**Design:**

Our map-Reduce frame work can be split into two parts. One is map-reduce library and the other is DFS.

Map-Reduce:

The main concepts for our map-reduce framework are jobs and tasks.

Job is the unit of one map-reduce job.

Task can be viewed in two ways:

1.The map or reduce task that has all the information about how to handle one portion of the initial input or reducer input( input split and reducer input).

2. The actual task threads running on task-node that do the actual task after they load the class defined by user.

Surround these two core concepts, we develop the system with all the important components.

The main components are described below:

JobConf:

This component is used for parse the configuration file submitted by client.

JobClient:

This is the component that running on user’s terminal, which handled the job submitting and monitoring current job status.

JobTracker:

This is the center of our system. It first built connection with all the task-nodes, and listen for clients to submit jobs(with client configuration file). After it receives the configuration file, it analysed it and wait for the task-tracker to grab it(actually using JobInProgress). It also periodically checked status of each task-nodes to see if they died. JobTracker has a list of jobs(JobInProgress) to run. Different clients can submit job concurrently.

JobInProgress:

It represented one job, it responded to the TaskTracker which ask for map and reduced tasks, and produces said map and reduce tasks for the JobTracker to manage and hand out to the TaskTrackers. It also do sorting for mapper output and merge all the mapper output to one file.

TaskConf:

Describes what a particular task should do (e.g. start and end locations of tasks, which mapper/reducer task to use, etc.)

TaskTracker:

It is the center of a task-node. It asked the jobtracker to give him new task to do, and tell the jobtracker how many task it can run. The number of tasks that can be run concurrently is given by system configuration file. It also monitors and returns the current status of all tasks currently running on this node.

Mapper:

A type of task that takes in fixed-length key-value pairs from the input file and outputs any number of similarly fixed-length key-value pairs for each input record.

MasterNode:

It is actually runs on Master node and we put both NameNode and JobTracker here.

Reducer:

A type of task that takes in a fixed-length key and a list of fixed-length values corresponding to that key, and outputs any number of similarly fixed-length key-value pairs for each input key/value-list (by presumably aggregating the value-list in some meaningful manner).

TaskNode:

Each Task node run a task tracker and several threads of mapper task and reduce task.

DFS:

Our DFS’s main components are: NameNode and DataNodes.

For the most components in DFS:

NameNode:

This is the center of DFS. It handle the actual IO, and remember which file(replica) is located on which data\_node. With the NameNode, map-reduce task can read and write like locally, they doesn’t need to know where to read and write, just hand it to DFS. DFS keep all the track of each datanode and each file generated by mapreduce. It also periodically checked the data\_nodes if they are alive or not. If a data\_node is died, the DFS will get the replicas of all the files in the dead node and write it to some where else to keep the replica the number equals to replica factor.

DataNode:

The datanode is the node that actually store or the middle files and final files and all the replicas in DFS. Client is also a datanode(ReadOnly) and the mapper and reducer will get the input file through DFS.

DistributedInputStream and DistributedOutputStream:

They are higher level of NameNode, which represent the inputstream and outputstream in DFS.

**Assumption:**

MasterNode(JobTracker) and NameNode will never die.

**Requirements we meet:**

DFS: We have a pretty nice DFS that handle all the file IO and make the mapper and reducer task read and write as if they are doing local IO. The DFS can also handle datanode failure.

Minimize dispatch latency: We maintain process running on each task-node and let them periodically ask for tasks.

Initiate the execution of the program from any participating node.

Execute several jobs concurrently and correctly: JobTracker has a list of JobInProgress that is running concurrently and correctly.

Schedule and dispatch maps and reduces, to maximize the performance gain through

parallelism within each phase, subject to the constraints of the initiating program.

Recover from failure of map and reduce workers:

JobTracker will periodically check if each task\_node is ok, if someone is down, we will redo all the work it has done and rewrite all the file it has written to make the process logically simple. We have timeout exception to avoid forever waiting.

Provide a general-purpose I/O facility to support the necessary operations:

It is within the DFS.

Documents for both system and user.

Two examples: wordcount and uniquewords (to obain a list of unique words in document).

**Requirements we haven’t meet:**

We doesn’t have a very good tools to mange the system. User and system manager has to use ACTL+C to end each process.

There’s a bug with comparing key values that we haven’t figured out yet, causing reducer to sometimes be unable to combine all values for a given key together (and causing JobInProgress to incorrectly split files for reduce tasks as well).

**The capabilities and limitations of your project**

Our Map-reduce framework together with DFS can be use as real map-reduce. We can start from any node as client and any node as task nodes or data nodes. Just the master must be certain, the same as in the system configuration.

Future works: If we have more time, we would like to develop a more intelligent way for file replica management. That is to say, every time we wants to write replicas, we always choose nearby nodes to the origin node which can lower the Internet IO expense.

As mentioned, there’s a bug with comparing byte arrays for the keys.

**Highlights in our Project**

In the map-reduce project, we use Java’s RMI nearly everywhere. We use it for all the communication between different nodes. In this case we don’t have to deal with the sockets directly. As we can see, if helps our system to be easily maintained. Moreover, we take advantage of multi core CPU for multi threading, which makes the system has good scheduling ability.

**Highlights in our Project**

**How to use our system for testing:**

Use “default.conf “ for system and “sample.conf” for client. Typically you can start 1 master, 2 task nodes , 3 data nodes and 2 clients. This can basically cover the experiment parts. Because we set the replica factors to be 2, so if you want to test the failure recovery on data nodes, you should have 3 data nodes at the beginning (otherwise the replication factor will be unable to be met for future file writes). You can shut down task nodes or data nodes for failure recover testing.

In bin folder, type “java ha.mapreduce.MasterNode” + system configuration file will start the master and name nodes.

For task nodes, type java ha.mapreduce.TaskNodes + system configuration file + your host+ your port.

For data nodes, type java ha.mapreduce.DataNodes + system configuration file + your host+ your port.

Just a FYI that all host:ports typed in when starting task nodes, data nodes and clients can be random but should be the full path and real host. For example, you can start a task node in ghc70:8000 so you will type in

java ha.mapreduce.TaskNodes ../sample.conf [ghc70@ghc.andrew.cmu.edu:8000](mailto:ghc70@ghc.andrew.cmu.edu:8000)