Jiachen Yuan & Sizheng Zhang 10/25/23

CS-425 MP3 Report

Design & Algorithm Overview

The high-level design of our system is to have an elected leader server that processes all the GET/PUT requests from other servers. The leader keeps an in-memory record of where all files are stored, and it has a locking mechanism (which will be covered later) to avoid conflicting read/write operations and starvation. Here is the details for each command:

PUT: When a client wants to put a file to SDFS, it will first send a request to the leader inquiring which VM it can store the file on. After receiving the request from the client, the leader will check the lock for that file to see if the client can proceed. If yes, it will return a list of VMs to the client and if no, it will tell the client to wait by spinning. After receiving the VM list from the leader, the client will use scp command to copy its local file to the designated directory on the target VMs. After completing the file copy operation, it will send an ACK message to the leader server to let it release its write lock for the file.

GET: Similar to PUT, the client will send a request to the leader to get a list of VMs that store the designated file. Upon receiving the list from the leader server, the client will iterate over the list to request for the file until at least one of the requests is successful. It will then send an ACK message to the leader server to release its read lock.

DELETE: The client will send a delete message to the leader server to remove the in-memory record of the file. This operation will not actually delete the file and its replicas, but since its record at the leader has been erased, no clients will be able to read/write them.

LS: The client will send a request to the leader to ask for which VMs store the designated file.

STORE: The client will send a request to the leader to ask for which files are stored in itself.

A naive hardcoded leader node would represent a **single point of failure**. To avoid it, we implemented the Leader Election algorithm of Raft as a background process. Whenever the current leader fails, a new leader will be elected very quickly. The leader's in-memory state information and file metadata are replicated and piggybacked through gossipping from MP2 messages, so whenever a leader comes up, it can immediately take over the situation because of the up-to-date leader states it already possesses.

In order to **avoid starvation** for both reads and writes, The leader server will maintain two FIFO queues for each file, one for GET requests and one for PUT requests. Incoming GET requests will wait in the queue until there is no write and less than two reads going on for that file. When the number of consecutive reads goes over 4 and the write queue is not empty, the tasks in the read queue will not be processed until a write request is processed to reset the consecutive reads. The write queue will take a similar approach to ensure there is no starvation.

To tolerate 3 simultaneous failures, our SDFS will **store 4 replicas of each file on 4 different VMs**. We have implemented another background process on the leader that, upon detecting file server failures, the leader quickly recognizes which files' replicas are stored on those particular machines and then instructs live servers with the same replicas to re-replicate a copy to other available servers until we have in total 4 replicas for each of those files.

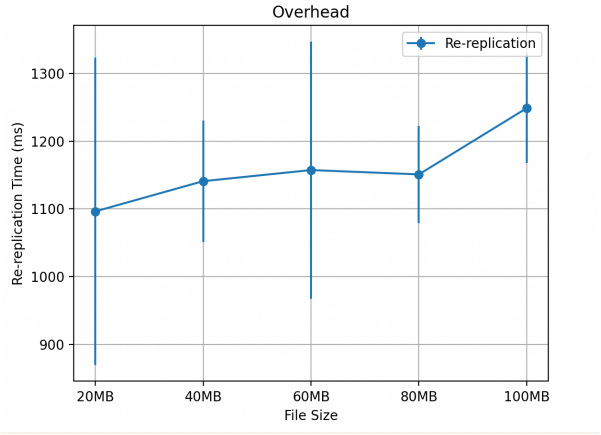
Use of Previous MPs

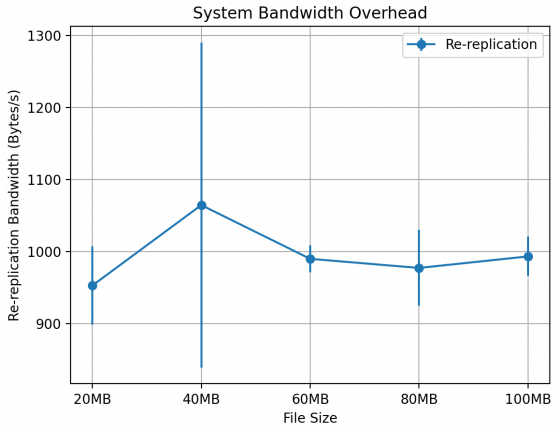
We used MP2 for file re-replication on node failure, leader election, and leader state replication. We also used MP1's log for debugging, especially for functionalities introduced by node failures like replication, leader election and mutual exclusion.

Measurements

i) Overheads

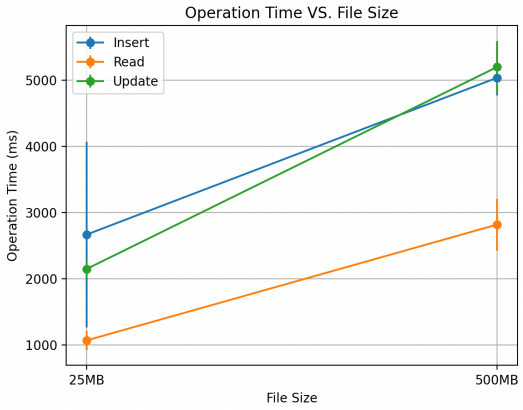
The re-replication time with respect to file size (the graph on the left) shows a linearly increasing trend, which fits our expectation since larger files will require more time to transfer. For bandwidth overhead, we only plotted the bandwidth overhead introduced by our SDFS system because file sizes are huge compared to our system overhead and if we include file sizes when calculating bandwidth, we cannot clearly see the overhead introduced by our system. From the graph on the right, we can see that the system bandwidth overhead is mostly consistent across different file sizes because no matter what the file size is, the mechanism and the number of messages sent by SDFS to assist the replication (not including actually transferring the file) is the same. The system bandwidth overhead for the 40MB file is large, but since its standard deviation is also large, we can assume that it was due to network fluctuations.





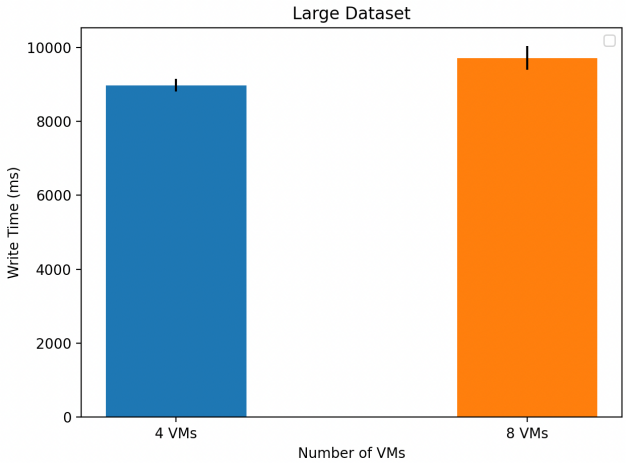
ii) Op times

For files of both sizes, the insert time and update time are basically the same because we used SCP commands to transfer files and make replicas. The read time for both sizes is substantially smaller than the write time, because for read we only need to transfer a file from one VM, while for insert/update we need to transfer the file to 4 VMs. Even though we started the SCP commands concurrently, the transfer speed is bounded by the maximum network bandwidth. Hence, the read time is substantially smaller than that of write time.



iii) Large dataset

The average write time of the large dataset with 4 VMs and 8 VMs are quite close to each other, each with an average around 9000ms. This is because even with 8 VMs available in the cluster, we only replicate the file 4 times when we are writing a file to SDFS. Therefore, the write time should be similar. It is worth noting

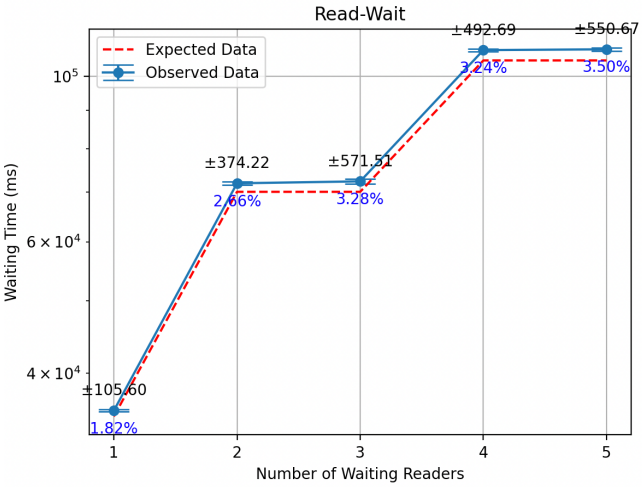


that the write time with 8 VMs is slightly higher than that with 4 VMs, and we suspect that it is due to measurement errors and variances in network conditions.

*******For question iv) and v), we limited our file-transfer bandwidth to be 30MB/s to satisfy the requirement of "picking a large file that takes at least 30s to read/write". If we do not limit our bandwidth, we would need to use a 15GB file, which goes beyond the storage limit of our VMs.*******

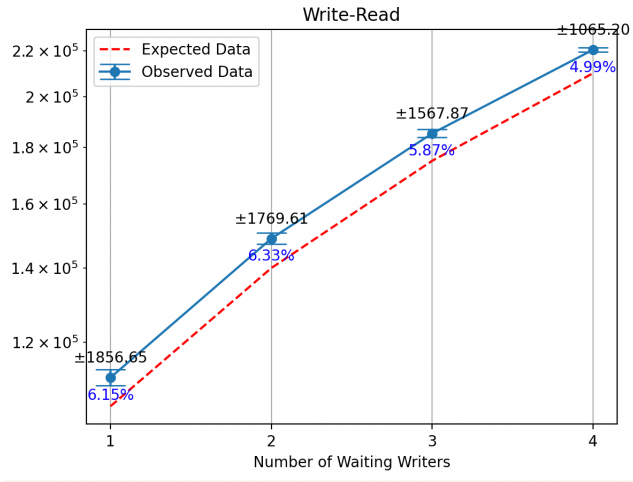
iv) Read-Wait

The file size of this experiment is 1000 MB, and the typical read and write time are both around 35000 ms (35s).[1] We can see that as the number of waiting readers increases, the waiting time also tends to increase. However, the relationship between them is not linear, because our system allows at most two reads at the same time. Therefore, the waiting time will increase when the number of waiting readers reaches an even number. The actual observed data's trend (blue line) is consistent with the expected data (red dotted line), with a time overhead of around 3%, which is likely to be caused by the internal communication in the SDFS.



v) Write-Read

The file size of this experiment is 1000 MB, and the typical read and write time are both around 35000 ms (35s). We can see that as the number of waiting writers increases, the waiting time also tends to increase in a linear way. This is because write-write and write-read are conflicting operations, and they should exhibit the behavior of serial operations. The actual observed data's trend (blue line) is consistent with the expected data (red dotted line), with a time overhead of around 5%, which is likely to be caused by the internal communication in the SDFS.



[1] Different from experiment ii, the read time and write time for iv and v are quite similar. This is because we limited the bandwidth in iv and v, so the 4 concurrent puts will not take up maximum network bandwidth, and it will have the similar execution time as one scp transfer (read).