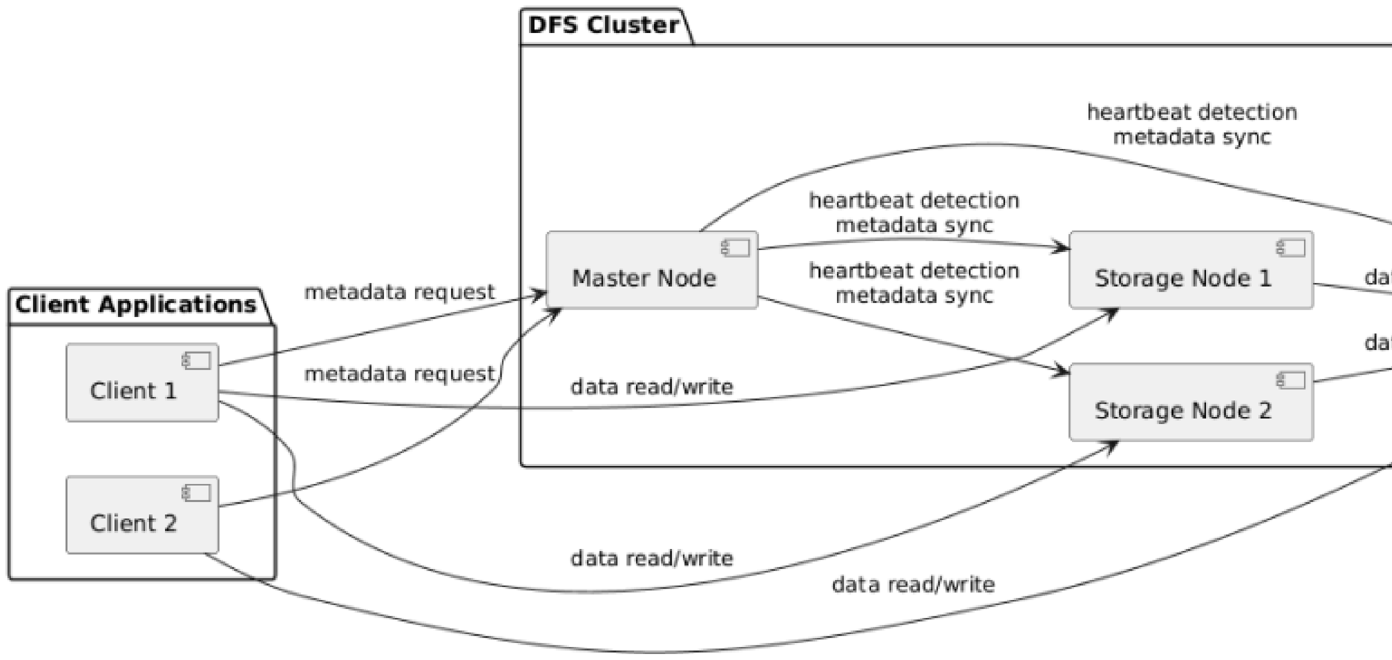
Design of Distributed File System

Overall System Architecture

The system utilizes a custom RPC framework to transmit messages between different components and respond to results. The data interaction relationships are illustrated in the diagram below. As shown, the controller serves as the central hub for metadata and data interactions between the client and storage nodes. The client directly interacts with storage nodes for reading and writing data, while data transmission also occurs between storage nodes.



RPC Framework

All interactions between system components are based on RPC, with specific encapsulations abstracting underlying message serialization processes. This simplifies communication between different components.

Inspired by how the HTTP server in Golang’s standard library handles different request methods and paths, this project’s RPC framework adopts a **"register handler"** mechanism. Different Protocol Buffers message types are used to distinguish requests, directing them to the corresponding handlers. However, designing a mechanism that allows the server’s multiplexer (Mux) to receive and deserialize different types of Protocol Buffers messages in a unified manner requires a more refined approach.

The solution in this project is relatively straightforward: Different message types are wrapped using any message type before being serialized with Protocol Buffers. On the server side, the payload encapsulated in the any type can be extracted into a specific message type simply by calling the UnmarshalNew method.

The RPC server’s message routing mechanism is simple: It maps message type URLs to specific handler functions, which share a unified signature and use the proto.Message interface as the request and response message type. When the RPC server receives a message, it invokes the registered handler function based on the message type. After processing, the response proto.Message is wrapped using the any Protocol Buffer message type and sent back to the client, which can then deserialize the message using the method described above.

Controller

The controller serves as the brain of the distributed file system. It is responsible for storing file metadata (including file names, file sizes, the sequence of file chunks, and the storage nodes where these chunks are located) as well as chunk metadata. Additionally, it manages and maintains these metadata records. The controller’s core functionalities include the following:

File Upload

When the controller receives a file storage request from a client, it generates the corresponding **file metadata**, determines the required **number of chunks**, and assigns **chunk sizes** along with **storage nodes** for each chunk. Each chunk is mapped to a list of storage nodes, where the **first node in the list** serves as the **primary storage node** for that chunk. The controller then returns the chunk list to the client, which subsequently sends chunk data to the respective primary storage nodes.

The specific steps are as follows:

1. The **client** sends a file storage request to the **controller**, including the **file name** and **file size**.

2. The **controller** checks whether sufficient storage space is available, generates a **series of chunk metadata**, and assigns **multiple storage nodes** to store **replicas** of each chunk.

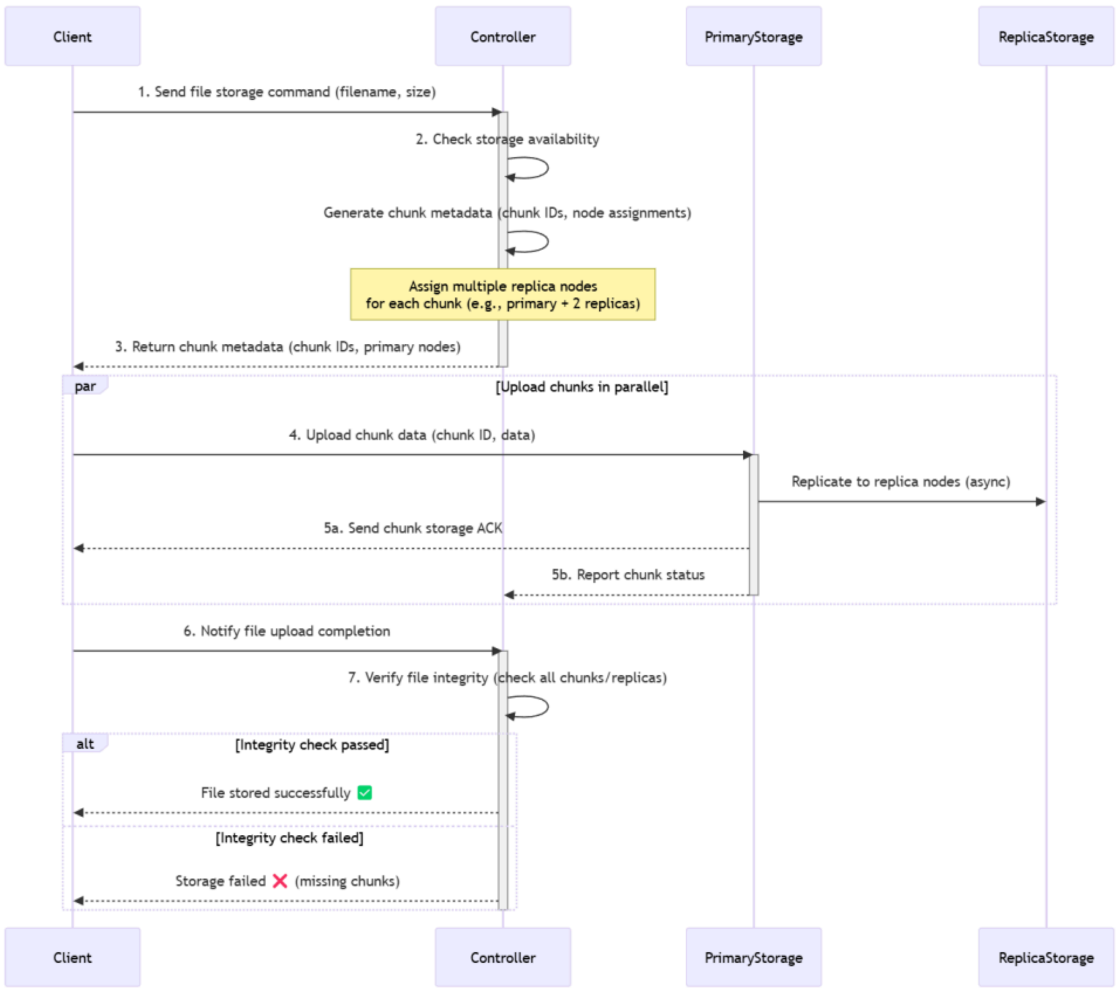
3. The **controller** sends the chunk metadata back to the **client**.

4. The **client** uploads chunk data **in parallel** to multiple **primary storage nodes**.

5. Each **primary storage node** performs the necessary **storage operations** and reports the **storage status** to both the **client** and **controller**.

6. After uploading all chunks, the **client** sends a **file upload completion message** to the **controller**, triggering the **file integrity verification process**.

7. The **controller** returns the **file integrity verification result**. If the file passes the verification, the upload is considered **successful**.



Client
Controller
PrimaryStorage
ReplicaStorage
1. Send file storage command (filename, size)
2. Check storage availability
Generate chunk metadata (chunk IDs, node assignments)
Assign multiple replica nodes for each chunk (e.g., primary + 2 replicas)
3. Return chunk metadata (chunk IDs, primary nodes)
par
[Upload chunks in parallel]
4. Upload chunk data (chunk ID, data)
Replicate to replica nodes (async)
5a. Send chunk storage ACK
5b. Report chunk status
6. Notify file upload completion
7. Verify file integrity (check all chunks/replicas)
alt
[Integrity check passed]
File stored successfully ✅
[Integrity check failed]
Storage failed ❌ (missing chunks)

File Deletion

When the **controller** receives a file deletion request from a client, it removes the **metadata** associated with the specified file and notifies the relevant **storage nodes** to delete the stored chunks.

Listing Files

The **controller** retrieves the **file metadata** stored in the system, generates a **file metadata list**, and sends the result to the **client**.

Chunk Replica Maintenance

At regular intervals, the **controller** checks the **replica count** of all chunks. If the number of replicas for a chunk falls below the required threshold, the **controller** selects **new storage nodes** to store additional replicas. This process prioritizes chunks with the **fewest replicas**, ensuring that chunks at the highest risk of **complete loss** are replicated first.

After determining the new storage nodes for all affected chunks, the **controller** groups them based on their **primary storage nodes**. Then, for each group, the **controller** sends **batch requests** (as collections) to the storage nodes, instructing them to replicate the chunks. The **storage nodes** then process each chunk replication request sequentially. Using **grouping and batch processing** helps prevent excessive **network I/O** from blocking other operations of the **controller**, thereby improving system **availability**.

Node Status Maintenance

Similarly, the **controller** periodically checks the **status** of all storage nodes. Each storage node sends a **heartbeat signal** to the controller every **10 seconds**. The controller determines whether a node is alive by checking if too much time has elapsed since its last received heartbeat. If a node is deemed **dead**, the controller removes it from the **replica lists** of all chunks it previously stored.

> **Note:** If the failed node was the **primary storage node** for any chunk (i.e., the first node in the **replica list**), removing it will cause the **next node** in the replica list to take over as the new **primary node**. However, this transition does not affect the storage nodes functionally, as there is no strict **primary-secondary** relationship between them. The primary node distinction is relevant only during **chunk uploads**, where the first node in the replica list is responsible for propagating the chunk to other storage nodes.

Once a node failure is detected, the **chunk replica maintenance mechanism** is triggered to redistribute the affected chunk replicas to other storage nodes.

Automatic Deletion of Expired Temporary Files

Temporary files refer to **incomplete files** that are created when a client initiates a file upload request but fails to **fully** or **successfully** upload all chunks to storage nodes. Upon receiving an upload request, the **controller** temporarily records the file as a **"candidate file"** in its metadata and adds it to the **candidate file list**. If the file is successfully uploaded, the **controller** moves it to the **regular file list**. However, if the upload is interrupted and the temporary file metadata and chunks remain undeleted, it can lead to **wasted server resources**.

To prevent this, the **controller** periodically checks for **expired temporary files** (currently set to **30 minutes**). If a file remains **unfinished** after **30 minutes** from the initial upload request, the **controller** automatically deletes its metadata and notifies the relevant **storage nodes** to remove the associated chunks.

Client

The core function of the **client** is to execute different **custom RPC commands** based on the requested operation. The most critical capability is **parallel file upload and download**. The implementation of this functionality consists of two key components:

1. **Parallelized File Reading and Writing**
2. **Parallel Task Manager**: Handles file upload/download to/from remote storage nodes and includes a fault-tolerance mechanism.

Parallelized File Reading and Writing

This project includes a **custom parallel file I/O package** (located at cmd/client/pfile). When a file read or write operation is required, this package **spawns separate goroutines** to handle the requests. It reads from or writes to the **filesystem** within the specified file range and then sends the results back through the response channel included in the request. The calling goroutine, which initiated the read/write operation, processes the result accordingly to determine the next steps.

Parallel Task Manager

The **Parallel Task Manager** (located at cmd/client/taskmgr) introduces the following key concepts and abstractions:

1. **Parallel Goroutine Context (context)**:
   - o Each parallelized task is associated with a **shared goroutine context**.
   - o Tasks within the same **task group** share the same **goroutine context**, meaning they can only run within that specific goroutine and cannot be transferred to another.
2. **Task Context (taskContext)**:
   - o A **task-specific execution context** that contains metadata relevant to the **current task**.
   - o This context is passed to the task function to ensure it has the necessary information for execution.
3. **Task Handoff**:
   - o If a **task fails**, the task function can invoke a **specific method** provided by the **Task Manager** to transfer the task to another **parallel goroutine**.
   - o The failed task is then reassigned and bound to a different **parallel goroutine context**, allowing continued execution while maintaining **fault tolerance**.

Client Parallel Upload/Download Logic

The logic for **upload** and **download** is similar. Below is an example focusing on the **upload** process.

When uploading a file, the **client** first retrieves the **chunk metadata** from the **controller**, which indicates the storage nodes to which each chunk needs to be uploaded after the file is split into multiple chunks. The **client** then collects all the relevant **storage nodes** and establishes **RPC connections** with these nodes. Each connection channel is treated as part of the **parallel goroutine context**.

Next, the client groups the chunks based on the **primary storage node** for each chunk. Each group is associated with the **RPC connection channel** of the primary node's **parallel goroutine context**, which is used as the **task context**. Once the grouping is completed, the client creates a separate **goroutine** for each context to **serially execute** all tasks within the group. The task function reads the specified file range from the **task context** and uploads the chunk data to the corresponding storage node using the **RPC connection channel** provided by the parallel goroutine context.

If the upload fails, the task function removes the failed storage node from the chunk' s **replica list** and hands the task over to the next storage node in the replica list. The handoff occurs to the **parallel goroutine context** associated with the next node, where it will attempt to upload the chunk. If it fails again, the task will proceed to the next node in the list. If all nodes fail, the upload fails; otherwise, the upload is successful.

Storage Node

The **storage node** stores the actual content of the chunks along with their **metadata** (including **ID**, **hash**, and **size**). Its primary functions include:

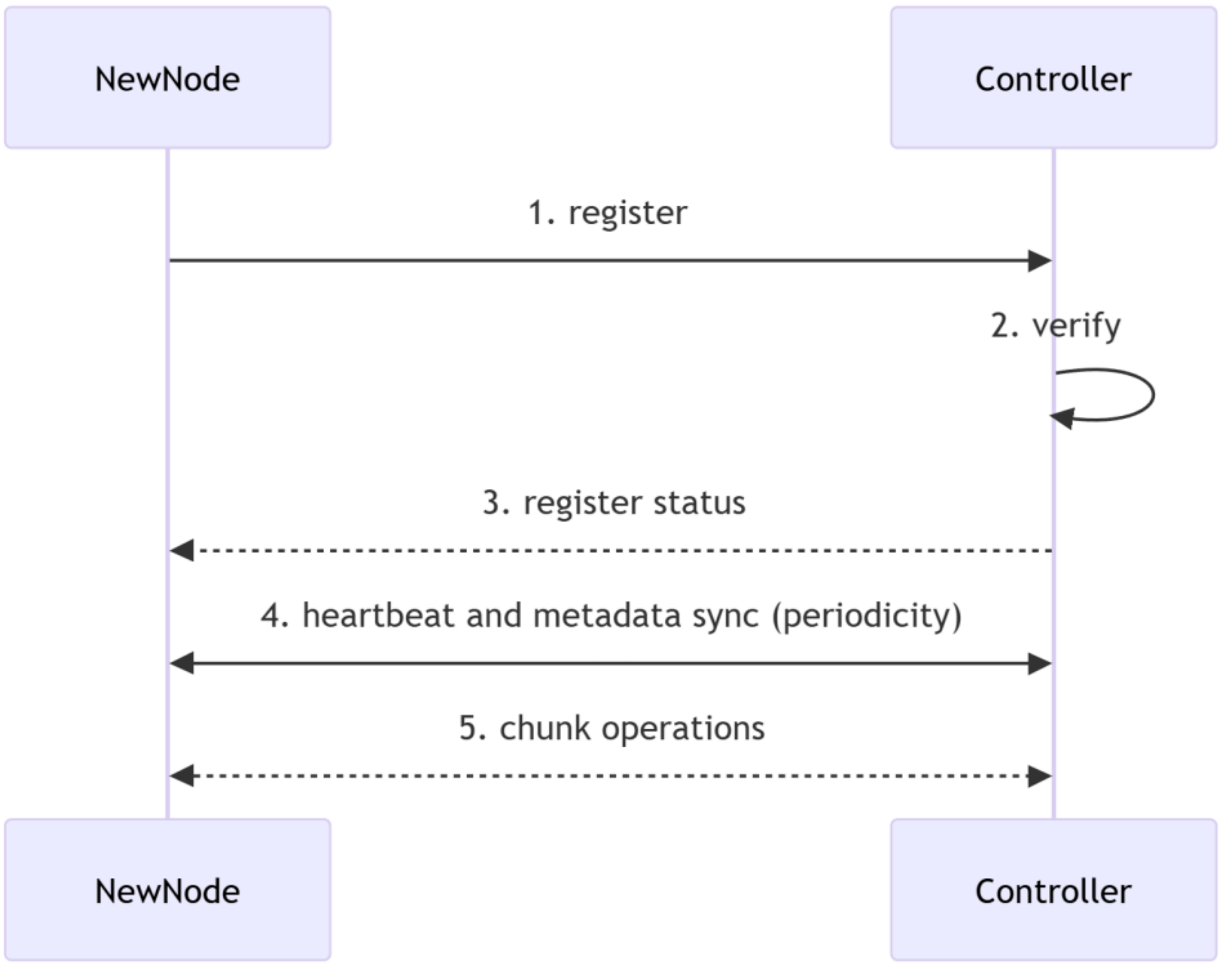
- **Providing read/write interfaces for chunks**: These interfaces allow the **client** to upload and download chunks.

- **Providing interfaces for adding and deleting chunks**: These are used by the **controller** and other storage nodes to coordinate the number of chunk replicas (e.g., deleting chunks, sending chunks to other storage nodes, etc.).
- **Heartbeat**: The **heartbeat** ensures the **controller** is aware of the storage node’s **status** and transmits chunk-related metadata, providing **guaranteed consistency** between the **controller** and storage nodes.

Storage Node Startup Process

After starting, the **storage node** first creates a directory for storing chunks and listens on a specific port to receive **RPC requests** from the **controller** and **client**. Then, the **storage node** follows the process outlined below to establish a connection with the **controller**. The detailed interaction steps are as follows:

1. **Registration**: The storage node sends a **registration message** to the **controller**.
2. **Verification**: The **controller** verifies the node's **uniqueness** and checks whether it can join the **distributed file system** managed by the current controller.
3. **Registration Response**: The storage node is informed of the result of its **registration**.
4. **Heartbeat and Metadata Synchronization**: The storage node periodically sends **heartbeat signals** to the **controller**, which includes the list of **chunk IDs** owned by the node. The **controller** also uses the heartbeat to send the list of chunks that need to be deleted to the storage node, ensuring **eventual consistency**.
5. **Chunk Operations**: The **controller** can send commands to the storage node, such as **deleting chunks** or **sending chunks to other storage nodes**. The storage node, in turn, notifies the controller about the creation and storage status of new chunks.



Chunk Data Integrity Check

The **storage node** uses **MD5 hash** to verify the integrity of chunk data. Integrity checks are involved in the following two processes:

1. When a command from the **client** or **controller** requires the storage node to read chunk data from its filesystem, the node reads the entire chunk data and checks if its **MD5 hash** matches the **MD5 hash** stored in the chunk's metadata. If there is a mismatch, the node returns an **error** to the client or controller.

2. The storage node actively performs a **data integrity check** on any chunk that has not been checked for over 20 minutes. It reads the chunk data, computes the **MD5 hash**, and compares it to ensure the chunk data is not corrupted.

If any chunk data is found to be corrupted, the storage node will remove the chunk from its **metadata list**. During the **heartbeat and metadata synchronization** process, it will synchronize the chunk deletion information to the **controller**. The **controller’ s replica maintenance mechanism** will then restore the chunk’s multi-replica status.

Chunk Upload Process

The **client** does not upload the chunk to all the servers that should hold the chunk. Instead, it only uploads the chunk to the **first server** listed in the **replica list** (the **primary storage node**). After receiving the chunk from the client, the **storage node**, upon recognizing that it is the **primary storage node**, automatically sends the chunk data to the other storage nodes listed in the chunk’s metadata that should hold the chunk. Once these replicas are successfully uploaded, the node informs both the **client** and the **controller** that the chunk has been successfully uploaded.