Whispering Woods

Introduction

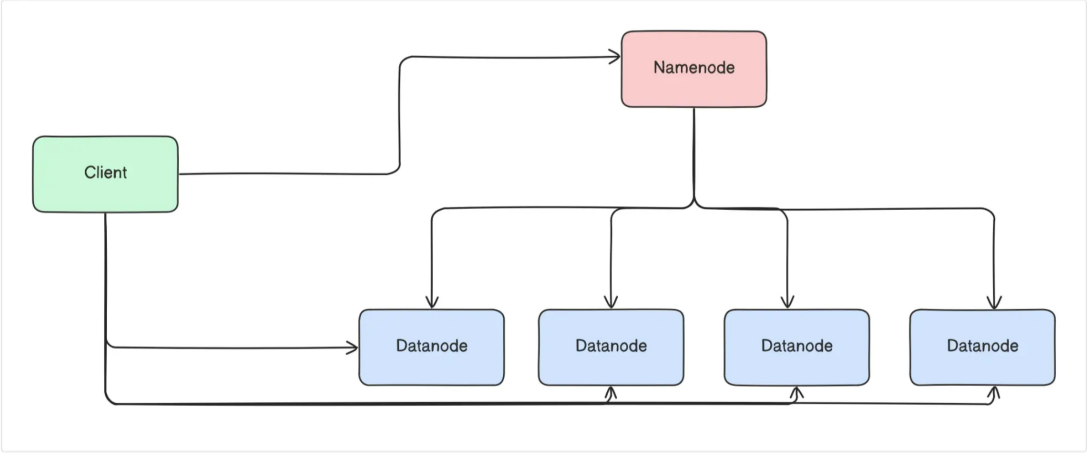
This project is a Distributed File Storage System inspired by the Google File System (GFS) paper. The motive of this project is to create a file system which will support storing a file, retrieving

an existing file and deleting a file. It should also be fault-tolerant against network partitions and node failure, Disk failure, Namenode Crash.

High Level View

In this section we will discuss the high level design of this system. The system consists of three major components forming one file storage cluster. Each of these has its specific role in the system. System will also use the external components like Namenode will use the Log Server for persistent storage , ElasticSearch and Kibana for distributed log management, APM server for performance monitoring. Each of these component is explained below :

1. **Client** : This is a user facing component of the system. User will only connect using the client which will be responsible for connecting to Namenode and Datanode, to fetch and store data. It is basically an abstraction on whole system to make it easy for users to use the system. Primary tasks of client are listed below :
    1. **Store file** : It will receive file from user and then request target Datanodes from the Namenode. For the file storage and then connect to Datanode sent by Namenode to store the data, It is responsible for dividing the file into chunks based on the offsets provided by the Namenode and transferring these chunks to assigned Datanode.
    2. **Fetch File :** Second task is to fetch the file from system as chunks and provide it as a single file to the user, it will connect to the Namenode to fetch the details and location of the chunks of stored file and then it contacts to the Datanodes to fetch chunks, merge them into single file and return it to the user.
    3. **Delete File** : Client can also request the Namenode to delete a file. In this task clients only responsibility is to send request to Namenode after that it is Namenode's responsibility to remove all the chunks associated with file.
    4. **Retry on failure:** It is also responsibility of the client to retry in case recoverable/ retry-able error occur in above tasks, the retry mechanism used is based on the policy (Currently uses exponential backoff) selected.



Components of Whispering Woods

2. **Namenode** : This is primary component of the whole system, it is responsible for maintaining entire metadata about the data nodes , stored files location, providing client with locations, checking the heartbeat of the Datanodes. It does not store actual file data but maintains metadata. Persistent state is managed via the Log Server. It is responsible for load distribution across Datanodes and maintaining the replication factor (handle under and over replication blocks). Primary tasks of Namenode include below tasks :

    1. **Provide data nodes to store** : This component is responsible for serving the client with data node list for file storage. This is done based on the current Datanode status and load, It uses configurable policies to choose best candidates among all.
    2. **List chunk location of file** : Client can request Namenode about the location of the stored chunks. It is done based on the metadata maintained by it using the state sync messages, and current load on the Datanode to avoid the hotspots.
    3. **Maintaining the replication factor** : It is responsible for working as watchdog to check if any block is under replicated/ over replicated, it will connect to data nodes to order them for replication or deleting the block in order to handle both under and over replication.
    4. **Maintaining metadata** : It will maintain the metadata about current status of cluster i.e. which Datanode is under load, where is chunk located, remove node from location if Datanode goes offline etc.
    5. **Certificate Authority**: The Namenode also acts as a Certificate Authority (CA). All requests made to the NameNode's gRPC interface are first authenticated using a certificate provided by the requester. These certificates are issued directly by the NameNode through its exposed REST API endpoint. Clients or administrators can request new certificates by authenticating via the API. Possession of a valid certificate is mandatory for any communication between a Client or DataNode and the NameNode.
    6. **Key Distribution Center**: The NameNode functions as a centralised Key Distribution Center. Each node in the system (except the NameNode itself) is assigned a unique secret key, generated through the NameNode's REST API by an authorised (admin) client. This secret key is known only to the respective node and the NameNode and is used to encrypt and decrypt Kerberos-inspired tickets.
    7. **Ticket Authority**: The NameNode also serves as the Ticket Authority for the entire system. Every operation that targets a component other than the NameNode (for example, Client → DataNode or DataNode → DataNode communication) requires a ticket issued by the NameNode. Each ticket contains encrypted metadata that verifies the requesting node's identity, target node permissions, and validity period. Tickets are encrypted with the client's secret key and embed a server key encrypted with the target node's key, allowing secure mutual verification — similar to the Kerberos model.

3. **Datanode** : This component is primarily responsible for storing the data. It will respond to client requests to store chunks and is also responsible for piping the data stream received from the client to other peers in order to do replication. Primary task of Datanode include below tasks :

    1. **Store data** : Primary task of this component is to store the data. It stores data in two cases one when client sends data or from other data node (during the data piping for replication). This is done using the TCP based protocol.
    2. **Delete data :** Datanode can delete the stored chunk. It will do this only after receiving a delete chunk message from Namenode, The client cannot directly request Datanode to delete the chunk this request must go through the Namenode.

Namenode can also instruct the Datanode to delete the chunk in case of over replication.

3. **Heartbeat messages** : It is responsible for sending heartbeat message to the Namenode to prove its liveliness to the Namenode.
4. **State sync message:** Datanode also share its current state to Namenode which helps Namenode to maintain metadata and take decisions based on this. This message includes stored chunks, available storage and details regarding the current load on Datanode.
5. **Piping data** : When Datanode receives the chunk store request (gRPC) it also receives details of next peers to which this chunk should be replicated based on this request Datanode will forward the data stream to peer Datanodes for replication.
6. **Chunk Replication**: When Namenode Finds one chunk under Replicated or over replicated it sends a gRPC message to datanode, datanode handles these requests by deleting a chunk or replicating the chunk to target datanode.

The above high-level view outlines the core responsibilities of each component that should be fulfilled in order to system to work.

Tech Stack

For this project I am using Rust as a primary language for programming. Project uses both **gRPC**, custom **TCP** protocol for communication, Beside from this namenode also expose the **REST api** for **certificate** and **key generation**, system state snapshot for dashboard I have used multiple Rust crates in this project, most important among them is described below:

- **Tokio**: Tokio is a one of the most popular Rust crates it provide the async runtime for Rust. I am using async methodology in this project since this is more of IO bound application rather than CPU bound. It is also one of the dependencies of other crates i am using like tonic.
- **Tonic**: Tonic is Rust crates which provide gRPC implementation in Rust. I have used this crate in order to establish gRPC based communication between different components.
- **Tracing**: This is widely used crate for logging purpose This crate make it easier to log based on the spans created by different functions. These logs are easily integrated with opentelemetry for APM integration.
- **Once_cell**: This crate is used to store non-copy types this allow us to declare a static variable which will be shared by all functions. With this we can declare the variable lazily. I have used this crate to make connection pools available to all.
- **nix**: This crate is used to get system information like cpu ussage, storage availabilty, bandwidth etc.This provide access to the lower level OS utilities which are helpful in getting available storage, bandwidth etc.
- **Figment**: This crate is used to load configuration from files. In our system we are using this to load configurations from yaml files.This carte needs the Serde crate in order to use derive macro for structs. Future plan is to implement runtime configuration changes based on the file changes.
- **UUID**: we use UUID to assign unique ids to chunks, this is used by Namenode to generate unique identifier for each chunk.
- **Rocket**: This crate is used to implement REST Api server Namenode use this crate to provide endpoints for certificate and key generation and provide the system snapshot for dashboard.
- **Jsonwebtoken**: This crate is used for authentication both utilities and namenode crate use this crate to generate and validate JWT for authentication.
- **Rcgen**: This crate provide the cryptography related functionalities used in both certificate generation and key generation for ticketing.
- **Base64**: This crate is used to handle encoding decoding of keys and certificates for better transmission.
- **Rcgen**: This crate provide the cryptography related functionalities used in both certificate generation and key generation for ticketing.
- **Webpki**: This crate is used to verify the certificates, using this crate we can generate the cert anchor by providing the root cert and then check if the cert is valid or not.
- **X509_parser**: This crate is used to generate the X509 certificate from the Der format.
- **Aes-gcm**: This crate provide implementation of the AES algorithm which will be used for both ticket encryption and ticket decryption.

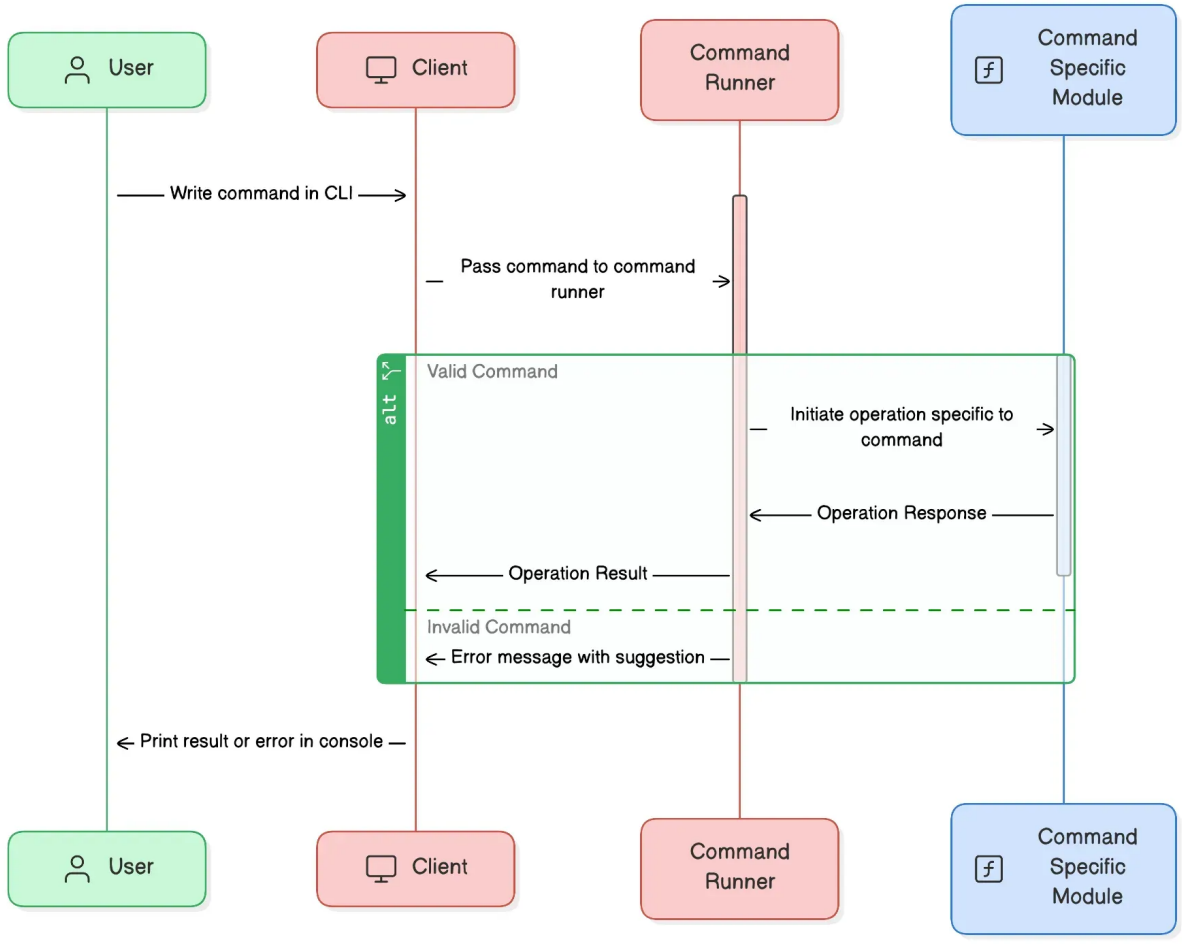
Deep Dive: How Things Work

In this section we will discuss high level view of system and how different tasks are supported by system. It will provide us with the clarity how each component of system interact with each other. we are using CLI as the medium for sending instructions from the user to the client. The user can type the instruction with arguments to client CLI and client will complete task as instructed by user. Complete details related to each operation is discussed below:

User interaction with client

User interact with the client with the help to CLI, to execute an command user need to write this command in the console of client. Client provide commands to store, fetch and delete file. Each command typed by user is passed to a command runner module which check for command validation and pass it to the specific module responsible for running that particular command.

To run commands on the namenode client will need a certificates which will be generated using the admin credentials through rest api, client will need both its certificate and secret key. We will explain the authentication in details in separate section.



User Client Interaction

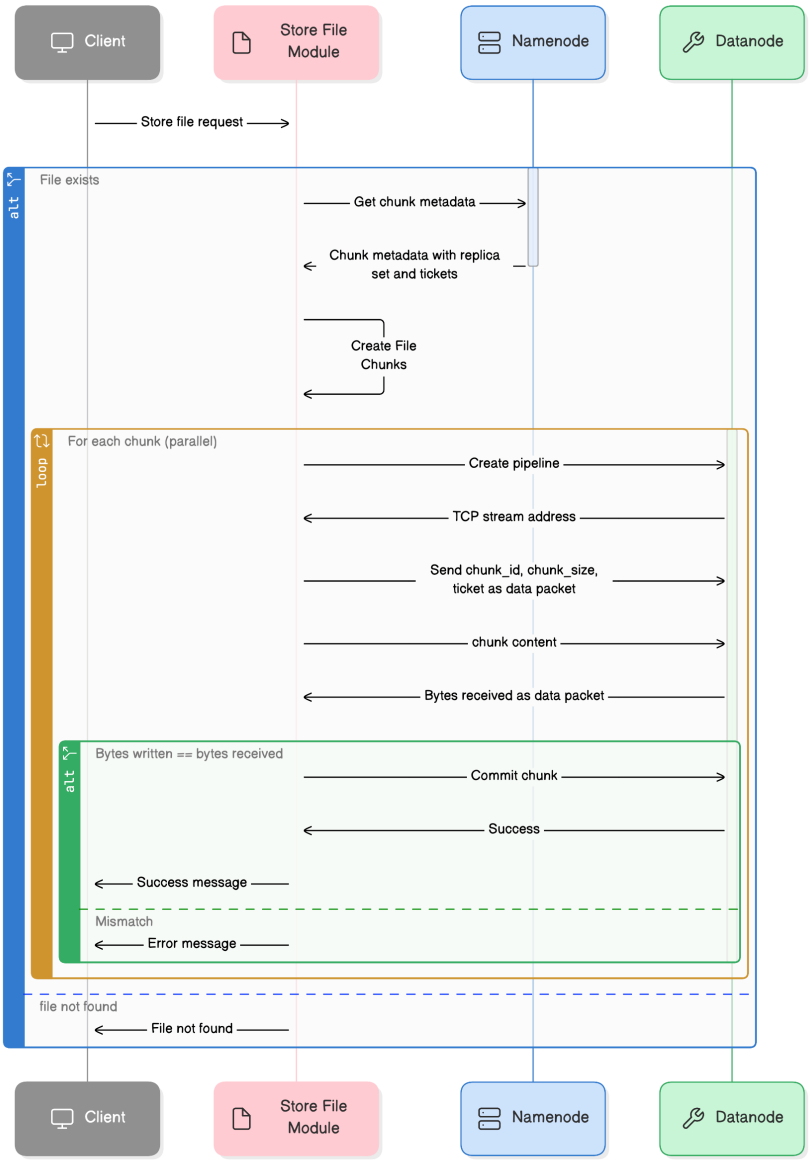
From now on we will skip the user from the flows, and discuss flow between client, Datanode and Namenode.

Store File

This task is initiated by the client and involves multiple steps. It starts with a simple gRPC message to the Namenode and completes when the client receives an acknowledgement from the Datanode over a TCP stream. we will discuss each step in view of all client, Namenode and Datanode in details here.

In **clients view** following operation will be completed:

1. Client will receive store file instruction from user with the path to file user want to store.
2. Client will check for the existence of the file and then read the metadata of the file, Now client will have the size of file which is required to get the chunk metadata from Namenode.
3. After that client will send the store file request to Namenode and in return the Namenode will be reply with chunks metadata along with tickets for direct datanode access.



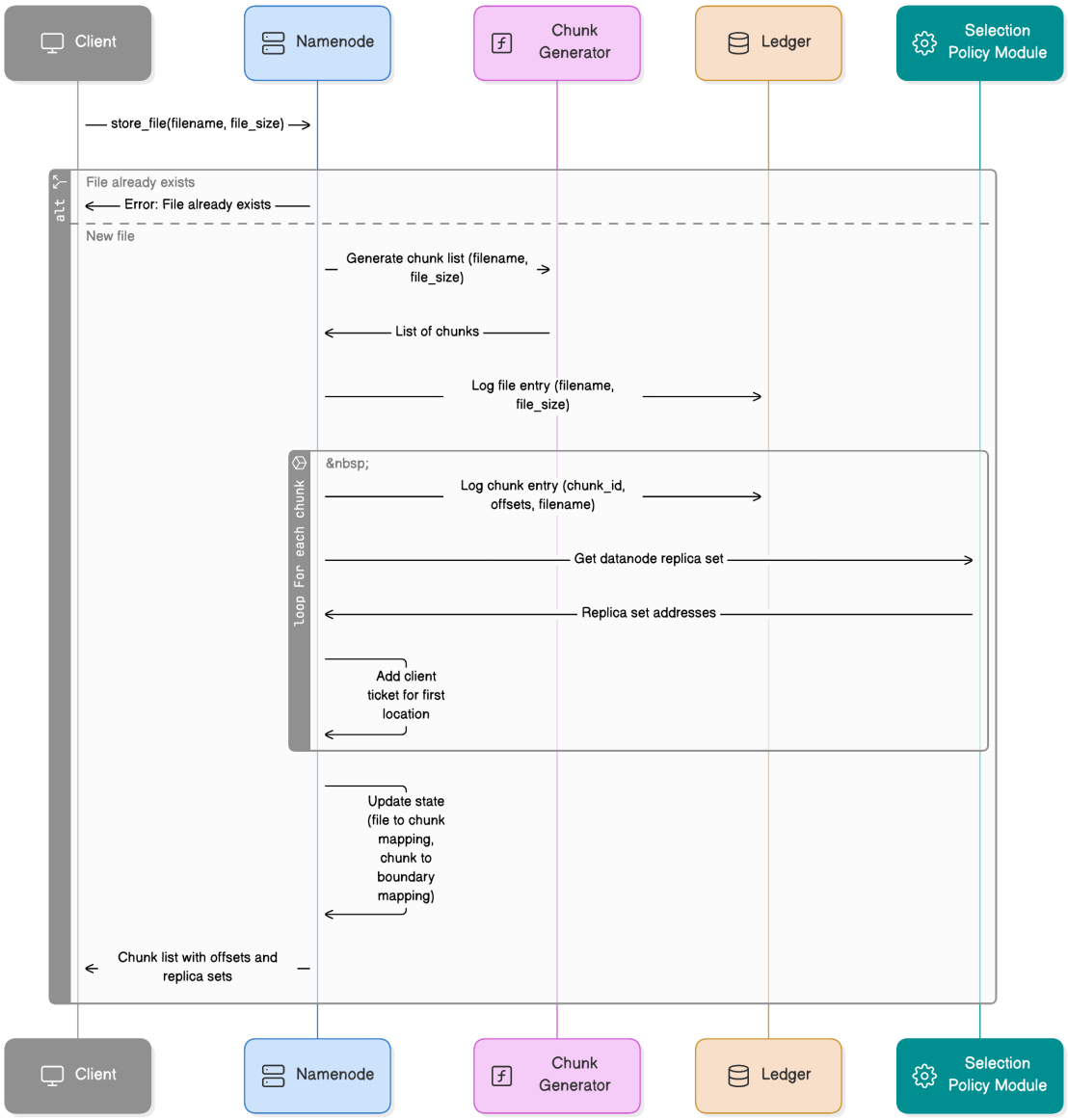
Client View of store file operation

4. Based on chunk metadata client will divide the file into chunks and then for each chunk send Store chunk request to the Datanodes specified in chunk metadata, providing chunk_id and replica set for that chunk.

5. As response to Store chunk request client will receive the TCP listener address for that Datanode, client will try to connect to this TCP listener and transfer the chunk.
6. On connection client will send a data packet containing mode, chunk_id, number of bytes, ticket.
7. After that client will transmit all the chunk data to stream and receive reply packet containing the received byte count.
8. Client will check for mismatch between bytes received and bytes sent. If everything is according to requirement client will send the commit message to the Datanode.

In **Namenode View** following tasks are done in order to complete this operation

1. Namenode will receives the store_file gRPC request from the client. This request will contain filename and file size, file name here is target filename not file location on client instead it is a filename that client provided to store file as.



Namenode view of store file operation

2. Namenode will pass these details to Chunk Generator which in return give us list of chunks, each chunk will be of same size (except last one, it may be same or not), each element in list

contain the details about chunk unique chunk_id (generated using UUID), start_offset and end_offset.

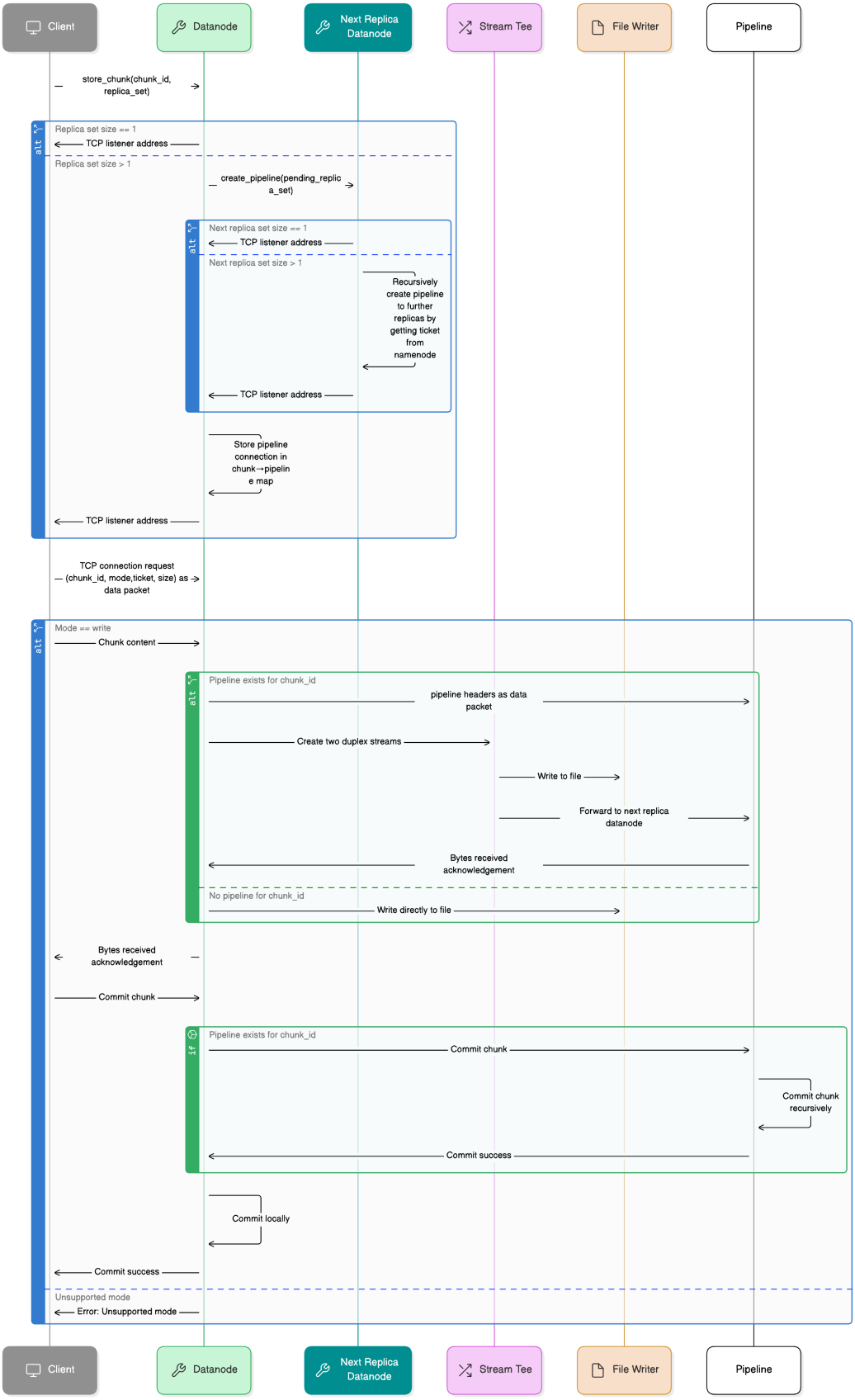
3. After receiving these details an log entry will be made into the Namenode ledger to store details for crash recovery.
4. For Each chunk, we will do the following two operation
    1. Make a log entry for this chunk with offsets, id and file it is associated with.
    2. Find the suitable Datanode replica set which will be ideal for storing the chunk based on the selection policy currently selected.
    3. Create a ticket for first datanode in the replica set so that client can directly connect to that datanode.
5. Now we will have all the details which we are going to send to client, but before that we will update the Namenode state and insert **file to chunk(id)** mapping along with **chunk to boundary** mappings so that we can provide file details during fetch operation.
6. Namenode reply the request with the list of chunks each item containing chunk_id, offsets (based on file), replica set (Datanode gRPC address), ticket for first datanode in each replica set.

The default replication factor for the system is 3 so the size of replica set will be at max 3.

In **Datanode View** multiple tasks are done in order to complete the operation. Datanode doesn't have the view of the complete file, it only deals with chunks, client and Datanode uses both gRPC and TCP connection to communicate, Datanode can store chunk in two cases first is when it receives request directly from client and second is when it receives data through pipeline. In the below discussion we are taking a case in which replica set for chunk is two so two Datanode need to create pipeline and pass the data to next Datanode in replica set.

1. Datanode receives a store_chunk gRPC request from the client which will contain the chunk_id and replica set, first element in the replica set will be receiver itself, client will only make this request to first Datanode in replica set.
2. After receiving the request from the client, Datanode will check if there are any other replicas in the replica set provided, If replica set only contain the receiver itself Datanode just return the TCP listener address to Client. If the replica set contain more than one Datanode address following steps will be taken to handle gRPC request.
3. Current Datanode will fetch the next replica from the list and send a ticket generation message to the namenode namenode will send the ticket in response.
4. Next Datanode will send create_pipeline request to it passing pending replica set (by removing itself) and server side ticket. Create_pipeline request is similar to the store_chunk request this is handled in same way:
    1. If replica set size is 1 i.e. no other replicas just return the TCP Listener address as response of gRPC.
    2. If there are other replicas create pipeline request will be send it to next one, after fetching ticket from the namenode which will send to next one and so on recursively.
    3. Response of create_pipeline request is TCP Listener address in both case.
    4. After receiving the TCP address Datanode will try to connect to the TCP listener and store that in chunk to pipeline map in Datanode state.
4. Irrespective to replica set the response to create_pipeline/store_chunk request is TCP Listener address of current Datanode.
5. After serving gRPC request Datanode will receive a TCP Stream connection request from client which it will handle, This communication works on simple protocol, client will send chunk_id, chunk_size, mode and ticket as data packet.
6. Datanode will check the ticket validity using its secret key and check if the ticket is for the operation client wants to execute.

7. Mode will be write in current case, after that client will pass the chunk content to the Datanode.



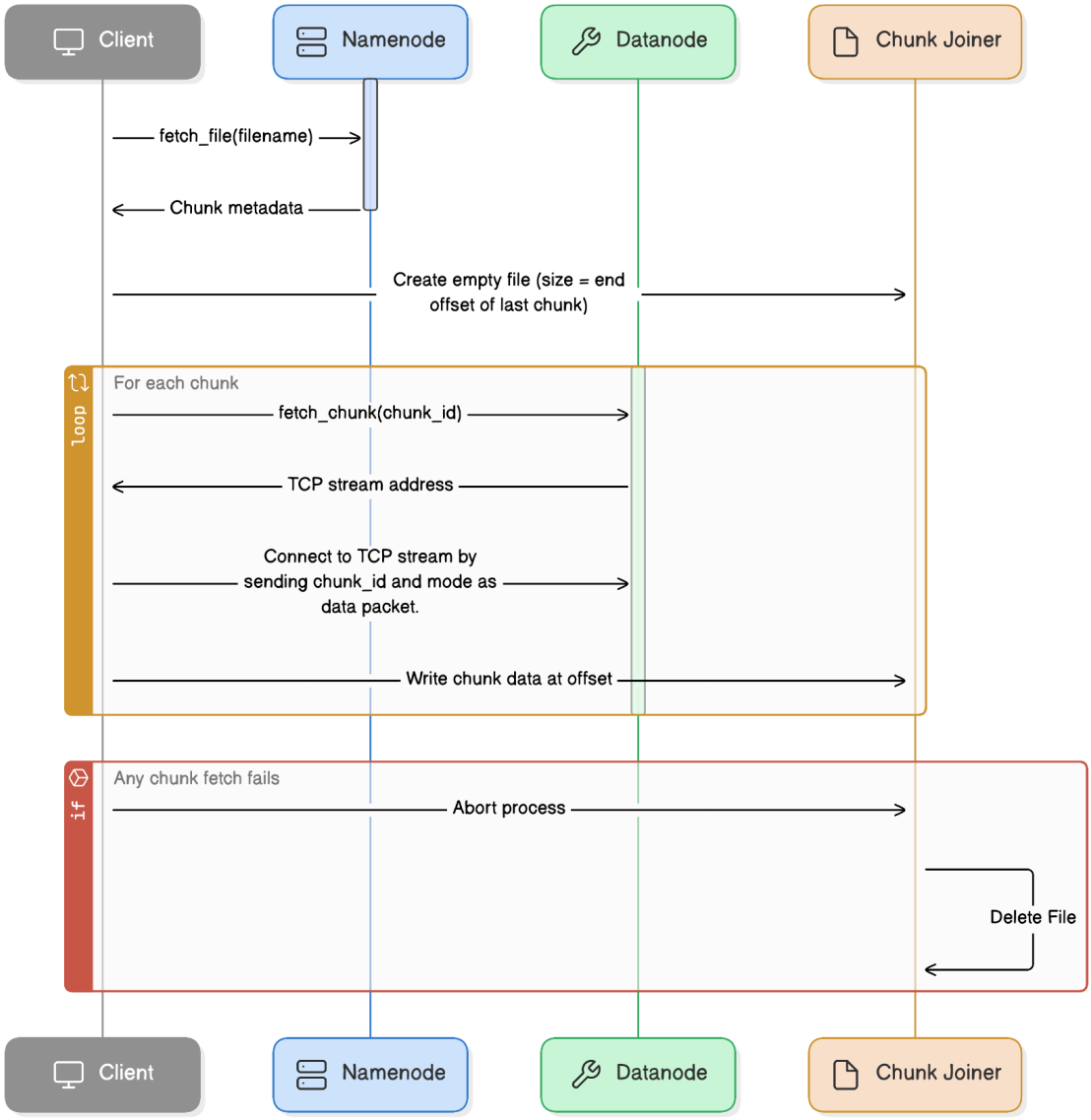
Datanode view of store file operation

8. After reading mode=write Datanode will check if there is any pipeline present for the chunk_id we read before, if pipeline is present Datanode will write the suitable headers along with the ticket to the pipeline.
9. Next it will pass the TCP stream to the stream_tee which will return two duplex stream one of which will be written to file and other will be passed to pipeline.
10. If there is no pipeline we will directly copy TCP stream to the file.
11. After reading chunk_size bytes from the stream Datanode will write the received byte to Stream as data packet which will work as acknowledgement to the client.

Fetch File
In this operation, client request the stored file from the system. This operation is simpler than the Store file operation since it involves only single Datanode per chunk.

In **Client view** following operation are needed to complete this task:

1. This Task start with the fetch_file gRPC call from client to Namenode, in this gRPC call client send filename to Namenode and in response Namenode will return chunk metadata and replica set for each chunk (size of this replica set will be always one).
2. Client will create an empty file sized equal to the end offset of the last chunk.



Client View of fetch file operation

3. After receiving the chunk metadata client will do following operation for all chunks
    1. Send fetch_chunk gRPC call to Datanode in replica set, this request contain the chunk_id client want to fetch. In response to this request Datanode will send the TCP stream address from which this chunk can be fetched.
    2. After receiving the TCP stream address, client will connect to that TCP stream, and write chunk_id to stream.

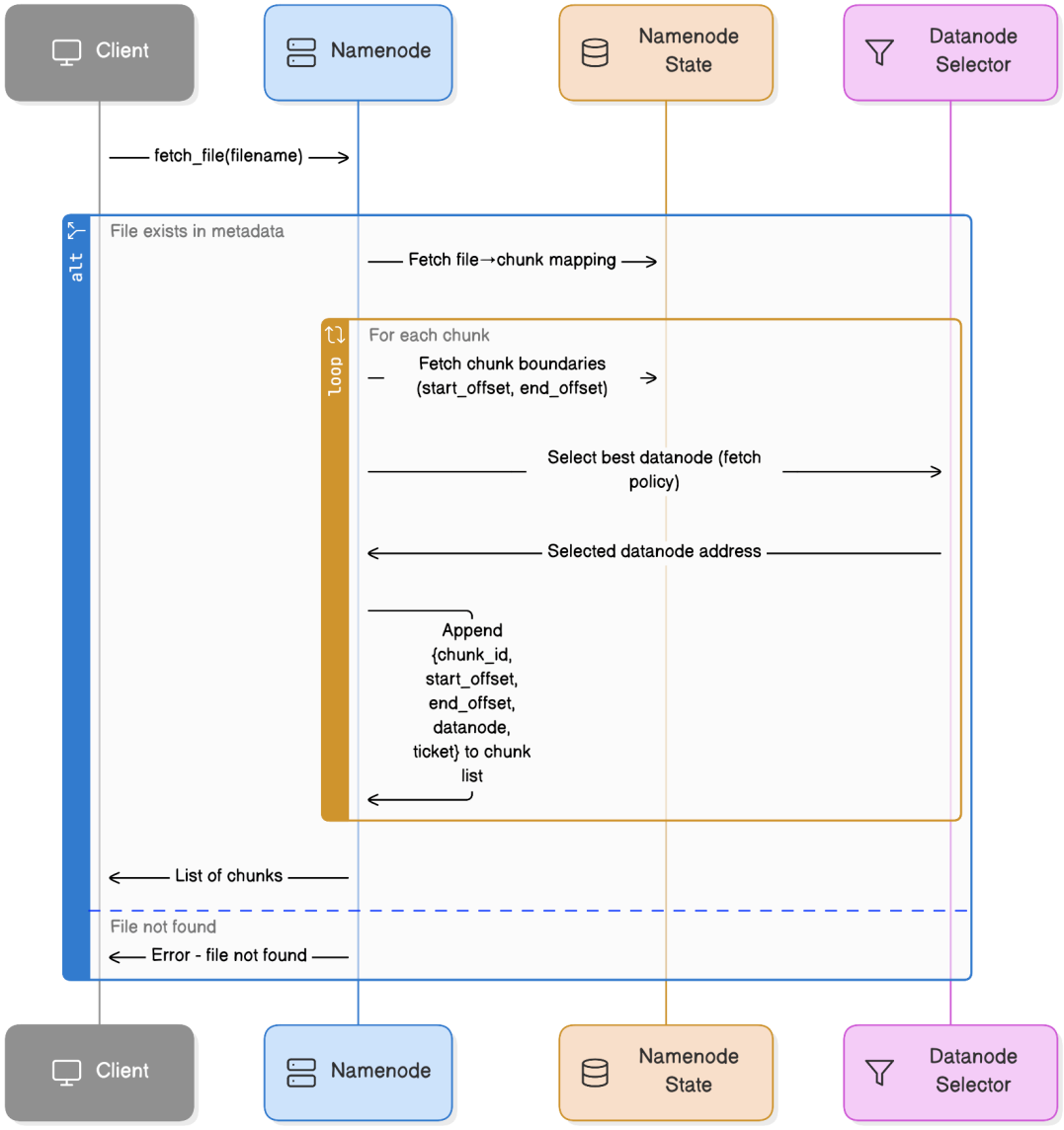
3. After writing chunk_id next it will write the mode to the stream which will be read to the TCP stream.
4. Now client will start reading from this TCP stream and will read until EOF, basically the stream will be written to the created file at specified offset.

4. When this operations is finished we will have full file copied to our local device from Whispering Woods.
5. In case error happen while handling any chunk all this process will be aborted and file will be delete.

Client View of fetch file operation.

In **Namenode View** following operations are required to complete this task

1. Namenode will receive request from the client containing the filename client want to fetch.
2. First Namenode check if the details of file present in its metadata i.e. how file is divided into the chunks.



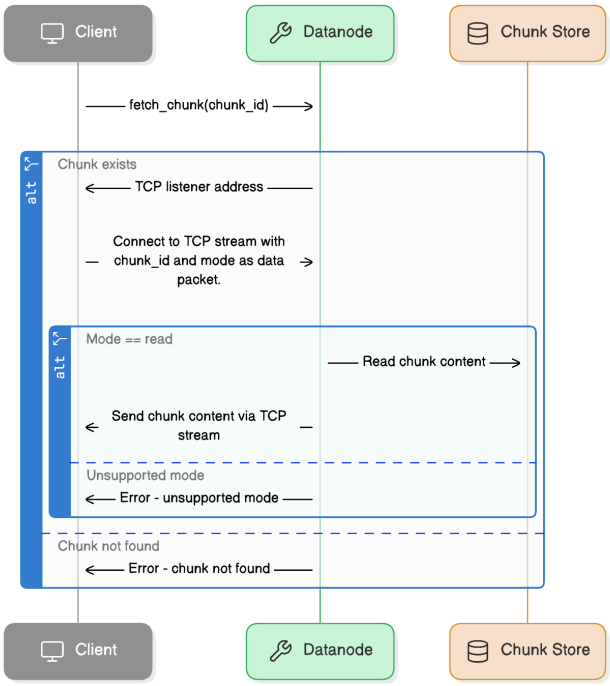
Namenode View of fetch file operation

3. After fetching all chunks related to the file following operations will be done for each chunks.
    1. Fetch the chunk boundaries for current chunk.
    2. From Datanode selection policy fetch the Datanode which will be ideal (According to policy) for client to fetch chunk from.
    3. Merge all these details (chunk_id, boundaries, serving Datanode) into a list.
4. After above operation Namenode will have list of chunks , with each items containing the chunk_id, start_offset, end_offset, locations (Datanode selected).
5. This list is then sent to client as a response of store_file message.

In **Datanode View** following operations are needed to be complete in order to fetch file:

Datanode deals with chunks of file, they don't have a whole view of the system. While fetching the file Datanode is requested for chunk of file, This operation start with simple gRPC request to chunk and end with the chunk content transfer using TCP connection. Below are steps Datanode takes to serve this request:

1. Datanode receives fetch_chunk gRPC request from the client, this request contain only chunk_id client wants.
2. To reply this request Client first checks if the chunk mentioned in request available or not. If chunk is not available an error response will be sent to the client.



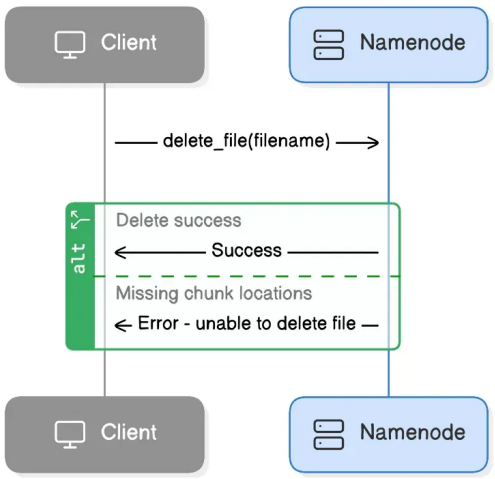
Datanode view of fetch chunk operation

3. If chunk is present the response will be the TCP listener address which can serve client with chunk requests.
4. After receiving success response client will initiate a TCP connection request to the Datanode (address returned from gRPC call).
5. After establishing request client will send chunk_id and mode read to the that TCP stream.
6. Datanode will read this data and after finding the mode is read, Datanode will push chunk content to the TCP stream.
7. Client can read this stream to get the contents of the chunks.

Delete File

This operation is comparatively simple than other, This operation starts with the simple gRPC request from the Client to Namenode. In this operation client doesn't connect to Datanode. This operation is handled in lazy way namenode will delete the file virtually first and then use state sync messages to delete chunks by marking tombstone in its state. This operation is explained in details in view of different component below:

In **Client View** , this operation consists of a single step: sending a delete_file gRPC request to the Namenode. In response to this gRPC request client receives response, In case Namenode finds any chunk for which it can't get locations client will receive error.



Client view of Delete File operation

In **Namenode View** this operation starts with gRPC request from the client. Actually we can safely say that this task is delegated to Namenode Since it is responsibility of the Namenode to delete all the chunks of the file. To do so Namenode take following steps:

1. After receiving the gRPC request from the client, Namenode first Add this operation to ledger.
2. Then Namenode check if the file is present or not if the file is not present it will just send the success response to the Client since there is no need to delete anything.
3. If file is present than Namenode will find the chunks associated with file.
4. For each chunk in the list following tasks are needed in order to delete them in background
    1. Make entry in ledger mentioning that this chunk is deleted.
    2. Update the namenode state and mark each chunk of this file as deleted.
5. Send success message to client as acknowledgement for file deletion.

Now namenode will have all the chunk details which have been deleted, Now for each state sync message from the datanode namenode will do following steps:

1. After receiving the receiving the state sync message namenode fetch the available chunk list from the request.
2. Then Namenode will filter all the available chunks which have been marked deleted.
3. Namenode will send this list to the datanode as a reply so that datanode can delete these chunks.
4. Tombstone will be deleted after some interval (default to 23 secs , approx. 4 state sync intervals).

In **Datanode view** Delete operation applies to chunk Datanode receive the reply of the state sync message from the Namenode following steps need to be taken in order to complete this operation, these steps are fire and forget datanode just try to delete, if some error happen this whole cycle will be repeated during the next state sync :

1. After receiving the list from Namenode Datanode update its to_be_deleted chunk list.
2. During the datanode state maintenance cycle this list is replace with empty list.
3. Store will get request to delete each of this request.
4. Store will delete these chunks , if error happened to_be_deleted list will again be updated during state sync with namenode.

Above Notes Explain all the task user can perform with this system. These are tasks which are initiated by the external entity (user). Apart from these task there other task which are performed by system in order to do health checks, state sync etc. In the next section we will discuss all these tasks, how these tasks performed and what are there significance in brief.

Failure Detection and recovery

Namenode Failure and recovery

In Whispering Woods, the Namenode is the most critical component of the system. It acts as the *control plane* — maintaining global metadata, managing chunk placements, issuing tickets for client-datanode interactions, Acting as KDC (key distribution centre) and serving as the system's Certificate Authority (CA) for secure communication between nodes.

Since the Namenode is responsible for maintaining all file-to-chunk mappings, chunk locations, and security credentials, it naturally becomes a single point of failure in the architecture. To mitigate this, Whispering Woods implements a log-based recovery mechanism that ensures durability and consistent state reconstruction after restarts or unexpected crashes. Every critical operation that mutates the system state — such as File creation and deletion, Chunk allocation and replication, key generation, root cert is persisted to an on-disk log.

The Namenode maintains the ledger for these logs and root cert is stored in separate file.

Recovery Process

1. **Root Certificate Restoration:** The CA key-pair and root certificate are reloaded from persistent storage to ensure continued trust continuity across datanodes.
2. **State Reconstruction:** Operation logs are read sequentially to rebuild the complete in-memory metadata state — including file, chunk mappings, datanode registrations, and secret keys.
3. **Resumption of Service:** Once the state is fully rebuilt, the Namenode resumes normal operations without requiring any manual intervention.

Datanode Failure and recovery

In Whispering Woods, Datanodes form the data plane of the system. They are responsible for storing and serving file chunks and performing replication. meaning the system can tolerate their failure without metadata loss, thanks to redundancy and recovery protocols managed by the Namenode.

Failure Detection

The Namenode employs a **heartbeat-based failure detection mechanism** to continuously monitor the health of all active Datanodes.

- Each Datanode periodically sends **heartbeat messages** to the Namenode.
- If a Datanode fails to report within a configured timeout interval (6 sec), the Namenode marks it as **unreachable**.
- Location of chunks will be updated the this node will be removed from them.

Impact of Failure

When a Datanode fails, the system might temporarily lose access to the chunks stored on that node. However, the **replication strategy** ensures that multiple replicas of each chunk exist across other Datanodes. The failure of one node therefore does not result in data loss — only a temporary reduction in replication factor or read throughput.

Recovery and Re-replication

1. **Metadata Update:** The Namenode updates its in-memory metadata to mark the node as inactive and removes its chunk references from the active replica set.
2. **Re-replication :** The Namenode identifies all chunks whose replication factor has dropped below the desired threshold and sends replication message across healthy Datanodes.
3. **Data Reconstruction:** Healthy Datanodes holding existing replicas of those chunks participate in reconstructing and transmitting data to new targets, ensuring the system returns to the configured redundancy level.

Node Recovery and Rejoining

When a failed Datanode comes back online:

1. It re-registers with the Namenode, presenting its node ID, issued certificate, and storage manifest.
2. The Namenode performs a **state reconciliation**, comparing the Datanode's reported chunks with its authoritative metadata.
3. Any outdated or redundant chunks on the Datanode are deleted, while missing or required chunks are scheduled for replication onto it.
4. Once synchronisation is complete, the node is marked **healthy** and reintegrated into the active cluster.

State Maintenance

State Maintenance in Datanode
State maintenance in the Datanode is an internal background operation responsible for keeping the node's local view of storage consistent and up to date.
Every 5 seconds, the Datanode queries its chunk store to collect information about available storage capacity, list of chunks currently stored.

This information forms the Datanode's local state, which is used to send precise state updates. Regular state maintenance ensures that the Datanode always reports an accurate snapshot of its health and storage condition to the Namenode.

State Maintenance in Namenode
State maintenance in the Namenode is responsible for tracking the overall health and connectivity of the cluster. Each time the Namenode receives a state sync and heartbeat from a Datanode, it updates the corresponding last-seen timestamp for that node in its in-memory state.

A periodic background task (executed every few seconds) scans these timestamps to identify stale entries—Datanodes that haven't sent a heartbeat within specified interval. Any such nodes are marked as inactive and removed from the active Datanode list.

This mechanism allows the Namenode to maintain a real-time, accurate view of the system's topology. It ensures that all scheduling, replication, and data retrieval decisions are made based on the current health of available Datanodes.

Authentication and Authorisation

Security in Whispering Woods is designed around a layered model inspired by Kerberos, combining certificate-based authentication for cluster entities with ticket-based authorisation for controlled data access.
The system ensures that only authenticated and authorised nodes or clients can perform operations, without requiring direct credential exchange between all components.

Authentication

Whispering Woods supports multiple authentication mechanisms through a pluggable module, enabling flexibility across development and production environments.

NoAuth Mode

- Used only for local testing or standalone setups.
- Disables all authentication and authorisation checks.
- Should **never** be used in distributed or production deployments.

JWT-Based Authentication (Client-Level)

- Used primarily for client access to Namenode APIs.
- Clients authenticate using admin credentials (username + password) and receive a JWT.
- The JWT is attached to subsequent API requests to prove the client's identity and privileges.
- This mechanism is stateless and lightweight, ideal for user-facing REST endpoints.

Certificate-Based Authentication (Node-Level)

- Used for Datanode ↔ Namenode and Client ↔ Namenode communication.
- The Namenode acts as a Certificate Authority (CA):
  - Admin clients can request the Namenode to generate a certificate and secret key for a Datanode.
  - The issued certificate and key are added to the Datanode configuration during startup.
- Every request to the Namenode must include a valid signed certificate.
- The Namenode validates the certificate's signature, expiration, and identity before processing any operation.

This guarantees that only trusted and registered components can interact with the control plane.

Authorisation (Kerberos-Inspired Ticket System)
Whispering Woods implements a ticket-based authorisation protocol modelled after Kerberos, but optimised for distributed file operations.

Ticket Generation

1. When a client requests to perform an operation (e.g., store, fetch, delete), it first authenticates with the Namenode using its credentials or certificate.
2. The Namenode then issues a ticket, which contains Chunk Id, Target Datanode Id, Operation Type (read, write, replicate, delete), Expiry Timestamp.
3. The ticket is encrypted in two layers:
    - The outer layer is encrypted with the client's secret key, ensuring only the client can read it.
    - The inner server portion is encrypted with the Datanode's secret key, containing the server key and authorisation data.

This structure ensures that:

- The client cannot tamper with the server's portion.
- The Datanode can only decrypt its own portion, verifying the Namenode's signature and server key.

Ticket Usage

1. The client, after receiving the ticket from the Namenode, sends it to the target Datanode as part of the request.
2. The Datanode decrypts its portion using its secret key, retrieves the server key, and validates the Namenode's signature.
3. Once validated, the Datanode accepts the operation — ensuring both the client and Datanode trust the ticket issued by the Namenode.

This process removes the need for direct trust or shared secrets between clients and Datanodes, while ensuring that all access remains securely mediated through the Namenode.

Whispering Woods Dashboard — Overview

The Whispering Woods Dashboard provides a real-time visualisation of the distributed file system's internal state, helping administrators monitor the health and performance of the cluster. It serves as the monitoring and observability interface for the Whispering Woods Distributed File System, powered by the NameNode's */monitoring/snapshot* API.

Authentication & Access Control

Before accessing the dashboard, users must log in using valid credentials (username and password). The login request is sent to the NameNode's authentication service (/auth/login), which returns a JWT token if credentials are valid.

Once authenticated:

- The JWT is stored in memory and automatically attached to every */monitoring/snapshot* request.
- The token uses the JwtTokenAuth authentication mechanism defined in the Whispering Woods authentication module.
- Unauthorised users are denied access to the system state.

Dashboard Functionality

After successful login, the dashboard unlocks the system overview and starts periodic updates (every 5 seconds) to fetch the latest cluster state from the NameNode.

The main sections of the dashboard include:

1. Cluster Overview
   - Total Nodes: Count of all registered DataNodes.
   - Active Nodes: Nodes currently reachable (sending regular heartbeats).
   - Inactive Nodes: Nodes that missed heartbeats beyond the allowed interval (considered stale or failed).
   - Total Storage Remaining (MB): Aggregated available space across all DataNodes, dynamically computed from storage_remaining fields.
2. Files Stored
   - File Name
   - File Size (bytes): Computed dynamically by summing chunk sizes.
   - Chunks: The IDs of all chunks that make up the file.
3. Chunk Details
   - Chunk ID
   - Start Offset / End Offset
   - Size (bytes)
   - State: (e.g., active, under replication, or unavailable)
   - Locations: List of DataNodes currently storing that chunk.

The dashboard auto-refreshes every 5 seconds, ensuring administrators always see the latest cluster state without manual reloads.

Future Enhancements

1. **Advanced User Management System**
   Extend the current password-based authentication to a full-fledged user management module with role-based access control, user registration, password reset, and audit logging.

2. **Graphical User Interface (GUI)**
   Develop a web-based dashboard for cluster management, file operations, certificate handling, and real-time monitoring instead of relying solely on the CLI.

3. **Load-Adaptive Datanode Allocation and Balancing**
   Implement intelligent datanode selection policies that adapt based on node load, network latency, and storage utilisation to improve efficiency and fault tolerance.

4. **Enhanced Monitoring and Observability Tools**
   Integrate Prometheus and Grafana for real-time metrics, add anomaly detection, alerting mechanisms, and distributed tracing for better visibility into system performance.

5. **DNS-Based Service Discovery**
   Introduce a DNS or service registry mechanism to replace static configuration, allowing dynamic node registration, automatic failover, and simplified address management.

6. **Testing and Benchmarking Framework**
   Create an automated framework for integration, stress, and fault-tolerance testing, including benchmarking tools to measure throughput, latency, and recovery performance.