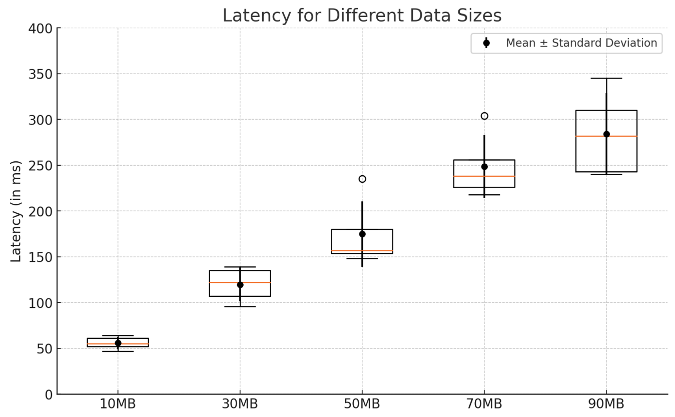
Distributed Systems (CS 425) - Fall 2024 - MP 3 Report

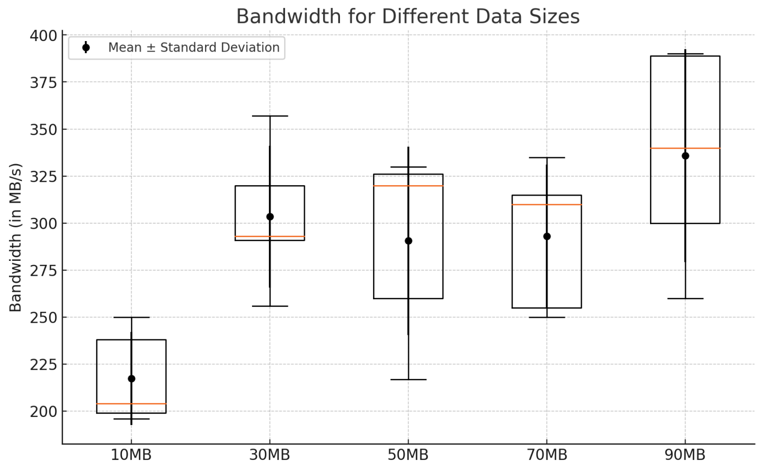
Akhil Sundaram(akhils7), Anurag Choudhary (anuragc3)

Design Overview: The distributed file system is implemented using go tcp sockets, grpc's and ring hash algorithm(similar to chord ring). (**Ring**)We use MurmurHash for consistent hashing with 2^10 or 1024 points in the ring. When a node joins the system, we compute its location in the ring through hostname (not ip+port to ensure the node joins at the exact same position everytime. This change allows for easy debugging). The new node initializes the ring based on its membership list. The ring data structure is then updated in every node that detects a join or failure. We use channels in golang, to directly pass a signal/data from membership change events to the ring component. (**Replication**) We implemented a pull-based strategy for file replication. On node **join**, the new node computes the ranges of files it is responsible for - both primary and replication. Instead of getting exactly what ranges of files it should request from each successor/ predecessor node, we calculate the overall ranges of files we need to store at that node. It will then make two requests to its successor and predecessor nodes. One, to get the files within its file range. Second, to pull the files that it requires based on the replication list.This is to ensure we don't pull data we don't require/ already have. Every other node in the ring, check their file-ranges to decide whether to drop files. On node **delete**, a similar signal is passed from the failure detector, and every node recomputes their rings. The first replica number of nodes after the index of the deleted nodes will pull files from the replica number of nodes behind them. After a merge this in-memory table is refreshed, and the separate appends in disk are concatenated into one file. (**MD5**) MD5 Hash is used to find differences between files when we are performing appends and merges, since we cannot rely on timestamps for ordering the appends (since appends can come out of order).For the (**File system**), when we trigger a create command for a file, we first calculate the hash of the filename and get its ID using out hash function. We check the first successor primary node (and subsequent successor replica nodes) that should store the file. Since the file names are supposed to be unique, we have added a check in the create functionality to error out if there is a clash because of duplicate filenames. Once this is done, we send TCP connection requests to all 3 nodes (Write Quorum = 3, ensuring string consistency) to save the file in the remote VMs file system, and also create entries in the in-memory representation of the filesystem on the above nodes. These in memory representations will store file attributes which will later be used for lookups and checks, making this process easier and simpler. Few important points to note are that the timestamp that is being stored for the file is the timestamp when the initial command is triggered, so that the same timestamp is saved). (**Appends**)When we send an append request for a filename, we ensure the same initial process is followed, and we follow the same quorum level as in write (N = 3). On each node, when there is an append to a file, we store it in two different places. The actual append to the file is stored in disk with a sequence number associated with it. We also store an in-memory table with the details of each append, including the md5 hash of the append file. We also store a timestamp of each append. These timestamps are calculated at each client when sending the appends. (**Merge**) When we call a merge from a client, it sends the request to the first successor of fileID. This node acts as the leader to merge all the appends based on the in-memory append table. We use timestamps as the sequence numbers as they are set at each client before sending the append request. Therefore, when we order appends based on timestamps, they are always in the correct order for all appends from the same client. (**Get**) When we call get we ensure that a N=2 quorum is met before the file is returned to the client. If a quorum is not met, then we request the primary node to return the content which it has. For the quorum condition, we check whether timestamps and the md5 hashes of the files are the same. For any two similar timestamp + hash file responses, we output that as the final response. (**Cache)** - When a file is requested, we first check if it exists in the cache (in-memory table and on disk. If the file is not found in the cache, we send requests to its corresponding HDFS nodes to retrieve it. If the file is present in the cache, we perform a 'preflight' of the cached file's metadata, md5 hash. If there are inconsistencies with md5 hash, we invalidate the cache.

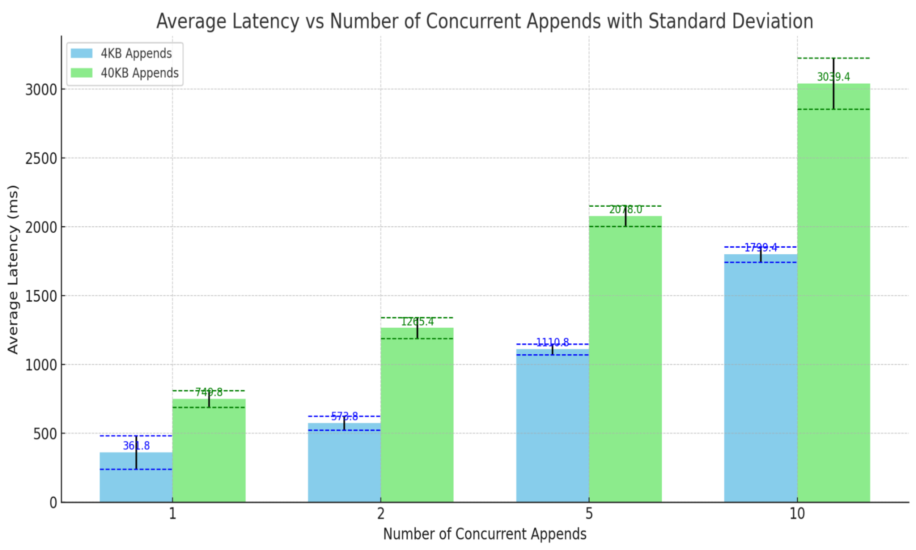
MP1 Usefulness: MP1 was used to grep the logs that are generated on the VM locally, and search for specific messages to help debugging. (We generate logs for every action - create,replication,get,merge,etc.) It is also being reused when we want to invoke the suspicion logic on all VMs simultaneously (sending a cmd on all vms, and using that to invoke a signal on a shared channel).
MP2 Usefulness: We use the membership and failure detection mechanism, to identify when nodes are leaving and joining the ring. We integrated this by creating channels that our ring listens to. Each node will use this to detect joins and leaves. Ring will replicate data (pull data) from other node or to drop data it should not contain based on the channel events.

Measurements:
[Part 1 - **OVERHEADS**]

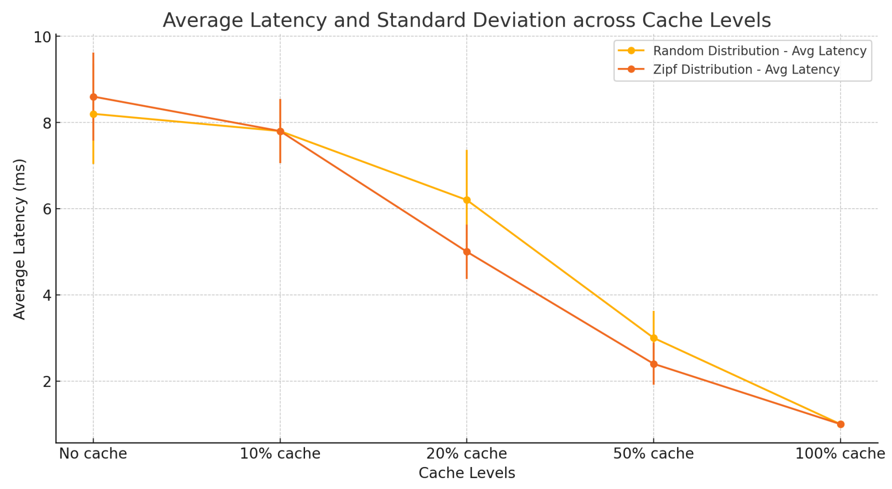




We calculated total latency of replication by marking times between when a deletion is detected by the failure detection, and the point when all data has been pulled.For bandwidth, we calculated it by using time taken and response payload.If we had calculated the time from when we failed the node, every datapoint would increase by 2 sec (+5 with suspicion). **Data**: With latency, we see it increase linearly for larger files as expected - it takes longer to get larger files. For bandwidth, we initially see a lower value which spikes with increase in data. Then it more or less stays constant. Our understanding is that the actual bandwidth capacity is higher, but is not seen when transmitting less data (~10MB). The actual realistic bandwidth should be around 300MB/s.
[Part 2 - **Merge Performance**]
We measured the average latency of the merge operation by timing from when it's initiated on the primary replica until it's fully propagated to all replica nodes(the entire file is sent after merging to ensure consistency across replicas). Each VM concurrently performed 1000 append operations to simulate high load. Using this methodology, we generated the data shown below, which reflects the system's performance under these conditions :



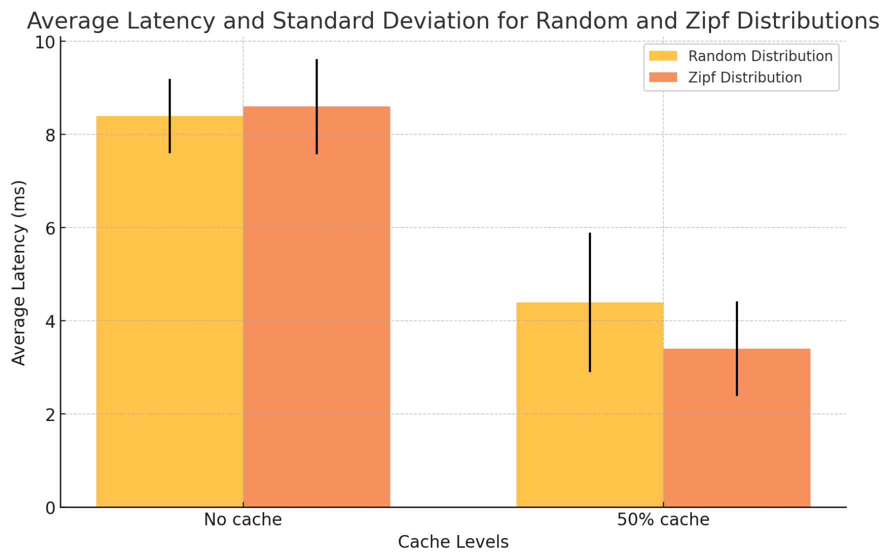
Data: The average latency for 4KB appends increases linearly with the number of concurrent appends. Both 4KB and 40KB appends show significant increases when scaling from 2 to 5 VMs, and again from 5 to 10 VMs. We expect this as(1), merge has to iterate through all append chunks in disk, (2)we send the final merged file again to replicas.The similar trends in both graphs suggest that this distributed filesystem's performance is primarily constrained by its ability to handle simultaneous disk read/write operations (which is how merge operation compiles all the appends).



[Part 3 - **Cache Performance with gets**]
For calculating the cache performance, we create 10,000 files of 4KB and push them onto the distributed filesystem, naming them 1,2,3….. 10,000.txt. Correspondingly, our cache size becomes 4MB, 8MB, 20MB and 40 MB respectively. This naming convention allows us to get files according to random and Zipf distribution as we can add weights to each filename stored. After the files are created, we perform 25,000 gets, and the filenames to get were chosen following the random or zipf distribution mechanism. The latency of get request is then calculated over all the requests, and averaged out for each request.

Data : As expected, increasing the cache size reduces the per-request latency. Few conclusions that we can draw from this are that there is little difference between the "No cache" and 10% cache data points due to our cache check mechanism. Even with cached entries, we send preflight requests to ensure they aren't outdated, which adds latency and diminishes the cache's benefit. However, once the cache size reaches 20%, this overhead is overcome, resulting in a significant improvement. Furthermore, the Zipf distribution outperforms the random distribution because our LRU cache retains frequently accessed files longer. With Zipf's pattern, popular files remain in the cache, leveraging the LRU mechanism more effectively.

[Part 4 - **Cache Performance with 10% appends, 90% get requests**]



The process for Part 4 remains more or less the same as part 3. Instead of doing all get requests, we split it 90:10 for gets vs appends using a random.

Data : In our cache implementation, if the preflight requests for the cache entry returns that there are appends to the file made, then we make sure that the cache entry is invalidated and requests for the file are sent out again. Keeping this point in mind, we can see that the average latency for 50% cache has increased slightly for both the random and Zipf distribution. This can be attributed to the 10% appends that happen and cause additional delays of fetching entries again , as they might have had appends. We invalidate all cache entries with newer appends. Another point to note is that for No cache scenarios, both Random and Zipf distributions have nearly the same average latency, since there is no benefit of Zipf distribution in a no cache scenario.