CS425 MP3 – Simple Distributed File System

Group 04: Hsin-Yu Huang (hyhuang3), Yun-Liang Huang (ylh2)

Design and Algorithms

Introduction

We implement the Simple Distributed File System (SDFS) in C++ with six file operations, and the system is able to tolerate up to three simultaneous machine failures. The file system is built upon the Membership System we built in MP2, and we extend the Membership System by adding a leader role and Leader Selection algorithm. The leader handles centralized decisions, such as increment the sequence number and generate the replica list for a file. However, every member in the group is designed to accept requests from the client, and it will follow the Read/Write Flow to complete client requests. Moreover, we design the Fault Recovery Flow to ensure the system could tolerate failures.

Filesystem Status

Global statuses are written-only by the leader, and will be piggybacked and spreaded with the membership ping messages. Global statuses are readable by all members. We spread the global statuses to all members in case the leader fails. On the other hand, local statuses are stored locally on each server.
Global status:

1. FileMetadataMap:
    a. It is a hashmap maintaining the global file metadata. For each file, it maintains two sets: (1) replica server ids, and (2) the versions being committed to the file system.
2. SequenceNumber:
    a. It is used to ensure total order for all updates to a given file.
    b. Leader increments the SequenceNumber when it receives an insert/update/delete/commit to a file, or replicates files of a failed server.

Local status:

1. FileMap:
    a. For each file, this structure maps each version to the filename on the server
    b. Each server writes its FileMap when there is an insert/update/delete to a file on the server, and reads FileMap when it needs to find the filename on the server.

Number of Replicas

Number of replicas is 4. SDFS with four replicas is tolerant up to three simultaneous machine failures. There is always at least one replica for every file on non-faulty servers if three servers fail simultaneously.

Quorum Sizes

- R = 3. In the case of two replica servers failing simultaneously, enable immediately read file. (Demo Test 5)
- W = 3. Two conflicting writes intersect in at least one replica.
- R+W > number of replicas. Any write and read intersect in at least one replica.

Leader Selection

Every member in the group will set the member with minimum id as the leader of the group. We assume the membership system is reliable and every member in the group knows the full membership list. If the current leader crashes or leaves the group, some members will detect it and update the membership list. The new membership list will be spread by the ping messages we implemented in the MP2, so all members will get the up-to-date membership list. Therefore, every member could know who is the leader server asynchronously.

Read/Write Flow

Clients could send requests to any member in the group, and we call the server receiving the client request as initiator.

For write operations, the initiator will ask the sequence number and the replica list of the file from the leader. The leader will randomly generate a replica list if the file is not found in the FileMetadataMap. After the initiator gets the replica list and the sequence number, it will start the replication process. The initiator will send the file to the servers on the replica list. After all of the replica servers successfully receive the files and response ACK, the initiator server will ask the leader to commit the file with that sequence number. Notice that the replicated files on each replica server will stay uncommit until the leader commits the file.

For read operations, the initiator will check the FileMetadataMap first. If the file is not found, the initiator will respond "file does not exist" to the client. Otherwise, the initiator will look up the FileMetadataMap to get replica servers storing this file and the latest committed version. Then it uses a read quorum to ask file sizes of the latest committed version file from any three of the replica servers. We assume that the latest written and acknowledged file has the maximum file size among that quorum (if a replica server is failed, its file size is 0). Finally, the initiator fetches the file from that replica server with maximum file size and sends it back to the client.

For read versions operations, the initiator does the same process of reading. Instead, it fetches num-versions committed files from the replica server and sends those files back to the client.

Fault Recovery Flow

Once a server finds itself becoming the leader of the group, it will create a dedicated failure recovery thread. With the failure recovery thread, the leader will periodically collect recently failed members and collect the files replicated on these servers. For each of the files, the leader will replace the failed replicas in the replica list and request one of the non-faulty replica servers to start the re-replication process. In the re-replication process, the non-faulty server will send the file, including all committed versions, to other servers in the replica list. After all of the replica servers successfully receive the files, the non-faulty server will request the leader to commit the file with the new replica list. After all the files of a failed server have been re-replicated, the leader will mark that server as recovered.

Past MP Use

We rely on the Membership System built in MP2 to spread global status of the file system, and we add a leader role into the Membership System. We also modify the successor selection algorithm to make the system more reliable. The member list is now saved in a sorted set, and the successor of a server to be pinged is the next 3 servers in the sorted order. Additionally, we also use MP1 to grep keywords for debugging purposes.

Measurements

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

Average re-replication time	0.21 s (std: 0.028)
Average bandwidth	190.48 Mbps

We expect the re-replication time of a file should be close to the execution time of a read operation of that file since the most part of these two operations are sending file data. The measurement result matches our expectation, where the re-replication time is similar to the read operation of the same file.

- **Times to insert, read, and update, file of size 25 MB, 500 MB (6 total data points), under no failure**

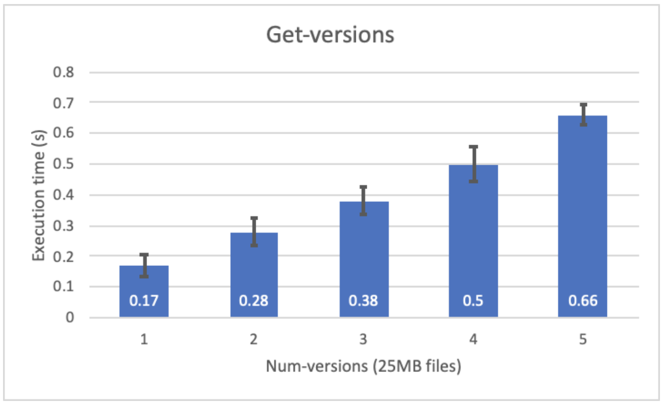
File size	Operations	Average time (s)	Standard deviation

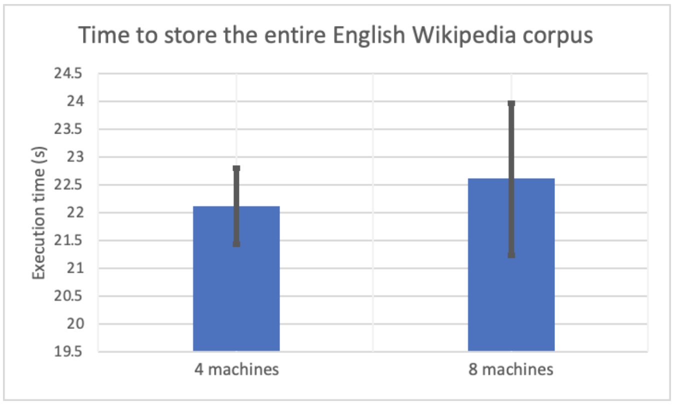
25MB	insert	0.19	0.026
	read	0.17	0.036
	update	0.19	0.018
500MB	insert	5.68	0.696
	read	2.27	0.432
	update	6.58	0.563

We expect read costs less execution time than insert and update since a read operation will ask file sizes from three replica servers and only fetch from one server with maximum file size according to our design. On the contrary, a write operation (insert/update) will send replica files completely to other replica servers, which costs more execution time.

The measurement result matches our expectation. The execution time difference between read and write operations increases when file size is larger, which also matches our assumption that there is more data to be transferred.

- **Plot the time to perform get-versions as a function of num-versions and the time to store the entire English Wikipedia corpus into SDFS with 4 machines and 8 machines**





Get-versions

We expect there is more execution time when num-versions increase since it needs to transfer more data. The measurement result matches our expectation. Higher num-versions costs more execution time.

Wikipedia corpus

We expect there should be no difference between uploading the corpus to a cluster with 4 machines and 8 machines, because the number of replicas is the same regardless of how many machines are in the system. The result matches our expectation. Though 8 machines have a higher uploading time, we think it is due to the sample size being quite small.