System Design

For our system, we chose to arrange our nodes in a chord-like ring, where each vm maps to a given node with ID 0-9. Like in Chord, a file with hash n is assigned to the first node in the system with ID >= n. Each node maintains a routing table, where Routing table[n] = ID of node which has all files hashing to n. Each node also maintains a list of successors, which is simply a list of nodes “clockwise” from the current node, including the current node. Nodes are added to and removed from the routing table/successor list when a membership change is detected by the failure detector. We replicate files at the first *two* successors of the owner node (assuming the system has >= 3 nodes), to guarantee that if two nodes fail at the same time a client’s data is still preserved. To implement re-replication and de-duplication we made heavy use of our lists of successors and routing tables. When a node (say A) joins the network, every other node checks if A has become one of its two successors by consulting its list of successors. Say node B has determined that A is now its first successor (i.e. successors[0] = A). B will transfer the files it owns to A. B will also tell the node that *was* its second successor before A joined to remove the files it was previously replicating for B. In this way we ensure that for all files there is one owner node and two replica nodes. Similarly, on failures, if a node determines via its successors/routing table that one of its successors has failed, the node will replicate its files at the node that has now become a successor (successors[1]).

To simplify merges, we maintained a good deal of bookkeeping information for files in a manner similar to Cassandra. Each file maintains a log of append IDs, which consist of a VM number and the timestamp when the append was sent. Each append is not written to the file it maps to until merge time; appends are stored as separate file blocks with a random name, with additional bookkeeping information mapping each append ID to its corresponding block. When we need to merge our files, the owner of the file to be merged simply sends its log for the file over to the two successors replicating the file, and the successors write the append data to the original file using the owner’s log. Since all files are writing their append chunks using a single log, once the merges have concluded all appends to the file are guaranteed to have been written in the same order at all replicas. We use RPCs to ensure that all appends go through at the owner node and both successors to ensure that all nodes have all the append blocks necessary to correctly merge.

To satisfy the eventual consistency requirement, we simply had a thread in our program which calls merge periodically (every minute) on a random file. Since we may have appends going on at the same time as our random merges, we also implemented a locking mechanism for our file logs to ensure consistency even under simultaneous merges and appends.

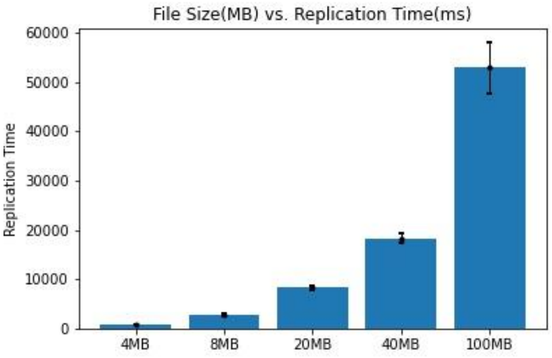
Past MP Use

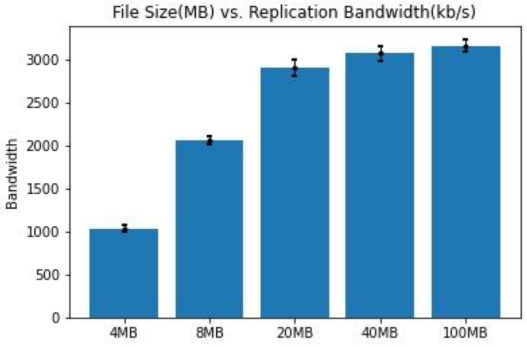
We used MP2’s failure detector to determine when a node failed/joined the network in order to keep routing tables and successors current. We used MP1 to grep our log files during debugging.

Measurements:

Overheads

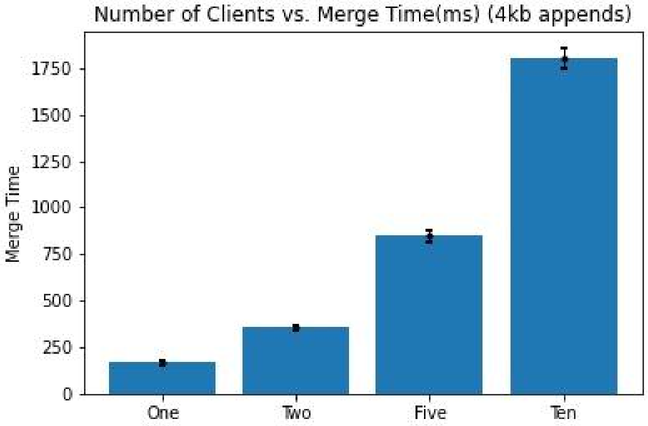
Replication time increases in a roughly linear fashion with file size being replicated. Since we don’t merge files before replication, each replication involves creating the file on the node where the data is being replicated, and then making append calls to the node where the data is being replicated to ensure the replicated logs and data are correct. Accordingly, replication time should increase both as a function of the number of append calls we have to make and the total size of the data being transferred, which matches up with the relationship we see above. Bandwidth starts off relatively small for smaller file transfers, but for transfers of 20MB and above it saturates at about 3MB/s; this is likely approaching the bandwidth limit of the network, at least for the thread we have running the transfer. It is worth noting that the tool we used to measure bandwidth, Nethogs, began performing very badly over ssh connection for larger transfers, further indicating that for larger transfers the bandwidth limit of the network is pushed, capping the speed of our file transfers.

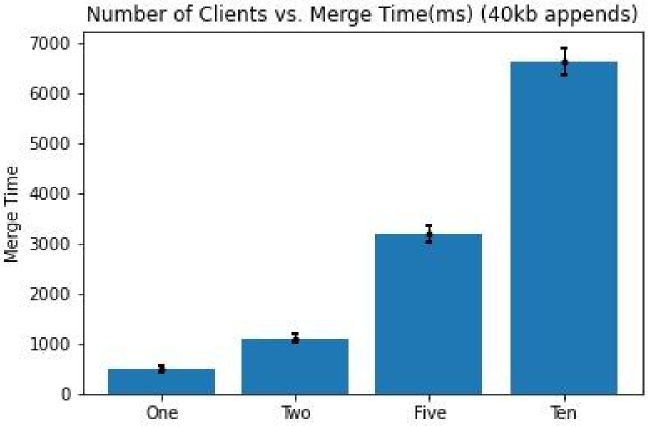




Merge Performance

Merge time increases in a roughly linear fashion with the number of concurrent client appends (1000 appends for one client to 10000 appends for ten clients). This is expected; as more appends are sent the size of the file log that must be transferred on merge increases and the size of the data that must be written to the merge file also increases. Interestingly, the 40kb appends performed much better than the 4kb appends in terms of data written over time. This suggests that a non-insignificant portion of the time spent merging is spent transferring the owner's file log to the replica machines; while ten times as much data is written to the merge file for the 40kb appends, the file log is the same size for both 4kb and 40kb transfers. That the 40kb merges take about double the time of their 4kb counterparts instead of the ~10x merge time we might expect just from data size indicates the part of the merge aside from the file writing, the log transfer, takes up a fair proportion of total execution time. Also worth noting is the speed of our merges (400MB merge in 6 seconds). By maintaining the amount of bookkeeping information we do for each file, we are able to ensure data transfer for merges is minimized and all replicas of a file can simply perform a series of O(1) lookups to write the correct blocks to the file to be merged.

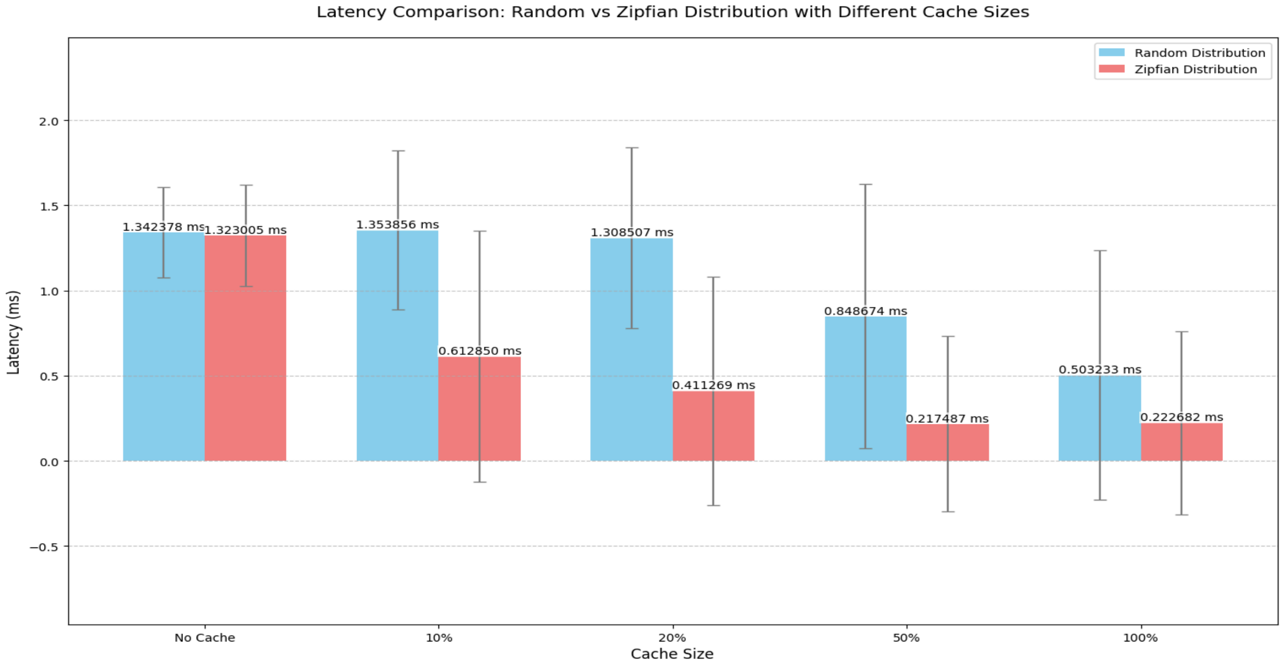




Q3. For Q3 and Q4, measurements are for (100 files in HyDFS, 300 Requests for each config)

1. **No Cache**: Both distributions exhibit high latency, with random distribution higher than Zipfian
2. **10% Cache**: Latency decreases significantly, especially for Zipfian distribution, suggesting caching benefits for frequently accessed files in skewed distributions
3. **20% Cache**: Further reduction in latency, with Zipfian still outperforming random due to the cache favoring frequently accessed files.
4. **50% and 100% Cache**: Latency drops further, with diminishing returns. The Zipfian distribution consistently maintains lower latency.

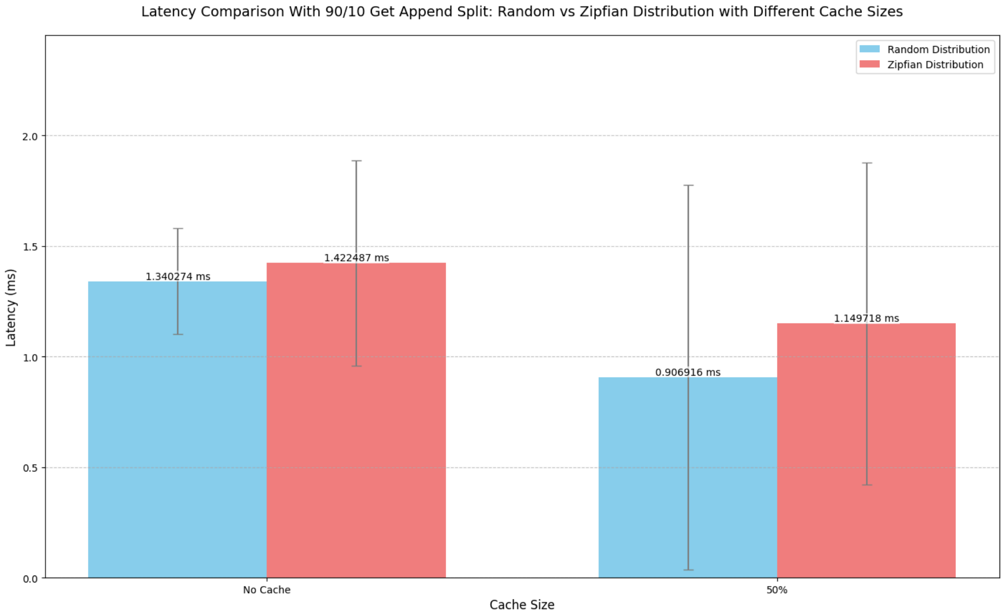
Trend: Latency decreases with increased cache size, more effectively for Zipfian distribution, showing the benefits of caching with non-uniform access patterns.



<u>**Q4**</u>. The graph shows latency for a 90/10 get-append workload in HyDFS with random and Zipfian distributions across different cache sizes

1. **No Cache**: Both distributions have similar latencies, with Zipfian slightly higher, indicating no significant benefit without caching.
2. **50% Cache**: Latency reduces for both, with the random distribution benefiting more than Zipfian, as caching reduces lookup times for randomly accessed files.

Trend: Cache reduces latency in both distributions, but the effect is more pronounced in random distribution, possibly due to a more even access pattern that better utilizes cache.


Latency Comparison With 90/10 Get Append Split: Random vs Zipfian Distribution with Different Cache Sizes
Random Distribution
Zipfian Distribution
Latency (ms)
1.340274 ms
1.422487 ms
0.906916 ms
1.149718 ms
No Cache
Cache Size