CS 425 MP 3: Hybrid Distributed File System

Shubham Thakar (sthakar3) Saumitra Chaskar (chaskar2)
Group 69

Cluster Design and Architecture: The cluster consists of 10 total nodes, where any node can act as a coordinator. Clients can connect to any coordinator node to perform operations such as create, get, append, or merge.

Threads:

Server Thread: Listens for incoming TCP connections on a server socket and manages sending and receiving messages over established connections. This thread uses Python's select.select paradigm with non-blocking TCP sockets to handle communication efficiently.

Membership Monitoring Thread: Monitors the membership list for node joins, leaves, or failures and triggers replication processes when changes are detected.

Replication Strategy: To ensure fault tolerance against up to 2 simultaneous node failures, the system uses active push-based replication with data replicated across 3 replicas (1 primary and 2 secondary). The coordinator uses active push based replication and writes to all three replicas itself. If a node failure is detected, the first two predecessor nodes and the first successor node of the failed node will handle the re-replication of their files using push-based replication to maintain data availability and consistency.

Create and Get Operations: The system uses a quorum size of 2, ensuring that both create and get quorums overlap at at least one replica. This approach ensures consistency and high availability. While performing get in case of inconsistent append logs due to an append failure, the coordinator returns the version of the file with the largest append sequence number, as this represents the most recent data. The coordinator sends an acknowledgment to the client only after receiving acknowledgments from at least 2 replicas.

Append Operations: We maintain a unique append log for each client-file pair on each replica, with each append assigned a unique sequence number by the replica. To ensure consistency in append ordering in case of back to back client writes on the same file, the coordinator always writes to replicas in the same order (e.g., replica 1 [primary], then replica 2 [secondary], and so on). This guarantees consistent sequence numbers even if the client sends back-to-back appends to different coordinators.

Merge Operation: The merge process for each file involves sorting the appends:
Sorting by Client: Ensures that appends are consistently ordered by client across all replicas.
Sorting by Sequence Number: Ensures that appends from the same client maintain the same order across all replicas.
This dual sorting strategy guarantees that, post-merge, all replicas have the same file contents in the same order, preserving data consistency throughout the cluster.

Past MP use: Our MP2 fault detection was running on a different port and we used the membership list to detect failures/joins/leaves and replicate accordingly

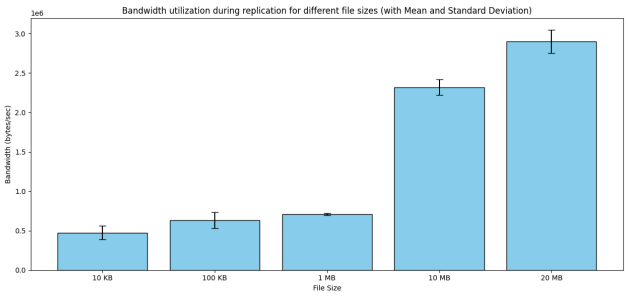

Fig 2.

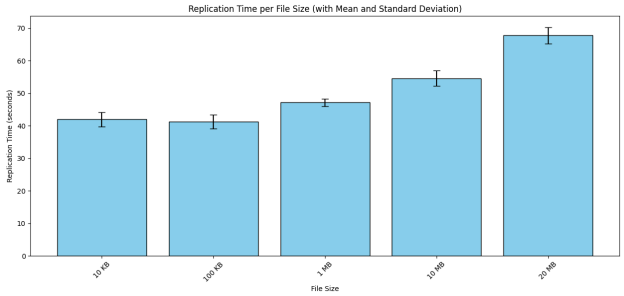

Fig 1.

Fig 1. Replication time reported here includes both the failure detection time (Ping Ack) and time to replicate. The larger file sizes (1 MB, 10 MB, and 20 MB) show a noticeable increase in replication time, with 20 MB having the longest time and the largest standard deviation, indicating more variability in replication time for larger files.

Fig 2. For this experiment we measured the average bandwidth of nodes participating in file exchanges during replication and are reporting this value. We monitored the interface at each of these nodes to get packets sent and received per second and used that to calculate the bandwidth. The graph shows that bandwidth utilization increases as file size increases, with larger files (10 MB and 20 MB) using significantly more bandwidth compared to smaller ones.

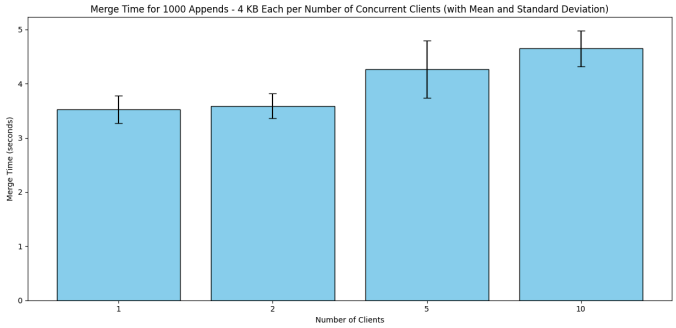

Fig 3.

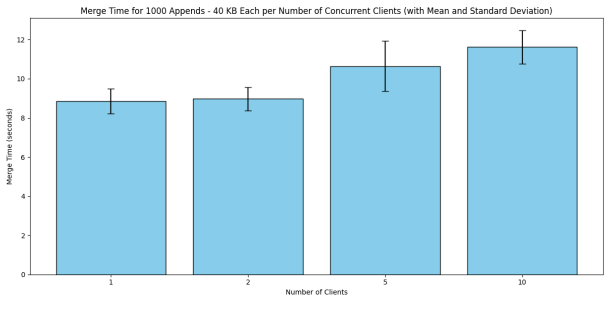

Fig 4.

Fig 3. For a single client we did 1000 back-to-back appends, for 2 clients we did 500 back-to-back appends each and so on. As the number of concurrent clients grow merge time increases but the increase is not a lot because of the way we are handling merge and appends

Fig 4. Merge times for 40 KB appends are consistently higher than for 4 KB, and the increase with more concurrent clients is more pronounced, indicating larger data sizes lead to longer and more variable merge times.

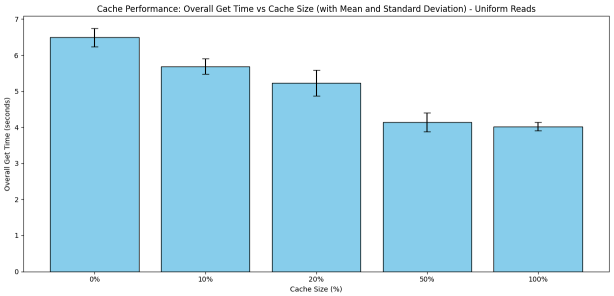

Fig 5.

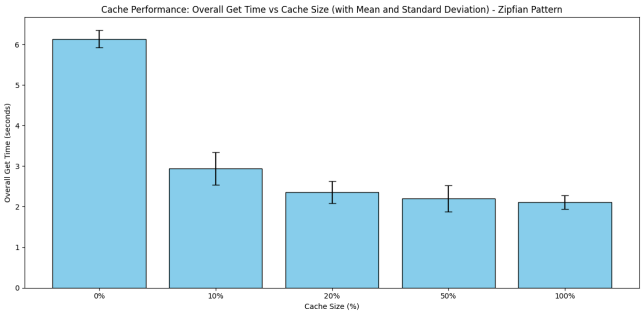

Fig 6.

Fig 5. For this experiment we wrote 1000 files to hyDFS each of size 4kb and performed 2000 sequential gets. The graph shows that as cache size increases, the overall get time decreases, improving performance. The difference between 50% and 100% cache size is small because, at 50%, the cache already holds most frequently accessed data, so increasing it to 100% offers minimal additional benefit.

Fig 6. The Zipfian graph shows a sharp drop in get time from 0% to 10% cache size, with minimal improvement beyond that. This reflects the benefit of caching the most frequently accessed data early on. In contrast to this the uniform distribution has a steady improvement across all cache sizes.

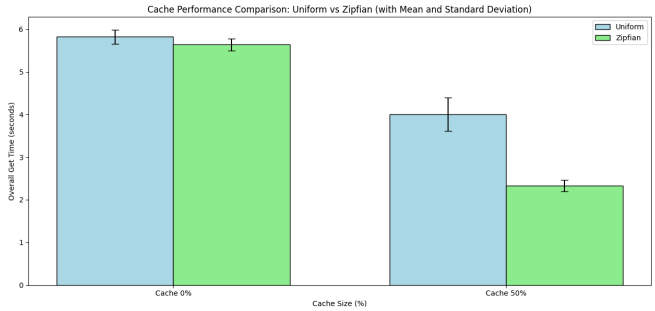

Fig 7.

Fig 7. This graph compares cache performance for uniform and Zipfian distributions at 0% and 50% cache sizes. For this experiment we wrote 1000 files to hyDFS each of size 4kb. Our query workload had 1800 gets and 200 appends. At 0% cache, both distributions have similar get times, showing no cache benefit. However, at 50% cache, the Zipfian distribution shows a significantly lower get time compared to the uniform distribution, indicating that caching benefits the Zipfian pattern more due to its skewed data access.