CS425 MP3 Report

hcma2 & chinhao2

Design:

We implemented a leader-based simple distributed file system. For each sdfsfile, we would have 4 replicas so that we can prevent up to 3 simultaneous failures as one of the four is guaranteed to reside in the system safe and sound. We elect a leader based on the highest id of all alive processes, while setting the initial leader to be the introducer of the system. The leader is responsible for several tasks, including (1) maintain a FIFO task queue, named TaskQueue, where the front of the queue is the currently executing task (2) keep track of a dictionary, FileToAddrs storing where sdfsfiles are stored, (3) decide where should files stored at a failed process should be re-replicated to, and (4) notify and send three other processes, as back-up leaders, TaskQueue and FileToAddrs whenever they change. We make sure that our replication level is just one more replication than the maximum number of simultaneous failures so that we won't be risking losing all the data at once, including files stored in non-leader servers as well as metadata stored in the leader server.

Available operations include put, get, delete, ls, store, and multiread. These operations can be entered in any server and the server, say Server_req, will send a SEND_OP message to the leader. The leader will add READ/WRITE operations (put, get, and delete) and arguments into the queue, waiting for them to be executed when they become the front of the queue. The queue helps avoid starvation for reads and writes, as operations are executed in FIFO order on their SEND_OP message arrival to the leader. When executing a write operation, the leader would first send a WRITE_REQUEST, which contains sdfsfilename, localfilename, and addresses where the sdfsfile is/should be stored at, message to Server_req that requested the operation. Server_req would then establish a connection and send the localfile to all addresses specified by the leader, sending an ACK message back to the leader when each send is finished. The leader will update FileToAddrs when it receives an ACK message and will pop the WRITE operation when it receives four ACKs as we are writing to four servers. When executing a READ operation, the leader would send a READ_REQUEST to Server_req, containing a list of servers that contain the file Server_req attempts to read. Server_req will attempt to get the file from one of the provided servers, sending only ACK after it reads the file successfully. For ls and store, we would just request the information from the leader without the task queue and output the response in Server_req.

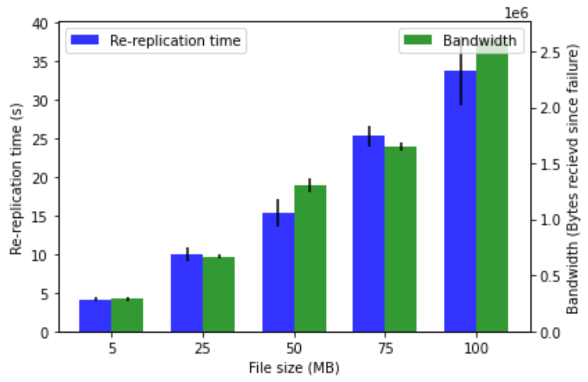
As for file transmission, we use TCP as it provides a stable and reliable connection that preserves the order of data delivery. To simplify our design, we put files sequentially, only updating FileToAddrs when ACK is received at the leader.

Past MP Use:

We used MP2's gossip and gossip with suspicion to maintain a membership list, which is extremely useful as the leader would periodically check if servers stored in FileToAddrs are still alive. If they aren't, then the leader would remove the server from FileToAddrs and re-replicate files that were stored in the failed server. We used Docker to help run and debug for this MP.

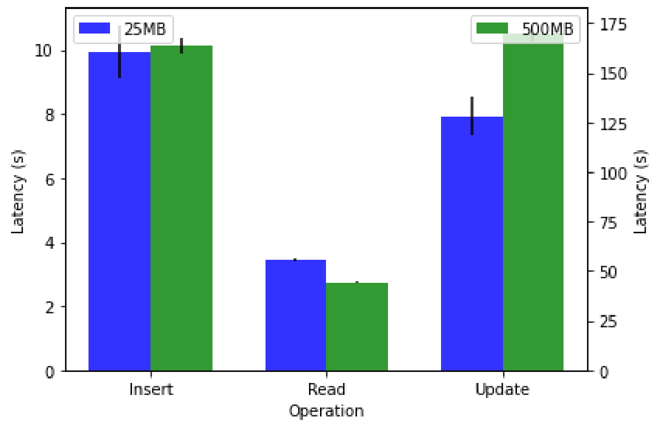
Measurements:

(i) Overheads



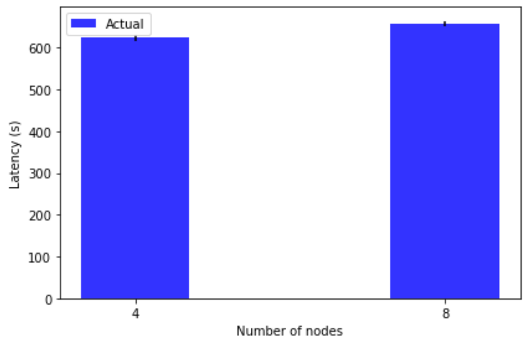
The bar graph above shows re-replication time and bandwidth for file sizes of 5, 25, 50, 75, and 100 MB. We can see a general linear trend of increase for both re-replication time and bandwidth as the file size increases. This is expected behavior as the time and bandwidth required to transfer files increase as the file size increases. We write the data into disk whenever data stored in memory exceeds 10 MB, improving memory management and performance.

(ii) Op times



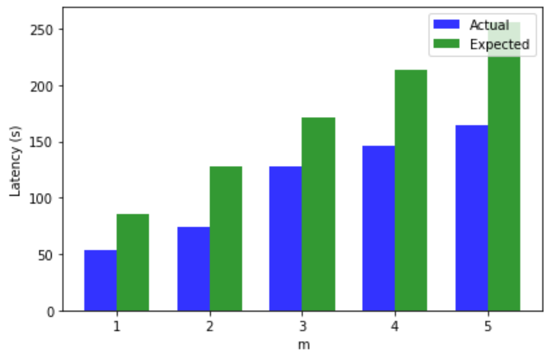
For latency in inserting and updating, it takes 9 seconds to write 25 MB and 170 seconds to write 500 MB. We can roughly see that they are linearly proportional. Latency scales around 20 times just as the file size does. For reading, it also scales linearly as latency increases from around 3 seconds to 45 seconds.

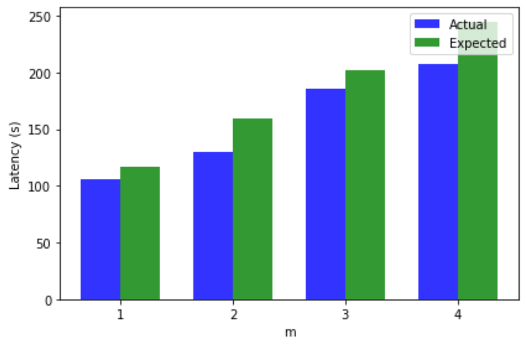
(iii) Large Dataset



In our design, the leader would select 4 replicas to send files to sequentially. Therefore, having more servers than four theoretically will not influence the latency in file transfer. Thus, we believe that the difference appears in the above graph is more about some noise that happens during testing and estimating processes, instead of having 8 servers up will bolster the efficiency of putting large files into our SDFS.

(iv/v) Read-Wait/Write-Read





Read/Write time: 31.3339/42.7308. File size: 200MB
In the left figure, we can see a linear relationship between the number of simultaneous reads and latency. This is the result of our sequential task queue design. We noted that actual latency is lower than expected latency, by around 30-40% of the time. We conclude that it could be because of local read, which happens when the file we want to read already exists on the server. It takes way less time to perform a local read compared to reading through TCP file transfer.

Here in the right figure, we also see a linear relationship between actual latency and expected latency. Compared to Read-Wait, we can see that the actual latency is not that much lower than expected. We conclude that this is still a side effect of local reading; however, here we only attempt to read one-time per testing, resulting in actual latency being only slightly less than expected latency.