UNIVERSITY OF MARYLAND BALTIMORE COUNTY

CMSC 626 – PRINCIPLES OF COMPUTER SECURITY

A PEER-TO-PEER DISTRIBUTED ENCRYPTED FILE SYSTEM

PROJECT REPORT

TEAM 03

Anjani Lingamallu BG39251 anjanil1@umbc.edu
Varunpal Gopidi DW89091 vgopidi1@umbc.edu
Seetaram Oruganti QL21570 seetaro1@umbc.edu
Sai Teja Challa JA52979 schalla4@umbc.edu
Kavya Vellanki PI66332 kavyapv1@umbc.edu

Under the guidance of,

Professor
Gerald S. Tompkins
Teaching Assistant
Ran Liu

CONTENTS

1. ABSTRACT
2. INTRODUCTION
3. SYSTEM OVERVIEW
4. FEATURES
    4.1 Encryption
    4.2 Permissions and User Management
    4.3 Replication
    4.4 Security
    4.5 Logging
5. IMPLEMENTATION
6. CONCLUSION
7. FUTURE SCOPE
8. REFERENCES

1.ABSTRACT

In today's digital era, the need for secure and reliable file sharing services is more critical than ever. To address this need, we propose a novel P2P distributed file system that prioritizes user privacy and data integrity. This innovative system enables users to store and transfer files on untrusted remote servers while maintaining complete control over their data. P2P distributed file systems revolutionize file sharing by dispersing data across multiple nodes, ensuring continuous access even if individual nodes fail. Encryption safeguards the integrity of files, preventing unauthorized access. By distributing client requests among multiple servers, P2P DFS alleviates the burden on a single server, enhancing overall system performance. This robust P2P DFS architecture lays the foundation for developing secure and efficient file sharing solutions.

2. INTRODUCTION

A typical file system manages files and directories stored on local storage devices such as hard disks or solid-state drives within the constraints of a single computer or computational node. File and directory access is often confined to the computer on which the file system is mounted, and file paths are unique to that system. All the applications must be installed locally to store and manage data. This design presents a single point of failure, which means that if the system fails or if the storage device malfunctions, access to the stored files may be compromised.

The distributed file system on the other side is designed for scalability, allowing the users to expand their storage and capacity to handle large number of applications which are concentrated towards the data or file operations. It is also effective in replicating the data on multiple nodes which ensures data availability even if one of its nodes fails.

We have used socket programming and SQL database to cater to development of a file system as such. Our system is designed to safeguard user confidentiality by employing robust encryption techniques such as RSA and AES along with integrated log mechanisms to detect and deter malicious activity. Additionally, permission-based user roles ensure that only authorized individuals can access and modify files, further enhancing security. To guarantee data integrity, we leverage cryptographic hashes and ensure that files remain unaltered and accessible in their most recent versions. By combining the efficiency of distributed networks with advanced security features, our P2P file system represents a significant step forward in file sharing technology. Our solution is not only secure but also streamlined and user-friendly, catering to all file storage and sharing needs.

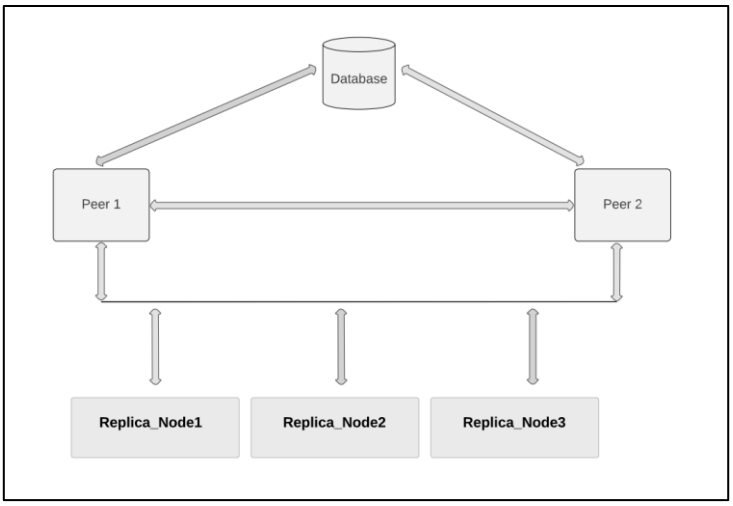


Figure 2.1

As shown in Figure 2.1, The Peers acts as both Client and Server at the same time while they can share the resources and perform all the file operations like read, write etc. securely. These peers also interact with the Database to retrieve and store files. The replica Nodes acts as replicas to the Peers and back's up all the data or the files which eventually achieves availability of the data.

3. SYSTEM OVERVIEW

A DFS is a network of interconnected computers, also known as peers, that share files and resources. Each peer acts as both client and a server, contributing its resources to the network. To ensure everyone has access to the latest version of the documents, the system allows users to create, write, read, delete, and restore files. For security reasons, special permissions and access restrictions are implemented to control user access. To share a file within P2P DFS network, a user simply adds it to their local file system.

The file is broken into smaller segments and encrypted for security purposes by the user's P2P DFS client software. These encrypted file segments are then distributed among different nodes in the network. When another user requests a specific file, the client software contacts other peers in the network to locate file segments. If a peer has the requested file segments, it sends them to client software, which reconstructs the complete file on the requesting user's system. In the scenario where two systems want to share a file stored in the distributed system, the client sends a request to network peers. Each peer checks if it has a copy of the requested file. If a peer has the file, it acknowledges the request and establishes a communication channel to share the file. If no peer has the entire file, they collaborate to request the file segments from other nodes in the network. Once all the file segments are gathered, the file is reconstructed and sent to the client. The client can then read and write to the received file, and any changes made are saved and the new version of the file.

4. FEATURES

4.1 Encryption:

To safeguard data during communication and file sharing within the file system, encryption is employed. The RSA algorithm is implemented due to its asymmetric nature, enabling us to achieve both confidentiality and integrity using digital signatures. Enhancing confidentiality involves preventing unauthorized access to sensitive information. RSA, with its public-private key pair mechanism ensures that only authorized individuals can decrypt and access the encrypted data. Integrity, on the other hand, guarantees that data remains unaltered during transmission or storage. RSA's digital signatures provide a tamper-proof mechanism to verify the authenticity and integrity of the data, ensuring that it hasn't been modified or corrupted.

4.2 Permissions and user management:

The Role-based access control (RBAC) is employed throughout the file system to manage permissions. In RBAC, permissions are assigned to predefined roles, and users are granted specific roles based on their job duties.

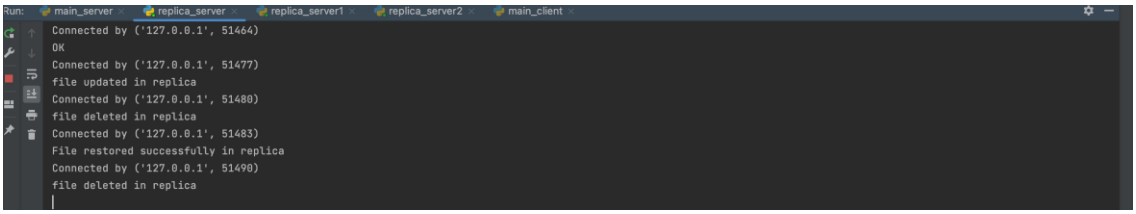
Here, we restrict the permissions to the users on the file operations, such as read, write, delete, or create, are defined and grouped according to these roles. When a user requests access to a file, they send an encrypted request to the system. The system authenticates the user based on their role and the corresponding permissions associated with that role, such as read, write, or delete access. This authentication occurs during the login process, where the user enters their username and password.

Once the authentication is successful, the authorization process takes place. The system checks the user's permissions against the access control model to determine if they have the necessary rights to perform the requested action. If authorization is granted, the system proceeds to execute the requested action.

4.3 Replication:

To ensure data consistency and availability, our system employs 3 synchronized server replicas. When a client initiates an operation, the data is replicated across all three replicas: replica_node1, replica_node2, replica_node3. To achieve this, socket connections are established between the server and replicas, and the pickle library in Python is utilized to convert Python objects into

byte streams for seamless transmission across networks. The server maintains a list of port numbers for all replicas. Once an operation is completed, the server sends a command specifying the operation to be performed, the filename associated with the operation, and the relevant data to be added, updated, or deleted to the replicas. The replica servers then deserialize the object received from the server and execute the operation as instructed, updating their respective directories accordingly. Upon successful data transfer to all replicas, a confirmation message is displayed at the end of each transfer.

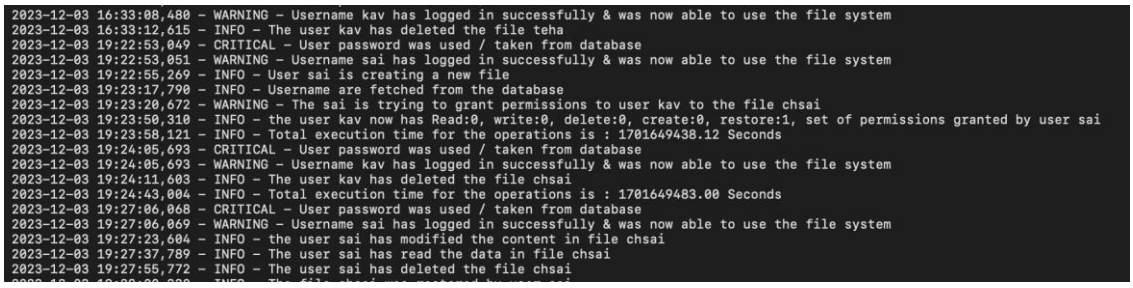


4.4 Security:

Prior to performing any operations, clients must register with the server. During registration, the client's username and password are stored in the 'users' database table. Subsequently, before executing any operation, the server verifies the client's authorization. To further enhance security, we implement an additional layer of protection by employing digital signatures. This involves generating a signature on one side and verifying it on the other. This measure ensures that the server rejects any malicious operations initiated by unauthorized clients.

4.5 Logging:

To facilitate comprehensive system monitoring and error detection, the application employs a robust logging mechanism. This mechanism enables the recording of informational messages and those of higher severity levels to a designated log file. Each log entry is accompanied by a timestamp and an indication of its severity level. This detailed logging proves invaluable for tracking the application's execution flow and promptly identifying potential issues during runtime. The logging module is configured to adhere to the INFO logging level. This implies that only messages with a severity level of INFO or higher, such as WARNING, ERROR, and CRITICAL, will be included in the log file. Messages with a lower severity level, such as DEBUG, will also be acknowledged. This filtering ensures that the log file remains focused on critical information, avoiding excessive clutter and enhancing its usability. The format of each log entry is meticulously defined to provide a clear and concise representation of the logged event. Each entry includes a timestamp, denoted by %(asctime)s, indicating the exact time of the event. The log level, represented by %(levelname)s, clearly identifies the severity of the logged message. Finally, the actual log message, represented by %(message)s, provides a detailed description of the event. This standardized format ensures consistent and easily interpretable log entries, facilitating efficient troubleshooting and system analysis.

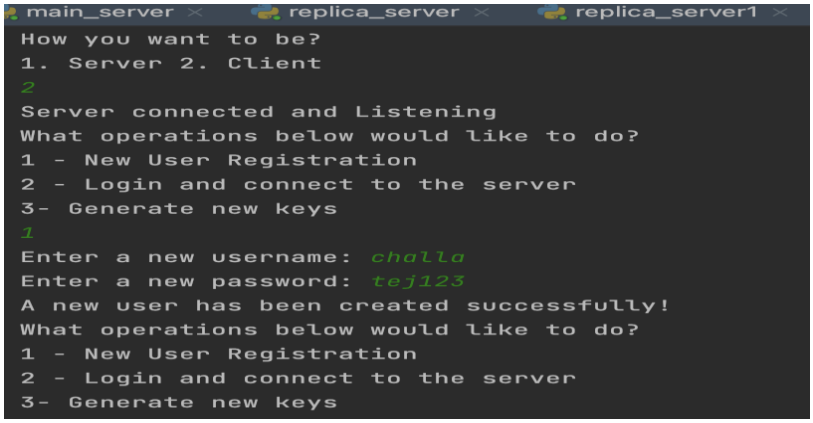


4.5 Key Revocation

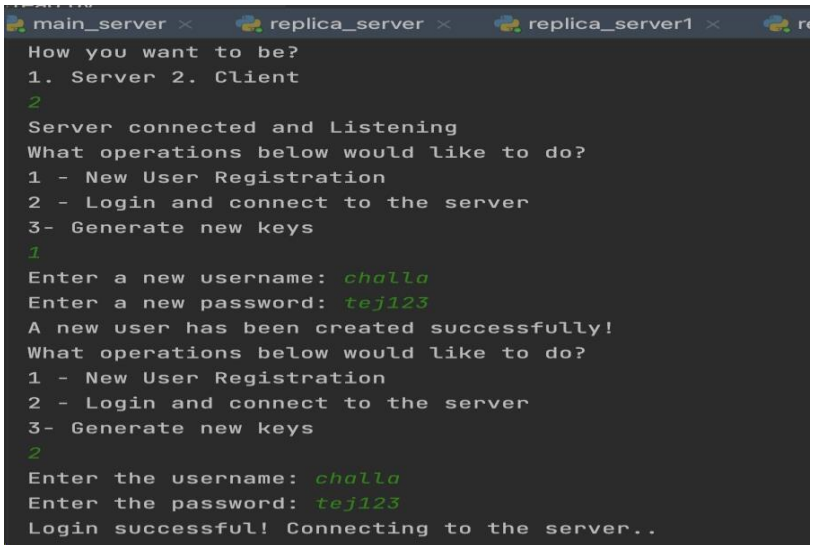
To implement key revocation functionality, we used RSA encryption to generate public and private keys for users during their registration process. These keys, along with the user's access permissions (read, write, create, delete, and update), are then stored in a database table named "access_control". Initially, a private key is generated using RSA encryption with a public exponent of 65537 and a key size of 1024 bits. Subsequently, the public and private keys are extracted from the generated private key using the PKCS1 and PKCS8 formats, respectively. An SQL query string is then constructed to insert the user's public and private keys, along with their access control permissions, into the "access_control" table. This table includes columns for public_key, private_key, username, re (read access), wr (write access), del (delete access), cre (create access), and rest (restrict access). The username serves as a foreign key in the "access_control" table, referencing the primary key attribute username in the "users" table. This ensures that when a user generates new public and private keys, the newly generated key pairs replace the old ones in the database, preserving the user's access permissions. Key revocation functionality is achieved by maintaining the user's access permissions in the "access_control" table, even when new public and private keys are generated.

5. IMPLEMENTATION

User Registration:
If a user is new to the file system, they get to register with a new username and password. Once the registration is done, they have to login into the file system to be able to access the files and perform operations on it.

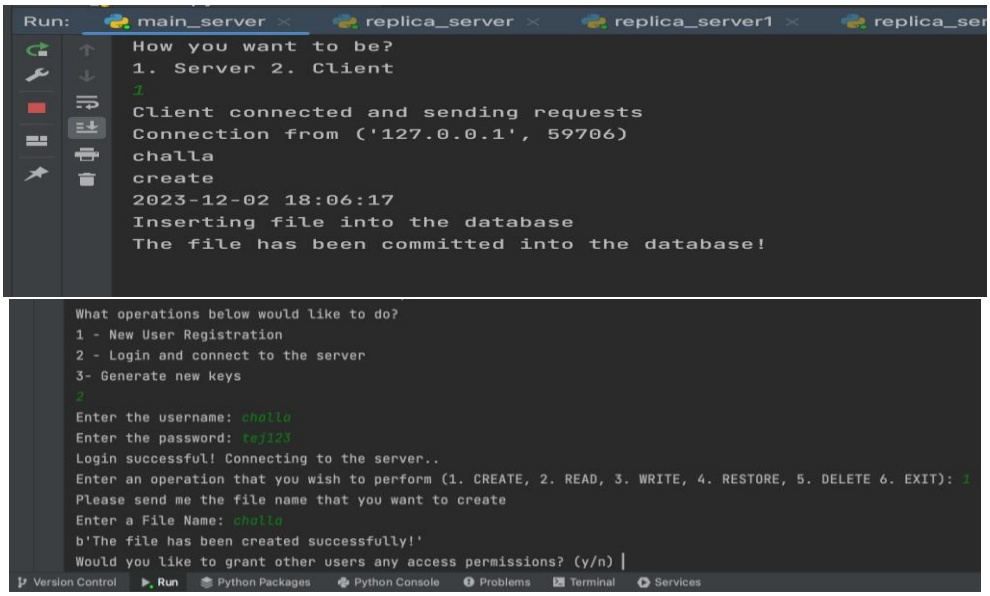


User Login:



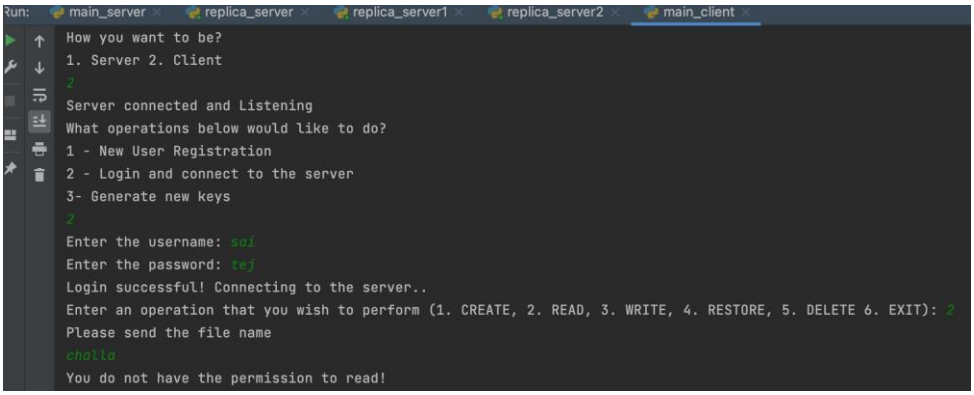
5.1 File Operations (Users can create, delete, read, write, restore files)

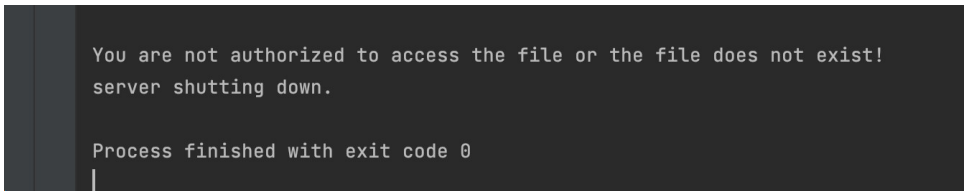
5.1.1 CREATE: To create a file, a client initiates a connection with the server and sends an encrypted message containing their username and the 'create' operation. The server decrypts the message, verifies the user's authorization. And prompts the client for the filename. The client sends the encrypted filename to the server, which decrypts it and creates a new file with that name. Upon successful file creation, the server sends an encrypted message to the client, which the client decrypts and displays. If the client lacks authorization or file creation fails, an appropriate error message is displayed.



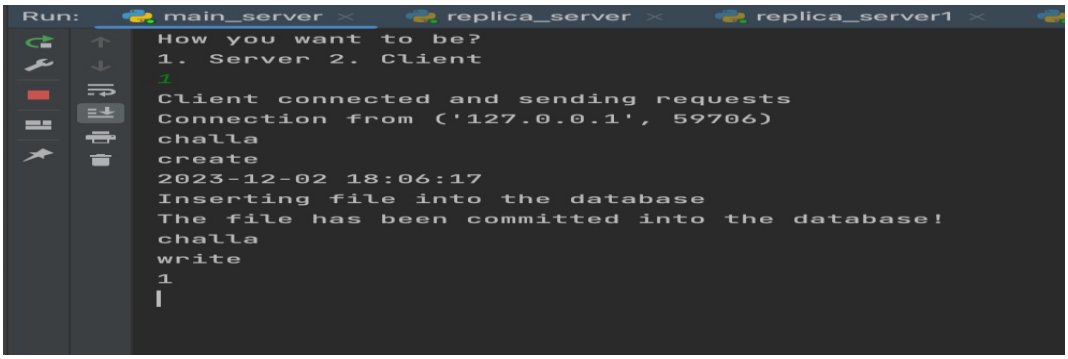
5.1.2 READ: To read a file, the client sends an encrypted message to the server containing their username and the 'read' operation. The server decrypts the message, verifies the user's

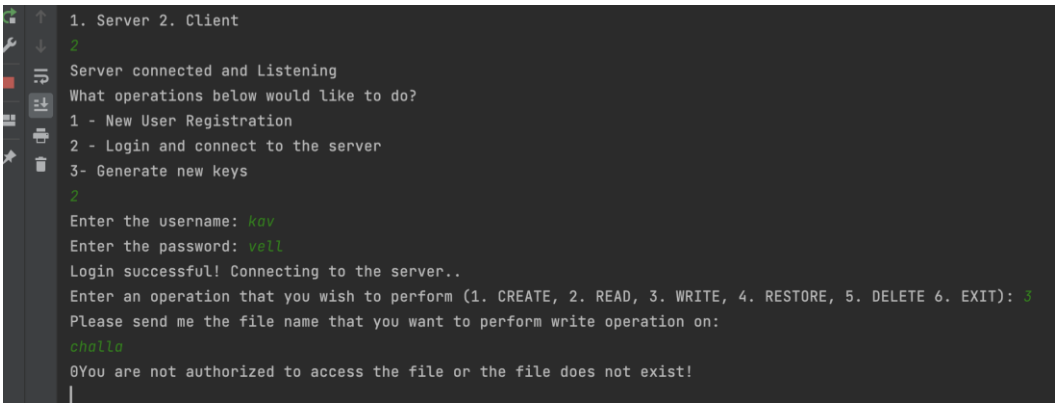
authorization, and prompts the client for the filename. The client encrypts the filename and sends it to the server, which decrypts it. The server then retrieves the client's public key from the database and uses it to encrypt the file. The encrypted file is sent back to the client, which decrypts it using its private key. If the file is not found or the client lacks authorization, an appropriate error message is sent to the client.

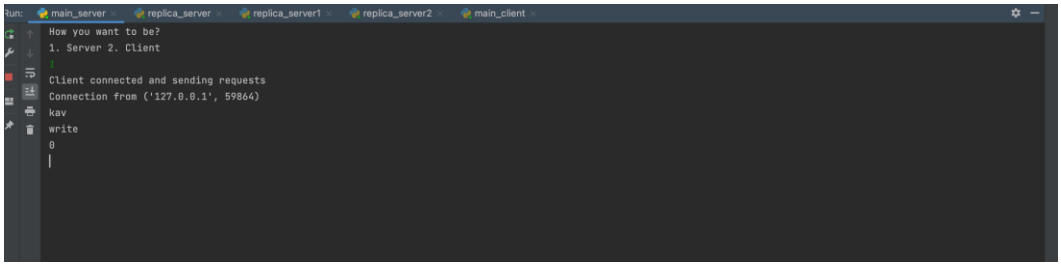




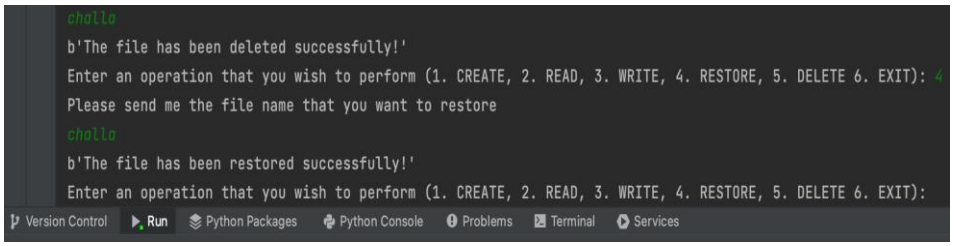
5.1.3 WRITE: To write to a file, the client sends an encrypted message to the server containing their username and the 'write' operation. The server decrypts the message, verifies the user's authorization, and prompts the client for the filename. The client encrypts the filename and sends it to the server, which decrypts it. The server then retrieves the client's public key from the database and encrypts the file contents with it. The encrypted file contents are sent back to the client, which decrypts them using its private key. The client then appends the new contents to the existing file and sends it back to the server. The server decrypts the file and saves the updated contents. If the user lacks authorization or the write operation fails, an appropriate error message is sent to the client**.**



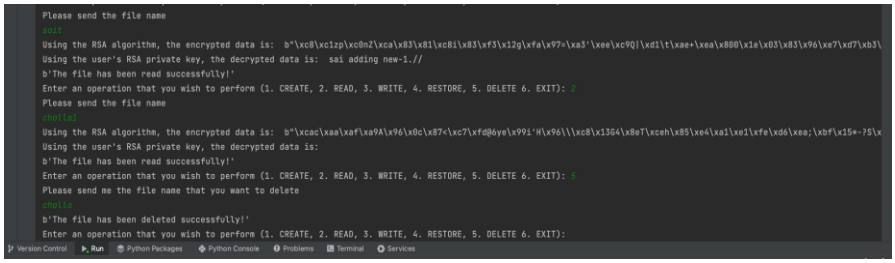


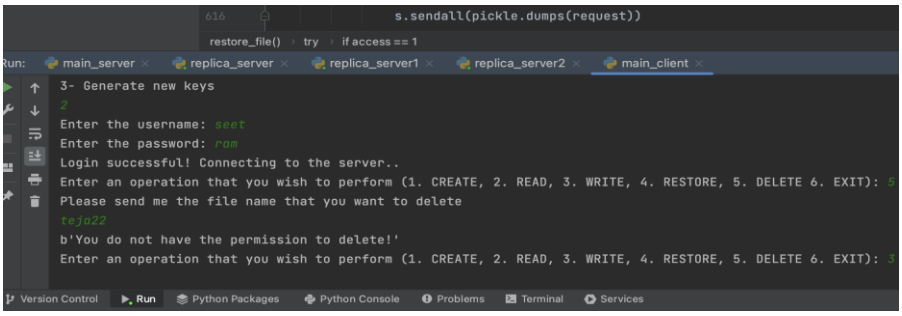


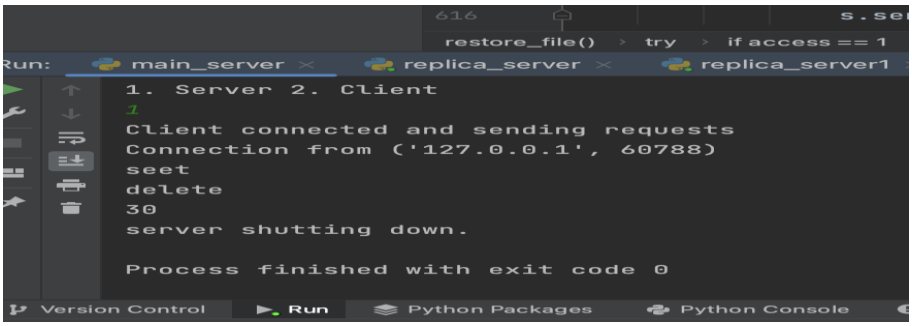
5.1.4 RESTORE: To restore a file, the client sends an encrypted message to the server containing their username and the 'restore' operation. The server decrypts the message, verifies the user's authorization, and attempts to locate the file in the 'restore_files' directory. If the file exists, it is copied to the main directory with its original filename. The transaction table is updated to reflect the restoration, and a success message is sent to the client. If the file cannot be found or the client provides an incorrect filename, an error message is sent to the client.



5.1.5 DELETE: To delete a file, the client sends an encrypted message to the server containing their username and the ‘delete’ operation. The server decrypts the message, verifies the user’s authorization, and attempts to locate the file. If the file exists, it is moved to a folder named ‘restore_files’. A new transaction record is added to the database to document the delete operation, and a success message is sent to the client. If the file cannot be found or the client provides an incorrect filename, an error message is sent to the client.





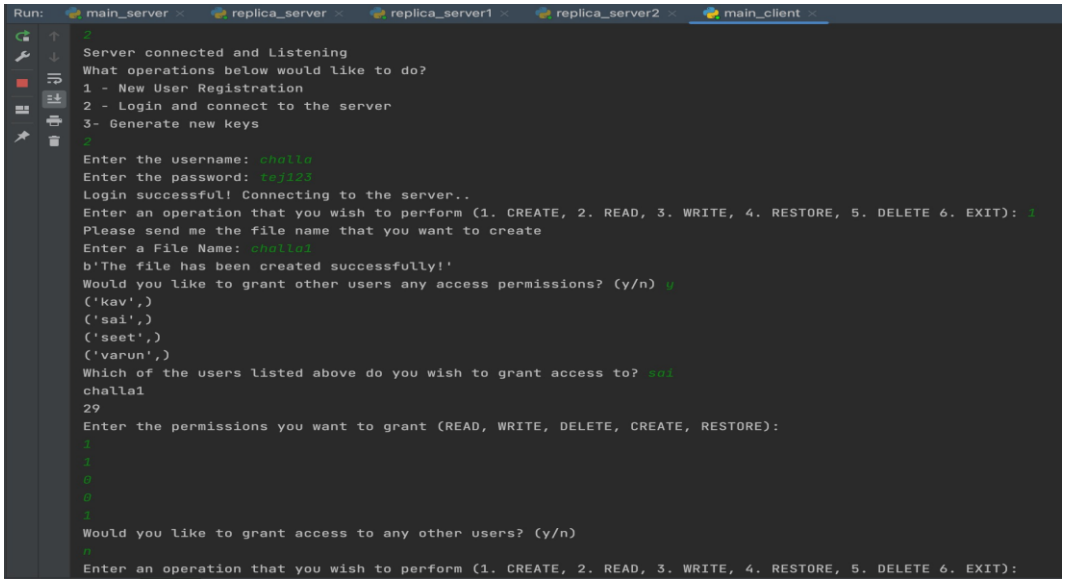


5.2 A client should always see the latest version of a file, or at least that a client should never see an older version of a file after it sees a newer one

Once a secure connection is established between the client and server using the designated port number and addresses, any file operations initiated by the client, such as read, write, or delete, should always utilize the most recent version of the file. For read operations, the client should always retrieve the latest file version. In the case of write operations, when a client modifies the file, the updated content is saved simultaneously on the server and replicated across the system's distributed servers using Python functions. This ensures that outdated file versions are never presented to the user. Similarly, for delete operations, if a client deletes a file, it is stored in a designated folder, enabling its retrieval for potential restoration purposes. Upon deletion, the file is no longer visible to the client.

5.3 Users should be able to set permissions on files and directories, which also requires that your file system be able to name users

Our system empowers users to establish granular control over file access by defining permissions for each file they create. These permissions dictate who can read, update, delete, or restore specific files. To achieve this, we utilize two database tables: “files” and “access_control”. Upon file creation, the filename, the owner's username, and an auto-incrementing file_id (serving as the primary key for the “files” table) are inserted into the database. Subsequently, the system prompts the user to specify the username of the individual they wish to grant permissions to and the specific access privileges they intend to provide. These permissions are then inserted into the “access_control” table, associating them with the corresponding username. Whenever a user attempts to access a file, the system meticulously cross-checks the permissions stored in the “access_control" table and grants access accordingly. This mechanism ensures that only authorized users can interact with specific files.



5.4 File names (and directory names) should be treated as confidential. The data stored in each peer node should be encrypted

Our system effectively facilitates file and directory operations through seamless communication between the client and server. For file operations, the server prompts the user to enter the filename associated with the desired action. The user inputs the filename as a string, which is then transformed into byte format using the UTF-8 encoding standard. This encoded filename is subsequently transmitted to the server via the send () method of the client socket object. Upon reception, the server socket object retrieves the encrypted filename using the Rec() method. Next, the received data is decoded back into a string using the UTF-8 encoding standard, restoring the original filename. Similarly, for directory operations, the user is prompted to specify the directory path where the operation needs to be executed. The provided path is then encoded into bytes using the UTF-8 encoding standard. On the server side, the Rec() method receives the encoded directory path, which is subsequently decoded back into a string format using the UTF-8 standard. Following decoding, the server carries out the requested operation within the specified directory. This robust

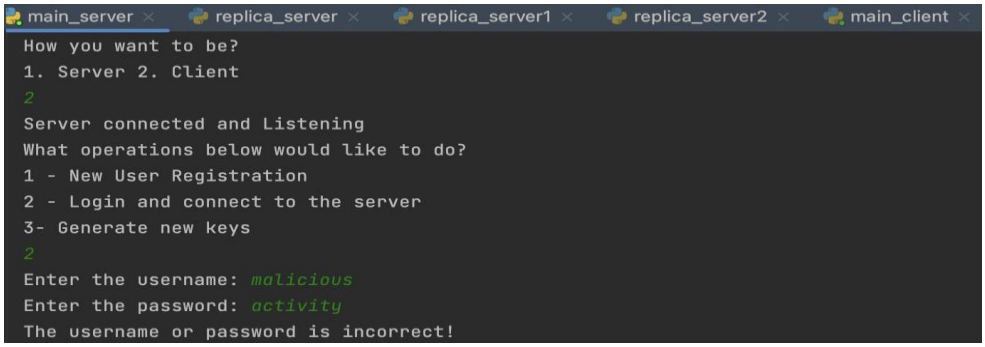
mechanism ensures efficient communication between the client and server, enabling seamless file and directory operations.

5.5 The communication between Peer to Peer should be encrypted

This functionality was successfully implemented using the robust RSA encryption algorithm. Communication between peers encompasses reading file contents, writing file contents, and exchanging communication messages, such as success or failure notifications. To achieve secure data transmission, the server encrypts the content using the client's public key, and the client, in turn, decrypts the contents using its private key. Conversely, the client encrypts the data using the server's public key, and the server decrypts it using its private key. For every communication, both the encrypted and decrypted contents are displayed, providing users with transparency into the encryption process.

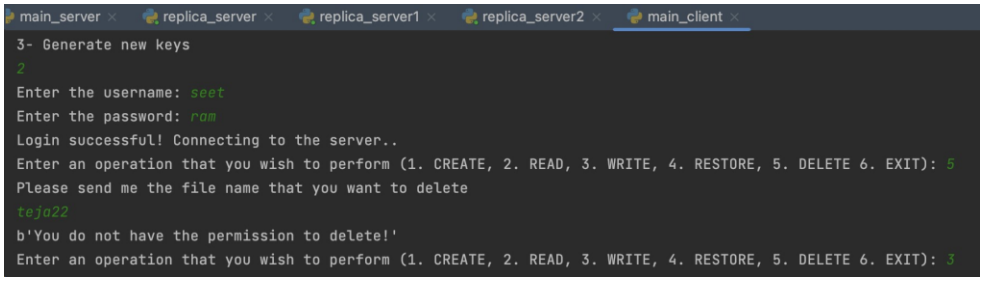
5.6 Users should not be able to modify files or directories without being detected unless they are authorized to do so

The initial step in this system, following connection establishment between the client and server, is user authentication. The client is prompted to provide their username and password, which are subsequently verified against the "users" SQL table at the backend. Upon successful authentication, the user is granted authorization to perform various operations within the system. However, if the user's credentials do not match the records in the database, they are denied access and presented with an error message stating that the username or password is incorrect. Alternatively, users who are not registered with the system can choose to register before attempting to perform any file operations.



5.7 A malicious file server should not be able to create or delete files or directories without being detected

Prior to performing any operations, each client must register with the server. During registration, the client's username and password are securely stored in the "users" database table. Before authorizing any operation, the server meticulously verifies the client's credentials against the database records. To bolster security, we have implemented an additional layer of protection in the form of digital signatures exchanged between the client and server. The client generates a digital signature for each action and sends it to the server, which verifies the signature's authenticity before authorizing the operation. This mechanism effectively safeguards the system against malicious operations initiated by unauthorized clients.



5.8 Incorporate a log into your file system to track each operation on your server. Using your knowledge from the first project, define features that will be used to detect attacks to your system. Generate logs with unauthorized access/modifications and logs with authorized access/modifications. Build your detector.

A logging mechanism has been implemented to track operations to improve the file system's security. Timestamps, usernames, actions (e.g., read, write, delete), and file names are all stored in a structured format in the log entries. Failed login attempts, unauthorized access, and unusual activity patterns are all attack detection features. An Isolation Forest model is trained on the log data using the scikit-learn library, and anomaly scores are generated for each operation. Anomaly scores which are negative indicate potentially malicious incident. The system is tested by generating logs with both authorized and unauthorized access/modifications, which serve as a foundation for training and evaluating the detection model. Once trained, the model can be

integrated into the file system to monitor and identify potential security threats based on logged activities in real-time.

	Timestamp	Message	Anomaly_Score
0	2023-11-27 13:54:06,350	User password was used / taken from database	0.066774
1	2023-11-27 13:54:06,350	Username %s has logged in succesfully & was no...	0.066774
2	2023-11-27 13:54:35,625	The user sai has tried to read the data in fil...	-0.044236
3	2023-11-27 13:54:53,336	The user sai has deleted the file tejnew	-0.044236
4	2023-11-27 13:55:22,908	The file seet was restored by user sai	0.067893
..	...	...	...
482	2023-12-03 16:29:00,346	The user sai has deleted the file saiteha	0.007652
483	2023-12-03 16:31:59,022	Total execution time for the operations is : 1...	0.103074
484	2023-12-03 16:33:08,478	User password was used / taken from database	0.100367
485	2023-12-03 16:33:08,480	Username kav has logged in successfully & was ...	0.024451
486	2023-12-03 16:33:12,615	The user kav has deleted the file teha	-0.001980

[487 rows x 3 columns]

5.9 Define the vulnerability of your detector. Use your knowledge of the first project to attack your detector.

Algorithm Bias:

One of the potential vulnerabilities of our detector is Algorithmic Bias. Algorithmic bias is the tendency of a machine learning model to produce results that systematically deviate from the truth due to imbalances or limitations in the training data. In the context of anomaly detection, the Isolation Forest may exhibit bias by either producing a higher rate of false positives (flagging normal activities as anomalies) or false negatives (failing to identify actual anomalies). Several factors contribute to algorithmic bias in the detector provided. The training data's quality is critical, as an unrepresentative or imbalanced dataset may result in a model that fails to generalize across diverse user behaviors and attack scenarios. The features used to train the model also have an impact on bias, emphasizing the importance of selecting comprehensive features that accurately capture the nuances of both normal and malicious system activities.

To reduce algorithmic bias in the provided malware detector, use representative and regularly updated training data, select comprehensive features, choose interpretable models, use continuous monitoring, and introduce diversity via model ensembles. Regular audits, as well as collaborative efforts between data scientists and cybersecurity experts, are required for ongoing bias identification and correction, ensuring the detector's effectiveness and fairness in detecting anomalies.

6. CONCLUSION

This P2P file system isn't just a file sharing tool; it's a fortress for your data, built on the bedrock of decentralization. Create, delete, read, write, and resurrect files with ease – all while enjoying a sleek interface that always displays the latest version. Want to keep things private? Granular permissions on files and directories become your castle walls, allowing only authorized eyes to peek inside. But security isn't an afterthought. Encrypted file names, directories, and even peer communication shield your data from prying eyes and meddling hands. This is your information, and this system ensures it stays yours. This isn't just code; it's a comprehensive platform for P2P file sharing. Functionality, security, and user control are intertwined, creating a haven of reliability for your data in the decentralized wilderness. It's the P2P file system you've been yearning for.

7. FUTURE SCOPE

- File Versioning
- Dynamic Peer Management
- Enhanced Security Measures
- Integration with Blockchain
- Graphical User Interface (GUI)
- Automated Backups
- Machine Learning for Anomaly Detection

Embracing these future enhancements, the P2P file system will morph into a powerful and feature-packed solution, perpetually adapting to user needs while staying firmly anchored in security and functionality.

8. REFERENCES

1. *Committee on National Security Systems National Information Assurance (IA) Glossary*. 2010.
2. "Kerberos (Protocol)." *Wikipedia*, 9 Nov. 2023, en.wikipedia.org/wiki?curid=16947.
3. Stallings, William. *Cryptography and Network Security : Principles and Practice : William Stallings*. Upper Saddle River, N.J., Pearson/Prentice Hall, 2006.
4. Tanenbaum, Andrew S, and Maarten Van Steen. *Distributed Systems : Principles and Paradigms*. Niederlande] Maarten Van Steen, 2016.
5. "Web Server." *Wikipedia*, 15 Nov. 2023, en.wikipedia.org/wiki?curid=33455. Accessed 4 Dec. 2023.
6. Celko, Joe. *Joe Celko's SQL for Smarties*. Elsevier, 22 Nov. 2010.