Distributed File System

Final Project Report

Principles of Computer Security(CMSC 626)

Department of CSEE, UMBC, SPRING 2023

Sai Kumar Vaddepally(ci94858@umbc.edu),

Vamshi Krishna Ginna(ty66290@umbc.edu),

Harsha Meghana Reddy Guntaka(lb2345@umbc.edu),

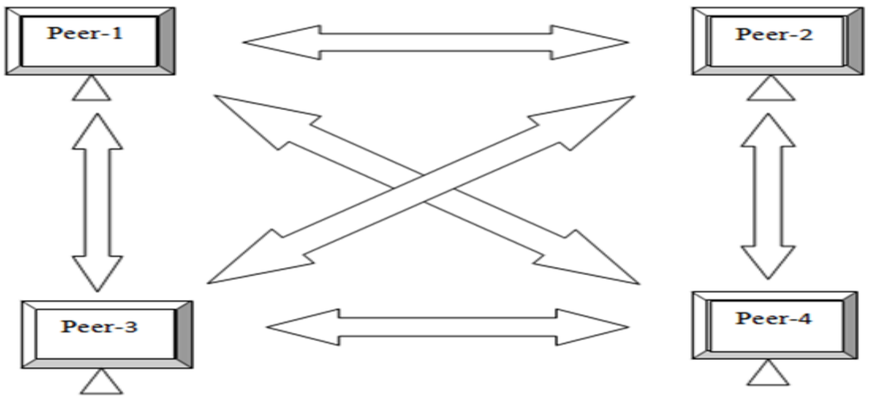
Mohan Manjunath Gujjar(m349@umbc.edu),

Saketh Ram Vangeepuram(xv45698@umbc.edu)

Abstract

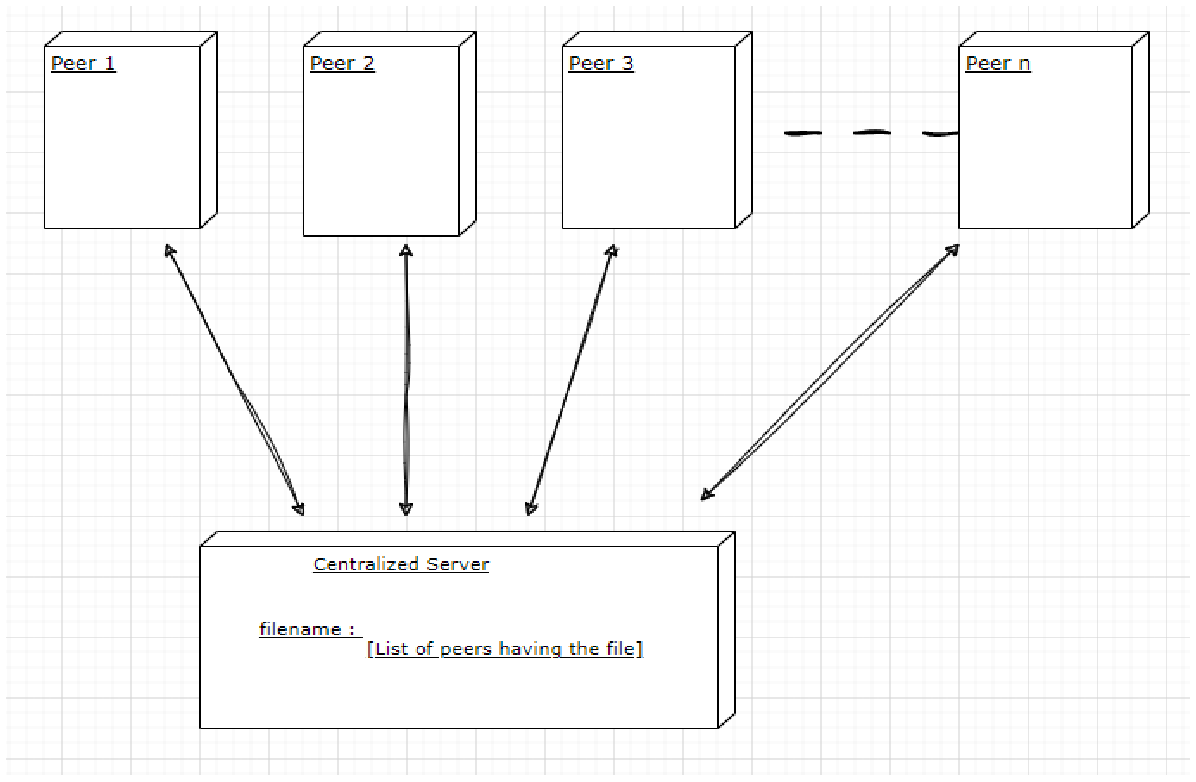
Storing data in traditional storage methods makes it difficult for the user to store and access them securely. Distributed File System allows for storing a large amount of data divided into small chunks and stored between different systems. Peer-to-peer communication is a method by which two or more peers (devices) communicate using encrypted data. Every device is responsible for sending and receiving data, so every device will act as a client and server in this network. Peer-to-peer (P2P) technologies are often utilized for content sharing. P2P file-sharing applications like Napster, WhatsApp, Gnutella, Freenet, and others are some examples that are currently accessible.

We have created a distributed file system with a peer-to-peer architecture and an additional layer of encryption utilizing the Fernet algorithm. This system enforces rigorous permissions while yet allowing users to carry out basic file operations, including creating, reading, writing, deleting, and restoring files. File replicas are encrypted on at least three separate peers to ensure redundancy and prevent data loss. Additionally, the system securely permits peer-to-peer file sharing without running the risk of interference from outside parties or data tampering. All communication is encrypted, including file names and content. Data consistency and dependability are guaranteed since several users can execute operations simultaneously while keeping the ACID properties.



Peer-to-Peer Communication

System Design:



Centralized Peer-to-Peer architecture

Our Peer-to-Peer file system consists of the following components:

Centralized Server:

The Centralized Server(CS) is similar to the directory service. It stores the information of all the connected peers and their IP addresses. It maintains a list of files each peer has made available to share. Each peer would first connect to the Centralized Server for any query. This Centralized server with all the information about the files and the peers will respond to the query with the IP addresses of all other peers with the requested file.

The CS will authenticate the read and write permissions for the file and then responds to the peer :

- CS will respond with IP addresses if the requesting peer has all the permissions to perform operations on the file.

    - Else, CS will deny the request by saying “Access denied to the requested file” message.

Peer :

Each peer will first connect with the Centralized Server(CS). Every file request(read/write) will be sent to the CS first to obtain the list of IPs of the other peers. If CS responds with an IP, the requesting peer will connect to that IP to perform the operations on the file.

Working:

- To participate in the system, a peer must first register with the Centralized Server(CS), without which it is not allowed to be a part of a file system.
- Each client would have its thread associated with it to submit requests.
- A common thread(Centralized Server) would always be listening to handle individual client requests such as reading/writing/updating files.
- Any peer in the system that creates a file will replicate it on other active peers in the file system. Files are saved with specific attributes, such as the owner (the peer who created the file), access permissions, replicated peers, encryption key, lock, and deleted status.
- A Lock is employed with each peer to control concurrent users. If the Lock attribute is active and a file is already in use, another peer requesting the same file will get a message from the CS stating that "another peer is using the file.”

Operations allowed:

- **Create file:**
    1. To create a file, the peer connects to the Centralized Server(CS), and the CS authenticates the peer before allowing the peer to create a new file on the particular server.
    2. The peer will then connect to a server and can create a new file/folder by inputting the name. The server will determine whether the file/folder with the

same name is already present. If a file/folder does not already exist, a new file/folder with the specified name and the specified content is created. The user can set the permissions to the newly created file/folder.

- *Permissions*:
    1. *read-only*:

        The other peers can only read the newly created file/folder.
    2. *read -write*:

        All the other peers will have access to read and write the file.
    3. *private*:

        Only the owner of the file has access to the file.
    4. *default*:
        a. By default, every user will have access to its directory(its local directory) or the path where it creates the file.
        b. If the owner of the file sets permission as *'private'* to all the files the owner owns, then other peers will not have access to the owner's path and hence, cannot access the files in the path.
        c. Instead, if the file has read or write permissions, its present path is accessible to the other peers, using which they can perform read/write/update operations.
    5. The file owner can change(grant additional permissions/revoke existing ones) the permissions of the files it owns at any time, and the same is updated for all the peers in the system.
    6. **Command**: *create [filename] [permissions]*

- **Update file:**
    1. The update operation will first confirm the file's existence when a peer wishes to update a file by supplying the file name. If the file is present, it then verifies the read-write permissions of the peer to the file to see if they are satisfied. If yes, the peer can access the file and update its contents.
    2. If the same file is on different servers, each file is updated with the current changes. This is done by checking the modified timestamp of the file.

3. Each update will check if the file is present on any other servers. If yes, all such files are updated.
    - *Permissions*:
        - Any user with *read-write* permission can access and update the file if it is present.

- **Read file:**
    1. The peer must send a request with a filename to the Centralized Server(CS). CS will first confirm the file's existence when a peer wishes to update a file by supplying the file name. If the file is present, it then verifies the permissions of the peer to the file to see if they are satisfied.
    2. CS will respond with the requested file's IP address if all permissions are satisfied. To access and read the file, we may connect with one of the peers with this file and then access the contents.
        - *Permissions*:
            - Any user can read the file if it has *read* permission set by the owner.
    3. **Command**: *read [filename]*

- **Delete file:**
    1. Each peer can delete the file if it is the file's owner. The deleted file will be removed from the peer's directory.
        - *Permissions*:
            - The owner of the file is the only one who can delete it.
    2. **Command**: *rm [filename]*

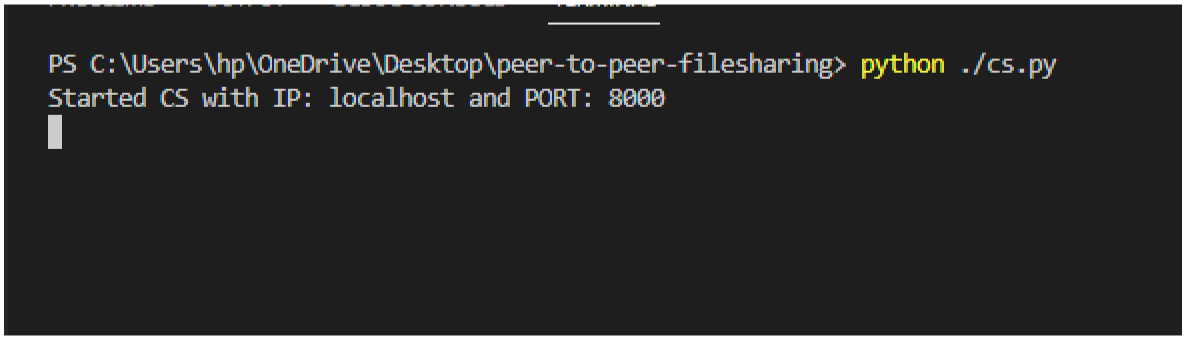
- **Restore file:**
    1. The peer enters the deleted file name if it wants to restore the file. The Centralized Server receives this request and retrieves all the peers who own this file. The

requesting peer can then send the connection request to restore the file by connecting to any of the peers to restore the file.

- *Permissions*:
    - Any user who at least has read permission on the file can restore the file if it is present with any other peers.

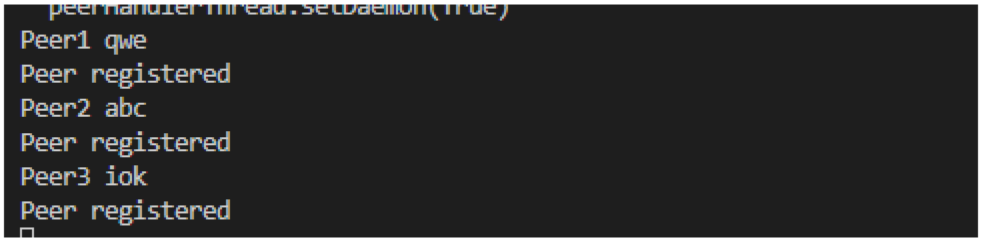
2. **Command**: *restore [filename]*

Sample screenshots:



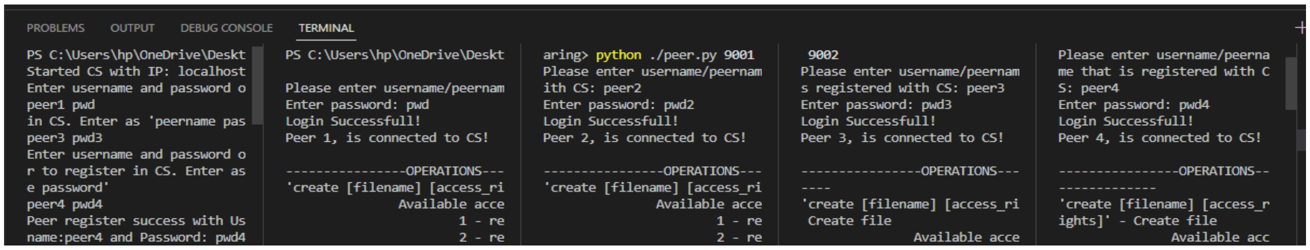
```
PS C:\Users\hp\OneDrive\Desktop\peer-to-peer-filesharing> python ./cs.py
Started CS with IP: localhost and PORT: 8000
```

Executing Centralized Server(CS)



```
  peerHandlerThread.setDaemon(True)
Peer1 qwe
Peer registered
Peer2 abc
Peer registered
Peer3 iok
Peer registered
```

Registering peers



```
PS C:\Users\hp\OneDrive\Deskt
Started CS with IP: localhost
Enter username and password o
peer1 pwd
in CS. Enter as 'peername pas
peer3 pwd3
Enter username and password o
r to register in CS. Enter as
e password'
peer4 pwd4
Peer register success with Us
name:peer4 and Password: pwd4
```

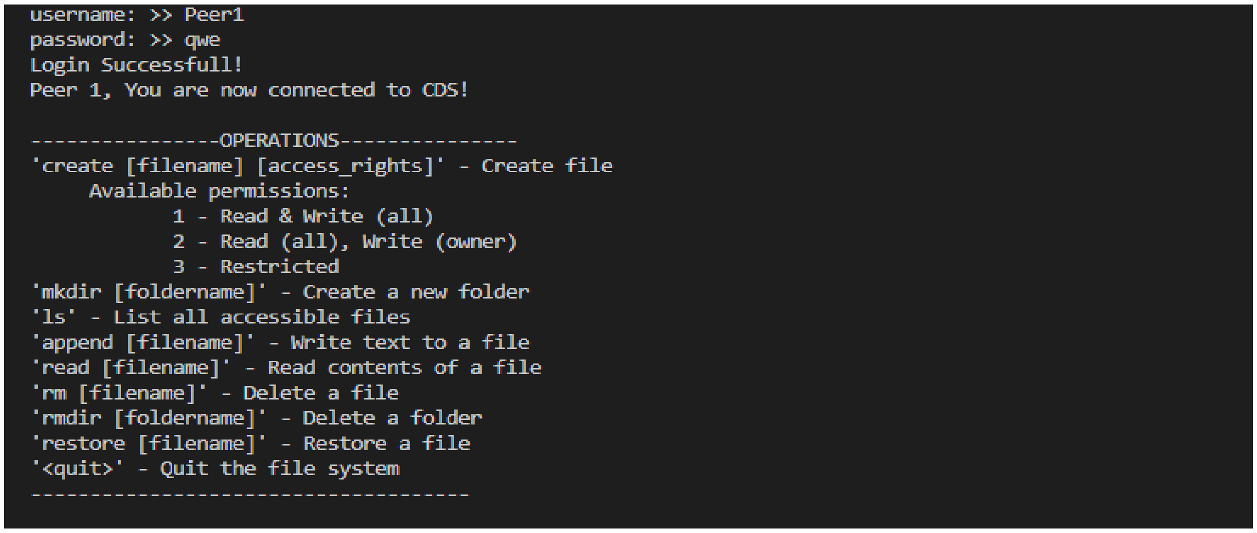
```
PS C:\Users\hp\OneDrive\Deskt

Please enter username/peernam
Enter password: pwd
Login Successfull!
Peer 1, is connected to CS!

----------------OPERATIONS---
'create [filename] [access_ri
               Available acce
                       1 - re
                       2 - re
```

```
aring> python ./peer.py 9001
Please enter username/peernam
ith CS: peer2
Enter password: pwd2
Login Successfull!
Peer 2, is connected to CS!

----------------OPERATIONS---
'create [filename] [access_ri
               Available acce
                       1 - re
                       2 - re
```

```
 9002
Please enter username/peernam
s registered with CS: peer3
Enter password: pwd3
Login Successfull!
Peer 3, is connected to CS!

----------------OPERATIONS---
----
'create [filename] [access_ri
 Create file
               Available acce
```

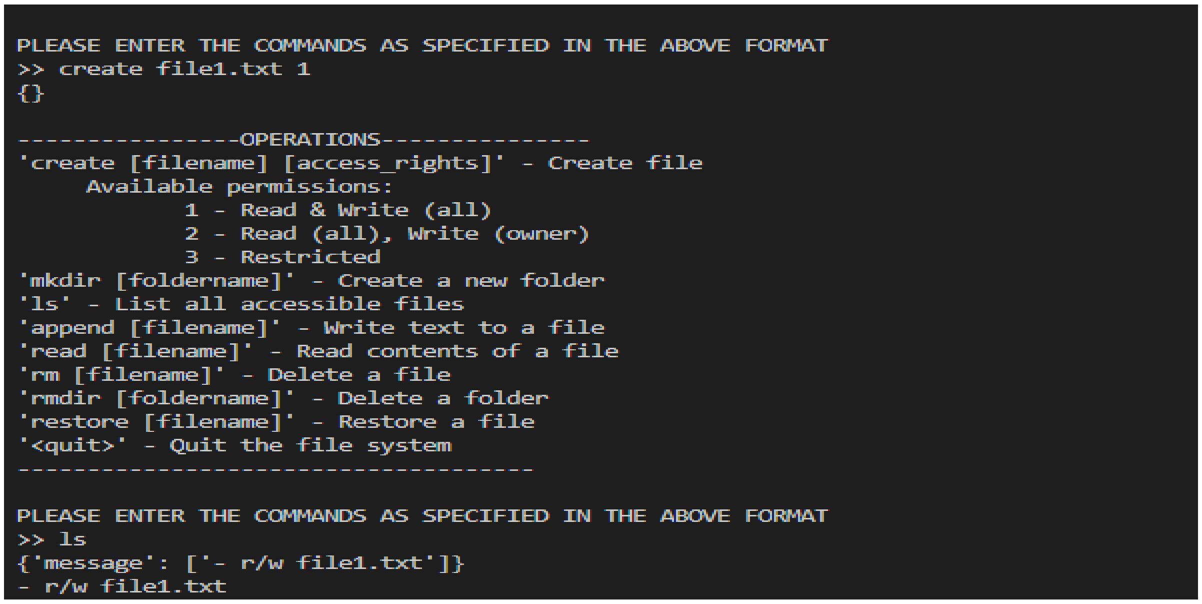
```
Please enter username/peerna
me that is registered with C
S: peer4
Enter password: pwd4
Login Successfull!
Peer 4, is connected to CS!

----------------OPERATIONS--
-------------
'create [filename] [access_r
ights]' - Create file
               Available acc
```

Connecting multiple peers to CS and performing operations



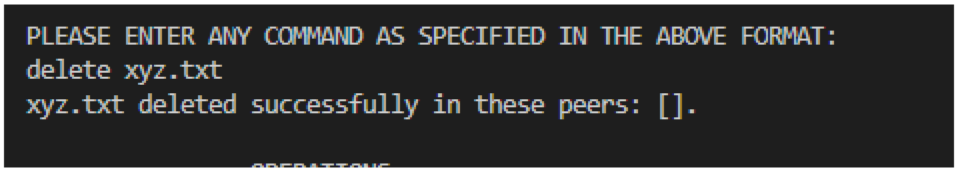
```
username: >> Peer1
password: >> qwe
Login Successfull!
Peer 1, You are now connected to CDS!

----------------OPERATIONS---------------
'create [filename] [access_rights]' - Create file
     Available permissions:
            1 - Read & Write (all)
            2 - Read (all), Write (owner)
            3 - Restricted
'mkdir [foldername]' - Create a new folder
'ls' - List all accessible files
'append [filename]' - Write text to a file
'read [filename]' - Read contents of a file
'rm [filename]' - Delete a file
'rmdir [foldername]' - Delete a folder
'restore [filename]' - Restore a file
'<quit>' - Quit the file system
-------------------------------------
```

Logging in with credentials and connecting to CS



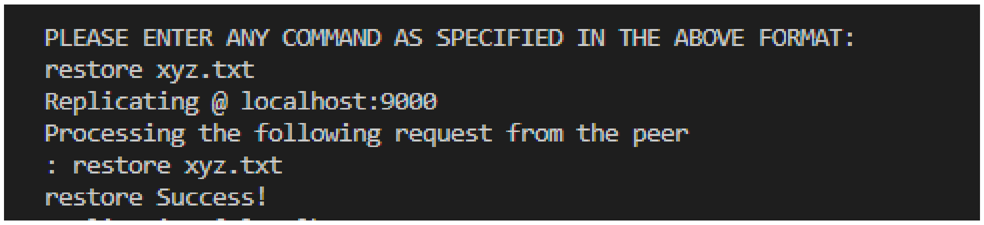
```
PLEASE ENTER THE COMMANDS AS SPECIFIED IN THE ABOVE FORMAT
>> create file1.txt 1
{}

----------------OPERATIONS---------------
'create [filename] [access_rights]' - Create file
     Available permissions:
            1 - Read & Write (all)
            2 - Read (all), Write (owner)
            3 - Restricted
'mkdir [foldername]' - Create a new folder
'ls' - List all accessible files
'append [filename]' - Write text to a file
'read [filename]' - Read contents of a file
'rm [filename]' - Delete a file
'rmdir [foldername]' - Delete a folder
'restore [filename]' - Restore a file
'<quit>' - Quit the file system
-------------------------------------

PLEASE ENTER THE COMMANDS AS SPECIFIED IN THE ABOVE FORMAT
>> ls
{'message': ['- r/w file1.txt']}
- r/w file1.txt
```

Creating a file with read-write permissions and listing the files



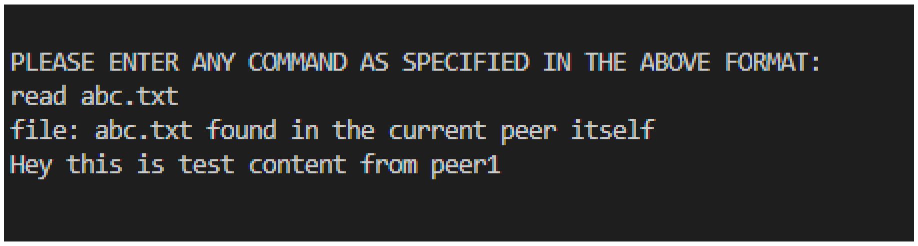
```
PLEASE ENTER ANY COMMAND AS SPECIFIED IN THE ABOVE FORMAT:
delete xyz.txt
xyz.txt deleted successfully in these peers: [].
```

Deleting a file



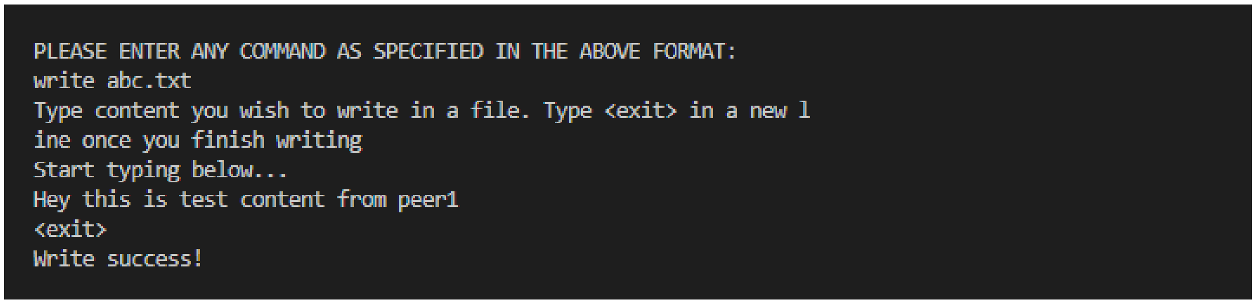
```
PLEASE ENTER ANY COMMAND AS SPECIFIED IN THE ABOVE FORMAT:
restore xyz.txt
Replicating @ localhost:9000
Processing the following request from the peer
: restore xyz.txt
restore Success!
```

Restoring a deleted file



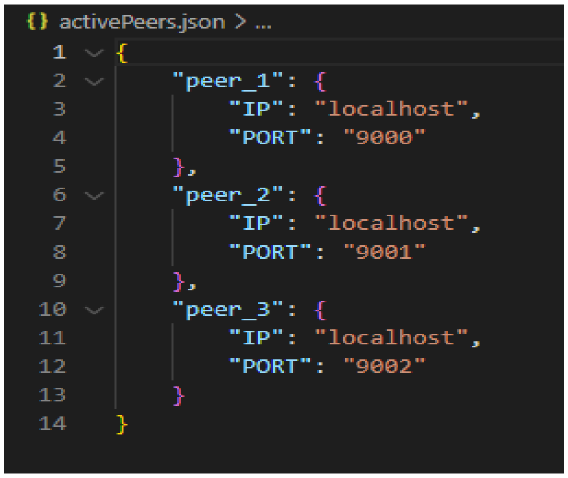
```
PLEASE ENTER ANY COMMAND AS SPECIFIED IN THE ABOVE FORMAT:
read abc.txt
file: abc.txt found in the current peer itself
Hey this is test content from peer1
```

Reading contents from a file



```
PLEASE ENTER ANY COMMAND AS SPECIFIED IN THE ABOVE FORMAT:
write abc.txt
Type content you wish to write in a file. Type <exit> in a new l
ine once you finish writing
Start typing below...
Hey this is test content from peer1
<exit>
Write success!
```

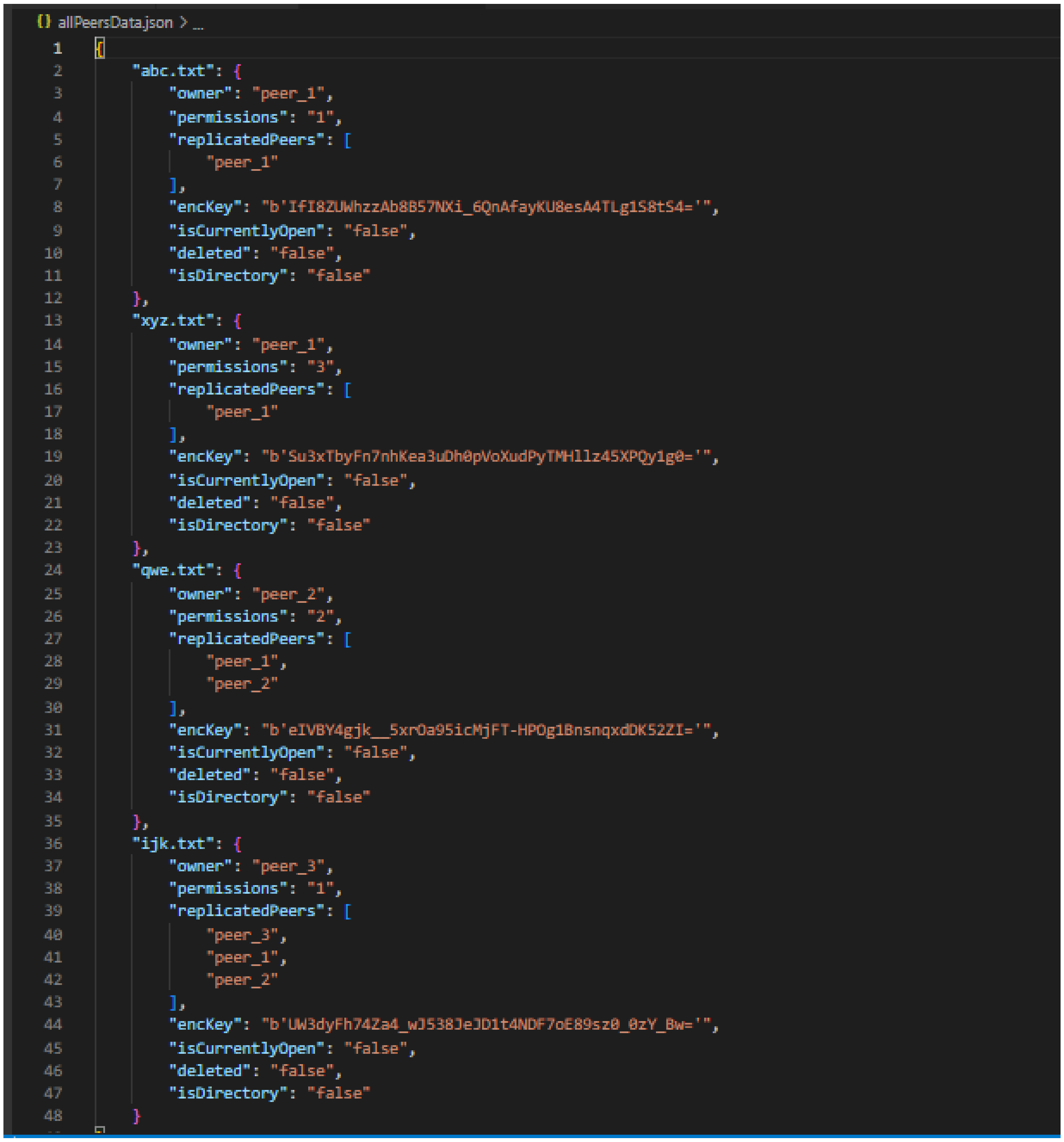
Writing contents to a file



activePeers.json > ...

```json
{
    "peer_1": {
        "IP": "localhost",
        "PORT": "9000"
    },
    "peer_2": {
        "IP": "localhost",
        "PORT": "9001"
    },
    "peer_3": {
        "IP": "localhost",
        "PORT": "9002"
    }
}
```

Showing all peers that are connected.



allPeersData.json > ...

```json
{
    "abc.txt": {
        "owner": "peer_1",
        "permissions": "1",
        "replicatedPeers": [
            "peer_1"
        ],
        "encKey": "b'IfI8ZUWhzzAb8B57NXi_6QnAfayKU8esA4TLg1S8tS4='",
        "isCurrentlyOpen": "false",
        "deleted": "false",
        "isDirectory": "false"
    },
    "xyz.txt": {
        "owner": "peer_1",
        "permissions": "3",
        "replicatedPeers": [
            "peer_1"
        ],
        "encKey": "b'Su3xTbyFn7nhKea3uDh0pVoXudPyTMHllz45XPQy1g0='",
        "isCurrentlyOpen": "false",
        "deleted": "false",
        "isDirectory": "false"
    },
    "qwe.txt": {
        "owner": "peer_2",
        "permissions": "2",
        "replicatedPeers": [
            "peer_1",
            "peer_2"
        ],
        "encKey": "b'eIVBY4gjk__5xrOa95icMjFT-HPOg1BnsnqxdDK52ZI='",
        "isCurrentlyOpen": "false",
        "deleted": "false",
        "isDirectory": "false"
    },
    "ijk.txt": {
        "owner": "peer_3",
        "permissions": "1",
        "replicatedPeers": [
            "peer_3",
            "peer_1",
            "peer_2"
        ],
        "encKey": "b'UW3dyFh74Za4_wJ538JeJD1t4NDF7oE89sz0_0zY_Bw='",
        "isCurrentlyOpen": "false",
        "deleted": "false",
        "isDirectory": "false"
    }
```

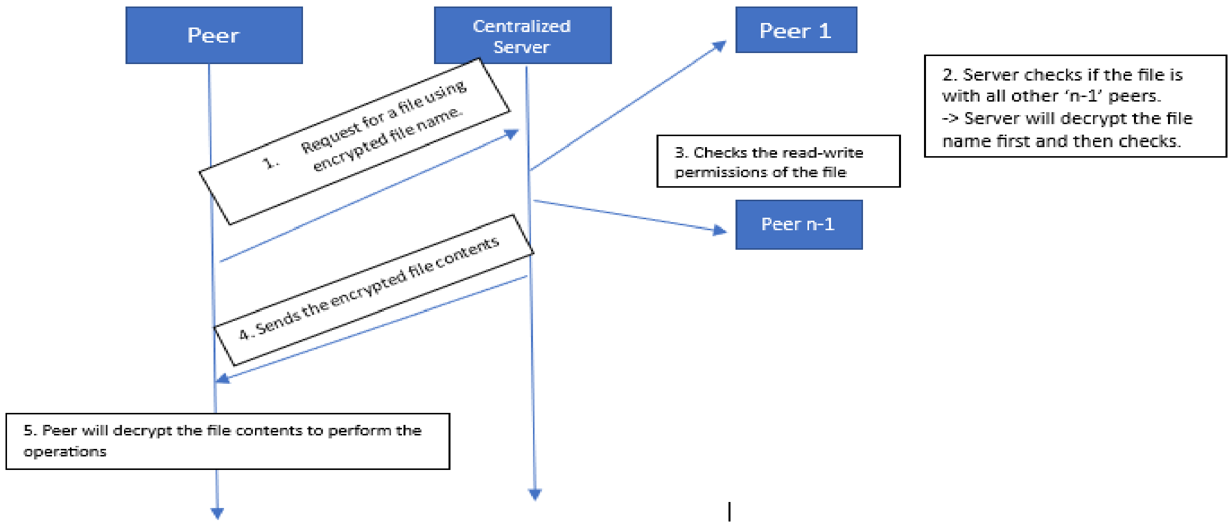
Logging information of each peer

Benchmark results:

- We connected 20 peers to the server at a time and performed a series of read, write, delete, and restore operations on the file system.
- 200 ‘read’ requests took around ~3.988 seconds, each taking around 0.0019 seconds to execute.
- 200 ‘create’ requests took around ~2.085 seconds, each taking around 0.00104 seconds to execute.
- After executing a series of read, write, delete, and restore commands, our file system processes around ~52 to ~96 requests per second approximately.

Security with Encryption :

- The Centralized Server(CS) will assign each user a unique secret key for the encryption. The same key is used to encrypt and decrypt the file contents.
- To achieve encryption, we used the Fernet from Python's cryptography package. Fernet uses AES in CBC mode with a 128-bit key for encryption using PKCS7 padding.
- Fernet guarantees that a message encrypted using it cannot be manipulated or read without the key. Fernet implements symmetric (or “secret key”) authenticated cryptography.



Peer
Centralized Server
Peer 1
2. Server checks if the file is with all other 'n-1' peers.
-> Server will decrypt the file name first and then checks.
1. Request for a file using encrypted file name.
3. Checks the read-write permissions of the file
Peer n-1
4. Sends the encrypted file contents
5. Peer will decrypt the file contents to perform the operations

Encryption algorithm in action

Key Revocation :

If the key assigned by Centralized Server(CS) is not the one we want to use anymore or if we feel that the key has been compromised, we want to be sure that nobody can use that key to do any type of encryption. We can use Certificate Revocation List(CRL) maintained by Certificate Authority(CA). In order to access the CRL and look at the revocation list, we can use a protocol called OSCP(Online Certificate Status Protocol).

Our implementation of a Centralized Server(CS) has the functionality of checking for a compromise of a peer. Hence, Centralized Server(CS) will maintain a replacement key for every connected peer. If we ever feel that the key has been compromised, CS will send this replacement key and updates the same with the peer.

Conclusion

We have developed a Peer-to-Peer Encrypted Distributed file system(DFS) using Python Programming Language. The present distributed file system (DFS) included Fernet encryption and all the other components required for a reliable P2P-based system. We thoroughly tested the DFS with four peers to ensure all CRUD operations worked correctly.

Future Enhancements

- This project can be further improved to support other files besides .txt files.
- We can build a GUI(Graphical User Interface) for performing CRUD operations, making a user-friendly system.
- Adding consistency protocols and fault tolerance can make the current file system more consistent and reliable.

References

1. S. Malgaonkar, S. Surve, and T. Hirave, “Distributed files sharing management: A file sharing application using distributed computing concepts,” IEEE Xplore, Dec. 01, 2012. https://ieeexplore.ieee.org/document/6510207 (accessed Apr. 08, 2022).
2. A. Mitra, “How to encrypt and decrypt data using Fernet and MultiFernet in Python?” The Security Buddy, May 21, 2021. https://www.thesecuritybuddy.com/cryptography-and-python/how-to-encrypt-and-decrypt-data-using-fernet-and-multifernet-in-python/#:~:text=The%20Fernet%20module%20uses%20a%20symmetric-key%20encryption%20algorithm (accessed Apr. 28, 2023).
3. http://css.csail.mit.edu/6.858/2013/projects/ulziibay-otitochi-joor2992.pdf
4. https://medium.com/@amannagpal4/how-to-create-your-own-decentralized-file-sharing-service-using-python-2e00005bdc4a
5. https://www.geeksforgeeks.org/p2ppeer-to-peer-file-sharing/