Reimagining GFS: Achieving Exactly-Once Appends

Mitansh Kayathwal*
IIIT Hyderabad

Vineeth Bhat*
IIIT Hyderabad

Abstract

The Google File System (GFS) is a scalable distributed file system designed to handle large amounts of data storage across multiple commodity machines and supports Non-POSIX write, read and atleast-once record-append mutations. This paper presents the design and implementation of the GFS, with the addition of *exactly-once Record-Append semantics* to address consistency requirements in distributed storage systems. Developed in Go, our implementation is localized out of necessity due to the unavailability of commodity server infrastructure, enabling focused benchmarking and performance evaluation.

1 Introduction

The Google File System (GFS) was introduced to meet the demands of data-intensive applications that require scalable, reliable, and fault-tolerant storage solutions (Ghemawat et al., 2003). Designed for deployment on commodity hardware, GFS achieves high availability and fault tolerance despite frequent hardware failures. The system is optimized for workloads involving large files, sequential reads, and writes, making it particularly suited for applications like data analytics, web crawling, and search indexing.

However, the trade-offs inherent in its at-least-once record-append semantics can lead to duplicate records, complicating consistency in certain use cases. These limitations are mitigated within the Google infrastructure, where client applications are designed to tolerate and handle such inconsistencies. While this approach works effectively in tightly integrated environments, it *could* pose challenges for external systems or use cases that require stronger guarantees of data integrity and precision.

In this paper, we build upon the original GFS by incorporating exactly-once record-append semantics, enabling precise mutation guarantees while maintaining the scalability and robustness that the system is known for by implementing a localized version of the GFS[1] with 7000+ lines source code in the Go programming language.

Our system efficiently supports concurrent reads, writes, and exactly-once atomic record appends using a primary-secondary replication model to ensure high availability and fault tolerance. Failures trigger re-replication, dynamically redistributing chunks from failed nodes to operational ones, while stale replicas are systematically identified and resolved to maintain consistency. Master node failures are mitigated via checkpointing and operation logs, enabling recovery with minimal data loss, and offline chunk servers rejoin seamlessly by synchronizing with previously stored chunk data.

Storage efficiency is enhanced through automated garbage collection of obsolete or unused data. Configurability is achieved via YAML-defined parameters, enabling fine-grained control of system behavior, such as chunk size and replication factors. Comprehensive benchmarking evaluates throughput and latency across all operations, providing empirical validation of the system's scalability, fault tolerance, and consistency guarantees.

2 Design

Our implementation is identical to the GFS with minor differences that have been elaborated upon in the relevant sections.

2.1 Client APIs

Create Takes in the filename as the input and queries the master to create it within its metadata. At this point, no chunks have been allocated to this file. Potentially, we can add support for handling paths within our implementation.

Rename Renames the file given the old and new filenames. Results in the master updating it's mapping regarding the filenames.

* Equal Contribution

[1] **Will be released online post evaluation**

Delete Deletes the file given the filename. Results in the master adding the corresponding chunks to its garbage handling routine and making the filename available for use again.
Read Reads the data from a file given the filename, offset to read from and length of data to be returned. An error occurs for offsets exceeding the file's size, and the length of the data read is returned. This length may be less than the requested size due to insufficient remaining data in the file.
Write Writes the given content at the given offset and file. Internally handles chunk creation and assignment if the length of the content spans multiple chunks or exceeds the size of the last chunk assigned to the file.
Append Appends the given content to a provided file. If the length of the content is greater than one-fourth of the chunk size, results in an error to prevent internal fragmentation similar to the GFS. If the addition of the content will exceed the current chunk, asks the client to request a new chunk,
WriteFile Similar to write but accepts a local file as input, rather than the written content, which is read for the contents.
Chunks Returns the chunk handles given a filename and range of chunks indices.
Push Pushes data to a given chunk given the chunk handle and the content.

2.2 Communications

Communications occur between the clients and the master, the clients and the chunk servers and between chunk servers using `gRPC`. The remote procedure calls are defined in Table 1.

3 Implementation

We discuss key design choices that we took in our implementation. Other aspects of the system remain consistent with the original GFS design and are considered understood, requiring no further elaboration.

3.1 Master

Operation Log: The master server maintains an operation log, periodically persisting it to enable state restoration upon restart. This log serves as a critical component for reconstructing the in-memory metadata, ensuring the consistency and recovery of file and chunk states. It records a comprehensive history of operations, including file creation, updates, and deletions, as well as changes to chunk metadata such as additions, deletions, and version updates.
Chunk Server Monitoring: The master server actively monitors the health and operational status of chunk servers, manages chunk server leases, and ensures that chunk replication adheres to the required replication factor through several background processes that parse *heartbeats* from the connected chunk servers. It periodically verifies the health of chunk servers and marks them as "INACTIVE" if no heartbeat is received within a predefined timeout interval. Persistent failure to reestablish communication escalates "INACTIVE" servers to "DEAD" status, triggering their removal from the system.

To maintain the desired replication factor, the master identifies under-replicated chunks and initiates replication processes to restore redundancy. Stale replicas, identified through heartbeat metadata, are updated to ensure consistency across the system. The master also manages primary leases for chunks by removing expired leases, resetting primary designations, and updating lease expiration timestamps as necessary. These mechanisms collectively ensure data availability, consistency, and resilience against failures.
Garbage Collection and Other Cleanups: The garbage collection process periodically removes files marked for deletion, processing them in batches to ensure efficient cleanup without hindering active operations. Mechanisms are in place to prevent concurrent garbage collection cycles, preserving system stability.

Additionally, the system performs cleanup of expired or failed pending operations, removing those that exceed the permitted retry count or time limit. This includes the core logic for managing and cleaning up pending operations, alongside logging failed operations to facilitate auditing and debugging. These maintenance tasks collectively enhance the reliability and operational efficiency of the system.

3.2 Chunk Servers

Initialization Workflow: The initialization process establishes the server directory, connects to the master server, and initializes internal structures, including the chunk metadata map, leases map, and operation queue. Upon completion, the chunk server begins operation by reporting its chunk inventory to the master, initiating the heartbeat mech-

RPC	Sender	Receiver	Description
`ReportChunk`	Chunk Server	Master	Facilitates chunk servers to report all the chunks they host, along with metadata, to the master server during initialization.
`RequestLease`	Chunk Server	Master	Allows a chunk server to request a lease for a specific chunk handle, ensuring it can act as the primary for write operations within the lease period.
`HeartBeat`	Chunk Server	Master	Establishes a bi-directional streaming channel for chunk servers to periodically send health updates (e.g., available space, active operations) and receive commands from the master for replication, version updates, or chunk deletions.
`GetFileChunksInfo`	Client	Master	Retrieves metadata for specific file chunks within a given range, including chunk handles, locations, and versioning.
`GetLastChunkIndexInFile`	Client	Master	Fetches index of the last chunk in a specified file for appends.
`PushDataToPrimary`	Client	Chunk Server	Transfers data from the client to the primary chunk server for subsequent operations such as writes or appends, including checksums for integrity and secondary locations for propagation.
`PushData`	Chunk Server	Chunk Server	Streams chunk data from one chunk server to another, including metadata like offset, checksum, and operation ID for verification.
`ForwardWriteChunk`	Chunk Server	Chunk Server	Forwards a write request from the primary to a secondary server.
`ForwardAppendChunkPhaseOne`	Chunk Server	Chunk Server	Forwards the first phase (2PC) of an append operation to secondary chunk servers, preparing them for the append.
`ForwardAppendChunkPhaseTwo`	Chunk Server	Chunk Server	Executes the second phase (2PC) of an append operation.
`ForwardReplicateChunk`	Chunk Server	Chunk Server	Transfers a replica of a chunk from one server to another, including its data, checksum, and version, to maintain redundancy.

Table 1: Salient RPCs used in our implementation. Other RPC calls include `WriteChunk`, `ReadChunk` and `RecordAppendChunk` sent from a client to a chunk server, and `CreateFile`, `DeleteFile` and `RenameFile` sent from a client to the master.

anism, and activating both the lease requester and operation processor.

Operation Queue: Replication commands from the master server, along with write and append commands from the clients, are placed in the operation queue and processed serially to maintain consistency across operations. While the commands are handled in a sequential manner to preserve data integrity, asynchronous processes are spawned to execute the actual mutations, ensuring that the operations do not block the system and can proceed concurrently.

Metadata Management: Unlike the original GFS, where chunk servers do not manage their own metadata, our design incorporates local metadata storage within each chunk server to ensure state consistency and facilitate efficient operations. The metadata includes essential attributes for each chunk, such as its size, last modified time, checksum, version, and other relevant properties. The versioning mechanism within the metadata ensures that if a chunk server restarts and hosts an outdated version of a chunk, the master can issue an update command to synchronize it. This information is maintained in an in-memory map and periodically checkpointed to stable storage to preserve the chunk server's state across failures.

To ensure atomicity and prevent data corruption during writes, the metadata is serialized into a JSON file. The process involves initially writing to a temporary file, followed by renaming the file to its final destination, ensuring that partial writes do not compromise the integrity of the metadata. Upon initialization, the chunk server attempts to recover metadata from the JSON file. If the file is absent, an empty metadata map is created. Furthermore, the recovery process verifies the existence of chunk files, removing metadata entries for any missing chunks, ensuring that the stored metadata accurately reflects the current state of the chunk server.

Data for Mutations: Data mutations, such as writes and appends, involve distributing data across all relevant chunk servers. The process starts with the client sending a request to the primary chunk server, which validates the request, stores the data, and subsequently forwards it to the secondary chunk servers for replication (Figure 1). Every content pushed to the chunk server is marked with a unique operation ID which is used to retrieve the pending data in the chunk server in the event of a mutation operation.

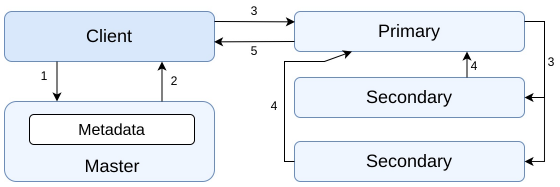

Figure 1: *Data Flow for Mutations*. Data flow from the client to chunk servers as a precursor to sending mutation requests. The client sends the content to the primary which returns an acknowledgment after it has sent the data to the secondaries.

Unlike the original GFS, where the client sends data to a chunk server based on proximity, determined by IP routing and managed by the master server, our system operates in a more localized manner. In our design, there is no predefined "closest" chunk server, and instead, the primary chunk server is chosen to handle the data mutation and coordinate the replication to secondary servers. This distinction simplifies the mutation process by centralizing the responsibility within the primary chunk server.

Reads and Writes: Read operations start by determining the chunks based on offset and length. The client retrieves chunk metadata, selects the appropriate server (uses secondary if available), and sends a read request. The chunk server processes the request, returning the data or an error if the offset exceeds the file size.

Write operations involve writing data to a specified offset within a chunk. The primary chunk server processes the write, updating the chunk file and metadata (size, modified time, checksum), and then propagates the change to secondary servers. If the offset exceeds the file size, the file is padded with null bytes.

Exactly-Once Appends: Appends in our system leverage a modified version of the 2-Phase Commit (2PC) protocol (Bernstein et al., 1987) to guarantee exactly-once semantics (Figure 2). A lock is obtained in the primary server and is held until the end of the second phase to ensure atomicity guarantees. The process begins with the primary server retrieving the pending data and initiating the first phase of the protocol. During this phase, null bytes are written to the chunk file at the append offset, which is chosen as the current file length, on the primary server's local storage. The primary then propagates this command to the secondary servers. This initial phase verifies that both the primary and

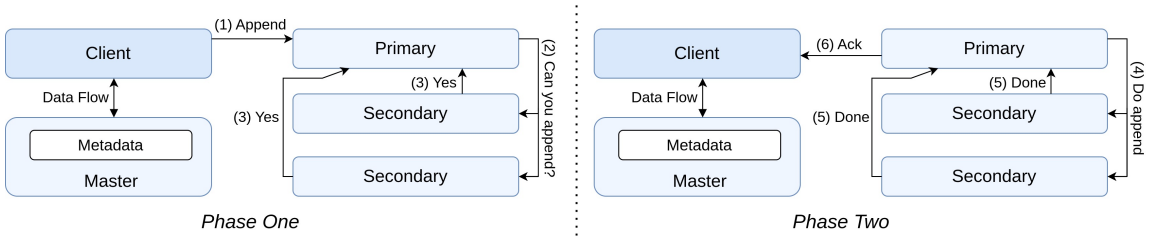

Figure 2: *Modified Two-Phase Commit Protocol for Exactly-Once Record-Append Semantics*. In phase one, the primary queries secondaries to confirm append readiness at a chosen offset. Upon success, phase two appends the data to secondaries and then the primary. Versioning enables recovery without persistent commit contexts, and unique idempotency IDs ensure idempotency. In case of secondary failure, the primary nullifies data on active secondaries to maintain consistency.

secondary servers can perform the write operation successfully.

Upon successful completion of the first phase, the second phase begins. Here, the actual data is written to the chunk file. The primary server first propagates the append operation with the actual data to the secondary servers, ensuring they complete the operation. Finally, the primary appends the data to its own storage, ensuring that all writes are consistent across the replicas.

In the event of a failure during the second phase at any secondary server, the primary server initiates a nullification process. This process removes the partially appended data at other secondaries to maintain consistency. The append operation is considered complete when the primary server successfully writes the data and receives acknowledgments from all secondary servers. Following this, the primary updates the chunk metadata and sends the operation result back to the client. This ensures the append operation is executed exactly once, even in the presence of system failures.

Unlike the traditional 2PC protocol, our system does not require the persistent storage of contexts after a server responds affirmatively during the "prepare" phase. This is achieved through the use of chunk versioning. Versioning allows the chunk server, upon recovery from a failure, to complete the commit operation since the master sends it updated commands to restore the system's state. This eliminates the need for persistent logs for incomplete transactions, reducing the overhead on storage and improving recovery efficiency.

To achieve idempotency, we employ idempotency IDs, a method inspired by systems like Stripe (Leach, 2017). Each append operation is assigned a globally unique identifier that is included in all communications between the client, primary, and secondary servers. If an operation with the same ID is received again, the server recognizes it as a duplicate and avoids re-execution. This mechanism ensures that even in cases of retries, due to network failures or other issues, the same operation is not applied multiple times. By integrating this approach with versioning and 2PC, our system guarantees robust, exactly-once append semantics while maintaining high performance and reliability.

3.3 Other Considerations

Go was chosen for its simplicity, concurrency support, and efficient memory management, making it well-suited for building scalable, high-performance distributed systems. Protocol Buffers (`Protobuf`) were utilized for their compact, platform-neutral serialization format, enabling efficient communication and interoperability across system components.

4 Analysis

The following analysis has been performed on an `11th Gen Intel(R) Core(TM) i7-1165G7` with 8 CPUs and 2 threads per CPU.

4.1 Reads and Writes

The expriment setup involves having multiple clients, 6 chunk-servers (to emulate a better load distribution) and a single master server.

Reads: Before spawning the read requests, we first write a lot of data to multiple files as contiguous data blocks. We then make concurrent read requests for chunks to the files through the different clients which we had spawned earlier. The experiment was conducted with a complete read workload and the key idea was to test how the system is performing with reading data from the stable storage of the chunks to finally the client receiving

the requested data. We increased the number of clients from 1 to 10, and observed that we are able to notice an almost linear increase in throughput which is expected with a decent latency.

Writes: We make concurrent write requests for chunks to the files through the different clients which we have spawned. The experiment was conducted with a completely random write workload and the key idea was to test how the system is performing with reading data from the stable storage of the chunks to finally the client receiving the requested data. We increased the number of clients from 1 to 10, and observed that we are able to get similar linear increase in throughput which as with the reads scenario. However, the overall value of the write throughput is larger than that of reads, again which is expected due to writes not having to read data from the disk which takes more IO operations and time as well.

4.2 Exactly-Once Appends

We conduct an experiment involving multiple clients appending data to a single shared file - each client performed 100 operations, appending 1 kilobyte of data. By increasing the number of concurrent clients from 1 to 10, we analyzed both the system's throughput and its ability to maintain atomicity under load (Figure 5).

Despite the apparent concurrency, the GFS architecture enforces strict sequential processing of append operations. This design ensures that appends occur in a strictly ordered manner ensuring that each append is executed precisely once in a deterministic order.

Throughput Scaling. Throughput exhibited near-linear scaling as the number of clients increased. This indicates that the system can efficiently handle growing concurrent access by multiple clients.

Latency Overhead. With more clients, latency per append operation incrementally increased suggesting coordination overhead in maintaining the strict sequential processing of requests.

4.3 Other

The system's design explicitly assumes a non-Byzantine failure model, wherein components fail by crashing rather than exhibiting arbitrary or malicious behavior. Handling Byzantine failures, such as tampered or rogue behavior by nodes, would require additional mechanisms like Byzantine Fault Tolerance (BFT), which are not considered in this implementation.

In the event of master failure, the system relies on recovery mechanisms using the operation log. Upon restarting, the master replays the log to rebuild its state. However, consistency is only guaranteed up to the last stable checkpoint. Any operations recorded after the checkpoint but before the failure may lead to inconsistencies.

5 Conclusion and Future Scope

The design and implementation of this distributed file system exemplify its robustness and scalability, providing a dependable framework for managing shared file resources in distributed environments. Core functionalities, such as atomic and sequentially ordered appends, uphold data integrity by guaranteeing exactly-once semantics and mitigating race conditions, even under concurrent client access. Although the system demonstrates effective scaling with an increasing client base, the observed latency overhead underscores the inherent trade-offs involved in synchronizing access to shared resources.

Beyond append operations, the system architecture addresses essential challenges in distributed file management. The chunk-based storage model, augmented by replication across multiple servers, significantly enhances fault tolerance and ensures high availability. Versioning mechanisms further reinforce system consistency by enabling recovery during partial failures, obviating the need for persistent storage of 2PC contexts. Additionally, the integration of idempotency IDs bolsters operational reliability, allowing retryable append operations without compromising data integrity, thereby ensuring uninterrupted functionality in the presence of network disruptions or server failures.

Future Scope

Inclusion of Snapshots. Future work could explore incorporating snapshot capabilities into the system, allowing users to capture consistent views of the data at a particular point in time. This would enhance the system's ability to recover from failures.

Inclusion of Directory Management. The current design focuses primarily on chunk and file storage. Integrating a robust directory management system will allow for improved file indexing, metadata handling, and more efficient access to data. This would also simplify tasks like file renaming, moving, and deletion within the distributed file system.

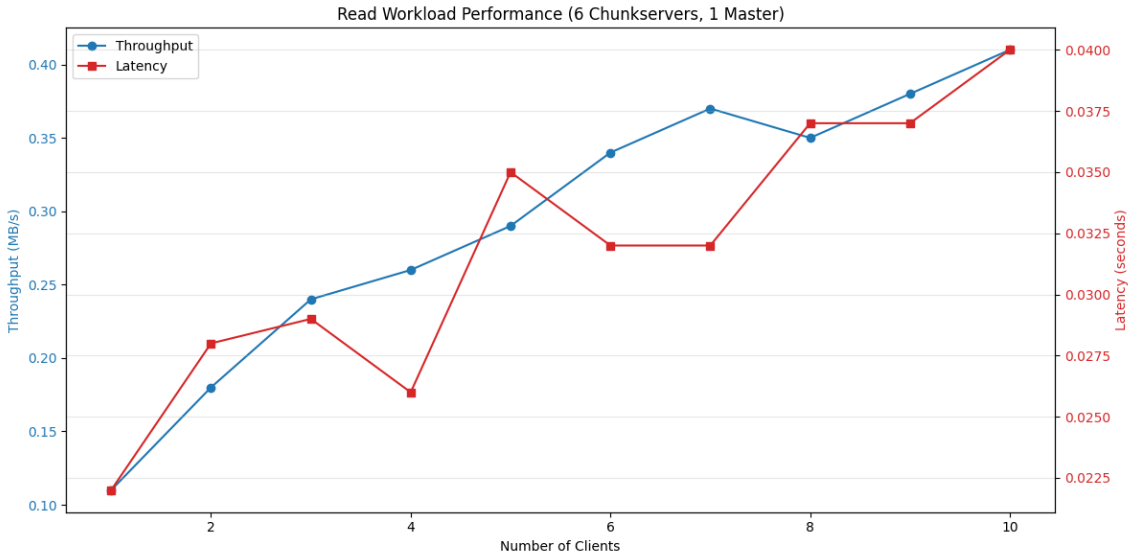

Figure 3: *Analysis of Reads*. Performance of 10KB read operations with varying client load.

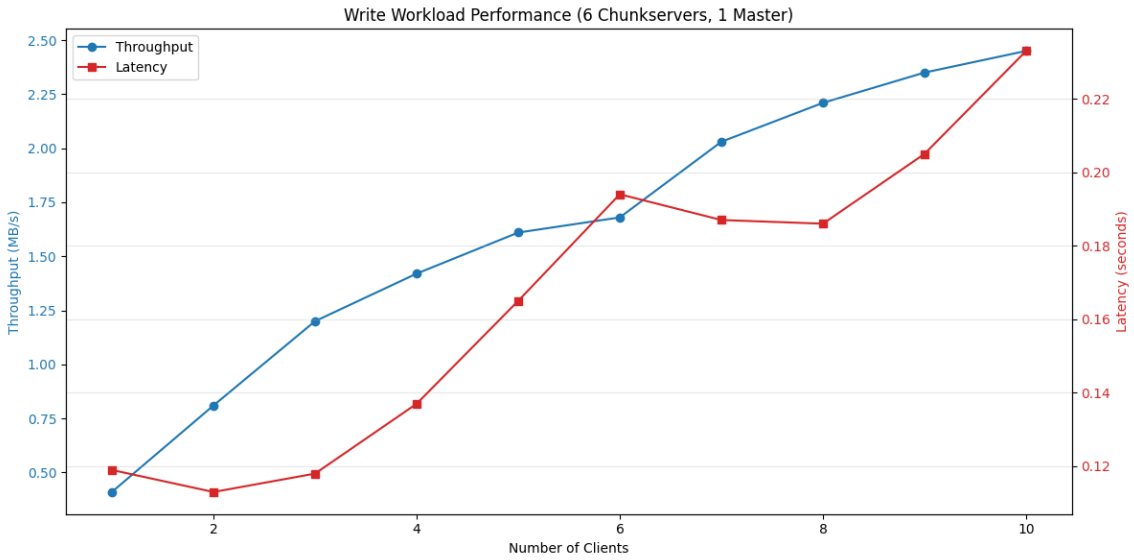

Figure 4: *Analysis of Writes*. Performance of 10KB random write operations with varying client load.

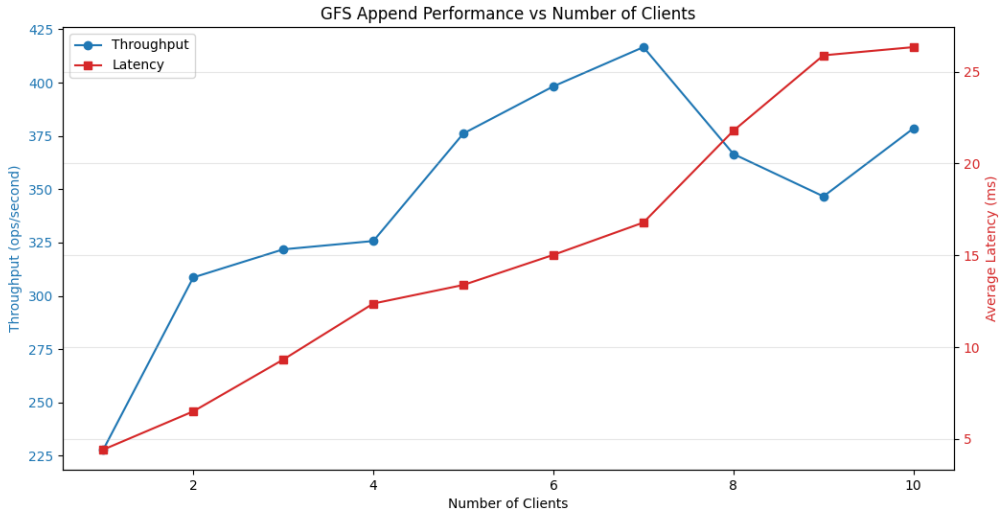

Figure 5: *Analysis of Append Mutations*. Performance of 1KB append operations with varying client concurrency to a shared file resource.

Comparison with original append semantics. A detailed comparison of the append operation's exact-once semantics with the original GFS append behavior is needed. This will provide insight into the trade-offs between reliability, performance, and complexity, and help highlight the advantages of our approach in maintaining consistency across replicas.

Analysis in non-localized environments. The current design assumes localized chunk server selection. Expanding this to non-localized environments, where chunk servers may be geographically distributed, could introduce new challenges in latency, consistency, and fault tolerance. Future work will analyze how to adapt the system to such environments while maintaining high performance and consistency.

Handling thundering herds. The "thundering herd" problem arises when a large number of clients simultaneously attempt to access or modify a shared resource, leading to performance bottlenecks or system overload. Addressing this challenge in the system could involve implementing intelligent client request scheduling and load distribution mechanisms.

Addendum

Acknowledgments

The authors are grateful for the opportunity to work on this project as part of a Distributed Systems course taken by Dr. Kishore Kothapalli[2].

Large Language Model Usage

In our implementation, we utilized `Claude-3.5-Sonnet` for aiding us in development and `GPT-4o-mini` for refining this paper, including grammatical corrections and rephrasing.

References

Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. 1987. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley Reading.

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43.

Brandur Leach. 2017. Designing robust and predictable apis with idempotency. Accessed: 2024-11-20.

[2]Dean (A.), IIITH. Webpage.