Design: Our HyDFS implementation is a client-server model where the Client acts as the user interface to perform file operations create, get, append, and merge, and the Server manages the file storage, replication, and client requests. We define global structure for client and server requests and responses using TCP to communicate operations reliably. We also have a member info struct to store server metadata (ring id, status, address) and a HyDFS server struct (holds hashring, server map, membership list, replica map, etc.) to perform server operations.

Consistent Hashing: to map servers and files on a **hash ring**. Each server is assigned a unique Ring ID using SHA1 hash on its address and servers are arranged in a sorted hash ring based on Ring ID. For each file, the primary replica is the first server in the hash ring with ID >= to file's Ring ID (SHA1 hashed) and subsequent replicas are placed in the servers after primary. There can be up to 2 failures, so we ensure a replication factor of 3 where 3 copies of the file are stored on different servers. After the primary server, files are replicated on the next 2 successor servers in the ring.

Failure Detection: Use MP2 heartbeat and timeouts (adhering to SWIM protocol) to check for failures. Each server gossips to maintain a membership list of active servers in the system.

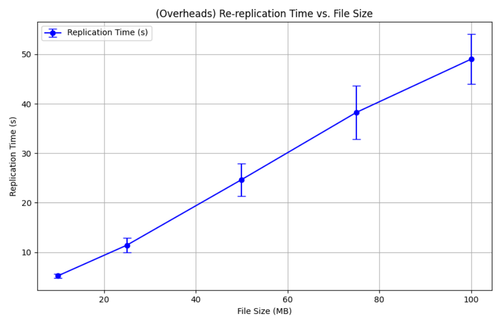
Replication:
Active Replication: on file creation, it is stored on the primary replica, then the primary replica asynchronously sends replication requests to designated replica servers to store replicas. On append, the primary replica updates the file and propagates the append operation to all replicas.
Re-replication: servers check if they need to replicate files to other replicas periodically. Iterate through all files and ensure all replica servers have that file. When a server failure is detected, the system identifies affected files and replicates them to new replicas. This maintains the replication factor so that our data is available and consistent. If the primary replica fails, the system reassigns a new primary from the remaining replicas so write operations still work.

Client-side Caching: when a file is fetched (GET), if not in cache, client retrieves it from server and adds to cache. To ensure client-side cache consistency, after write operations (APPEND) client invalidates cache for the affected file to prevent stale reads. Uses LRU eviction policy.

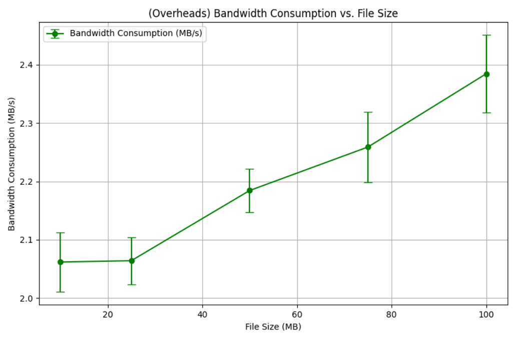
Client Requests: (ls, store, get from replica, list mem ids retrieve info from server struct)
To ensure **consistency**, all write operations are handled by primary replica and replicated after.
CREATE: client sends CREATE req with LOCAL file and HyDFS file. Server checks if the file exists and creates HyDFS file in .files directory. Server initiates replication asynchronously.
GET: client sends GET req with local file and HyDFS file. Server forwards to primary replica and sends to client. Client writes received content to LOCAL file and **caches** it.
APPEND: client sends APPEND req with LOCAL and HyDFS file and content. Server appends content to HyDFS file and propagates req asynchronously. Client invalidates cache for file.
MERGE: client sends MERGE req with HyDFS File. For consistency, merge operations aggregates content from all replicas in hash ring order to merge, and writes unified content back. We aggregate based on hash ring order to ensure append operations are reflected uniformly across all replicas.

Past MP Use: MP2's SWIM membership protocol was used to propagate each server's membership list and to detect failures which is essential for re-replication. MP1's log grepper was used to query log entries at each machine to test the success of file operations.

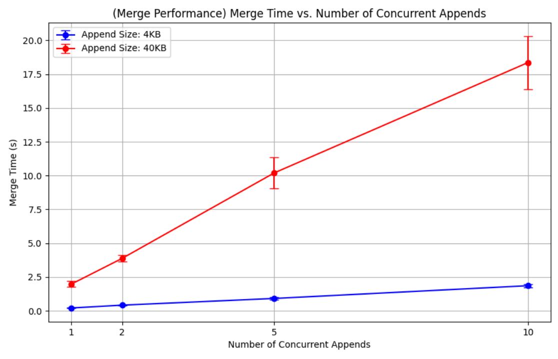
Measurements:
(i) There is a linear trend in replication time as file size increases. This makes sense since larger files take more time to replicate due to the increased amount of data that needs to be transferred and processed.



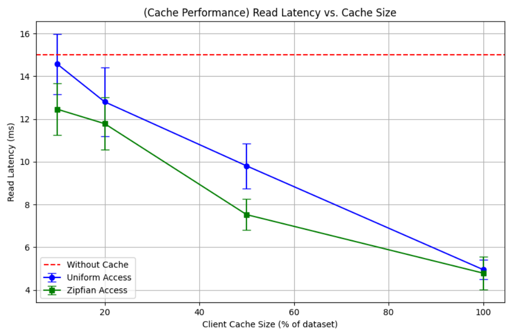
There is a slightly increasing bandwidth consumption with larger file sizes. This makes sense because bandwidth usage per MP remains relatively constant, but larger files result in higher total bandwidth consumption during replication.

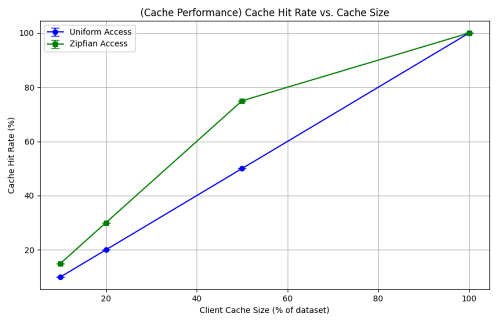


(ii) 4KB: merge time increases with concurrent appends. 40KB: merge time increases at a steeper rate. This makes sense because more concurrent appends increases contention and synchronization overhead which leads to higher merge times. Larger append sizes have long merge times since more data needs to be reconciled.

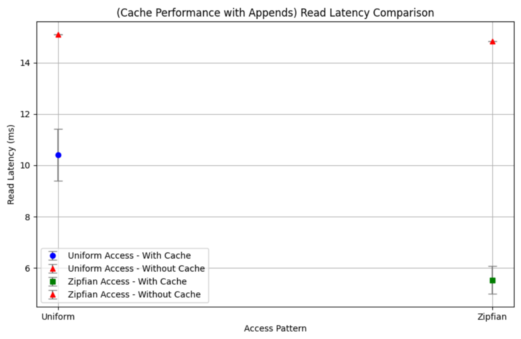


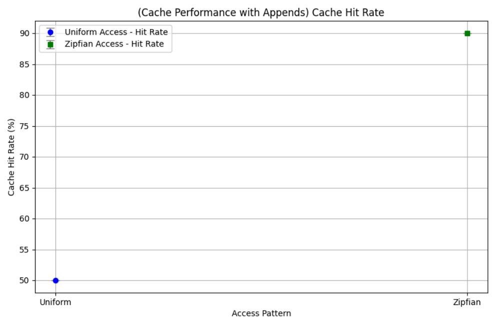
(iii) Uniform access: read latency decreases as cache size increases. Zipfian: more significant decrease. Without Cache: read latency constant. This makes sense because in Zipfian access cache has higher hit rates due to skewed access (see graph on right) so it will have more performance improvements than uniform access. The trend without cache makes sense as it doesn't contain performance improvements from caching.





(iv) Uniform access: average read latency ~9.78 ms with cache, without cache is constant at 15 ms. Zipfian access: average read latency ~8.56 ms with cache, without cache constant at 15 ms. This makes sense because read latency decreases with cache compared to without due to caching performance benefits. Zipfian benefits slightly more from caching compared to uniform due to higher cache hit rates (see graph on right) because the skewed access patterns favor frequently accessed files. This makes sense because higher hit rate in Zipfian access (due to concentration of accesses on subset of files) enhances cache effectiveness.


(Cache Performance with Appends) Read Latency Comparison
Read Latency (ms)
Access Pattern
Uniform
Zipfian
Uniform Access - With Cache
Uniform Access - Without Cache
Zipfian Access - With Cache
Zipfian Access - Without Cache


(Cache Performance with Appends) Cache Hit Rate
Cache Hit Rate (%)
Access Pattern
Uniform
Zipfian
Uniform Access - Hit Rate
Zipfian Access - Hit Rate