Sahan Kumarasinghe (sahansk2), Tejus Kurdukar (tkurdu2)
MP3: Distributed File System

Design

First, we discuss the number of total replicas for a given file is replicated (N), the number of replicas for which `getfile` is registered (R), and the number of replicas for which `putfile` is registered (W). To prevent data loss by three simultaneous failures, writes are acked by **W=4** replicas. Then, we would set the number of total replicas to be 4; but the MP3 requirements say N=/=W. Therefore, the minimum number of replicas must be **N=5**. In order to pigeonhole the reads and the writes such that they intersect at one process, we set **R=2**. Therefore, the value of (R+W) in our system is 6. `deletefile` deletes the file at all replicas to avoid mistaken rereplicated by the system (discussed later). Our `getversions` queries all replicas to ask which versions of the file they have; to ensure all fetched versions are truly the latest ones. Otherwise, in order to require that five consecutive version-writes will intersect at one replica, we would have to require that the five overlapping write-quorums intersect at one process. Asserting this would drive the values of W and N up (see HW3 P2b), which would have a detrimental effect on the re-replication bandwidth and file transfer time, which defeats the purpose of using quorum reads/writes to make the system “fast” in the first place..

Files are assigned to replicas using consistent hashing, reminiscent of Chord and Cassandra. Nodes are hashed to a virtual ring, and the filename is also hashed. The replicas which are assigned to a particular file are the N replicas that are immediately clockwise to the file’s hash (where clockwise is increasing hash values).

In our code design, we have three types of “entities” - a client entity, a master entity, and a replica entity. Each node in the system has each of these entities running. Client entities communicate with the master and replica entities to complete tasks requested by the user of our system (via CLI). The master entity (of which there is only one) is responsible for totally ordering all requests (by request time) from multiple client entities. There is only one master entity active in our system, which is implicitly reelected on a detected failure in MP2 as the node with minimum ID in the whole membership list. The master entity is invoked via gRPC over a TCP connection. Replica entities manage file storage, and they communicate with the master and client to serve requests about files. The notion of “replica entity” should not be confused with “replicas” as per the MP3 document - all nodes in the system are replicas. To reduce verbosity, we will refer to “client entity”, “master entity”, and “replica entity” as “client”, “master”, and “replica” from now on.

When a client puts a file to SDFS, it contacts the master replica to identify the replicas to write to. The client then initiates a TCP connection to communicate incoming data to the replica, and sends the file (gunzipped) over the network. The replica names the blob as the sha256sum of the received data, and then ACKs to the client with the content hash. Once the client has the writes the data blob to the write quorum, it issues a request to master to register that write (with the contained hash). The master acquires a mutex (to ensure ordering) and then issues a request to the replicas with the hash to register the blob with SDFS. The replicas create a folder with the SDFSName, and assign the filename of the blob to be the version (as a Unix timestamp). The replicas then ack the master and the master propagates the success to the client.

We also implemented active replication. When MP2 contacts MP3 to notify about a membership list update, each node immediately runs the partitioner on all files (and versions) that it stores, and immediately makes a replication offer to the other replicas that would be responsible for the file and version. The replica then transfers all requested files and versions. The active replication process also results in all replicas containing the maximum 5 versions, bringing the whole replication cluster up to date.

Our project leverages a microservice architecture in order to utilize the MP2 C++ solution, and in order to decouple the MP3 logic from the user interface (for clean, blackbox integration with MP4 in the future). As such, we make use of multiple running binaries (an MP1 binary, an MP2 binary, the MP3 binary implementing this project and an MP3 binary for the CLI). All these programs are essential; therefore, we need to ensure that if any one of the binaries is killed/crashed, then all the binaries are killed/crashed to avoid strange edge cases (e.g. MP3 crashes but MP2 still runs). To achieve this, we run all binaries via Bash script as a single Linux process group, and if a binary exits, the Bash script kills the process group. We emphasize that MP2 still only uses UDP for failure detection, just as the original recommended C++ MP2 submission; HTTP is used strictly for the microservice architecture, and only the MP3 binary in the same process group can talk to MP2 via HTTP (and so that MP2 can notify MP3 of failures on the same node). We strongly desired to use the C++ MP2 submission for its robust failure detector.

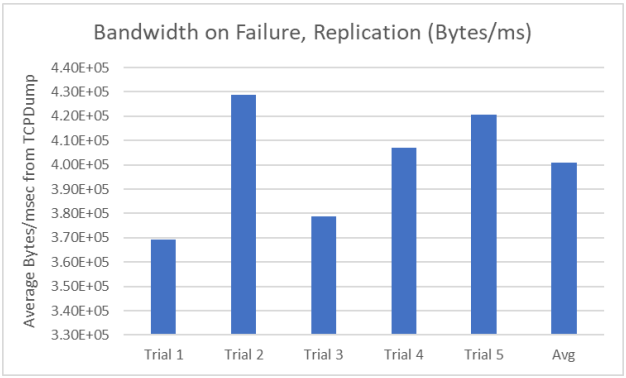
Past MP Use

We implemented some changes to the recommended MP2 submission in order to adapt it for a non-fixed introducer. Instead of programming a new DNS-like program from scratch, we leverage a modified version of our MP1 submission which can run an arbitrary command on a VM. The modified MP2 submission will run the MP1 querier (which can filter based on VMs), and query the MP1 logger which running on the DNS VM. This logger will cat the contents of a particular file, which contains the address/port of the introducer. The MP2 binary will then initiate the introduction protocol with that introducer. The MP2 binary implicitly elects a new introducer as the minimum member in the membership list.

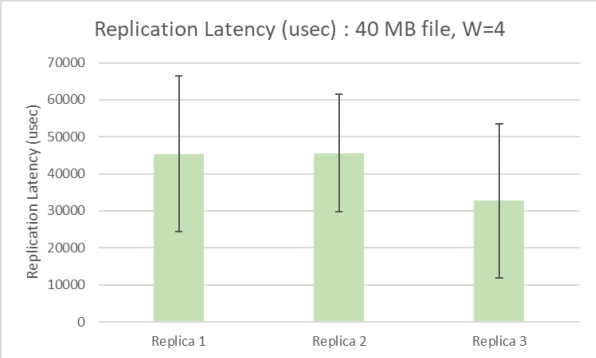
Measurements

To measure re-replication time and bandwidth for a 40 MB file, we performed the following procedure for 5 trials:

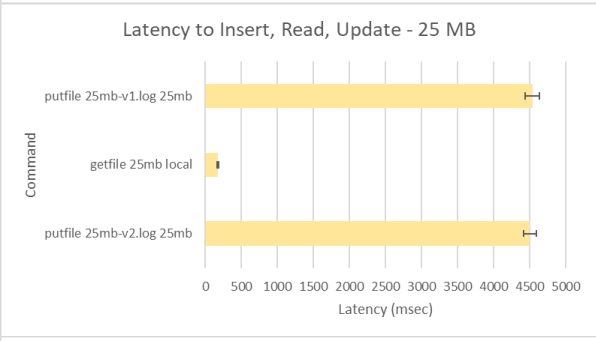
1. Issue command: "putfile 40mb.log 40mb". Since W = 4, 4 machines will store the file.
2. Start tcpdump on one machine, filtering packets by VM host ips.
3. Crash one machine containing the 40mb file.
4. Measure delta for replication by marking timestamps in the code. Collect replication deltas on the 3 machines that did not fail.

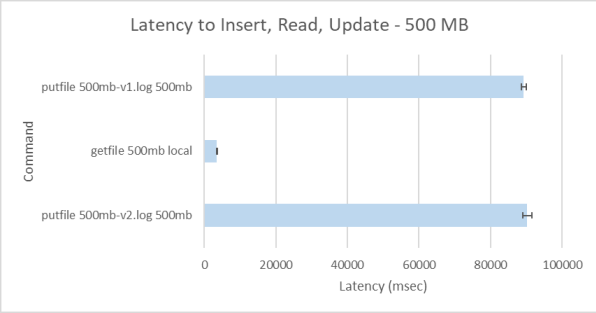


5. Collect tcpdump to analyze bandwidth.
The results are shown in the graphs on the below. On average, it takes approximately **41248.13 usec** to replicate the file. The bandwidth is on average, **400800 Bytes/msec** with a stdev of 25909. This is despite taking ~7 seconds for the putfile to execute; replicas are exchanging already compressed files, whereas a putfile command must go through the process of compressing.

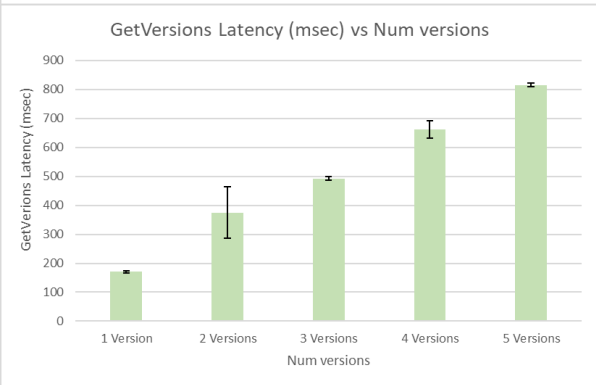


We also measure latencies of inserting a file, getting a file from a different machine, and updating a file, averaging over 5 trials. The following graphs depict latencies for insertion, retrieval, and updating for a 25 MB file and a 500 MB file. One immediate observation is that our system is quite optimized for reads; "putfile" takes far longer than "getfile." Again, this is because of compression; it takes significantly longer to compress a file and upload it to SDFS than to retrieve a file and decompress it. The "putfile" takes on average **~4500 msec** on a 25 MB file, and **~90255 msec** on a 500 MB file. In contrast, "getfile" takes **~172 ms** and **~3417 msec** respectively.





The latency of "getversions" appears to scale linearly with the number of versions. This makes sense; we retrieve one version of a file at a time from a replica. The data below was collected by using five versions of a 25 MB file and fetching on a VM not containing the file locally.



Finally, we measured the delta of storing the entire Wikipedia corpus onto SDFS, for 4 machines and 8 machines. Notice that the difference is minimal between the two; it takes **~1904 sec** on average for 4 machines + master, and **~1914 sec** for 8 machines + master. This makes sense, since a file is assigned to 5 replicas with a write consistency of 4 in our system. The additional machines are not storing the file.