CS 425 MP3 Report
GA6 - Saurabh Darekar(sdare1) & Ankit Chavan(auc3)

To design the Hybrid Distributed File System (hyDFS), we divided the design into several components, each handling a specific function:

Creation of the Ring: We used the Membership protocol from the second MP, adding ring IDs to each member. The ring ID is calculated by hashing the ASCII values of the machine name. We use the same function to hash a file and assign it a ring ID.

File Transfer: A dedicated service continuously runs to receive files, directly accepting any file transfer requests from machines in the ring. We send a JSON file with metadata about the incoming file ahead of sending the file, which indicates whether it is a hyDFS file, an upload, an append request, or a requested file. On the sender side, a queue handles requests to send files from either the local or hyDFS file system to other machines. When a service on a machine wants to send a file, it prepares a request and pushes it to this queue.

Coordinator: We use a coordinator-based approach, where the coordinator is the node responsible for storing the original file, not its replicas. Any file update requests are directed to this node, which performs the necessary actions.

Replication Process: Our system uses a push-based replication method. When a node fails, all nodes check if this affects them. If the failed node was a successor, its predecessor pushes files to its next two successors. If the failed node was a predecessor, its first successor marks replicated files as its own and pushes them to its next two successors. If a node rejoins, its two predecessors and its successor handle the update. The predecessors push replicas to the newly joined node, while the immediate successor recalculates the hash of its files to identify any file that should belong to the new node. It sends these files to the new node, updates its owner list, and informs its own successors of the change.

Eventual Consistency: To ensure eventual consistency, a file owner checks for file consistency every minute by merging files that have had recent changes. Each machine keeps a log for every file it stores, and during a merge, the file owner requests logs from replicas to check for inconsistencies. If inconsistencies are found, the owner pushes the latest file version to the replica nodes. As the coordinator for the file, the owner always has the most recent version.

Client-side File Caching: We implemented caching with a Least Recently Used (LRU) policy. A queue stores file metadata and tracks the total size of files it is queuing, with a limit on the overall cache size so that only a specific number of files and total file size are saved. When a new file arrives and the cache is full, the oldest file is removed to make space. Files that remain in cache beyond a certain time are checked against the owner's hash to verify that they are still up-to-date. If the hashes match, the cached version is marked consistent, and the user is notified. If not, a fresh copy is requested from the owner.

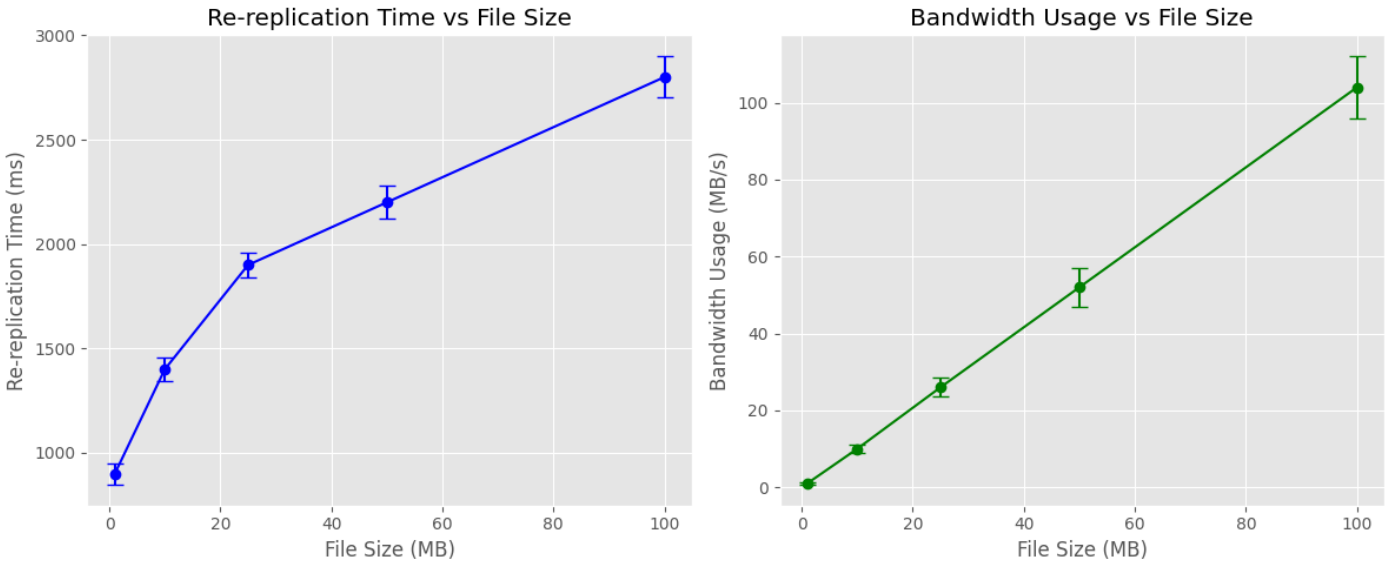
Parallel Services with MP2 Code: We used MP2 code as a foundation and developed additional services in parallel. Multiple receiver services run on different ports and threads from the MP2 services. We used MP1 code to debug while testing the code on VMs.

CS 425 MP3 Report

GA6 - Saurabh Darekar(sdare1) & Ankit Chavan(auc3)

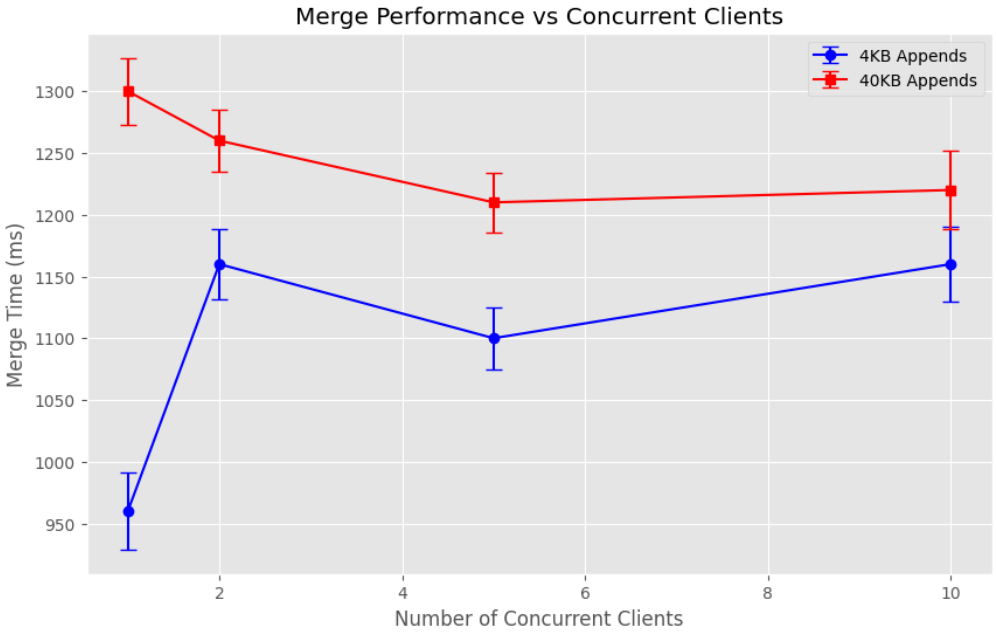
Graph 1: We observed that during re-replication, the time required grew as the size of the file that needed to be replicated increased. Bandwidth usage also rose with larger file sizes, but the file transfer speed was very high. For 100MB they took max time around a second.

(Overheads)



Graph 2: For Merge performance, we observed that appending a 4KB file to the original file took the least time with one concurrent client. Time increased sharply when requests were sent from two clients, and showed only minor fluctuations with five and ten clients. However, for the 40KB file append, the longest time was recorded with a single client, and the time gradually decreased as the number of concurrent clients increased.

(Merge performance)

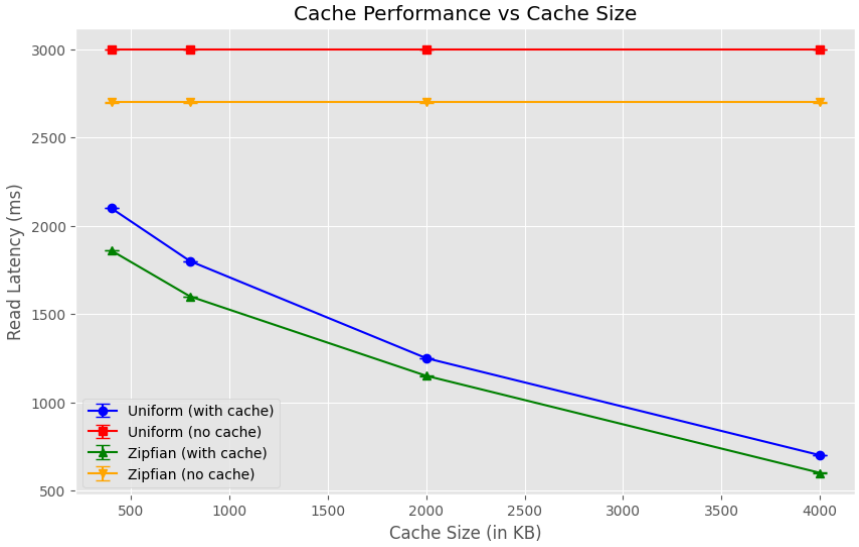


CS 425 MP3 Report

GA6 - Saurabh Darekar(sdare1) & Ankit Chavan(auc3)

Graph 3: We observed that when we used caching, the read latency decreased as the cache size increased. Additionally, the latency was lower when we used a Zipfian distribution workload compared to when we read the files uniformly. In cases where we didn't use caching, we found that the latency for the Zipfian distribution was still lower than that for the uniform distribution.

(Cache Performance)



Graph 4: When we repeated the experiments from iii) but with 10% appends and 90% reads and observed only a small difference in latency across both uniform and Zipfian distributions. In both distributions, latency was approximately 50% lower when cache was used, compared to when it was not.

(Cache Performance with Appends)