HEIG-VD - CAA

Projet

Christophe Roulin

Contents

1 Description of the project

1.1 Project Overview

The project aims to develop a robust and secure shared encrypted network file system. The system will prioritize user access, confidentiality of file and folder names, protection against active adversaries, and efficient sharing mechanisms while ensuring usability and ease of interaction.

1.2 Project Goals and Objectives:

- **User Authentication:** Implement a user-friendly username/password authentication system requiring minimal logins for seamless user experience.
- **Security Measures:** Ensure robust protection against active adversaries while maintaining confidentiality of file and folder names.
- **Trust Model:** Assume an honest but curious server scenario, focusing on safeguarding data from potential breaches.
- **Device Flexibility:** Enable users to access the file system from various devices effortlessly.
- **Sharing and Access Control:** Facilitate folder sharing among users and implement access revocation.
- **Password Management:** Enable users to securely change their passwords.

1.3 Features and Functionalities:

The system will provide the following key functionalities:

- **Secure File Operations:** Support secure downloading and uploading of files within the system.
- **Folder Management:** Allow users to create folders and manage their structure securely.
- **Sharing Mechanism:** Enable users to share folders securely with other authorized users.
- **Access Revocation:** Implement a secure process for revoking access to shared folders.
- **Password Change:** Provide a secure mechanism for users to change their passwords.

2 System modelling

2.1 User management

2.1.1 User registration

The server will be responsible for user registration while most of the operations will be done by the client locally.

The registration process is as follows:

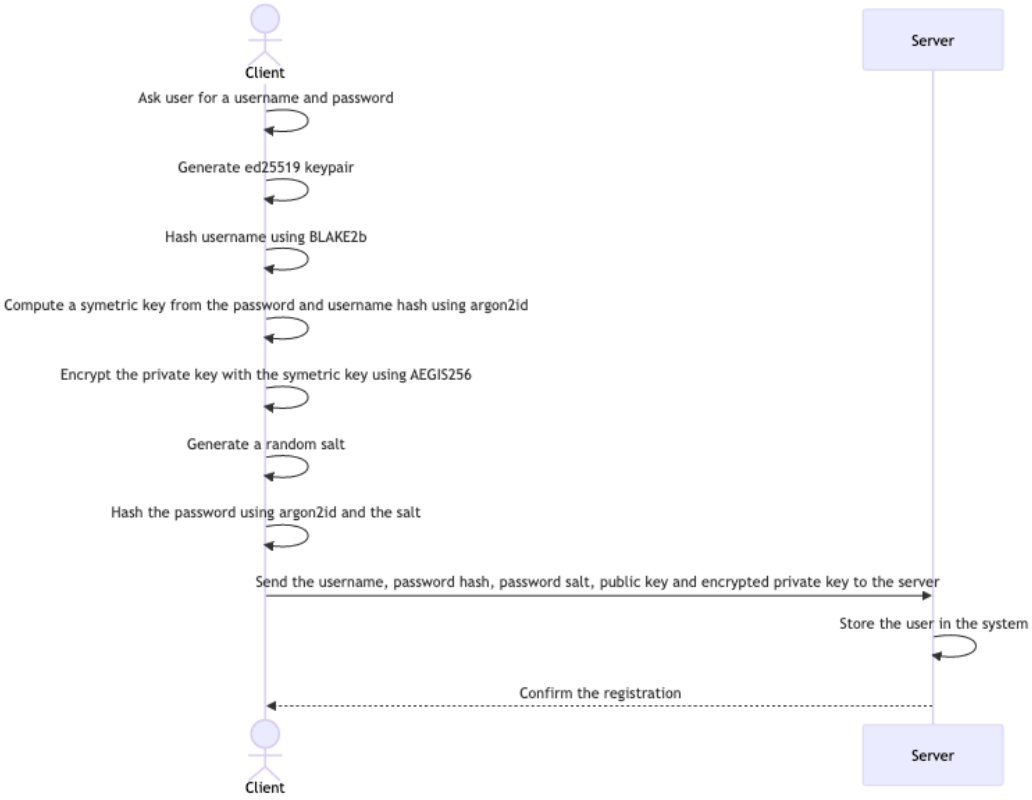

Figure 1: User registration

After the registration, the client will have to create a root folder for the user. This is described in the `Root folder` section.

2.1.2 User login

The login process is as follows:

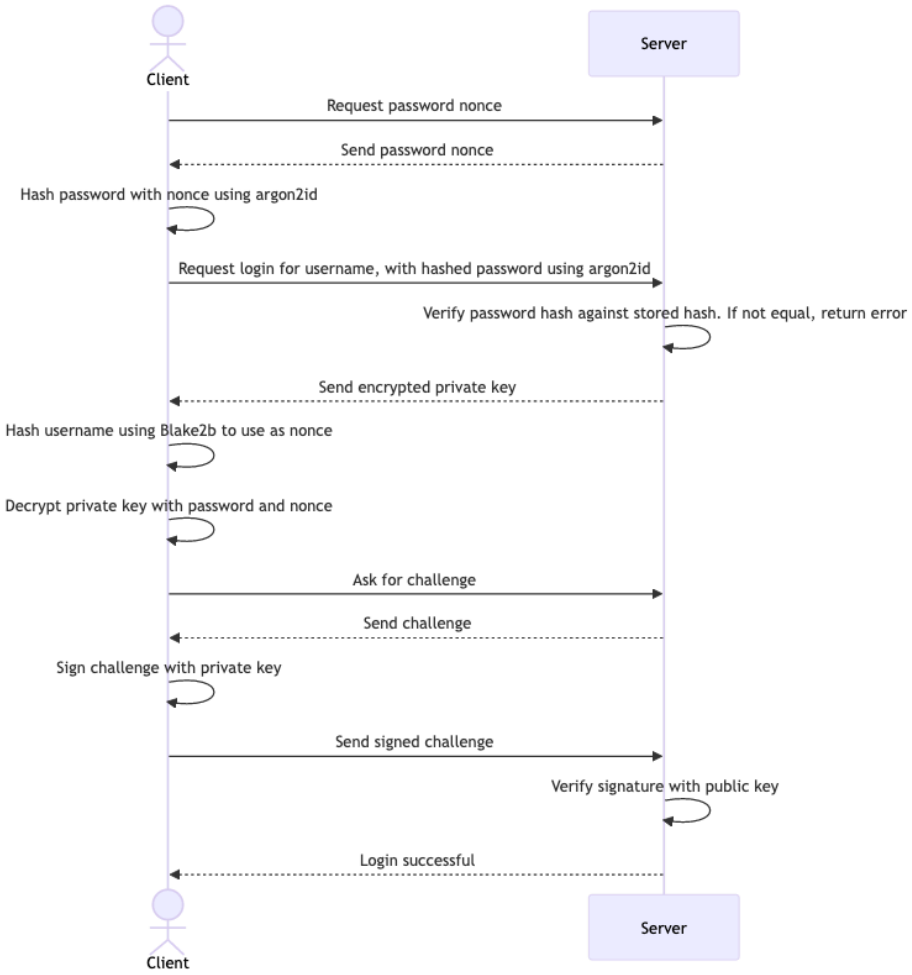

Figure 2: User login

2.1.3 Users list

The server will maintain a list of registered users. Anyone can query the server for the list of users. The server will then return the list of users. Anyone can ask for any user's public key.

2.2 Files and folders

The system will be based on a tree structure. Each user will have a root folder that will contain all of the user's files and folders. Each element inside a folder will have a key derived from the folder's key. The key derivation process will be done using a key derivation function (KDF), argon2id. This will allow us to easily share a folder with another user by simply sharing the folder's key.

2.2.1 Metadata

Each file and folder will have a metadata file that will contain the following information:

- **Nonces**
- **Encrypted key**
- **Encrypted name**
- **Sharing list**: The sharing list will contain the list of users that have access to the file or folder.

2.2.2 Root folder

The root folder is a special folder that is created when a user registers. It is the only folder that has a key that is not derived from the folder's parent key since it has no parent.

The root folder creation process is as follows:

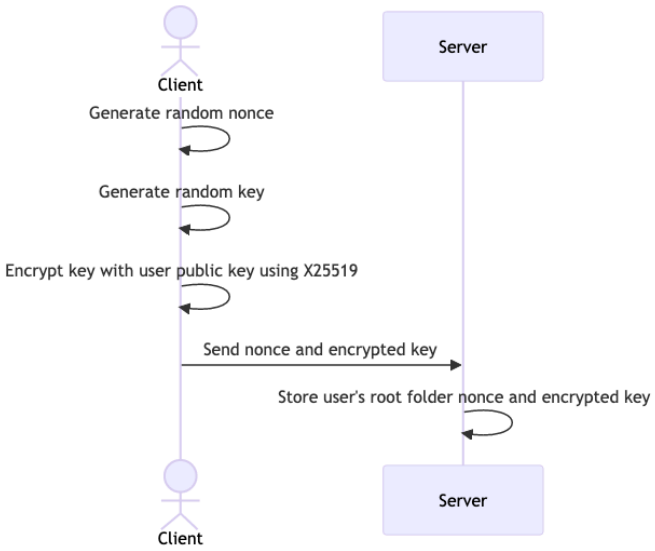

Figure 3: Folder creation

2.2.3 Folder creation

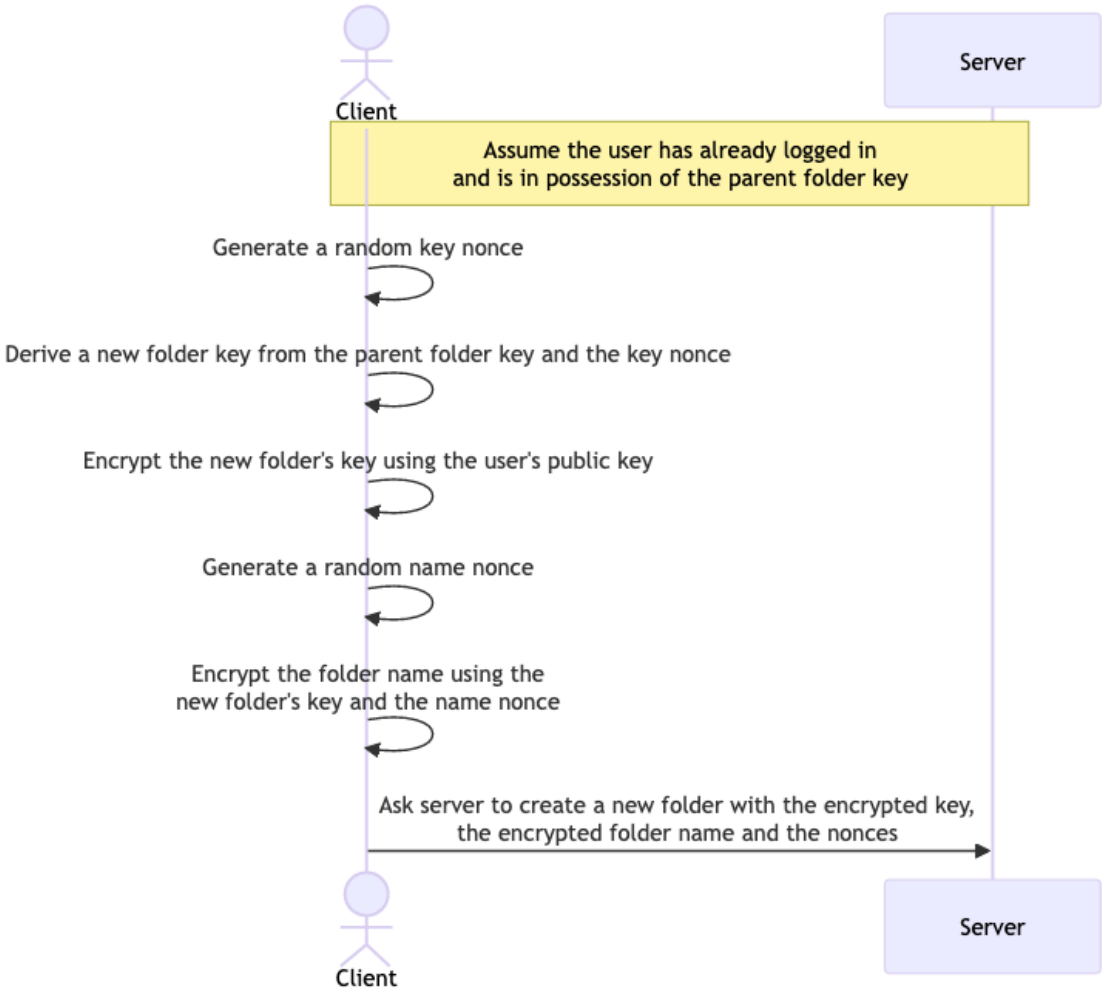

Figure 4: Folder creation

2.2.4 File creation

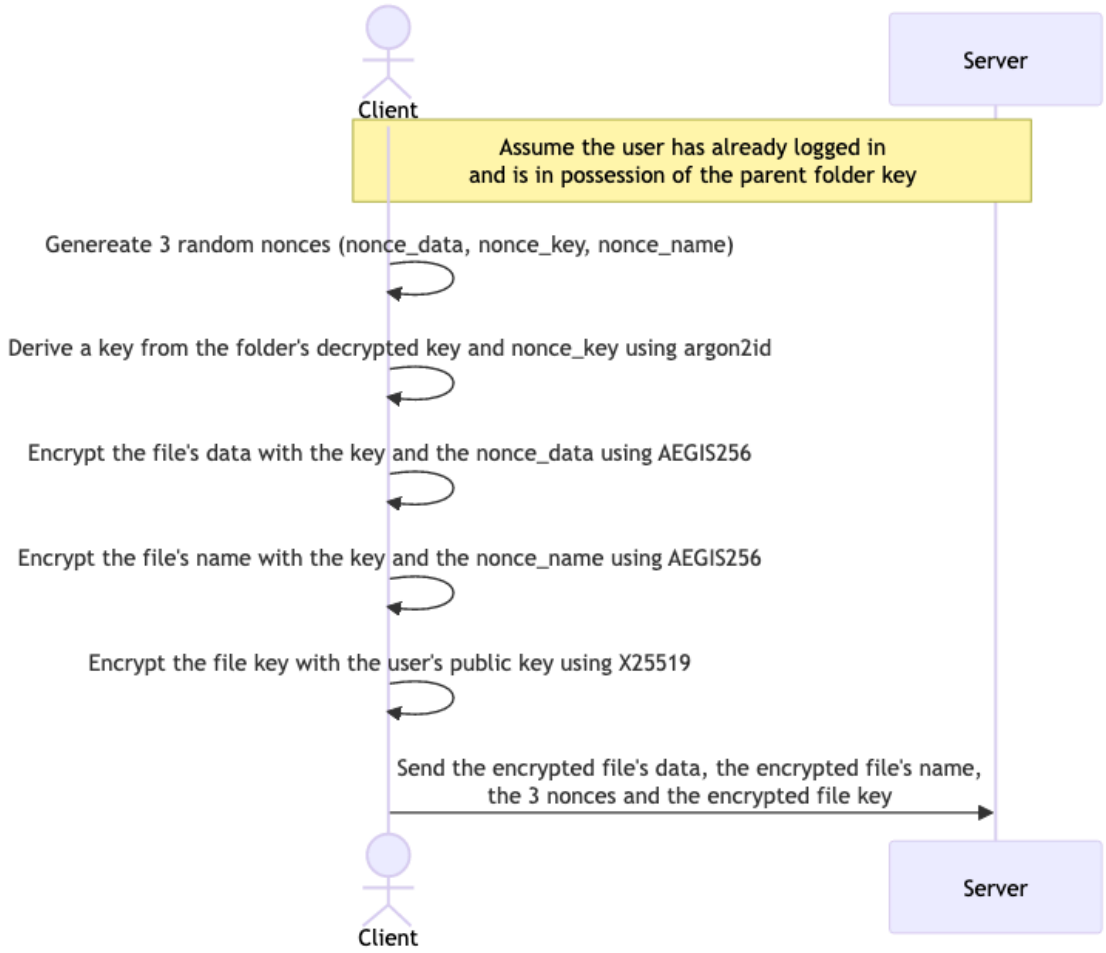

Figure 5: File creation

2.2.5 Access files and folders

The process of accessing a file or folder is as follows:

2.3 Access root folder

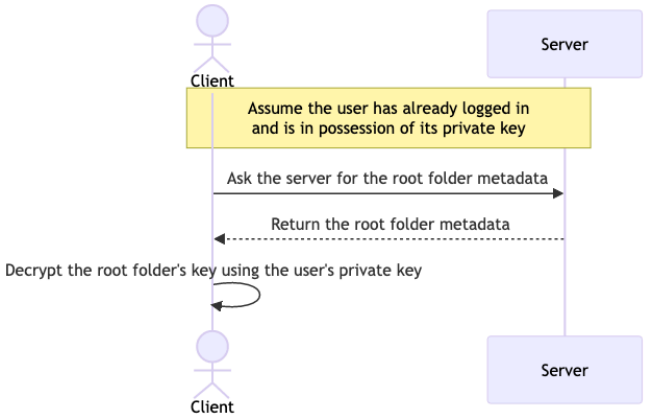

Figure 6: Accessing the root folder

2.4 Access folder

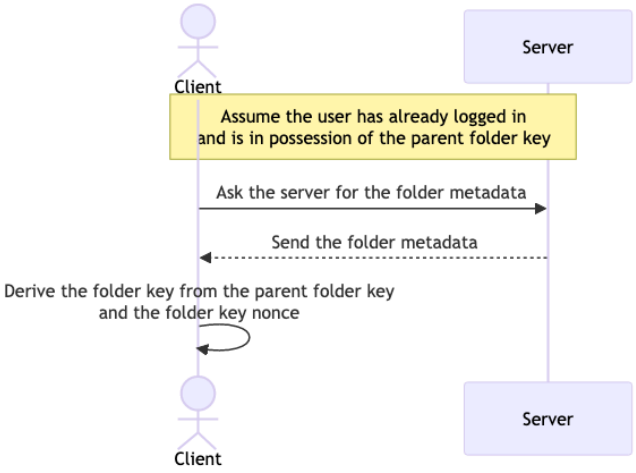

Figure 7: Accessing a folder

2.5 Access file

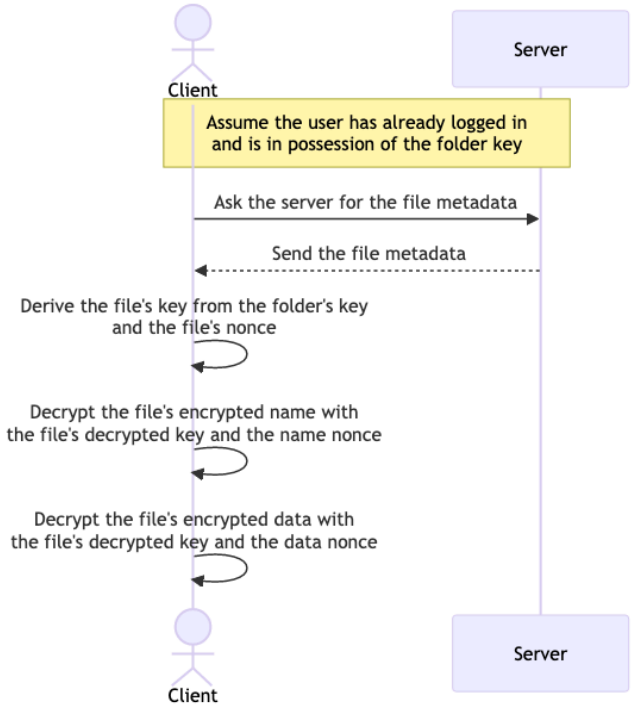

Figure 8: Accessing a file

2.5.1 Traversing and cache

Accessing a specific folder will be possible only if the clients knows the encrypted path of the folder (since the folders names are encrypted and the server don't know the "real" names).

The client will cache the encrypted path of the folders that the user has accessed. This will allow the client to access the folders without having to traverse the whole tree. If the client does not have the encrypted path of a folder, it will have to traverse the tree to find the folder, decrypting each folder's name along the way.

If a folder is renamed or moved, the client will have to traverse the tree again to find the folder. This is because the encrypted path of the folder will have changed. The general idea is that the client will try to traverse the tree if the direct access throws an error.

2.6 Sharing

2.6.1 Folder sharing

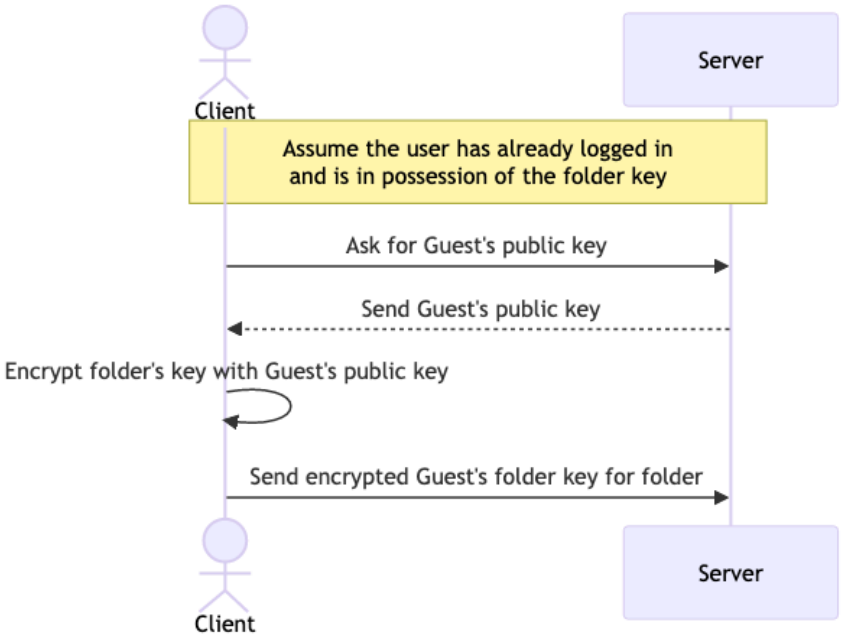

Figure 9: Folder sharing

2.6.1.1 Root folder special case

The root folder cannot be shared or revoked. If a user wants to share their root folder, they will need to create a new folder inside their root folder and share that folder instead.

2.6.2 File sharing

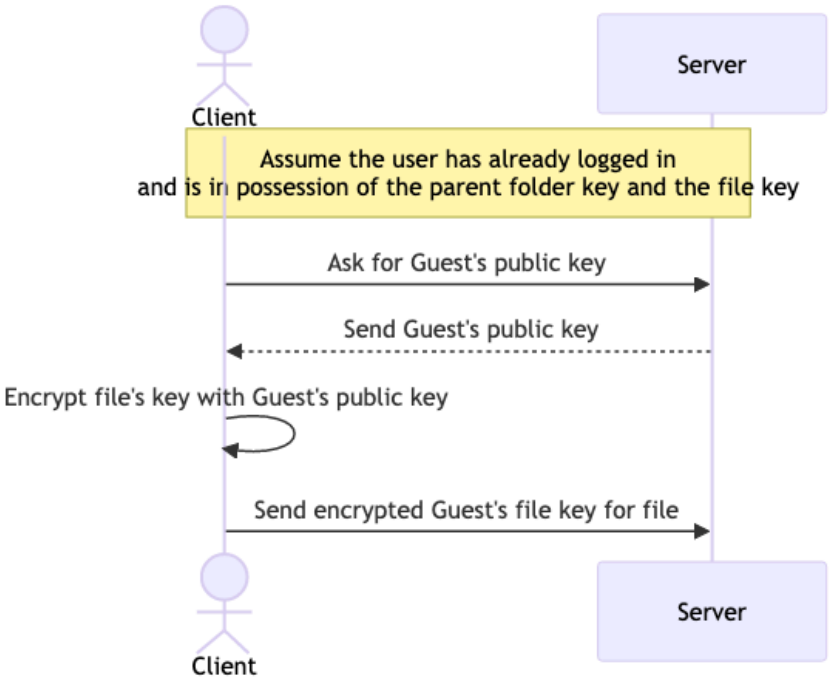

Figure 10: File sharing

2.7 Revoking access

The process of revoking access to a file or folder is as follows:

2.7.1 Folder

Note: The revokation process is done recursively. If a folder is revoked, all of its subfolders and files keys need to be updated as well. If any subfolder or file was shared with another user, we'll need to send the server the new keys for those files and folders for those users to be able to access them.

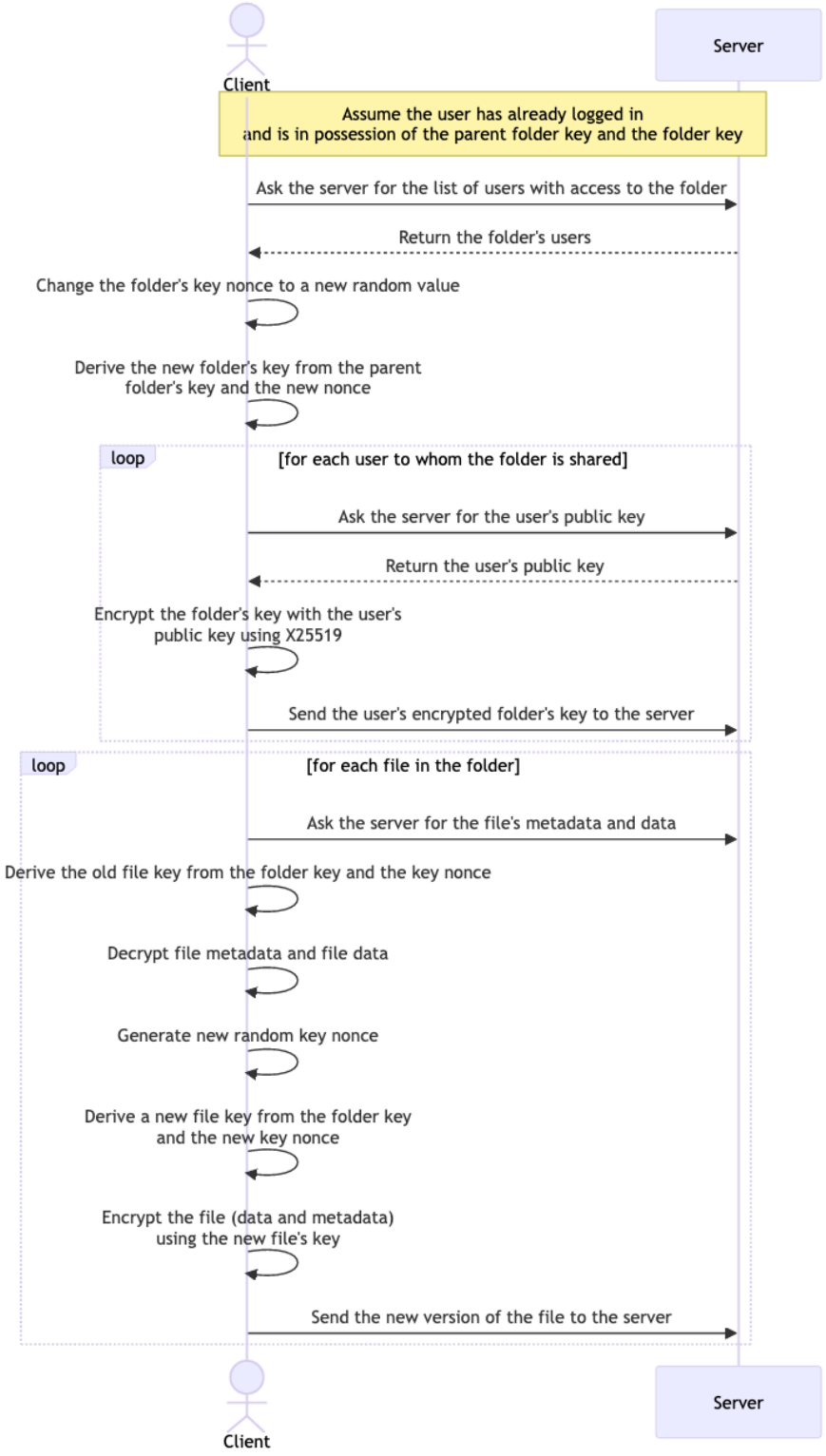

Figure 11: Folder revokation

2.7.2 File

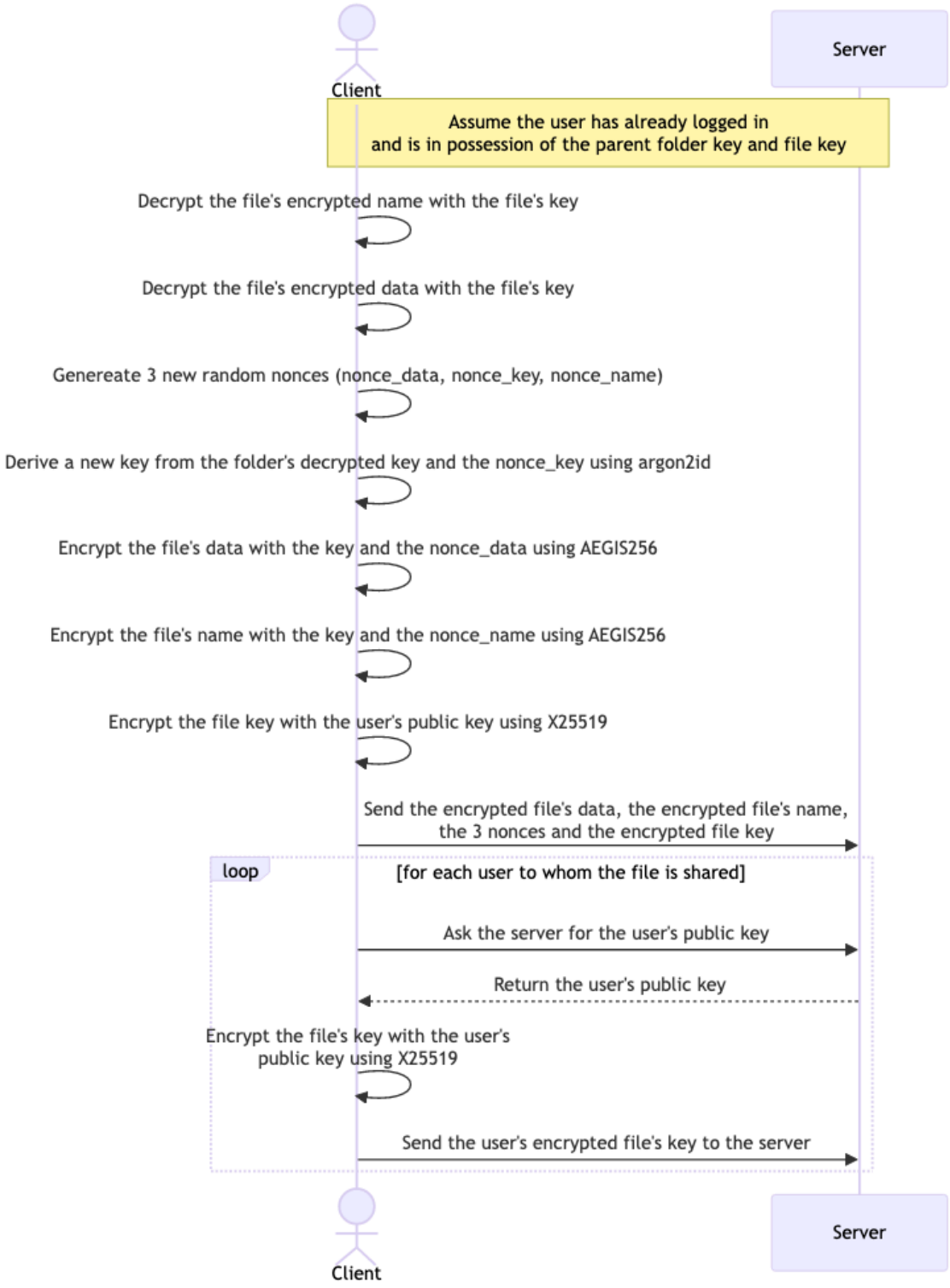

Figure 12: File revokation

2.8 Server role

In our model, the server is only acting as a storage server. It does not have access to the user's private keys and cannot decrypt any of the user's files or folders. The server is only responsible for storing the user's files and folders and for sharing the user's public keys with other users.

The server is also responsible of verifying each user identity when they register or login. This to ensure that no one tries to update something that he's not allowed to.

2.9 Client-Server communication

TLS 1.3 will be used for client-server communication. The server will be authenticated using a valid X. 509 certificate. The client will verify the server's certificate and will only communicate with the server if the certificate is valid.

When connecting to the server, the client will ask the user for the server's hostname. The client will then try to connect to the server using the hostname and will verify that the server has a valid certificate for the hostname.

The client has a trusted certificate that it will use to verify the server's certificate. If the file is a CA, the client will use it to verify the server's certificate. If the file is not a CA, the client will use it as the server's certificate.

2.10 Threat model

The following are the various parts of our threat model:

- **Honest but curious server.** The server is assumed to be honest but curious. This means that the server will not try to attack the system but will try to learn as much as possible about the system. We can assure a perfect security against this type of adversary by encrypting everything that is sent to the server. This includes the user's files, folders, and metadata. The server cannot decrypt any of the user's data since it does not have access to the user's password. The only added capabilities that a passive adversary installed on the server would have over anyone else is the ability to bruteforce the user's password without any limitations. This is a negligible threat since the user's password is hashed using argon2id with a moderate memory and ops limit. This means that the adversary would need to spend a lot of resources to bruteforce the user's password.
- **Active adversary.** An adversary that can intercept, modify, and inject messages. We consider this adversary to be installed in between the client and the server. We are protected against this type of adversary by using TLS 1.3 for client-server communication. This ensures that the adversary cannot intercept, modify, or inject messages.
- **Stolen client device with closed session.** We assume that the client device can be stolen if the user is logged out. This is possible because once a session is closed and the user logs out, nothing stays on the device.
- **Access to server files after sharing revoked.** We assume that the server files can be accessed by user2 after user1 revoke user2 access to a file or folder even if user2 stored the folder or file key. This is because we will re-encrypt the file or folder (and everything under it) with a new key when the access is revoked. This means that the old key will not be able to decrypt the file or folder anymore.

The following are not considered part of our threat model:

- **Stolen device while openned sessions.** We do not consider the case where the device is stolen while the user is logged in. This is because the user's private key is stored in config and can be accessed by anyone with access to the device.

2.11 Storage overhead

Here's a way to compute the storage overhead of a complete system (counting overhead only, not counting the size of original data and original name).

2.11.1 Base

$$\text{nonce} = 32B$$

$$\text{aegis256 key} = 32B$$

$$\text{aegis256 encryption} = \text{size} + 32B$$

$$\text{ed25519 private key} = 64B$$

$$\text{ed25519 public key} = 32B$$

$$\text{x25519 encryption} = \text{size} + 48B$$

2.11.2 Root folder

$$1 \text{ nonce} = 32B$$
$$1 \text{ X25519 encrypted key} = 80B$$

$$\text{total} = 112B$$

2.11.3 Folder

$$2 \text{ nonces} = 64B$$
$$1 \text{ aegis256 encrypted name} = \text{name} + 32B$$

$$\text{total} = 96B + \text{size of name}$$

The overhead for a folder is 96 Bytes.

2.11.4 File

$$3 \text{ nonces} = 96B$$
$$1 \text{ aegis256 encrypted name} = \text{name} + 32B$$
$$1 \text{ aegis256 encrypted data} = \text{data} + 32B$$

$$\text{total} = 160B + \text{size of name} + \text{size of data}$$

The overhead for a file is 160 Bytes.

2.11.5 Sharing

For each file or folder that is shared with another user, we need to store the encrypted key for that user. This means the following overhead for each file or folder that is shared:

$$1 \text{ x25519 encrypted key} = 80B$$

$$\text{total} = 80B$$

2.11.6 Total

The total overhead (in bytes) for a complete system can be computed using the followin formula :

$$\text{size} = (\text{users} * 112) + (\text{folders} * 96) + (\text{files} * 160) + (\text{shares} * 80)$$

With

- `users` beeing the number of users registered in the system
- `folders` beeing the number of folders in the system
- `files` beeing the number of files in the system
- `shares` beeing the number of files and folders that are shared by a user with another user

3 Cryptographic choices

3.1 Key pairs

The system uses the curve25519 elliptic curve for key pairs.

They are used in two different shapes:
- **X25519:** Used for asymmetric encryption.
- **Ed25519:** Used for digital signatures.

3.1.1 Key pair storage

All key pair parts are stored in Ed25519 format. This format allows us to easily get the X25519 key pair from the Ed25519 key pair.

3.2 Data hashing

Hashing data is done differently depending on the type of data.

3.2.1 General data

For non-sensitive data, we use the `crypto_generichash` libsodium function. This function is a BLAKE2b hash function.

This function is used to hash the following data:
- **Username** to use as nonce for the user key pair encryption.

3.2.2 Passwords

Sensitive data is hashed using the `crypto_pwhash` libsodium function. The function is configured to use the argon2id1.3 algorithm. The parameters used are:
- **Memory limit:** `crypto_pwhash_MEMLIMIT_MODERATE`
- **Ops limit:** `crypto_pwhash_OPSLIMIT_MODERATE`

Configured with theses parameters, the function takes about 1 second to hash a password on a modern computer. This is a good tradeoff between security and usability.

This function is used to hash the following data:
- **Password** to store on the server and act as a verification layer before sending the user's encrypted private key.

3.3 Key derivation

Key derivation is done using the `crypto_pwhash` libsodium function. The function is configured to use the argon2id1.3 algorithm.

Parameters used change depending on the type of data that we want to derive a key from.

3.3.1 Low entropy data

We consider here low entropy data to be anything coming from the user. In our case, mainly the user's password.

For low entropy data, we use the following parameters:
- **Memory limit:** `crypto_pwhash_MEMLIMIT_MODERATE`
- **Ops limit:** `crypto_pwhash_OPSLIMIT_MODERATE`

Configured with theses parameters, the function takes about 1 second to derive a key on a modern computer. This is a good tradeoff between security and usability.

3.3.2 High entropy data

We consider here high entropy data to be anything that is not coming from the user. In our case, mainly any existing key.

For high entropy data, we use the following parameters:
- **Memory limit:** `crypto_pwhash_MEMLIMIT_MIN`

- **Ops limit:** `crypto_pwhash_OPSLIMIT_MIN`

This allows us to derive a key quickly from an existing key. This is very useful here since we'll need to derive **many** keys to do any operations on the user's files and folders.

3.4 Data encryption

3.4.1 Symmetric encryption

Symmetric encryption is done using the `crypto_aead_aegis256` libsodium function. This function is an authenticated encryption function. It has the following advantages over the standard AES-GCM encryption see official documentation. The main advantage for us is that it allows us to use random nonces, greatly simplifying the encryption process.

3.4.2 Asymmetric encryption

Asymmetric encryption is done using the `crypto_box_seal` libsodium function. We use the user's public key (X25519) to encrypt the data.

3.5 Data signing

We use signature only to verify the user's identity when they login using a challenge-response process. We use the `crypto_sign_detached` libsodium function to sign using the user's private key (Ed25519).

3.6 Random data

Random data is generated using the `randombytes_buf` libsodium function. This function is cryptographically secure and is used to generate nonces and random keys.

4 System architecture

4.1 Session handling

A session is created for each user when they login. The session is used to remind ourself that the user is logged in and authenticated. The session is closed when the user logs out.

This is done by sending the client a session token when the user logs in. The client will then send this token with each request to the server. The server will then verify the token and will only respond to the request if the token is valid.

4.2 Protocol design

Data between the client and sever is sent in JSON format. Each request will contain the following fields:

```
{
  "session_token": "session token", # Optionnal, not needed for login and register
  "request": "request type",
  "data": "request data"
}
```

4.2.1 Request types

The request type can be one of the following

4.2.1.1 Register

- **register_user:** Used to register a user. The data field will contain the username, encrypted private key and public key.
- **create_root_folder:** Used to create the user's root folder. The data field will contain the encrypted root folder's key and the key's seed.

4.2.1.2 Login

- **get_user_password_salt:** Used to get a user's password salt. The data field will contain the username.
- **prepare_login:** Used to prepare a login. The data field will contain the username and the hashed password. The server will then send the user's encrypted private key if the hashed password is correct
- **login:** Used to login a user. The data field will contain the username. The server will then send the challenge for the user and a temporary session token.
- **verify_login:** Used to verify a login. The data field will contain the session token and the challenge response. The server will then send the final session token if the challenge response is correct.

4.2.2 User and sessions management

- **logout:** Used to logout a user. The data field will be empty.
- **change_password:** Used to change a user's password. The data field will contain the new encrypted private key, the new hashed password and the new password salt.

4.2.2.1 Users

- **get_users:** Used to get the list of users. The data field will be empty.
- **get_user_public_key:** Used to get a user's public key. The data field will contain the username.

4.2.2.2 Files and folders

- **create_folder:** Used to create a folder. The data field will contain the folder's metadata and the parent folder's path.
- **create_file:** Used to create a file. The data field will contain the file's metadata and the parent folder's path.
- **get_file:** Used to get a file's content. The data field will contain the file's path.
- **get_folder:** Used to get a folder's content. The data field will contain the folder's path.
- **list_folder:** Used to list a folder's content. The data field will contain the folder's path.

- **rename_file:** Used to rename a file. The data field will contain the file's path and the new file's encrypted name and linked seed.
- **rename_folder:** Used to rename a folder. The data field will contain the folder's path and the new folder's encrypted name and linked seed.
- **delete_file:** Used to delete a file. The data field will contain the file's path.
- **delete_folder:** Used to delete a folder. The data field will contain the folder's path.

4.2.2.3 Sharing

- **share_folder:** Used to share a folder. The data field will contain the folder's path, the user to share the folder with and the encrypted folder's key. If the folder is already shared with the user, the encrypted folder's key will be updated.
- **share_file:** Used to share a file. The data field will contain the file's path, the user to share the file with and the encrypted file's key. If the file is already shared with the user, the encrypted file's key will be updated.

4.2.2.4 Revoking access

- **revoke_folder:** Used to revoke a folder. The data field will contain the folder's path and the user to revoke the folder from. This will have the effect of removing the user's encrypted folder key.
- **revoke_file:** Used to revoke a file. The data field will contain the file's path and the user to revoke the file from. This will have the effect of removing the user's encrypted file key.

4.3 Represenation of data

4.3.1 User

Users are stored in the server memory and saved to disk when the server is stopped.

A user is represented by the following JSON object:

```
{
  "p_hash": "",   # Hashed password
  "p_salt": "",   # Password salt
  "b64_pk": "",   # Base64 encoded public key
  "e_b64_sk": ""  # Base64 encoded encrypted private key
}
```

4.3.2 Folder

A folder is represented by a folder in the data directory of the server. The folder's name is the folder's base64 encoded encrypted name (replacing '/' with '&'). Inside each folder is a metadata file (metadata.json) representing the folder's metadata.

The folder's metadata is represented by the following JSON object:

```
{
    # Key
    "b64_seed_k": "", # Base64 encoded seed key
                      # (used to derive the folder's key from the parent folder's key)
    "e_b64_key": {    # Base64 encoded encrypted key for each user that has direct access to
the folder
        "user1": "",
        "user2": "",
    },

    # Name
    "b64_seed_n": "", # Base64 encoded seed name
                      # (used with the folder's key to encrypt the folder's name)
    "e_b64_name": "", # Base64 encoded encrypted name
}
```

4.3.3 File

A file is represented by two files in the data directory of the server. The first file is the file's content. The second file is the file's metadata (name.metadata.json).

As for a folder, the name of the file is the file's base64 encoded encrypted name replacing '`/`' with '`&`'.

The file's metadata is represented by the following JSON object:

```
{
    # Key
    "b64_seed_k": "", # Base64 encoded seed key
                      # (used to derive the file's key from the parent folder's key)
    "e_b64_key": {    # Base64 encoded encrypted key for each user that has direct access to
the file
        "user1": "",
        "user2": "",
    },

    # Name
    "b64_seed_n": "", # Base64 encoded seed name
                      # (used with the file's key to encrypt the file's name)
    "e_b64_name": "", # Base64 encoded encrypted name

    # Data
    "b64_seed_d": "", # Base64 encoded seed data
                      # (used with the file's key to encrypt the file's content)
}
```

5 How to run the project

5.1 Requirements

5.1.1 Server

- **Python:** Version 3.10 or higher to run the server
- **Pip:** To install the server's dependencies
- **docker (optional):** To run the server using docker
- **docker-compose (optional):** To run the server using docker

5.1.2 Client

5.1.2.1 Tools

- **CMake:** To build the project
- **Make:** To build the project
- **C++ compiler:** Supporting C++23

5.1.3 Libraries

- **libsodium:** To handle all of the cryptographic operations (https://libsodium.gitbook.io/doc/installation)
- **nlohmann_json:** Version 3.11.3 or higher to handle JSON data (https://github.com/nlohmann/json)
- **restclient-cpp:** To handle HTTP requests (https://github.com/mrtazz/restclient-cpp)

5.2 Certificates

5.2.1 Generate selfsigned certificate

The server uses TLS 1.3 to communicate with the client. This means that the server needs a valid certificate to be able to communicate with the client.

In the `certs` directory, there is a `generate.sh` script that will generate a self-signed certificate for the server. This script will generate a certificate for the hostname `localhost`. This means that the client will need to connect to the server using this hostname.

The script will generate a `cert.pem` and a `cert.key` file in the `certs/out` folder. The server will use these files to communicate with the client.

5.2.2 Using an existing certificate

If you already have a certificate for the server, you can simply replace the `cert.pem` and `cert.key` files in the `certs/out` folder with your own files. If you want to change the path of the certificate files, you can change the docker compose file.

5.2.3 Trusting the certificate

The client is configured to only accept certificates that are either signed by a given CA or that are given. The client will search for the certificate file next to the executable with the name `cert.pem`.

If the file is a CA, the client will use it to verify the server's certificate. If the file is not a CA, the client will use it as the server's certificate.

5.3 Run the server

5.3.1 Using docker

At the root of the project run the following command:

```
docker-compose up
```

This will automatically build the server and run it.

The server will then be accessible at `https://localhost:4242`.

5.3.2 Using python

5.3.2.1 Install dependencies

```
pip3 install -r requirements.txt
```

5.3.2.2 Run the server

```
python3 wsgi.py
```

5.4 Run the client

5.4.1 Build the project

```
mkdir build
cd build
cmake ..
make
```

5.4.2 Run the client

```
./eos
```

5.4.3 Get help

```
./eos --help
```

5.5 Tests

At the root of project is a `tests.sh` file. This script will run all commands of the project and will test the output of some of them. This script is used to test the project and to make sure that everything is working as expected.

The tests require docker to be installed. They also require the client to be built and the trusted certificate to be in place.

The script will start a new server and will run all commands against it. At the end, it will stop the server. **Beware that the script will empty the data directory of the server.**

It can be run using the following command:

```
./tests.sh
```