Group 14 – MP3 Report
Manan Patel (manandp2), Mihir Shah (mihirs2)

Design

All nodes that join the HyDFS are mapped to a position on the ring using consistent hashing (SHA-1). Each node on the ring stores a list of all other nodes, ordered by their position on the ring. This list is used to find which nodes are responsible for a given file.

All files are initially stored in the HyDFS as a struct with a buffer and the HyDFS file name. This buffer consists of append blocks, which hold data up to a certain size, the node the block came from, and a per-node timestamp. Blocks are created based on the size of the content being written, and appends/creates may be split into multiple blocks. The buffer for each file uses a linked list paired with insertion sort to preserve per-client ordering for each file. When a create occurs, the file is split into append blocks based on the size of the content. Each time an append occurs, the write gets added to the buffer for that file.

We choose a replication level of N=2, so there is a primary node and two successors that are replicas. This satisfies the requirement of being able to handle two simultaneous failures, as even if two nodes fail at once, we still have one replica of the file. Then, when rebalancing occurs, the file contents are copied to the new replicas from the surviving node.

We have a write and read consistency level of 2 because two is a quorum of three replicas. For writing, files are created on all three replicas. When appending, the writes are sent to two randomly chosen replicas out of the 3. We decided to write to only two replicas because it is faster, and we chose availability over consistency. When reading, the consistency of two ensures that all of the writes are seen by the client.

A file's primary node will initiate merging for that file. On a merge, the primary and the first successor exchange their buffers for that file. Upon exchanging buffers, both nodes will create a local merged buffer using insertion sort. With a write consistency of 2, the buffers at both the primary and successor 1 will be fully updated. Once the buffers have been merged, their local buffers are cleared. The merged buffer is then sent to successor 2, which compares its local buffer to the one it was sent. If there is anything in the local buffer that is not in the merged buffer, that means it was appended to the file after the merge began and will be left in the buffer. All matching append blocks will be deleted from the local buffer. Once a node completes its merge, it writes the merged buffer to the disk.

For rereplication, the system utilizes the failure detector. When a failure is detected, the updated primary node will send the new node for a file both the contents in its buffer and on the disk. The content previously written on the disk is then written on the newly assigned node's disk, and the buffer is added to its local buffer for the file. On a join, a similar process occurs where if a node has a file and is no longer responsible for it, the node sends all of its content for that file to the new node.

Past MP Use

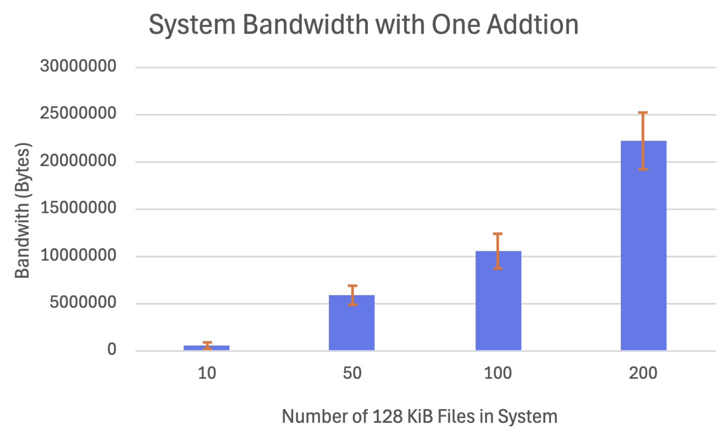
The failure detector from MP2 was used to add and remove members from the HyDFS system. MP2 was converted into a struct that is initialized on the server side of MP3. MP3 uses channels to remove and add members to its list of nodes on the ring. Whenever a member is added or removed, after MP3 receives the message, rebalancing occurs to ensure that there are three copies of the file (in the case of removal), and that the proper nodes are responsible for a file (in the case of addition). Using MP2 like this allowed MP3 to only focus on handling file storage without worrying about membership information.

We did not use MP1 to debug, but rather we used lots of print statements to see whether the expected sequence of events was happening. In hindsight, it might have been easier to use logs and then use distributed grep to check which events occurred.

LLM Use

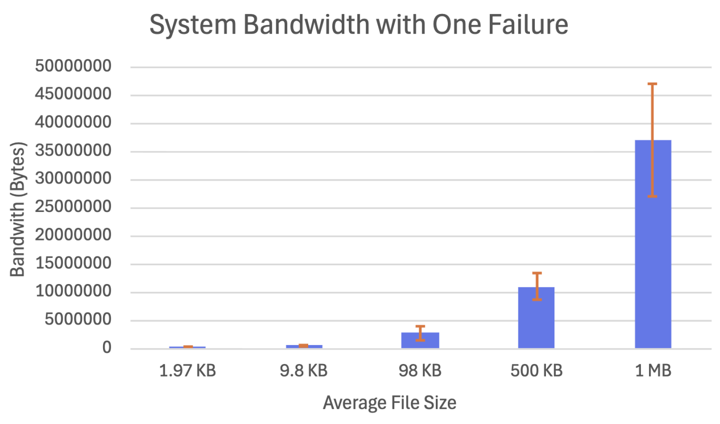
We only used LLMs to help us debug. Whenever we implemented new functionality in our program, we would test it. Since it wouldn't work the first time, we would start to debug using print statements. If we got stuck and could not pinpoint the cause of the issue, we would ask an LLM to see if it could tell us where to look. Many times, it wouldn't tell us the right thing, but the wrong suggestion/fix led us to find the correct issue. LLMs looked at both server.go and client.go, and usage is in the LLM directory.

Measurements



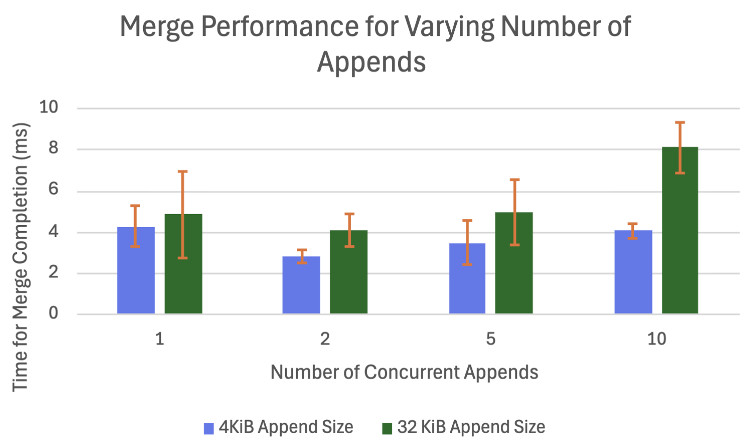
As the number of files in the system increases, the bandwidth of rebalancing increases proportionally. A greater number of files means it is more likely that a file is at the "wrong" node, so when a node joins, the number of files rebalanced will increase as well, explaining the increased bandwidth.

The bandwidth during a failure is proportional to the average file size. Additionally, the standard deviation increases as the file size increases. This was a little surprising, but it makes sense because the failing node is chosen at random and each node stores a different number of files. As the file size increases, this variability has a larger impact because a difference of a couple of files still results in a large difference in bandwidth.



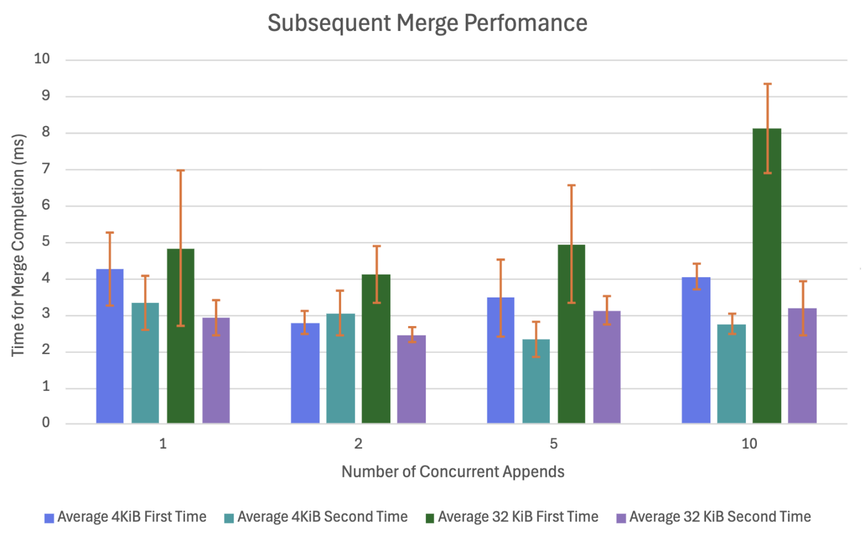
Group 14 – MP3 Report

Manan Patel (manandp2), Mihir Shah (mihirs2)



Merge time generally increased with both the number of appends, and the size of each append. Surprisingly, the merge times for a single append were longer than the merge times for two and five appends. We are unsure what caused this behavior, but since merge times for all the 4-KiB appends were all similar, we believe much of the latency was due to network delay and not actually the time it took to transfer the data for the merge. For the 32-KiB appends, the sudden jump for 10 concurrent appends is due to the additional data that needed to be transferred for the merge.

Most cases showed that the second merge is faster than the first merge, regardless of the file size. This makes sense because the buffers will be empty, as the first merge already merged all content, meaning the merge does not actually merge anything. However, the second environment (2 concurrent appends) was surprising because the second merge took longer than the first. We are unsure why this occurred, but we suspect it was due to a random spike in the network's latency at that time.


Subsequent Merge Perfomance
Time for Merge Completion (ms)
Number of Concurrent Appends
Average 4KiB First Time
Average 4KiB Second Time
Average 32 KiB First Time
Average 32 KiB Second Time