CSCI 5105 Introduction to Distributed Systems

Programming Assignment 3

(Simple MapReduce-like Compute Framework)

Arjun Subrahmanyan (Student ID : 5217513)
Devavrat Khanolkar (Student ID : 5211324)

Introduction

The following is a **design document** that describes the different components of the Simple MapReduce-like compute framework that was implemented in **Java** using **Apache Thrift**.

Starting the system

A simple assumption of the system is that each **compute node** that will be a part of it knows the IP and port number of the **job server** service. The job server service is started with the following parameters:
(1) The *chunksize*, which represents the smallest size into which the input file will be divided into and distributed among the various compute nodes
(2) The *mergeFileListSize*, which represents the number of intermediate files (sorted) that are going to be merged together at once, in each stage of the merge.

When each compute node joins the system, parameters like it's own port number and the IP and port number of the coordinator are provided via command line. The compute node then notifies the job server that it has joined the system.

Job Server Service

The job server is a multi-threaded service, which means that a new thread is created which processes each request separately.

1. *join(IP, Port)*
   Once a compute node starts, it notifies the job server with its own IP and port number. The latter then updates its *nodes* list with this information.

2. *submitJob(inputFile)*
   This is the method which is called once a client issues a request to the job server, where *inputFile* is the file that contains the numbers to be sorted. The program rides on the assumption that all input files are located under a directory named *input*.

The first thing this method does is create a directory *intermediate* for all the intermediate files generated during execution of a job. To divide the input file into chunks, we make use of *offset* and *chunksize* parameter. While atleast *chunksize* bytes of data is present in the input file from the current *offset*, we designate each *chunk* to a different compute node (using the *nodes* list, and in a round-robin fashion). Once designated, *offset* is incremented by *chunksize*. The details of how the exact sort works is under the section *Compute Node Service*.

We cannot start the merge process, until all the individual sorts are completed. For this reason, we need to track the tasks that were initially assigned and those that were completed. We make use of a map *sortAssignedTasks* and update it whenever a task is given to a compute node. Once a task is finished, it is removed off the map (see the method *done()* below). We wait until this map is empty, which means all sorting tasks are completed. This loop also involves reassigning of tasks in case of faulty compute nodes, and this is discussed under the section *Fault Tolerance*.

As the first step in merging, we compute the number of merge passes needed for the given number of sorted files and the parameter *mergeFileListSize*. In each pass of merge, a compute node is assigned with *mergeFileListSize* number of intermediate files to merge. Details of the merge can be found in the section *Compute Node Service*.

Like before, the next pass of merge cannot happen before the previous pass has completed fully. Again, we maintain a map *mergeAssignedTasks* to keep track of the tasks in progress. The loop also takes care of reassigning of tasks in case of faulty compute nodes.

Once the merges are done, all intermediate files are deleted and the output file is moved to a separate directory *output*. During job execution, the job server prints out information relating to the tasks-compute nodes mapping, the total time taken for the entire merge sort and also the number of faults that occurred in this time frame.

3. *done(intermediateFile, taskNumber, type)*
   This is a method that is invoked by the compute node once it finishes either a sort or merge task (determined by *type*). Here, we remove the task from the corresponding map (described above) and the server prints out the task that was completed.

Compute Node Service

This component performs all the tasks (sort or merge) that are sent to it by the Job Server. Each compute node is also associated with a *failProb*, which is the probability of itself failing. To simulate this, we run a separate thread after starting a compute node, which tosses a coin to decide if the compute node should fail or not. The actual failure is done by a call to *System.exit()*.

1. *startSort(inputFile, offset, size, taskNumber)*
   This is the method which starts off the actual sorting, after updating the number of tasks that the particular compute node has received. The actual sorting is started

via a new thread, and hence control can be returned back to the Job Server. This ensures that the compute node need not wait for all tasks to arrive before it begins one, and also concurrent execution of all the sort tasks. The thread information along with the task information is stored in the map (see below). Details of the sort procedure is under the section *Sort*.

2. *startMerge(fileList,taskNumber)*
   This is the method which starts off the actual merging, after updating the number of tasks that the particular compute node has received. Like above, the actual merging is done on a separate thread and control is returned, for the exact same reasons stated. Also, the thread details along with the task information is stored in the map (see below). Details of the merge procedure is under the section *Merge*.

3. *printStatistics()*
   This method prints out the total tasks that were executed at the particular compute node. The time taken for each task is printed as it finishes in the respective thread. This method is invoked at the end of the job, by the server.

Sort

This method works with underlying parameters as *inputFile*, *offset* and *chunksize*. To read the contents of the files without loading the entire contents on to memory, we make use of the *RandomAcessFile* class, with its *seek()* and *read()* methods. The call to sort would initially mean "read all numbers starting from *offset* and till *offset + chunksize* ".

This cannot be done directly, because offset could be holding a byte, which is part of an entire number by itself. For eg, it might point to the digit '2' in the number '12'. So, we maintain two pointers *actualOffset* and *rightPointer* that are initialized to *offset* and *offset + chunksize*. We then move both pointers leftwards until we encounter a space character. In effect, we are now reading bytes between two spaces and which is what we need. This consistent procedure also ensures that no two sort tasks will ever encounter the same number again, because the previous one would end with a space, which is the same space that the latter one would begin with.

After reading the bytes, we convert it into an array of the corresponding methods. The array is then sorted and written directly into the corresponding intermediate file. Finally, the *done()* method is invoked at the server, and the time taken for each sort task is printed out at the compute node.

Merge

This method is invoked with underlying parameter as *fileList*, the list of intermediate files. We merge the files in the *fileList* two at a time by standard merging procedure.

The resources are closed the elements of *mergeList* are written to the intermediate file. Finally the *done()* method is invoked at the server, and the time taken for the merge task is printed out at the compute node.

Fault Tolerance

Each compute node is assigned a *failProb*, the probability that it will fail at any point of time. As stated above, when a compute node is up, it runs a separate thread, which decides whether the node should fail or not, every 10 seconds. By fail, we mean exit the program using *System.exit()*

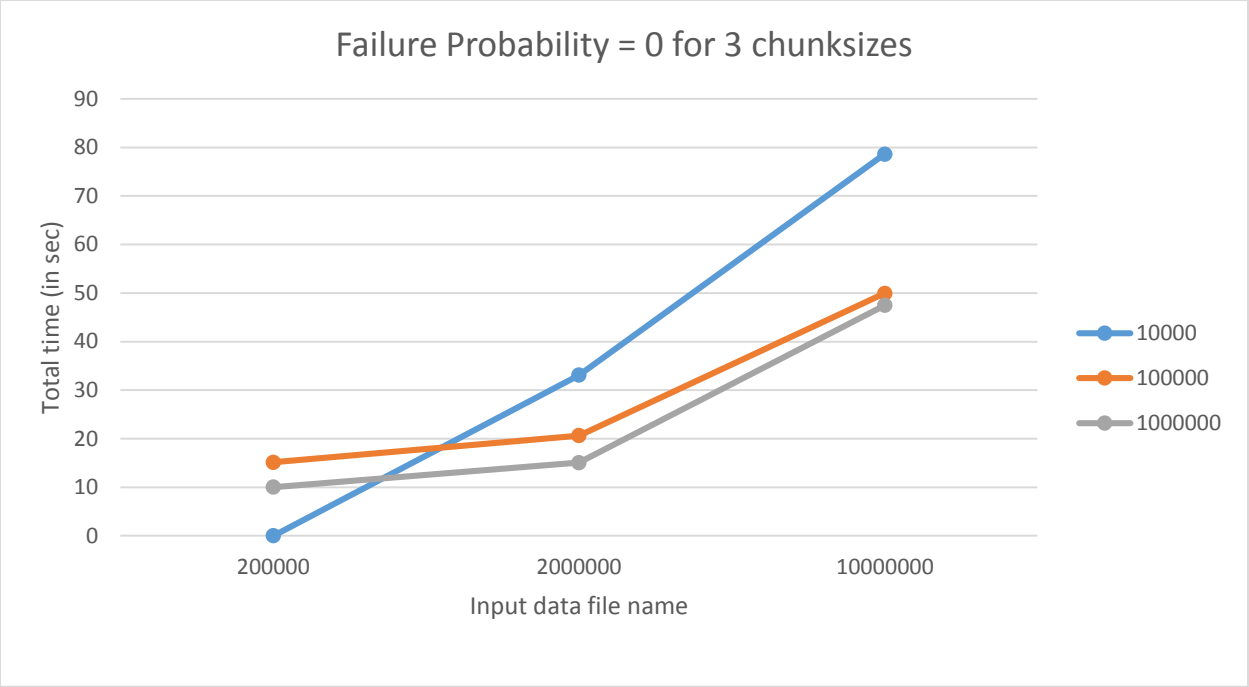
As per *submitJob()*, tasks are assigned to the nodes in a round robin fashion. We cannot start merging before all sort tasks are completed. Hence we have a while loop that keeps checking every 5 seconds whether the nodes that received the tasks are alive or not. This simulates a **heartbeat** mechanism. If the job server finds the node to have failed, then it reassigns the task to another node, and the corresponding map is updated.

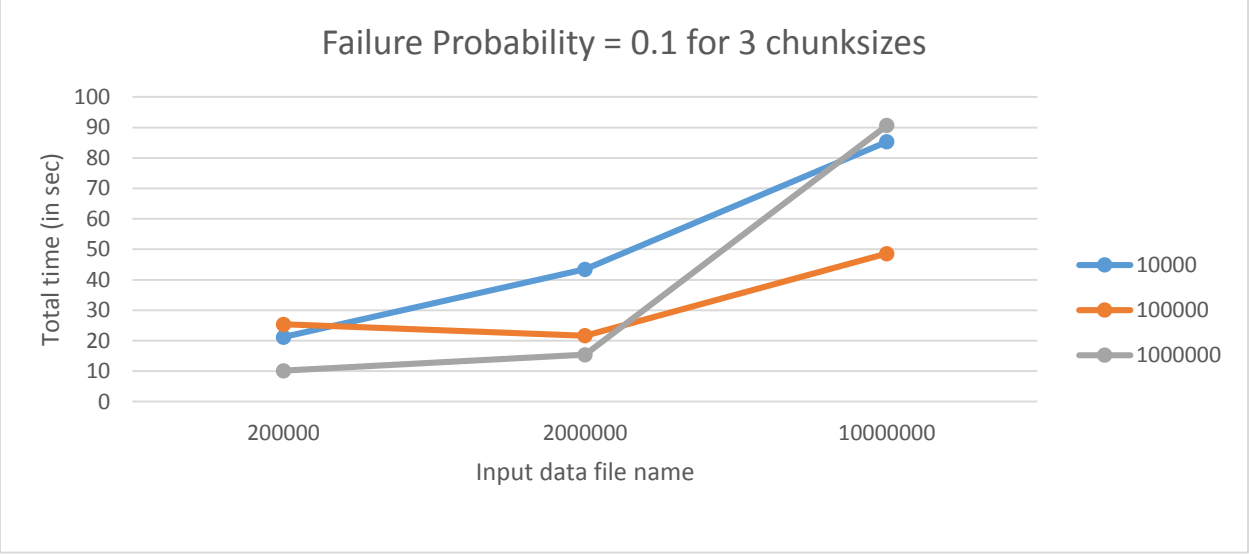
The same mechanism is followed at the end of every pass of merging, to ensure that the next pass cannot begin until the previous pass has completed.

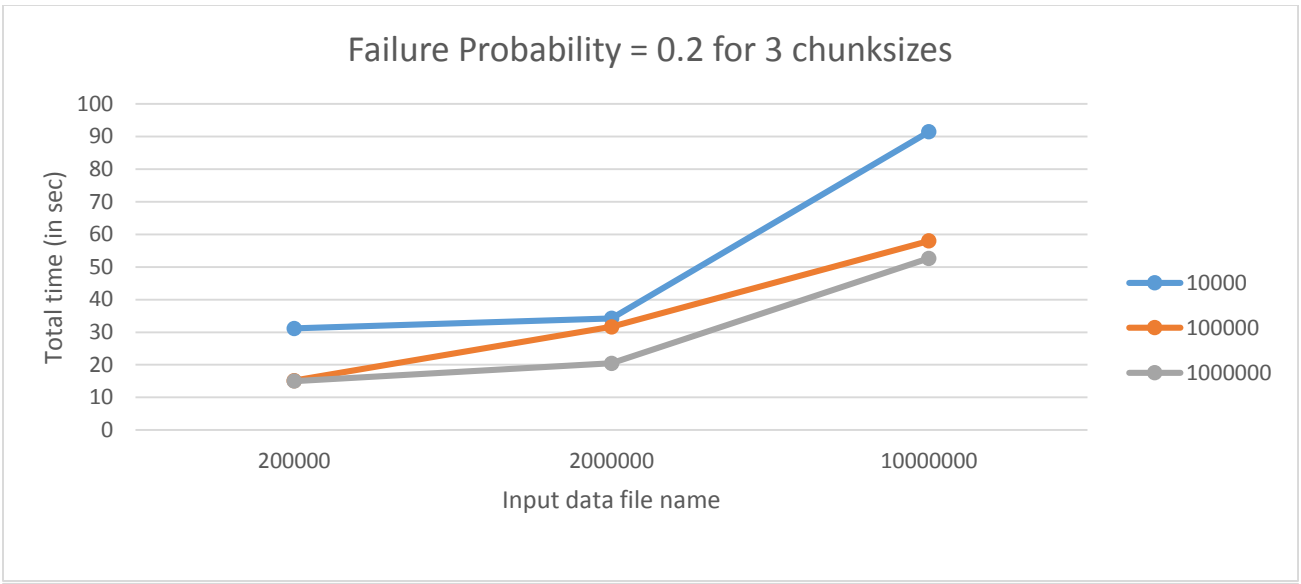
Miscellaneous

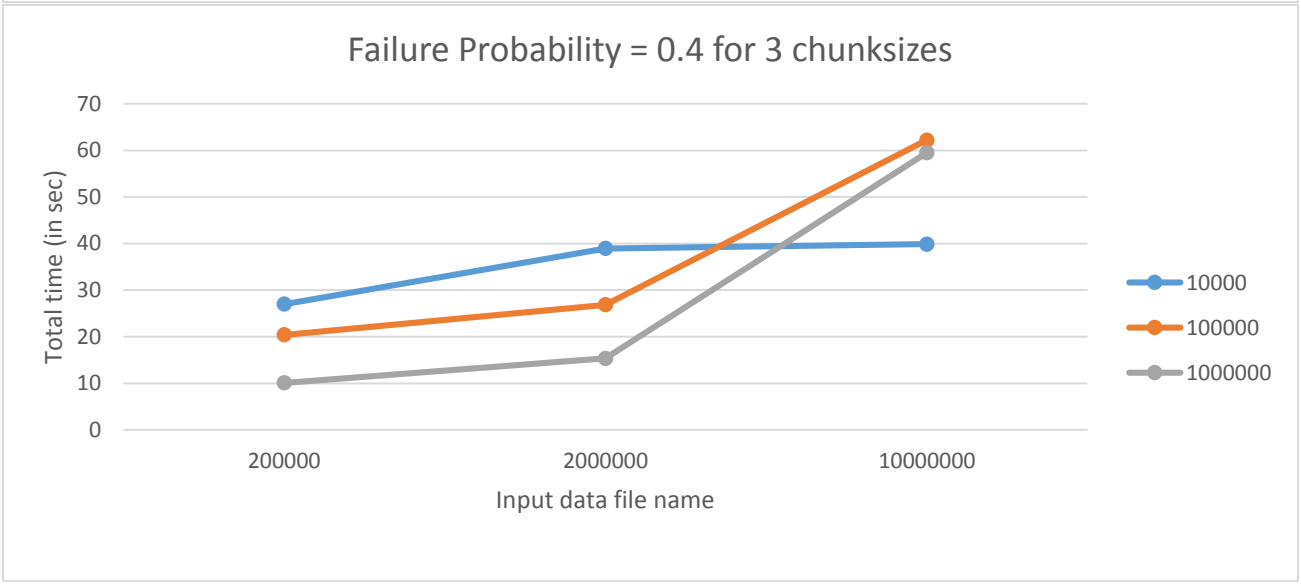
The job server prints various statistics relating to the tasks completed and pending. It also clocks the total time of execution for the entire job. Finally, it deletes all of the intermediate files from the disk and moves the final merge file into a folder **output** in the same directory. The filename of the sorted file is then returned to the client in the UI.

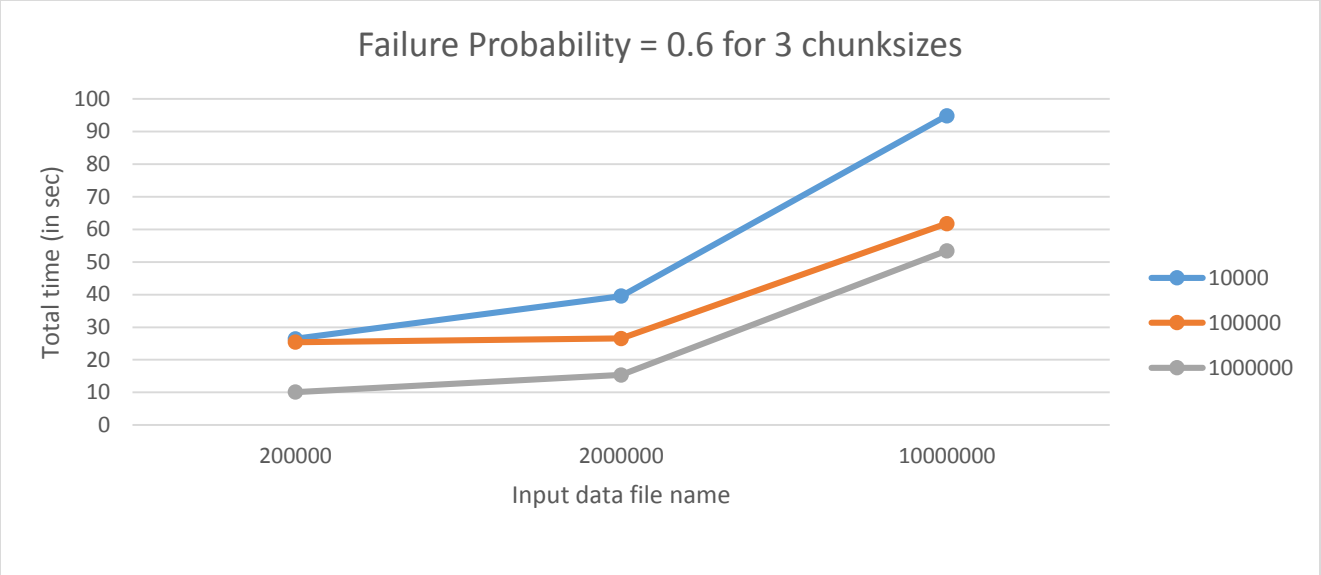
PERFORMANCE EVALUATION GRAPHS:

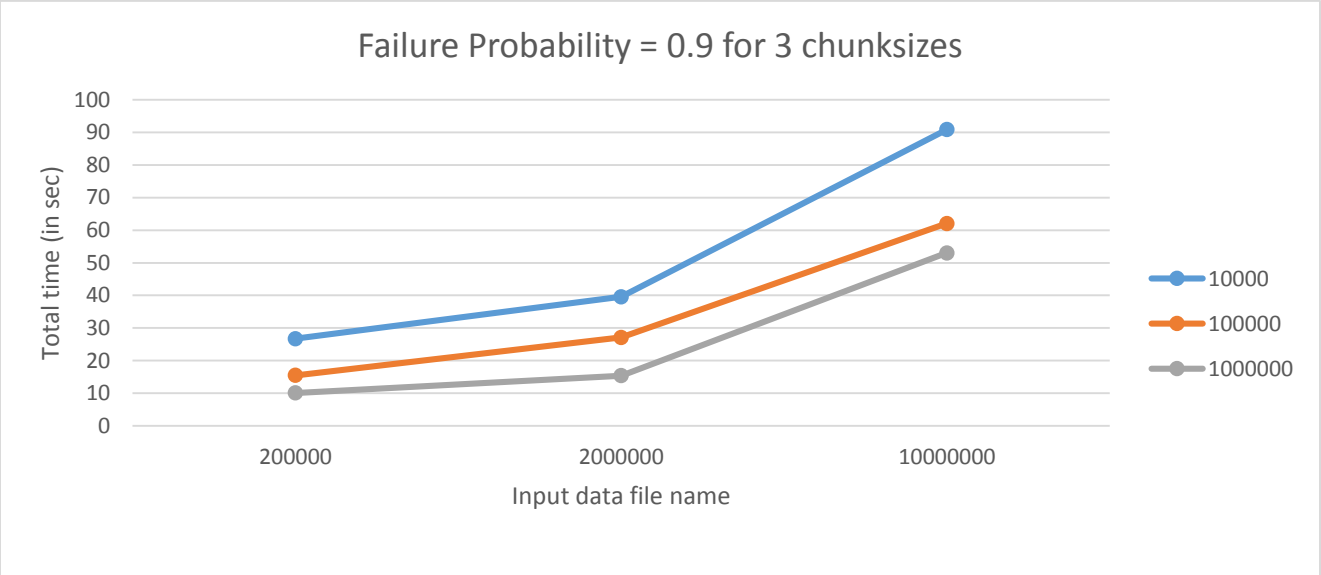





Failure Probability = 0.2 for 3 chunksizes
Total time (in sec)
100
90
80
70
60
50
40
30
20
10
0
200000
2000000
10000000
Input data file name
10000
100000
1000000


Failure Probability = 0.4 for 3 chunksizes
Total time (in sec)
70
60
50
40
30
20
10
0
200000
2000000
10000000
Input data file name
10000
100000
1000000


Failure Probability = 0.6 for 3 chunksizes
Total time (in sec)
100
90
80
70
60
50
40
30
20
10
0
200000
2000000
10000000
Input data file name
10000
100000
1000000


Failure Probability = 0.9 for 3 chunksizes
Total time (in sec)
100
90
80
70
60
50
40
30
20
10
0
200000
2000000
10000000
Input data file name
10000
100000
1000000