CS 425
Nov. 10th, 2024
Group 14: Ryan Wu, Oliver Rogalski

MP3 Report

Design: We created the distributed file system using Go. Consistent hashing is done using the FNV Hash library. Ring IDs for files and nodes are obtained by hashing the file name and VM address respectively. For ring membership, each node has an ordered map that maps network node ring IDs to a membership struct from MP2. Inter-process commands such as create, get, and append use an RPC call with an associated protocol buffer containing the necessary information to execute the command.

For simple file access, each node has a map that maps each file's hash value to the HYDFS filename. This map is used to quickly check if a file is stored at a specified node. File storage is done using blocks to ensure consistency and efficiency. Each file is stored as multiple blocks which are created when appending. Blocks store both the block node that created it and a monotonically increasing ID. When reading a file, the blocks are sorted by both the block node and ID and used to construct the entire file to ensure consistency across replicas.

Caching is done using a least recently used style doubly linked list that holds get command file data in memory. When a file is read, the cache stores the file and keeps track of the total amount of data. If storing the read file causes the storage limit to be exceeded, the oldest node is removed from the list. When a file is appended to, the cache is checked for that file and removed if it is there. This ensures that file reads are not reading an out-of-date file from the cache.

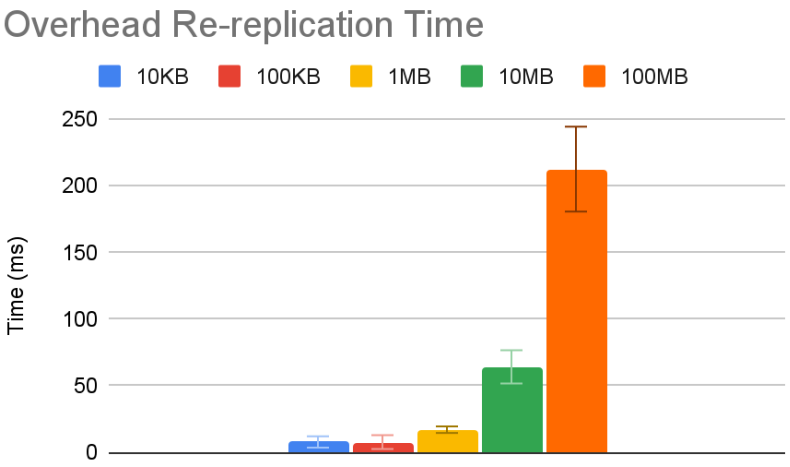
Since the distributed file system tolerates up to two simultaneous node failures, the replication level is 3, which means that each file at a node is replicated in two of the node's successors. When a node joins or leaves the network, every node checks if they have become the main coordinator node for their files and sends re-replication RPC calls to the replica nodes if necessary.

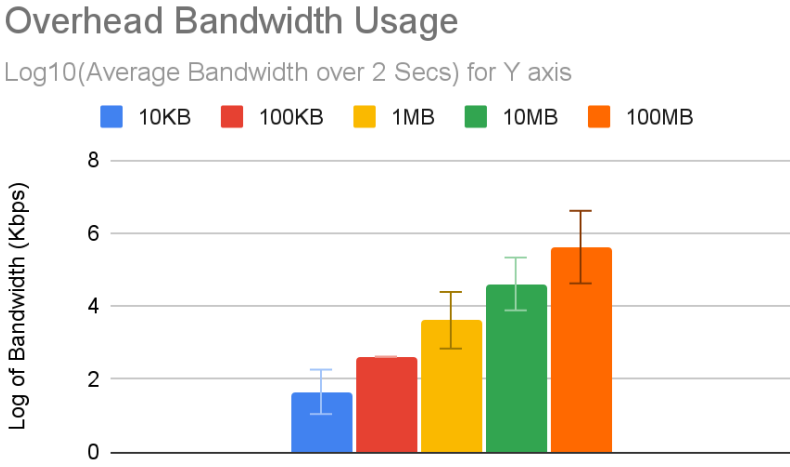
Re-replication and merge are done through Ask and Send calls. An asker node sends its file hashes and corresponding blocks to a receiver node and asks if the receiver is missing any file blocks that the asker node has. The receiver replies with the file blocks that it needs from the asker, and the asker then sends the files to the receiver which copies the file data to itself. This lowers bandwidth usage by only sending file data across the network when necessary and using file hashes to check replicas, not the entire files themselves.

Past MP Use: We used MP2's membership protocol to set up the distributed file network by adding a Hash variable to each node in addition to MP2 variables. MP1 was used to check multiple node logs for errors or specific events when testing commands.

Measurements

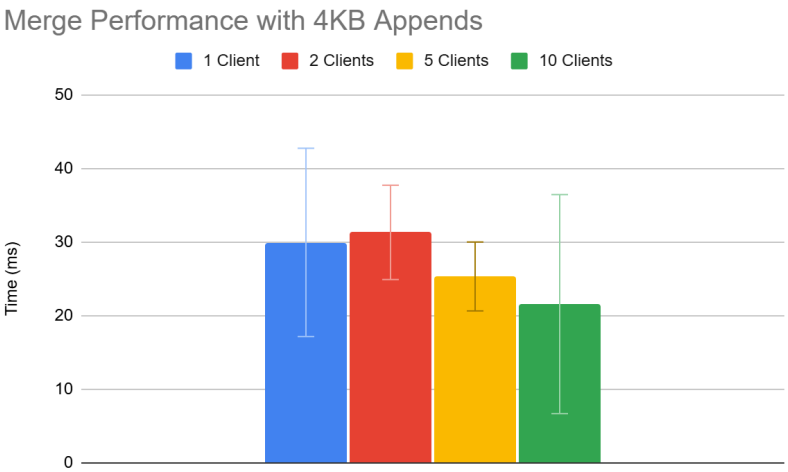
Overhead:

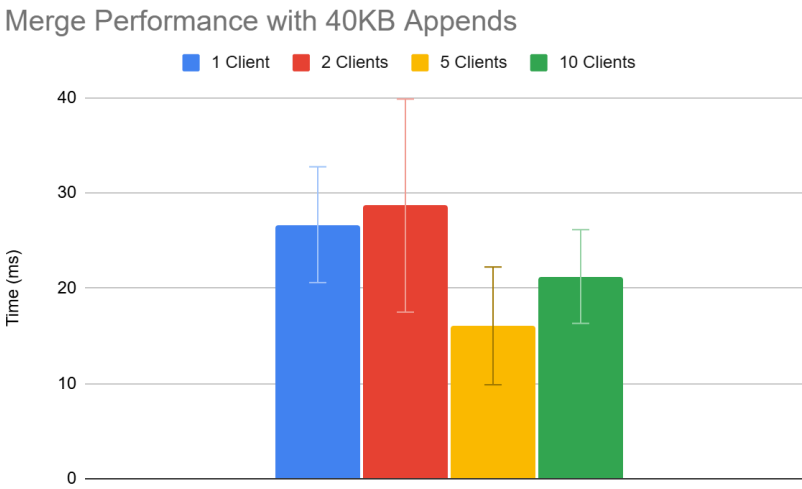




Both re-replication time and bandwidth usage increase as the file size increases. Re-replication time seems to be mainly impacted by file write time, which is why there seems to be no difference in time from 10KB to 100KB but a drastic difference from 10MB to 100MB as the replica nodes need to write larger files. The bandwidth usage started at 40Kbps for a 10KB file and increased by the same factor of 10 that the file size increased by, so the log of the bandwidth needed to be plotted to display the bandwidth for smaller file tests. The bandwidth usage seems to scale at the same rate as the file size.

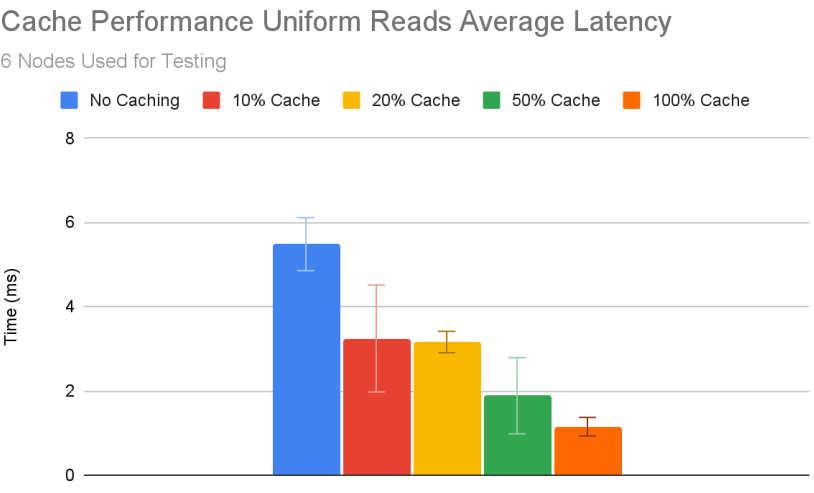
Merge Performance:

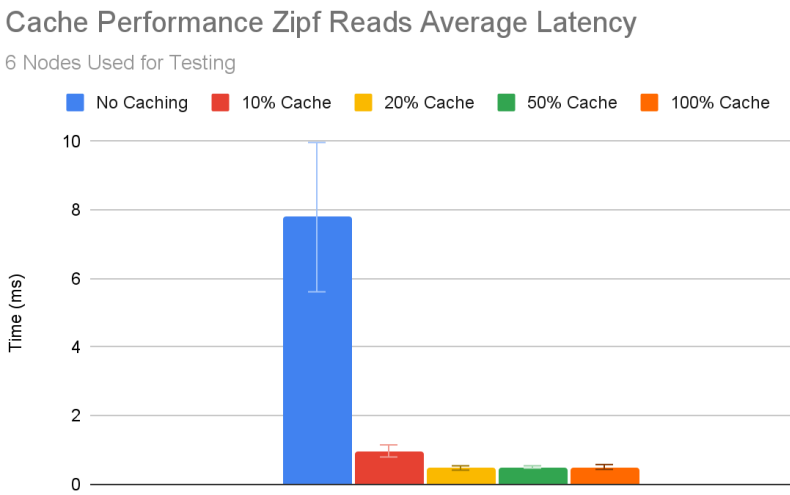




Merge performance is mostly the same for differing numbers of concurrent clients appending as the number of replicas for a file will always be the same and depends more on the random network latencies. There is a slight decrease in the time for merge with more clients as the appends will be more evenly distributed across replicas resulting in less data to merge for a single replica.

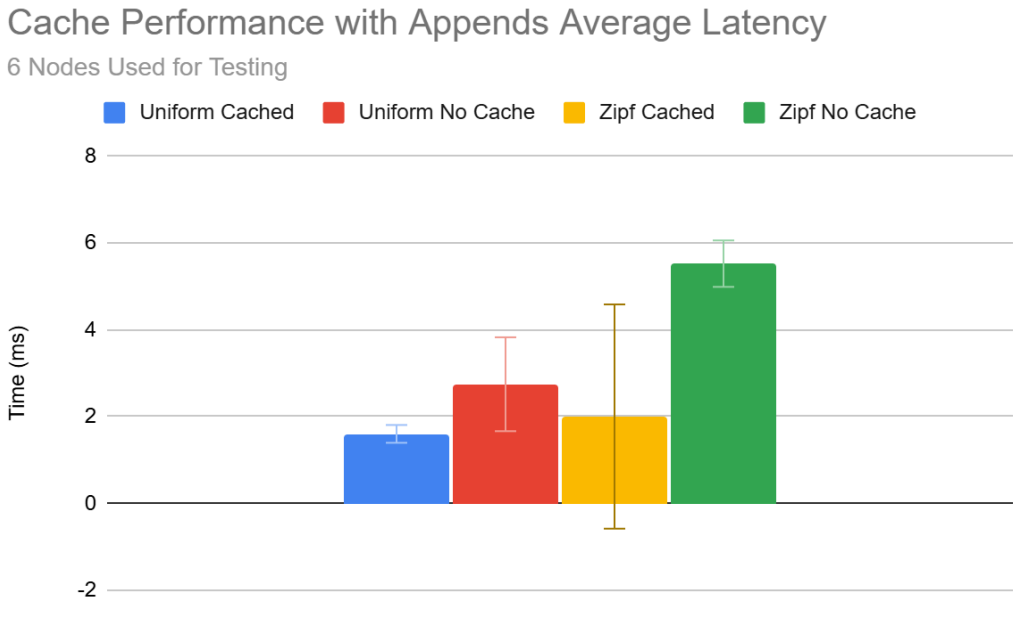
Cache Performance:





Utilizing the cache significantly decreased the average read time, especially for the Zipf distribution tests. For uniform reads, the cache size increasing means more files able to be stored, reducing runtime consistently since each file is equally likely to be read. For the Zipf distribution, increasing the cache size yields diminishing results because the frequently read files can already be stored in the cache. The Zipf tests show that all frequently read files can be stored with 20% of the cache and increasing the size past 20% has insignificant results.

Cache Performance with Appends:



Adding appends to the workload changes how caching works since we have to remove a file from the cache every time we append to it. This slows down future read requests which have to get the file again. The Zipf distribution also runs considerably slower than uniform because the majority of appends are going to a small number of files, which invalidates them from the cache constantly while the uniform distribution will spread appends to files evenly resulting in fewer cache removals.