CS 425 MP3 Report

Karan Sodhi (ksodhi2) & Nashwaan Ahmad (nahmad31)

Design:

For our SDFS implementation, we built off of our previous MP where nodes could join a group and each node held the full membership list. The first node that started the group was set as the original master server. The master server is responsible for deciding which files get stored where and stores a metadata hashmap containing the sdfs file names and which nodes they are replicated at.

When a node issues a PUT query, it sends a TCP request to the master server containing the sdfs filename, then the master server responds back with three randomly chosen server hostnames from the membership group, and the timestamp of when the server received the request. The node then simultaneously sends TCP requests containing the file contents, timestamp, and sdfs filename to the servers that the master responded back with. The nodes receiving the files store it in a directory containing all the sdfs files and the node that sent the file creates a local copy of the file in its sdfs directory. After a node successfully stores a copy of that file it responds back with an ack message to the node that issued the PUT. When **W = 3** acks are received the original node responds to the master server with an ack letting it know that the PUT was complete and prints out a message to the user letting it know that their put was successful. Then when the master server receives the ack letting it know that the put was complete, it stores what nodes the file was replicated at in its metadata hashmap.

Therefore, a total of **4 replicates** of the file are stored in the system, once locally on the server conducting the PUT and three times on other servers in the membership group. We chose to store 4 replicates of the file because it would allow SDFS to tolerate up to three simultaneous machine failures still allowing for one replicate to be available.

When a node issues a PUT query with a sdfs filename already known to the system in order to update the file, the same process as regular PUT happens except the master server will now respond back with four server hostnames where the file is already being stored, which the node will send requests to in order to write the update. Additionally, when a node receives the updated file it stores the filename and timestamp in a queue which holds all versions of that file sorted by the timestamp of when the master server received the PUT request. This allows the updates to be **Totally ordered.** When the queue is greater than 5, the oldest version of the file is deleted from the queue and disk.

When a node issues a GET or GET-VERSIONS query, it sends a TCP request containing the sdfs filename to the master server which responds back with the nodes holding the replicates of the requested file. Then the node simultaneously sends TCP

requests to all of the replicates asking for the latest timestamp of the file. When **R = 2** responses are received, the node that responded back with the greatest timestamp is sent an ack message notifying it to send the file or files(if GET_VERSIONS > 1) which are then stored on the requesting node. If both the nodes responded with the same timestamp and one of the nodes was the requesting node, it would be preferred because it would save on bandwidth.

For our Consistency levels, we used W=3 and R=2 because there are four replicas in total and W=3 would make a quorum, guaranteeing that two conflicting writes intersect in at least one replica. R=2 guarantees any write and read intersect in at least one replica. This also satisfies that W and R are less than the total number of replicas.

In order to delete a file, the node sends a TCP request containing the sdfs filename to the master server which responds back with the replicas of the requested file. The node then sends simultaneous TCP requests to the replicas asking them to delete the file and then responds back to the master with an ack to let it know to remove from its metadata.

Additionally, whenever a node left or failed in the system (not including the master node) the master node would look through its metadata to find what replica files that node was storing and then it would tell a node not holding the file to request to a node that is holding a replicate to send the file to it, this process would be similar to a GET request. This guarantees that four replicas of a file are always stored.

If the master node left or failed, the nodes go through an election process to elect a new master. First, whenever a node detects that the master node has failed it sets a variable “masterElected” to false which temporarily disallows file operations. Then if the node detects itself to be at the beginning of the membership list, it starts the election process by sending simultaneous TCP requests to the other nodes in the group asking for what replicates they store. It uses the responses to reconstruct the old metadata hashmap. Then it tries to rebuild the system into a good state by iterating through the metadata and seeing that if a file has less than 4 replicas go through a process as described in the last paragraph to replicate that file. When the rebuilding is complete it sends a TCP message to all the other nodes in the group letting them know that the election is complete and that they can start accepting file operations once again. Lastly, in order for new nodes to be able to contact the master to join the group, the DNS server, which in our case is represented as a file, should be updated to point at the new master.

Past MP Use:

Like stated above, MP2 was used to build the distributed membership list which helped to elect the master and to know what nodes to contact. MP2 was also used to detect

failures which helped to trigger the replication of nodes. Additionally, whenever a node put, deleted, got, or re-replicated a node, that information was logged in a file. That file was able to be queried by MP1 which helped us a lot when debugging.

Measurements:

i) Upon a failure, the average re-replication time of a 40 MB file was 5.004 seconds, with a standard deviation of .421, of that time on average 4.86 seconds was spent detecting the failure while it took only 144 ms to re-replicate the file on to another machine. Using iftop, the measured average peak bandwidth on a failure is 158 MB/s up and down for a node to send a replica of the file and for the new machine to receive said replica.

ii) On average, inserting a file of size 25 MB takes 166.8 ms with a standard deviation of 9.02 ms, while inserting a file of size 500 MB takes 4438.6 ms with a standard deviation of 168.64 ms.
On average, reading a file of size 25 MB takes 38.2 ms with a standard deviation of 1.17 ms, while reading a file of size 500 MB takes 2393.6 ms with a standard deviation of 123.26 ms.
On average, updating a file of size 25 MB takes 168.2 ms with a standard deviation of 10.19, while updating a file of size 500 MB takes 4879.6 ms with a standard deviation of 111.31 ms.
Inserting, reading, and updating files with size 500 MB took much longer than 25 MB which makes sense because the bigger the file is the more data would need to be sent through the network which takes more time. Additionally, inserting and updating files took a similar amount of time which also makes sense since both of them are implemented almost identically.

iii) The time it took to perform get-versions increased linearly as num-versions increased which makes sense because the more versions that are being requested the more data the node holding the replicate will have to send over the network. Additionally, the time to store the entire English Wikipedia corpus did not change much whether the num-machines in the system were 4 or 8. This also makes sense because in both cases storing the file would cause it to be replicated a total of 4 times no matter how many machines were in the system.