Programming Assignment 3: A simple MapReduce-like compute framework

Execution Steps

Java Version - 1.7.0_76

Thrift Version - 0.9.3

Compilation Steps:

1) *javac -cp .:<thrift_jars_directory>/* Server.java*
2) *javac -cp .:<thrift_jars_directory>/* ComputeNode.java*
3) *javac -cp .:<thrift_jars_directory>/* Client.java*

Steps to make system up and running:

Important Note : Failed Probability should be between 0 and 1. Please make sure there are no other files in the folder where sort file is residing

1) *java -cp .:<thrift_jars_directory>/* Server <server_port_no> <chunk_size> <merge_size><replication_factor>*
2) *java -cp .:<thrift_jars_directory>/* ComputeNode <server_ip> <server_port_no> <compute_node_port_no> <failure_probability>*
3) *repeat Step 2 for k-1 number of times, where k = number of compute nodes you want to add in the system*
4) *java -cp .:<thrift_jars_directory>/* Client <server_ip> <server_port_no> <file_name_with_path>*

Design Architecture

Our System has the following main modules/components :

Compute Node is node where the actual computation tasks such as sort and merge will happen. There can be multiple compute node in our system. Compute Node knows the address of server and therefore server must be up before any compute node comes into system. Once Server is up, compute nodes can join and leave system at any point in time. When a compute node joins a system, it notifies the server node about its presence. Server will store this information at local list. It is similar to data node in Hadoop System.

Server Node is master node which is responsible for assigning tasks to compute node. It receives request from client, assign tasks to compute nodes, gather result from compute node and returns final solution to requesting client. Server is the first node in system. It is similar to namenode in Hadoop System.

Client sends request to server for sorting. He sends filename information to Server and in return receives sorted file with filename. There can only be one client in a system at any point in time.

It is important that there should be at least one compute node up when client sends request otherwise, client will block until any compute node joins system.

Operations :

Timeout mechanism of Thrift - We used timeout provided by thrift to make a decision whether a particular task is progressing or not. If there is a time, we will add this task again to task list. However, it is important that timeout should be large enough otherwise, operations will keep on getting timeouts.

Sorting Algorithm:
This algorithm runs at compute node. It receives filename, offset and Chunk size as parameter and return Intermediate file name. Algorithm will read bytes maximum of chunkSize + buffer_size (In our case 5 bytes, so as to avoid extra read I/O) starting from offset. Algorithm will sort these numbers and store them back into intermediate file and return intermediate file name. Intermediate file name has naming convention in which filename is offset name. When our job completes we are deleting all intermediate files.

Merge Algorithm:

This algorithm runs at compute node. It receives list of filename (which are intermediate sort files created in sort algorithm) as parameter and return intermediate merge file name. Our Merge Algorithm works on concept of merging k sorted list using heap data structure. In our Algorithm , we will first open all files (provided in the list of filename), read first integer of every file, finds minimum among those, store in intermediate merge file, removes elements from heap and insert next element which is fetched from same file which has minimum element. Intermediate file name has naming convention in which filename is name of first filename in list. When our job completes we are deleting all intermediate files.

Fault Tolerance Algorithm :
Objective of Fault Tolerance algorithm is that our system should not start job from beginning in case of either task or node failure. Instead, it should start only from point where it failed. We implemented Fault Tolerance algorithm with help of local information at server node and timeout feature of thrift. If we did not received message back, within certain time frame, there are two possibilities :

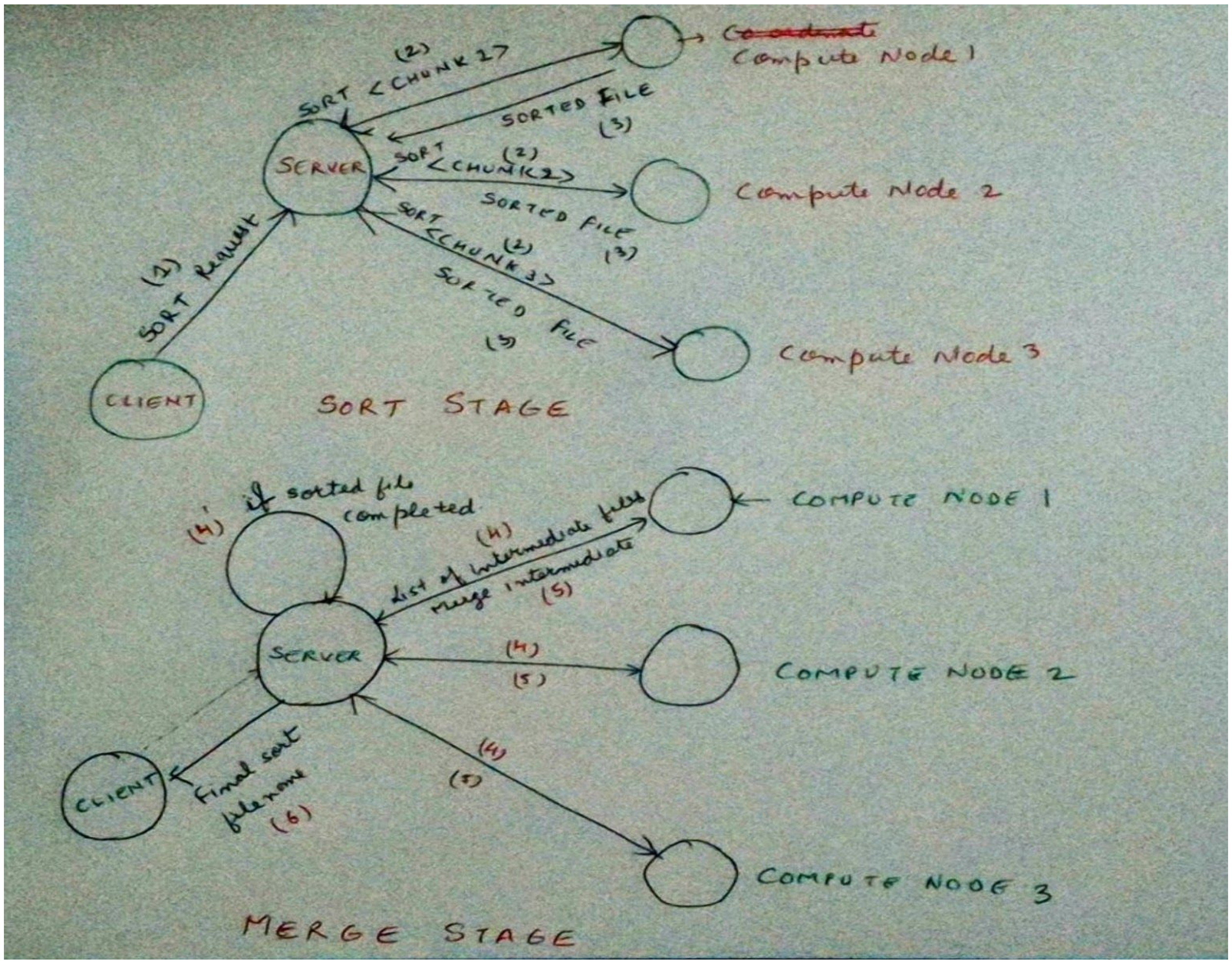
a) Compute Node failed :- If we did not receive result within certain time frame, we declare task as dead and add this task again into server's local task list. Since this node is unreachable from server, server will remove this node from available compute node list and at this point this node will be declared as dead.
b) Task Failed :- It works on same lines as above except that here node is reachable from server and therefore we do not remove this compute node from available compute node available list.

Proactive Fault Tolerance Algorithm :
Here, instead of waiting for task to get failed and get assigned again to tasklist, we are replicating task at front which will decrease response time. However as the tasks are getting replicated, it will burden our entire system. Here, initially instead of sending one copy, we are sending k copies of tasks(here k is replication factor). As we are given shared storage, we are using poll based mechanism to check whether intermediate file for this particular task has already been created or not. If the intermediate file exist in system, this task will kill itself, otherwise it will proceed. Here one thread is continuously polling and another thread is carrying out sort or merge operation.

Code Flow :
Program runs in two phases. In first phase, Sort task is done in which Server will send chunk size and offset to compute node and compute node will sort these chunks and returns intermediate sorted file. These intermediate files are stored in local pending list data structure. When sort stage ends, merge phase begins.

In merge phase, server sends list of maximum of ‘M’ (Merging factor) available intermediate files(received in sort phase) to compute node. Compute Node merges these files into one sorted file and returns to server. Server will add this into pending list and this step is repeated until Server detects that final sorted file has been created.

Finally, Servers returns this filename to client.



In above Figure, we have shown both stages :

Sort Stage:

Step 1: Client sends Sort Request to server with filename

Step 2: Server divides file into chunks and sends different offset and chunk size to Compute Node

Step 3: Compute Node sort their respective chunks, creates intermediate file and return intermediate file to Server.

Step 2 and Step 3 are repeated until server detects that sort stage has completed.

Merge Stage:

Step 4: Server will send (Compute Node pulls) list of intermediate files.
Step 4’: This step runs in background in which server checks about the presence of final sorted files
Step 5: Compute Node merges list of intermediate file and returns merged filename.
Step 4 and Step 5 are repeated until server in step 4’ detects about creation of final sorted file.
Step 6: Server return final sorted filename to client.
Fault tolerance :
In case of fault tolerance, step 3 or step 5 does not return anything and there will be timeout. In this case, we again add task and assign task to compute node which is similar to step 4 and step 2. In case later we detect that there is compute node failure, we remove compute node from list of available compute nodes.

Proactive Fault Tolerance:-
In this we replicated ‘k’ (replication factor) copies in step 2 and step 4. Also, when one of the replicated task is done, other replicated tasks will kill themselves.

Probability task failure :- We implemented this on compute node by generated random number between 0 and 1 and if it happens to be a failure, we throws an exception and terminate our task.

Thrift File
Thrift file contains two services -

1) **ServerService** exposes the following methods :
    - sort
    - systemInfo
    - joinSystem

2) **ComputeNodeService** exposes the following methods :
    - sort
    - merge

We have defined a new structure **ComputeNodeDAO** to facilitate exchange of node information between server and compute nodes. A new exception **TaskFailureException** has also been defined which is thrown whenever you try to fail a task with the configured probability.

Assumptions

We have assumed that system will run in stages. In first stage, we will make our main Server up. In next stage, we will make at least one compute node up and finally, in last stage, clients will issue sort operation from server. **We have assumed that our main server will never fail.** All sort operations will be issued to server only. All compute nodes and clients will knows IP address of the Server. Once a compute node joins system it will notify server about its presence. For Compute Node Heartbeat detection, we used thrift timeout option. Filesystem is shared among different nodes and renaming of file is atomic. Test file will start from digit ranging from 0 to 9999. Also, there will only be one space between any two digit and there will not be any other character. We have tested our solution on maximum file size of 100 MB. Also, we have assumed that for testing, you should have sufficient disk storage as temporary files are generated in between which gets cleaned up only in the end. Our merge stage, is optimal in the sense that it does not wait for "M" intermediate jobs before triggering merge operation. Instead it works on how much jobs are available. If there are M intermediate jobs available, it will send list of M files. However, if there are less than M intermediate jobs, it will send those jobs for merging rather than waiting for intermediate jobs to complete. One important assumption is that our timeout should be sufficiently large otherwise, it keeps on getting timeout and job will not be able to proceed further.

Test Cases

a) Client sends request to system having Single server and single compute node
b) Client sends request to system having Single server and multiple compute node
c) Client sends request to system having Single server and single compute node with Replication factor as 3 (can be any number)
d) Client sends request to system having Single server and multiple compute node with Replication factor as 3 (can be any number)
e) All compute nodes and server are on same machine (same location) and client on remote location
f) All compute nodes, client and server are on same machine (same location)
g) All compute nodes, client and server are on different machine (different location)
h) Repeat test cases (a-g) with and without proactive fault tolerant algorithm
i) Repeat test cases (a-g) with different failed probabilities such as 10%, 20% and so on
j) Repeat test cases (a-g) with fixed chunk size and different file size
k) Repeat test cases (a-g) with fixed file size and different chunk size
l) Explicitly closing compute nodes with CTRL+C

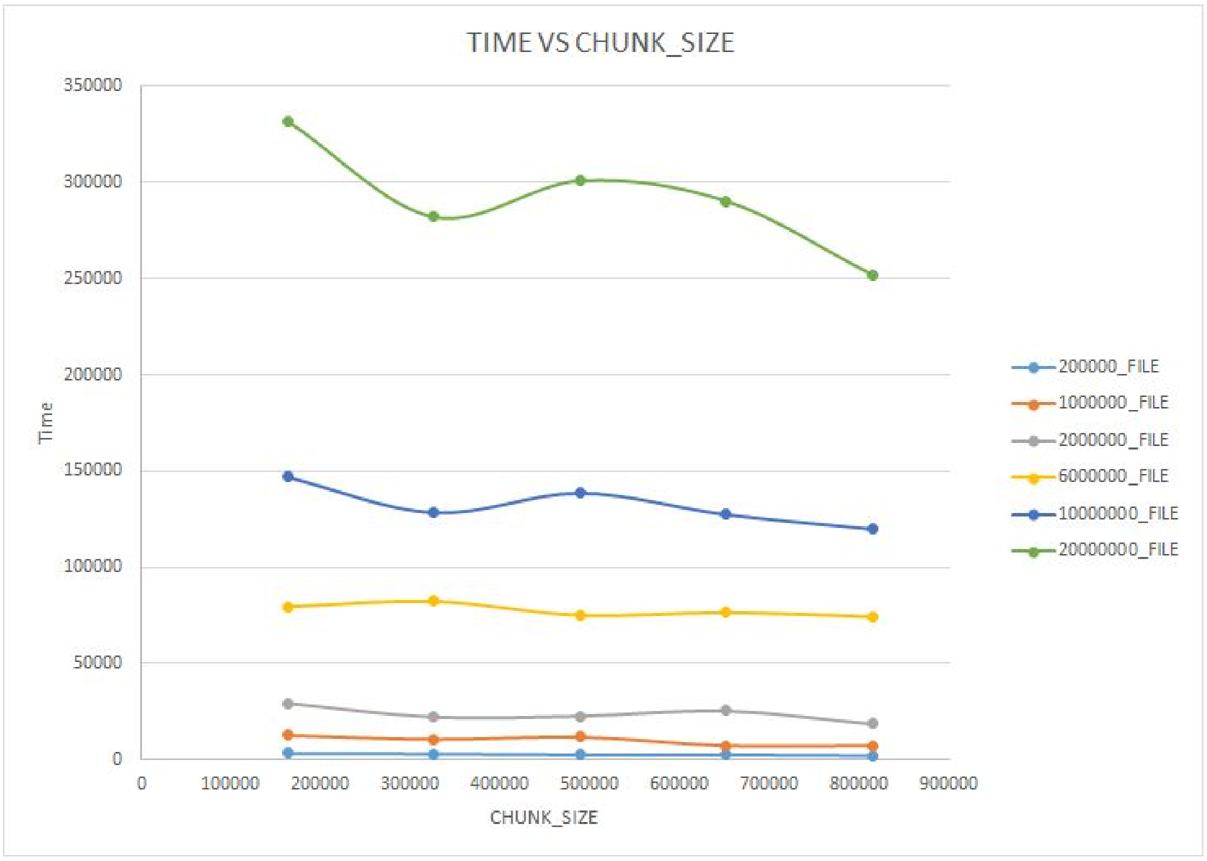
Negative Test Cases:

a) When Firewall is enabled which is blocking port number on client or compute nodes.
b) When we are using special port numbers such as 8080

c) When we are using port number which are already in use
d) When passing wrong IP address
e) When passing wrong port number
f) If server is down and we are trying to add compute node in system
g) If server is down and client is trying sort operation
h) If there is no compute node and client is trying to do sort operation
i) When replication factor is negative
j) When chunk size is 0
k) When file does not exist on which sorting has to be performed
l) When disk quota exceeded
m) When all compute node do not have permission to access file
n) When we are explicitly deleting intermediate files generated in sort and merge phase

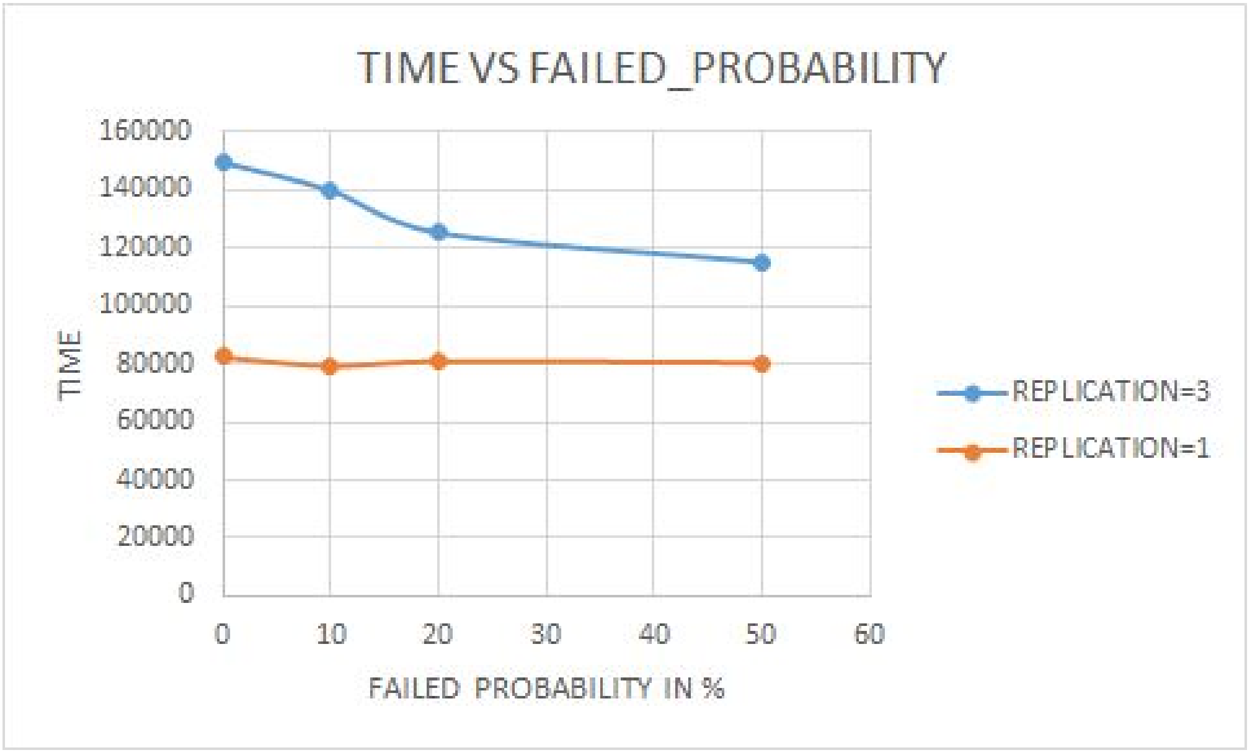
Result:
We successfully implemented A simple MapReduce-like compute framework using thrift and with proactive fault tolerance. We tried different chunk size, file size and task failure probability.
Different files on which we run our experiment
200000_file size : 977.6 kB (977,556 bytes)
1000000_file_size: 4.9 MB (4,889,007 bytes)
2000000_file_size: 9.8 MB (9,777,806 bytes)
6000000_file_size: 29.3 MB (29,333,297 bytes)
10000000_file_size: 48.9 MB (48,889,651 bytes)
20000000_file_size: 97.8 MB (97,777,906 bytes)
We executed the following test runs -

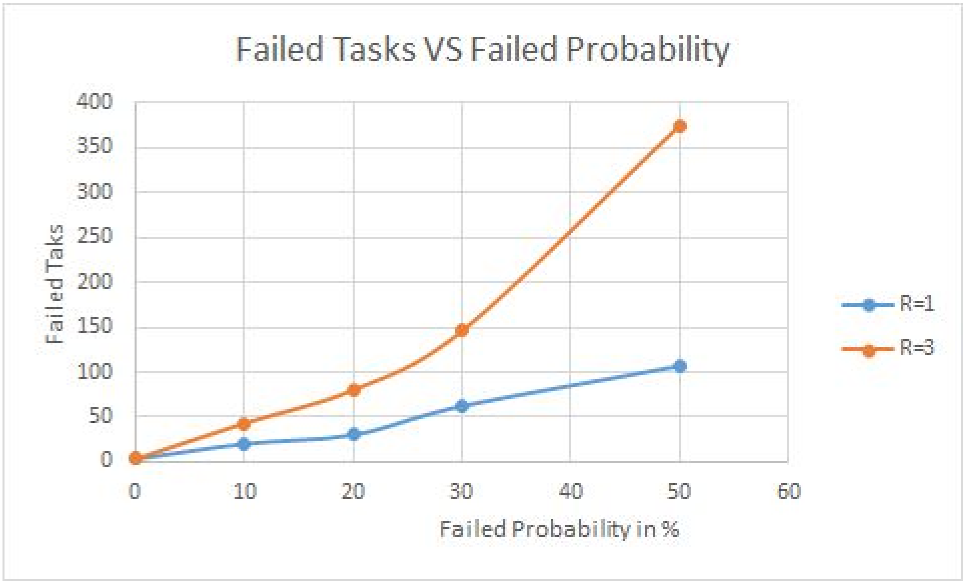
1) We run on experiment on 5 compute nodes on different servers and client connects with server. We varied chunk size and observed time. We have plotted this graph for different file sizes. We observed that in general as the chunk size increases, time decreases.



2) In this experiment we used Proactive Fault Tolerance. We take two replicas, for Replication = 1 and Replication =3. We used chunk size of 325852 and we tested our experiment on 6000000 file. We varied failed probability from 0%, 10%,20% and 50% and calculated time taken by client to get sorted file. We observed that in case of replication 1 as the failed probability changes, there is a minor change in time. However, in case of Replication =3, we get declining time trend. This is actually not expected but reason for this may be attributed to several factors such as network traffic, network load etc.

Also, we observed time taken by Replication =1 is less than Replication =3 which was expected. However, that gap decrease with increased failed probability.





Expected Results -

1) Time should decrease as the size of chunk increases, keeping other parameters constant
2) Time should increase as the size of file increases, keeping other parameters constant
3) Time should increase as the failed probability increases

4) Time should increase as the number of replication increases for low value of failed probability. As the failed probability increase, the gap between replication should decrease.

We observed that in general (from graph and from table in appendix), as the chunk size increases, time decreases. Also, as the number of replication increases, time increases. However, as the failed probability increases, this gap reduces. The plots do not completely comply with the expected results due to the following reasons. One of the main reason is varying network traffic.

Further experiments on similar test cases are performed :

	Failed Task	
Failed Probability	R=1	R=3
0	3	3
10	20	43
20	30	80
30	62	146
50	107	375
	Cancelled Task	
Failed Probability	R=3	
0	152	
10	134	
20	133	
30	168	
50	717	
	Total Task	
Failed Probability	R=1	R=3
0	117	345
10	134	385
20	144	416
30	176	488

50	220	717
	Time taken	
Failed Probability	R=1	R=3
0	80810	260451
10	83451	144296
20	102058	133154
30	85514	148338
50	83163	137938

REFERENCES :

1) Thrift RPC tutorial by Kwangsung Oh
2) Distributed Systems: Principles and Paradigms (2nd Edition)", by Andrew S. Tanenbaum and Maarten van Steen
3) Thrift Apache Documentation (https://thrift.apache.org)
4) Stack OverFlow

Appendix

For Replication =1

chunk size	File_Name	time
162926	200000	3246
325852	200000	2959
488778	200000	2492
651704	200000	2556
814630	200000	2068
162926	1000000	12713
325852	1000000	10628
488778	1000000	11799
651704	1000000	7379
814630	1000000	7444
162926	2000000	29399

325852	2000000	22230
488778	2000000	22708
651704	2000000	25329
814630	2000000	18588
162926	6000000	79565
325852	6000000	82342
488778	6000000	75132
651704	6000000	76541
814630	6000000	74316
162926	10000000	146879
325852	10000000	128366
488778	10000000	138517
651704	10000000	127443
814630	10000000	119928
162926	20000000	331975
325852	20000000	282154
488778	20000000	300910
651704	20000000	290455
814630	20000000	252151

For Failed Probability

Failed Probability	Replication =3	Replication =1
0	149576	82342
0.1	139962	79319
0.2	125356	80923
0.5	115231	80518