks, and data blocks to their replicational data nodes. Reversely, it also records the mapping between data nodes and the blocks each of them stores.

The MapReduce framework uploads and downloads file via Name Nodes. In this case, they don't need to know about the number of data nodes in the Distributed File System. And there is only one single Name Node in the Distributed File System. Once it fails, there are no way that the system could continue working. This is to say, the whole system fails at this point.

However, if the system terminates normally, it should record the mapping information at that name node. If the name node restarts, it is able to recover the mapping information of the file system. Since we assume the bootstrapping process happens at AFS, so in theory the name node can be restarted at any node again. After the restarting of the name nodes, all the data nodes should re-connect to the data nodes in the original order they registered before. Otherwise, the mapping information would be wrong.

2.1.1 Register

Data node registering process takes the data node stub as input. The Name node tries to allocate a global data node id to the new node. In the mean time, it tries to put the data node stub in the map from data node id to data note stub itself. A data node meta data that records data node id, the state of the data node, the block IDs currently available on the data node is also generated. One single data meta is put into both a map from data node id to meta and a heap for the later choice of data node allocation. The heap sorts the data node based on the length of the block id list so that we can allocate the least loaded data node when trying to uploading files to data node.

2.1.2 Create New File

Since both class files and normal files are treated as blocks, name node and data node does not try to distinguish between them at the storage side. Thus, when client side needs to upload files to the dfs, it calls to create file with the number of replications they want to have. DFS only allows the replicas to be between 1 and the number of replication factors when the node starts. Any value beyond the range would be regarded as the default replication factors. Based on the file name and replication number, the file system will return a FileInfo object to the client side, for the client to fill in the block ids. The file system does not allow duplication of files. That is, if a file name already exists in the DFS, the local uploading will do nothing. The reason to do so is that our design of DFS does not allow modification or deletion of files. If we want to override the files already exist. We have to either modify original blocks or delete them and add totally new blocks to the file system. Potentially, it would cause the system’s information inconsistent during this process, if other jobs are trying to read the files while the modification is taking place.

2.1.3 Allocate Block Id

For the next step, the client side tries to ask for allocation of a block id. The block IDs are global for the name node, so all the data blocks can be saved directly with <block ID> as the file name on the local storage on each data node. The blockId acquired will be used to store exact piece of data among different data nodes. Also, at the client side, we update the FileInfo to make sure it just gets a new block id.

2.1.4 Determine Data Node

After we get the block id, we have to figure out which data node(s) to store the data. For data node allocation, we use the strategies as follows: 1) the data node should not already contain the block, otherwise the replication makes no sense; 2) the data node should not be dead; 3) the data node should be the node with lightest load meeting the above two conditions. If we could not provide such data node, then the creation fails. We use a heap to choose the data node. Before we return the data node, we also try to heart beat the data node to make sure it is live. The data node is acquired with block id as input. After the data node is returned, we put the block id to the node's block id list of the meta data.

2.1.5 Commit File Change

If we managed to put all the file blocks onto the distributed file system, we commit the changes by updating the file block information to the name node server side so that name node now knows a file, from its name to data block ids. This finishes the process of file uploading.

2.1.6 Uploader

We implement the File Uploader and Class Uploader a little differently. Since the project assumes that file is split by lines, so we read the file as lines of strings to partition the files. However, the class files are in binary format. Because most class files are quite small, we simply put each class file as a single block. This could be problematic when the class file is large, but for the usage for this project, it provides enough simplification.

2.1.7 Downloader

When downloading the files, no matter the format of the original text file or class file, it could be regarded as binary as we don't need to do the splitting. Thus, we only provide one downloader. The Downloader tries to fetch a stub of the name node. Firstly, it tries to get the block Ids from mapping between file name and file info. Secondly, it retrieves the list of each block with a list of data node ids that store the block. It will return a map of <Block Id, List<Node Id>>. We implemented this using tree map, so the traversal of block Ids are ordered. As we allocate data block id one by one in ascending order, thus traversal through the tree would give the right order of the data blocks. Combining them together would finally give the file in its right order.

As we traverse the map, we try to get each data node id for each block id, and tries to get the data from the data node. Since we only need to combine the files rather than spliting them, binary I/O is enough for all the types of files. If some data node is dead, we go on to the next data node id until we have traversed all the possible data node ids. If still we could not have data, we declare the downloading as failed as we could not fetch the single data block here. And there is nothing we can do to recover that.

2.2 Data Node

Data node implementation is rather simple comparing with name node since its primary role is to provide data.

2.2.1 Register

To register itself to the name node, it should firstly locate the registry of the name node with previously known host and port in conf file , and call data node's register method to put its own stub into the name node. After the registering process, it now can be used to put data.

2.2.2 Put Data Block

As the data node has made sure that there are no duplicate data blocks on the same data node, all it needs to do is to create a data block on the local file system, which has a name of the data block id and put data via I/O. We provide two methods to create data blocks. One is to provide the content with as String, and the other is with byte array directly. String is used for generating files and byte array is used for putting binary class.

2.2.3 Read Data Block

Simply we combine the path of the data directory and data node id to be the file name, and tried to open the file and read the data in binary format. In this case, we can directly put the data block as part of the file to be downloaded.

2.2.4 Other

We also provide heartbeat, which actually does nothing but throw a RemoteException when connection loses.

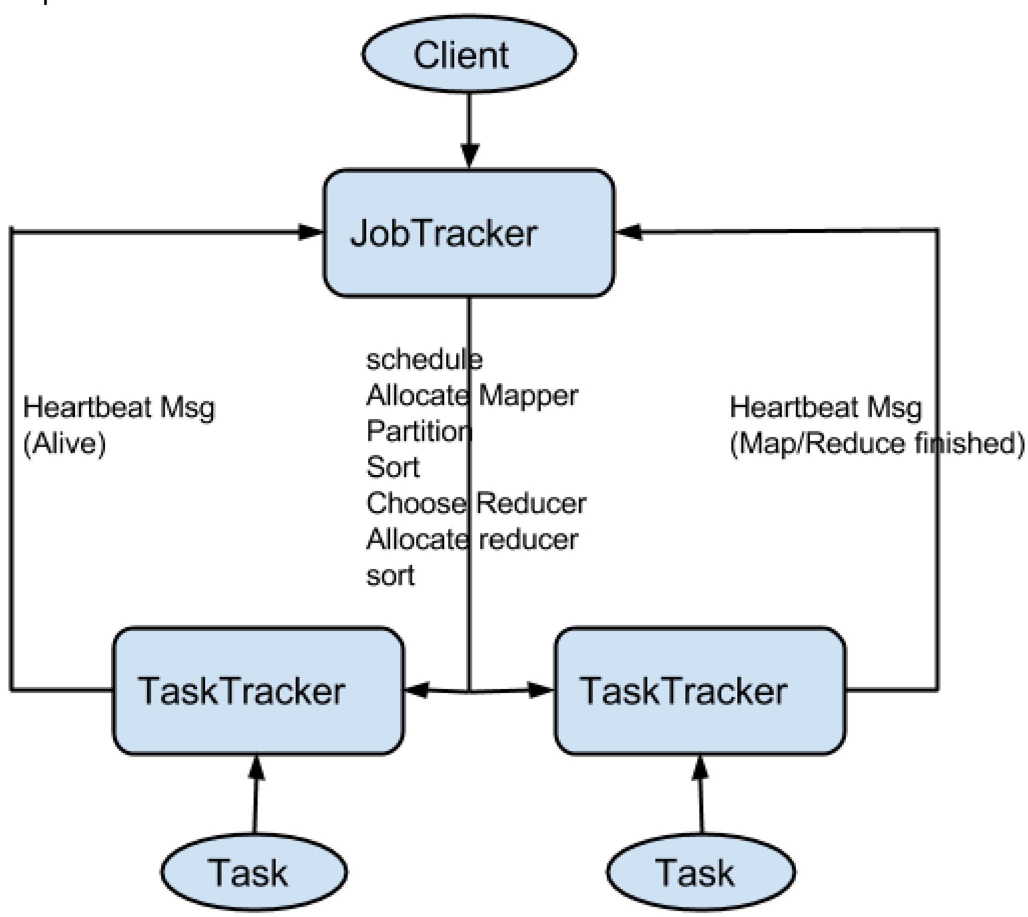
Name Node could call Data Node's terminator to be part of the terminating process of Distributed File System.

3. MapReduce Framework

The MapReduce framework consists of a single master JobTracker and one slave TaskTracker per cluster-node. The master is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master.

3.1 Interaction between JobTracker and TaskTracker

MapReduce Framework uses RMI for communication between JobTracker and TaskTracker



3.2 MapReduce Framework

3.2.1 Mapper Phrase

After the client upload the MapReduce job class files and input file, DFS would split the file into different blocks with replications of user's choice. Then the JobTracker assigns mapper tasks to empty slots of TaskTrackers based on the principle of one mapper per block as well as locality. TaskTracker then wraps the mapper task into Task instance, starting a new thread to execute task and periodically check if the thread has finished. TaskTracker reports to JobTracker once any thread is completed through heartbeat message. For each Job, when JobTracker received all mappers finished message from TaskTrackers, it will step into the partition part.

Sort and Partition

After the map step finished, MapReduce Framework splits and combines the mapper result files into partitions based on the number of reducers. In order to save memory, mapper writes each line into a fixed size of ArrayList<RecordLine>, when the fixed size buffer is filled up, it dumps the buffer into temp file. After all the mappers have finished, it would split the temp files into different partitioned files(For different reducers) using external n-way merge.

3.2.2 Shuffle Phrase

After Mapper Phrase completes, MapReduce Framework would first allocate TaskTrackers to do the reducer based on the number of available slots. Then JobTracker would move mapper partitioned files into corresponding nodes(reducers) based on the hashcode, ensuring that same keys would be processed in the same reducer.

Merge

When shuffling, each reducer node will receive many partitioned mapper files. Before Reducer Phrase, it would combine them together using external n-way merge. Then there is only one sorted file for each reducer node to do the reduce function.

3.2.3 Reduce Phrase

Each reducer node would execute the reducer function based on the combined mapper files and write the output file in to local disk. After all the reducers completes, the JobTracker would upload the reducer output files into specified NameNode path.

3.2.4 Failures Handling

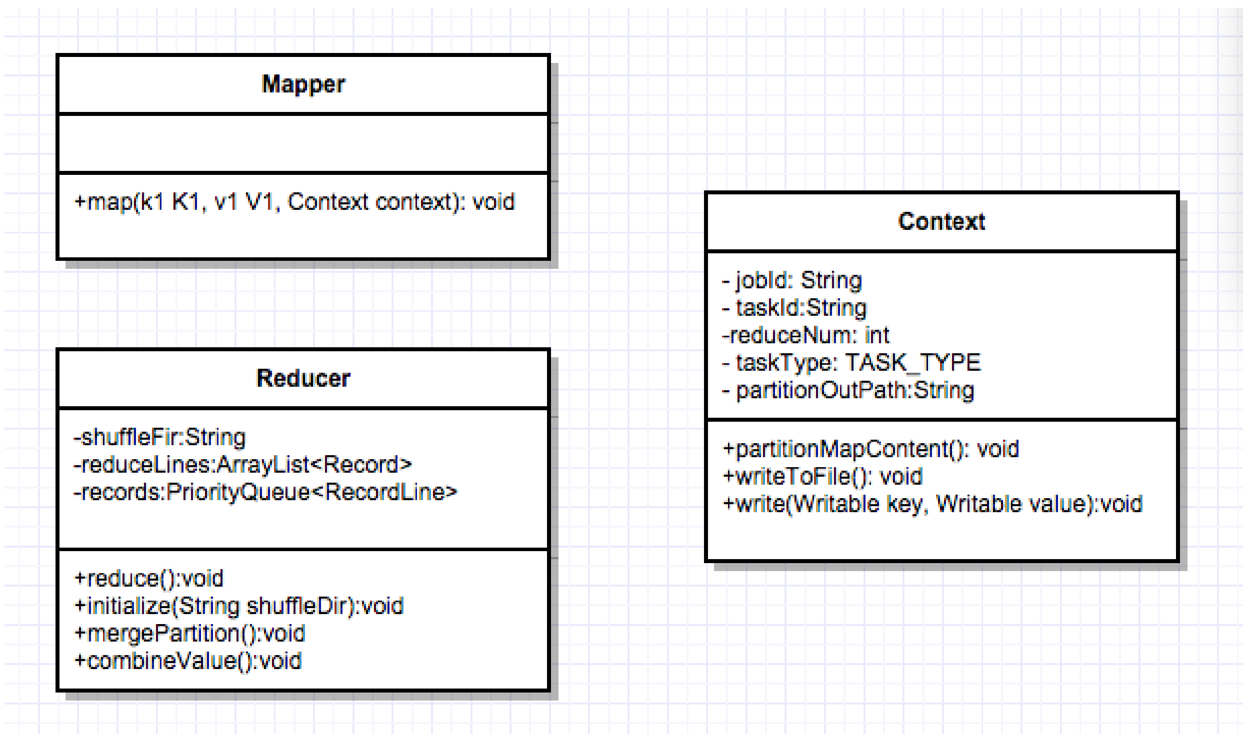
We only deal with TaskTracker failures in our system. TaskTracker periodically send heartbeat message to JobTracker. If a TaskTracker is down, JobTracker will notice this failure and terminate corresponding Jobs whose tasks were running by on the failed TaskTracker. JobTracker would reschedule these Jobs from the start. We chose this design because all mappers and reducers are running in parallel, the difference of time between restarting one mapper and all the mappers is small.

3.2.5 MapReduce Monitor

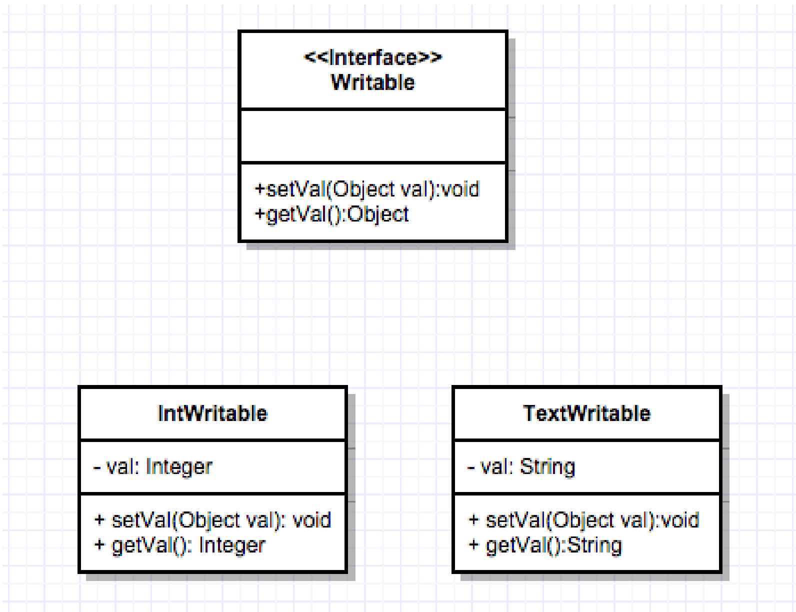
Client could check Job status by using MRMonitor, checking status of specified Job Id or status of all the Jobs.

Client could also kill one specific Job or all the Jobs by using MRMonitor, as well as terminate the MapReduce Framework.

4. MapReduce API Design



Client can write Mapper class and Reducer class which extends Mapper and Reduce Class, overriding the map and reduce function.



K1, V1 can be any kind of IntWritable or TextWritable

5. Implementation Detail

In this section we will talk about the general Java techniques we used in our implementation.

Firstly, we used RMI to do the communication. Both NameNode and JobTacker are registered with the registry running on the same node. In that case, by looking up the registry, we can get these components' stubs there. And call them remotely as they are running locally. We eliminate the usage of messages based on this RMI Server Design.

Also each datanode and tasktracker is also declared as standalone running server, so that they can be called via the stub generated through certain port specified. In that case, we don't need to care about the implementation socket communication at the low level and we can focus more on the functionality.

In terms of the asynchronous task computation in the mapper and reducer task involved, we use Future to accomplish that. Detailed documentation of future can be found here: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html . In our implementation, we use future to submit the task to be run, and check the status regularly to make sure all the mapper works done or all the jobs are done. Here, we start the reducers only after all the mappers are done with their jobs. The Future class also provides us with the ability to check the status of the jobs submitted.

What's more, our file system doesn't have hierarchy in storing the data. Firstly, all the data are stored in data blocks all together. Secondly, even if the input file name contains folder "/" to indicate layers, we simply ignore that and regard the path as part of the file name.

Other implementation details have been covered in the explanation above.

6. Deployment and Test

To build the project, firstly we assume that you have ant installed on your computer. If you don't have Ant, please refer to http://ant.apache.org/bindownload.cgi for installing.

Enter the parent folder, under which you should see a /src folder and a build.xml file and this report, as well as the configuration file conf.

To build, run **ant build**. It will generate the bin folder and jar

After build, you can start all components as following.

Purpose	Command
Start name node	**ant NameNode.**
Start data node (with self port in conf)	**ant DataNode**
Start data node with specific port and folder	**ant MyDataNode** **-Darg0 =<self port>** **-Darg1 =<data folder>**
Start job tracker	**ant JobTracker**
Start a new task tracker	**ant TaskTracker** **-Darg0=<self port>** **-Darg1=<data node id>**
Status of the distributed file system	**ant DFSStatus**
Terminate the file system	**ant DFSTerminator**
Status of the MR Job Trackers, and also terminate the Job trackers.	**ant MRMonitor** (and follow the instructions for more input)
Upload files	**ant FileUploader** -Darg0=<local file path> -Darg1=<replicas> -Darg2=<remote file names>
Upload Class files	**ant ClassUploader** -Darg0=<local file path> -Darg1=<replicas> -Darg2=<remote file names>
Download files	**ant Downloader** -Darg0=<local file path> -Darg1=<remote file path>

Please note that each data node is supposed to run on different hosts, that's why we can assume that they could share the same port in the conf file. However, if you wish to start several Data Nodes and several Task Trackers on the same host, please be sure to use MyDataNode to specify the port and path of the local storage folder.

Finally, run **ant clean** to clean all the /bin folder and delele .jar file.

To run MapReduce test, run **ant WordCountTest** to test. This test is classical Word Count Program of MapReduce process. Please make sure you have already start the name node, data nodes with total number greater than or equal to the number of replications, JobTracker, TaskTrackers corresponding to the number of data nodes before you run the test.

You should be able to see the status of the file system after the program successfully runs.

Another similar test is Supporters Count, where the input is from SNAP(http://snap.stanford.edu/data/wiki-Vote.html), recording the voters for the wikipedia community about the voters. We calculate the supporters for each voted user in this example.

To run the example, run **ant GraphMining.** Also, make sure you have started the nodes correctly. The input file is **./wiki-Vote.txt** and you can replace it with any <src, dst> format data. To change that, you need to change the <graphIn> parameter in conf file.

Below are the configuration files in the conf file. Modify as you wish.

Description	**Key**	**Default Value**
Name Node's host	NameNode.Host	localhost
Name Node's port for registry	NameNode.RegistryPort	15440
Name Node's port to communication	NameNode.Port	15439
Name Node's path to store the name node's mapping data	NameNode.Image	/tmp/FileImage
Block size for each node	NameNode.BlockSize	20480
Time interval to check status	NameNode.Interval	3600
Number of replications	NameNode.Replication	2
Host for Job Tracker	JobTracker.Host	localhost

Registry Port	JobTracker.RegistryPort	15640
Communication Port for JobTracker	JobTracker.Port	15639
Number of Reducer	JobTracker.Reducer	2
Data Node’s Communication Port	DataNode.Port	15441
Data Node Local Path for Storing the data	DataNode.Dir	/tmp/data/
TaskTracker’s Communication Port	TaskTracker.Port	15641
Input file path for Word Count	WordCountIn	movie_processed.dat
Input file path for GraphMining	GraphIn	wiki-Vote.txt

7. Credit

As we approach this project, we thank the teaching assistant who graded our Project 2 and gave us feedback on deployment and run. Thus, we use ant to build and deploy data for this project. And we improve our implementation by adding more comments and make the style better to understand.

Also, there are many sources available about the map reduce implementation, we wish to thank https://github.com/YaanJian/MapReduceFM for inspriting us the current design. And also https://github.com/bruingao/MapReduce/ for the usage of RMI framework. Other source we refer includes https://github.com/gogol008/MapReduceCodeExamples/ for the Mapreduce example, but we never use any of them.