CS425, Distributed Systems: Fall 2023
Machine Programming 3 - Simple Distributed File System

Group No: 58 **Names (netId):** Risheek Rakshit S K(rrs7), Siddharth Lal (sl203)
Repository link: *https://gitlab.engr.illinois.edu/sl203/cs425-mp3-group-58.git*

The Simple Distributed File System (SDFS) is built using Golang and we have built on top of our MP-2, i.e. the gossip based membership list implementation. The first machine we start acts our introducer and when it crashes no new node can join the network, but the file operation can be carried on. While uploading a file for the first time on the SDFS, there are 'r' replicas created for that file on r different machines. The machine uploading that file becomes the leader for that file. We deploy a gRPC server on each machine to handle all file put, get and delete requests, and to send and receive replicas to/from various peer nodes. The gRPC server is run in a separate thread from the gossip threads.

Design:

1)Replication level or r: We have set the replication factor as 4 (i.e. r = 4) since as per the MP specs, there can be at-most 3 simultaneous failures and when we have this replication factor, the file stays in the system despite failures, ensuring availability. The locations of the replicas are selected by a method which uses a mixture of a hash function (sha256) to find some nodes and randomly selecting the remaining nodes from the active nodes.

2) Starvation avoidance: Any node that first uploads the file will be considered the leader and the leader for that file will maintain a queue through which all the read and write requests by other nodes are regulated. The queue does not allow a conflicting operation to wait for more than 4 consecutive operations. For example, the queue rearranges the requests in a way that if the top of the queue contains 6-R and then receives 1-W, it is reshuffled to be 4-R, 1-W, 2-R. Similar rearrangement happens when writes are at the top and a Read comes in. So when a process wants to read a file, it will enqueue its requests and will constantly keep checking the top of the queue with gRPC requests and if it finds that if it at the top of the queue, it access the file. Once the operation is complete it dequeues itself from the queue. As we allow two simultaneous reads, we check the top of the queue, if it is a read operation and the second element of the queue is also a read operation, then we allow both the read operations. We do not allow the write to happen if it is at the second of the queue with the top being a read. When a Write is at the top of the queue no other writes or reads are allowed. When we detect failures, we dequeue all the read or write requests posted by the failed node to prevent starvation of healthy requests.

3) Past MP use: We rely heavily on the MP-2 for failure detection, and we piggyback the crucial information of whichever file is stored in that particular node along with the list of files for which the node is the leader. We used MP-1 to check the logs and debug the code.

4) Leadership election: We do not maintain a single leader for the entire SDFS. Instead we have opted to use a per file leader approach in the system and the read and writes requests are performed on the replica present with the leader. This is similar to the passive replication discussed in the lecture. The leader maintains a list of nodes which has the remaining r-1 replicas and so when it detects failure of the nodes through the underlying gossips, it gets to know that it has to create a replica and figures it out from the membership list the node which doesn't not have the replica of the file and creates a replica on the node. Whenever a leader is failed, the node which marks a node as failed, checks whether it can be the leader by comparing its node Id with the node Id of other nodes which have the replica. If it has the highest node ID among these candidate nodes, then it sets itself as the leader else it doesn't do anything. In case the node detecting the failure is elected as the leader, it calls for the creation of new replicas to satisfy the minimum replica count. We have also handled the case when a potential leader may also fail by monitoring whether the files present on the 'to-be-deleted' potential leader have elected a leader, and if not we again elect the new leader and the leader would take care of creating the required amount of replicas.

Measurements:

1) **Overheads:**

We used five different files for each of the sizes 5MB, 10MB, 25MB, 50MB, 100MB, randomly crashed one of its replica and measured the time taken to create another replica. For every alternate run the leader node

for the file was crashed. The bandwidth for the re-replication was measured using IFTOP on the leader node (in case when leader was crashed we checked the bandwidth on the elected leader).

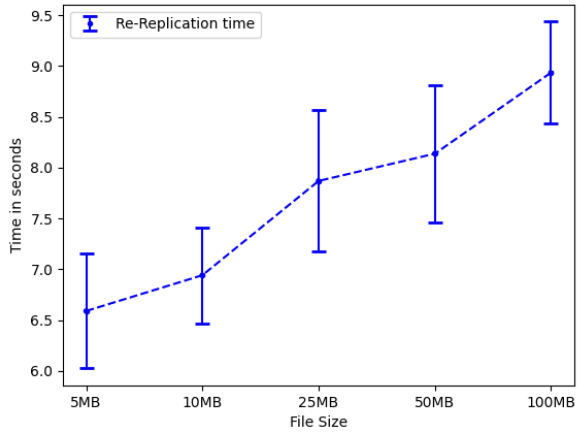

Fig 1: Re-Replication time (Overhead)

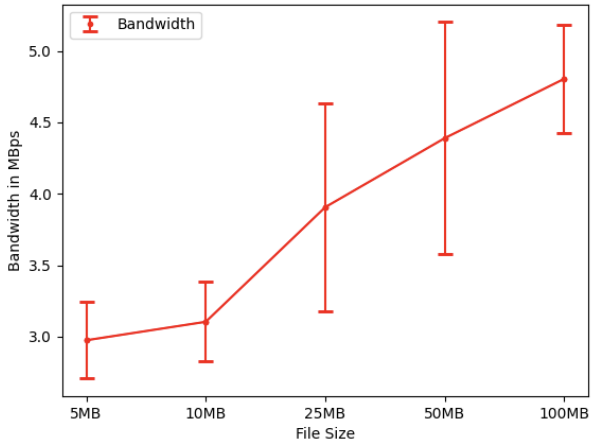

Fig 2: Bandwidth (Overhead)

As shown in Fig1 and Fig2 we can see that the time required for replication and the bandwidth used in the due process is directly proportional to the size of the file. As the size of the file increases, more data has to be sent across the network for successful completion of the replication and hance it is in accordance with our expectations.

2) Operation times:

We used five different files for each of the sizes 25MB and 500MB and recorded the times taken for each of the operations Read, Insert and Update.

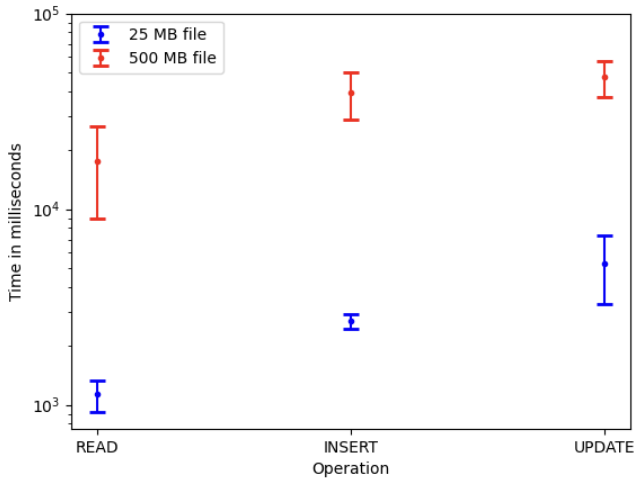

Fig 3: Operation times

Again here the time taken is in accordance with our expectation that time taken for any of the read write and insert operations are directly proportional to the size of the file involved (as shown in Fig 3), so any operation for 25MB takes less time than that for 500MB. And the read times are less than the other operations as it involves the time taken to obtain access to the file and then downloading from the respective node. Incase of insert and update time increases due to the operation of writing it to all the replica nodes.

3) Read-Wait:

For the purpose of calculating the time taken by the last reader to complete the reads, we have used a file of size 400 MB and it approximately takes 14 seconds for the read operation and 32 seconds for write operation. The test was conducted when there are m=1,2, 3,4 and 5 consecutive read operations behind an existing read. The results of this operation are shown in figure 4 and 5. And they are in accordance with our expectation that the time taken is proportional to the number of readers in the queue and since we allow two consecutive reads to take place we can observe that the time taken by two consecutive reads on the top of the queue take the same time. When m=1, the queue has the primary read on top followed by a read. Both take the same time and when they are finished and if m is higher the next two are pushed to the top so they take similar times. We calculate overhead by dividing the actual time with the expected time. The higher overhead for higher m can be accounted for the fact that since we do passive replication, we are increasing the load on the leader of the file which has queued requests, hence causing network congestion and subsequently higher read times.

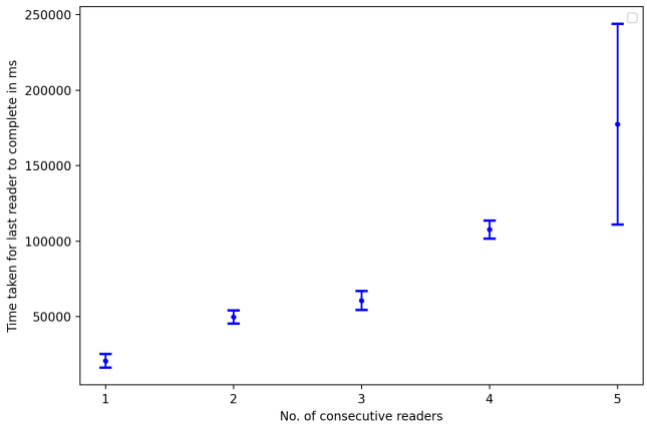

Fig 4: Time (Read-wait)

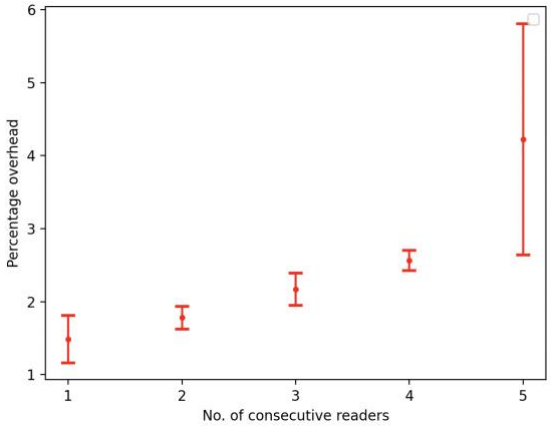

Fig 5: Percentage Overhead (Read- Wait)

4) Write-Read:

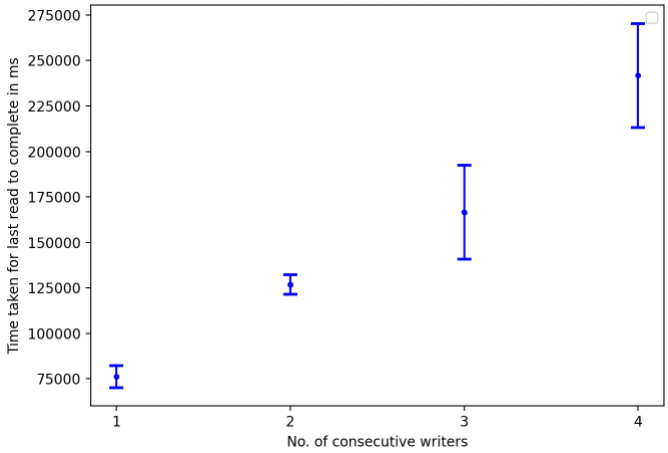

Fig 6: Time (Write-Read)

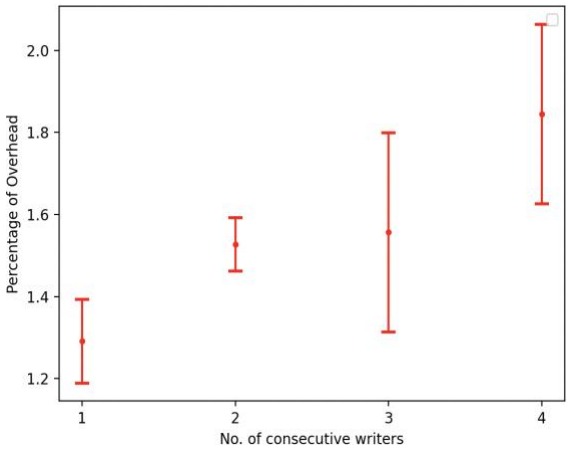

Fig 7: Percentage Overhead (Write-Read)

For the purpose of calculating the time taken by the last reader to complete after consecutive rights, we have used a file of size 250 MB and it approximately takes 11 seconds for the read operation and 24 seconds for write operation. The results shown in figure 6 and 7 are in accordance with our expectations that time taken for last read to complete increases as the number of prior writes increases. Since we allow only a maximum of 1 write operation to take place and there are no concurrent operations, the time has to increase. The higher overhead involved here is also accounted by the fact that we use passive replication similar to what we saw in Read-Wait.

5) Large Dataset

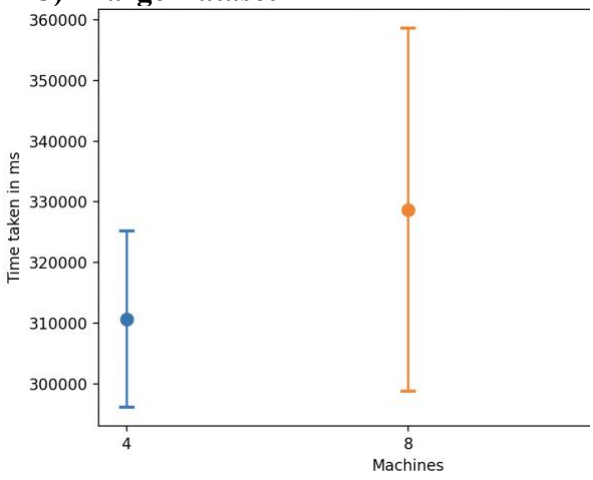

Fig 8: Time (Large Dataset)

Figure 8 denotes the time taken for the Wikipedia English corpus in zipped format (raw.en.tgz) to put (upload) in an SDFS with 4 machines and 8 machines respectively. The file size is 1.3 GB. One can observe that the time taken for upload in both these settings is almost similar. This can be attributed to the fact that the way we choose replicas locations is independent of how many machines are there in the network. Also, the slight increase we notice for the 8-machine case may be attributed to the fact that a larger network may lead to increased congestion due to gossip. But this difference will be small, as seen, compared to the upload times.