MP3 Report (SDFS - Simple Distributed File System)

Powei Wang (poweiww2) and Krishna Phani Datta Chivukula(kc62)

a) Design: For the system to be fault tolerant we need at least 4 replicas as 3 simultaneous failures are possible. We used a quorum with W=3 and R=2, thus W+R = 5. Each read needs to read from 2 replicas as we want to make sure that all the writes to the SDFS files are up to date. We used the same failure detector as MP2. For leader election, ring election protocol was employed. We used passive replication as there is a primary coordinator for a more responsive system. In our design both the coordinator and introducer are the same. To ensure total ordering of messages, a sequencer based approach was used where the coordinator maintains the global counter of all the file versions thus ensuring total ordering.

We have implemented the functionalities of SDFS as follows:

Get: The get request is received by the coordinator from a client machine and then the coordinator checks for (file location,latest version) in its hashmap and waits until the coordinator gets a quorum of ACKs and sends the file to the client machine

Put: The put request first reaches the coordinator from the client machine and the coordinator then puts the file in the 4 randomly chosen nodes, adds these 4 nodes information in coordinator's hashmap and acknowledges the client machine once the coordinator gets a quorum of ACKs from the chosen machines

Delete: The delete request reaches the coordinator from the client machine and the coordinator deletes the file in the replicas containing it and if the coordinator gets a quorum of ACKs then the coordinator acknowledges the client machine

ls: This command lists all VM addresses where the file is currently stored with all the versions of the corresponding file

store: This command lists all the files currently being stored at this machine

Replication: Once a replica failure is detected the coordinator will replicate the contents of the failed replica to the remaining active replicas thus ensuring 4 replicas always, as 3 simultaneous failures are allowed.

File message format: The file message contains *file_name*, *file_size* and *file_version* as shown

```
25      public FileMsg(
26              String fileName,
27              int fileSize,
28              int version) {
29          this.name = fileName;
30          this.size = fileSize;
31          this.version = version;
32      }
```

b) Use of MP1 and MP2: As mentioned above we reused MP2 code for underlying failure detection. We used MP1 code to debug the log lines. MP1 was very useful as we can easily grep the logs onto a single machine instead of handling logs on multiple machines and debug where the code has gone wrong.

c) Measurements:

(i) Re-replication time and bandwidth upon a failure for a 40MB file

We used ***iftop*** (command line tool) to obtain network traffic and bandwidth usage details

	Mean	Standard deviation
Re-replication time (sec)	1.2832	0.29516
Bandwidth upon a failure (MBps)	42.1829	6.0356

The bandwidth is somewhat above 40MBps in case of a failure and the re-replication time is slightly above 1 sec, as additional pings and ACKs due to failure, contribute to the bandwidth being greater than 40MB.

(ii) Time to insert, read and update for files of size 25MB and 500MB (total 6 data points) under no failure. The bars that are not visible as they are close to zero. (Please refer to table values)

From Fig 1 the mean time to **insert** and **update** in case of **25 MB** file is almost the same and the mean time to **read 25 MB** file is less than both insert and update.

The difference becomes more clear for **500MB** file where the mean time to **insert** is the highest, to **update** is second highest and mean time to **read** is the least

Mean time to insert > update > read is the expected trend as inserting inserts the file for the first time and updating updates the existing files on the nodes. So in case of update the file need not be inserted into the 4 randomly chosen nodes which contributes to the additional latency in case of insert

	Insert (sec)		Read (sec)		Update (sec)	
File size	25MB	500MB	25MB	500MB	25MB	500MB
Mean	0.454211	20.48392	0.151707	5.24194	0.488628	15.82932
Std dev	0.035627	3.48226	0.078456	1.00058	0.019792	5.41792

Fig 1: Time to insert, read and update files of 25MB and 500MB

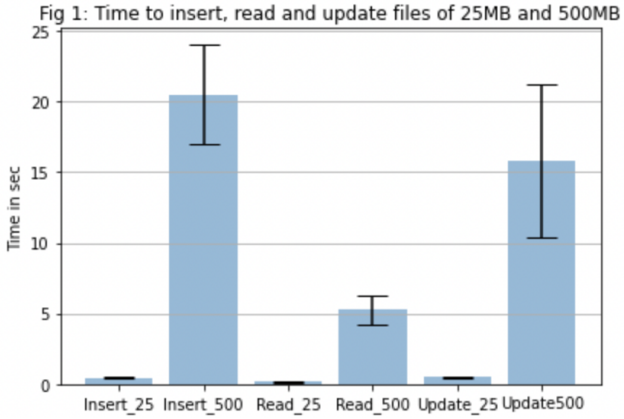
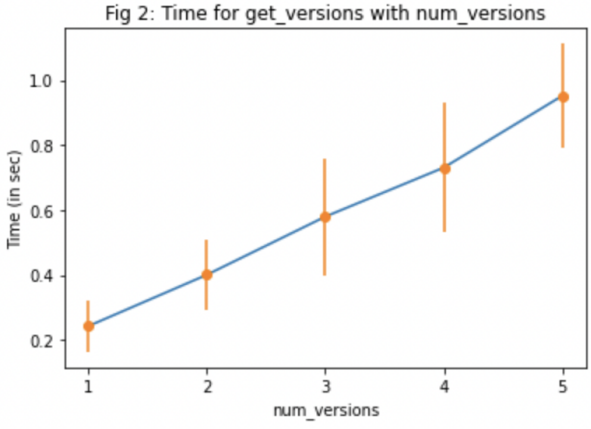


Fig 2: Time for get_versions with num_versions



(iii) Time for ***get_versions*** as a function of ***num_versions***

From Fig 2 the mean time to ***get_versions*** increases linearly with the ***num_versions***. The file size in consideration is 25MB. The standard deviation on the other hand doesn't change much as the same node is being queried for the versions 1 to 5

num_versions	1	2	3	4	5
Mean (sec)	0.24303	0.40127	0.58009	0.73103	0.95282
Std dev (sec)	0.04202	0.06038	0.10323	0.12888	0.15090

(iv) Time to store the entire English Wikipedia corpus into SDFS with 4 and 8 machines is given below. Here the file is English raw text file is of size 3.48GB

Number of machines	Average time to insert (sec)	Standard deviation (sec)
4	81.88517	8.92540
8	73.95335	5.82749

As we can see from Fig 3, the time to insert is almost same in both cases (4 and 8 machines) as the number of replicas is constant i.e. 4 replicas and it should take similar amount of time in both cases and the results are as per the expectations