CSCI 5105 Distributed Systems- Project 2

Sathishkumar Vijayakumar - 5068160 - vijay072@umn.edu
Bernardo Augusto Godinho de Oliveira - 5217756 - godin040@umn.edu

Components:

CentralNode:
The Central node is responsible for three different actions:

1. Joining and Building the compute network
2. Issuing Merge/Sort Jobs to Compute Nodes
3. Maintaining Tracking Information
4. Fault Tolerance

Design Decisions used in CentralNode:
1.1 Joining and Building Compute Nodes: A list is built containing entries for each of the compute nodes that join the central node. The *ListOfNodes* datastructure that we maintain internally contains node information and the sort/merge job list that the compute node currently executes.

1.2 Issuing Merge/Sort Jobs to Compute Nodes: For the Merge/Sort jobs, we used non-blocking calls, since the merge/sort jobs that we use inside the compute nodes spawn parallel threads for each of the jobs that they receive, since blocking calls will result in serializing the threads.

1.3 Tracking Information: Two types of tracking is performed, in order to enable flexible rescheduling of jobs.

- File related information: This contain information such as the number of parts for a particular file, the status of sort of each part of the file, the status of each of the temporary merge files. Once sort/task successfully gets executed in the compute nodes, the status of the sort/merge task is updated in the return_sort()/return _merge() functions inside the CentralNode, based on which further actions on sorting and merging are performed. The update procedures for these status elements are performed using locks to be consistent.
- Node Related information: This is maintained in a list containing number of active nodes and their sort/merge tasks executed currently. This type of tracking is found to be important during fault tolerance, since the tasks executed by the faulty nodes were extracted and reissued on other non-faulty nodes.
- Statistics: Finally, a set of statistics such as number of faults, execution time of the system and number of file chunks for the given size of the file chunks.

1.4 Fault Injection and Tolerance: We model fault tolerance by sending inverse of the failure probability as an input to the individual compute nodes. Also, compute nodes are polled once in every 100 ms to check for their availability. Once the CentralNode finds that a particular node is faulty, it will trigger the fault tolerance routine which dispatches its currently executed tasks to other routines. For an integer N sent to the compute nodes, the fail probability is 1/N(the node is guaranteed to fail within N execution steps). As mentioned in the previous section, we use node-based tracking information and re-execute tasks on other available compute nodes. When no other node is available to execute, the system gracefully terminates and a message is sent to the client which requested the sort that the sorting job is aborted due to node failures.

ComputeNode:
This component will execute the actions assigned by the central node. It performs sorts and merges depending on what task has been assigned by the central node. A heartbeat call to the central node is done regularly to show that the node is still running. It takes as parameter the probability N to fail were N means that it has 1 out of N chances to fail for each heartbeat. However, the value zero means that there will not be any failure in the current node until manual termination.

Design Decisions used in compute node:

- In order to handle fault tolerance, we designed Compute nodes as slave nodes, which do not have much intelligence attached to them. Most of the status tracking management except for managing the statistics of the individual compute node themselves, are handled in the central node. This is to ensure consistency and fault tolerance.
- Compute nodes are multi-threaded and are capable of handling any number of sorts and merges.
- Further, it can also handle multiple files requested by multiple clients due to the centralized tracking mechanism available.

FileClient
FileClient is a simple single threaded client that send files to the central node to be sorted. It connects to a central node to issue it based on the user's request. It calls a blocking call, clientsort() to the central node, which exits upon successful completion of the sorting of the file issued by it to the central node.

Operation flow

We are running a multi thread pool server that fires a new thread for every requests that it receives from the client. Once a request is received, the centralnode will split the file into chunks with size defined as its argument and write those chunks into files in the InterDir. Then the merge is called to merge and sort the parts of the file inside the intermediate directory. After performing the merge and sort for the different levels necessary to sort the entire file it will save the result to OutputDir and notify the client that the process has finished.

Thrift RMI Interface:

Fig. 1 shows the overall block diagram for the system that we use. It has a central node and a set of compute nodes. Any client that wants to execute sorting contacts the central node. In the shown methods in the figure, except for ClientSort(), through which the client communicates the server, all the other methods are non-blocking calls.

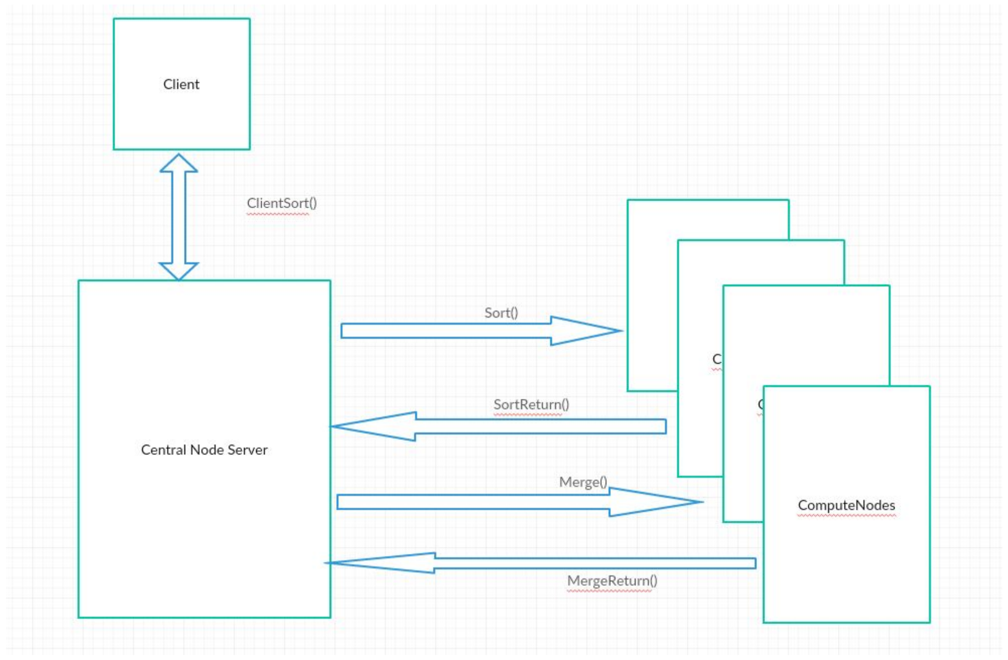

Fig.1 Thrift Interfaces between Client/CentralNodeServer/ComputeNodes

Performance Evaluation:

In order to understand the performance evaluation of our system, we used 4 metrics. They are

1. Chunk Size of the file
2. Size of the input file
3. Fault Tolerance probability
4. Number of Nodes in the system

Also, in order to study the effect of each and every parameter separately without interference from the other metrics, we fixed the other parameters to a fixed value across experiments.

1. **Chunk-size of a File:**

In order to evaluate the effect of chunk-size of a file, we fixed the filesize to be the largest from the given sample files, 100,000,000 and the number of nodes to 4. Also, we disabled the fault injection capability from the compute nodes. Average execution time in the compute nodes and the central nodes are shown. Three different chunk sizes were studied, 256, 1024 and 4096. We observed trend in increase in execution time for the medium sized chunk, since, in smaller sized chunk, sort tasks will complete faster and in larger sized chunks sort tasks will take a larger amount of time and merge tasks will take less amount of time. But in case of 1024, both merge and sort takes considerable amount of time resulting in increased overall average execution of the task.

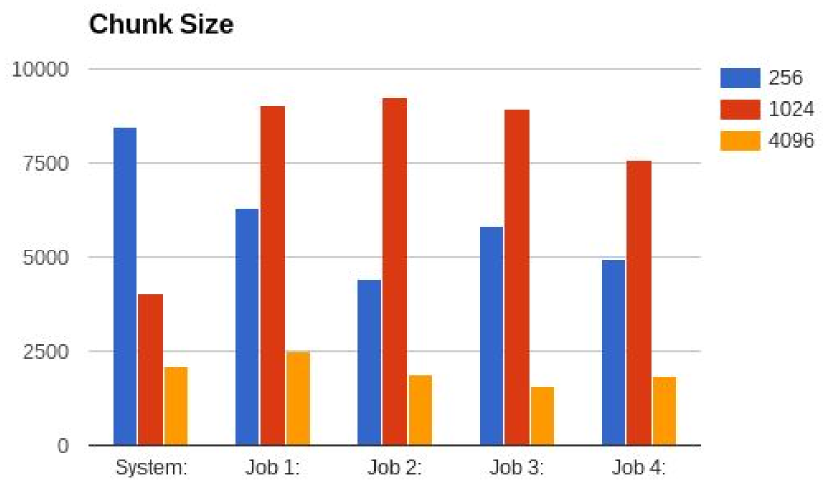

Fig 2. Variation of Average execution times across central node and compute nodes with respect to file chunk size

2. Size of the input file

Three file sizes 10,000,000, 6,000,000, and 200,000 were used. Also in this analysis fault tolerance was disabled. A medium sized chunk of 512 bytes were used. There is a clear decrease in the average execution times across nodes, with decrease in the size of the file.

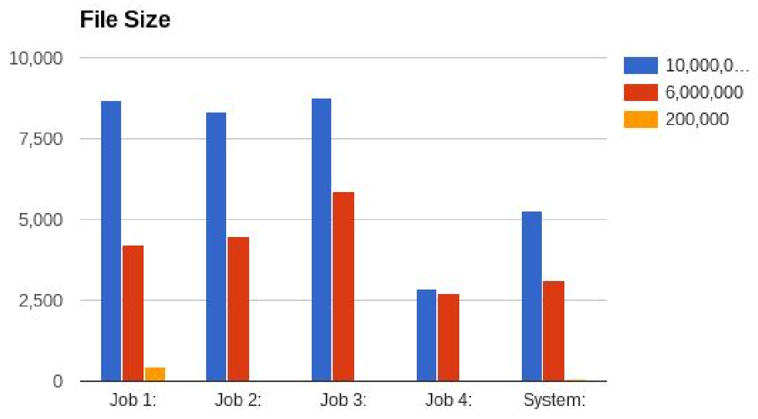

Fig 3. Variation of Average execution times across central node and compute nodes with respect to total file size

3. Fault Tolerance:

We selected three fault tolerance parameters, to inject faults of varying frequency, 200, 600, 2000 with 200 being highly probable state of failure and 2000 being low state of failure. A file size of 10,000,000 was used with a chunk size of 512 bytes. Clearly, there was a significant increase in execution time of the central node with increase in number of faults of the system. However, execution times of individual compute nodes increase or decrease depending on the number of tasks being reassigned to them. In blue trend, node 4 was killed first, followed by node 3, followed by node 1 and followed by node2). Performance parameter were extracted until all nodes stop working and the system comes to a complete halt.

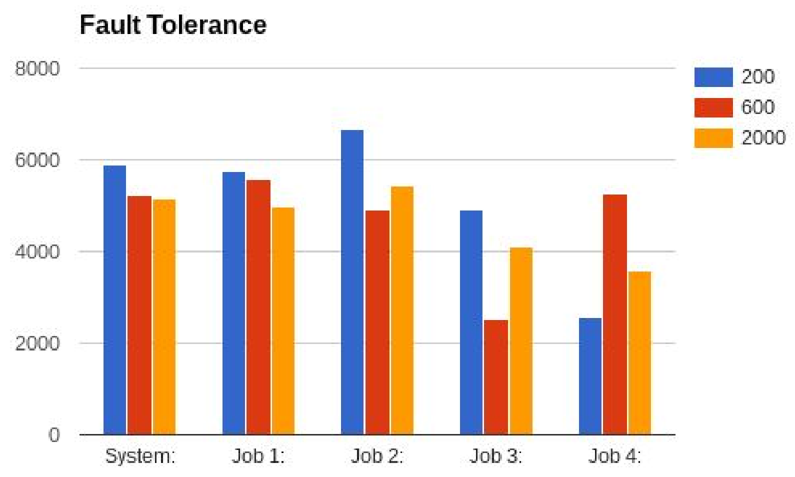

Fig 4. Variation of Average execution times across central node and compute nodes with respect to Fault Tolerance

4. Number of Compute Nodes:
In this experiment we saw a decrease in the execution time of overall system with increase in the number of nodes. This is because of the fact that, increase in the number of nodes increase the computing capacity which reduces the execution times.

Note: We provided a separate document containing information on the numerical values of the results of these parameters from the experiments that we ran.

Testing Description:

1. **Functionality:**
Several tests varying the filesize were performed from the client side in order to check the correctness of the functionality of the system. This step is to check the functionality correctness of the merge and the sort, comparing its result to a output proven to be correct. We have used the difftool to compare the file obtained by the last sorted merge with the correct output provided.

2. **Parallelism:**
To test if each node was processing just a part of the payload we saved the file that has been processed in the intermediate directory and then checked each file. By comparing each generated file we noticed that it is processing different numbers and thus it is being divided correctly.
3. **Heartbeat and Fault Injection/detection/recovery**
A heartbeat is issued by the compute node periodically so that the central node can keep track of the available nodes and if one of those fails. To test the failure detection and recovery we added a probability for the compute node to crash when a heartbeat is issued. Since the central node keep track of the jobs assigned to each node it can simply reassign it to another node. After testing the system we noticed that it is capable of reassigning many tasks to other nodes when multiple nodes fail and still generate a consistent output file. Several random sleeps were provided in the fault tolerance handling function to test consistency of the status elements, which are key to the execution of correct number of sorts and merges in order for the output to be valid. During each of these runs, we observed that our system performed consistently.

Instructions on How to run experiments:

Note:

- Please provide input files to InputDir/ within the project directory.
- Please clean the InterDir/ after client executes the completion of one file
- Output can be found in the OutputDir/

Compile:

```
$ source compile
```

First, it is necessary to run the central node. Run command (where 1024 is the size of the chunk and 2 is the number of files merged in the merge):

```
$ java -cp .:./jars/libthrift-0.9.1.jar:./jars/slf4j-api-1.7.14.jar project3.CentralNode 1024 2
```

<Command><Chunk_size><number of merge batch>

Run compute node (where 0 means it will not fail):

```
$ java -cp .:./jars/libthrift-0.9.1.jar:./jars/slf4j-api-1.7.14.jar project3.ComputeNodeServer 0
```

<Command><Int for fail probability(bigger value for lesser faults in time)>

Run file client:

```
$ java -cp .:./jars/libthrift-0.9.1.jar:./jars/slf4j-api-1.7.14.jar project3.FilesClient 200000
```

<Command><Filename(without the InputDir/)>