- Sanket Donty [sdonty2] and Saksham Gera [gera3]

Design

Our code builds upon the MP2 codebase, leveraging it for failure detection and membership change dissemination. This project adds several new features on top of this framework. It enables the creation of files in a distributed system across multiple servers, similar to how Cassandra works, and offers various methods for interacting with a distributed file system, including reading and appending data. Because this is a distributed system, the MP ensures that concurrent actions are managed smoothly, maintaining total order for appends across all replicas. The system is designed to handle failures and new server joins efficiently by redistributing files during such events, ensuring that file hashes are correctly mapped to server hashes in spite of changes in the network.

Replica Design:

The system always keeps three replicas of each file. When a user creates a file on a server, it identifies the appropriate servers to store the file using hashes, writes the file to three servers, and returns after receiving confirmation from at least two servers (quorum size of 2). The same process is followed for reads: when a get request is made, the system waits for fetches from two servers to display the most up-to-date file.

Consistency During Failures:

To maintain consistency during failures, the system handles both primary files owned by the failed node and replica files stored on it. For primary files, the replica nodes take responsibility for re-replicating the lost data. The immediate successor of the failed node becomes the new owner and ensures that a copy of the file is created to maintain three replicas in the system.

For replica files lost due to failure, the owner node upon detecting the failure sends a copy of the replica to the successor of the failed node to maintain consistency.

Appends and Merging:

All append operations for a file are directed to the primary owner node to obtain a global sequence number for the append. Once the sequence number is received, a quorum write is performed, storing the append on the primary node and its two replicas. Appends are stored in chunks for improved performance, with each chunk having a sequence number appended to the filename.

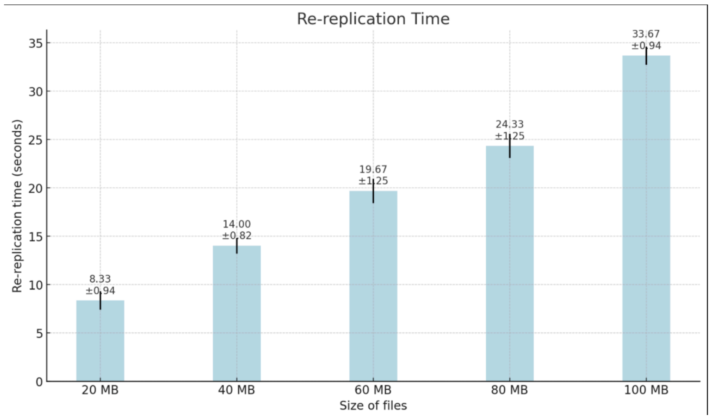
When a user issues a merge command, it is sent to all replica nodes for processing. Each node retrieves the append chunks associated with the file from the local HYDFS folder, orders them by their sequence numbers, and merges them into a single file. This merging process occurs concurrently across all replica nodes.

Use of Previous MPs:

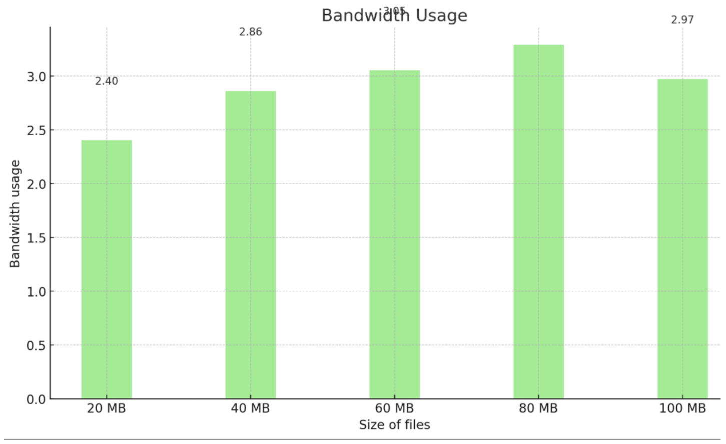
The first two MPs were integral to this project. MP1 was used extensively for debugging during development, while MP2 provided the foundational framework for MP3. The failure detection and membership change dissemination features in MP2 were crucial for building a resilient and scalable distributed file storage system. When a change in the membership list was detected by MP2, it triggered callback methods in MP3 to handle failures or new server joins, which included redistributing files accordingly. Thus, MP1, MP2 and MP3 collectively work together to build this scalable distributed file system.

- Sanket Donty [sdonty2] and Saksham Gera [gera3]

Overheads charts

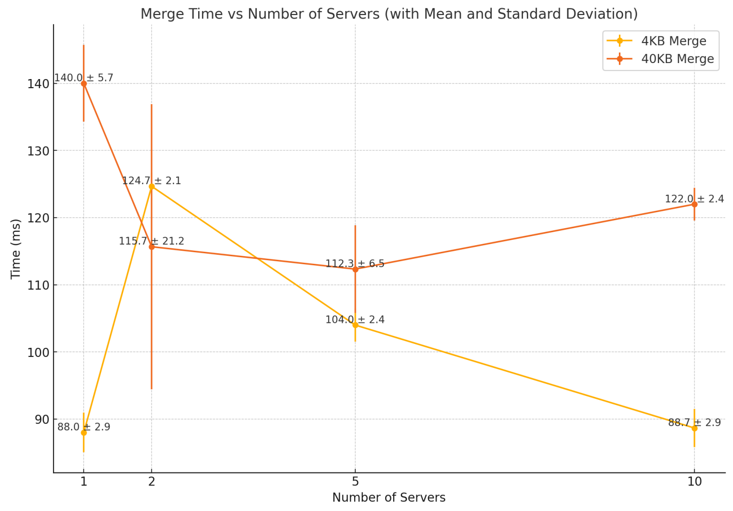


As expected, the chart shows an increase in re-replication time with increase in size of files. This is expected because our code handles sending of large files by splitting them into chunks. Hence, larger files would require more network calls, adding to the total latency.

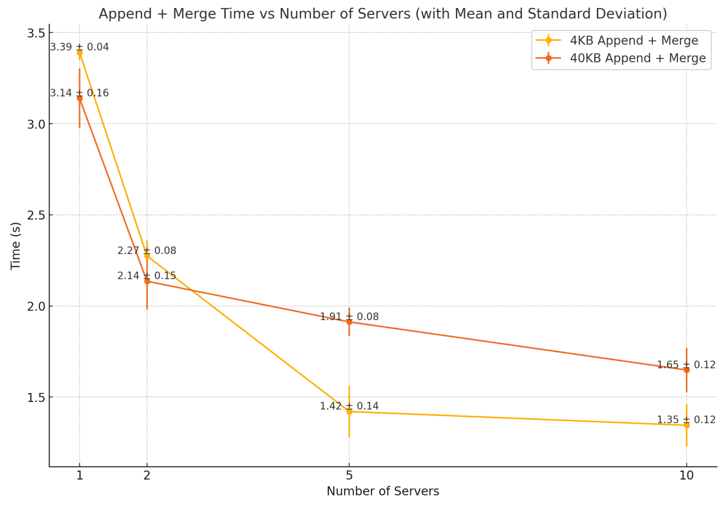


The bandwidth increases marginally. This is reasonable, and we were not expecting any increase in bandwidth since we send file chunks sequentially in our code. So regardless of how large the files are, since they are split into chunks and sent over the network sequentially, the bandwidth usage should not increase significantly as we increase the file size.

Merge Performance:



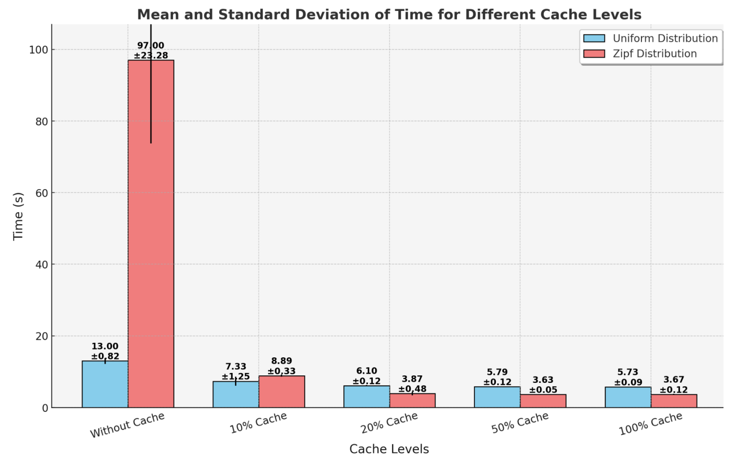
This graph depicts the merge time to merge 1000 appended files, Although the trends show some discrepancy in the graph but that might be due to network latency issues.Overall the graph behaves consistently without much deviation. The merge time does not fluctuate too significantly because it does not depend on the number of servers doing concurrent appends



This graph shows the time taken to perform 1000 concurrent appends and then merge them. The graph behaves as expected. The 40Kb append files take more time to append and merge than the 4kb files.

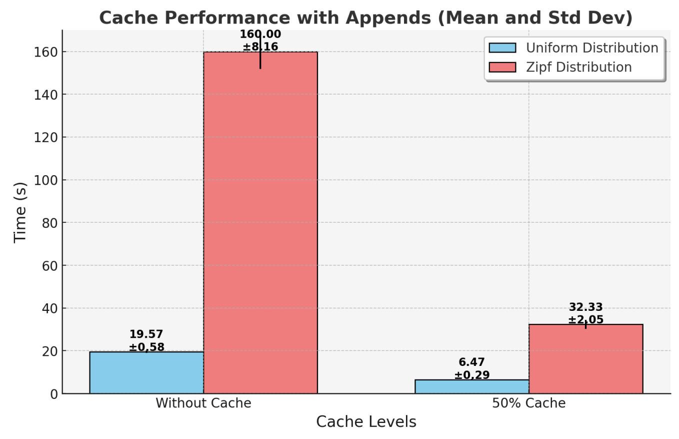
- Sanket Donty [sdonty2] and Saksham Gera [gera3]

Cache Performance:



The time taken to read from zipf distribution is comparatively higher than the uniform distribution without caching which is the expected behavior. When using 10% cache we see that the time taken to get zipf distribution decreases but still remains more than uniform distribution. As we increase cache size to 50% or more, we see that the uniform and zipf distribution read times are almost constant and achieve the lowest possible times.

Cache Performance with Appends:



Appends affect the performance of gets with or without cache. As we clear our cache every time an append is performed from the same client, therefore even though we have allocated 50% of cache size, the time taken in zipf distribution is higher. Without caching, the performance degrades a lot and zipf distribution takes around 2 mins which is worse than 50% cache performance.