pShare Writing 4

Team Taco Bell: William Mai, James Xu, Joshua Sherwood, Mazen Saadi

Abstract

Nowadays, people often use cloud storage to store all kinds of information or files, but there is an increase in data breaching in cloud storage. In order for people to safely store their private information, we developed an encrypted distributed storage system that allows users to use their spare computers as storage, which is connected by a P2P network to their commonly used devices.

Table of Contents

User Stories

Office Administrator	**System Setup & Configuration** ● *Designating Nodes:* ○ As an Office Administrator, **I would like to** set up the system by designating a registry node and connecting storage nodes **so that** our office's unused computers can be repurposed for data storage. ● *Encryption Settings:* ○ As an Office Administrator, **I would like to** configure encryption settings (e.g., setting default passwords or generating keys) so that employees have a secure and consistent way to interact with the system. **Monitoring & Maintenance** ● *Monitoring Node Status:* ○ As an Office Administrator, **I would like to** monitor the status of connected storage nodes (e.g., storage capacity, uptime) **so that** I can proactively address potential failures. ● *Ensuring Redundancy:* ○ As an Office Administrator, **I would like to** ensure that chunks of files are evenly distributed across storage nodes **so that** the system maintains redundancy and efficiency. ● *Data Recovery:* ○ As an Office Administrator, **I would like to**

Office Administrator (cont.)	recover data from the system even in the event of a node failure **so that** no information is permanently lost. **Security & Access Control** ● *Managing User Permissions:* ○ As an Office Administrator, **I would like** to manage user access and permissions **so that** sensitive files are only accessible to authorized personnel.

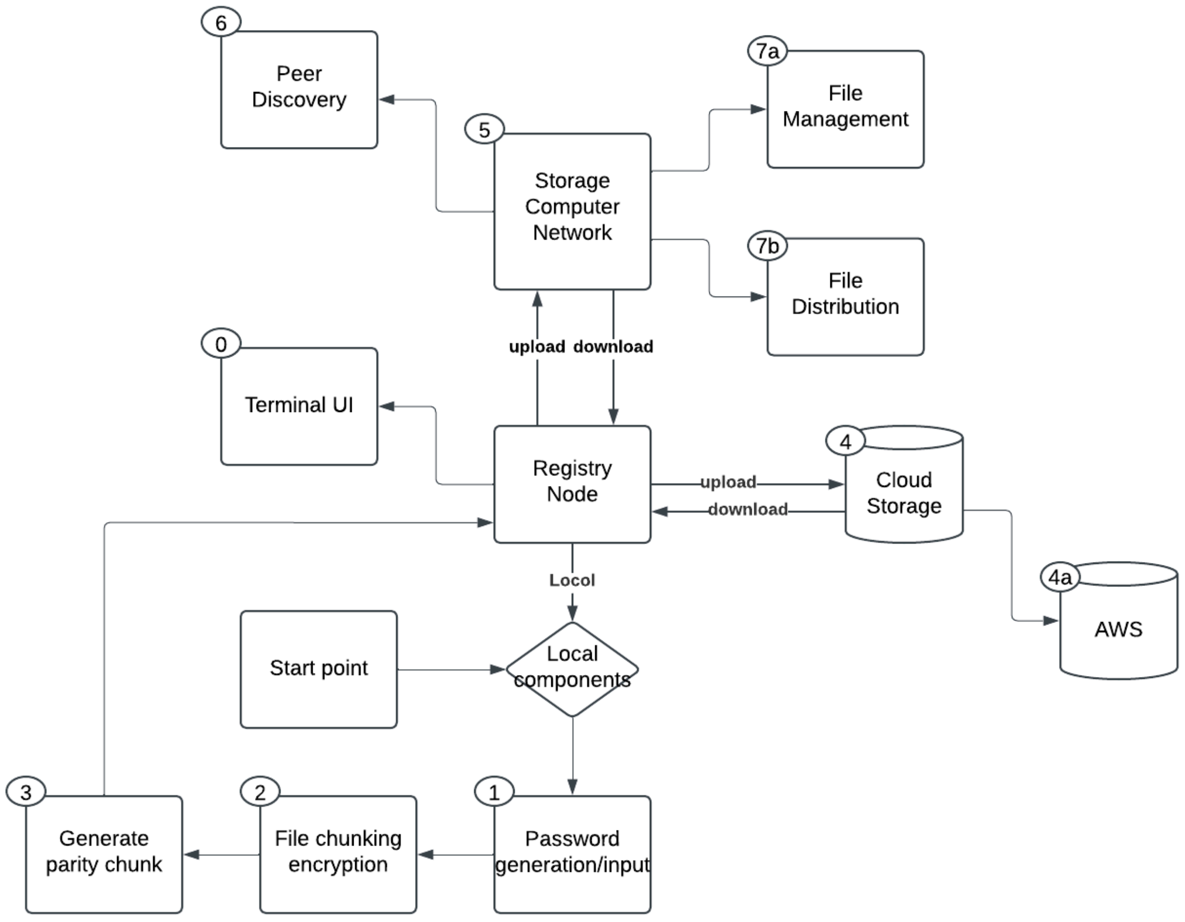
Office Employee	**File Management** ● *Secure File Upload:* ○ As an Office Employee, **I would like to** securely upload files to the system using an intuitive interface **so that** I can focus on my tasks without needing extensive training. ● *File Retrieval:* ○ As an Office Employee, **I would like to** retrieve stored files using my password or authentication **so that** I can access my work securely and conveniently. ● *Search & Filter Files:* ○ As an Office Employee, **I would like to** search and filter stored files using metadata (e.g., file name, upload date) so that I can quickly locate what I need. **Security & Integrity** ● *Checksum Verification:* ○ As an Office Employee, **I would like to** verify the integrity of retrieved files using a checksum **so that** I can be confident the data hasn't been altered or corrupted. ● *File Integrity Assurance:* ○ As an Office Employee, **I would like to** verify the integrity of retrieved files **so that** I can be confident the data hasn't been altered or corrupted.

Office Employee (cont.)	**System Feedback** ● *Upload Notifications:* ○ As an Office Employee, **I would like to** receive notifications when a file I uploaded has been successfully encrypted and distributed **so that** I can confirm its security. ● *Storage Status View:* ○ As an Office Employee, **I would like to** view the storage status of my uploaded files (e.g., which nodes hold chunks) **so that** I can track the system's reliability.
General Home User **General Home User (cont.)**	**File Management** ● *Personal File Upload:* ○ As a Home User, **I would like to** upload my personal files (e.g., photos, tax documents) to the system **so that** I can securely store them without relying on third-party services. ● *File Retrieval:* ○ As a Home User, **I would like to** retrieve files from the system using a password **so that** I can avoid managing multiple passwords for each file. ● *Automatic Backups:* ○ As a Home User, **I would like to** set up automatic backups of files from specific folders **so that** my important data is consistently protected.

	Security & Redundancy ● *File Encryption & Distribution:* ○ As a Home User, **I would like to** encrypt and distribute my files across multiple devices in my network **so that** I can protect my data even if one device fails. **System Monitoring** ● *Node Management Dashboard:* ○ As a Home User, **I would like to** manage my storage nodes (e.g., old laptops or desktops) from a single dashboard **so that** I can keep track of my storage setup. ● *Storage Node Status:* ○ As a Home User, **I would like to** view the status of connected storage nodes (e.g., online/offline, storage usage) **so that** I can ensure my data is secure and accessible.

Flow Diagrams

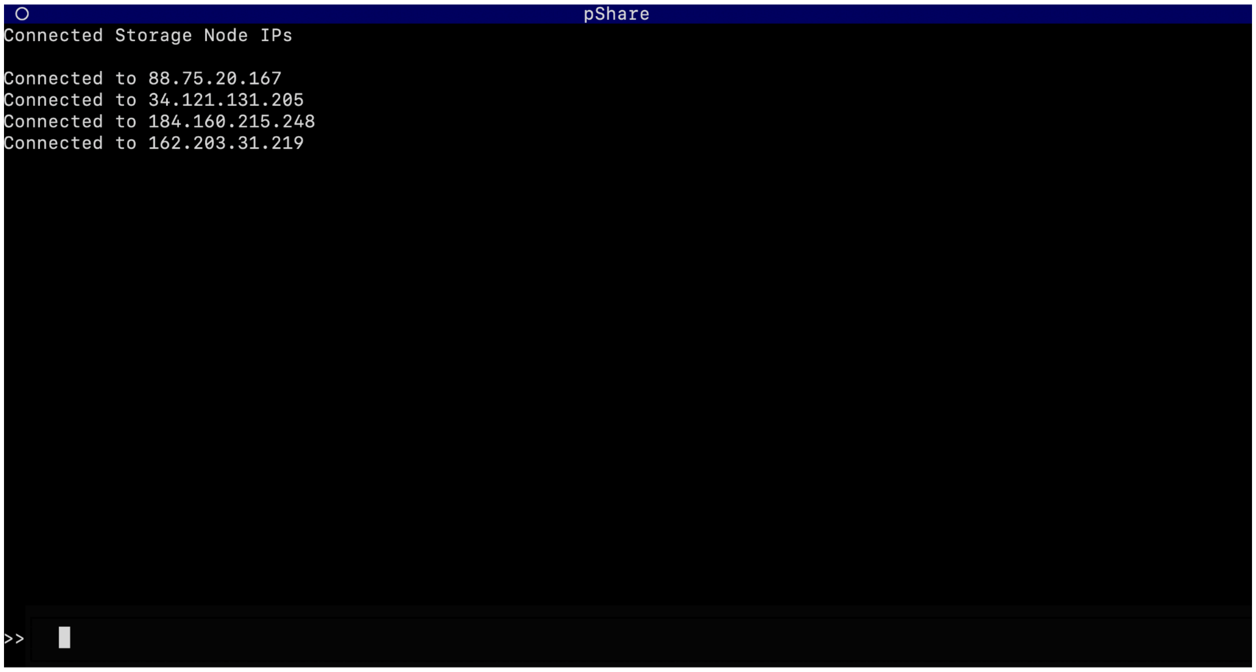
The following is a diagram of how the system will interact with itself including what steps it takes to execute a specific action.



User enters command to interact with our architecture (0), then enters the password when encrypting a file (1), which is then encrypted (2) and a parity chunk is added (3) and the result is sent to the registry node, which optionally uploads the encrypted file to cloud storage (4). The only option we currently support is AWS (4a). The registry node can access the file by connecting to the storage computer network (5), which connects to the registry node through peer discovery (6). The storage computer network has file management (7a) and file distribution (7b) practices and algorithms. `Local components` are represented by a diamond as it is a **gateway entity** between the registry node and the password generation. Others are simple entities and thus, following the ER diagram best practices, are represented by rectangles. Cloud storage and AWS are represented by cylinders, as is customary with representing cloud entities.

Mockups

The following is our UI mockup. It has a terminal for user input, and the user can type commands such as download, upload, list nodes <file> (to see the breakdown of the file across the nodes), etc. It lists active connections and their status, and stored files, two crucial components the user should be aware of.

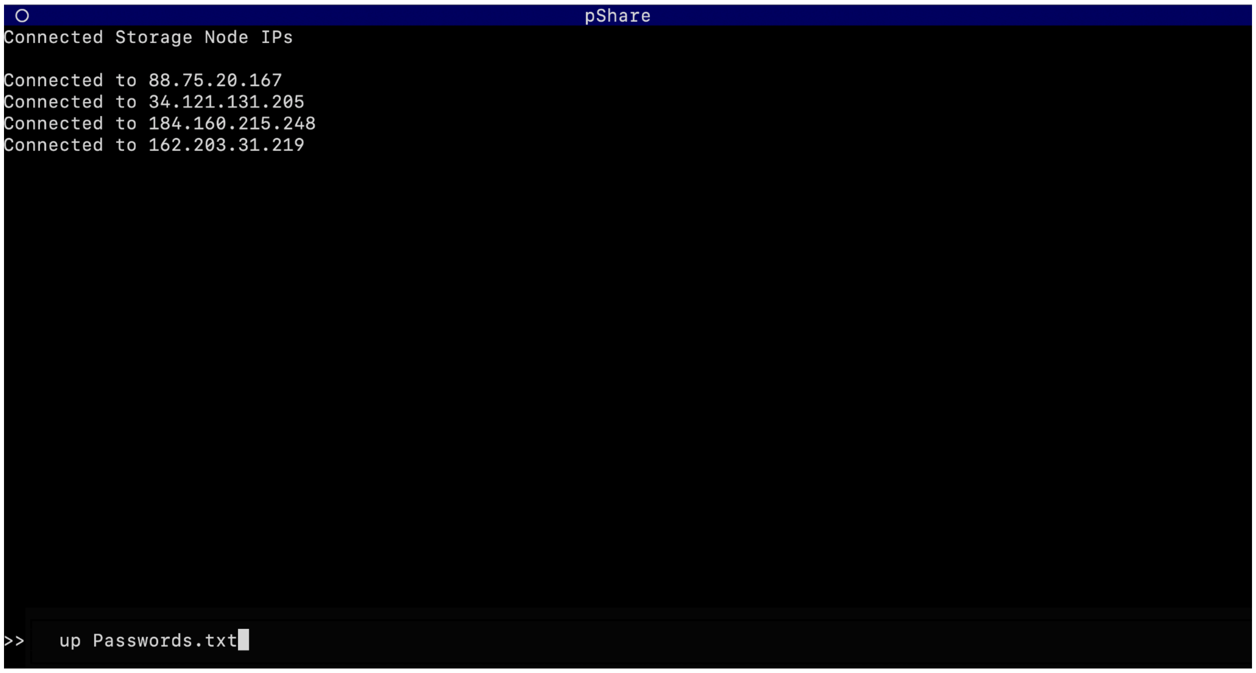


```
Connected Storage Node IPs

Connected to 88.75.20.167
Connected to 34.121.131.205
Connected to 184.160.215.248
Connected to 162.203.31.219

>>
```

Figure 1: Base terminal

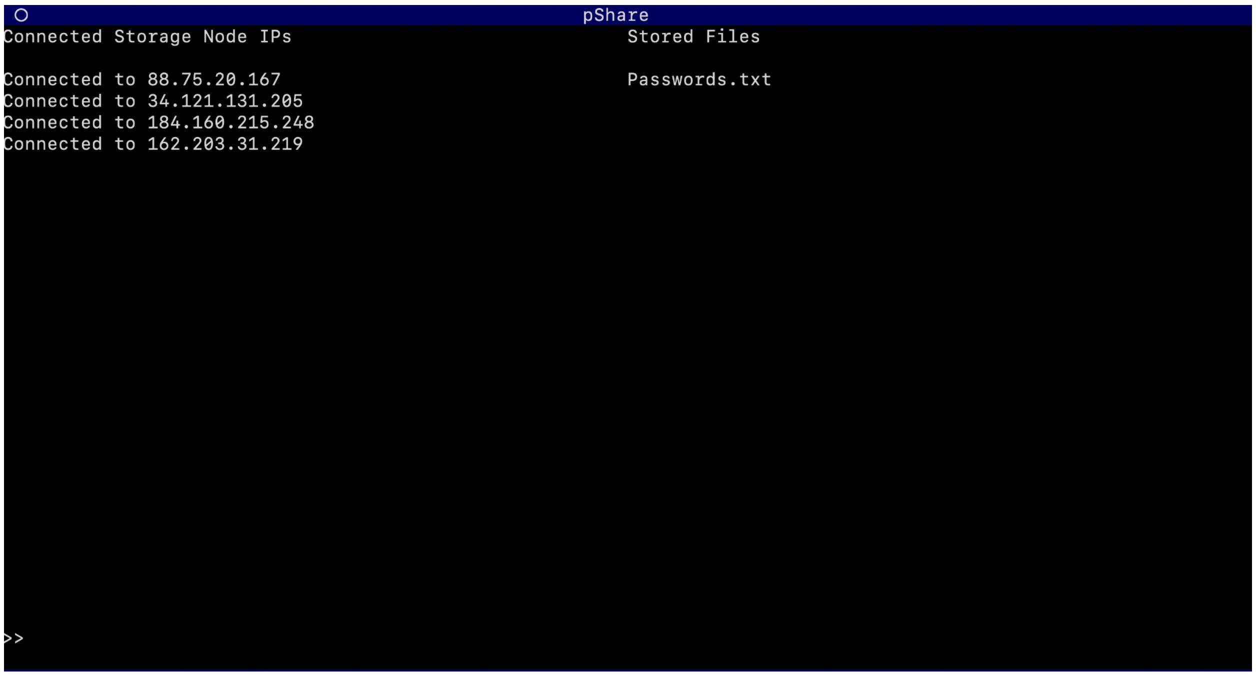


```
pShare
Connected Storage Node IPs

Connected to 88.75.20.167
Connected to 34.121.131.205
Connected to 184.160.215.248
Connected to 162.203.31.219

>>   up Passwords.txt
```

Figure 2: Uploading a file

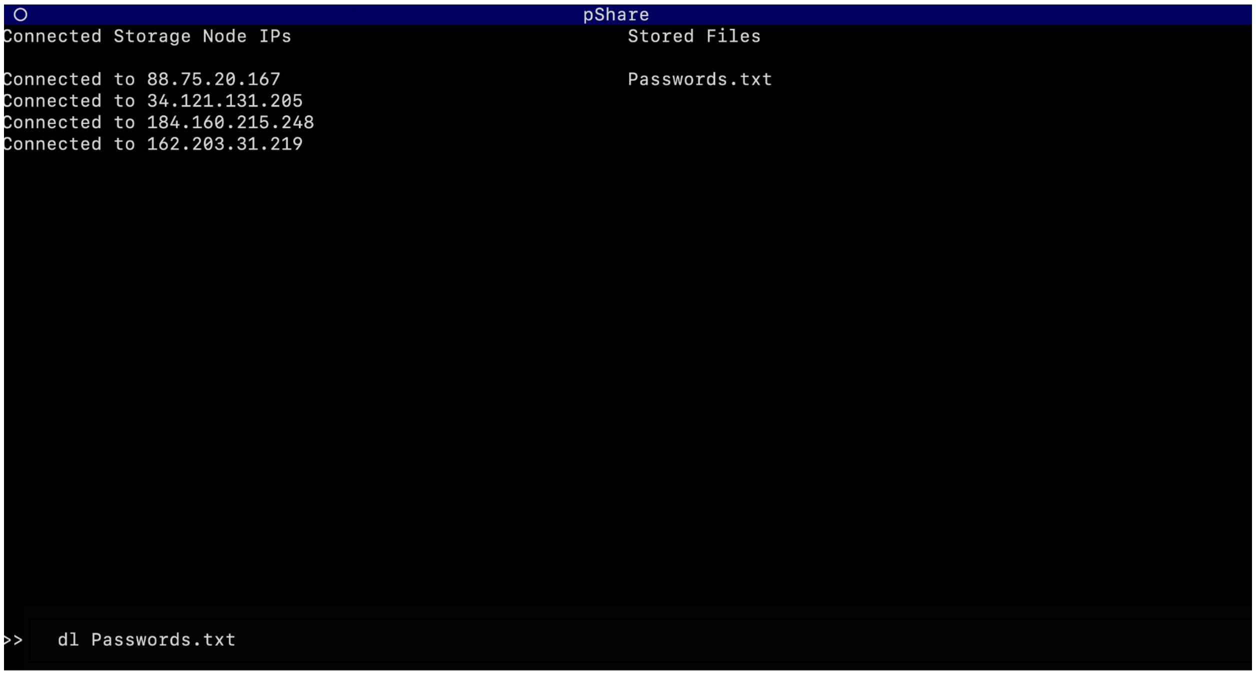


```
pShare
Connected Storage Node IPs                              Stored Files

Connected to 88.75.20.167                               Passwords.txt
Connected to 34.121.131.205
Connected to 184.160.215.248
Connected to 162.203.31.219

>>
```

Figure 3: File uploaded



```
 O                                                pShare
Connected Storage Node IPs                            Stored Files

Connected to 88.75.20.167                             Passwords.txt
Connected to 34.121.131.205
Connected to 184.160.215.248
Connected to 162.203.31.219




>>   dl Passwords.txt
```

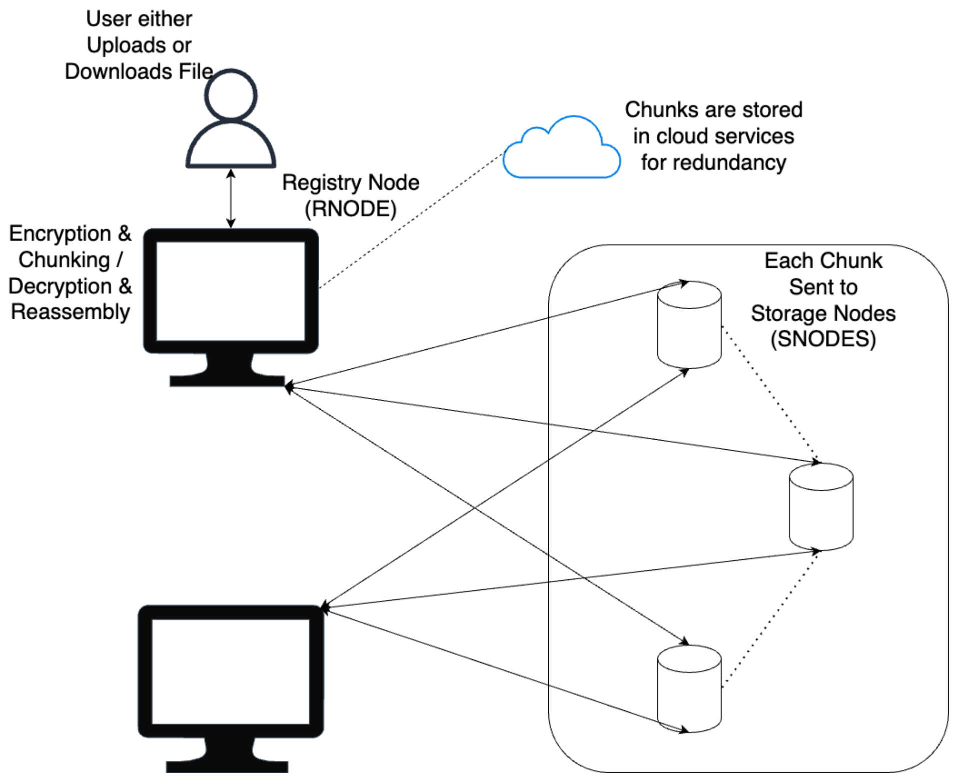
Figure 4: Downloading a file

**Will show up in the user's directory of choice (but currently in a designated directory)

<u>Technical Specifications</u>

1. User (Client):
    a. Interacts with the system to upload or download files through the Registry Node (RNODE).
2. Registry Node (RNODE):
    a. Role: Manages file encryption, chunking, and distribution to Storage Nodes (SNODES).
    b. Processes:
        i. Encryption: AES-GCM (using pycryptodome).
        ii. Chunking: Files are split into chunks using Python's concurrent.futures for parallel processing.

3. Storage Node (SNODE):
    a. Role: Store the encrypted file chunks.
    b. Technologies: Use TCP sockets for communication and UUIDs for node identification.
4. Data Flow:
    a. Upload: RNODE encrypts, chunks, and distributes the file to SNODES.
    b. Download: RNODE retrieves, decrypts, and reassembles the file for the User.
5. Below is an abstract, high level diagram of the system:


User either Uploads or Downloads File
Chunks are stored in cloud services for redundancy
Registry Node (RNODE)
Encryption & Chunking / Decryption & Reassembly
Each Chunk Sent to Storage Nodes (SNODES)

Local Components (Registry Node):

- **Terminal UI**
    - User-Friendly Interface: Ensure the terminal interface clearly displays available options and provides intuitive navigation, making it easy for users to understand what actions they can take and what files are already managed.
    - Status Overview: Present a concise summary of file statuses, including encryption, parity chunk generation, and upload/download progress, so users can quickly gauge system activity.
    - Real-Time Feedback: Integrate live updates, such as progress indicators for active processes, ensuring users are always informed about ongoing tasks.
- **File Processing**
    - Data Encryption: Secure files before storage with robust encryption methods to safeguard data integrity and privacy.
    - Parity Chunk Generation: Implement a reliable system for creating parity chunks to ensure data redundancy and recovery capability in case of corruption or loss.
- **Storage Interaction**
    - Integrated Framework: Develop a seamless upload/download framework to handle interactions with both peer-to-peer (P2P) and cloud storage systems, ensuring efficient and reliable file transfers.

Cloud Components:

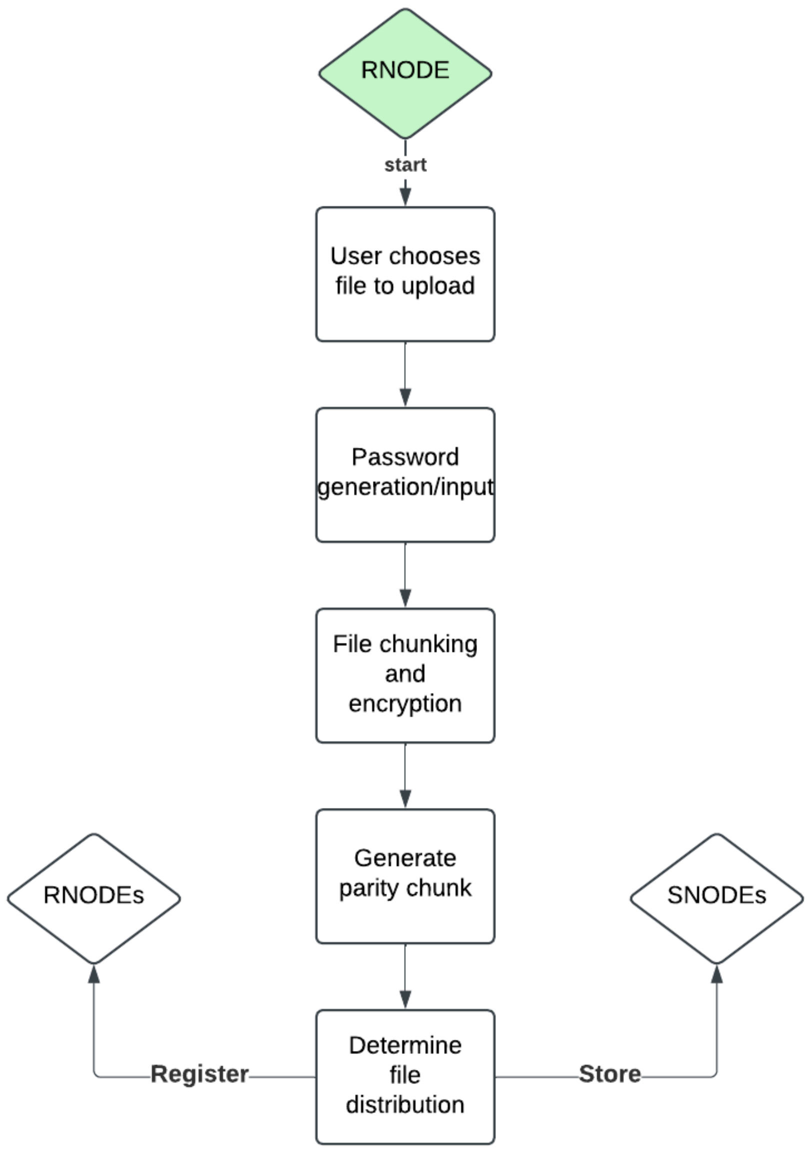
- AWS-S3 storage: Called by upload and download framework, transition call to amazon boto3 python API to achieve upload and download function

P2P Component:

- **File Management**
    - When receiving a file, the file will be stored in the storage directory setup when connection is established for the first time. A JSON file will store which file belongs to whom.
    - When the registry node wants to request for file, it will locate the corresponding chunks of the file, and request the storage nodes for the file
- **Peer Discovering**
    - A broadcast message will be sent via LAN by Registry Node and Storage Node will respond, and authenticate.
    - The Registry Node verify the authentication and establish connection if verification success
    - For WAN, a storage node will connect to the registry node via UDP hole punching. The implementation for this is still being worked on, but the following UUID interactions still hold.

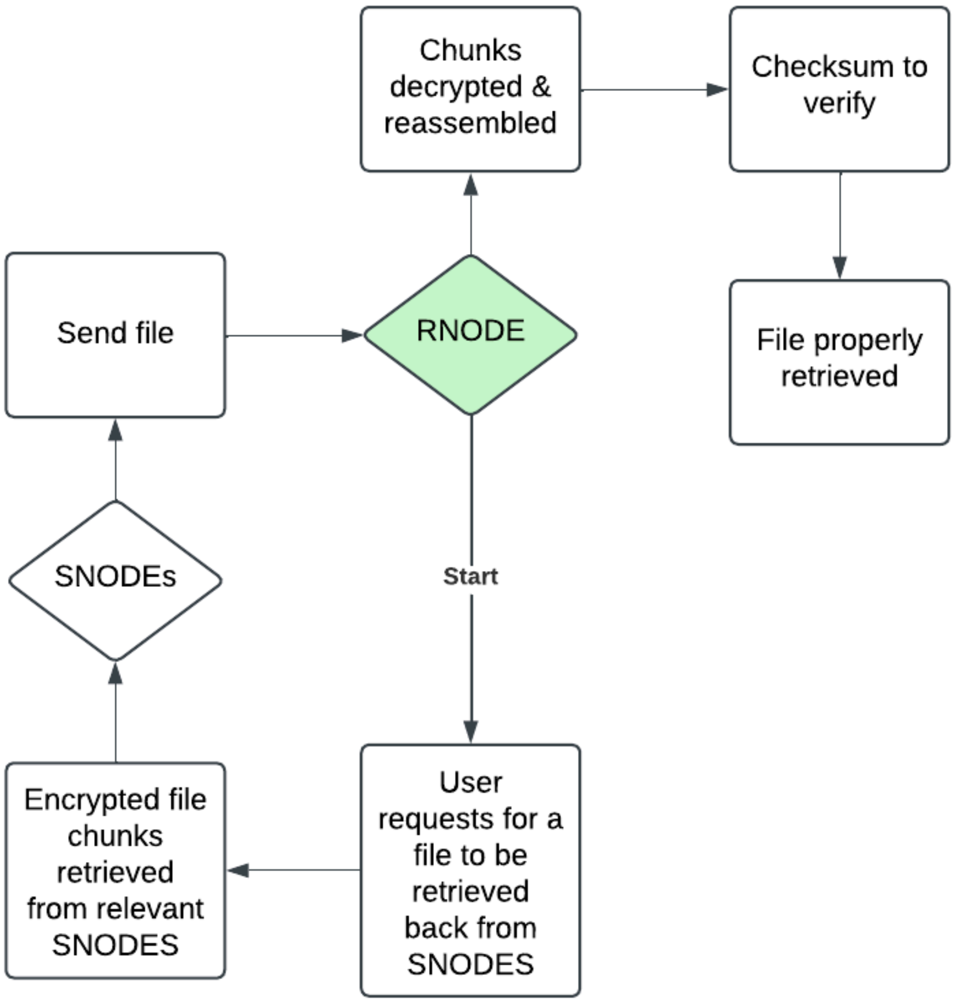
We will now describe the four primary functions the user calls when using our application. These involve the interaction of multiple components of our architecture.

<u>Scenario 1</u>: Upload a File


RNODE
start
User chooses file to upload
Password generation/input
File chunking and encryption
Generate parity chunk
RNODEs
SNODEs
Determine file distribution
Register
Store

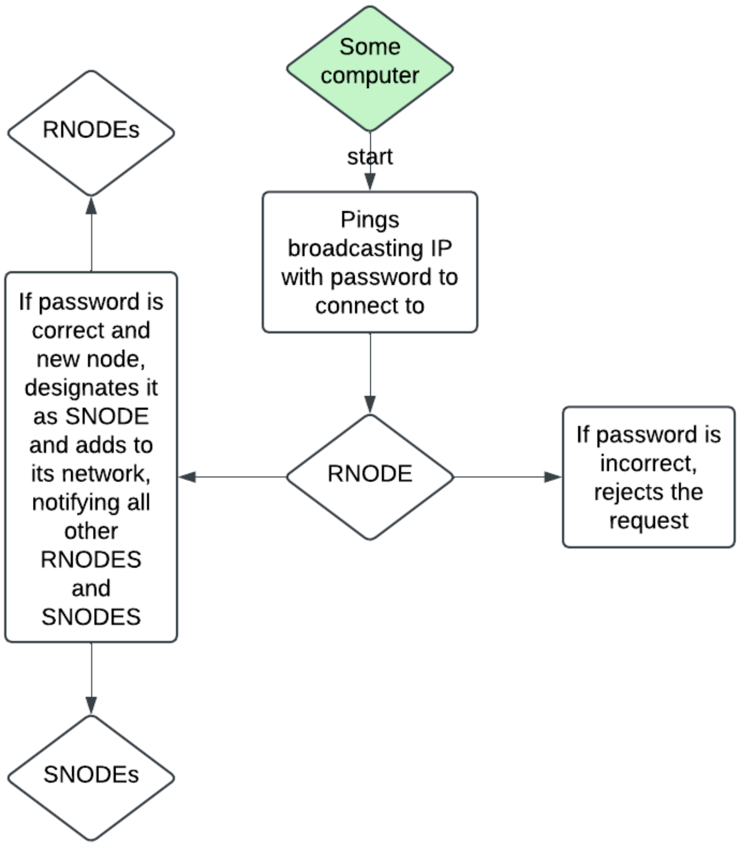
A user can choose to upload a file from the device to the network using our TUI. They select the file on their device that they want to upload. Our application generates a key per file by first generating a password, a random salt per file, then uses the pycryptodome library, scrypt function to transform hash of password and salt into a key. Each RNODE stores the salts specific to that device, and so by default is unable to decrypt files stored from other RNODEs.

Scenario 2: Download a File


Chunks decrypted & reassembled
Checksum to verify
Send file
RNODE
File properly retrieved
SNODEs
Start
Encrypted file chunks retrieved from relevant SNODES
User requests for a file to be retrieved back from SNODES

To download a file, the user types `dl filename` in our TUI's terminal interface. For example, to download `mymovie.mov`, the user would type `dl mymovie.mov`. All registry nodes have the location of each encrypted file chunk and the associated user, as well as which file chunks can be retrieved to form the full encrypted file. Due to our RAID-like implementation, there will be many options for retrieval, and the registry node determines which method is optimal. Once retrieved from the relevant SNODEs, each chunk is decrypted and then all are put together to form the full decrypted file. Then a checksum is performed, comparing the decrypted file's hash with the known hash stored in the registry node. Once the decrypted file is validated, it is copied to the path of the user's choice.

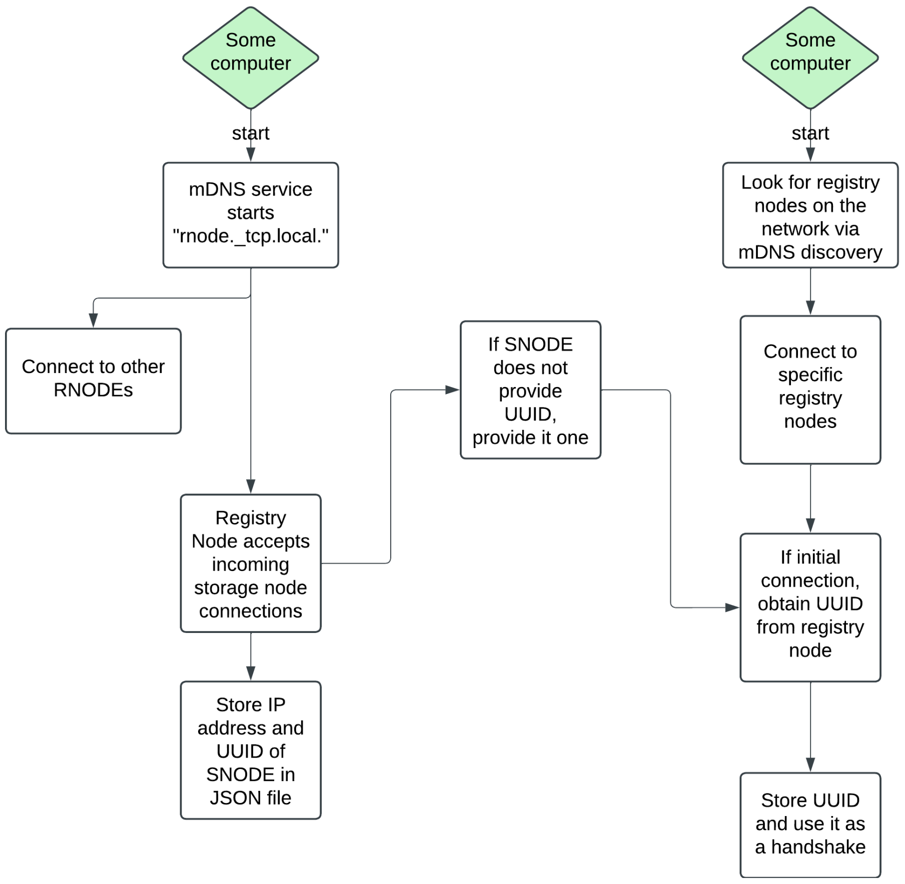
<u>Scenario 3</u>: Add SNODE or additional RNODE


Some computer
start
Pings broadcasting IP with password to connect to
RNODE
If password is incorrect, rejects the request
If password is correct and new node, designates it as SNODE and adds to its network, notifying all other RNODES and SNODES
RNODEs
SNODEs

RNODEs on a network routinely broadcast the local network so that new snodes can find them. This is the LAN approach, devices on different networks will likely need to know the RNODE's

IP, or connect to a central server that has the list of registry node IPS. To become an SNODE on a network, the computer must provide the broadcasting RNODE with the password to connect. Each RNODE has its own network password. For the WAN approach, this may be changed to make our application more user friendly. The first RNODE on a network must be added with the approach shown in Scenario 4, but additional RNODEs can be added in a similar way to SNODEs. The only difference is that the password will be different. Once the request by a computer to become an SNODE or additional RNODE is approved, all existing RNODEs and SNODEs on the network are notified, and the new node is whitelisted on each of those devices, with the appropriate permissions depending on whether it is an SNODE or RNODE.

<u>Scenario 4</u>: Adding RNODEs and Connecting SNODEs


Some computer
start
mDNS service starts "rnode._tcp.local."
Connect to other RNODEs
Registry Node accepts incoming storage node connections
Store IP address and UUID of SNODE in JSON file
If SNODE does not provide UUID, provide it one
Some computer
start
Look for registry nodes on the network via mDNS discovery
Connect to specific registry nodes
If initial connection, obtain UUID from registry node
Store UUID and use it as a handshake

External APIs and Frameworks

API/Framework	Purpose	Key Functions
AWS-BOTO3	● Cloud storage integration	● Upload files to AWS S3 buckets ● Download files from AWS S3 buckets
PyCryptodome	● Encryption and decryption of data	● AES-256 GCM encryption ● Password hashing with scrypt
Zeroconf	● Network service discovery	● Broadcast RNODE availability ● Discover SNODEs on the local network
ReedSolomon (reedsolo)	● Erasure coding for data redundancy	● General parity chunks ● Recover missing or corrupted chunks
Textual	● Terminal User Interface (TUI)	● User input handling ● Displaying node status and file lists

Algorithms

File Encryption & Chunking

Goal: Ensure encrypted file chunks appear random, preventing attackers from recovering information.

Description: The system uses AES-256 Galois/Counter Mode (GCM) to ensure both data confidentiality and integrity. Files are split into smaller, equal-sized chunks to distribute them across storage nodes efficiently.

- **Encryption:** Each chunk is encrypted with AES-256 GCM using a key derived from a user-provided password, a randomly generated salt, and nonce. This ensures even identical files produce unique ciphertexts when encrypted with the same password.

- **Chunking:** Files are divided into equal-sized chunks, with padding applied to ensure uniform chunk sizes. Python's concurrent.futures handles parallel encryption of chunks, optimizing performance on multi-core systems.

Erasure Coding

Goal: Enable recovery of missing or corrupted chunks if fewer than half of the parity chunks are lost.

Description: Reed-Solomon error correction code generates parity chunks for fault tolerance.

- **Parity Generation:**
    - Each byte of the encrypted chunks forms a byte string across chunks (e.g., the first byte of each chunk).
    - Reed-Solomon generates additional parity bytes, which are written to separate files as parity chunks.
- **Decode:**
    - During reconstruction, if chunks are missing, the system checks if the number of missing chunks exceeds half the parity chunks.
    - If recoverable, the Reed-Solomon algorithm reconstructs the missing chunks by copying data from existing chunks to enable recovery.

Checksum Verification

Goal: Maintain the integrity of the file, make sure every modification in the ciphertext will result in a decryption failure.

Description: A SHA-512 checksum algorithm validates the integrity of files before and after encryption/decryption.

- **Hash Generation:** SHA-512 reads the file in 4 KB chunks to compute a hash, preventing memory overflow with large files.
- **Storage & Comparison:** Hashes are stored in hashes.json. Upon decryption, the computed hash is compared with the stored hash to verify integrity.

UUID Management for Nodes

Goal: Reliably identify nodes without relying on static IP addresses.

Description: UUIDs are assigned to nodes to ensure consistent identification even when IP addresses change.

- **Generation & Handshake:**
- When a storage node first connects to the registry node, the registry node generates a UUID and sends it to the storage node.
- The storage node stores this UUID and uses it in all future interactions with the registry node.

Peer-to-Peer Networking

Goal: Enable scalable communication and distributed storage across multiple computers.

Description: The system consists of two node types:

- **Registry Nodes (RNODEs):** Manage the storage network, assign files to storage nodes, and maintain metadata.
- **Storage Nodes (SNODEs):** Store encrypted file chunks and provide status updates.
- **Integration:**
    - **Erasure Coding:** Improves fault tolerance by breaking files into chunks and generating redundant parity data.
    - **File Encryption & Chunking:** Ensures secure, efficient distribution of encrypted chunks.
    - **UUID Management:** Facilitates reliable identification of storage nodes within the network.