CS425 MP3 Group 41 Report

James Hsu *chsu41*, Zitong Chen *zitongc2*

Introduction:

In this machine problem, we implemented a simple distributed file system in python. In addition to the distributed query and the heartbeat group service functionalities, which were implemented in the previous machine problems, we integrated a distributed file system in the VM cluster modeling Cassandra.

Design:

Put Request: Upon receiving a put request at any node within the cluster, the server will determine where the file should be stored by querying a hash function. In particular, we simply hash the filename and then computes *hash_val mod x* (where x is the number of live nodes in the cluster) to determine the target storage location of a file. After a target node receives the file, it will send a copy of the file to its three successors. While this design eliminates the need for a master node and simplifies the storage target selection logic, it could induce an imbalanced distribution of workload among servers as a tradeoff.

Get Request: Upon receiving a get request, the server will query all the machines to see what the available servers are if itself does not contain the requested file. It will then fetch the file from an available server with the lowest id.

Delete Request: Upon receiving a delete request, the server will again, compute the name hashing algorithm and determine which server is the host. The host server will be notified, and it will delete the specified file from its storage while also notifying its three successors of the deletion.

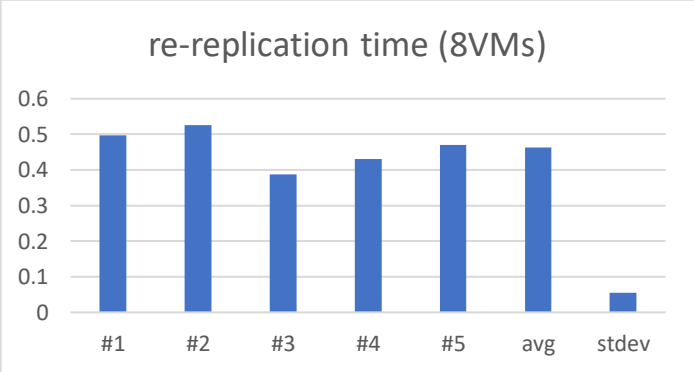
Ls and Store Request: Ls and store requests are handled similarly as the first part of get request. Store queries the local filesystem while ls queries the entire SDFS cluster.

Utilization of Distributed Querying (MP1) Functionality: The distributed grep we implemented in MP1 continued to be useful, as now we can not only search for specific patterns in log files, but also query the files within the SDFS to monitor the correctness of our file duplication process.

Evaluations (time units are in seconds, bandwidth Bps):

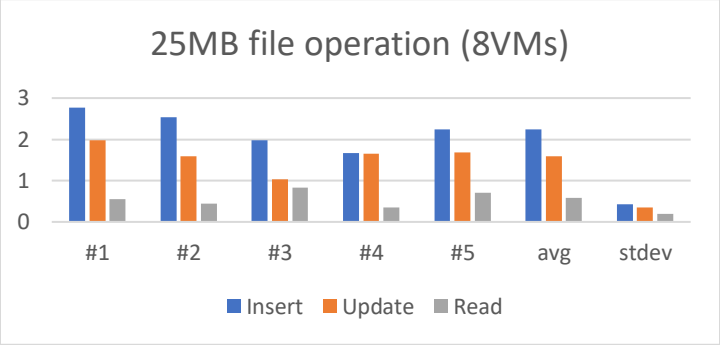
Below are is the data of the re-replication time for an 8VM cluster. As we can see, handling a 40MB file upon failure can be completed very quickly, within our membership list update period(1 sec). The bandwidth is the incurred by file transferring. Note that a 4 VM cluster would not need re-replication of file upon failure, since each file has 4 replicas in the SDFS.

trials	8VMs	bandwidth
#1	0.497	26570
#2	0.526	24961
#3	0.388	23724
#4	0.43	30649
#5	0.471	31558
avg	0.4624	27492.4
stdev	0.054546	3462.401

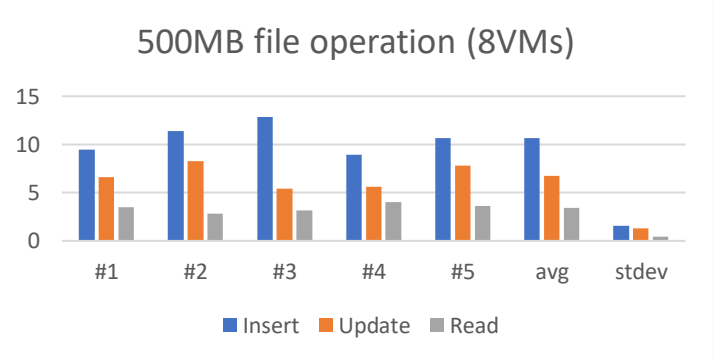


Below are the statistics for the 25 MB and 500 MB file operations. While read takes minimal time, notice that update takes slightly less time than insert because the servers already know which servers to send the updated replicas to. In terms of the relationship between different operations, both graphs exhibit similar patterns of comparison.

25MB	Insert	Update	Read
#1	2.766	1.982	0.557
#2	2.541	1.595	0.438
#3	1.985	1.032	0.831
#4	1.672	1.649	0.342
#5	2.237	1.68	0.709
avg	2.2402	1.5876	0.5754
stdev	0.434435	0.345183	0.198036



500MB	Insert	Update	Read
#1	9.49	6.58	3.51
#2	11.37	8.26	2.84
#3	12.86	5.43	3.17
#4	8.93	5.59	4.02
#5	10.65	7.78	3.64
avg	10.66	6.728	3.436
stdev	1.557723	1.270382	0.451143



Below are the statistics for transferring the English Wikipedia corpus file on the SDFS. Notice that the times between the two scenarios do not differ much, because the system only stores 4 replicas of the same file within the cluster.

trials	4 VMs	8VMs
#1	14.57	15.4
#2	14.08	12.94
#3	12.74	12.5
#4	14.04	14.35
#5	13.61	16.72
avg	13.808	14.382
stdev	0.687074	1.74104