# SHFS: A modern gRPC Distributed File System

Siddharth Narsipur
University of Rochester
snarsipu@u.rochester.edu

Henry Liu
University of Rochester
hliu68@u.rochester.edu

## Abstract

*We have designed and implemented SHFS, a scalable distributed system for file storage and transfer. Built on top of gRPC, a high-perfomance communications framework, our system provides system-wide fault tolerance, and is optimized to conduct parallel operations when possible.*

## 1. Introduction

Modern computing increasingly relies on scalable, fault-tolerant systems to manage and transfer large volumes of data efficiently. Distributed file systems play a critical role in enabling these capabilities, especially in environments where performance, reliability, and scalability are necessary.

SHFS leverages gRPC, a high-performance, open-source universal RPC framework, as its communication framework. This enables efficient, language-agnostic interaction between system components and supports robust client-server communication with built-in support for streaming and parallelism. The system architecture emphasizes fault tolerance, allowing it to continue operating despite the failure of individual nodes, and parallel operations, which significantly improve throughput and reduce latency during large-scale file transfers.

In this report, we describe the design and implementation of SHFS, along with an evaluation of its performance and reliability. We also discuss the design choices that shaped the system and how they contribute to its scalability and robustness.

## 2. Related Work

The design of SHFS draws inspiration from established distributed file systems, notably the Google File System (GFS) and educational projects like Carnegie Mellon University's 14-736 course assignments.

GFS, introduced by Google in 2003, was engineered to manage large-scale data processing across numerous machines using commodity hardware. Its architecture emphasizes fault tolerance, high throughput, and scalability.

In an academic setting, CMU's 14-736 course project tasked students with implementing a simplified distributed file system comprising a naming server and multiple storage servers. We used this project for inspiration and to structure our ideas.

## 3. Architecture

Our file system architecture is comprised of three independent but correlated programs — the naming server, the storage server, and the client interface.

### 3.1. Naming Server

The Naming Server is the main point of contact between clients and the SHFS system. It maintains a global view of the file system and coordinates interactions with the distributed storage servers. Clients rely on it for operations such as file uploads, lookups, deletions, and for discovering which storage servers hold specific files.

Internally, the Naming Server manages metadata using a NamingDataManager, which stores active storage servers, file-to-server mappings, heartbeat timestamps, and replication tasks. These are wrapped in thread-safe containers to support concurrent access and ensure data consistency across multiple threads.

When a storage server starts, it registers with the Naming Server and begins sending periodic heartbeat messages. The server is added to the active server set, and its availability is tracked over time. Clients uploading a file receive a list of target servers selected from the active set, and once the file is uploaded, the Naming Server records its location.

If a server becomes unresponsive through missed heartbeats, the server is then removed from the active list, and any files it held are flagged for re-replication. Files are then assigned to other servers to restore the intended replication factor, preserving system fault tolerance.

### 3.2. Storage Server

The storage server is responsible for storing, retrieving, and managing file data on disk. Each server runs independently and registers itself with the Naming Server upon startup.

Once registered, it periodically sends heartbeat signals to report its availability and receive any tasks assigned to it by the Naming Server.

Storage servers handle file uploads and downloads directly with clients. During an upload, the server receives data in chunks via a constant gRPC stream, writes it to its local file system, and updates its internal metadata. Similarly, for downloads, the server reads the requested file from disk and streams the contents back to the client. Files are stored in a local directory, and file metadata is tracked using a thread-safe data manager.

The storage servers also support file removal and file sharing between storage servers. If a server is assigned a replication task—typically after another server fails—it downloads the specified file from another storage server to help maintain the desired replication factor.

### 3.3. Client Interface

The client interface provides users with a command-line tool to interact with the SHFS system. It communicates with the Naming Server to perform metadata operations and directly with storage servers to handle file content. Users can upload, download, list, or remove files, as well as inspect the distribution of files across storage nodes.

When uploading a file, the client first sends a request to the Naming Server to receive a list of available storage servers. The naming server returns a list of storage servers that maintain consistent load across storage servers and the required replication factor. It then transfers the file in parallel to each server using gRPC streaming. For downloads, the client queries the Naming Server to locate the file, selects one of the available servers based on (simulated) latency, and retrieves the file content.

Additional commands include listing all files, removing a file, and viewing a mapping of files to servers (or servers to files). The interface abstracts away the distributed nature of the system, offering a simple and familiar experience while ensuring reliability and performance under the hood.

## 4. Distributed System Principles in SHFS

SHFS was designed from the ground up with the core principles of distributed systems in mind—scalability, fault tolerance through replication, and safety via redundancy. This section outlines how SHFS achieves these goals through its architectural and operational design.

### 4.1. Scalability through Concurrency and Decentralization

Scalability is one of the most important criteria in evaluating a distributed system. SHFS is engineered to perform well even as the number of clients and storage nodes increases significantly.

First, all internal data structures in the Naming Server are protected by fine-grained thread-level locking. Each data structure is encapsulated within a custom thread-safe class, each maintaining its own mutex. This ensures that operations on one structure (e.g., heartbeat tracking) do not block operations on others (e.g., file-to-server mapping), enabling high levels of concurrency and responsiveness even under heavy load.

Second, all file transfers occur directly between clients and storage servers. The Naming Server only facilitates metadata coordination and is never in the data path. This design prevents the Naming Server from becoming a bottleneck during large-scale file transfers and ensures that network bandwidth and compute resources are distributed across the system.

Third, SHFS employs a heartbeat mechanism based on a push model. Storage servers independently and periodically send heartbeat messages to the Naming Server. This design scales well because the Naming Server remains passive, not needing to poll or actively ping thousands of storage nodes. Instead, servers that go silent for a defined timeout are considered down, which significantly reduces network traffic and CPU load on the coordinator as the cluster scales.

### 4.2. Fault Tolerance through Replication

SHFS supports configurable file replication to ensure high availability and durability. The **replication factor** specifies how many distinct storage nodes a file should be stored on. This mechanism provides resilience against node failures—so long as at least one replica remains available, the file can be retrieved.

If the Naming Server detects that a storage node has failed (due to missed heartbeats), it marks the server as offline and re-evaluates the replication state of all files previously hosted on that server. Any file whose replication factor has dropped below the configured threshold is automatically flagged for re-replication. The Naming Server then coordinates new replication tasks by assigning those files to active storage nodes. This process is transparent to clients and ensures continuous fault recovery without any manual intervention.

### 4.3. Dynamic Rebalancing for Resource Efficiency

To maintain system balance and avoid storage hotspots, SHFS implements dynamic file rebalancing. When a new storage node joins the system—or a previously failed node recovers and rejoins—the Naming Server initiates a rebalancing process. This process identifies overburdened storage servers and migrates a subset of their files to the newly available node. The rebalancing decisions consider current load distribution and aim to equalize storage utilization across the cluster. This dynamic redistribution avoids overreliance on specific nodes, reduces read/write contention,

and makes full use of available resources.

## 5. Implementation

SHFS is implemented in modern C++20, utilizing the latest syntax and features to ensure clean and efficient code. We exclusively use `std::unique_ptr` and `std::shared_ptr` for memory management, which helped eliminate memory-related bugs during development. The system architecture follows the dependency injection pattern, separating concerns such as gRPC service implementations, internal data structure management, and the heartbeat monitor into distinct classes for better modularity and maintainability.

To optimize the development workflow, we leveraged advanced CMake techniques, including precompiled headers (PCH), which reduced build times from 30 seconds to just 1 second, and automatic generation of proto files for gRPC. Additionally, we employed `spdlog` and external Python script for testing, which greatly streamlined the testing process and enhanced the overall development experience.

## 6. Limitations and Future Work

While SHFS demonstrates the core principles of a fault-tolerant and scalable distributed file system, several limitations remain that present opportunities for future improvement. These limitations span from architectural trade-offs to missing robustness features.

### 6.1. Concurrency Model and Performance Trade-offs

To support concurrent operations, our system uses fine-grained locking, with each internal data structure protected by its own mutex through a custom thread-safe wrapper. While this theoretically minimizes contention by allowing independent access to unrelated data structures, the actual performance benefit remains uncertain. Due to limited testing, we have not verified whether this design yields significant improvements in throughput or latency.

In practice, this approach may introduce overhead due to frequent locking and unlocking of multiple resources within a single operation. Such overhead could negate the benefits of fine-grained concurrency and even result in slower performance compared to a coarser-grained locking strategy that acquires a single global lock for critical sections. As an alternative, future iterations of SHFS could investigate replacing our thread-safe dictionaries with concurrent data structures from libraries like oneTBB.

### 6.2. Missing Fault and Data Integrity Handling

Currently, SHFS optimistically assumes that file transfers between clients and storage servers will always succeed. This assumption is fragile in real-world networked systems, where transient failures, dropped connections, or incomplete transfers are common. The system lacks retry logic on the client side, which would ensure that failed uploads or downloads are retried with exponential backoff or redirection to alternate servers.

Additionally, SHFS does not address data corruption or bit rot, both of which are serious concerns for long-term storage systems. Future versions could include checksums or cryptographic hashes to verify file integrity during uploads and downloads. Coupled with regular background integrity scans, this would allow early detection and correction of corrupted files via replication.

Another missing piece is metadata durability. The Naming Server currently stores all file system metadata in memory, meaning a crash or restart would result in total loss of the system's state. In contrast, GFS periodically persists its metadata to disk and logs all updates to an operation log, allowing it to recover its state upon restart. Implementing similar snapshot and logging mechanisms would make SHFS more resilient to crashes and power failures.

## References


[1] Carnegie Mellon University. *Project 3: Distributed File Systems*. 14-736: Distributed Systems, Spring 2020. Retrieved from https://www.andrew.cmu.edu/course/14-736-s20/applications/labs/proj3/proj3.pdf

[2] Ghemawat, S., Gobioff, H., Leung, S.-T. *The Google File System*. SOSP'03: Proceedings of the 19th ACM Symposium on Operating Systems Principles, 2003. Retrieved from https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

[3] Mixu, P. *Distributed Systems: Concepts and Design*. Retrieved from https://book.mixu.net/distsys/single-page.html