UNIVERSITÀ DI PISA

Computer Engineering

Foundation Of Cybersecurity

Cloud Storage

Team Members:
Arsalen BIDANI
Bashar HAYANI

Academic year: 2023/2024

Contents

1. Introduction

This project involves creating a Client-Server application resembling Cloud Storage. Each user is allocated dedicated storage on the server and can only access their own storage. Following authentication, users can securely upload, download, rename, or delete files within their storage.

Three users are pre-registered on the server namely user1,user2, and user3. Each user possesses a long-term RSA keypair, the RSA private key for each user is password-protected. The server maintains records of each registered user's username and their RSA public key, and has allocated dedicated storage for each user.

2. Implementation and protocols choices

1. Security Aspect

- **Perfect Forward Secrecy (PFS)** was achieved by using Elliptic-curve Diffie–Hellman (ECDH) , **ANSI X9.62 Prime 256v1** curve is used, by generating temporary session keys for each communication session, ensuring that compromising one session's key does not compromise the security of past or future sessions.

- We have used **AES-128-CCM** (Advanced Encryption Standard-Counter with Cipher Block Chaining Message Authentication Code). It combines symmetric key encryption with authentication, making it suitable for securing data in various network protocols and applications where both **confidentiality** and **integrity** are crucial.

- **Counters** were implemented by server and client side to mitigate **replay attacks** by incorporating unique values for each encryption operation, this prevents adversaries from reusing intercepted ciphertexts to impersonate legitimate messages.

- **SHA-256** was used to obtain the key session by hashing the derived shared secret as enhances security by reducing the risk of certain cryptographic attacks and ensures that the resulting key material has a fixed length.

2. Application Aspect

- Our project is developed in **C++17** language on **Ubuntu 18.04** and consists of two executables: **client** and **server**.

- We have used **CMake** to wrap and build the applications which is a cross-platform build system generator.

- We have chosen to use **TCP** over UDP as a protocol of communication over sockets as it ensure **reliable**, ordered, and **error-checked** delivery of data as necessary.

- We have build the application in a **multi-threaded** manner that allows to have multiple users connected to the server at the same time.

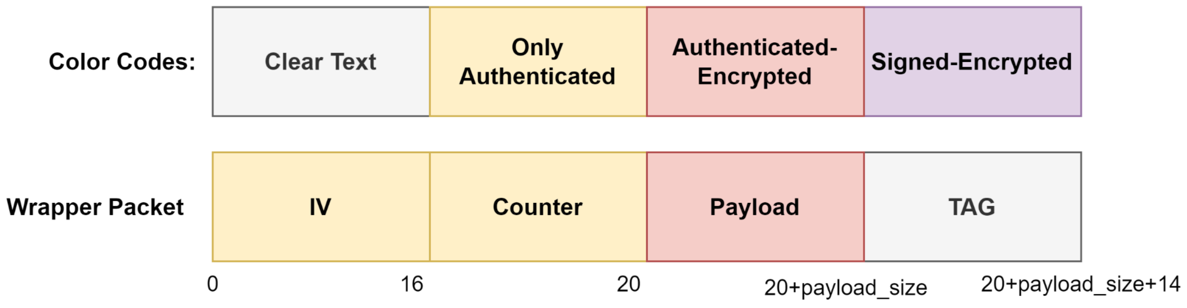
3. Wrapper Packet

After the successful login to the application, all of the communications requests will involve an encrypted-authenticated parts which includes a payload to deliver.

We have opted to developer a class that incorporates a **Wrapper packet** that acts as a universal packet utilized in all communication between the client and server post-login phase.

From here after on the communication packets displayed will be the payload that is inserted in the wrapper packet.

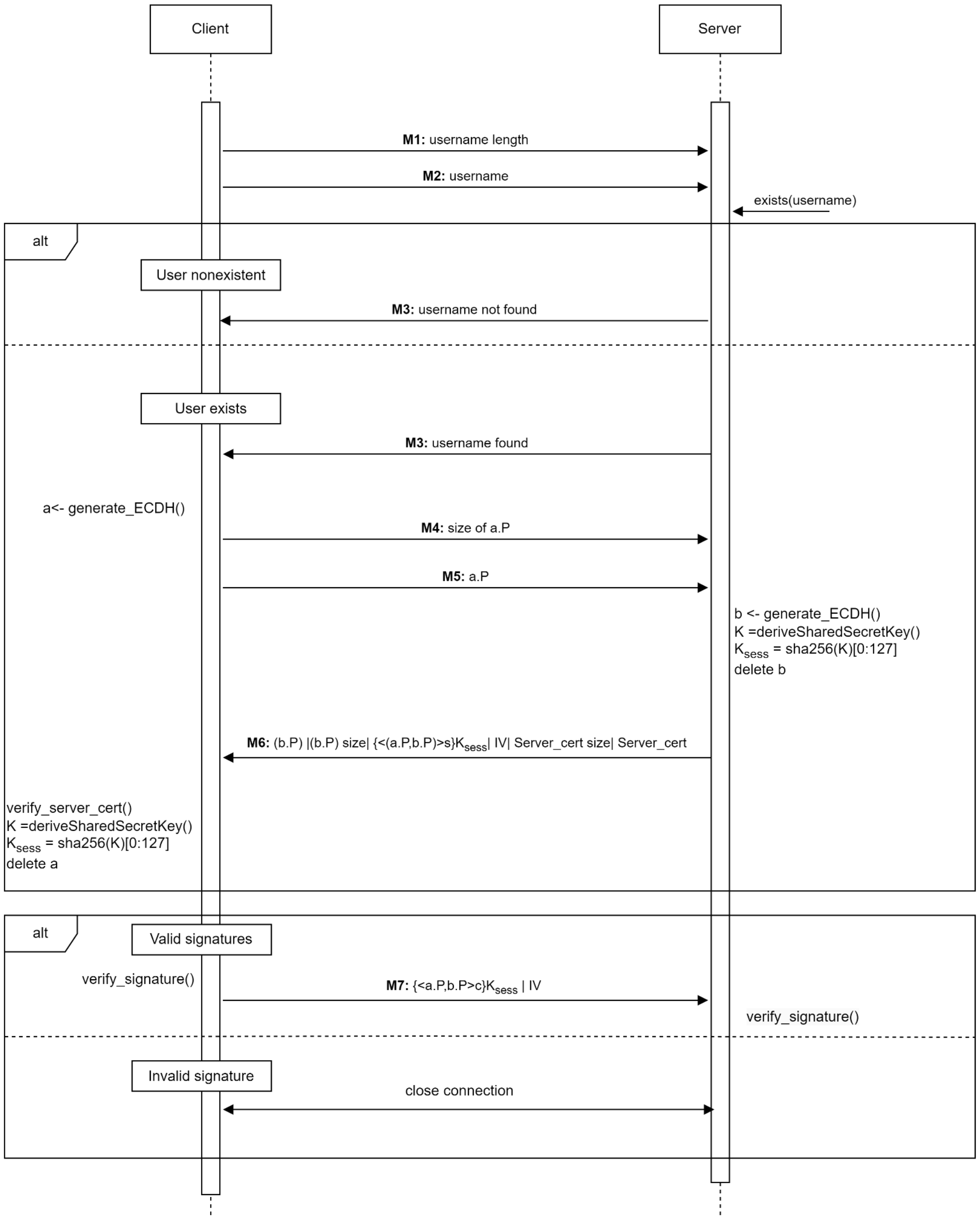
It consists of four fields: IV, counter, payload, and TAG.



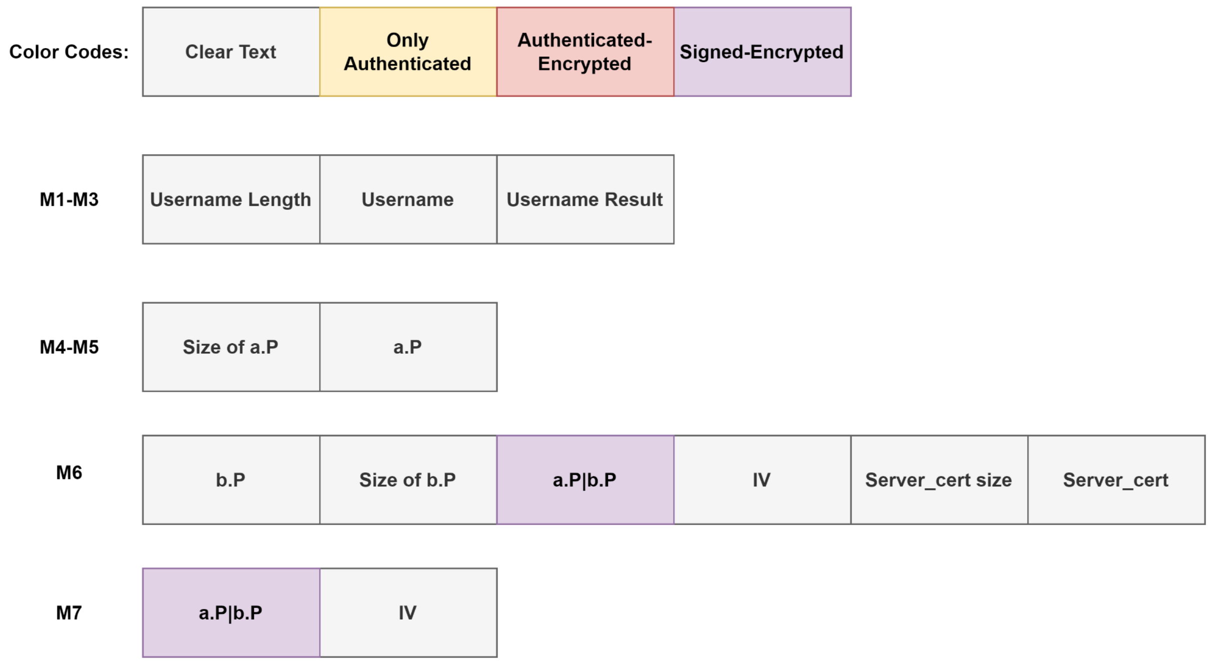
Wrapper packet

4. Login phase

In login phase the **Station-to-Station (STS)** protocol was implemented which is a cryptographic key agreement protocol that enables mutual authentication and secure key exchange between two parties.



Login phase sequence diagram



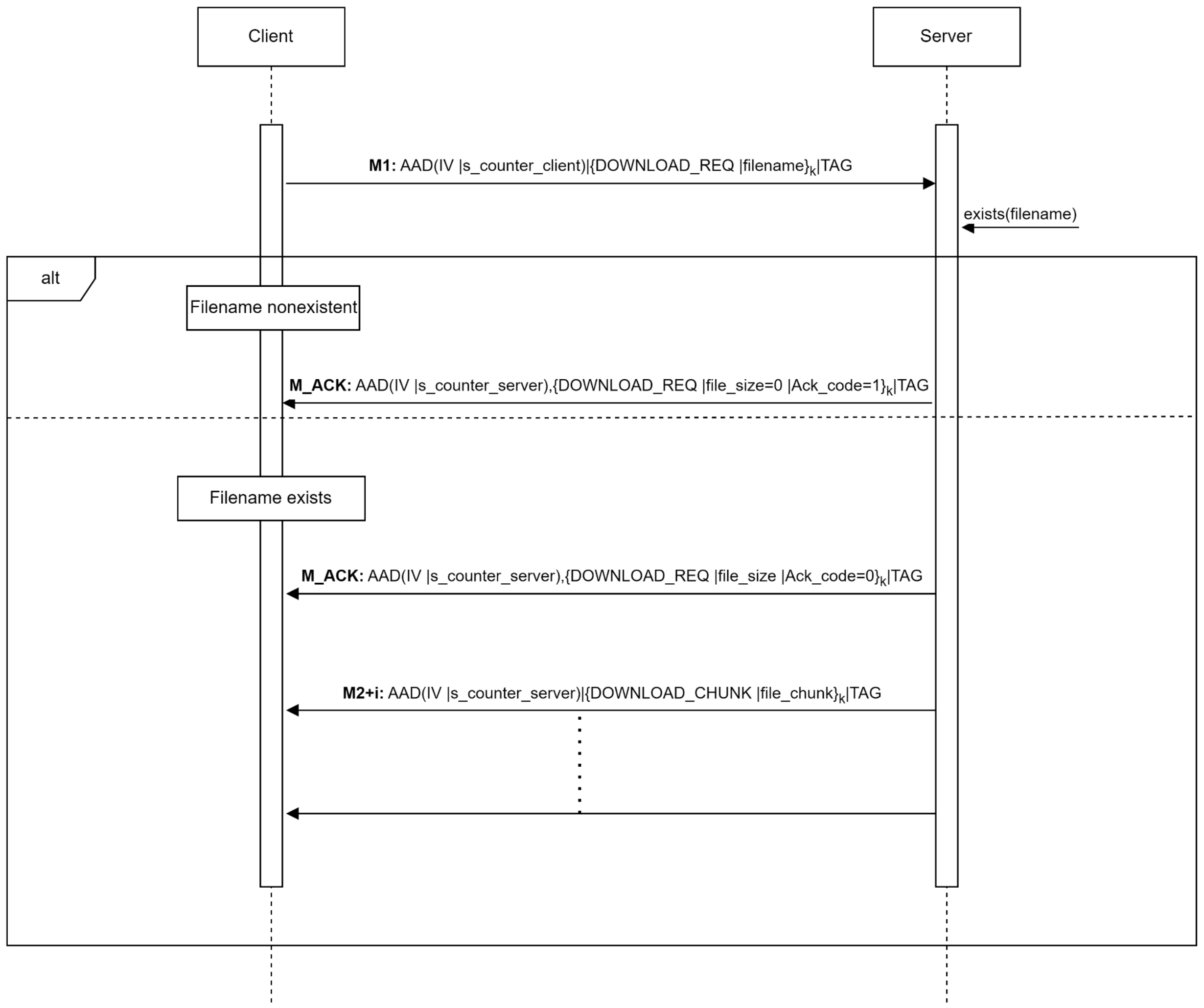
Login packets

5. Download

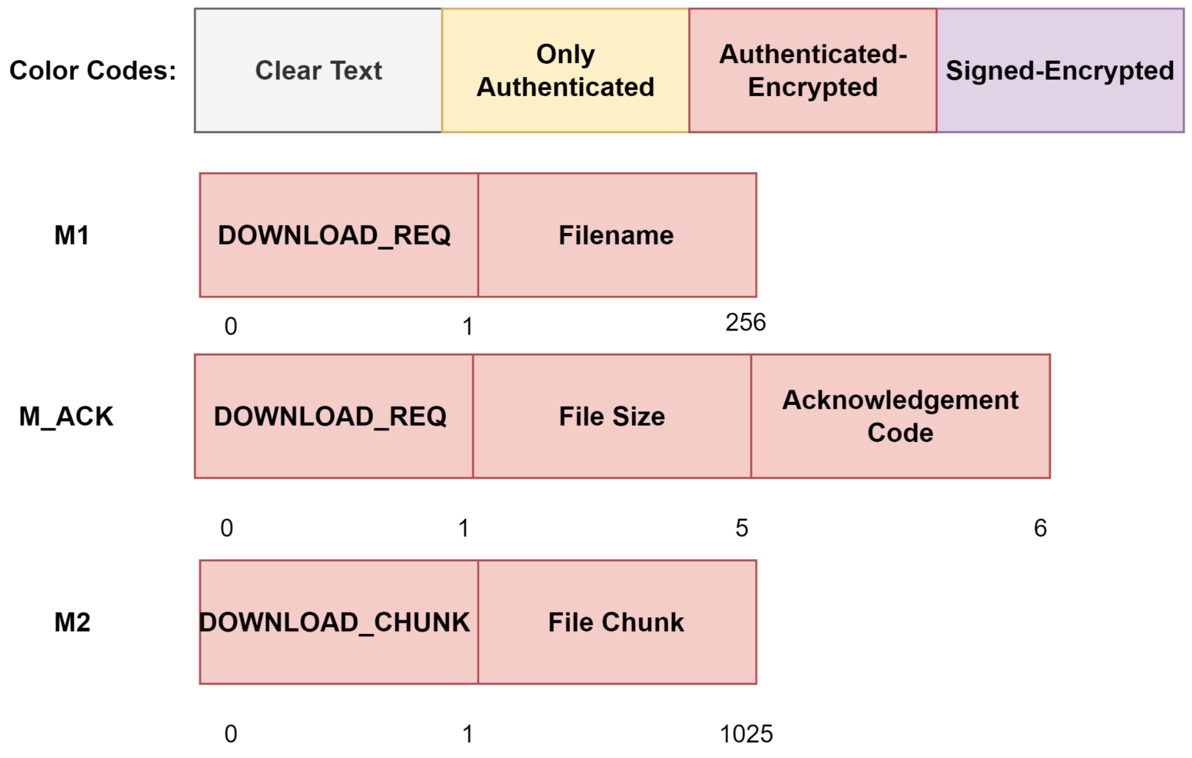
The client initiates the download protocol by sending the name of the file to be downloaded to the server.

If the file exists in the dedicated user's storage, the server responds by sending the file size to the client.

Subsequently, the server begins transmitting the file in **fixed-size** chunks, with each chunk set at **1KB**. This chunk size is predetermined and known by both the client and the server beforehand.


Client
Server
M1: AAD(IV |s_counter_client)|{DOWNLOAD_REQ |filename}k|TAG
exists(filename)
alt
Filename nonexistent
M_ACK: AAD(IV |s_counter_server),{DOWNLOAD_REQ |file_size=0 |Ack_code=1}k|TAG
Filename exists
M_ACK: AAD(IV |s_counter_server),{DOWNLOAD_REQ |file_size |Ack_code=0}k|TAG
M2+i: AAD(IV |s_counter_server)|{DOWNLOAD_CHUNK |file_chunk}k|TAG

Download sequence diagram



Download packets

6. Upload

The client initiates the upload protocol by sending the name and the size of the file to be uploaded to the server.

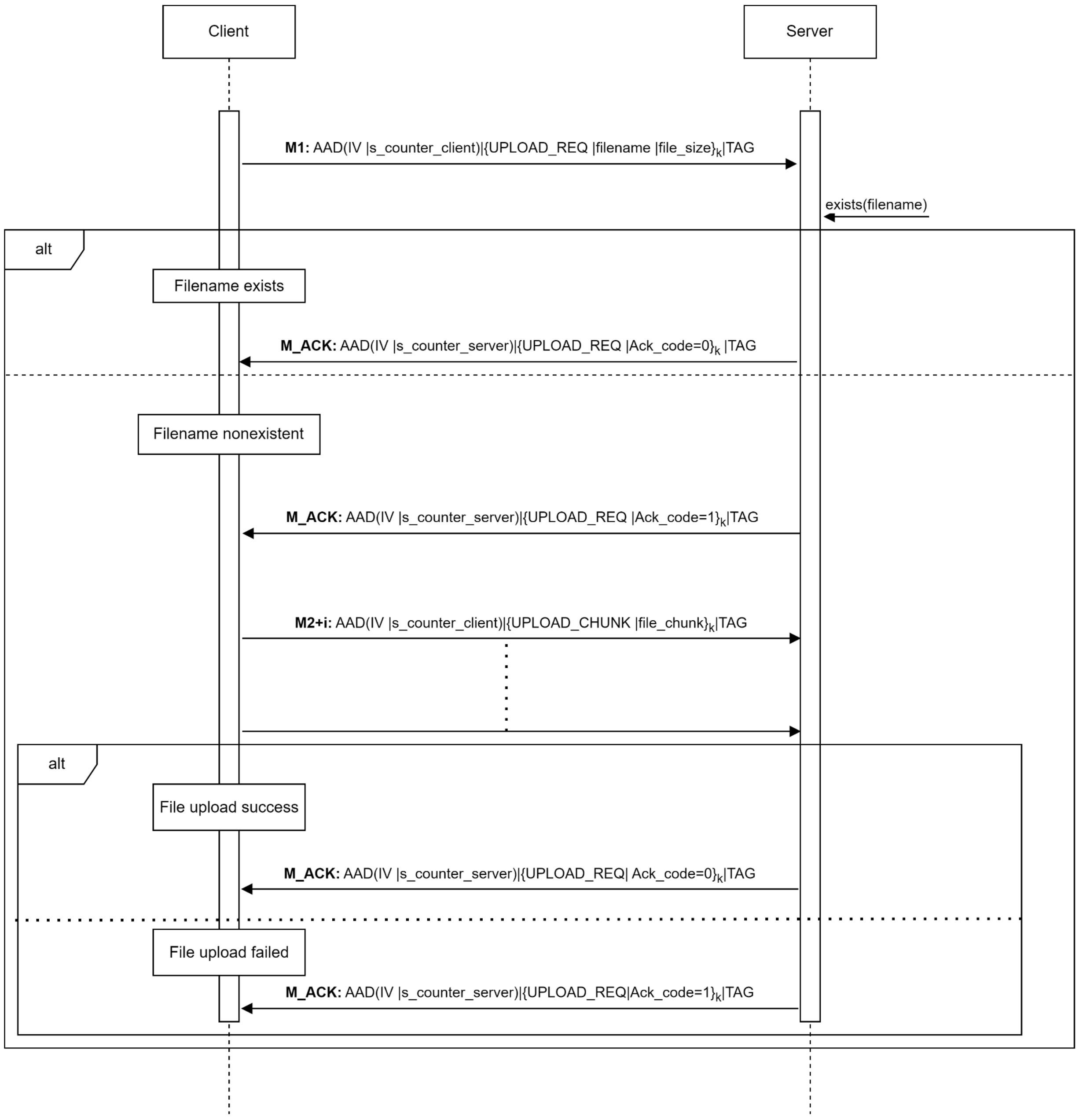
Before starting the protocol, the client verifies the file size on its end to ensure it does not exceed the **maximum allowed size of 4 GB**.

If the file does not exist in the dedicated user's storage, the server responds by sending **positive acknowledgement** code the client.

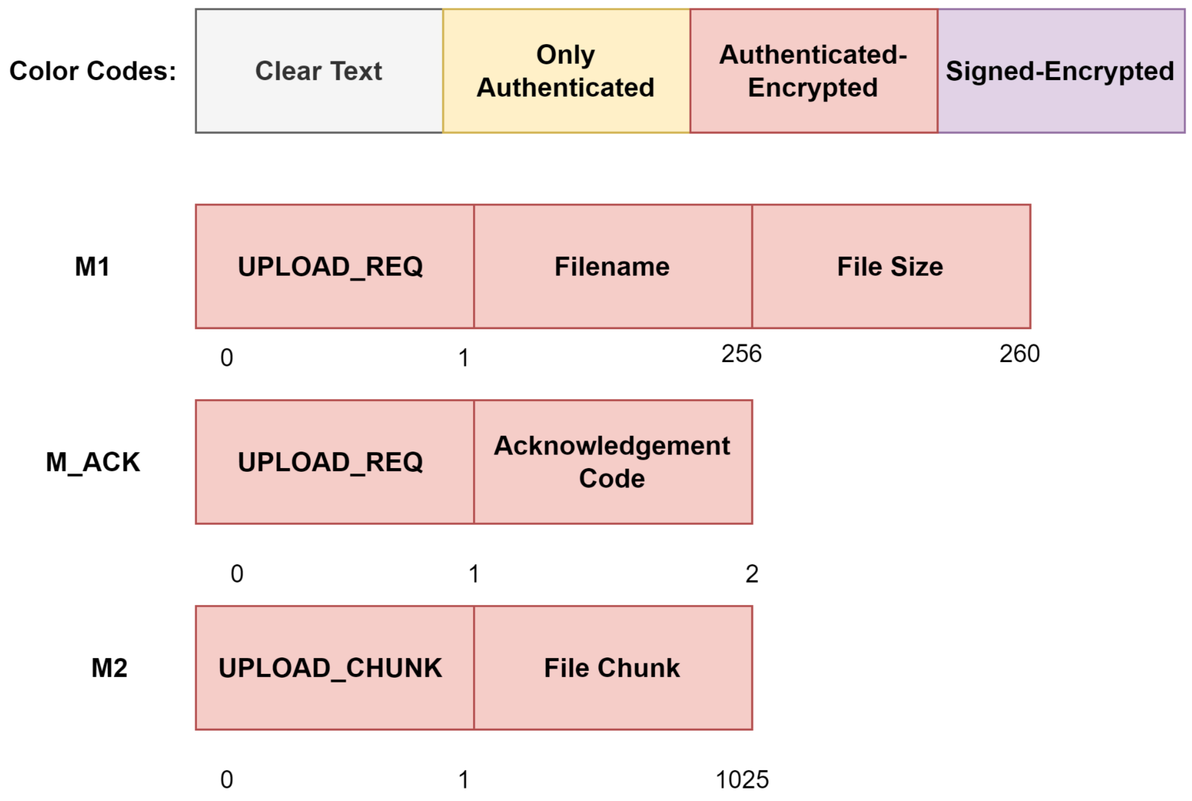
Subsequently, the client begins transmitting the file in fixed-size chunks, with each chunk set at 1KB.

This chunk size is predetermined and known by both the client and the server beforehand.

At the end of uploading, the server sends an **acknowledgement code** to the client determining whether the upload was successful or not.


Client
Server
M1: AAD(IV |s_counter_client)|{UPLOAD_REQ |filename |file_size}k|TAG
exists(filename)
alt
Filename exists
M_ACK: AAD(IV |s_counter_server)|{UPLOAD_REQ |Ack_code=0}k |TAG
Filename nonexistent
M_ACK: AAD(IV |s_counter_server)|{UPLOAD_REQ |Ack_code=1}k|TAG
M2+i: AAD(IV |s_counter_client)|{UPLOAD_CHUNK |file_chunk}k|TAG
alt
File upload success
M_ACK: AAD(IV |s_counter_server)|{UPLOAD_REQ| Ack_code=0}k|TAG
File upload failed
M_ACK: AAD(IV |s_counter_server)|{UPLOAD_REQ|Ack_code=1}k|TAG

Upload sequence diagram

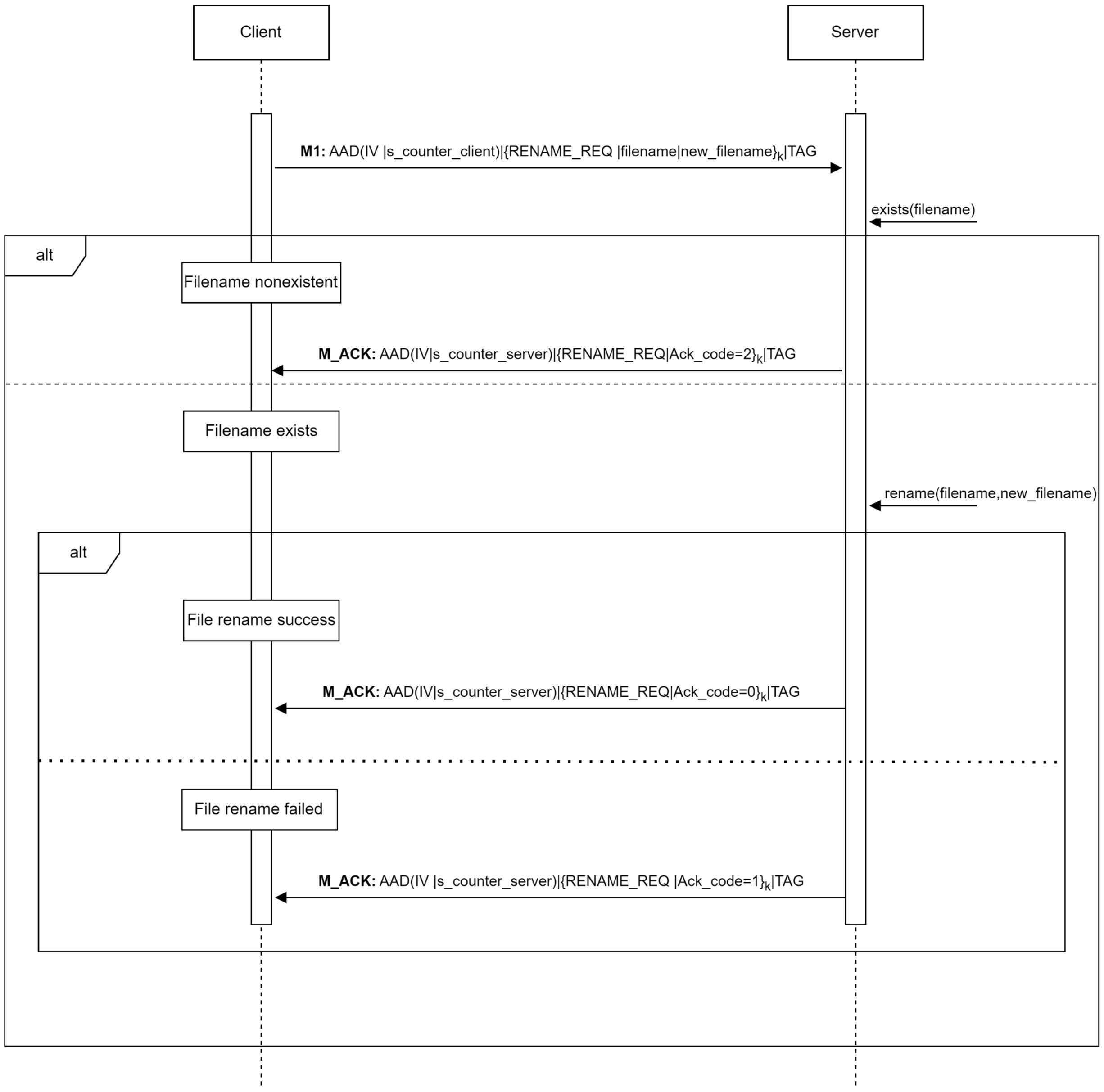


Upload packets

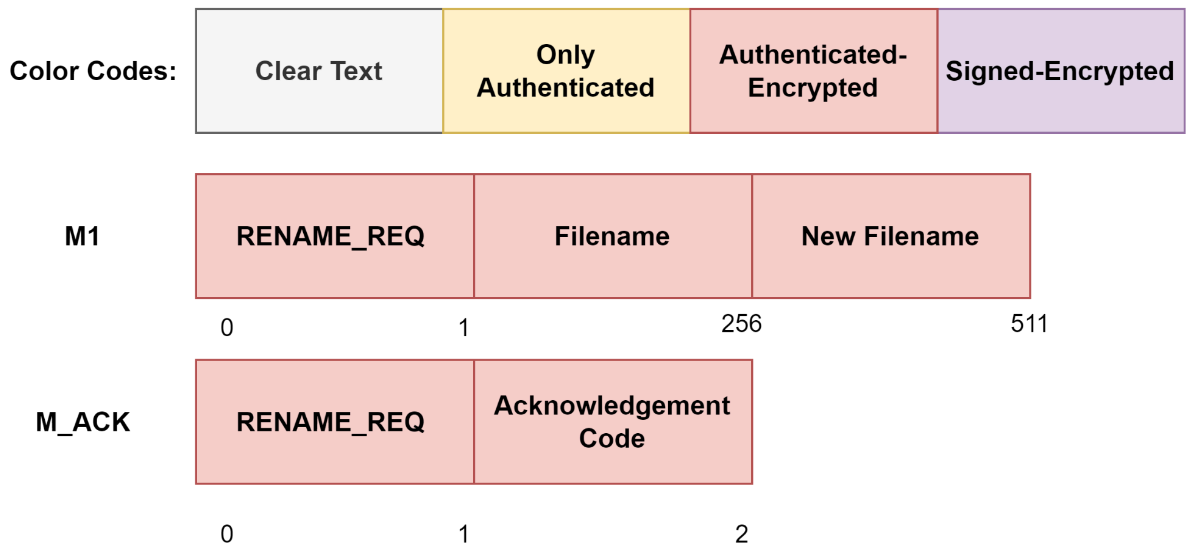
7. Rename

The client initiates the rename protocol by sending the **filename** and the **new filename** of to be renamed to the server.

As a result, the server sends an **acknowledgement code** to the client determining whether the rename was successful or not.


Client
Server
M1: AAD(IV |s_counter_client)|{RENAME_REQ |filename|new_filename}k|TAG
exists(filename)
alt
Filename nonexistent
M_ACK: AAD(IV|s_counter_server)|{RENAME_REQ|Ack_code=2}k|TAG
Filename exists
rename(filename,new_filename)
alt
File rename success
M_ACK: AAD(IV|s_counter_server)|{RENAME_REQ|Ack_code=0}k|TAG
File rename failed
M_ACK: AAD(IV |s_counter_server)|{RENAME_REQ |Ack_code=1}k|TAG

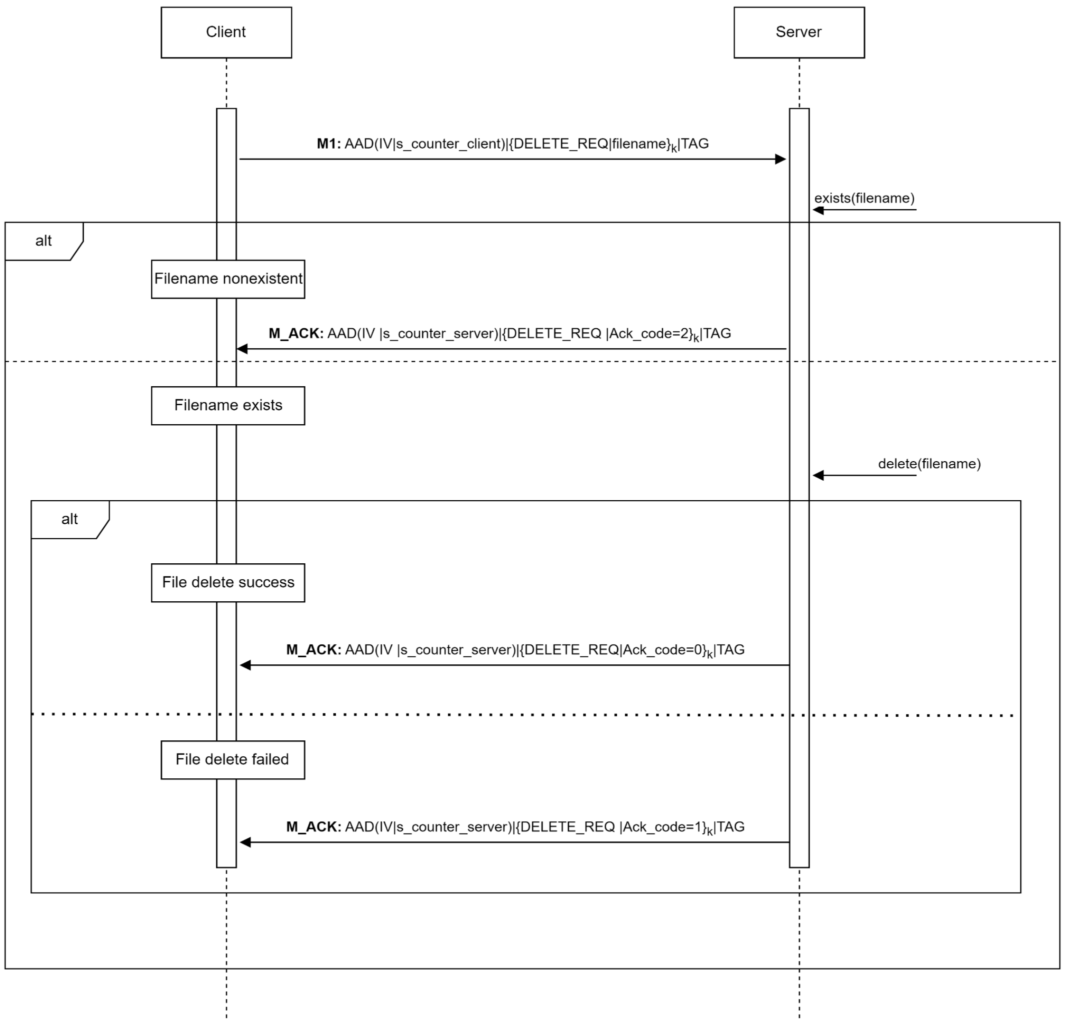
Rename sequence diagram



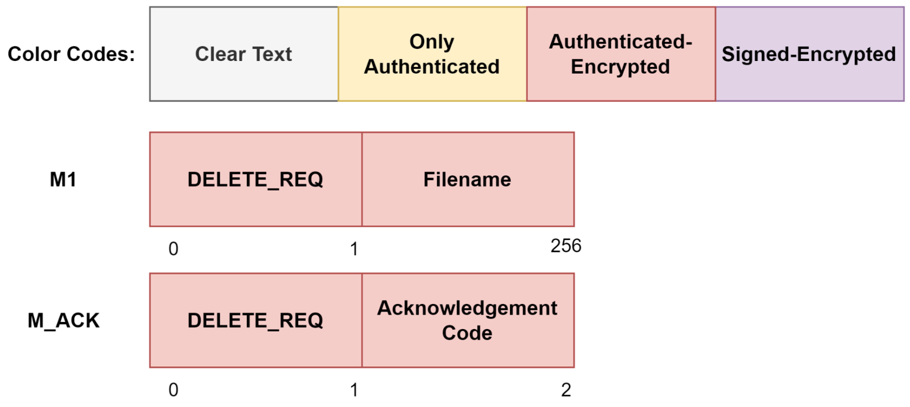
Rename packets

8. Delete

The client initiates the delete protocol by sending the **filename** to be deleted to the server.
As a result, the server sends an **acknowledgement code** to the client determining whether the delete was successful or not.


Client
Server
M1: AAD(IV|s_counter_client)|{DELETE_REQ|filename}$_k$|TAG
exists(filename)
alt
Filename nonexistent
M_ACK: AAD(IV |s_counter_server)|{DELETE_REQ |Ack_code=2}$_k$|TAG
Filename exists
delete(filename)
alt
File delete success
M_ACK: AAD(IV |s_counter_server)|{DELETE_REQ|Ack_code=0}$_k$|TAG
File delete failed
M_ACK: AAD(IV|s_counter_server)|{DELETE_REQ |Ack_code=1}$_k$|TAG

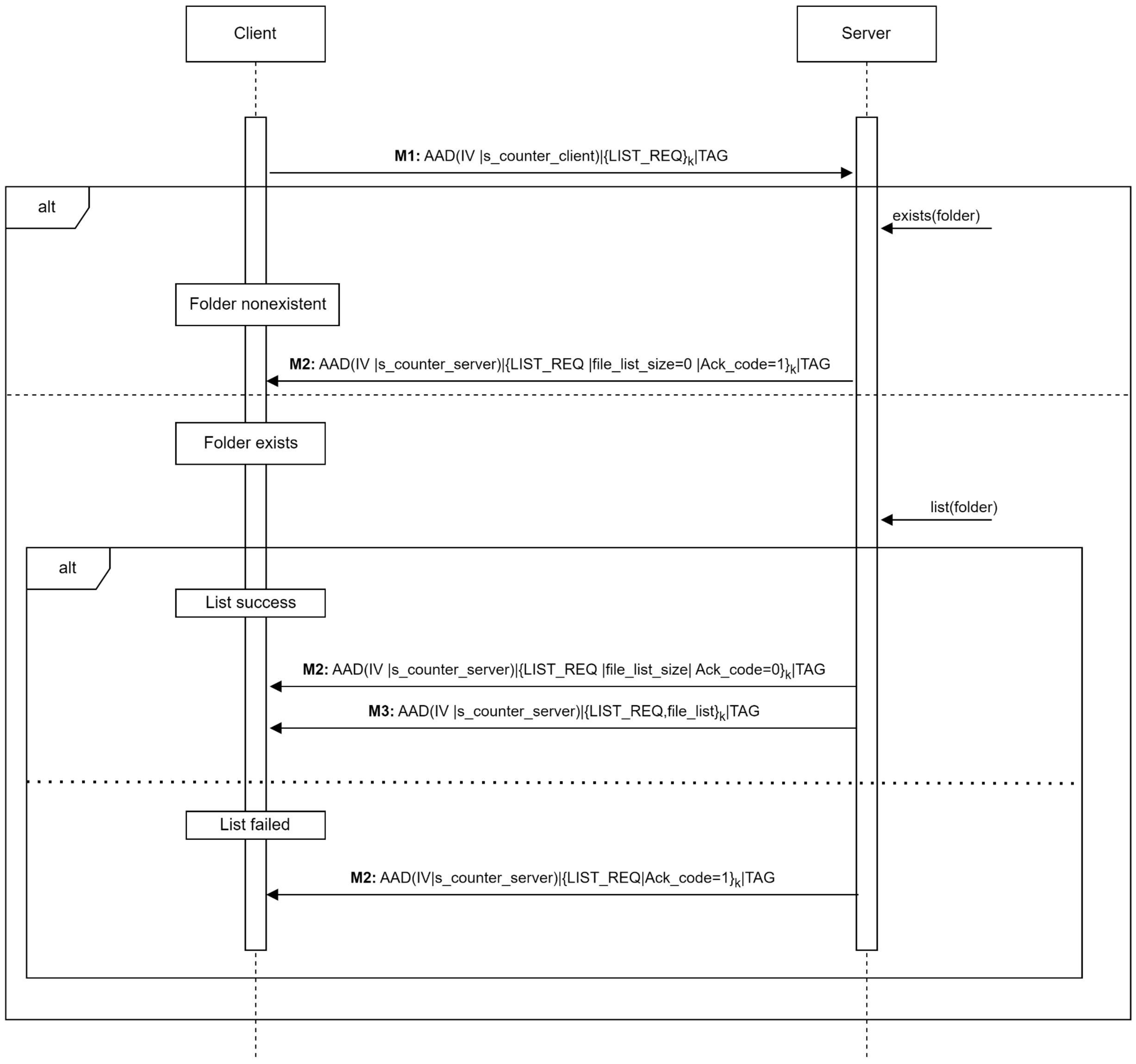
Delete sequence diagram



Delete packet

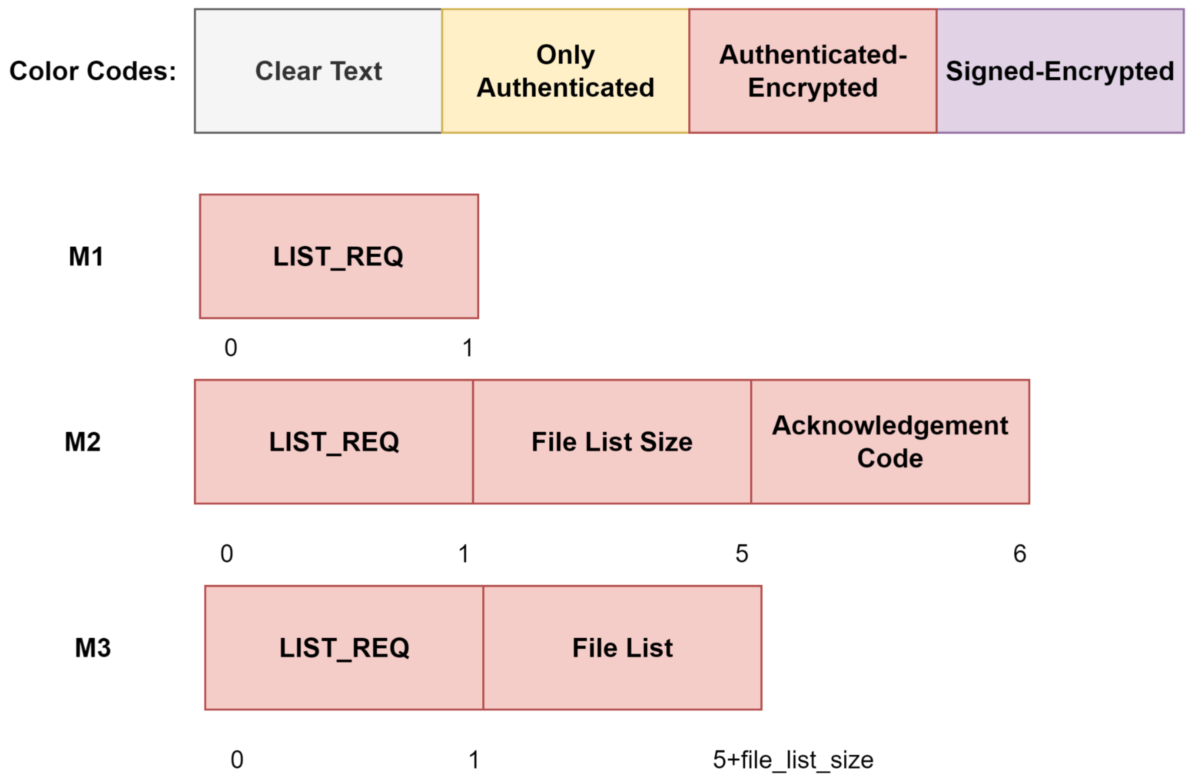
9. List

The client initiates the list protocol by sending a request packet to the server. As a result, the server replies by sending the **size** the of the **file list** and then **the file list** to the client in case of success.



Client
Server
M1: AAD(IV |s_counter_client)|{LIST_REQ}k|TAG
alt
exists(folder)
Folder nonexistent
M2: AAD(IV |s_counter_server)|{LIST_REQ |file_list_size=0 |Ack_code=1}k|TAG
Folder exists
list(folder)
alt
List success
M2: AAD(IV |s_counter_server)|{LIST_REQ |file_list_size| Ack_code=0}k|TAG
M3: AAD(IV |s_counter_server)|{LIST_REQ,file_list}k|TAG
List failed
M2: AAD(IV|s_counter_server)|{LIST_REQ|Ack_code=1}k|TAG

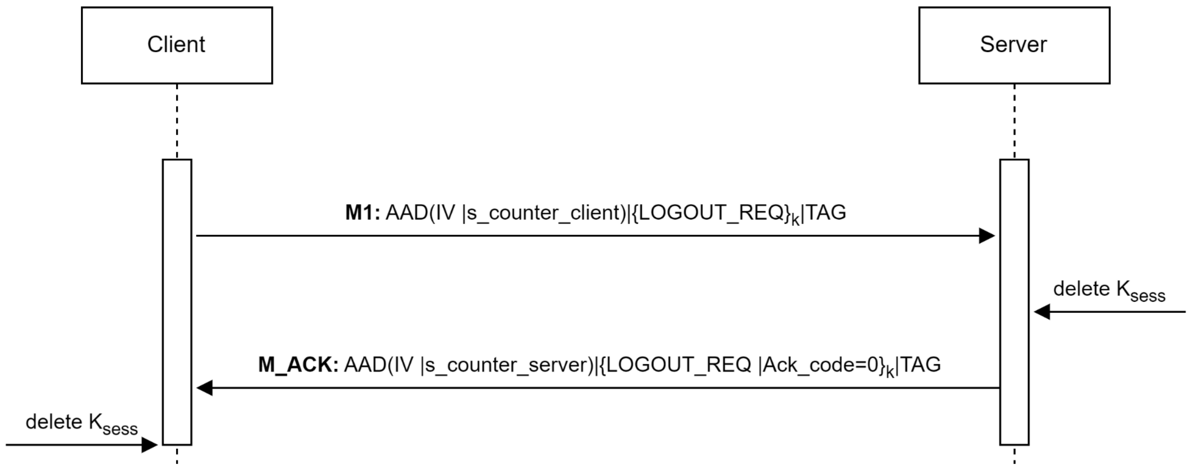
List sequence diagram



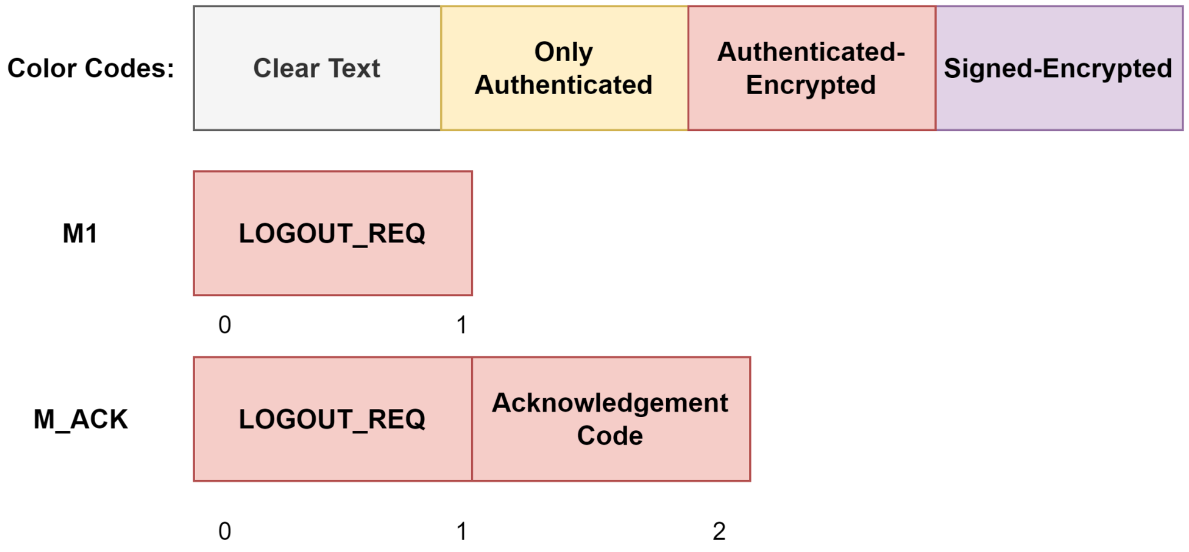
List packets

10. Logout

The client initiates the list protocol by sending a request packet to the server.
As a result, the server **deletes the session key** and sends acknowledgement code to the client, which by consequently deletes the session key also from the client.


Client
Server
M1: AAD(IV |s_counter_client)|{LOGOUT_REQ}$_k$|TAG
delete K_{sess}
M_ACK: AAD(IV |s_counter_server)|{LOGOUT_REQ |Ack_code=0}$_k$|TAG
delete K_{sess}

Logout sequence diagram



Logout packets