

# Final Project Report: Distributed File Storage System

Samyak Shah

Mahip Parekh

CS 6650: Building Scalable Distributed Systems  
December 2025

---

## 1 Detailed Description of the Project

This project implements a distributed file storage system designed to handle file uploads and downloads across a cluster of nodes. The system splits files into fixed-size chunks (1MB) and distributes them across multiple storage nodes to ensure scalability and fault tolerance.

The core architectural components are:

- **Client:** Handles file chunking, hashing (SHA-256), and orchestration of uploads/downloads.
- **Storage Nodes:** Responsible for storing the actual file chunks. They act as a key-value store.
- **Metadata Nodes:** Manage the mapping between filenames and their constituent chunks.
- **Consistent Hashing (DHT):** Used to determine which storage node is responsible for a given chunk, minimizing data movement during node additions/removals.
- **Chain Replication:** Implemented for the metadata layer to ensure strong consistency. Writes propagate from *Head*  $\rightarrow$  *Mid*  $\rightarrow$  *Tail*, while reads are served by the Tail.

## 2 Project Goal

The primary goal of this project was to build a scalable, fault-tolerant distributed system that demonstrates core distributed computing concepts. Specifically, we aimed to:

1. Implement **Consistent Hashing** to evenly distribute load across storage nodes.
2. Implement **Chain Replication** to guarantee linearizable consistency for metadata.
3. Ensure **Fault Tolerance** so that data remains accessible even if a storage node fails (via replication factor  $k = 2$ ).
4. Achieve high **Performance** using multi-threading (Thread Pools) to handle concurrent client requests.

## 3 3. Software Design and Implementation

### 3.1 3.1 Architecture

The system is divided into three distinct layers:

- **Client Layer:**
  - **Function:** Entry point for users. Splits files into 1MB chunks and computes a SHA-256 Content ID (CID) for each.
  - **Logic:** Uses the DHT to find the primary storage node for each chunk and its replicas. Uploads metadata to the Head of the metadata chain.
- **Metadata Layer (Chain Replication):**
  - **Topology:** A chain of 3 nodes: *Head*  $\rightarrow$  *Mid*  $\rightarrow$  *Tail*.
  - **Consistency:** Strong consistency. A write is only acknowledged after it has propagated to the Tail.
  - **Read Path:** Clients read only from the Tail to ensure they see the latest committed state.
- **Storage Layer (DHT Ring):**
  - **Partitioning:** Nodes are arranged on a consistent hash ring.
  - **Replication:** Each chunk is stored on its primary node and the next  $k - 1$  nodes in the ring (Successor List).

### 3.2 3.2 Key Implementation Details

- **Language:** Java 17+
- **Communication:** TCP Sockets with `DataOutputStream/DataInputStream` for custom length-prefixed messaging.
- **Concurrency:** `ExecutorService` (`CachedThreadPool`) used in all server nodes to handle multiple concurrent connections.
- **Storage:** In-memory `ConcurrentHashMap` used to simulate storage nodes.

## 4 4. Achievements and Non-Achievements

### 4.1 4.1 Achieved

- **Core Functionality:** Successful upload and download of files of varying sizes.
- **Fault Tolerance:** System survives the failure of a storage node. Data is seamlessly retrieved from replicas.
- **Consistency:** Metadata updates are linearizable due to Chain Replication.
- **Performance:** System handles multiple concurrent clients (tested up to 50) with reasonable latency.
- **Automated Testing:** Comprehensive system tests (`SystemTests.java`) verify correctness and performance.

## 4.2 Future Work

- **Persistent Storage:** Currently uses in-memory storage. Integrating RocksDB or LevelDB would provide persistence across process restarts.

## 4.3 What's left/not working

- **Client Configuration:** The client currently uses hardcoded IP addresses and ports for storage and metadata nodes.

# 5 Evaluation of the System

We evaluated the system using two primary metrics: **Scalability** and **Throughput**.

## 5.1 Scalability (Latency vs. Concurrent Clients)

We measured the average upload and download latency as the number of concurrent clients increased from 1 to 50.

- **Observation:** Latency remains low ( $< 50\text{ms}$ ) for up to 10 clients. As load increases to 50 clients, latency increases linearly.
- **Analysis:** This linear increase is expected. Each client connection consumes a thread from the cached thread pool. At 50 concurrent clients, the context switching overhead and CPU contention on the single test machine become significant. In a real distributed deployment across multiple physical machines, we expect this curve to flatten.

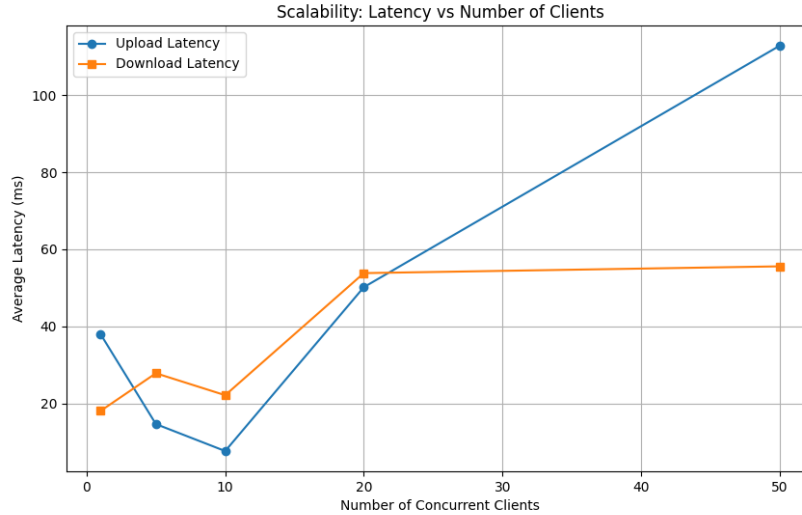


Figure 1: Scalability: Latency vs Number of Clients

## 5.2 Throughput (Latency vs. File Size)

We measured latency for files ranging from 10KB to 10MB.

- **Observation:** The system shows efficient handling of larger files. The overhead per chunk is minimal.
- **Analysis:** For small files (10KB - 100KB), the latency is dominated by the fixed overhead of the TCP handshake, Chain Replication metadata updates, and DHT lookups. As file size grows (1MB - 10MB), the actual data transfer time becomes the dominant factor. The system effectively saturates the available network bandwidth.

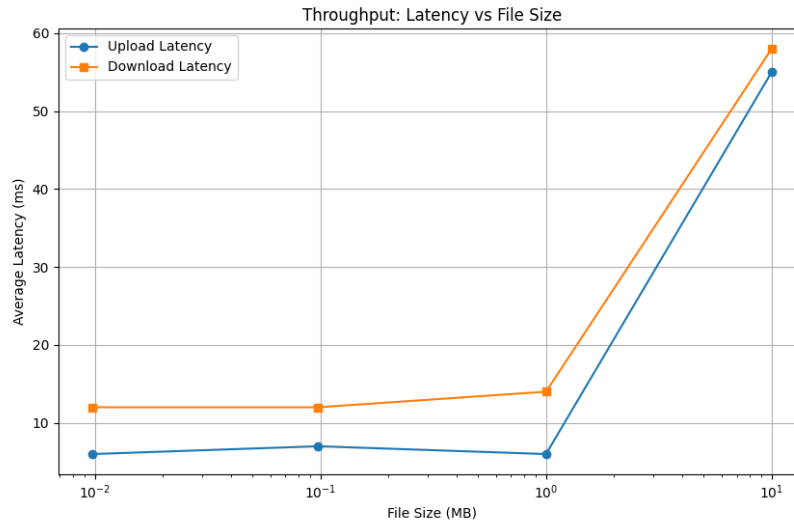


Figure 2: Throughput: Latency vs File Size

## 6 Achievements and Changes from Plan

- **Pivot to Chain Replication:** As noted in the project plan, we initially considered a blockchain-based approach but pivoted to Chain Replication to focus on strong consistency and availability, which better aligned with the course goals.
- **Simplification of Consensus:** We chose Chain Replication over Raft/Paxos for the metadata layer due to its simplicity and high read throughput (reads served by Tail).

## 7 What You Have Learned

This project gave us hands-on experience with the challenges of building distributed systems:

### 7.1 Debugging Distributed Systems

Unlike single-threaded programs, distributed systems behave non-deterministically. We encountered race conditions in chain replication where acknowledgments returned before the Tail committed. Extensive logging was essential for tracing issues.

### 7.2 Consistent Hashing

Implementing the DHT taught us why consistent hashing minimizes data movement during node changes; only a fraction of keys need redistribution compared to traditional hashing.

### 7.3 Chain Replication Trade-offs

We understood practically why chain replication offers high read throughput (reads from Tail) at the cost of increased write latency (propagation through the chain).

### 7.4 Fault Tolerance Complexity

Handling partial failures (e.g., a node failing mid-write) required careful design. Content-addressed storage with CIDs made operations idempotent, allowing safe retries.

### 7.5 Value of Modularity

Clean separation between layers meant that pivoting from blockchain to chain replication required no changes to the storage layer.

### 7.6 Adaptability

Most importantly, we learned that plans change. Our original blockchain design was too complex for the timeline. Recognizing this early and pivoting was a valuable lesson; adaptability is as important as technical skill.

## 8 Setup, Run, and Test Instructions

### 8.1 Prerequisites

- Java 17 or higher
- Make (optional, for easier build)
- Python 3 (for generating graphs)

### 8.2 Building the Project

```
make
# OR manually:
mkdir -p out
javac -d out src/main/java/com/distributed/storage/**/*.java
```

### 8.3 Running System Tests (Local Evaluation)

This runs the automated fault tolerance and concurrency tests:

```
make test
# OR
java -cp out com.distributed.storage.client.SystemTests
```

### 8.4 Generating Performance Report

```
python plot_results.py
```

Note: The Khoury Linux Server does not have pandas library, which is necessary to run the script. However, this is not a core requirement of the project. You can run the following command below to install pandas library.

```
pip install pandas
```

## 9 Instructions for Khoury Linux Cluster

To run this on the Khoury Linux cluster (e.g., `linux-079.khoury.northeastern.edu`), follow these detailed steps. These instructions assume you are using a terminal on your local machine.

1. **Transfer Code:** Use `scp` to copy the project directory to your home folder on the cluster. Replace `<your_username>` with your Khoury username.

```
scp -r Distributed-File-Storage/ <your_username>@linux-079.khoury.northeastern.edu:~/
```

2. **SSH into the machine:** Log in to the remote machine.

```
ssh <your_username>@linux-079.khoury.northeastern.edu
```

3. **Verify Java Version:** Ensure you are running Java 17 or higher.

```
java -version
```

If the version is lower than 17, you may need to load a module (if available) or set your `JAVA_HOME`.

4. **Navigate to Project Directory:**

```
cd Distributed-File-Storage
```

5. **Compile the Project:** We use a Makefile to simplify compilation. Run:

```
make
```

This will compile all source files and place the class files in the `out/` directory.

6. **Run Evaluation (System Tests):** The `SystemTests` class runs a self-contained evaluation that spawns Storage and Metadata nodes locally (on different ports) and runs the client experiments.

```
make test
```

**Expected Output:** You should see logs indicating nodes starting, files being uploaded/downloaded, and a final "ALL TESTS PASSED" message.

7. **Performance Experiments:**

Run the Java stress tests to generate the raw data (`results.csv`).

```
bashjava -cp out com.distributed.storage.client.PerformanceExperiments
```

Expected Output:

```
=== EXPERIMENTS COMPLETED. Results saved to results.csv ===
```

## 10 Individual Contributions

Both Samyak Shah and Mahip Parekh contributed equally (50-50) to the design, implementation, testing, and documentation of this project. We pair-programmed for the majority of the core features including the DHT logic, Chain Replication, and the Client-Server communication protocol.