MP3 Report

By Sagar Abhyankar (sra9) and Aditya Kulkarni (aak14)

Design

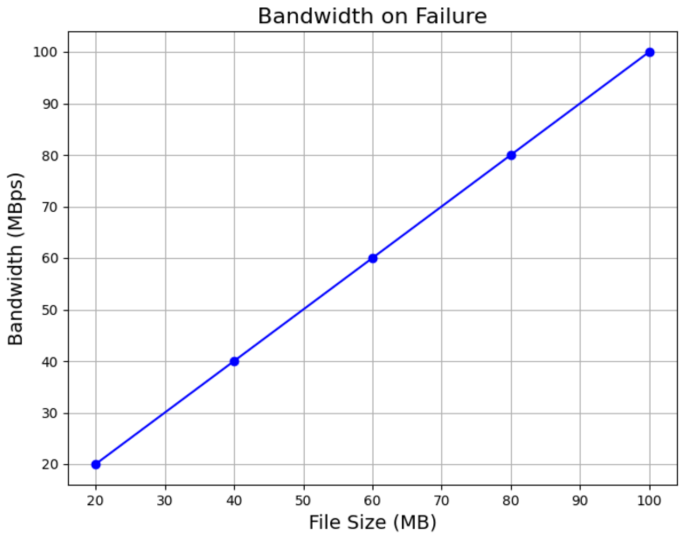
We have used go lang for writing this MP, we do not use the traditional rpc instead we use sockets in go. For this MP, we build on top of previous MP2, specifically using the membership list from MP2. The HYDFS utilizes the failure detection from MP2 and uses it to maintain the number of alive nodes and their nodeHash ids on the ring. The MP2 communicates the membership updates with the HYDFS layer using a self TCP pipe. We maintain a global hydfs TCP port on which all the VMs transfer messages and files, this layer is not used for failure detection in anyway. We store a chord like hashmap in the hydfs layer to store the nodeID (calculated using consistent hashing) and the range of Ids it is responsible for storing, against it. Eg Node29 -> [1020, 29] where 1020 is the node id of the predecessor. This allows us to quickly find out O (1) time where a particular file needs to be mapped or requested from. To tolerate 2 simultaneous failures, we maintain 3 replicas of a file, which is the minimum number as if two of these replicas crash, the third one can be used to re-replicate. We use per key coordinator for file reads and writes. The per key coordinator stores the file, and its two successors store the replicas for this file. We use a push-based re-replication. The coordinator for the file checks periodically whether the file is replicated on its two current successors, if not, it will perform it. When the coordinator for a file crashes, the next successor notices it when running periodic check for ensuring the coordinator for the replica it holds is alive. If it is the next successor, it becomes the coordinator for the file and performs the subsequent re-replication, otherwise it will ignore it since in a few seconds it will start considering the first successor as the coordinator after the nodes table is updated. Each time a file is appended the appends are simply stored logically i.e. in separate chunks and not merged until merge for that file is called. The per client ordering is done with use of lamport timestamps, each client sends a lamport timestamp for a particular append meaning that no two appends from the same client can have the same lamport timestamps! The logical appends are stored from each client separately which helps during merge operation, the operation simply sorts the contents of a per client append directory with lamport timestamp and then queues them. The global queue is then used to merge the file and create a single copy, the replicas also perform merge, they too hold all the appends necessary (to ensure appends are not lost when a coordinator crashes). Every time a write is sent to a coordinator, it ensures that a write is sent to at least one replica to accept the write so that $W = 2$ (write quorum). And for read quorum also $R = 2$, so that in total since we maintain 3 replicas, $N=3$ $W+R>N$ to ensure strong consistency in our design. We use a cache mechanism (which is not limited by size but by time). A cache is invalidated in following three scenarios (1) an append was sent from a client where the file is also cached (2) when file is merged a message is sent out to all nodes from the coordinator to invalidate cache (we understand this doesn't scale well and we should be using a hash checksum to verify if a file was changed) (3) After 60 seconds we deem a cache as stale is removed (purely out of design choice). Elegance in our design comes from the implementation of distribution of concern of routines for housekeeping (ensuring consistency and replication) and the core program logic for read and writes. MP2 was useful since we did not have to implement a separate failure detection system for this MP, and MP1 was used extensively to debug logs for failures and read, writes events. This allowed us to make sure that the system was always in a consistent state by cross verifying logs after a test run. Logical merges (merge op per client) are performed for a client reading before merge happens, the client then only sees appends that it initiated.

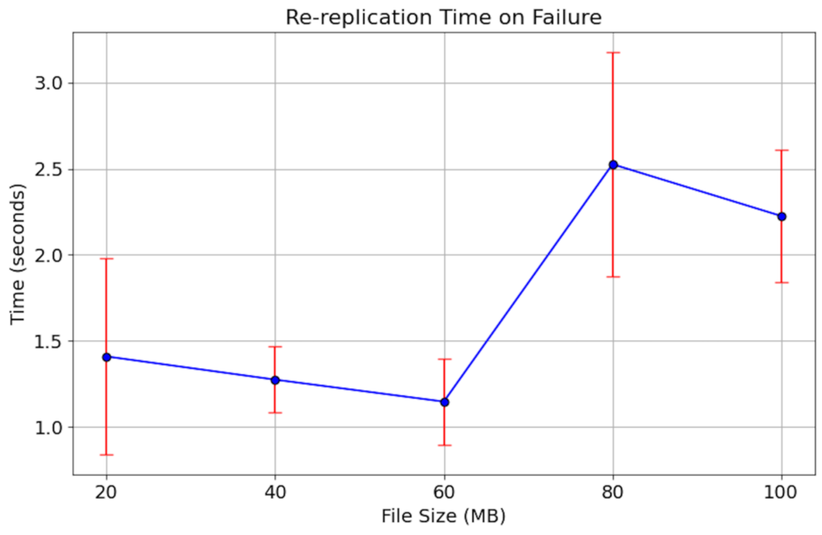
MP3 Report

By Sagar Abhyankar (sra9) and Aditya Kulkarni (aak14)

Measurements:

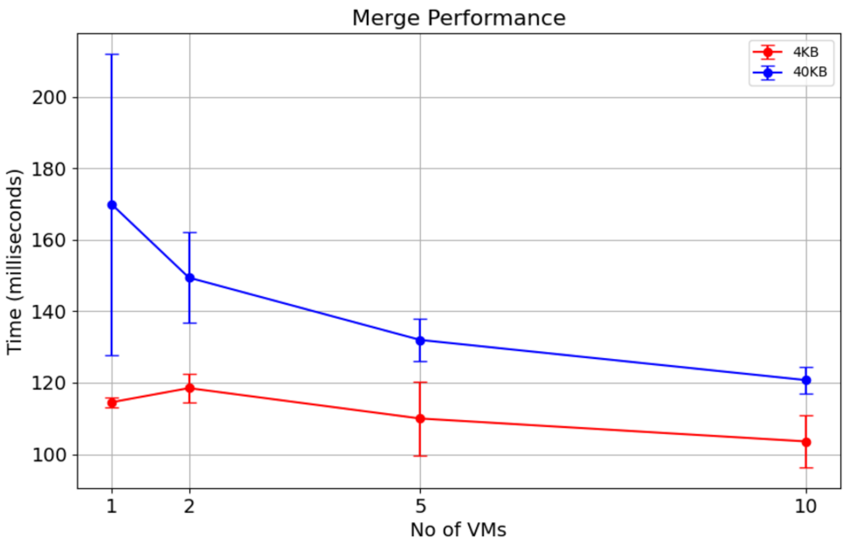
i. Overheads





The bandwidth used scaled linearly with file size (in the case of a single replica failure), this shows that there was no real cap on the bandwidth between the VMs during the file transfer. The meta messages associated with the File transfer housekeeping were in KB's and hence do not reflect at this scale. The bandwidth usage showed no variance on MB scale. The re-replication time was measured as the time it took between detection of loss of replica to completion of write of that replica on the relevant successor. Here we noted that file sizes above 60 MBs took slightly more time than the rest of file sizes. The trend really depended on network speeds. Maybe unrelated but we discovered during testing that each of our VMs were in different time zones, and hence were different distances apart from one another, this could potentially have affected the re-replication time, since the order of replica failures wasn't constant during the testing for different file sizes.

ii. Merge Performance

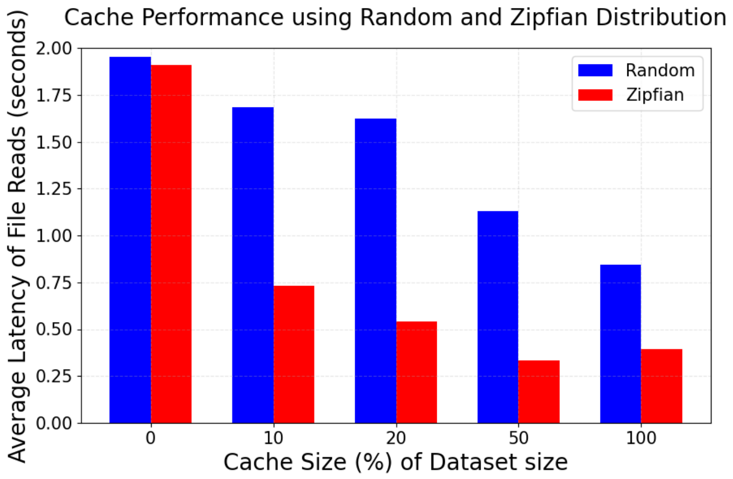


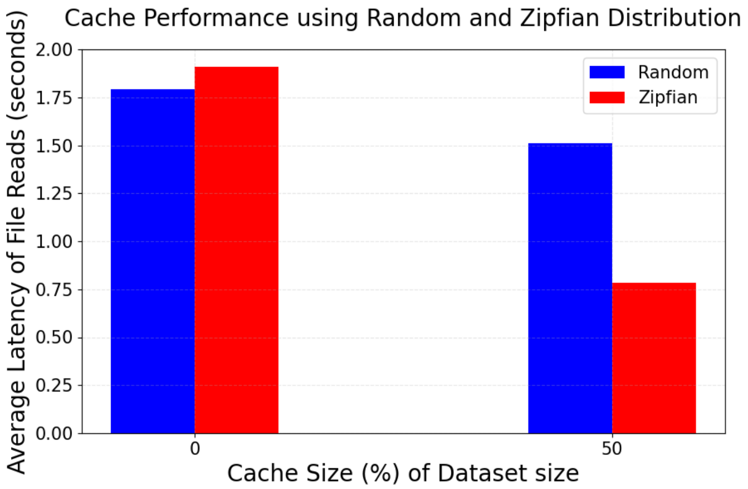
Merge performance graph as seen on the right, was computed as the time it took to merge the file on concurrent appends from 1,2, 5 and 10 clients. As expected, the merge time for 40kb files is larger than that of 4kb files (size matters!). The merge performance improved with the number of concurrent appends, this tracks with the fact that less work needs to be done for ordering (sorting) 100 appends from 10 client's vs ordering 1000 appends from 1 client (we have considered total appends to be strictly 1000). So, the difference at go lang level lies in the sorting of 1000 elements/ lamport timestamps vs sorting of 100 elements 10 times.

MP3 Report

By Sagar Abhyankar (sra9) and Aditya Kulkarni (aak14)

iii. Cache Performance





Figure(a) for test (iii) (0% cache size is without cache) Figure (b) for test (iv)

We implemented LRU cache for testing with a fixed cap size, go over it and the least recently accessed file will be removed. We removed the 60 seconds removal rule we use in our design for these tests. The graphs are as we expect them to be, with increasing cache size percentage you can fit more amount of dataset in the local cache which has fastest access time. With decreasing cache size, the number of cache hits reduce, and more work needs to be done to fetch a file from another VM. We measure the time required to read as the time required to get the file plus the time required to read the contents once, and this is constant across all the tests we performed. Because of the relatively slow time of reads (~2sec per read op) we only could perform the tests on file dataset size of 2000 files. In that, in test iii, as shown in figure(a), we see that Zipfian distribution performs much better than uniform random when a finite cache size is involved. This is consistent with the fact that Zipfian has less entropy meaning it is more affectionate to the files stored in cache, leading to higher cache hits and therefore lower latency times. But when no cache is involved since every operation is a cache miss for both, they have same timings. Two notable observations which we made is Zipfian is faster on 50% Cache size than on 100%. The other is that at 100% cache size we thought both random and Zipfian should perform about the same, but apparently, they don't as it still needs to get the file the first time and random distribution has more first times than Zipfian. For test iv, as seen in figure(b), at no cache, again both perform similar since cache is the only thing differentiating the latency at large. For 50% cache size, some interesting trends emerge. Firstly, when compared to 50% cache size case in figure(a) iii, we see that the performance got worse. Why is that? Well, we invalidate a cache when there are appends from the same client to that file, so that results in lesser cache hits in subsequent read requests. We ran it for 10% appends and 90% reads, and we think the drop in performance is proportional to the appends/reads ratio. A client performing less appends will get the results quicker. Also, in the appends case we have an additional overhead of logically merging the files for the requesting client to ensure consistency.