CS425 - MP3 - Simple Distributed File System

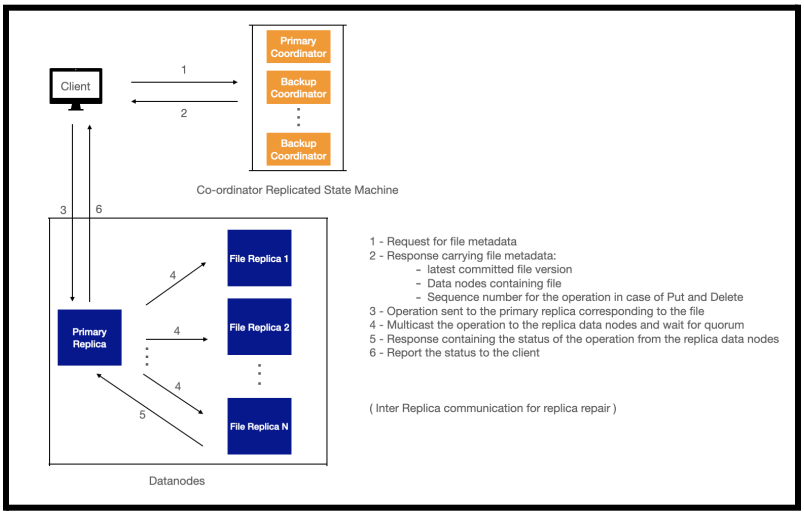
Lavanya Ramkumar (lr15)
Shahid Ikram (shahidi3)

Design

The file system we designed is inspired by certain design propositions we found interesting in GFS, its implementation by Yahoo, open sourced as HDFS; and Corfu. A general view of the high level architecture of the system is presented below.

Components of the SDFS:

- Coordinator Replicated State Machine
  - Stores metadata information of the files in the system
  - Sequences operations per file
  - Initiates replica repairs
- Data Nodes:
  - Stores the actual file data
  - Types (per-file):
    - Primary replica
    - Backup replicas



Ordering of operations: We support a total ordering on the operations (create, update and delete) performed per file. Operations on different files are allowed to proceed concurrently. The coordinator acts as a global sequencer. When multiple operations on a particular file are multi-casted to the replica nodes, we need to ensure that the operations that modify a file are carried out in the same order so that all the replicas are in sync. This is ensured by maintaining a local sequence number per file on each of the replica nodes for that file. The get operation on a file however, can happen concurrently with the writes.

Coordinator RSM: A pool of 4 nodes to tolerate 3 simultaneous failures. Responsibilities of a coordinator process include: Hosting information pertaining to files (Metadata server); Allocating datanodes on file creation; Sequencing of operations; Handling replica-recovery/re-replication for the files in case of replica failures; Synchronization of the coordinator state across all the backup coordinators

Data Replication and Quorum: Each file is replicated 5 times. We have configured the write quorum to 4 nodes, and a read quorum to 2 nodes which ensures that read and write quorum intersect in at least one node. Since the system should withstand 3 simultaneous failures, a write quorum of 4 nodes would ensure that we still have the one node with the latest committed version which can be used for replica repair for that file. Since read intersects in at least one node, we are guaranteed that we would get the latest committed version every time.

Re-Replication (Replica Recovery): The coordinator checks if all the replicas allocated for a file are active at repeated intervals and initiates a recovery protocol if it is not. The coordinator allocates a new node for that file and picks an active replica node for that file that can help the new node get data. The coordinator sends a request to the new node asking it to fetch data and state from the active replica. The new node then pulls data from the active replica which streams all the versions of that file to it and also sends the local sequence number for that file and other state information to ensure that this replica is in sync with the replica set for that file.

Operations

- **put (Create and Update)**
  - Client initiates a request for `put` to the coordinator which allocates data nodes for that file if it's a new file, or gets the allocated nodes for that file from its store, if it's an update and then sequences that operation and returns the datanode information and the sequence number to the client.
  - Client sends a request for `put` to the primary replica datanode whose operation will be stalled until the local sequence number on the primary replica matches the sequence number assigned for that put.
  - The file data will be streamed to the primary replica in chunks of 4MB and the whole file data is buffered until a commit is issued for the file from the client. On receipt of all the chunks of the file, it streams the data to all the backup replicas who in turn buffer the file till a commit is issued.
  - The primary replica waits for a write quorum and then sends an acknowledgment to the client.
  - The client then issues a commit to save the buffer contents into a file and increment the local sequence number so that the other operations on that file are allowed to make progress.
  - The client then informs the coordinator to bump the committed version of that file.
  - If the primary node times out waiting for a commit message from the client, it assumes that the client has failed and discards the file operation, deletes the buffer contents, increases the local sequence number and informs all the other replicas to do the same.
  - In case of failures, the put operation is retried 3 times.
- **get and get-versions**
  - The client fetches the allocated nodes and latest committed version for that file from the coordinator.
  - The client contacts the primary replica which checks if there is a read quorum that has the requested version. If not, get returns unsuccessfully. If quorum is attained, the primary replica streams the file data to the client in chunks of 4MB.
  - This operation streams all the latest ‘n’ versions of the file as requested or all the versions if ‘n’ is larger than what exists.
- **delete**
  - Client fetches the allocated nodes for the file and the operation sequence number from the coordinator.
  - The client then issues the `delete` request to the primary replica which multicasts the operation to all the replicas and waits for a write quorum. If quorum is achieved, the primary sends a positive acknowledgement to the client.
  - The client issues a commit message to the replica nodes to commit the delete which is when the file is actually deleted, local sequence number is increased and operation succeeds.
  - If the primary node times out waiting for a commit message from the client, it assumes that the client has failed and discards the file operation, increases the local sequence number and informs all the other replicas to do the same.
- **ls and store**
  - [ls] The client gets the nodes at which the file resides from the coordinator.
  - [store] This data node checks all the sdfs files that are stored in its local directory.

Past MP Use

The Membership list is being used in multiple scenarios to support our file system

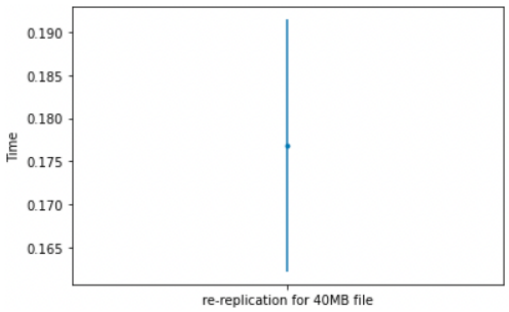
- The coordinator uses the membership list to **allocate data nodes** that are `Alive` for file creation.
- The **coordinator election** is carried out with the help of the membership list. All the potential coordinators in the list are marked. Since we have a full membership list consistently ordered, the first marked node in the list is always the active coordinator.

- The membership list is also used for **replica recovery(re-replication)** in case of node failures. The coordinator scans all the files in the system at intervals and uses the membership list to check if all the nodes allocated to it are active and if it is not, then it allocates a new node for that file and initiates recovery.
- The distributed log querier that was built in MP1 is now integrated to **search the log file generated during the current execution**. This was used in debugging issues on the VM. We used the querier to find all the log files that had some matches for a particular phrase in the log and then used that information to copy those log files to the local machine for debugging.

Measurements: (All the Times are in seconds)

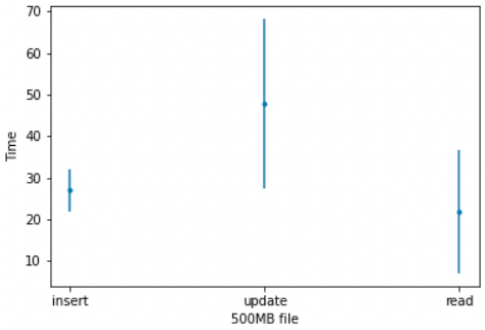
i) Re-replication time and bandwidth for 40M file

We see that, for a 40M file, on failing of nodes, the re-replication on each of the new nodes takes 0.7 seconds on average. The bandwidth for re-replication of a 40M file would be 57 MB per sec. This bandwidth conforms to the times seen for insertion of 25M and 500M files.



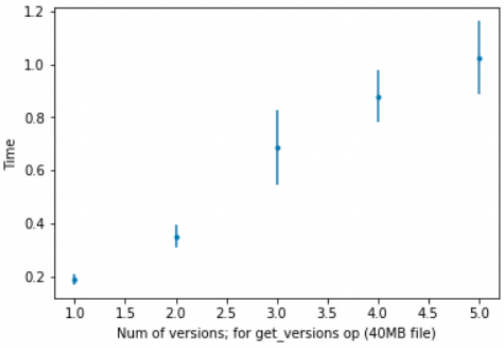
ii) Times to insert, read, and update, file of size 25 MB, 500 MB (under no failure)

Time taken for file insertion and updation are comparable. In the case of a large file the updation is taking longer due to the increased load in the system. However in both the cases, reads are prominently faster. This is because writes and updates include the time taken to replicate the data whereas in the case of read we just need to check if a quorum of nodes contains the version requested and the actual movement of data is just once across the wire.



iii) Time to perform get-versions as a function of num-versions

For a 40M file, we observed that as the number of versions requested increases, the time taken to fetch the file increases linearly which is an expected behavior as the amount of data to be streamed increases and hence time taken increases.



iv) Time to write wiki corpus

The insert of the wiki corpus (1.3G) took ~6.7 minutes to complete. This time includes the time to replicate the file across the replica nodes and wait time at the Primary replica to obtain a Write quorum of 4 before acknowledging to the client. We also observed that the times depended more on the write quorum and since in both the runs (8 nodes and 4 nodes) the write quorum was a constant 4, the times obtained are quite comparable.