Secure P2P File System

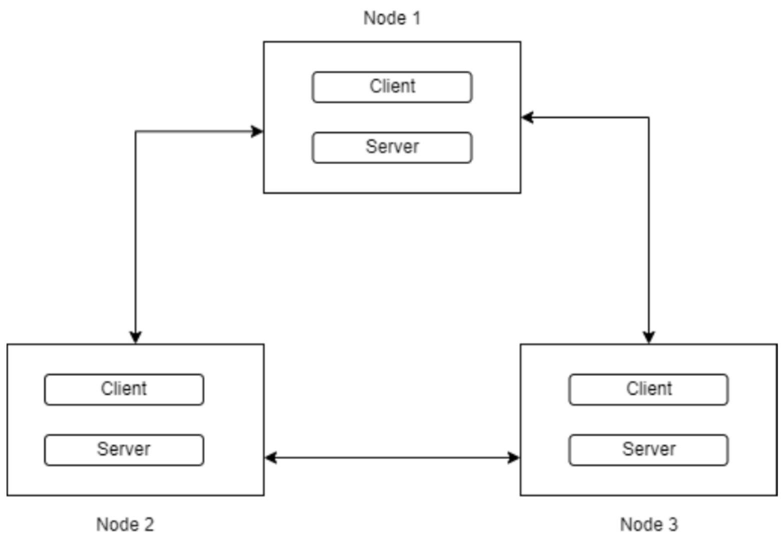
Group 3

Introduction

The project's objective is to create an encrypted peer-to-peer (P2P) distributed file system that will allow the nodes in the network to store the data on other untrusted servers. The project allows a peer node to create a file, replicate it in the network, grant read/write permission to other nodes, update, delete, and restore it. The emphasis of the system is confidentiality and integrity of the file contents by preventing unauthorized access. This report discusses in detail how the requirements have been satisfied in the project.

File System Design

The peer-to-peer system consists of nodes interconnected to each other with equal access to resources. For the project, each node maintains the list of all the other nodes part of the network. This static configuration simplifies the network configuration and initiates requests to other nodes. When a node in the network starts its terminal, it first creates an RSA key and shares the public key with other nodes.

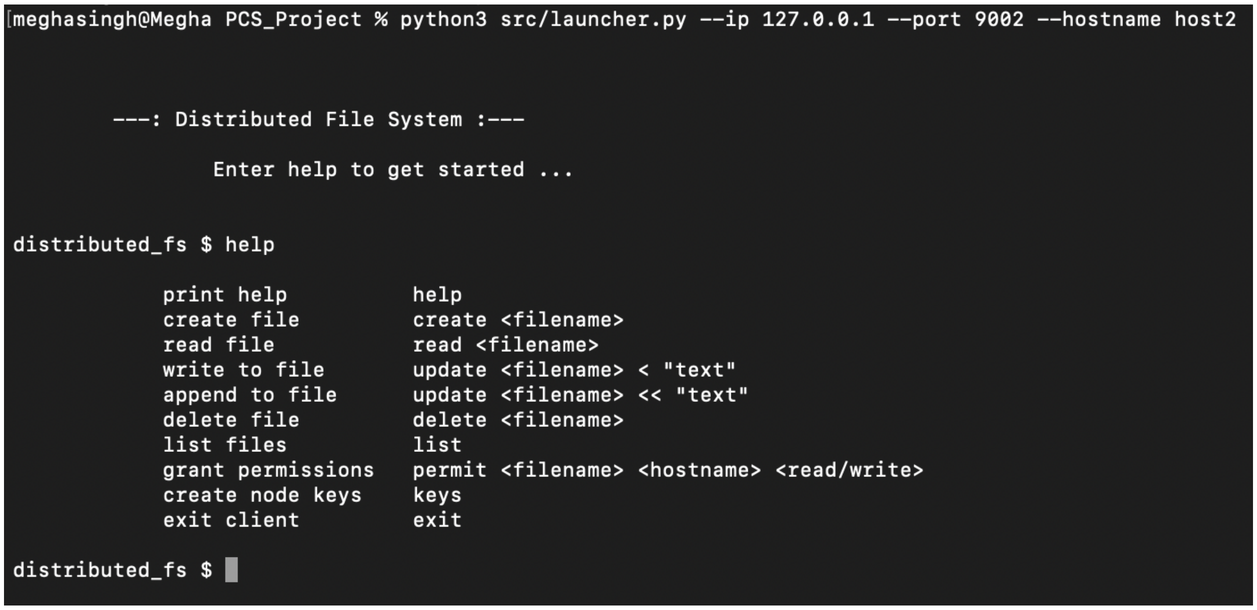


This public key will then be used to share the data between specific nodes with confidentiality. Since this is a peer-to-peer system, we will have all the nodes running both the server and the client.

Client Interface

The project interface will resemble a Linux terminal, with the client program capable of executing various functionalities. The supported features include :

- Generate RSA keys for the server
- Revoke server keys (update)
- Create a file
- Write to a file
- List all accessible files
- Read file
- Delete file
- Restore deleted files
- Grant read/write permissions to other nodes.



```
[meghasingh@Megha PCS_Project % python3 src/launcher.py --ip 127.0.0.1 --port 9002 --hostname host2



        ---: Distributed File System :---

                Enter help to get started ...


distributed_fs $ help

            print help          help
            create file         create <filename>
            read file           read <filename>
            write to file       update <filename> < "text"
            append to file      update <filename> << "text"
            delete file         delete <filename>
            list files          list
            grant permissions   permit <filename> <hostname> <read/write>
            create node keys    keys
            exit client         exit

distributed_fs $
```

Fig 1. Initial Client Interface

To encrypt any peer-to-peer communication, prior to accessing any feature, the user must create a private/public key pair, which will be used for encryption in the file system. The public keys of all users will be shared within the network to enable file sharing.

File Operations (CRUD)

Any node in the network can create, write, read, delete, restore, and list files as described below.

Create

The process of creating a file involves generating a random RSA private and public key by the client in order to encrypt the data. Once the keys are generated, the client interface allows for writing data to the file. When the content is complete, the data is encrypted with the public key generated for the file, and the encrypted data is then replicated on the peers. As the keys for the files are not shared with other users, they are unable to access the file content. For instance, if P is the private key and Q is the public key generated for the file F, the encrypted text will be

$$e = Encryption(F, Q).$$

And to decrypt, we will use the private key of the file as,

$$F = Decryption(e, P).$$

When a file is created in the file system, the node responsible for its creation is considered the file owner. The file owner is responsible for generating and maintaining the public and private keys for the file. The user will save the file with a name and a unique file ID. After the file is saved, the owner node will encrypt it, including its name, and sign it with its private key. The owner node will then replicate the file and its associated ID to all the nodes in the network. Upon receiving the replicated copy, each node will save it to its preferred location and make an entry into its database.

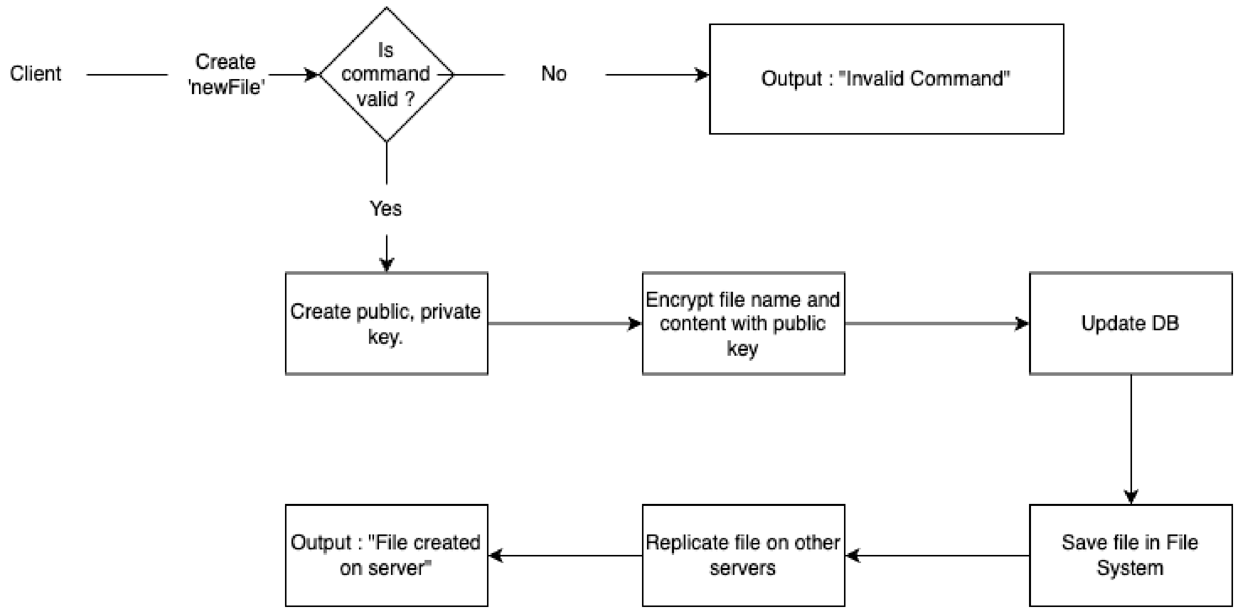

Client
Create 'newFile'
Is command valid ?
No
Output : "Invalid Command"
Yes
Create public, private key.
Encrypt file name and content with public key
Update DB
Save file in File System
Replicate file on other servers
Output : "File created on server"

Fig 1. Create File workflow diagram

Read

To read a file in the system, the node must either be the owner of the file or have the necessary permissions to read it. In order to access the content of the file, the private key associated with the file must be used to decrypt it.

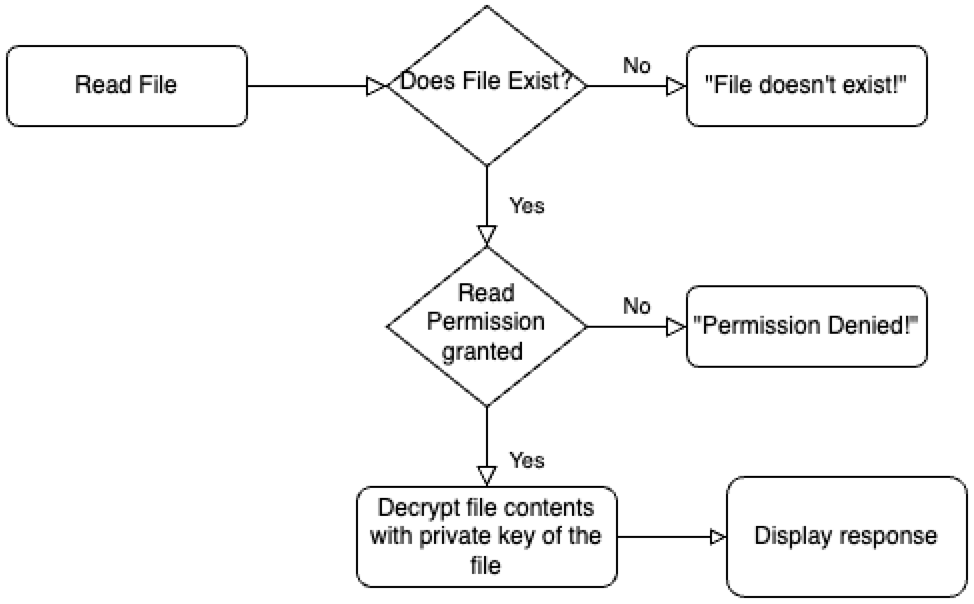


Fig 2. Read File workflow diagram

Write File

In order to write a file, a node must either be the owner or have permission to write. In this case, the node will require both private and public keys to make a write. The public key will be used to decrypt the existing content while the private key will be used to encrypt the file.

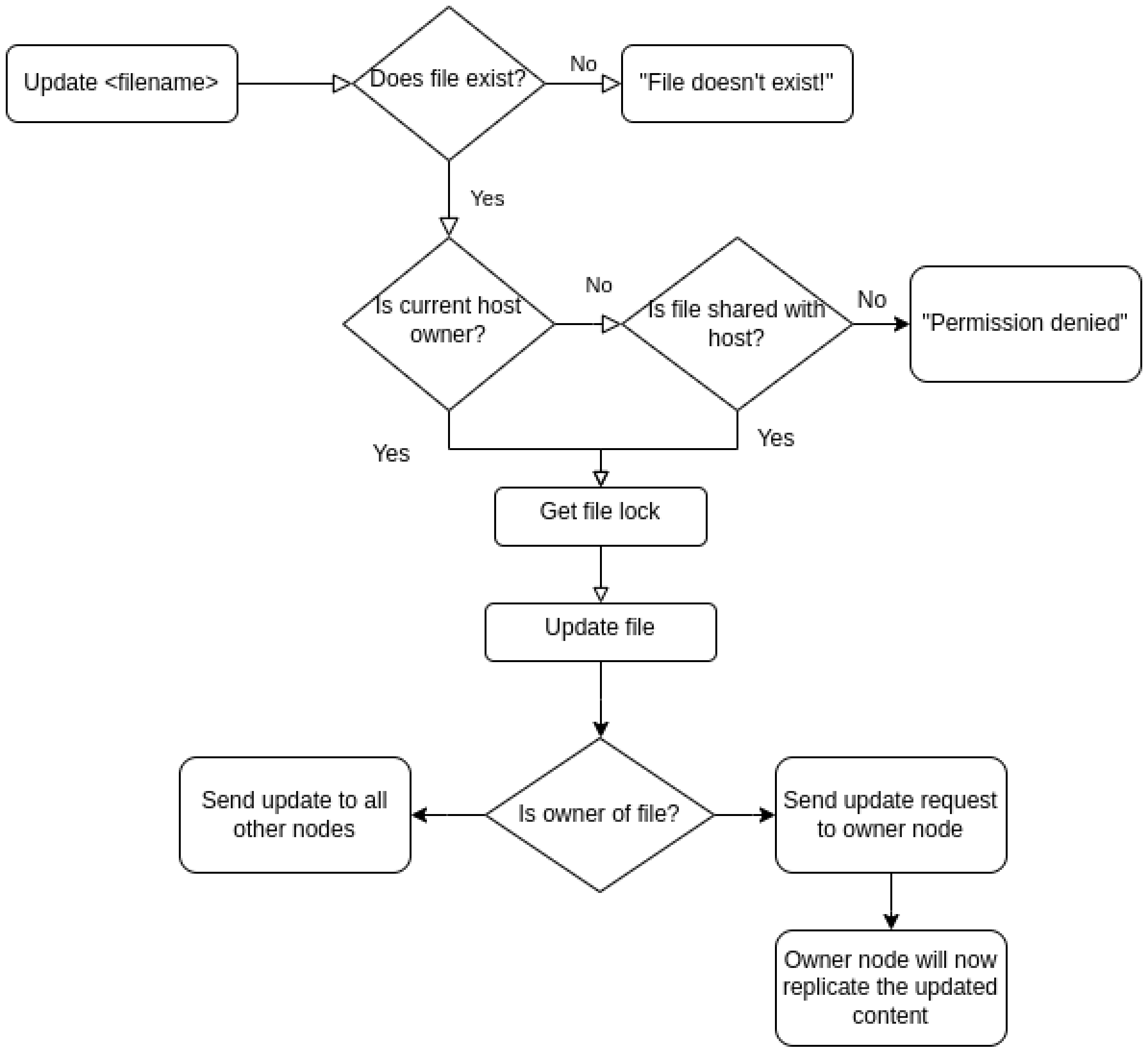


Fig 3. Write to File workflow diagram

Example: Node_1 wants to grant write access to Node_3 for the file F. The public key of the file is shared securely by encrypting it with the public key of Node_3, which is $Q(Node\_3)$. The encrypted key is the write key, which is $writeKey = Encrypt(Q(F), Q(Node\_3))$. Only Node_3, which has the private key associated with $Q(Node\_3)$, can decrypt the shared key and use it to update the file. Other users who don't have the public key of the file won't be able to update it, as they will be missing the key to encrypt the data.

List Files

Each node has restricted access to only the files that they own or have been shared with. To streamline this process, every node will keep track of the files they own and have been shared with. The file name will be encrypted and nodes will need the file's private key to decrypt it, similar to how the file is read.

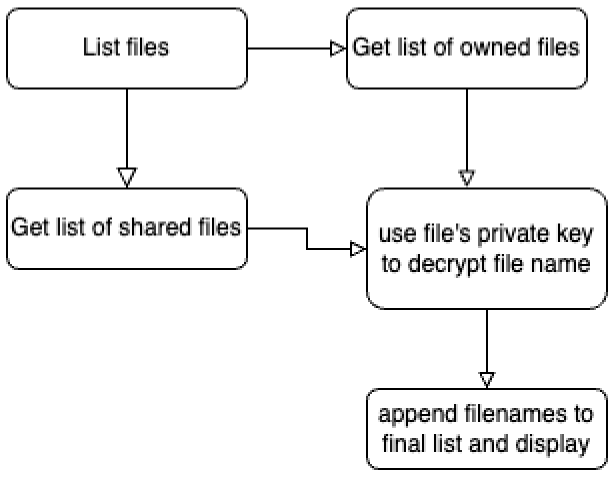

List files
Get list of owned files
Get list of shared files
use file's private key to decrypt file name
append filenames to final list and display

Fig 4. List Files workflow diagram

Delete File

The deletion of a file is a restricted operation that can only be performed by the owner node. When a deletion request is received by a node, it checks for the digital signature to ensure the validity of the request. If the signature is validated successfully, the node proceeds to delete the file along with any corresponding entries that it maintains. This ensures that the file is deleted completely from the network and cannot be accessed by any unauthorized nodes.

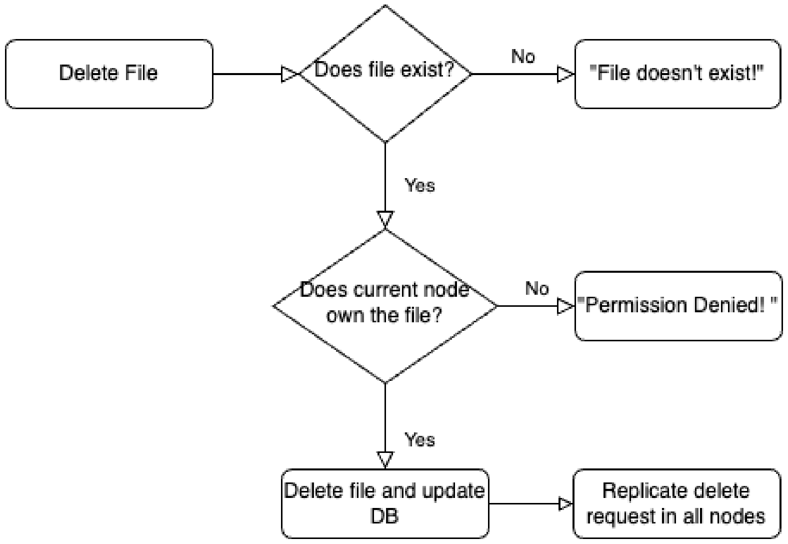

Delete File
Does file exist?
No
"File doesn't exist!"
Yes
Does current node own the file?
No
"Permission Denied! "
Yes
Delete file and update DB
Replicate delete request in all nodes

Fig 5. Delete File workflow diagram

Restore File

The project provides provisions to restore the deleted files. When a node deletes a file, we delete the files from the directory and remove all the relevant entries from the database. However, to restore functionality we maintain a copy in a trash directory. As the request to restore is received, the file is brought back from the trash.

Replication

We follow a simple strategy of replicating files on all the other nodes in the network. Each file is replicated on all nodes in the file system, regardless of the user or group that owns the file. The file content and the name won't be visible to other nodes unless they are granted permission by the owner of the file.

File Permissions

The ability to grant permission for a file is restricted to the owner node. The peer node's client interface will provide the necessary functionality for granting permissions. To enable granting permissions to read and write files, the owner node will keep a record of the nodes that have been granted permission using a ledger. File permissions can be granted by sharing the corresponding file's keys with the other node. These keys can be shared securely by encrypting with the other node's public key. This way only the other node will be able to decrypt it.

Example: Suppose we have private and public keys, $P(F)$ and $Q(F)$ respectively, for a file F that was created by Node_1. Node_1 wants to grant read access to Node_2. Node_2 requires $P(F)$ to read the file. Node_1 encrypts $P(F)$ with $Q(Node\_2)$ before sending it over to Node_2. Therefore, the read key for Node_2 will be $readKey = Encrypt(P(F), Q(Node\_2))$, which will be shared by Node_1 with Node_2.

Read Permission

To grant read permission to another node, the owner node will share the private key of the file with that node. To ensure security, the owner node will encrypt the private key of the file using the recipient node's public key. This way, only the recipient node will be able to access the key, and other nodes won't be able to read it. Once the other node receives the permission and decrypts the private key, it will make an entry for the shared file to keep track of it in the future.

Owner	Permission - R	File_ID	Private_Key_Of_File	<NO PUBLIC KEY>

Write Permission

The other node will receive both private and public keys from the owner node in this case. The table entry on the other node will appear as follows :

Owner	Permission - RW	File_ID	Private_Key_Of_File	Public_Key_Of_File

The file's content can be read using the private key, whereas the other node can use the public key to encrypt and share the updated file with the owner node. The owner node will then take care of updating the file on all the other nodes.

Key Revocation

All nodes in the network can change their own RSA keys. After generating a new public key, the updated key is distributed to all other nodes. Fortunately, there is no need to be concerned about the files since they have their own unique key. Consequently, the owner node will continue to access the files as they did before the key change.

Consistency

To ensure that only the most up-to-date file content is shown when requested, the content is loaded up to the time when the request was processed. If any user modifies the file, other users will have to reload the file to see the changes. As of now, the system doesn't support concurrent writes.

Concurrent Writes

The issue of concurrent writes when a file is shared with multiple nodes is addressed with a simple approach adopted for the project. Whenever a node wants to write to the file, it will first have to obtain a lock from the owner node. The lock will prevent any other nodes from writing to the file while it is being edited. Once the write operation is completed, the lock will be released. While this approach is relatively straightforward, it should be effective for the scope of the project.

Security Considerations

To ensure the security of the file system, it is important to prevent unauthorized nodes from making changes to the file content. In order to address this challenge, our approach was to designate the owner node as responsible for replicating the content. Other nodes with permission to edit the file will send the updated content to the owner node, which will then replicate the changes to all other nodes. This ensures that any changes made to the file are authorized and properly propagated to all nodes. To verify the authenticity of the updated content, each node will check if the update was sent by the owner node using a digital signature. This is possible because each node has the public key of all other nodes in the network.

Conclusion

The project has implemented user key revocation functionality, P2P file creation, sharing, deletion, reading, writing, and restoration. The project has also implemented file permission management, the confidentiality of file names, the detection of unauthorized modifications, encrypted communication between peers, and the detection of malicious file servers.

References

- https://docs.python.org/3/library/
- https://dev.mysql.com/doc/connector-python/en/
- https://grpc.github.io/grpc/python/
- https://protobuf.dev/getting-started/pythontutorial/
- https://www.pycryptodome.org/
- https://www.geeksforgeeks.org/distributed-file-systems/