UNIVERSITÀ DI PISA

Computer Engineering

Foundations of Cybersecurity

Secure Cloud Storage

Team members:

Francesco Martoccia

Salvatore Lombardi

Luca Tartaglia

ACADEMIC YEAR 2021/2022

Contents

1 - Introduction and Design Choices

1.1 Introduction

The project consists in implementing a Client-Server application that resembles a **Cloud Storage**. Each user has a "dedicated storage" on the server, and User A cannot access User B "dedicated storage".
The two parts have the following **pre-shared crypto material**:

Users:

- They have already the **CA certificate**.
- They have each a **long-term RSA key-pair**.
- The **long-term private key** is password-protected.

Server:

- It has its own **certificate** signed by the **CA**.
- It knows the **username** of every registered user.
- It knows the **RSA public key** of every user.

Users are pre-registered on the server and the "dedicated storage" is already allocated. When the client application starts, Server and Client must **authenticate**. The Server must authenticate with the **public key** certified by the **certification authority**, while the Client must authenticate with the **public key pre-shared** with the server. During authentication a symmetric session key must be negotiated.

Requirements:

- The negotiation must provide **Perfect Forward Secrecy**.
- The entire **session** must be **encrypted** and **authenticated**.
- The entire **session** must be **protected** against **replay attacks**.

Once connected to the service, the client can perform the following operations:

- **Upload**: Specifies a filename on the client machine and sends it to the server. The server saves the uploaded file with the filename specified by the user. If this is not possible, the file is not uploaded. The uploaded file size can be up to 4 GB.

- **Download**: Specifies a file on the server machine. The server sends the requested file to the user. The filename of any downloaded file must be the filename used to store the file on the server. If this is not possible, the file is not downloaded.
- **Delete**: Specifies a file on the server machine. The server asks the user for confirmation. If the user confirms, the file is deleted from the server.
- **List**: The client asks to the server the list of the filenames of the available files in his dedicated storage. The client prints to screen the list.
- **Rename**: Specifies a file on the server machine. Within the request, the clients sends the new filename. If the renaming operation is not possible, the filename is not changed.
- **LogOut**: The client gracefully closes the connection with the server.

1.2 Design Choices

The Design choices to guarantee security requirements are as follows.
To ensure **perfect forward secrecy**, the authentication phase employs the Station-to-Station protocol, leveraging the ephemeral Diffie-Hellman Key Exchange. This cryptographic exchange establishes a secure communication channel between the Client and Server. Upon successful negotiation, both entities possess a shared 128-bit session key, meticulously derived using the SHA256 algorithm. This session key serves as the foundation for subsequent encrypted message exchanges, ensuring the confidentiality and integrity of the communication between the parties involved.

In addressing the requirements for **message encryption and authentication**, the chosen approach involves employing the AES algorithm with a 128-bit key and GCM (Galois Counter Mode) encryption method. This strategic selection ensures the simultaneous provision of robust block encryption and message authentication.
The use of GCM contributes to the protocol's resilience against **replay attacks**. This is achieved through the utilization of a counter within the Additional Authenticated Data (AAD) field, specifically designed to keep track of the messages sent. By implementing such measures, the protocol enhances security by preventing unauthorized replay of previously transmitted messages, thereby reinforcing the integrity of the communication process.

The project was developed by using C++17, and the implementation of cryptographic protocols was realized through the OpenSSL Version 1.1.1 library. Communication aspects were established using POSIX sockets.

To facilitate concurrent connections from multiple users, a threading model was adopted. Specifically, a dedicated thread associated with the server is spawned for each user process initiated. This threading approach enhances system efficiency, responsiveness, and scalability, allowing the project to adeptly handle simultaneous interactions from diverse users in a secure and resilient manner.

1.3 Registered Users

In order to execute and test the application, 3 users were created and pre-registered. The CA Certificate, the long-term private key, and the RSA long-term key pair were generated using SimpleAuthority software.

The following table outlines the usernames and associated passwords required for accessing the dedicated storage of each user (username and password are case-sensitive).

Registered Users Credentials	
Username	**Password**
Luca	*lucapassword*
Francesco	*francescopassword*
Salvatore	*salvatorepassword*

2 - Exchanged Messages

This section will illustrate the structure of the main messages exchanged during the communication between the client and server.

2.1 Generic

All messages exchanged during the session between the client and server are encapsulated within a ***Generic*** message. This message contains fields of fixed size (**IV**, **AAD**, and **TAG**) transmitted in plain text, along with a variable-length **ciphertext**.
The Initialization Vector (**IV**) is used by the **AES GCM** algorithm to initialize the operation, and a different **IV** is used for each encryption. For **AES GCM**, its default size, which is the one used, is 96 bits (12 bytes).
The **AAD** (Additional Authenticated Data) field contains data that is authenticated but not encrypted. In this case, it contains the 32-bit (4 bytes) counter that keeps track of the current message number (incremented with each send or receive).
The authentication **TAG** serves as a cryptographic checksum. It is generated during the encryption process and is used for data **integrity** verification during decryption. The **TAG** is computed based on the **ciphertext**, the **AAD**, and the initialization vector (**IV**). In this case its size is 128 bits (16 bytes). During decryption, the recipient recalculates the **TAG** using the received **ciphertext**, the **AAD**, and **IV** and compares it with the received one to ensure the integrity of the encrypted data.
The **ciphertext**, instead, is obtained from the encryption of a serialized message, and its lenght depends on the size of the plaintext.

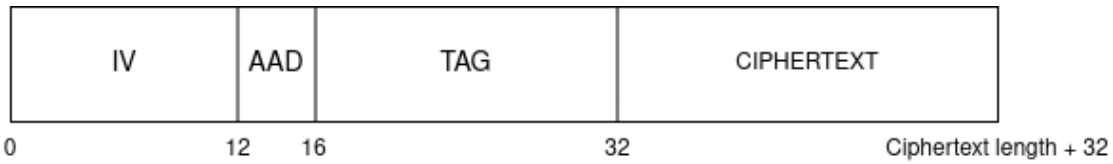

Figure 2.1: Generic message structure

2.2 SimpleMessage

The ciphertext inside each ***Generic*** message is obtained by encrypting a serialized message. The structure of each message varies depending on the operation to be performed and the data that needs to be exchanged between the client and server. In many cases, however, it is necessary to send only a message code to specify the operation to be executed, for example,

or to send an ACK/NACK. For this reason, a standard message called ***SimpleMessage*** has been created.
This message consists of a **message code** (1 byte) indicating the type or outcome of an operation and 70 other **random bytes**. This has been done to establish a standardized size for the initial messages of various operations, allowing the server to handle them in the same way.

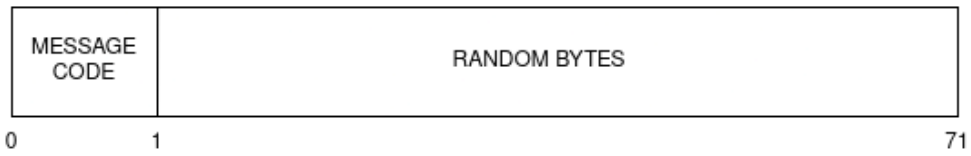

Figure 2.2: SimpleMessage structure

3 - Communication Protocols

3.1 Authentication Operation

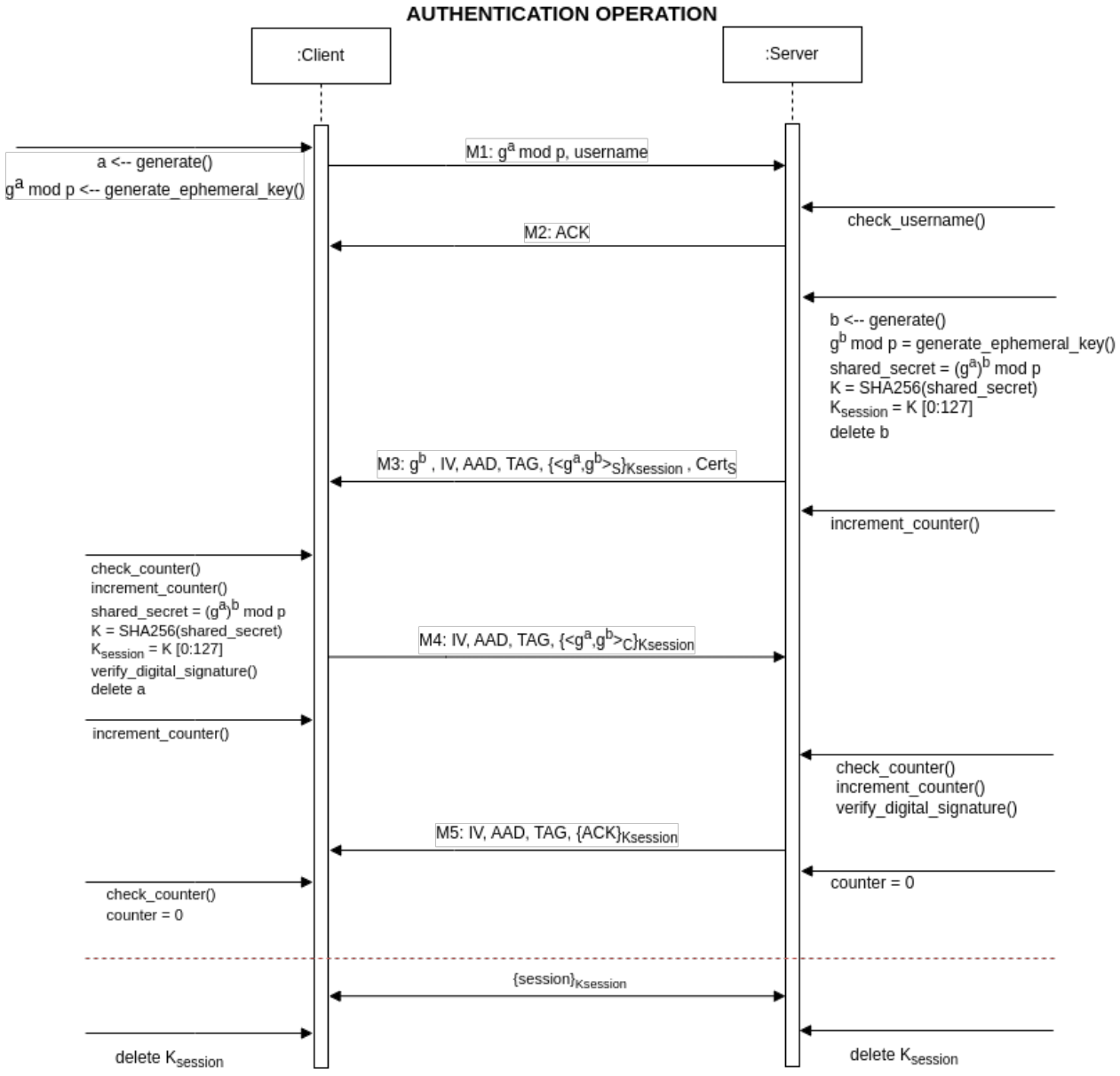

Figure 3.1: Authentication Operation Sequence Diagram

LEGEND

- a,b : private keys (ephemeral)
- g^a mod p / g^b mod p: ephemeral key generated from a/b
- K : Complete session key
- Ksession : First session key's 128 bits
- $Cert_S$: Server's Certificate
- $<>_{S/C}$: Digital signature with Server/Client private key
- $\{\}_{Ksession}$: Encryption with session key
- IV: initialization vector
- AAD: Additional Authenticated Data, contains the counter
- TAG : Authentication Tag
- ACK: ID of the ack response message (this field can take also NACK value)

The protocol used to allow a user to be authenticated is the Station-to-Station, which fulfills direct authentication while guaranteeing Perfect Forward Secrecy.
In the first message (Message M1), the client sends to the server its ephemeral key, generated via the Diffie-Hellman protocol, and the username of the user (sanitized) it intends to authenticate.

Upon receiving the client's request, the server checks if the username is present and, therefore, if the user is already registered.
In case the user name is not found, a message (Message M2 (NACK)) is sent to the client, and authentication is stopped. At this point, the client again asks the user for a username and password.
If the username, instead, is found, the server sends a message (Message M2 (ACK)) to the client, indicating that the process can continue.

In order to create a shared secret, the server generates its own ephemeral key via Diffie-Hellman and combines it with that of the client. From this, via the SHA256 algorithm, the K key is generated. The session key is generated by extracting the first 128 bits of K. Having done this, the server sends a message (Message M3) to the client containing its unencrypted ephemeral key, the encrypted combination of the ephemeral keys signed with its long-term private key, and its unencrypted certificate.

As soon as the message (Message M3) is received, the client can also generate the shared secret by combining its own ephemeral key with that of the server. Having done this, thanks to the SHA256 algorithm the K key is produced. The first 128 bits of K represent the session key. Having done this, the client decrypts through the latter the message signed with the server's long-term private key and verifies its signature. If the verification is unsuccessful, the process is stopped and it is necessary to start it over again. If the verification is successful, the client sends an encrypted message (Message M4) to the server containing the combination of the two ephemeral keys signed with its own long-term private key.

Upon receiving the message (Message M4) from the client, the server can decrypt it and verify the signature. If the verification is unsuccessful, the process must be re-initiated. Otherwise, the session key generation was successful and the client and server can exchange messages using the session key produced. As a final step, the server sends an encrypted message (Message M5 (ACK)), which notifies the client that the authentication process was successful.

To avoid an overflow error, should the counter reach the maximum allowed value, the entire

authentication procedure will be rerun, and, therefore, a new session key will be generated.

3.2 Upload Operation

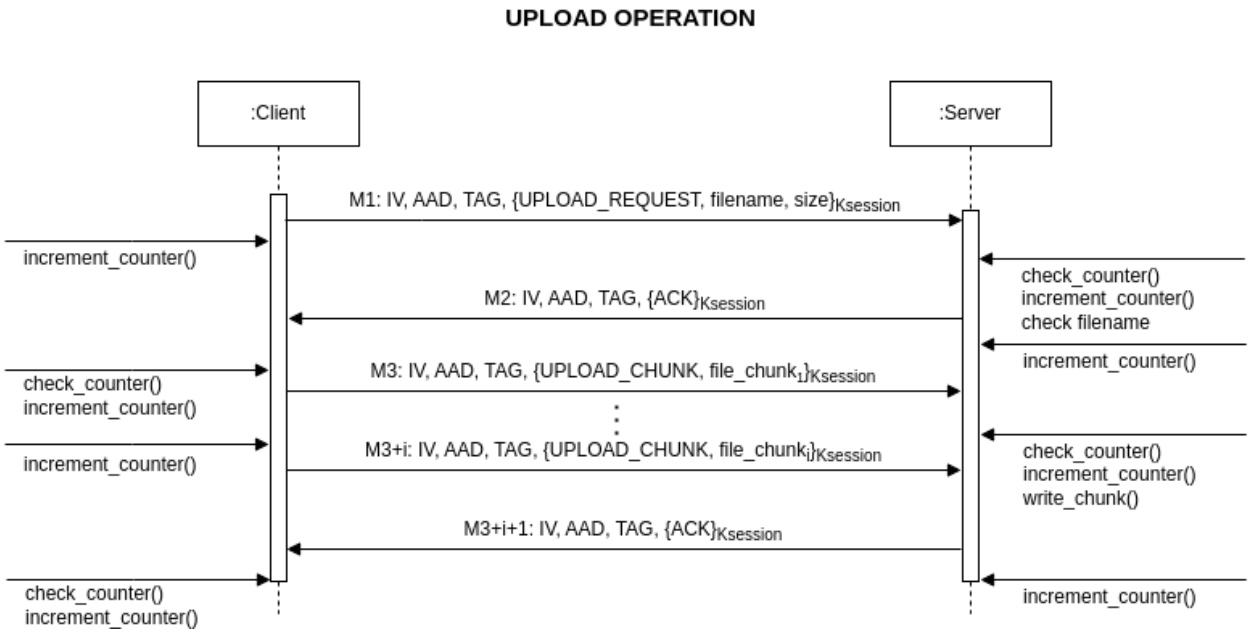

Figure 3.2: Upload Operation Sequence Diagram

LEGEND

- IV: initialization vector
- AAD: Additional Authenticated Data, contains the counter
- TAG: Authentication Tags

- UPLOAD_REQUEST: ID of the request message to upload a specific file
- ACK: ID of the ack response message (this filed can take also NACK value)
- UPLOAD_CHUNK: ID of the message containing the i-th file chunk to upload
- filename: name of the file to upload
- size: size of the file to upload
- $file\_chunk_i$: payload containing the i-th file chunk (fixed size at 1Mb)

In the first message (Message M1) the user sends an upload request to the server specifying the filename (sanitized) and the file size (automatically computed) which cannot be greater than 4GB or equal to 0 (empty file).

Once the server receives the request from the client it checks if the file already exists, in which case it sends the client a failure message (NACK), otherwise a success message (ACK) (Message M2).

If the client receives a success message from the server, it divides the file to be uploaded into chunks (fixed size at 1 MB) and sends them in separate messages to the server (Messages M3+i).

Once the server receives all the chunks and writes them into the created file it sends a final message (Message M3+i+1), which indicates the success (ACK) of the entire operation. In caso of failure in loading, the server sends a failure message (NACK) to the client.

3.3 Download Operation

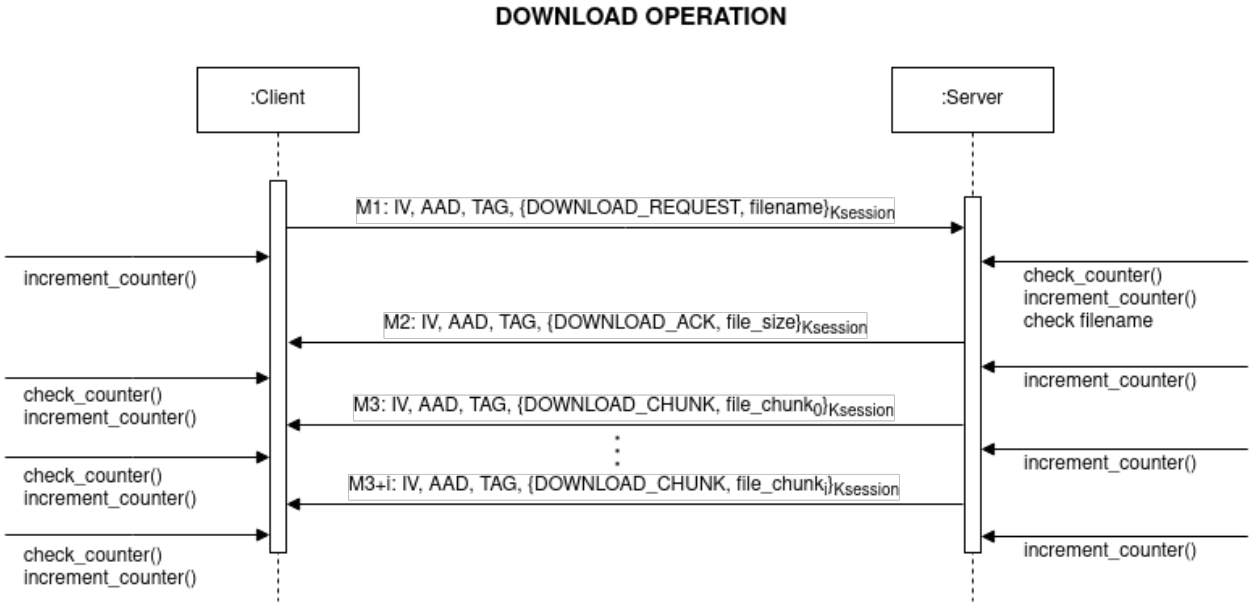

Figure 3.3: Download Operation Sequence Diagram

LEGEND

- IV: Initialization Vector
- AAD: Additional Authenticated Data, contains the counter
- TAG: Authentication Tag

- DOWNLOAD_REQUEST: ID of the request message to download a specific file
- DOWNLOAD_ACK: ID of the response message containing the file size, it is set to FILE_NOT_FOUND if there is no file in the user storage with the specified filename
- DOWNLOAD_RESPONSE: ID of the response message containing the i-th file chunk to download
- $file\_chunk_i$: payload containing the i-th file chunk (fixed size)

If the file name inserted by the user is valid and is not already present locally, a message M1 will be sent to the server containing the message code of the operation (DOWNLOAD_REQUEST) and the name of the file to be downloaded.

After receiving the request from the client, the server checks if the file is present within the user's storage. If the file is present, it sends a message M2 with the DOWNLOAD_ACK code and the file size in bytes. Otherwise, it sends a message with the FILE_NOT_FOUND code and a file_size of 0.

The client checks the received message: if the message code is FILE_NOT_FOUND, the operation terminates; otherwise, it prepares to receive a number of file chunks that depends on the file_size specified in the M2 message.

If the file was present, the server sends the file to the client in fixed-size chunks (1 MB, except for the last chunk, which is equal to the remaining number of bytes) through messages M3 and M3+i (if there are many chunks), with message code DOWNLOAD_CHUNK.

3.4 Delete Operation

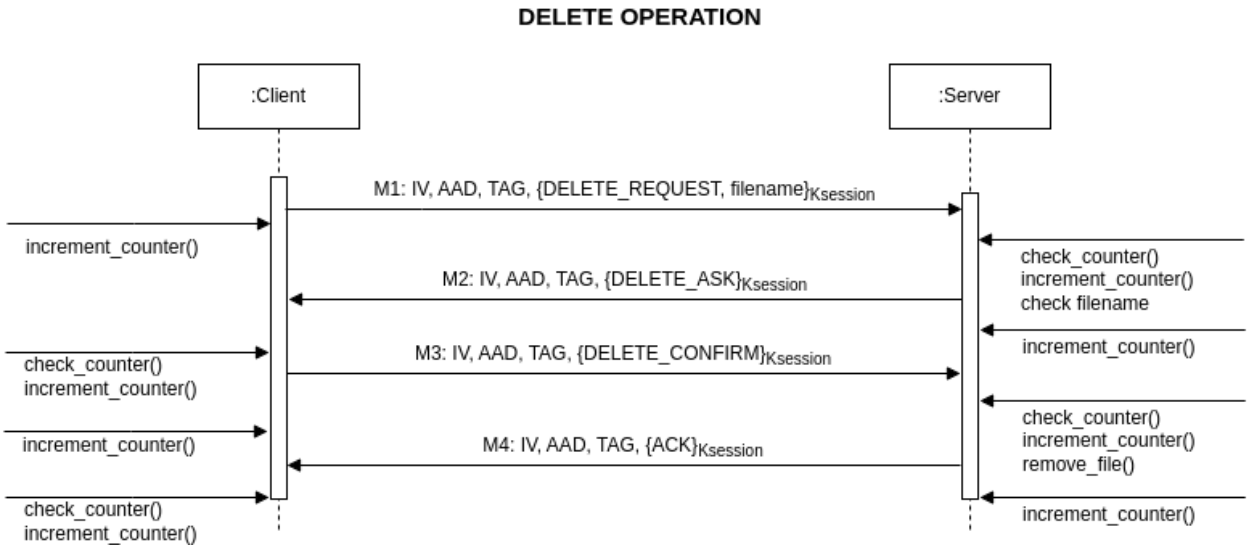

Figure 3.4: Delete Operation Sequence Diagram

LEGEND

- IV: initialization vector
- AAD: Additional Authenticated Data, contains the counter
- TAG / TAG': Authentication Tags

- DELETE_REQUEST: ID of the request message to delete a specific file
- DELETE_ASK: ID of the message to ask confirmation from user
- DELETE_CONFIRM: ID of the message to confirm the delete request (this field can take also NO_DELETE_CONFIRM value)
- ACK: ID of the ack response message (this field can take also NACK value)
- filename : name of the file to delete

In the first message (Message M1), the user sends a delete request to the server specifying the filename (sanitized) of the file to be deleted.

Upon receiving the client's request, the server verifies the existence of the specified file. If the file is found, the server sends a confirmation request (Message M2) to the client, seeking further acknowledgment for the deletion process (DELETE ASK).

Upon receiving the confirmation request, the client holds the authority to decide whether to proceed with confirming the file deletion by sending a confirmation message (DELETE CONFIRM) or to terminate the operation by responding with a denial (NO DELETE CONFIRM).

Responding to the client's command, the server takes appropriate actions. Upon receiving a confirmation, it promptly removes the file from storage and communicates a message to the client, signaling the successful conclusion of the entire operation (ACK).

Conversely, if the client's response indicates an operation abort, the server halts from file deletion and communicates an appropriate message to the client (NACK).

3.5 List Operation

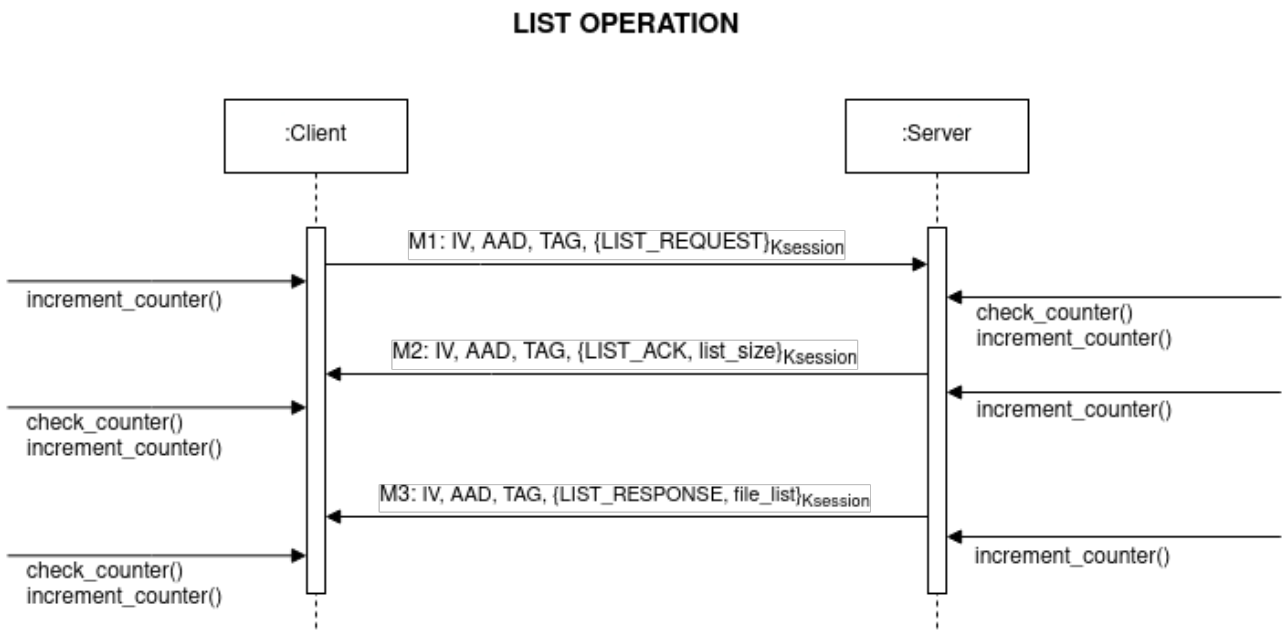

Figure 3.5: List Operation Sequence Diagram

LEGEND

- IV: Initialization Vector
- AAD: Additional Authenticated Data, contains the counter
- TAG: Authentication Tag

- LIST_REQUEST: ID of the request message to obtain the list of files
- LIST_ACK: ID of the response message containing the list size
- LIST_RESPONSE: ID of the response message containing the list of files
- file_list: payload containing the user's file list

The client sends a request to the server with message code LIST_REQUEST (Message M1).

The server generates the list of files for the user and sends a message M2 containing the LIST_ACK code and the size of the list.

The client checks the list's length; if it is 0, the function terminates. Otherwise, the client prepares to receive the list of files.

If the list_size was not 0, the server sends a message M3 containing the LIST_RESPONSE code and the list of files in the user's storage.

3.6 Rename Operation

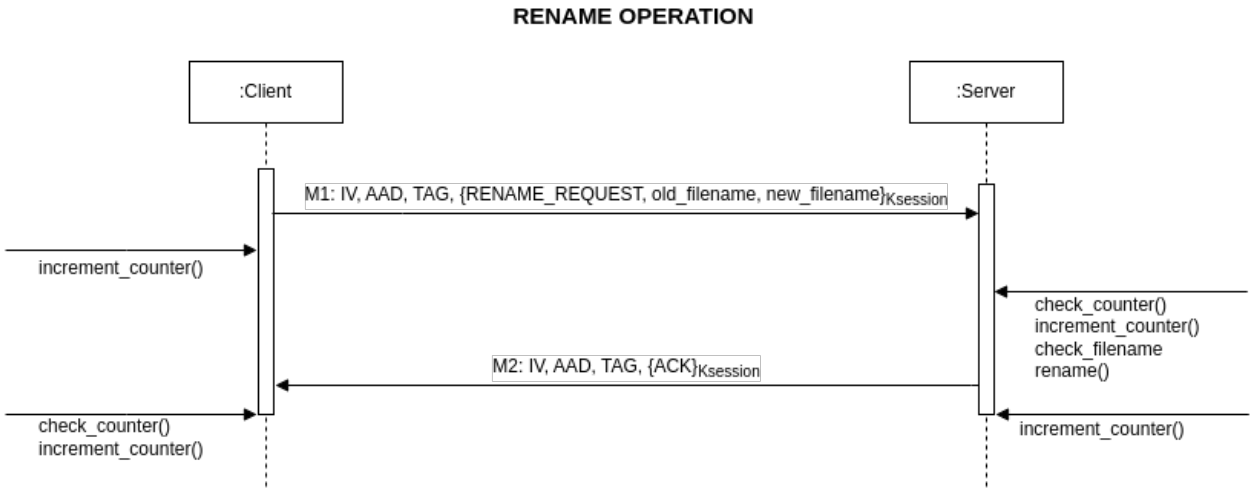

Figure 3.6: Rename Operation Sequence Diagram

LEGEND

- IV: initialization vector
- AAD: Additional Authenticated Data, contains the counter
- TAG : Authentication Tag

- RENAME_REQUEST: ID of the request message to rename a specific file
- ACK: ID of the ack response message (this field can take also NACK, FILE_NOT_FOUND and FILE_ALREADY_EXISTS values)

In the first message (Message M1), the user sends a rename request to the server specifying the "old" filename (sanitized) of the file that he wants to rename and the new file name (sanitized).

Upon receiving the message (Message M1), the server checks if the file is present in the storage and, if it is not, forwards an encrypted message (Message M2) to the client with ID: FILE NOT FOUND as a response.
If the file is present, but the new file name is already assigned to another file, the server sends an encrypted message (Message M2) to the client with ID: FILE ALREADY EXISTS as a response.
Finally, if the rename operation is possible, the server performs it and then sends a message (Message M2) to the client with an ACK that confirms that everything went well.

3.7 Logout Operation

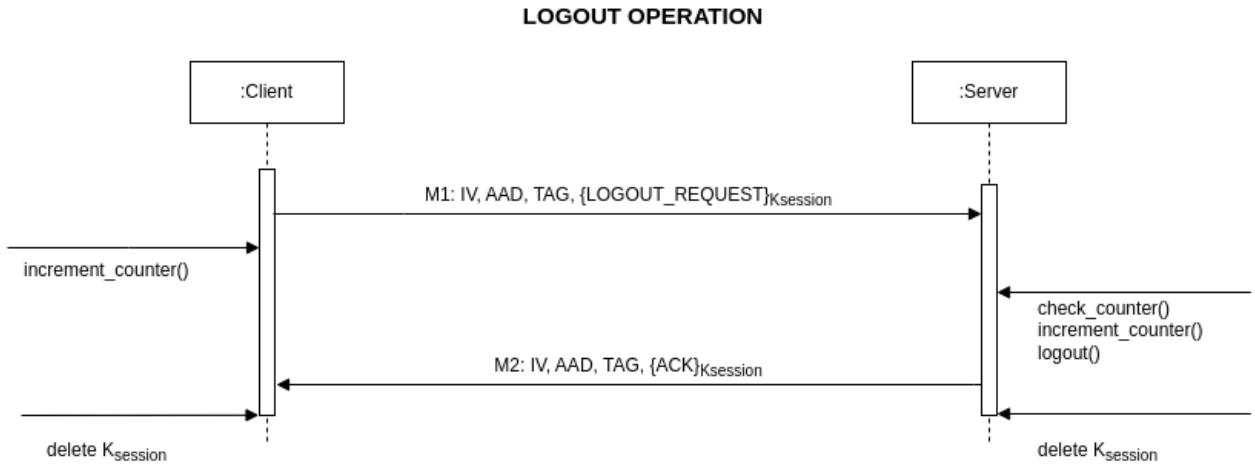

Figure 3.7: Logout Operation Sequence Diagram

LEGEND

- IV: initialization vector
- AAD: Additional Authenticated Data, contains the counter
- TAG: Authentication Tag

- LOGOUT_REQUEST: ID of the request message to execute the logout
- ACK: ID of the ack response message (this field can take also NACK value)

In the first message (Message M1), the user issues a logout request to the server.

Upon receiving the logout request from the client, the server proceeds to execute the requested operation. Subsequently, the server sends a message (Message M2) indicating either the success of the logout operation (ACK) or, in the event of an operational failure, a corresponding failure message (NACK).