Design Document

Shruti Mahale : 5160832 *Utkarsh Kajaria : 5205243*

In this project, we have implemented a simple MapReduce-like compute framework for implementing a sort program. The framework will receive jobs from a client, split job into multiple tasks, and assign tasks to compute nodes which perform the computations. This framework is fault-tolerant and guarantees correct result even in the presence of compute node faults.

Thrift Files:

1. Node.thrift: Contains one thrift struct. It is "node" which is data structure of the nodes in the File System. A node contains IP, port, and its ID which is randomly assigned to it by Server.

2. Server.thrift: This thrift file contains the thrift call functions the server implements.

3. ComputeNode.thrift: This thrift file contains the thrift call functions the compute Node implements.

Components:

We have implemented 3 components.

1. ***Client:*** The client sends a job to the Server called '*sortFile'* with filename as parameter. This file contains the data to be sorted. It also passes the *chunkSize* and *mergeNumber* along with FileName. ChunkSize is in bytes (so if you pass 1000 that means chunk size is 1000 B). MergeNumber is the number of intermediate files for merge tasks. When the job is done, the client gets the filename as a result of job which stores the sorted data. This file name is displayed on console.

2. ***Server***: The server is the central point of contact. It is configured to be on VM "csel-x32-01.cselabs.umn.edu" and on port "9098".

Fault Injection:

The parameters to be passed to the server are '*total number of compute nodes*', '*NumForRedundantTasks*' and '*fail probability*' out of 100. Fail probability is the probability with which this node will fail while executing the task. This is used to test if the system is capable of injecting faults, in order to implement fault detection and recovery mechanism. This probability is same for all the compute nodes. NumForRedundantTasks is the number of nodes to be chosen out of all the active nodes for performing proactive fault handling.

The client hands over the job to Server by calling '*sortJob'*. The server receives the sorting jobs submitted by the client. It splits the job into multiple tasks depending upon the chunk size parameter passed. Thus, *'mapper'* function call is made. Here, the input file is divided in various chunks and assigns each task to a compute node randomly. We have used threading for this purpose to achieve asynchronous nature of assigning tasks. The compute node sorts the chunks and store individual results in compute node. After,

the mapper finishes, '*reducer'* is called which takes all the intermediate files and merges them until we obtain results in a single file. This file is returned back to server.

The server also print the status of job. The '*goodJobCount'* variable indicates how many tasks are complete for the mapper and reducer phase. The '*badJobCount'* variable indicates how many tasks failed and due to what reason in the mapper and reducer phase. The bad jobs are the faults which occurred during one iteration of mapper and reducer. The mapper and reducer is called recursively until all tasks are completed.

When the job is completed, the server also prints the elapsed time to run it along with the output filename which includes sorted output returned back to the client. It also prints the total number of faults occurred in the system.

3. ***Compute Nodes***: Compute nodes will execute tasks (either sort or merge) sent to them by the server. Each compute node executes two kinds of tasks:

1. Sort: This is done in '*sortChunk*'. In this task, the compute node sorts the given chunk data and output the sorted result to an intermediate file (filename will be returned to the server). When the server receives all intermediate filenames for all tasks, i.e., when all sort tasks are done, it will assign merge tasks to compute nodes. All the intermediate files are stored on the current node.

Once we get the offset to form the chunks, the compute node calls a method called '*preprocessStart'* and '*preprocessEnd*'. These functions adjust the offset accordingly such that all numbers are processed only once without anyone being missed out. They handle the case in which offset is not space (the delimiter for the input file).

While implementing this logic, we encountered a case in which the chunk offset changes such that no number remains to be put in file. In such case, we pass the string "dummy" to the reducer.

2. Merge: This is done in '*mergeSortedChunks*'. In this task, the compute node will take a list of all intermediate filenames as an input. It will take a 'mergeNumber' number of files at a time. A compute nodes will merge those set of files and output a new sorted result. This goes on until we get one file which contain the whole of the data. When all merge tasks are done, i.e., when the job is done, the server will return the filename which includes the result (sorted output) to the client.

The mechanism used for this case is the external merge sort. We took this to efficiently handle the large data.

Naming convention for file:

All the files in mapper phase will be named as filename_0_chunknumber

All the files in reducer phase will be named as filename_roundnumber_mergeJobNumber.

RoundNumber increases from 1 onwards as described in the PA3 document.

Fault Detection and Recovery:

The system can detect faulty nodes and recover the task executing on them by re-assigning it to a different compute node. This fault detection mechanism uses heartbeat messages for its implementation. That is,

if the server notices that a compute node stops working (is crashed), the server will re-assign tasks previously assigned to that faulty node to other nodes. Any intermediate file generated by the faulty node will be stored without any loss. i.e., only the tasks which have not been done will be re-assigned. The server has a job tracker '*pendinglist*' which will be responsible for handling this functionality of the system.

Proactive Fault Tolerance (Extra credit)

The server assigns the same tasks to multiple nodes ('NumForRedundantTasks' number of nodes) rather than re-assign the task which was executed on failed node to another node. Once the server receives the result from any nodes, it calls 'killMapJobs' thus kills the redundant task(s) executed on other nodes. It does this by maintaining a map and checking if its value is false. In this case, "aborted" string is returned to the Server to identify that this was an aborted/cancelled job.