MP3 Report - CS425

Carson Sprague (cs104) & Rohan Kumar (rohandk2)

Overall Design:

To complete the HyDFS MP, we wanted to build on top of the existing MP1 and MP2 infrastructure, making particular use of the failure detection used in MP2 to aid in our re-replication of files under multiple failed nodes. While we previously used TCP and UDP for our log querier and failure detector respectively, we shifted our design significantly to accommodate the multitude of features within the HyDFS key-value store system. We used the MP1 and MP2 recommended solution to implement MP3, as we felt the structure of the recommended solution would facilitate our development of MP3. Our design for the various features of MP3 are described in detail below.

Node Design:

Since each node must be able to act as a client and a server, we decided to implement our entire system using HTTP servers in Golang. To accomplish this, we set up a UDP server for each running machine, and create five HTTP handlers for each of the operations which a node should be able to perform:

1. Uploading a local file to HyDFS
2. Downloading a HyDFS file to the local folder
3. Check if a file exists in HyDFS
4. Replicate a HyDFS file across a quorum of replicas.
5. Multiappend functionality

When each of these operations must be performed, we will first retrieve the correct node for which a HyDFS file is stored using a consistent hashing algorithm, then execute the operation on the specified HyDFS file, catching any errors which occur.

Replication Design:

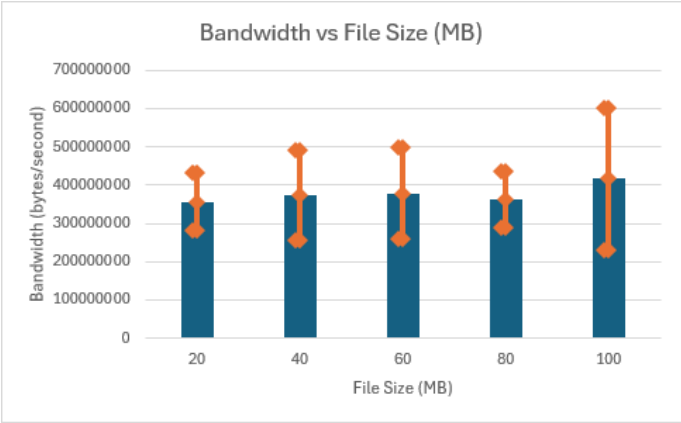
Each file in our system was stored on 3 locations - the main node where it mapped to, plus the next two nodes on the ring (wrapping around of course). This allowed us to handle 2 simultaneous failures. Since we used consistent hashing, every node would be able to calculate where a file maps to, as well as the replicas it's stored at. To handle failures, each node would remove the dead node from the hashing ring, and then see what files were mapped to itself. Then, it would upload the file to the next 2 nodes in the ring to ensure there were enough duplicates. To handle appends, the node making the append would send its append to the 3 nodes responsible for storing the file. Again, every node is able to calculate this because the ring is consistent across the system.

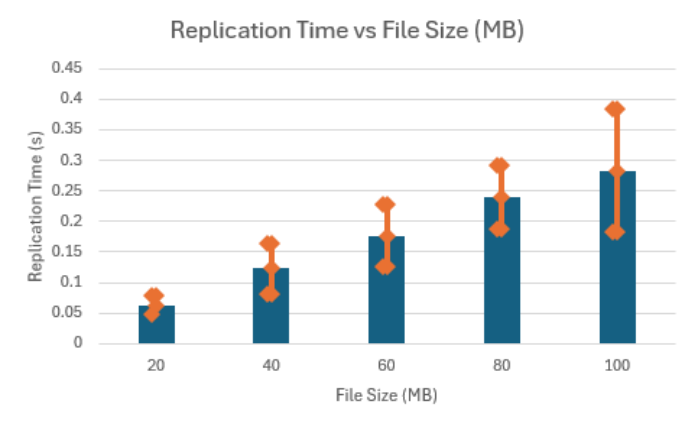
Past MP Use

The failure detector of MP2 was obviously essential for maintaining the list of active nodes, which could then be used to build and update the ring. The distributed grep of MP1 was very useful for not having to keep switching terminals to the correct VM.

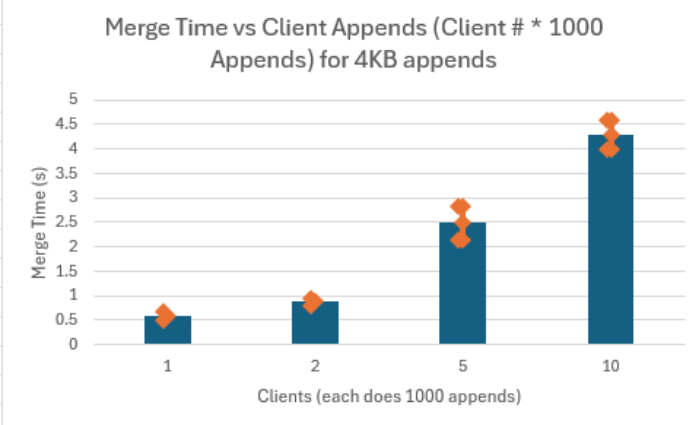
Testing & Analysis

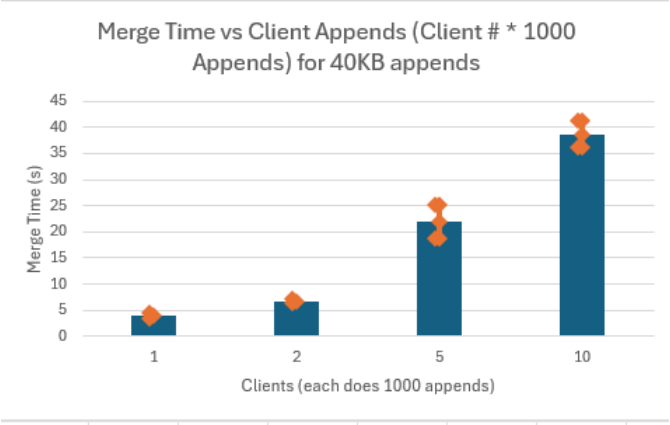
The first test we performed was to analyze the replication time vs file size as well as the bandwidth vs the file size. The results we got were pretty consistent with our expectations. The bandwidth (in bytes/second) stayed relatively constant with varying file sizes. This would make sense as the 'pipe' to send information is pretty constant. This leads into the replication time vs file size, which grows linearly. Again, these two compliment each other because if the file size grows, but the bandwidth stays constant, then the time to send the file would grow in proportion to the file size.





The next test we did analyzed how our merge time reacted to the number of client appends for different file sizes. The first thing to note is that for a 4 Kb vs 40 Kb file, our merge time had a roughly 1:10 scaling. The trend on both of these graphs is that as the number of clients increases, it seems that our merge time has what looks like an exponential curve. This is obviously not ideal, but seems to align with our implementation of merge and how we compared the replicas and ultimately decided on the final version.





Next we compared various workloads using caching vs not using caching. Caching had a net improvement overall compared to not using caching at all, but depending on the size of the cache and the specific workload (read heavy vs write heavy) the cache was more or less efficient. We found that for a larger cache (that included more of the files) we were able to get a smaller read time when compared to using a cache that only covered a small portion of the set, but this varied depending on the frequency of file access. For example, uniform distribution reads work better with a larger cache, but a Zipf distribution of reads can work well with a smaller cache because a small number of files are being 'hit' much more frequently, so other files don't necessarily need to be cached because the benefit is not really there. Also important is the number of writes vs reads in the workload. In a read heavy workload, we can often just read from the cache without having any overhead. Any time a write happens however, the cache needs to be refreshed because it doesn't reflect an accurate state of the file. The short of it is that the more write heavy the workload is, the less effective caching is.