MP3 Report

Jingting Feng(feng45) Qi Jing(qijing3)

1. Design

Initialization: After a node joins the member list, the Introducer adds it to the list, assigns its ID based on a hash function, and stores it in a ring structure similar to Cassandra. Then, the new member list and ring are broadcasted to all nodes.

File Operations:

1.1. Create: The function reads the content of the specified local file, locates the target server using a consistent hashing algorithm, and attempts to write the file to the target server and its successor nodes. This process continues until the required quorum for writes is met, at which point further replica writes stop.

1.2. Get: The function retrieves a file from the distributed file system and saves it to a specified local file. It locates the target server by hashing the filename and then attempts to read the replica from the target server and its successors.

Once the quorum for reads is met, the function compares the timestamp to get the latest version and then writes it to the specified local location

1.3. Append: The function appends content to a file in the distributed file system. After reading the content from a local file and locating the target server, it calls the sendAppend function to send the content to the target server and its successors.

1.4. Merge: It ensures consistency across multiple file replicas by merging the content of file from each replica. It retrieves files from the target server and its successors, identifies and merges differences across replicas, then sends the unified content back to each server. The merge completes once all replicas are consistent.

1.5. MultiAppend: The function performs concurrent append operations, adding file content to the same target file across multiple virtual machines. The function verifies that the virtual machine address list and local file list are the same length, then sends “multi append” to each virtual machine. When the virtual machines receive the “multi append” message, they read each local file and append its content to each designated server. The function waits until all append operations complete on each VM before returning.

Caching: Client-side caching are implemented by LRU. Before executing the Get function, the cache is checked; if the file has been recently accessed and exists in the cache, it is retrieved directly from the cache. Each time an append occurs, the LRU cache reloads the appended file to ensure that subsequent get requests can access the latest file version.

Replication: For replication, as there can be up to 2 node failures, we set the number of replicas to 3. After a node failure, all alive nodes in the member list check their files, visiting their next N nodes to verify that there are 3 replicas. If replicas are insufficient, they promptly create additional copies.

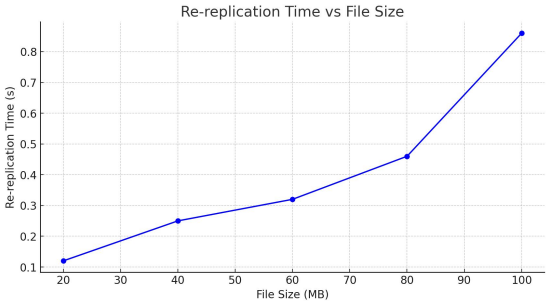
Consistency: We use quorum to ensure read-write consistency. For write and read operations, we proceed to the next operation step only after 2 replicas (as >n/2) return a response. This ensures consistency as there will always be at least one server in common between read and write quorums, meaning that at least one server in the quorum returns the latest write.

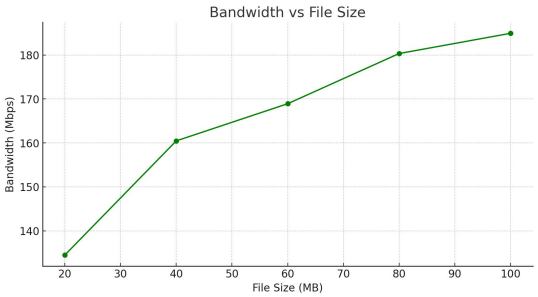
2. Past MP Use

We used MP2 to detect failed nodes and initiate file migration for these nodes. MP1's grep operation helped verify the consistency of different replicas by searching for specific keywords.

3. Measurements

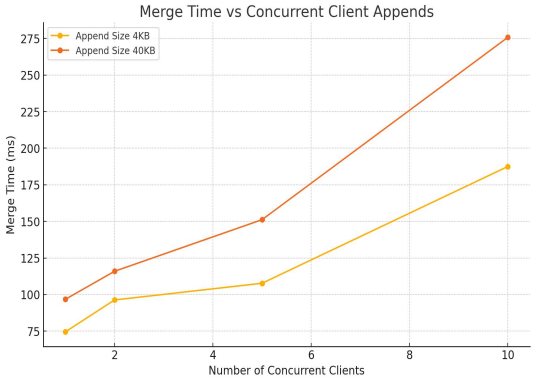
3.1 Overheads





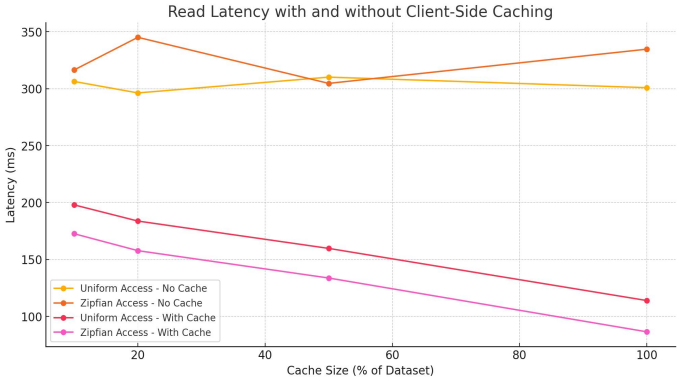
As the size of the file to be recovered increases, the time required also rises, since larger backup files take longer to transfer. For bandwidth, the change is not very noticeable, which could be due to network stability factors. However, there is a trend where larger files consume more bandwidth, as more data needs to be transmitted.

3.2 Merge performance



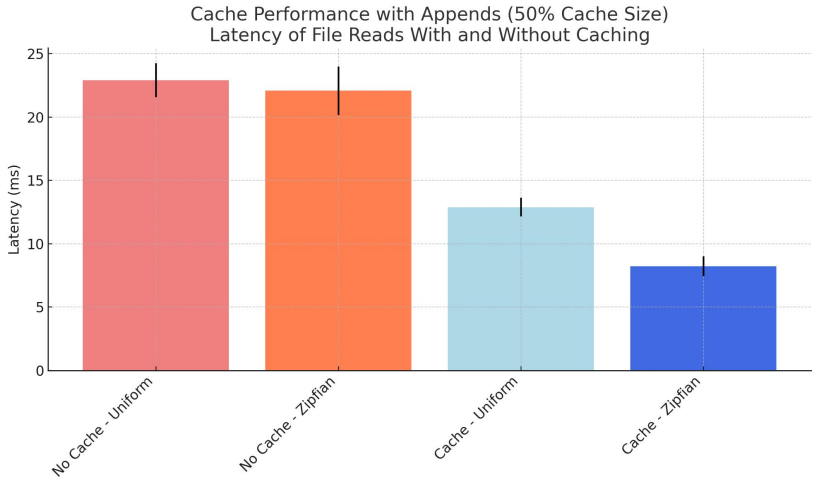
As the level of concurrency increases, the merge time gradually rises. This is because concurrent append operations lead to more frequent file lock contention and synchronization processes, requiring the system to spend more time coordinating these concurrent write requests. Overall, 40KB takes longer than 4KB because the read and write times for 40KB are generally longer.

3.3 Cache Performance



Without caching, latency is highest. The difference in latency between random file reads and reads following a Zipf distribution is minimal, as each read requires fetching data from HyDFS anew.
With caching, latency decreases. When requests follow a Zipf distribution, latency is lower than with random reads, as certain files are accessed more frequently than others. This results in higher access rates for hot files, leading to a more concentrated load on HyDFS.

3.4 Cache Performance with Appends



With Caching: In a uniform distribution, each file is accessed with equal probability, leading to a moderate cache hit rate since files are randomly distributed. With 50% cache size, only half of the files can be cached, so the hit rate and resulting latency reductions are moderate. Under Zipfian distribution, certain files are accessed more frequently, increasing the chance that these popular files remain in the cache. The LRU cache policy helps keep these frequently accessed files in cache memory, further reducing access latency. The latency for Zipfian distribution is significantly lower due to higher cache hits for hot files.With caching, the standard deviation difference between the two distributions is minimal, but under the Zipfian distribution, the standard deviation is larger. This may be because some hot files in the cache are frequently accessed, and occasional cache evictions can cause noticeable latency fluctuations. For example, when a hot file is evicted, subsequent cache misses lead to significantly increased access latency, resulting in a rise in standard deviation.
Without Caching: Regardless of access pattern, every request requires fetching the file from the storage layer, leading to consistently higher latency compared to cached reads. The uniform and zipfian distributions show little difference here since each read operation bypasses the cache. For the standard deviation, in the no-cache scenario, the standard deviation is larger. This may be because the system relies more heavily on underlying resources when reading without caching and is more susceptible to variations in network, I/O resources, and randomness in access paths.