CS425 MP3 REPORT

Yixuan Li, Netid: yixuan19; Tong Wei, Netid: twei11

November 11, 2024

1. Design

Algorithm and mechanisms

1. **Unified Hashing of Node IDs and File IDs on a Ring:** We use a consistent hashing approach where each node ID is prefixed with "node_" and each file ID with "file_" to ensure there are no hash collisions between node IDs and file IDs.
2. **File Replication Across Successor Nodes:** Each file is stored on k successor nodes, where k is set to 5 in our design. This allows us to handle up to two simultaneous failures while ensuring that more than half of the nodes still retain a correct version of the file. Thus, $2 + 2 + 1 = 5$ replicas are sufficient to meet these requirements.
3. **Re-replication Process:** Using a `membership_list` with a ping-ack protocol, we maintain an up-to-date list of all nodes in the membership list and track the files stored in the distributed file storage system (hydfs). Each node runs an independent process that periodically scans its files and calculates the responsible nodes (those tasked with storing each file). If a node finds that it is responsible for a file, it sends a `requestfile` to other current responsible nodes to ensure that it has a replica. In this way, re-replication is maintained. To guarantee file correctness, at least three copies of the file are checked, and the common version is kept to ensure the correct file version is always in the majority. This majority rule ensures that reads and writes consistently access the correct version.
4. **Ordering with Time Vector:** We use a time vector for ordering, which ensures that the order of file operations is preserved.
5. **Merge Operation:** The first node initiates a merge request to other responsible nodes, comparing file versions. If the files are consistent, the contents are merged, and the unified file is saved in a designated directory. This merged version is then sent back to all responsible nodes, resulting in a new, updated file version.
6. **Append Implementation Using File Blocks:** In this design, we handle the `append` operation by using file blocks. After each merge, a combined file is created by merging an `original` file (the previously merged file) with multiple `append` files (files resulting from each merge operation). This process combines one `original` file with multiple `append` files sequentially into a new `original` file after each merge, preserving the order of changes.

Consistency

- **Unified Hashing**: Ensures unique node and file IDs on a hash ring, avoiding collisions.
- **Replication**: Stores each file on five nodes, allowing majority-based version control to handle failures.
- **Time Ordering**: Uses a time vector to maintain correct operation sequence.
- **Merge Process**: Reconciles file versions across nodes, ensuring all replicas converge to the latest version.

Past Mps Use

- **MP1**: Utilized `grep` for local log debugging, which greatly improved the speed of identifying bugs within the virtual machines.
- **MP2**: Responsible for maintaining the `membership list` for the entire system and managing the global file storage.

2. Experiments Results

Experiment 1: Over-heads

From the plot below, we can observe that overall bandwidth exhibits a linear relationship with file size, which aligns with our expectations. This linearity reflects the common-sense notion that transferring larger files requires proportionally more bandwidth. It is worth noting that the observed errors primarily stem from the maintenance of the membership list, global file metadata, and file request headers. These factors contribute additional overhead that is not linearly related to file size, thereby introducing slight variations in bandwidth measurements.

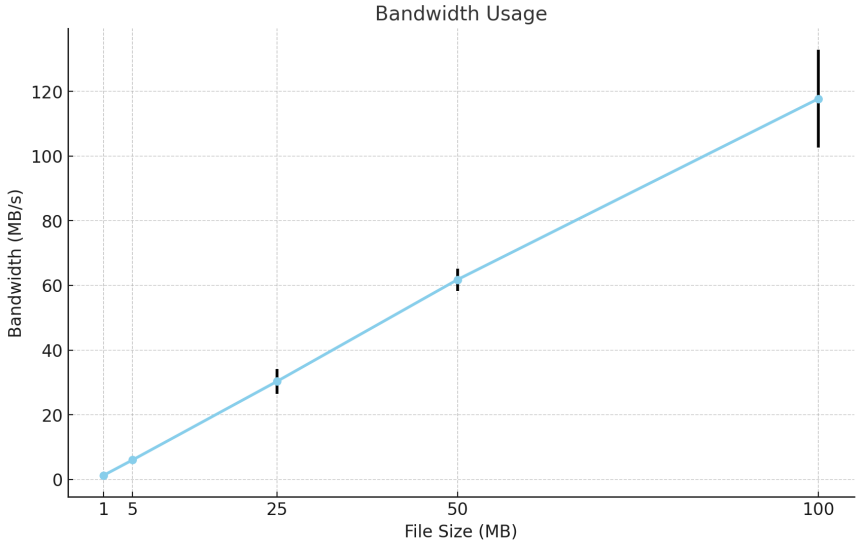

Figure 1: Bandwidth vs file size

Experiment 2: Merge performance

In the merge performance tests, we observe that the merge time remains generally consistent across different numbers of agents performing concurrent appends. This aligns with our expectations, as the merge process is unaffected by the number of agents; it depends solely on the number of files and agents involved in the merge. Additionally, we find that merging 40KB files takes longer than merging 4KB files, as anticipated. However, the increase in merge time is not directly proportional to the file size. We attribute this to our implementation approach: files are divided into distinct chunks, with each chunk uniquely identified by its filename. Since the append operations on each server maintain consistent filenames across replicas, the merge process only requires filename comparisons rather than examining the entire file content. This optimization accelerates the merge and minimizes the overhead associated with larger file sizes.

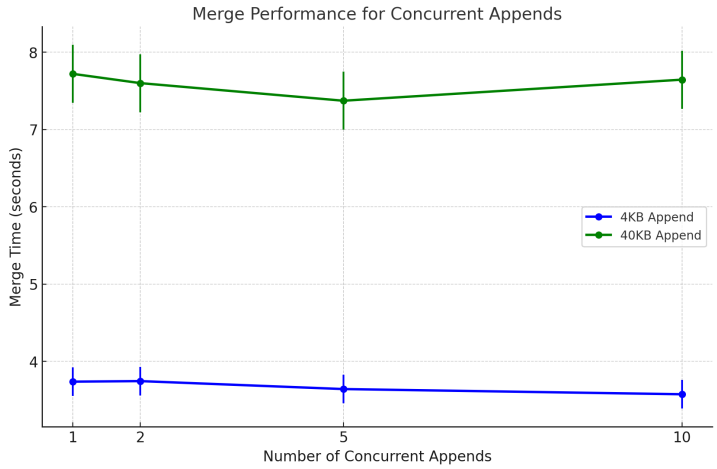

Figure 2: Merge performance vs number of append agent

Experiment 3: Cache Performance

The plot below demonstrates the impact of cache size and access patterns on read latency. Without caching, latency remains stable across all access patterns, as each read directly accesses the file system. With caching enabled, increasing cache size significantly reduces latency, especially when following a Zipfian distribution, as frequently accessed files are cached more effectively. In random access, larger cache sizes are needed to achieve similar performance gains due to the uniform access pattern, which reduces cache hit rates in smaller caches. Overall, the results align with expected behavior: cache size and access patterns significantly influence read performance.

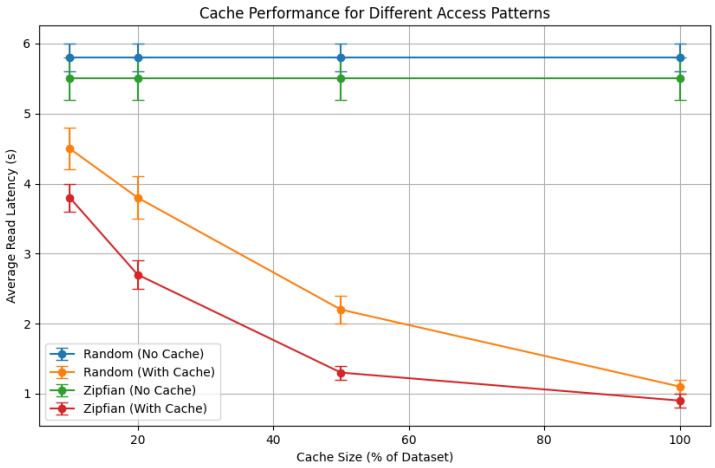

Figure 3: Cache Performance vs cache size

Experiment 4: Cache Performance

The plot below illustrates the impact of client-side caching on read latency with a mixed workload (90% reads and 10% appends). The results show that enabling caching significantly reduces latency, particularly under the Zipfian access pattern where frequently accessed files remain in the cache. In contrast, uniform access requires more cache space to achieve similar latency reductions due to the even distribution of file accesses. Standard deviation bars highlight the variance in latency across trials, indicating the consistency of these latency trends. Overall, these results align with expected behavior, where cache size and access patterns strongly influence read performance.

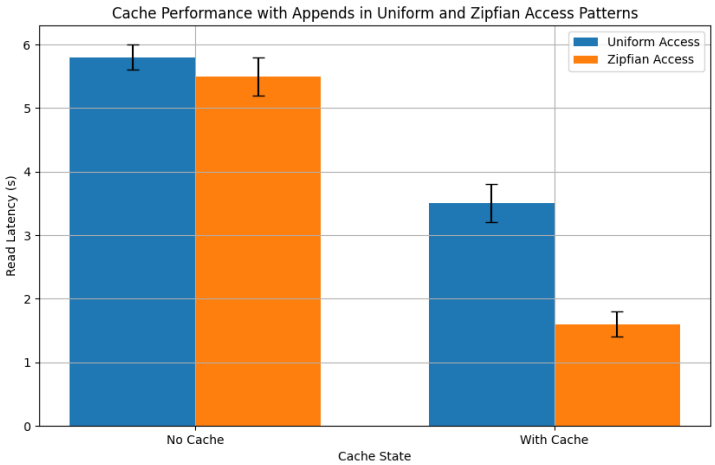

Figure 4: Cache Performance with Append vs different distributioon