EFDS: THE ENCRYPTED AND DISTRIBUTED FILE SYSTEM

BY:

Chris Abili, Devon Slonaker, Jay Paun, and Yukta Medha

ABSTRACT

EFDS is an encrypted Python file system where our main focus is on providing a P2P method for users to store their data on untrusted distributed file servers. This design decision creates challenges related to communication security and system resilience against unwanted access and modifications, but it also opens up new possibilities for accessibility and scalability. We use a centralized directory approach to implement peer-to-peer file sharing. We enable dependable and efficient file sharing by centralizing directory information, which simplifies peer coordination and communication. We accomplished in detecting an unauthorized user modifying the file in the system.

INTRODUCTION

Secure file-sharing solutions are now essential for a wide range of uses, from enterprise-level collaboration to individual data storage, in the ever-changing world of information exchange and collaboration. But the increasing need for adaptability and accessibility has prompted researchers to look into cutting-edge solutions, like peer-to-peer (P2P) file-sharing networks. The goal of this project is to create a distributed data storage system that is both secure and reliable, by utilizing a P2P architecture. This paper aims to provide a detailed explanation of the design and implementation of the encrypted and distributed file system.

DESIGN AND IMPLEMENTATION

Our project is able to design and implement a secure Peer-to-Peer (P2P) file system that enables users to create, delete, read, write, and restore files while ensuring data integrity, confidentiality, and secure communication between peers. Key considerations include concurrent write and read operations, user-defined permissions, and robust detection of unauthorized modifications.

Our file system is a “central directory” P2P system and includes a directory server and clients. The directory server serves no other purpose other than to store files, keep track of file versions, and to store public keys. The clients do all of the heavy lifting in our system Most, if not all, of our user and directory server communication is done through sshpass, ssh, and rsync by the client. We have specifically chosen rsync over scp as it will not copy over entire files, but instead just append/delete lines in a file, or append/delete files in a directory that have changed.

The directory server and clients keep files in a directory called “files” on the desktop, and each file is stored in said directory under another directory entitled the same name as the file. Inside the directory with the same name as the file name on the directory server side only includes the version file, the permissions file, and files named the ip of users who currently hold the file. The versions file includes the version number, and who last updated the file. As for the client side, the directory with the same name as the file holds the version number, the file itself, and the permissions file (if you are the owner of said file).

The directory server also has a directory called “keys” which holds the public RSA keys for every user. The directory server also holds a directory called the “deleted” directory which is where deleted files are sent. We use this method of deletion instead of outright deleting the file because we want to ensure that the file can be restored at a later date.

Core Features:

a. File Operations: A user can search, create, download, create, write, read, delete, and restore files, among other basic file operations.

- Searching - A user can search a file in the file system using the -s flag in the command line. The client system will query the directory server for the file and will return whether or not the file can be found. Search is used in many operations in the file system to ensure whether or not a file exists before an action is performed on it.
- Create - A user can create a new file using the -c flag. When a user creates a new file, they will produce a directory with the same name as the file and store that in their “files” directory. Inside the directory with the same name as the file, the file will be generated (and will be blank), a version file will be created starting at version 1, and a permissions file will be created giving the owner read and write permissions to the file.
- Download - A user can download a file using the -d flag. If a user does not currently have the file on their system, and assuming they have the proper read permissions, they will tell any one owner of the file to pull the public key of the client from the directory server, encrypt the file with said public key, rsync the version file and the encrypted file to the user, and then the user will decrypt the file with their private key. If the user has an outdated version of the file through comparisons of their version file to the version file on the directory server, they will download the new version of the file in a similar manner as explained previously. If the user already has the latest version of the file, no download will happen.
- Write - Assuming a user has the proper write permissions and holds the file on their system, the user can write to a file using the -w and -m flags and will first compare their version number of the file to the directory server’s version of the file. If the user’s version is out of date, they will be prompted to pull the latest version of the file. If the user is up to date with the latest version of the file, they will first check for a “.mutex” file in the directory server’s directory of the file. If the mutex file is there, it tells the user that they cannot modify the file at this time. This is to prevent concurrent writes. If the “.mutex” file isn't there, the user will place a “.mutex” file of their own and update the contents of the file on their system, update the version file, copy the version file to the directory server, and purge all files that include previous holders of the file, as they now hold outdated versions of the file and should not be contacted by other users to download from them. Written data is never stored on the directory server. Only clients hold actual file information.

- Read - A user can read a file using the -r flag and will only be able to read a file locally on their system. This is to give the user more control over what version they are reading from. A user may intentionally not want to download a newer version of the file until some point later. At that point, they can pull and download the new version and properly read the new version.
- Delete - Assuming the user has the proper write permissions, they can delete the file using the -x flag. This will essentially move the directory on the directory server that is the same name as the file to the “deleted” directory.
- Restore - Assuming the user has the proper write permissions, they can delete the file using the -z flag. This performs the reverse of delete and will move the file from the “deleted” directory back to the “files” directory.
- Permissions - The owner of the file can grant permissions to the file using the -p flag. The system will check to see if the owner is the one attempting to make modifications because the system will check for a .permissions file in the directory of the file with the same name as only the file owner will have the permissions file for said file on their system.
- Key Generation - Using the -g flag, any user can reset their public and private keys (assuming there is a missing public key on the directory server or there is a missing public or private key on the client’s side).

b. Consistency Guarantee: Guaranteeing that users always view the most recent version of a file, shielding older versions from view after coming across a more recent version.

- When a user downloads some version of a file, there will be a version check to tell the client whether they are up to date or if they can download a new version of the file. This is thanks to our git-style version control where every write to a file is recorded on the directory server in the form of a version number. This allows someone who already owns the file to compare their version of the file to the latest version of the file as recorded by the directory server and then allows the user to pull the latest version from another client who also has the latest version of the file.
- This also prevents sequential write issues, meaning that a user cannot just override a newer version of a file with an older version of the file. The user must first download the latest file version, make their changes, and then write. Doing this will increment the version number of the file and purge all

users from the list of holders of the file who do not have the newly up-to-date version of the file.

c. User Permissions: Enabling users to modify the permissions of files and directories through the implementation of user authentication and authorization mechanisms.

- Using a “.permissions” file, a file owner can modify who can read and write files they own. The owner is identified because they are the only one who has the “.permissions” file in the directory with the same name as the file. Anyone who does not have access to this permission file cannot modify permissions for the file. If a user is not in the permissions file, they are immediately barred from reading or writing that file. The owner can add users to the permissions file though the -p flag when running the program. They will be able to select who to grant permissions to, what permission the user can be granted, and what file to grant the permissions to. The owner can make it so that a user can read, read and write, or the owner can revoke the permissions by using ‘.’ as a permission.

d. Concurrency Handling: - To improve system responsiveness and efficiency, support concurrent write and read operations.

- Concurrent read access is not of a concern as people who own the file will only be reading the file locally. They will not be able to read directly from another user’s system
- Concurrent downloads are handled by granting users uniquely named public keys on the uploader’s side, and the encrypted file names are also unique to the downloader. This ensures that there are no conflicting file names for multiple people who want to download the encrypted file at once.
- Concurrent write access is handled by a file called “.mutex” on the directory server in the directory with the file name so that as long as the mutex file exists, no one else can concurrently write to the file until the current writing user is done and lifts the mutex file.

e. Confidentiality: Sensitive data is protected by encryption, and file and directory names are handled as confidential information.

- Whenever a user downloads a file, the person the user is downloading the file from pulls the user's public key from the directory server. After that, the uploader will encrypt the file with the user's public key and send over the version file and the encrypted file. Once the user receives the encrypted file, they will decrypt the file to reveal the contents.
- File and directory names are handled as confidential. File names are not listed out or given away. If a user wants a file, they must know what the file is titled in advance in order to download or access it. This also ties along with the user permissions, as even if a user tries to access a file that does exist but they are not in the permissions file for the file they are trying to access, they will get an error saying the file doesn't exist. We feel that saying a file does not exist protects a file a lot more than simply saying that access to a specific file errored out. Saying it doesn't exist even when the file does exist to a user that does not have proper permissions will make the user less likely to try to get into the file some other way. If it is implied that the file does exist, just the user cannot access it, they may try to find other means of getting access to the file when they shouldn't be allowed to. If we tell a user that a file does not exist, a user would potentially be less likely to try to find ways to circumvent the access control.

f. Secure Communication: - Encryption of peer-to-peer communication channels to protect transmitted data from possible manipulation or eavesdropping.

- All transmitted files to and from clients are encrypted with the public keys of those clients that are stored on the directory server. For any one client, an uploader will pull their unique public key to allow the client to access and download the file (assuming these clients have the correct read permissions).
- File data is NEVER stored on the directory server. Only owners and holders of the file hold the data to that file, and as described above, these files are encrypted whenever they leave a user's system. This prevents manipulation of the data, and if an eavesdropper attempts to access the encrypted file, they will be unable to decrypt it without the downloader's private key which is only ever stored locally on the client's system.

TESTING

Test Case	Description	Result	Passed/Failed
TC-01	Create a new file	File is successfully created	Passed
TC-02	Delete an existing file	File is successfully deleted	Passed
TC-03	Read a file	User can successfully read the latest version of the file	Passed
TC-04	Write to a file	User can successfully write to the file	Passed
TC-05	Restore a file to a previous version	File is successfully restored to the specified version	Passed
TC-06	Set permissions on a file or directory	Permissions are set successfully	Passed
TC-07	Check if a client always sees the latest version of a file	The client sees the latest version or is notified that a newer version is available	Passed

TC-08	Concurrent write and read	System handles concurrent write and read operations without data corruption	Passed
TC-09	File and directory names treated as confidential	Names are encrypted and not visible to unauthorized users	Passed
TC-010	Encrypted communication between Peer to Peer	Communication is successfully encrypted	Passed
TC-011	Unauthorized modification of files or directories	Unauthorized modifications are detected and logged	Passed
TC-012	Malicious file server creating or deleting files or directories	Detection of malicious activities and prevention of unauthorized file/directory manipulations	Passed
TC-013	Log each operation on the server	Logs are generated for every operation on the server	Passed
TC-014	Detect unauthorized access/modifications	The system detects and logs unauthorized access or modifications	Passed
TC-015	Detect authorized access/modifications	The system logs authorized access or modifications	Passed

TC-016	Build and test the detector for attacks on the system	The detector successfully identifies and alerts on potential security threats.	Passed

The resulting numbers are how long it took X files to be read sequentially (measured in seconds, rounded to nearest hundredth of a second)

Sequential Read	Test 1	Test 2	Test 3	Avg
1 Read	0.70	0.69	0.62	0.67
10 Reads	7.02	6.64	6.88	6.85
100 Reads	68.01	70.59	75.51	71.37

The resulting numbers are how long it took X files to be written sequentially (measured in seconds, rounded to nearest hundredth of a second)

Sequential Write	Test 1	Test 2	Test 3	Avg
1 Write	4.65	4.73	4.53	4.64
10 Writes	43.04	42.78	44.07	43.30
100 Writes	453.47	460.69	456.30	456.82

The above table lists out the times for the testing of sequential reads and rights in x sequence. The results are clearly linear for both reads and writes. Assuming 1,000 reads, we can infer the total time it would take would be around ~700 seconds. Assuming 10,000 reads, we can infer the sequential read time would be around ~7,000 seconds. Assuming 100,000 reads, we can infer that the total read time would be around ~70,000 seconds, and so on. As for the sequential writes, if we assume 1,000 writes, we can infer this would take around

~4,500 seconds, for 10,000 writes around ~45,000 seconds, and 100,000 writes would take around ~450,000 seconds.

With this data, we can infer that x sequential reads will take around 70% of the time as the number of reads, and for the writes, we can infer that x sequential writes will take around 450% of the time as the number of writes. It is also interesting to point out that reading any single file will be around ~6.5 times faster than writing to a file. The reason behind the disparaging gap in the times between read and write largely has to do with the fact that reads of a file are strictly local, and writing to a file requires version checking, mutex checking, and the modification of both local and remote files.

CONCLUSION

In conclusion, our project successfully designed and implemented a secure Peer-to-Peer (P2P) file system that prioritizes user-centric functionalities while maintaining robust security measures. We chose rsync over scp for better file updates, reducing unnecessary data transfer and boosting performance. The directory server and clients work together efficiently, managing files, tracking versions, and handling user permissions. Our system, like a digital guard, ensures your files stay private and protected, acting as a lock for your digital door in a time when privacy is rare.

REFERENCES

1. https://www.geeksforgeeks.org/p2p-peer-to-peer-file-sharing/
2. https://arxiv.org/abs/1710.09435
3. https://github.com/uvasrg/EvadeML
4. https://www.computerworld.com/article/2588287/networking-peer-to-peer-network.html
5. https://www.youtube.com/watch?v=VKSIacce9QQ
6. https://www.researchgate.net/figure/A-centralized-P2P-overlay-network-The-central-directory-server-keeps-record-of-all_fig16_271823049