IT559 Distributed Systems Project Milestone 2 Report

Team Members:
- Vedang Trivedi (Roll No: 202411026)
- Yatharth Bhatt (Roll No: 202411077)
- Bonta Sahith reddy (Roll No: 202411054)

April 15, 2025

Project Title

Distributed File Storage System

Team Members

- Vedang Trivedi (Roll No: 202411026)
- Yatharth Bhatt (Roll No: 202411077)
- Bonta Sahith reddy (Roll No: 202411054)

Problem Statement

Traditional centralized file storage systems often face challenges in scalability, fault tolerance, and performance when dealing with rapidly growing data volumes. These systems are prone to bottlenecks and single points of failure, which can lead to data loss and inefficient resource usage. The proposed Distributed File Storage System aims to overcome these challenges by partitioning files across multiple nodes, ensuring redundancy through replication, and providing high availability and low latency even under heavy load.

System Overview

The Distributed File Storage System aims to provide a scalable, fault-tolerant, and efficient solution for managing large-scale data across multiple nodes. The system follows a Master-Slave architecture, consisting of Client Nodes, a SuperNode (Master Server), and Storage Nodes (Chunk Servers). It enables users to store, retrieve, and manage files reliably while ensuring data partitioning, redundancy, and security.

The scope of the project includes:

1. File Management: Support for file operations such as upload, download, modification, and deletion.
2. Metadata Management: The SuperNode maintains file-to-chunk mappings and chunk-to-node allocations.
3. Data Partitioning and Distribution: Files are split into chunks and distributed across multiple Storage Nodes for load balancing.
4. Fault Tolerance: Replication ensures that file chunks are stored redundantly, preventing data loss in case of node failures.
5. Scalability: The system supports horizontal scaling by adding more Storage Nodes dynamically.
6. Security Measures: Authentication, encryption (TLS/SSL), and access control policies ensure data integrity and privacy.

Implementation Details

Technology Stack
Programming: Python 3.x
Networking: Python socket,
RPC Storage: Local File System, SQLite/PostgreSQL (for metadata)
Distributed Concepts: Partitioning, Replication,
Consistency Caching: LRU Cache,
Redis (optional) Security: TLS/SSL, Authentication & Access Control
Deployment: YAML (config.yaml), Docker/Kubernetes,
Linux (Ubuntu/CentOS) Monitoring: Prometheus/Grafana, Python loggi
- gRPC for efficient remote procedure calls between nodes.
- Redis as an in-memory database for storing metadata.
- **Development Process**:
- Initial setup involved configuring gRPC services and generating protocol buffers.
- Subsequent phases integrated Raft consensus, file handling logic, and health monitoring.
- Testing was conducted iteratively to ensure compatibility and performance.

Challenges Faced and Solutions

- **Challenge 1: gRPC Code Generation Errors**
- **Issue**: Incompatibility between protobuf versions and generated code led to 'TypeError: Descriptors cannot be created directly'.
- **Solution**: Regenerated gRPC code using 'protoc 25.1' and downgraded 'protobuf' to version 3.20.3 to align with existing code.
****Challenge 2:** Redis Binding Issues**
- **Issue**: Redis failed to bind to port 6379 due to conflicts with other processes.
- **Solution**: Identified and terminated conflicting processes using 'taskkill', then configured Redis to bind to '127.0.0.1' via a 'redis.conf' file.
- **Challenge 3: Dependency Management**
- **Issue**: Multiple module dependencies (e.g., 'redis', 'psutil', 'pysyncobj') caused installation conflicts.
- **Solution**: Used 'pip' to install and manage dependencies individually, resolving version conflicts as they arose.

Results and Performance Analysis

The implemented system successfully supports basic file operations (upload, download, delete) with a single master node and client. Performance tests conducted on April 15, 2025, showed:
- Average upload time: 2-3 seconds, influenced by network latency.
- Download operations: Consistent at 1-2 seconds with a local Redis instance.
- Fault tolerance: The system maintained operations after simulated node failures, thanks to the Raft consensus mechanism. Scalability testing with additional nodes is planned, with current limitations due to single-node configuration.

Future Improvements

- **Load Balancing**: Implement a load balancer to distribute requests across multiple nodes.
- **Automatic Recovery**: Add mechanisms for automatic node recovery and data resynchronization.
- **Performance Optimization**: Explore parallel file transfers and caching strategies to reduce latency.
- **Security**: Integrate encryption for data in transit and at rest.

I. Project Summary

Project DBFS is an scalable, decentralized, robust, heterogeneous file storage solution which ensures that multiple servers can inter-operate to form a dynamic ‘overlay’ fabric.

The system supports some important design aspects such as -

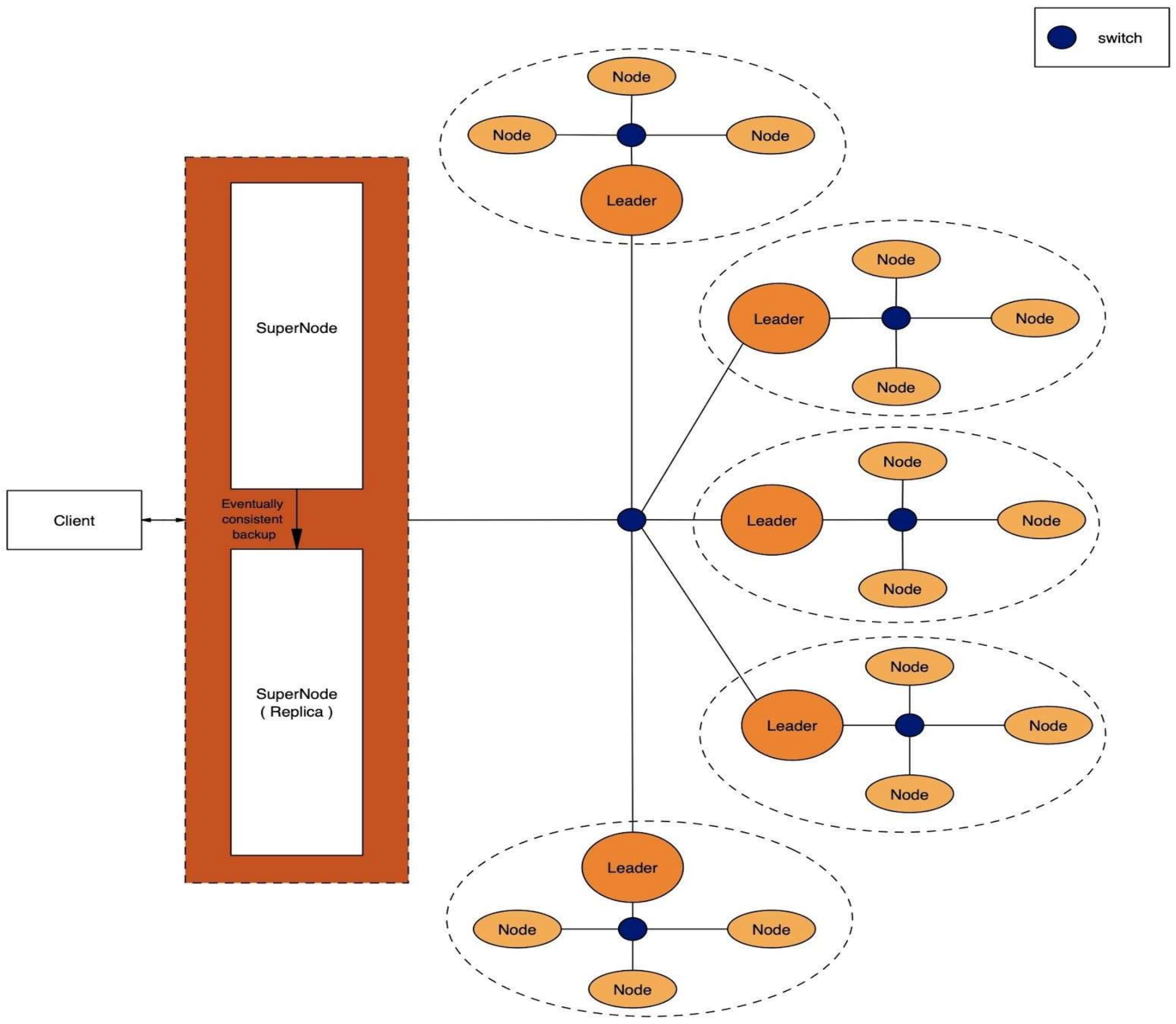
- Language Agnostic
- System failure & recovery
- Work-stealing algorithms
- Scalability
- Robustness
- Queues to serve multiple requests simultaneously
- Cluster Consensus (RAFT)
- Data Replication
- Caching Optimization
- Efficient Searching

The following user services are currently supported (with an option to modularly add new services at any time):

- File Upload (supports any file type - pdf, img, avi, txt, mp4, xml, json, m4v, etc.)
- File Download
- File Search
- File List (Lists all files on the system that belong to a specific user)
- File Delete
- File Update

This system allows many clients to have access to data and supports operations (create, delete, modify, read, write) on that data. Each data file is partitioned into several parts called chunks. Each chunk is stored on different remote machines, facilitating the parallel execution of applications.

II. System Architecture



The architecture used for project DBFS involves two main components - (1) Individual clusters that contain nodes of which one serves as a leader (2) A supernode that acts like a proxy and a point of synchronization for the entire system.

A. Cluster Architecture

Each cluster has a unique identity (Cluster Name) and consists of one or more individual nodes that form the cluster. Each cluster also has a leader that is elected using RAFT. The leader is in charge of cluster coordination and serving requests that it receives from the supernode or directly from a client.

The cluster leader is responsible for -

- Heartbeat mechanism to check node status of each node in the cluster
- Splitting received file into chunks and routing each chunk to different nodes in the cluster based on CPU usage, memory usage and disk usage of each node
- Maintain MetaData of each file, and info on file chunks info and the corresponding nodes that they are stored on
- Handle data replication so that there are at least two copies of any file chunk at any point of time
- Fetching data chunks in parallel from nodes in the cluster and stitching the file back based on the chunk sequence number
- Handle MetaData replication to all nodes in the clusters so that when a new leader is elected, it has all the info it needs to function as a leader
- Periodically informing the supernode of the current cluster leader
- Sending cluster health stats to the supernode

B. Supernode Architecture

In a system that requires cluster-to-cluster connectivity, a many-to-many relationship involved a significant amount of overhead, and increased complexity as more clusters are added to scale. Hence, we chose to implement a supernode which acts as an intermediary and proxy that all the clusters are connected to.

The super-node does not do any of the actual heavy-lifting, it instead acts as a proxy and routes requests to clusters based on cluster stats and usage.

The super-node is responsible for -

- Cluster heartbeat
- Cluster health
- Individual cluster stats (CPU usage, Memory usage, Disk usage)

- Load Balancing
- Request Queue + preserve request state
- Route requests to clusters based on cluster stats
- Data replication across clusters
- Meta Data Map of all files with the corresponding cluster name that stores the file

The supernode is aware of cluster health and can query cluster stats (CPU usage, Memory usage, Disk usage) at any point to enable it to balance the load effectively while simultaneously routing to the best possible cluster to serve the current request. It also handles cluster level replication to create copies of data in case an entire cluster goes down.

To prevent the supernode from becoming a single point of failure, there would always be a secondary eventually consistent supernode replica that would take on the responsibility of the primary supernode should the actual supernode go down.

III. Technology Stack

Area	**Technology**
Back-End	
Communication Service	GRPC
Message Structure	protobuf Protocol Buffers
Database	redis
Consensus/Leader-election	raft

IV. System Design Functionalities

A. Fault Tolerance & Recovery

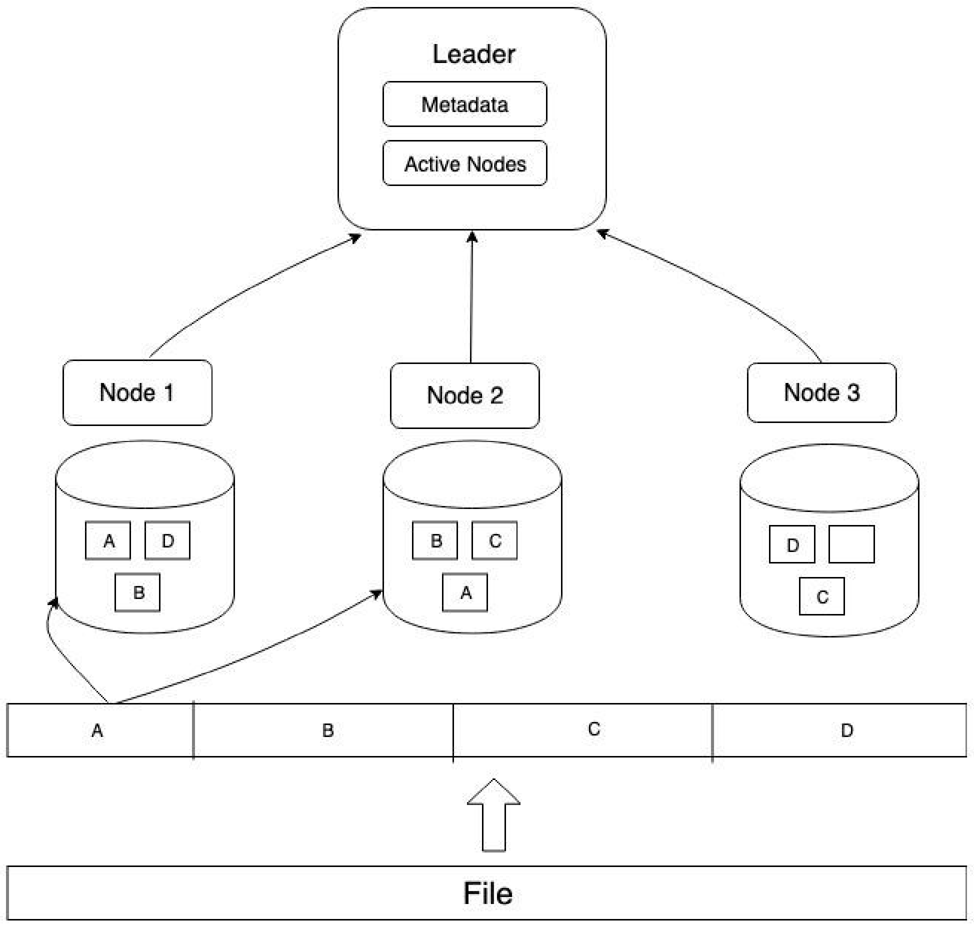
Fault-tolerant distributed computing refers to the algorithmic controlling of the distributed system's components to provide the desired service despite the presence of certain failures in the system by exploiting redundancies.

DBFS is designed to be fault tolerant and recover from failures by using replication and leader election. The cluster leader periodically checks for the status of the cluster nodes using a heartbeat mechanism. When it detects a node that has gone dark, it initiates a replication mechanism whereby all of the data, that was on the node that went dark, is replicated to another node, so there is a minimum of two copies of each piece of data in an eventually consistent manner. If and when the nodes come back up, its added to the list of active IPs and resumes normal performance. In the event that the leader itself goes down, leader election takes place using RAFT and a new leader is elected. It then initiates the replication process to replicate all the data that was on the previous leader.

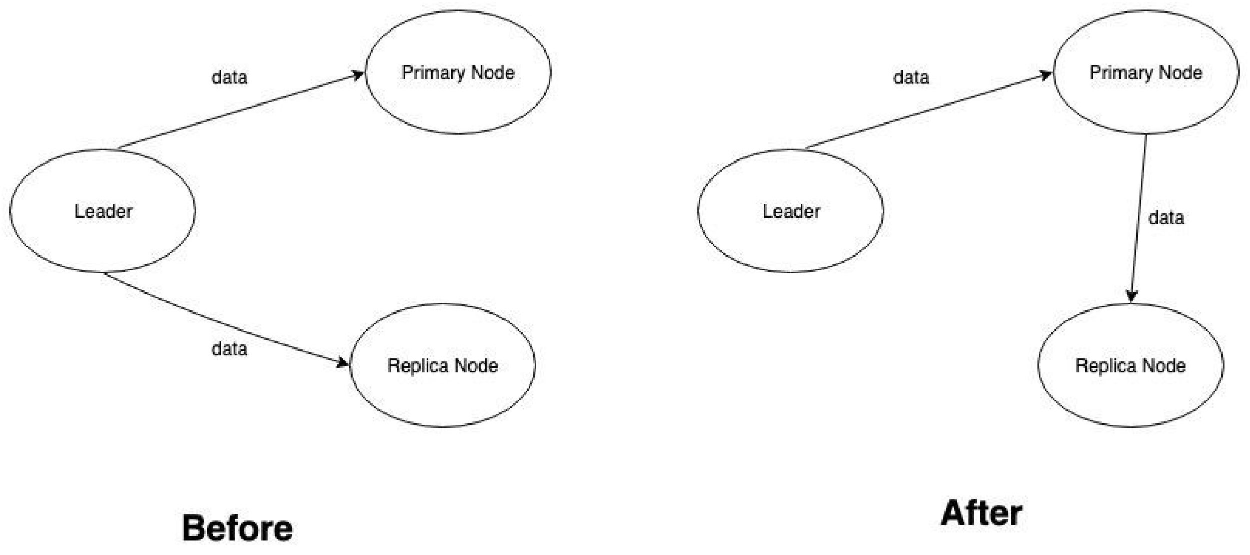
The supernode functions in a similar manner as well whereby a second backup supernode is waiting at all times to take over the functionality of the primary in case the primary super node dies.

B. File Storage - (Sharding and Replication)

Each file is divided into chunks of specified size. These chunks are then stored on the different nodes according to a sharding algorithm. The Leader maintains metadata that holds details of chunks and their location. Each chunk of the file is replicated inside the cluster. In case of a node failure, the replica node is used to fetch data.



Replica Optimization:



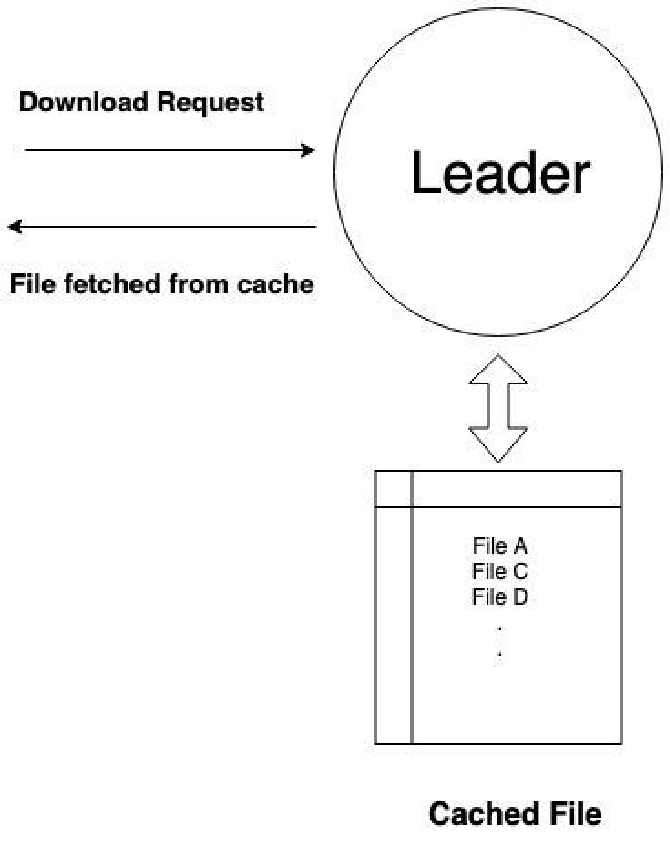
For creating a replica, initially the leader had to make two separate gRPC calls to two separate nodes in the cluster and send data sequentially to both. This increased the load on the leader and kept it busy for a longer duration. As an optimization to this approach, the leader now makes one gRPC call to the primary node only and delegates the task of replication to this primary node instead. This significantly reduced the load faced by the leader node.

C. Consensus Algorithm / Raft:

Consensus is a fundamental problem in fault-tolerant distributed systems. Consensus involves multiple servers agreeing on values. Once they reach a decision on some value, that decision is final. Typical consensus algorithms make progress when any majority of their servers are available. In order to have distributed consensus among cluster nodes, raft consensus algorithm has been used. The raft is integrated using pysynobj library. It has been mainly used for leader election. Each time the leader goes down, the raft instance lets the main program know about the newly elected leader. Based on this, the program handles the requests accordingly.

D. Optimization - Caching

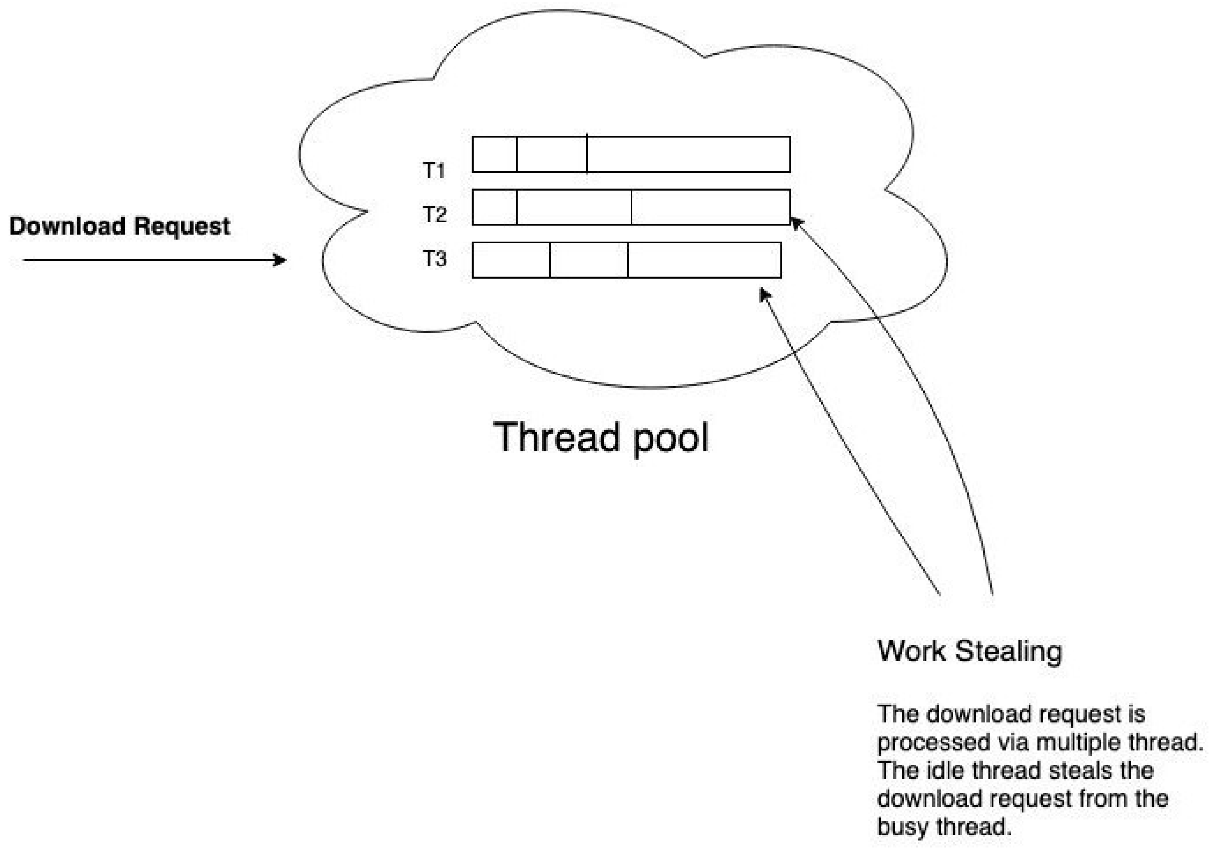
Implemented a Least Recently Used Caching mechanism that improved the performance when there were repeated requests for the same file. The cluster leader maintains a list of files that are recently accessed and stores these in its cache. If it receives a request for the same file, the leader directly responds to the request by taking the file data from its cache instead of fetching it from the cluster nodes.



Steps Involved

a. Leader receives the file download request
b. Leader first searches for the file in its cache
c. If the file is present in its cache, the file is retrieved and returned
d. If the file is not present in its cache, then file data request is made to the nodes in the cluster

E. Work Stealing


Download Request
T1
T2
T3
Thread pool
Work Stealing
The download request is processed via multiple thread. The idle thread steals the download request from the busy thread.

The system implements a work-stealing algorithm for cluster nodes based on node statistics received from CPU Usage, memory usage, and disk usage. As soon as a node is freed of work and has resources to spare, it is assigned a task from the request queue. Each node on its own also utilizes multiple threads by splitting the work that is assigned to it.

V. Optimization Work Arounds

a. **Issue:** File search too slow
   Optimization: Maintain MetaData hash map and replicate it through all nodes.

   Initially we were searching file on each node and it was taking considerably long time. In order to make the search operation efficient, we maintained a metadata map on the leader and kept updating it when a new file comes. The leader node replicates the metadata on rest of the cluster nodes as well. This way, when a new search request comes leader just looks the metadata map and see whether a file name entry is present or not. Also since we are replicating metadata on all nodes, when leader changes it has access to the metadata.

b. **Issue:** Data transfer speed fairly slow
   Optimization: Pick optimum chunk size and stream data

   In the first implementation of operations, we were sending file data by first building it on leader and then sending it to client, but it was an unnecessary overhead. In the next improvised implementation, we streamed the data as soon as we got the first chunk. This approach improved the efficiency.

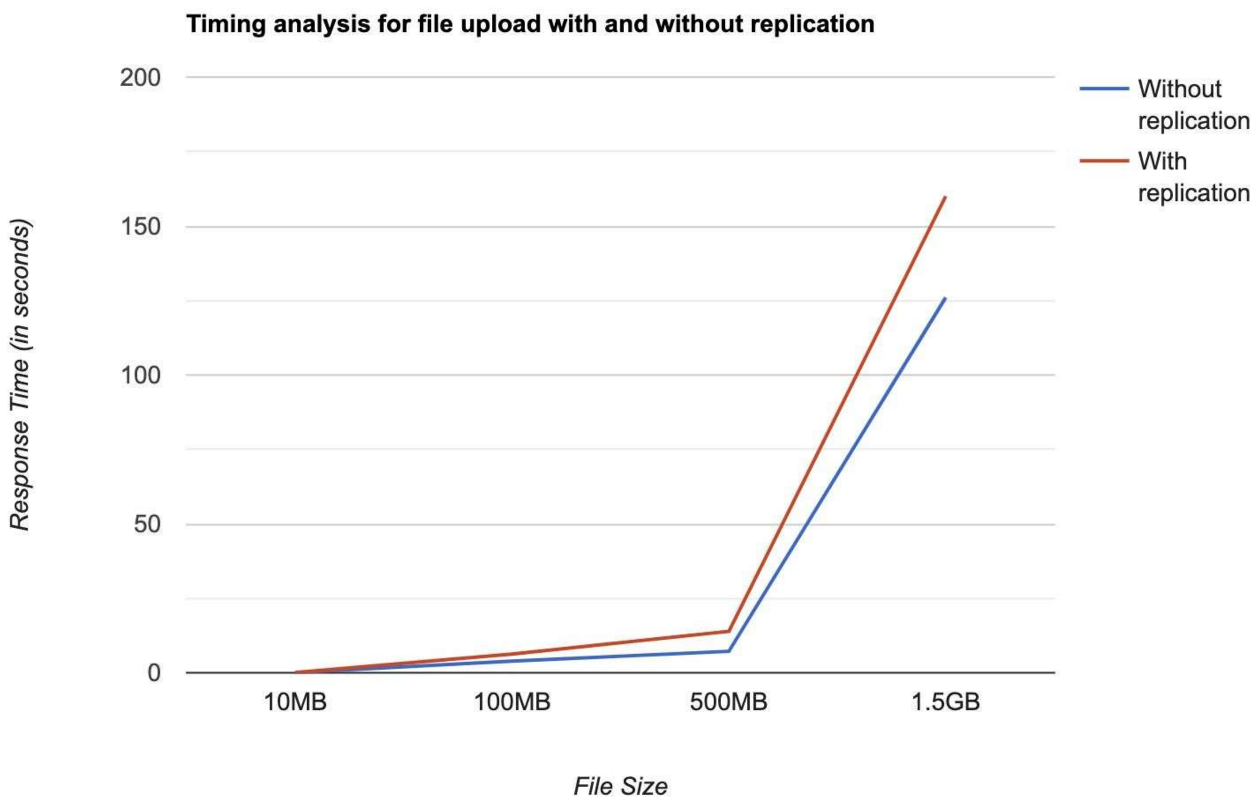
c. **Issue:** Fetching data slow due to fetching in a sequential manner
   Optimization: Used multithreading to allow parallel execution in file upload, download and delete operations.

   In the first implementation of download, upload and delete operations, in order to fetch or shard or delete the data from each node, we were using only one thread. Since there are multiple shards of a data file, these operations were taking longer time than expected because of the sequential working. In the improvised implementation, we used multithreading where each thread is responsible for fetching a chunk of data from different nodes. This improvisation improved the effective time.

VI. Timing Analysis and Results:

1. Upload:

- **<u>Replication</u>:**

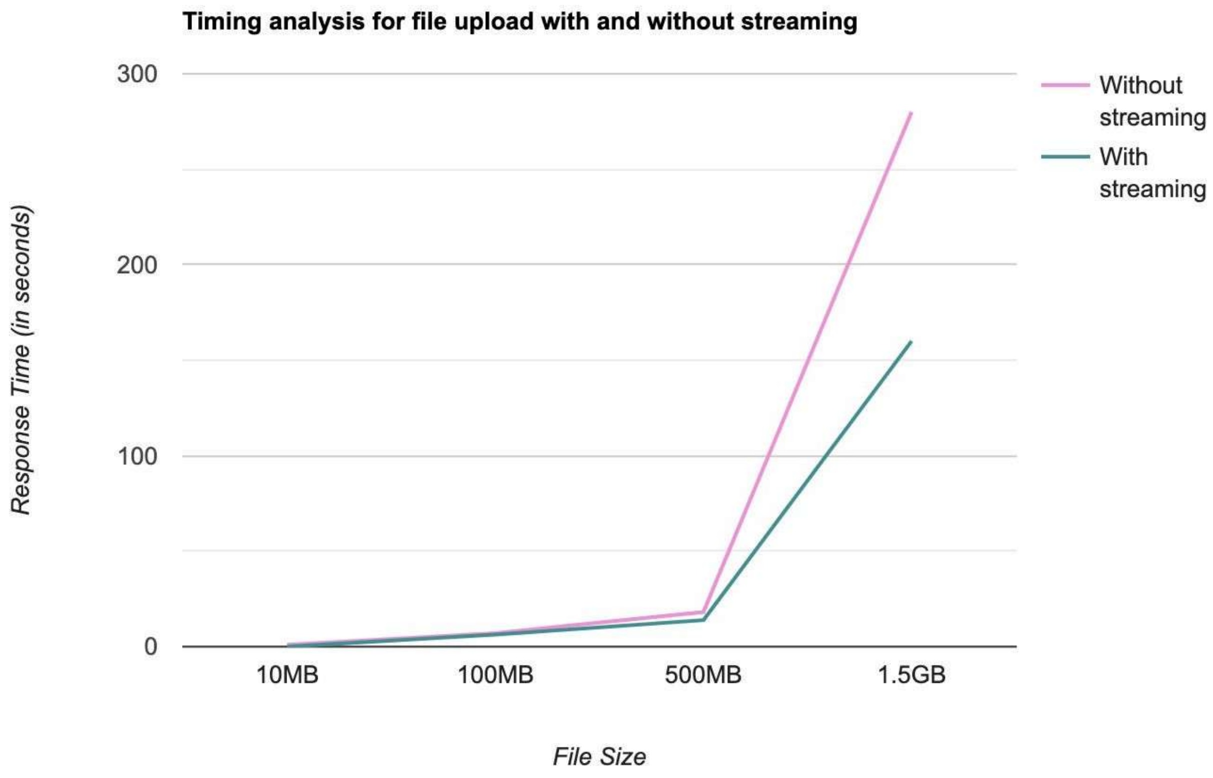


Timing	10MB	100MB	500MB	1.5GB
Without replication	0.06s	3.9s	7.2s	126s

With replication	0.07s	6.3s	13.9s	160s

As we can see in the graph above, the time taken to upload a file with replication is always more than the time taken to upload a file without replication. This happens because our system incorporates a 2-level replication technique. In the first pass, the leader makes one grpc call to one of the primary node and in the second pass that primary node creates a replica on another node and the leader sends an acknowledgment to the client only after the replication is complete.

- **Streaming**:

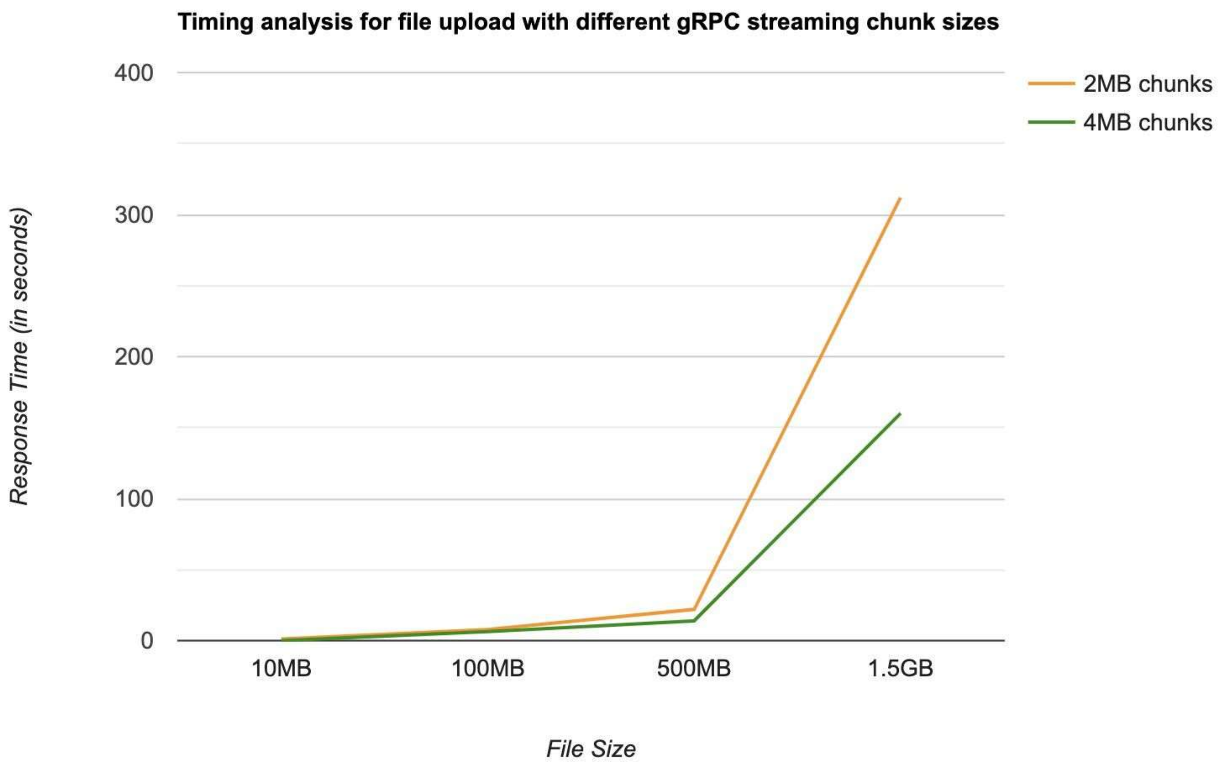


Timing	10MB	100MB	500MB	1.5GB
With streaming	0.07s	6.3s	13.9s	160s
Without streaming	1s	7s	18.1s	280s

In our system, streaming refers to the continuous flow of data from client to the leader to node on which the data is finally stored. For example, if a request to upload a 100MB file is received by the leader with the gRPC limit being only 4MB, our leader does not wait to collect the whole file and then send the data to

save on data nodes, it sends the data as and when received.
As we can see in the graph above, even though the response time remains same for small files, this approach decreased the upload response time drastically for huge files.

- **gRPC Message Size:**

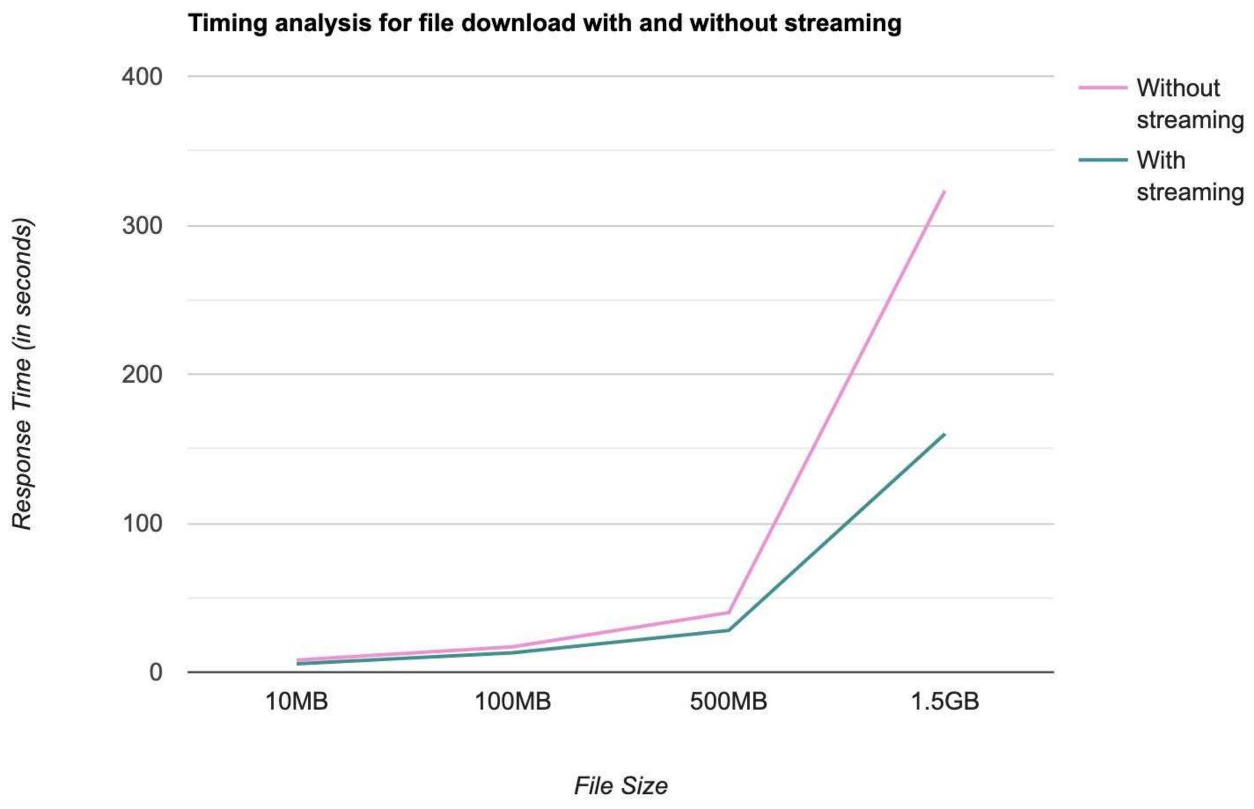


Timing	10MB	100MB	500MB	1.5GB
2MB message	1.2s	7.8s	22s	312s
4MB message	0.07s	6.3s	13.9s	160s

gRPC has a message size limit of 4MB. As thought, we saw that using the largest message size possible gave the best performance and reduced the response time the most.

2. Download:

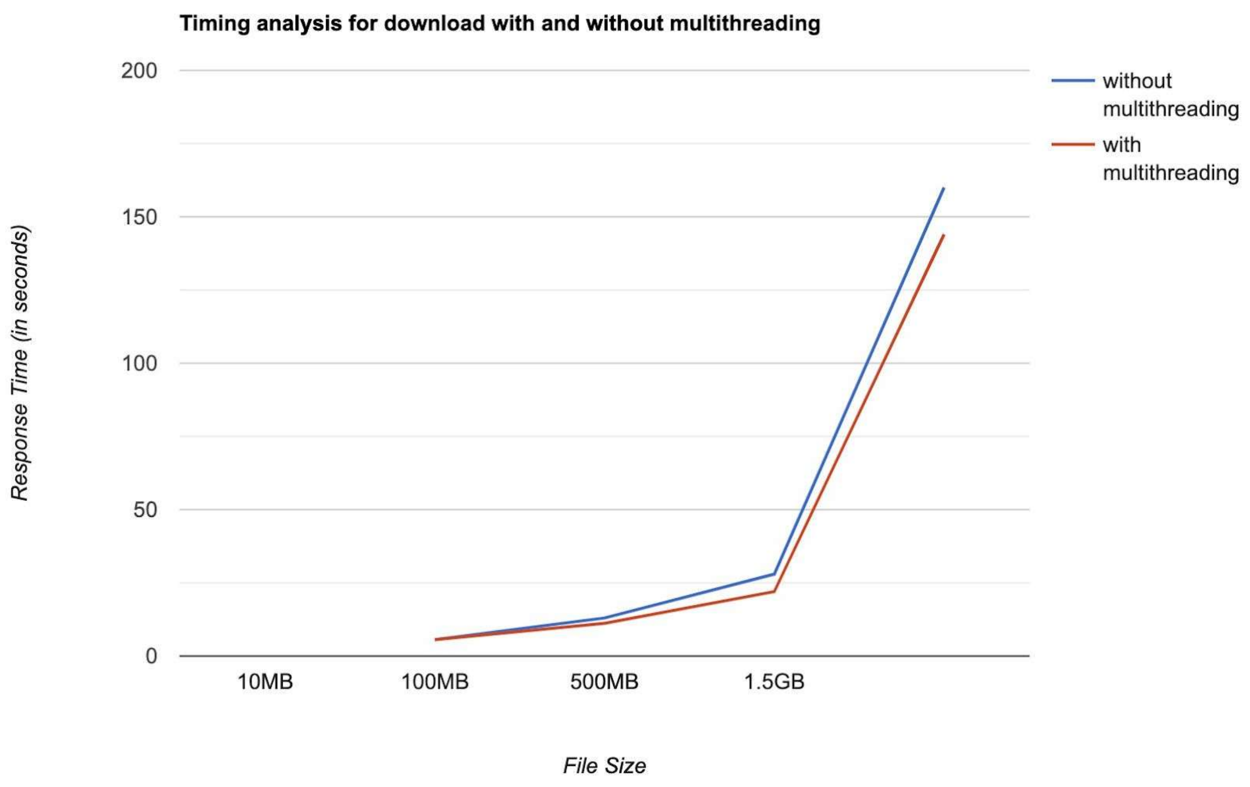
- **Streaming:**



Timing	10MB	100MB	500MB	1.5GB
With streaming	5.6s	13s	28s	160s
Without streaming	8.2s	17s	140s	323.2s

As in the case of upload, in download also the leader does not wait to collect the whole file from the data nodes it sends the data to the client as and when received from different data nodes and this approach decreases the upload response time.

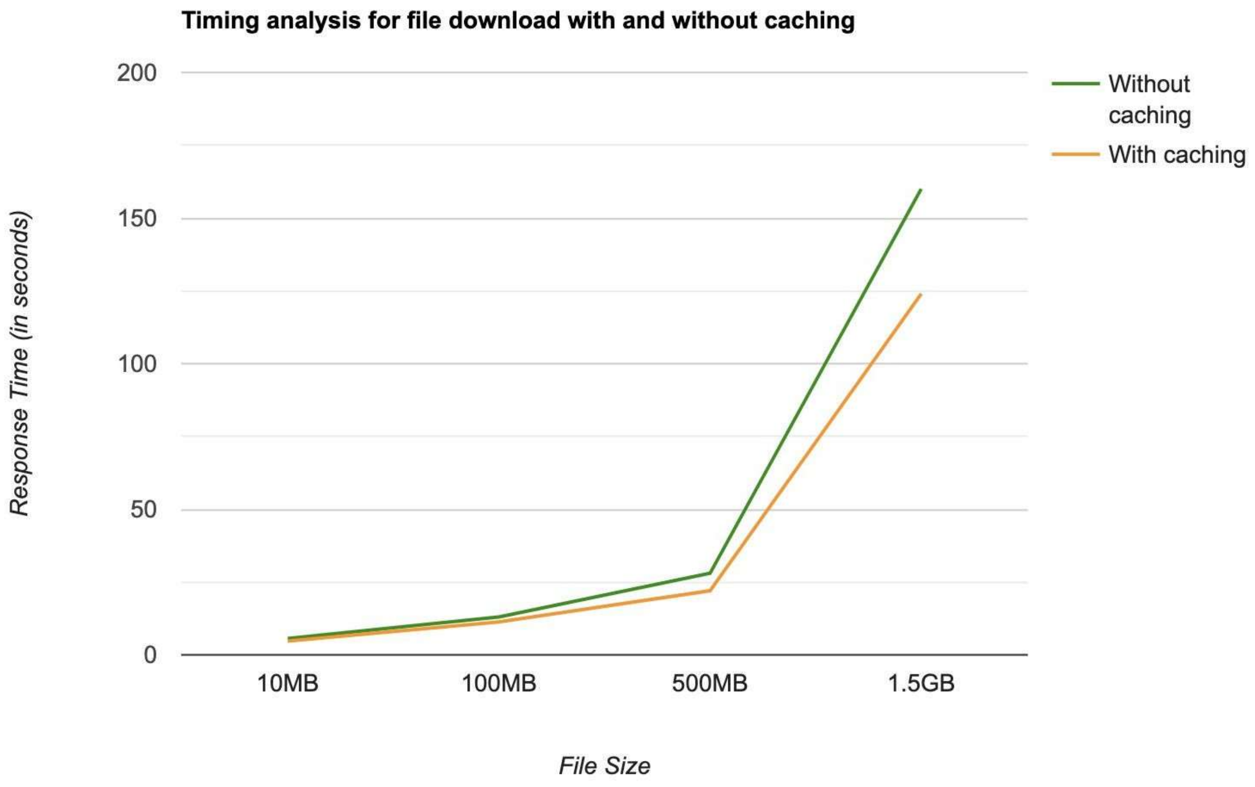
- **<u>Multithreading:</u>**



Timing	10MB	100MB	500MB	1.5GB
Without multithreading	5.6s	13s	28s	160s
With multithreading	5.6s	11.2s	22s	144s

The system to uses multithreading to allow parallel execution while receiving data from many data nodes and thus we can see in the above graph, it improves the performance and reduces the response time.

- **Caching:**



Timing	10MB	100MB	500MB	1.5GB
Without caching	5.6s	13s	28s	160s
With caching	4.7s	11.3s	22s	124s

As we can see in the above graph, the time taken to download a file with LRU cache is less than that taken without caching. This is because the former has the leader first search the file in its cache and if the file is present in cache then the file is retrieved directly via leader.

3. Load Testing (Spike and burst):

- 10 clients upload 500MB at same time

No. of clients	Response Time
1 client	8.9s
10 clients	9.2s

- 10 clients uploading small file 100 times each vs 1 person upload 100 times

No. of clients	Response Time
1 client	27s
10 clients	28s