Machine Programming 3 – Simple Distributed File System

Che-Kuang Chu(ckchu2), Jhih-Wei Lin(jhihwei2)

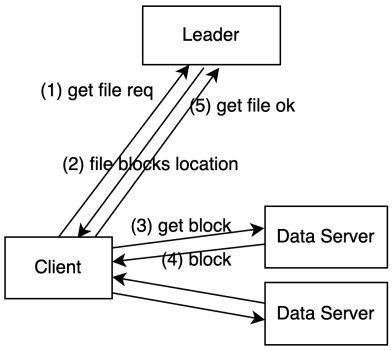
Design

On every machine, we implement four major components: SDFS Client, Data Server, Leader Server, and Member Server.

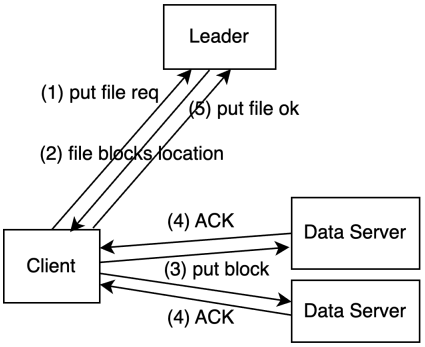
- SDFS Client: handles user commands, acquires r/w permission from the leader, shards the file into blocks, and transfers blocks between the data server.
- Leader Server: r/w scheduling, leader election, metadata maintenance, and replica recovery.
- Data Server: transferring and storing file blocks on the node.
- Member Server: responsible for membership maintenance and failure detection.

Operational Process

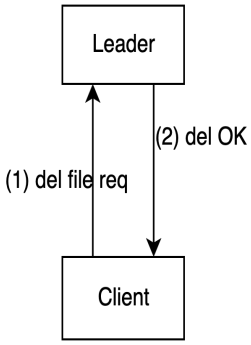
Get



Put



Del



Sharding Files

In our design, we shard a file into multiple 100MB blocks, offering two main benefits: enabling seamless load balancing across nodes and efficient control over the transfer bandwidth between the client and the data server.

To achieve load balancing, uniform block sizes, and random block allocation across nodes ensure a fair and balanced distribution of blocks among the nodes. In terms of bandwidth control, we can determine the number of blocks to be transferred concurrently, thereby maintaining control over the bandwidth allocation.

Read/Write Scheduling

We implement a FileLock mechanism on the leader server and every r/w operation must acquire the lock first. The mechanism is implemented with a counting semaphore weighted 2. The read operation deducts the semaphore by 1 and the write operation by 2. The design resolves conflict operations and only allows at most one machine to write a file at a time, but allows up to 2 machines to read a file simultaneously.

As for starvation-free, we implement a read counter and waiting write counter. The read counter will be reset by the write command every time and vice versa. Once one of the counters exceeds 4 and the process wants to perform the same operation, it will wait until another process resets the counter.

Machine Programming 3 – Simple Distributed File System

Che-Kuang Chu(ckchu2), Jhih-Wei Lin(jhihwei2)

Leader Election

We revised the leader election algorithm from the Bully Algorithm. Every node selects the smallest ID in its membership list, and the smallest one will become the leader. After the leader is elected, it periodically re-elects itself as a leader and broadcasts to other nodes. Once the leader fails, each node will restart the leader election.

Replication and Recovery

To withstand up to 3 simultaneous machine failures, the system ensures the existence of 4 replicas for each block distributed across the nodes.
For recovery procedures, the leader routinely verifies the membership list and metadata. If a block is identified as stored on a failed node, the leader randomly selects another active member and initiates the transfer of the block to that specific node for restoration.

Enhancing File Transfer Efficiency

We facilitate concurrent block transfers between the client and server. To enhance bandwidth utilization, we limit simultaneous transfers to 10 blocks, resulting in a maximum of 1GB bandwidth for read/write operations. This optimization significantly improves file transfer times. For instance, when storing Wiki English Text (1.3GB), the operation time reduces from 12 seconds to 6.8 seconds.

Improving File Read Time

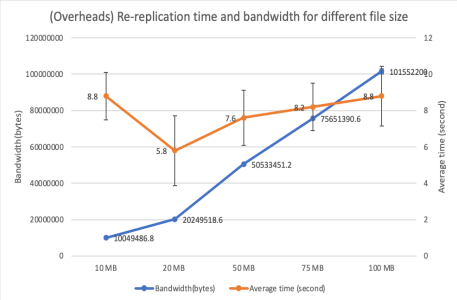
We employ a Read-One mechanism where we complete the retrieval of a block upon receiving the initial replica. Additionally, we write the file onto the disk at the block position instead of waiting for all blocks to be received and merged. This optimization significantly enhances our read time, reducing it from 14 seconds to 2 seconds for reading a 500MB file.

Past MP Use

The MP1 is used to find the specific logs generated by the MP3 system. The Member Server in MP3 is revised from the MP2, which helps detect failure and maintain a list of live machines, and the leader is chosen based on the process IDs in this membership list.

Measurements

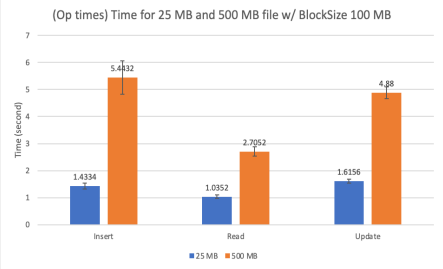
i. Overheads



The system's bandwidth scales in proportion to the file size, and the re-replication time remains similar across varying file sizes. This outcome is anticipated as the test files are relatively small, leading to similar transmission times within our system. However, our system monitors failures and the number of

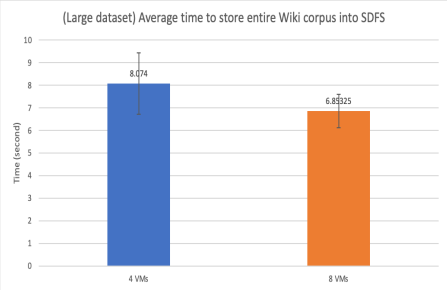
replicas at 5-second intervals, and the majority of re-replication time is due to the time taken for failure detection.

ii. Op times



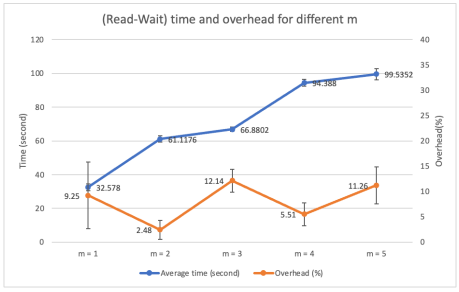
The system demonstrates significantly faster read times compared to write times. This is expected due to the necessity of creating four replicas during write operations, while reads involve retrieving the file only once.

iii. Large dataset



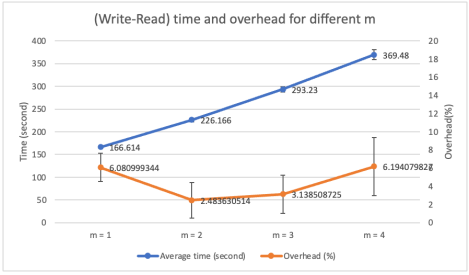
The performance from 8 VMs is faster than that from 4 VMs, as anticipated due to the system's replication process for each file based on the specified replication factor. When replicating to 8 VMs, the stored block size and network bandwidth of each VM are only half compared to replicating to 4 VMs, which explains the expected difference in performance.

iv. Read-Wait (File Size: 12GB, Read Time: 30.819s)



The outcomes remain consistent when the expression *ceil((m + 1) / 2)* yields the same result for different values of m, as the system permits up to 2 machines to read a file concurrently. However, in cases where the *ceil((m + 1) / 2)* values are identical, scenarios with higher values of *m* will introduce more overhead, as the initiation of read operations is not guaranteed to occur simultaneously.

v. Write-Read (File Size: 12GB, Read Time: 30.819s, Write Time: 63.622s)



Due to the conflict between write and read operations, the overall time is calculated as (m + 1) * write + read. Our resulting time for the entire operation aligns with the expected outcome. Regarding overhead, we anticipated an increase, but the results appear inconsistent. We hypothesize that the inconsistency may be due to slower file transfers during the experiment.