MP3 Report

Design

Our HYDFS algorithm follows the MP guidelines with a background SWIM protocol for node membership detection and file replication on the three successors using a Chord-like ring. We replicate on 3 nodes in order to satisfy the 2 simultaneous failures requirement.

Client Side

For all user requests, we use a deterministic hashing function on client side to obtain IDs for each filename and node (using its IP address) prior to any server requests. All server requests are blocking and use RPC. When receiving a file creation request, the client first registers the filename on the three corresponding servers along with the server's "order" for that file- 0 means node immediately after file on ring, 1 is after that, and 2 is after that, on the ring. This order will come into play with replication. If the file already exists on any of them, the client returns failure. Then, the client enters the append flow with that original content. The append flow removes the file from local cache and goes to all three servers, and must be successful on all three for the client to return success. Since the client blocks until all success, if the client submits multiple appends, they're guaranteed to be processed in that order by server. For get requests, we hash the client name (hostname) and file name together and take the result mod 3, to get a deterministic server to route to for the given client and file. We use an LRU cache for get requests - the first time a file is requested by a client or whenever an append was most recently performed, we have a cache miss, it's routed to the aforementioned server, and subsequently added to the cache.

Server Side

Each server is initialized with an empty KV store. Upon a new create and accompanying order, the server adds the filename and order to the store, and initializes an empty vector. For all subsequent appends, it confirms that the file key exists, and then populates a chunk corresponding to the file, stored at a separate filename derived from the queried file, and appends that to the current chunks for that file (sorted by received time by server). Upon a get request, the node concatenates its previously merged file, if it exists, with all its subsequent chunks by time, and returns this. Upon a merge request, guaranteed to be received at 0-order node, the same procedure is followed but creates/overwrites the previously merged file and writes it (to the actual filename not derived) and updates the value for this file key by removing the chunks. The server then acts as a client in issuing an overwrite request to the two following successors that it received. When receiving an overwrite for given file, the receiver overwrites the given file and removes any previous chunks they had. When a node crash is detected using our SWIM failure detection protocol on the lowest id node in the ring, that node messages the affected nodes (the 2 predecessors + 1 successor of failed nodes) which handles their separate cases (for the 2 predecessors- update/new successors for replication; for the 1 successor- to become 0-order for corresponding files and update new successors for replication). We 1) update the order for a given server for all files affected- either from 1 to 0, or 2 to 1- and 2) send new order to the new successors. New successors receive the overwrite request introduced earlier, as it works independent of if the file existed before. Since only the lowest id initiates this recovery, it can process failures sequentially so it can handle two failures. The first processing may error if failures were sequential, but second is guaranteed to exceed and maintain correctness at the end.

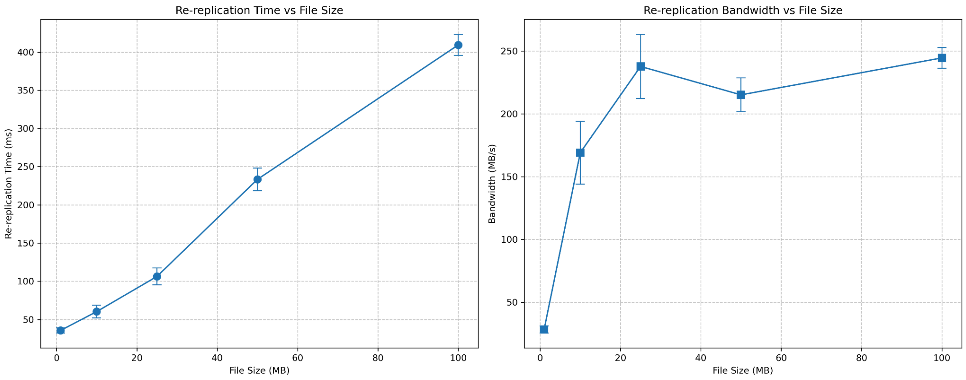
Past MP Use

When using our SWIM failure detection implementation from MP2, since all nodes detect failure in the given time range, we maintain correctness by having only the lowest ID (which can update upon failure) to handle notifying affected servers.

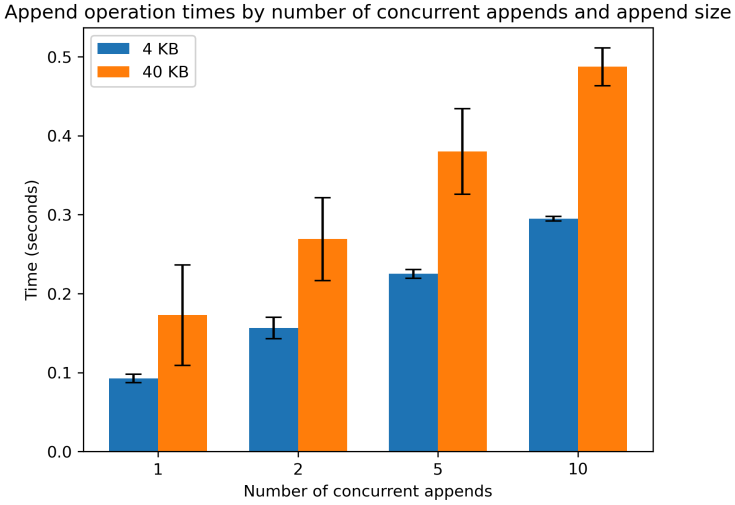
4-> 0.093, 0.086 sec 40-> 0.1165 sec, 0.312, 0.08

Measurements

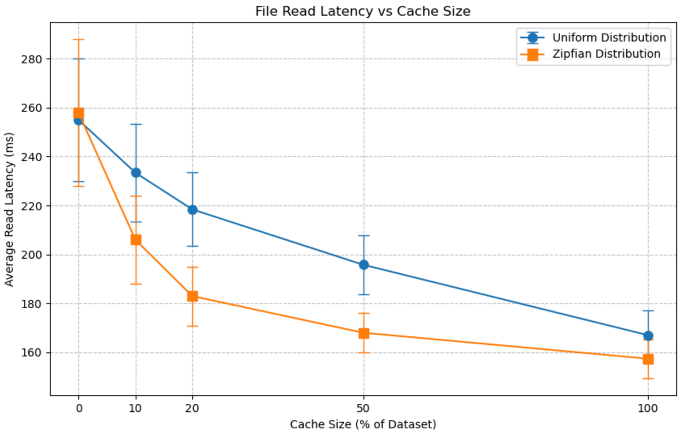
I. We record replication statistics for single failure on server with single file of sizes 1 MB, 5 MB, 10 MB, 50 MB, 100 MB.



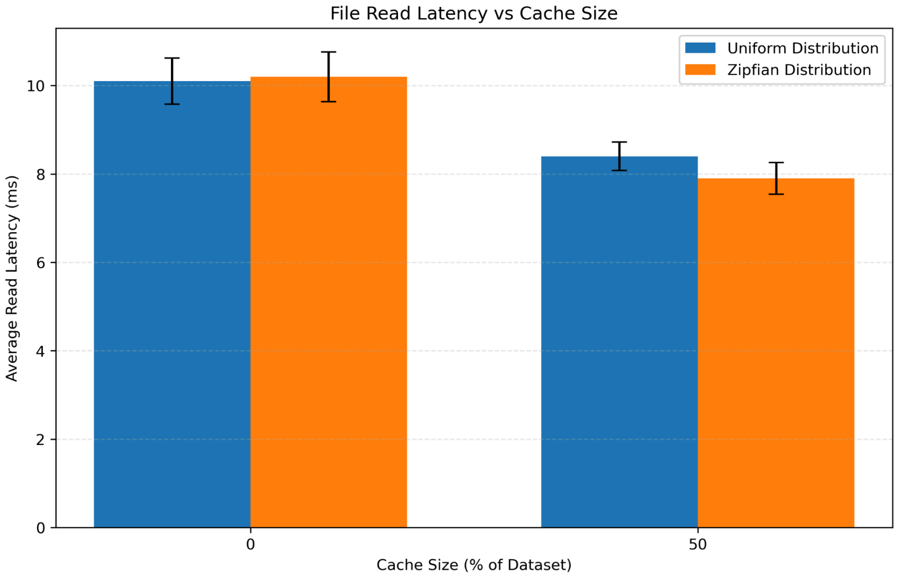
II. Since we're timing the merge after append has executed, the only variable factor is number of files being merged, linear in the number of concurrent clients. As expected see an upward trend in total time as the number of total files increases, but it's likely sublinear due to some fixed costs of file opening/closing and network delay.



III. Since the Zipfian distribution is more skewed, the performance is less affected as the cache size decreases relative to uniform. However at the two ends, with 100% cache and 0% cache, their performance is close to identical.



IV. We similarly noticed that for 0% the performance was close to identical, but Zipfian benefited more from the 50% cache due it’s skewed distribution. However the discrepancy vs the baseline 50% with no appends was larger for Zipfian, as it results in relatively more cache misses compared to before.


File Read Latency vs Cache Size
Uniform Distribution
Zipfian Distribution
Average Read Latency (ms)
Cache Size (% of Dataset)