Report

Our SDFS filesystem is based on project2, the membership system. Every member in membership list is an active node. Whenever membership changes, fault tolerance procedures would start. There is one master controlling the task allocation. To ensure the total ordering of all updates to a given file, the master would block any further request for the same file until the previous put request on the same file is finished. We followed the quorum rule described in lecture. Each file has four replicas, each of which is stored in a different node (when there are at least). For a put request, the writes needs ack-ed by four nodes. For a read, the master would randomly pick any node holding the specified file. So W needs N ack, R needs 1 ack. In case of failure, the master would detect membership list change, and check which files are on the failed node. Then it would issue other nodes holding the replicas of files on the failed node to forward their replica and all versions of it to a new node. After the recipient node finished saving a transferred file replica it would send master an acknowledgement. The master considers recovery as finished only after receiving all required acknowledgements. The master would then send affected nodes a complete list of information about which node is holding which file. Nodes does not need to manage it because multiple simultaneous failure can happen at the same time. During this process master would block all incoming requests. For version control, each version of a file is stored as a new file with "-version" appended to the end of its name. When there are more version than 5, the oldest version will be deleted.

We used MP1 to figure out what happened on each node in case of failure recovery and it turns out to be quite useful. It's nice and easy to view all of what happened in one window.

1. Time and bandwidth upon failure when there is a 40MB file stored in the system and there are 5 nodes available. Measured for 6 times

Time(S):	2.7	3.6	2.4	3.3	6.5	2.0
Bandwidth(MB):	8.9	9.3	8.7	7.2	8.2	8.5

The bandwidth is a little too low comparing to our estimation, since about 40MB of data needs to be copied to a new node. However, it's very hard to capture the exact bandwidth usage since there is large amount I/O mixed inside the entire execution. Thus such a low bandwidth usage is also a rational result.

2. insert time(s)for 25 MB file 7.01816105843s

   Each file needs to be stored on 4 nodes, with the heavy I/O and lots of socket communication the time cost is understandable.

   Update time(s) for 25 MB file:7.98851799965

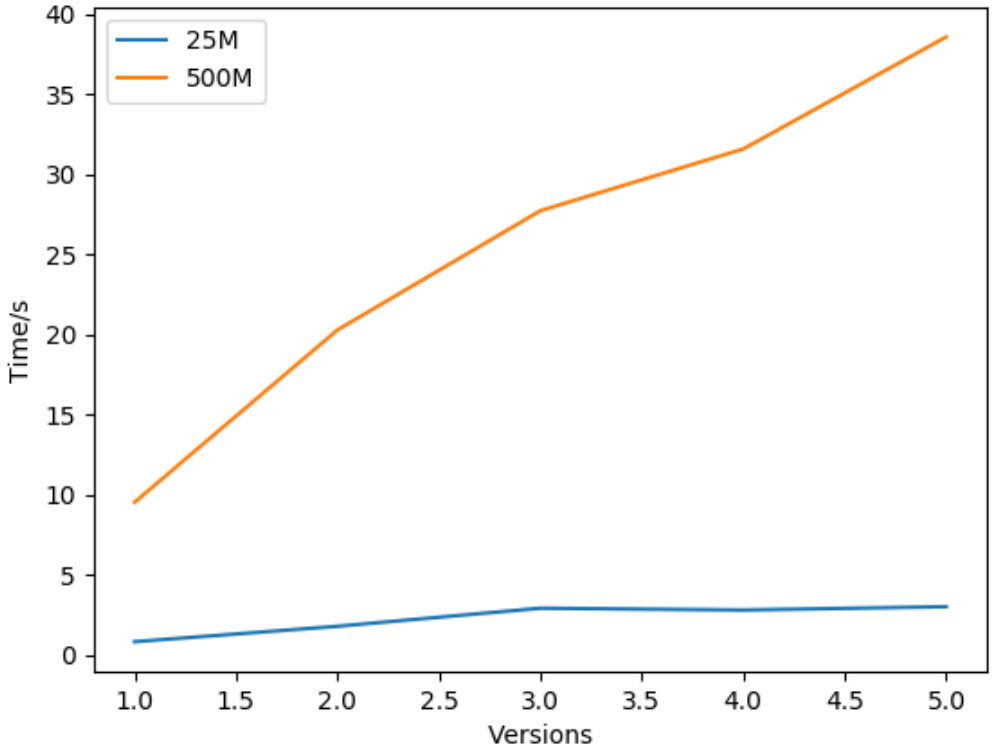
   For each update, in our implementation it's just a put with a different file name, so time is similar to insertion.

   Read time(s) for 25MB file :0.67107796669

   Read time is much faster, which is expected since W needs 4 ack, R only needs 1. Only one copy of the file is needed and there are less TCP connection needed to establish.

insert time(s) for 500MB:88.25729138154561 Update time for 500MB: 84.90341122091783s, read time for 500MB:12.3763361309s
Trend is about the same as situation for 25MB insertion

3.



This is expected. The time should go up linearly with the amount of versions get. Since 25MB file is too small comparing to 500MB, it might not be obvious here but it also goes up linearly

4.Inserting Wikipedia English raw text into SDFS with four machines active(Unit:s) five trials:1046.95, 1047.20, 1022.11, 1007.1380278800204, 1017.58

Inserting Wikipedia English raw text into SDFS LSwith eight machines active(Unit: s): Five Trials:1025.66,1050.87,1020.02,1025.72, 1058.76.