MP3 Report

by: Nicolas Yepes (nyepes2), Ivo Kusijanovic (ilk2)

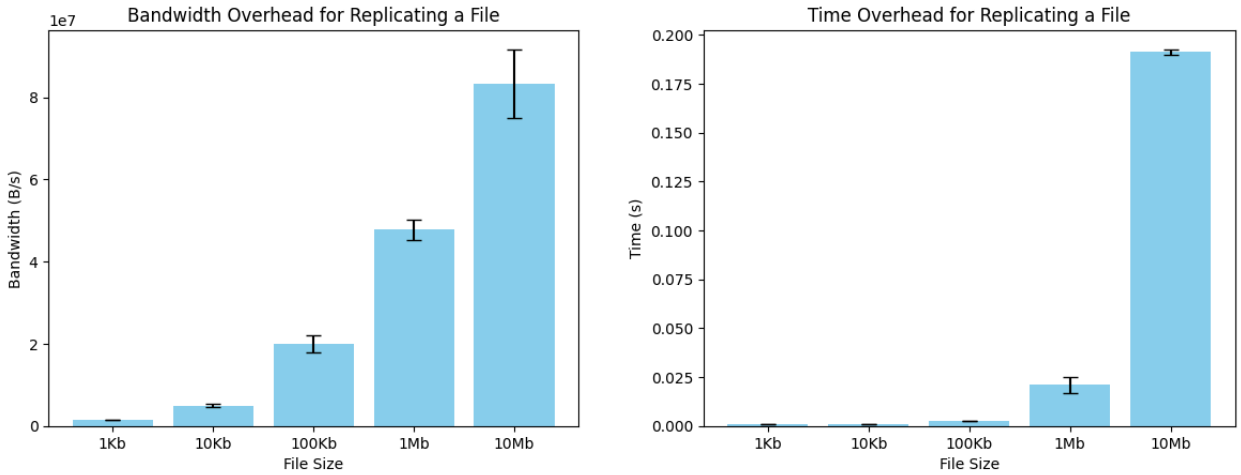
Design: As stated from the specification, the nodes are organized in a ring structure with full membership lists. Each node keeps track of recent updates to each file in a data structure analogous to a cassandra mem-table, which we will also call MemTable. When creating a file the client routes the request to every node that should have that file, but only appends the contents of the file to one of the replicas. Requests such as get and append, hash the requesters id and assigns a deterministic client to as a way to balance the load between many clients and the number of replicas and making sure every client is routed always to the same server so that getting a file after an append will contain the data that the client just appended. On the server, when a node receives an append request it does not write to disk instead it just writes to the in memory data structure that should be a lot faster and increase availability as well as keep track of unmerged changes in a single file across machines. If the MemTable gets too big it will start a merge protocol with the other replicas.

Here we define the *head* of a file as the replica of a file with the lowest id. This will be the leader for managing merges and replications. Merging is done by requesting the *head* to make the file consistent. The *head* will request the rest of the replicas to send their MemTables, which will contain unmerged appends. The head will merge the MemTables and send them back to all replicas. If at this moment the *head* fails, the other replicas will timeout, handle the failure and will tell the user that it failed (Since it is impossible to guarantee 100% availability). The Memtable also stores an attribute about the file version, this helps clients keep track of outdated cached files by just reading what is the version of that file, and if there is no change then they can continue using their cached version, reducing the transmission time to transfer the file over the network.

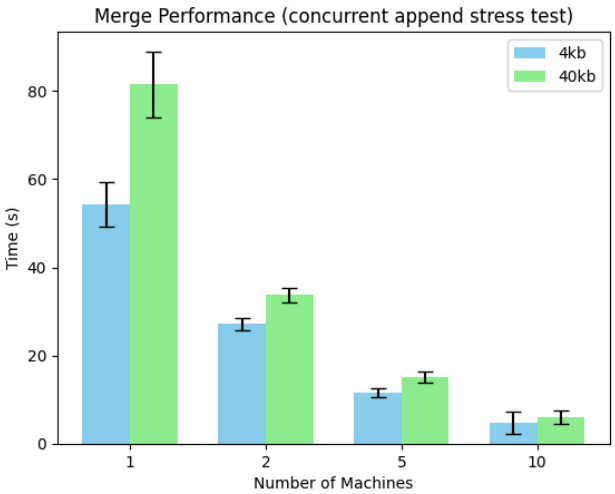
To enable re-replication every node will store a list of all the files it has and who the head of that file is. On new joins we have that the new node will ask its K - 1 predecessors and its successors for all files that it should know about. Here the new node and its successor must edit who is the head of each file they have. Some files will now have as a head the new node, while others will stay as the successor node. Notice that one of the previous replicas will no longer be a replica after the join, however it might still have appends on the MemTable that have not merged. This is why if a node notices that it is no longer one of the replicas it will send its MemTable to its predecessor, which should still be the same node that clients that appended to the leaving replica will now route to. Once a node detects a failure, it will re replicate the files by sending all files that they are the head of to the next available node, unless less than K nodes are available. Each node checks every second on changes to the membership list and does the same thing.

Past MP Use: The failure detector played a big role in creating the replication system and determining who was the node with the file so that queries could be quickly routed to the correct server to handle. From MP2 we kept a list of the current nodes in the system, when a server notices that a node has a replica of a file that they have then it will immediately start the transfer process to another node so that the file can be re replicated. It was also particularly useful in determining which node had a file, and which nodes should a server request to merge files from. Logging was also our main way of debugging and making sure that requests were routed correctly.

Overhead Performance (Rereplication)



The above shows the bandwidth and time of sending re-replications on failure, notice that the scale is multiplied by ten megabytes in the y axis. As we can see an exponential increase in both bandwidth and time which is expected as the file sizes we test increase by a magnitude of 10 between the next. An interesting thing is that the bandwidth std for the 10Mb is substantially larger than the others. We believe this happened as we tried different failures for the rereplication which caused different data to be sent. At times larger or smaller. Also note that the x axis has a log scale while the y axis has a linear scale which is why the graph also looks exponential. It is also good to note that the bandwidth seems proportional to the time it takes, which makes sense since the bandwidth should just be the amount of bytes

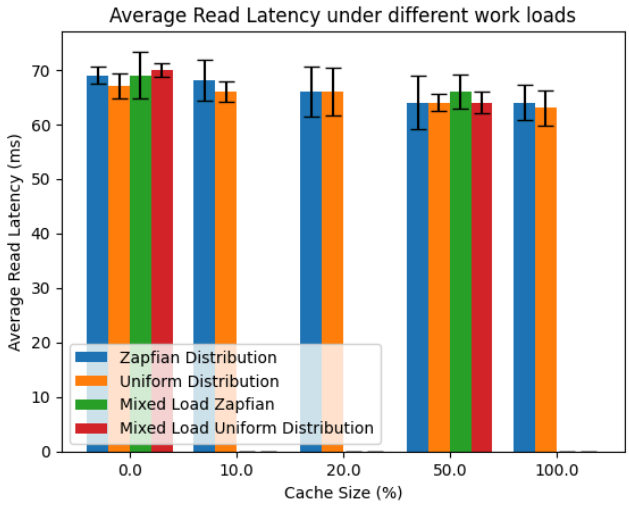


Merge Performance

The figure to the left shows an interesting behavior for the concurrent append stress test. The reason that the graph reduces is because since we are fixing the number of total appends among all clients, we can achieve greater

parallelism and load balancing. Since every node always writes to the same node, we have that if all appends are done by the same node then load balancing will be bad and performance will decrease. Once appends are done by many concurrent machines the appends are a lot better distributed among the system leading to faster average append latency.

Cache Performance



The figure to the left shows that the cache isn't as important as we expected it to be to decrease latency.

Also we notice that mixed load (i.e. append and get) take about the same time per file. Overall, this leads us to believe that either we have an operational overhead that is too large that makes any cache hits irrelevant. Note that we checked that we have just above 30% cache hit on average, so it isn't that our system isn't getting ready to send files. We also tried doing the same experiments over different machines and it all led to similar results. This overhead might be due to small files that were read since when reading files we assume that at least 10 kb of data arrive and we create a buffer of that size. When responding the client will also allocate 10kb of data but will receive a small message with an OK. Since there is small queuing delay and most of the time difference will likely be due to transmission time, which should not take too much. This leads us to believe that our implementation is correct despite caching not improving results that much.