Algorithm and Architecture

The system uses a Gossip-based protocol to maintain cluster membership. The Leaders gossip among themselves to prevent network partitions. If a non-leader node loses contact with all Leaders, it will restart and rejoin the cluster with a new ID. The FileManager provides APIs for local file operations on each node. Files in HyDFS are split into metadata and blocks. This block-based storage architecture allows independent replication and versioning of metadata and blocks.

Replication Level

We use **N=3**. Each file's metadata and blocks are replicated on the first three successor nodes on the consistent hash ring, providing fault tolerance against up to 2 simultaneous node failures.

Consistency Requirements

Write operations (Create/Append) use a variant of Two-Phase Commit (2PC) with write quorum W=2 and a coordinator-based lock:

1. **Phase 1 - Buffer**: The client acquires a write lock from the coordinator (first replica). It sends all new blocks to N replicas, which store them in temporary buffers (`BufferedBlockMap`) without committing. The client waits for at least W=2 acknowledgments.
2. **Phase 2 - Commit**: The client updates metadata with an incremented `Counter` to all N replicas and releases the lock. A background worker (`workerLoopBuffered`) periodically commits buffered blocks when their counter matches the metadata counter.

This ensures **per-client ordering** (via lock), **atomicity** (either all or none of the blocks are committed via counter matching), and **durability** (W=2 quorum ensures data survives single node failures).

Read operations use a **read quorum R=2** to satisfy Read-My-Writes:

1. The client requests metadata from all N replicas and waits for R=2 responses.
2. It selects the metadata with the highest `Counter` (latest version).
3. Blocks are fetched in parallel using R=2 quorum and reassembled locally.

Re-Replication and Merge

Merge is a background process that ensures eventual consistency and repairs failed writes or node failures. The `jobCreator` goroutine periodically triggers Merge (every 5 seconds), which pushes all locally stored metadata and block IDs into work queues. The `workerLoopMeta` and `workerLoopBlock` goroutines consume these jobs, check alive members from the Failure Detector, and send local data to the file's assigned replicas (found via `GetReplicas()`).

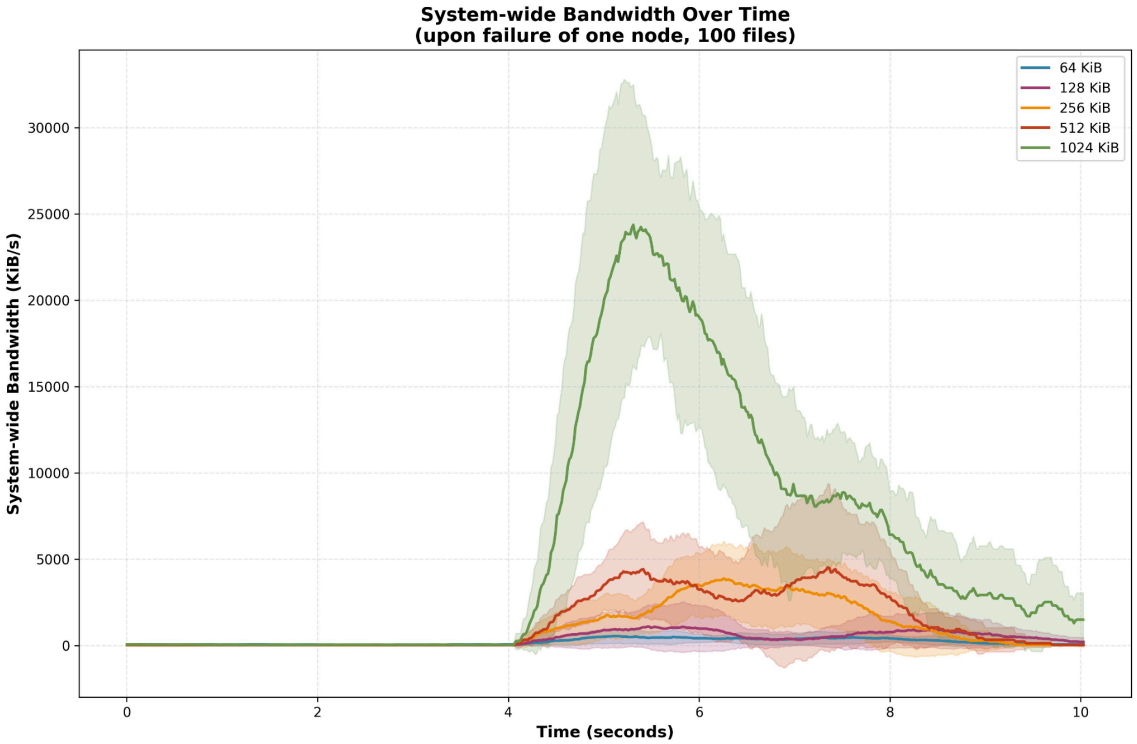
The receiving node compares the `Counter` version. If its local version is older, it accepts and updates. This ensures that data is re-replicated after failures and all replicas eventually converge to the same state (highest counter wins). Merge can also be invoked explicitly by clients for specific files.

a) Past MP Use:
The system uses a Gossip-based protocol (from MP2) to maintain a cluster membership list. MP1 is useful for us to grep patterns across nodes simultaneously, making it easy to identify distributed system issues, such as checking append concurrency, without manually checking logs on each machine.

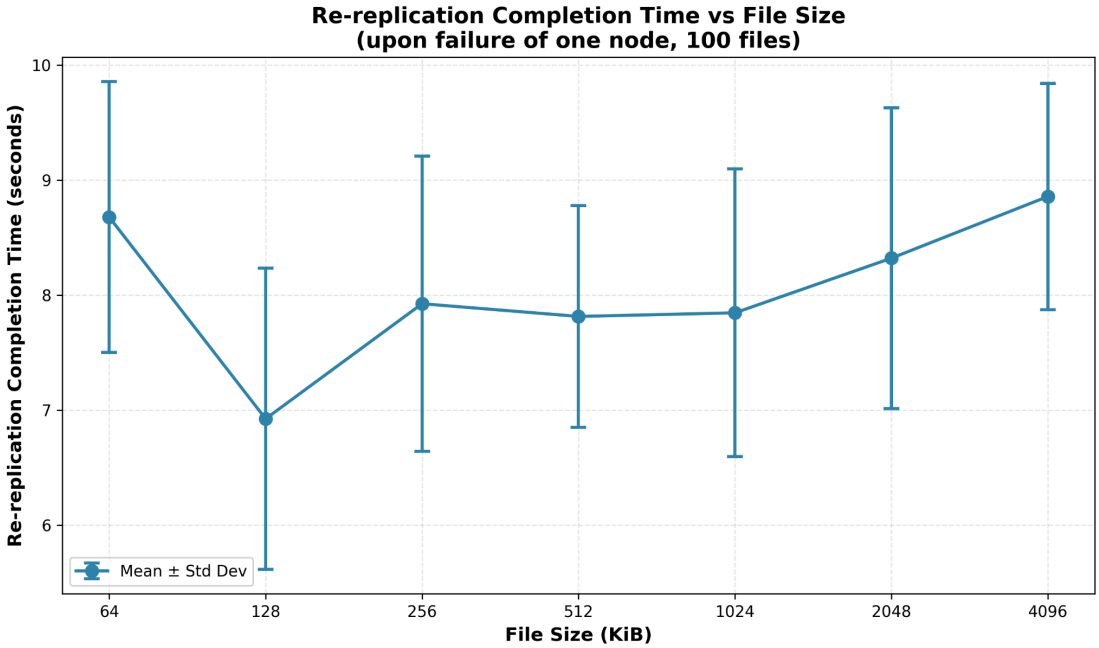
b) LLM use: In this MP, the LLM is used to help us debug the problem we have and help us with GoLang. It also helps us greatly in writing shell scripts for running the experiments.

c) Measurements:

i) (Overheads)

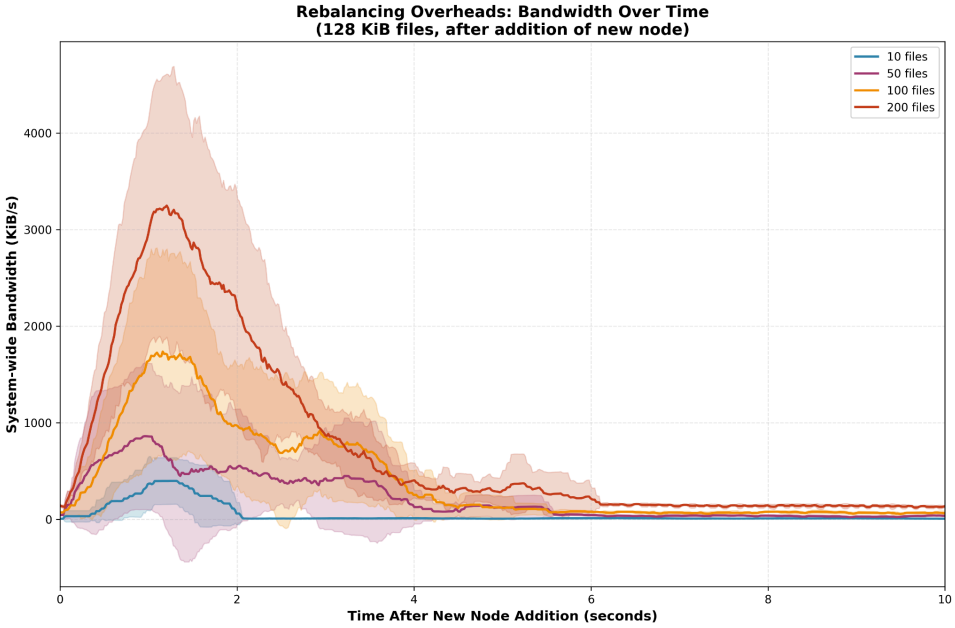


Bandwidth increases over time during re-replication, with larger file sizes requiring higher peak bandwidth than smaller sizes. All sizes exhibit a spike during active re-replication, followed by a gradual return to baseline (~7–9 seconds after node failure).



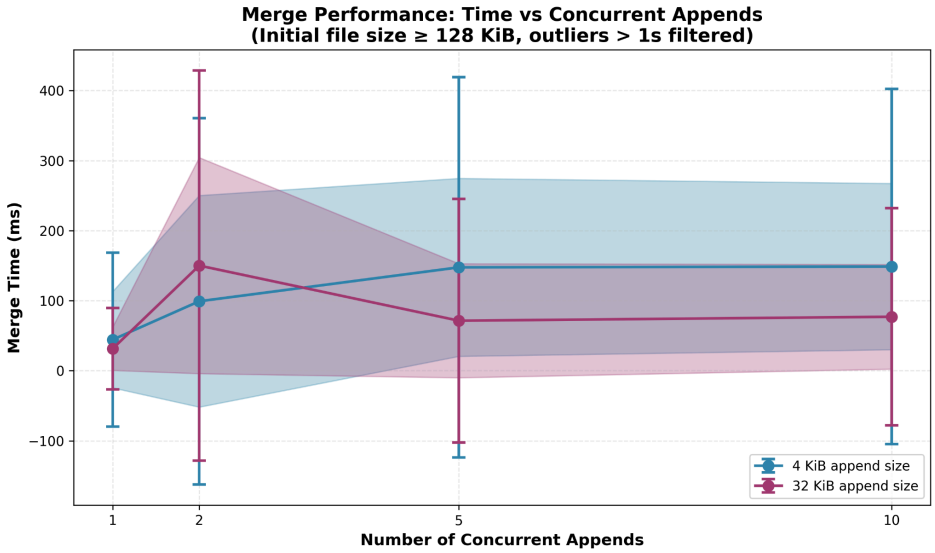
Re-replication completion time is relatively stable across file sizes, ranging from ~7 to 9 seconds, with a gradually increasing trend at the end. It suggests that replication time increases with file size, though it may not be a significant factor.

ii) (Rebalancing overheads)



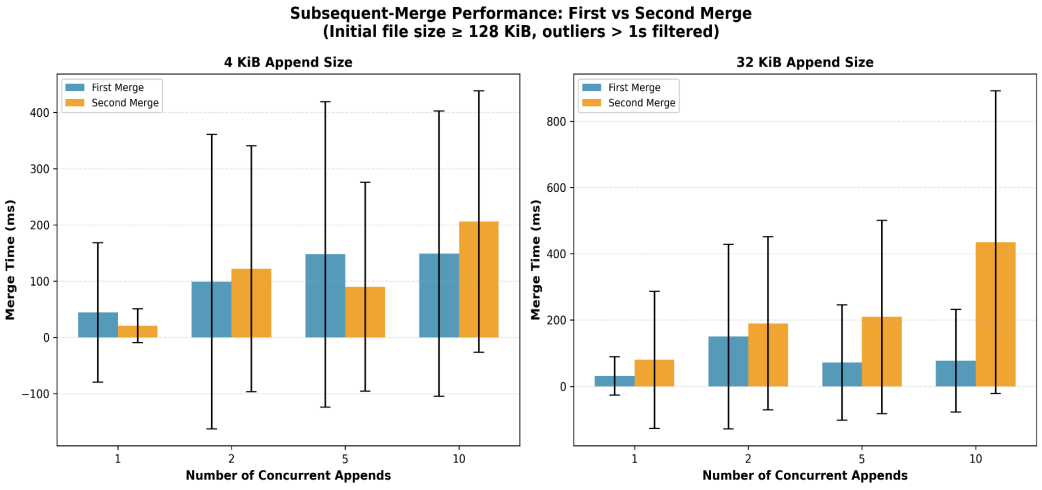
Bandwidth increases with the number of files: 200 files require significantly higher bandwidth (peaking around 3–4 MB/s) than 10 files (peaking around 400 KB/s), indicating rebalancing overhead scales with file count. Bandwidth also increases over time after node addition, suggesting the system progressively rebalances files across nodes during the 10-second window.

iii) (Merge performance)



Merge times are generally stable and very close across different numbers of concurrent appends. Multi-appends are slightly slower than single-appends, indicating the merge operation handles concurrency efficiently. Different append sizes demonstrate a similar trend, suggesting that it is not a significant factor affecting merge performance.

iv) (Subsequent-merge performance)



The graph compares first and second merge times across concurrent appends. When the number of concurrent appends increases, the second merges become slower than the first merges. This suggests that after an initial merge, subsequent merges face increased overhead, possibly due to accumulated metadata or fragmentation resulting from the first merge. This pattern is similar for different append sizes.