CMSC 626 PROJECT REPORT

TOPIC – ENCRYPTED DISTRIBUTED FILE SYSTEM

Haravind Rajula (HRAJULA1), Abhishek Chintalapati (ABHISHC1), Yashwanth Saladi (UX96332), Sai Sandeep Ravuri (SRAVURI1), Madhumitra Maladi (LU83139)

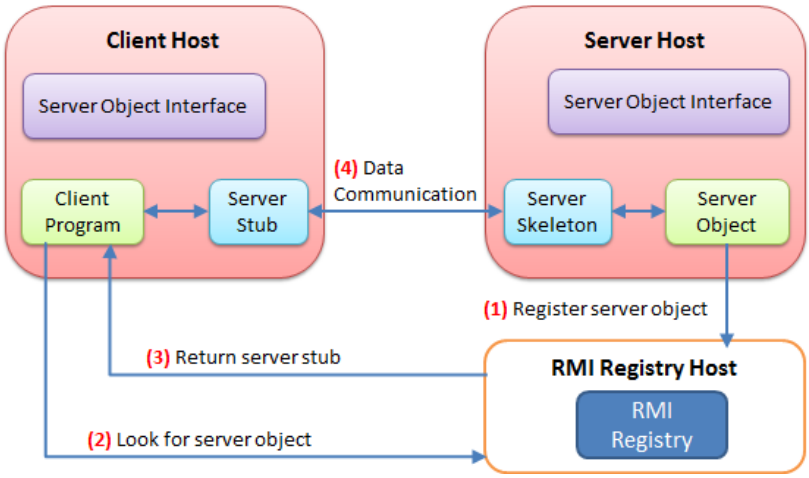
Overview

In this project, we are utilizing the RMI library and the AES algorithm to create a simple distributed and encrypted filesystem in this project. One or more remote storage servers will host the files. The files will be indexed separately by a single naming server, identifying where they are stored. If a client wants to access a file, it must first contact the naming server to obtain a stub for the file's storage server. The procedure is then finished by interacting with the storage server directly.

File reading, writing, creation, deletion, and renaming will all be supported by the filesystem. It will also do directory listing, creation, renaming, and deletion operations. File locking will be possible, and files that are frequently accessed will be replicated over different storage servers.

Detailed Description

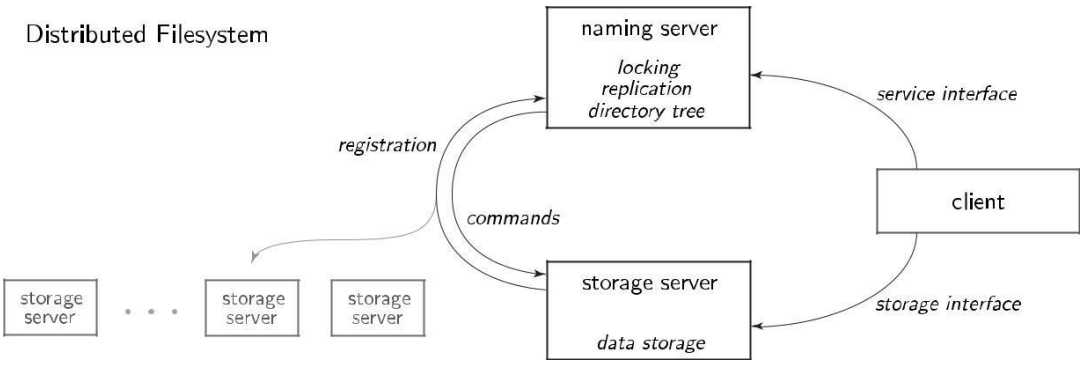
There are several important components that make up the filesystem. The RMI library is the first of these. The RMI library is used to communicate between servers and clients, as well as between naming and storage servers in the filesystem. The RMI library is found in the RMI package.



The main purpose of storage servers is to offer clients with file data access. Clients connect to storage servers to read and write files. Clients almost never have direct access to storage servers. Instead, clients' views of the filesystem are defined by the filesystem's single naming server, for which they have an RMI stub. The naming server keeps track of the filesystem directory tree and links each file to a storage server. When a client wishes to conduct an operation on a file, it first contacts the naming server to get a stub for the file's storage server, and then performs the operation using the stub. Storage servers can also use naming servers to record their presence.

The client interface lets clients to perform file operations on the storage server, while the command interface allows the naming server to give file management commands to the storage server. File reading, writing, renaming, deleting, and creating and removing directories are all possible with the Client interface. It allows the naming server to request that a file on the storage server be created, deleted, or copied from another storage server as part of replication. The storage server's role is to simply react to these requests. These requests may arrive simultaneously.

A question arises: where does the data for the files hosted by the storage server reside? The storage server must put all its files in a directory on the machine it runs on, referred to as the storage server's local or underlying filesystem, in this arrangement. This directory's structure should match the distributed filesystem's structure as perceived by the storage server. It is not necessary for a storage server to keep anything for a file in the filesystem if it is unaware of its existence (due to another storage server hosting the file). This approach makes data persistence between storage server restarts straightforward.



Because each server in the filesystem is thread-safe, performing multiple actions on the same server at the same time will never cause the server's state to become inconsistent. The consistency criteria throughout the entire filesystem, on the other hand, are significantly less stringent. The design is delicate, and it is reliant on well-behaved customers.

When the storage and name servers are implemented, a suitable time is chosen for the naming server to advise each storage server to add or remove files or directories, keeping the servers' representations of the filesystem constant. For example, a file that has been requested to be deleted by the name server should not be visible to subsequent queries to the storage server.

Features and how they are used:

1. Users who want to execute CRUD operations on files must first register with the application, after which they will be granted a private key with which to log in and do the CRUD actions. Each user will have their own AES key, which will be encrypted with their public key and decrypted with their private key. We take the user's secret key, obtain the original AES key, and then perform encryption/decryption while doing CRUD activities.
2. Users can control the permissions and access levels to the files they possess. The user can set the permission level of each file he or she saves in the system to public or private. Private files are accessible only by the user, but public files are accessible by any user of the application. The user will be able to adjust the permissions for individual files using the user interface.
3. The file system can support directory structures. When generating the file, users must input the absolute locations of the files. All the files will be stored in a root directory. For example, the user can make two files: Sample.txt and Directory1/Sample.txt.
4. The names of the files and directories are encrypted and kept private. The user's AES key created during sign-on is used to encrypt all file names, folders, and paths. The information is subsequently saved on the file server. They can only be decrypted at the request of the user, and the user's secret key is used to decrypt the AES key before decrypting the encrypted data.
5. Users should not be able to change or remove files without permission, and they should be logged. Before allowing users to modify or remove files, we authenticate them using their

secret key and then authorize them to do so after authentication. All CRUD operations are logged in the database and can be tracked with the help of a logger.

6. The files should not be corruptible by unauthorized users. Because we authenticate users using their private key before enabling them to proceed, unauthorized users cannot access or corrupt other people's data in the files.
7. Only the client with permission to access the file should be served by the server. Every user can see the public files, but only the owner can change or remove them because we have two levels of file access. Only the owner of a private file can see it, whereas recipients can only see shared files. This is possible because we employ private keys to authenticate and verify users.
8. The user should not be able to view data that he has not requested or is not permitted to see. When a user chooses to examine a file, only that file's data is displayed. And with help of authentication user will be shown only his private files and cannot see the data of other users.
9. Without being identified, the rogue server should not be able to create/delete files. With the use of secret keys and usernames, every UI operation is user driven. All actions are started after successful authentication and permission. As a result, a malicious server must first gain the users' private keys before it can create or delete files.
10. The files can be transmitted between any two network users. A file can be transferred from one user to another on the network. The receiver will see and be able to access a copy of the file.
11. Users should be able to exchange files on the server with other individuals, but unauthorized users should not be able to access those files. A file can be transferred from one user to another on the network. The receiver will see and be able to access a copy of the file. To achieve this and feature number 10, we will create a new AES key for sharing, then encrypt and save the file using that new AES key. The sender will then sign the new AES key and encrypt it with the recipient's public key. Only the receiver can see the decrypted data in this method, and he or she may be confident that it originated from the recipient because of the signature. And no other user can access the file since the recipient's secret key is required.