UNIVERSITÀ DI PISA

Computer Engineering

Foundations of Cybersecurity

Cloud Storage

Project Documentation

TEAM MEMBERS:
Biagio Cornacchia
Gianluca Gemini
Matteo Abaterusso

Academic Year: 2022/2023

Contents

1 Introduction

The project consists of a **secure cloud storage**. Each **client** connects to a centralized **server** that allows to perform operations on its **private** dedicated storage. In order to access to his storage, a user must **login**, after that he can execute the following operations:

- **List**: shows all files currently on the storage
- **Download**: retrieves a file from the storage to the local file system
- **Upload**: loads a file from the local file system to the storage
- **Rename**: changes the name of a file on the storage
- **Delete**: removes a file from the storage
- **Logout**: closes the connection to the server

The login operation creates a **secure connection** between the server and a client through the negotiation of a set of **session keys**.

The cloud storage application has to guarantee the following requirements:

- The key negotiation has to provide the *Perfect Forward Secrecy*
- The client and server communication has to be **encrypted** and **authenticated**
- The client and server communication has to be protected against **replay attacks**

Client and server authenticate each other using their own **private and public key pairs**. The server already knows all the registered users' keys and the client retrieves the server's key through the **certificate** released by a **Certification Authority**.

2 Design Choices

Starting from the **encryption** requirement, the *AES256* has been chosen. In order to encrypt message longer than a block (128 bit) the chosen encryption mode is the *CBC* that is **CPA-secure**.

Instead, to guarantee the **authentication** requirement, the *HMAC* is used. This is based on the *SHA256* algorithm and guarantee also the **message integrity**. The mount schema used is *encrypt than MAC*, that is **CCA-secure**.

About the **Perfect Forwarding Secrecy** requirement, the login phase is based on the *Station to Station* protocol that uses the *ephemeral Diffie-Hellman key exchange* protocol. At the end of the key exchange, the server and the client share the same secret and the *SHA512* algorithm is used on it to obtain the two 256 bits keys needed for the *AES256* and the *HMAC*.

When a key negotiation ends, a **session** starts. To ensure the **replay attacks resistance**, every packet uses a **counter** as fresh quantity, that avoids reusing old packets. The **counter** is incremented for each message and when its maximum value is reached, new session keys have to be negotiated.

Either client and server are developed using *C++17* for Linux systems. The communication is implemented through *POSIX TCP sockets*, that guarantee **packet delivery**, **error checking** and **congestion control**. During the packets exchange, a possible different **endiannes** between client and server is considered.

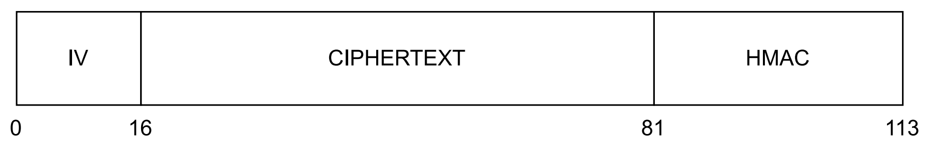
In order to communicate with many clients at the same time, the server associates a new **worker thread** to each of them (just before the login phase).

3 Protocols

In this section, it is shown the structure of the exchanged packets and the protocols' sequence diagrams.

3.1 Generic Packet

As said before, the packets used by the protocols have to be encrypted and the *HMAC* has to be provided. For this reason, it has been defined the **Generic Packet structure** shown below:

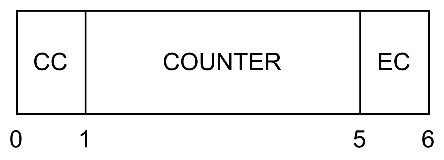


The first field of the generic packet is the **Initialization Vector** (IV) in clear which is needed for the *CBC* encryption mode. Then, the second field is the protocol packet to be sent, encrypted using *AES256*. Finally, the last field contains the *HMAC* of the previous fields concatenated.

It is necessary to highlight that every protocol packet starts with the same two fields which are the **COMMAND_CODE** (CC) used to identify and deserialize accordingly the packet, and the **COUNTER** (discussed in Section 2).

3.2 Result Packet

The majority of the protocols need to communicate the **result** of an operation. For this reason, it has been defined a **common packet** representing the result:

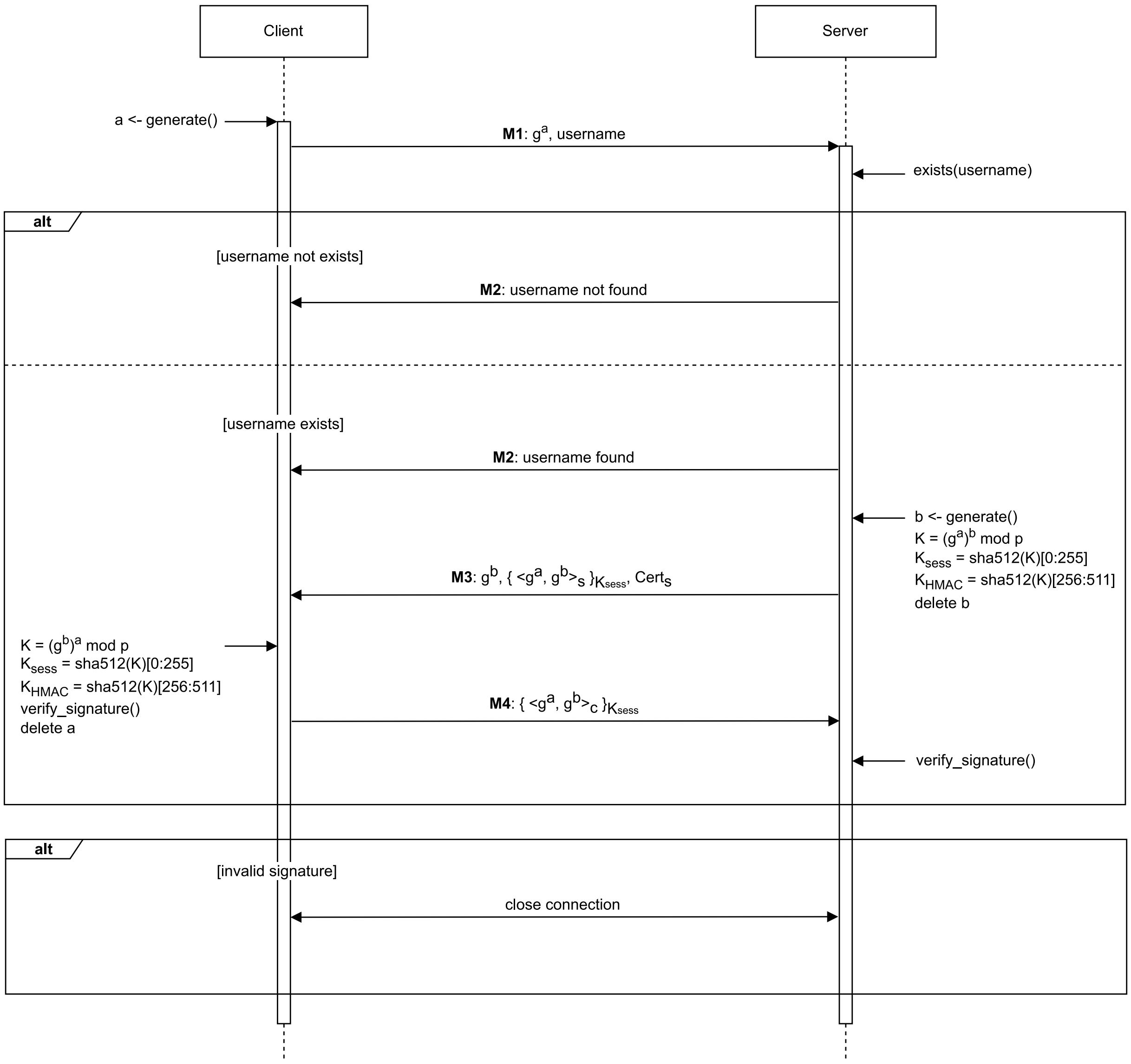


The result is represented by the **COMMAND_CODE** that can be *REQ_SUCCESS* or *REQ_FAILED*. In case of failure, the **ERROR_CODE** (EC) field contains a code that identifies the type of error occurred.

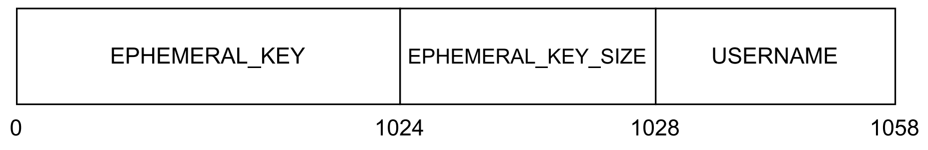
Since the packet size must be **fixed**, if the packet is a *REQ_SUCCESS*, the **ERROR_CODE** field is filled with a **random byte**.

3.3 Login

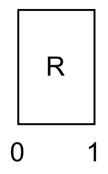
The first protocol to analyze is the login used to **negotiate** the **session keys**. Obviously, it doesn't use the **Generic Packet** since the session keys haven't yet been established.



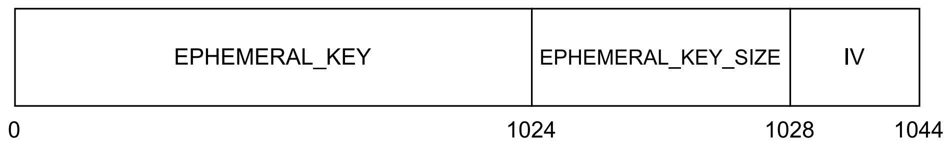
The communication starts with the client that sends the generated ephemeral key to the server and the username. Since the *OpenSSL* library doesn't generate serialized keys with a **fixed size**, the **EPHEMERAL_KEY** field size is an **upper bound** of the possible key sizes. So, the **EPHEMERAL_KEY_SIZE** field is needed to specify the exact size of the generated key.

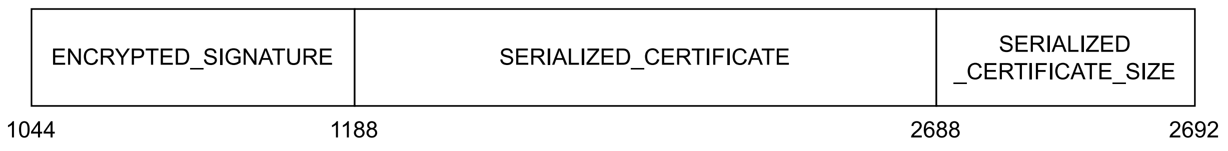


At this point, the server checks if the username belongs to a registered user and provides the operation result to the client. The used packet is the following:

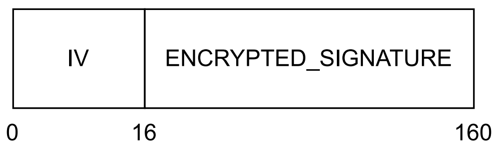


If the username is valid, the server generates its ephemeral key and combines it to the received one in order to calculate the session keys (as described in Section 2). Then, it sends its ephemeral key, the combination of the two ephemeral keys **signed** with its long term private key and **encrypted** with the session key and the certificate containing its public key. For the same reason mentioned before, the ephemeral key and the certificate are sent along with their size. Finally, in order to **decrypt** the field containing the ephemeral key signature, the packet has to contain the IV.



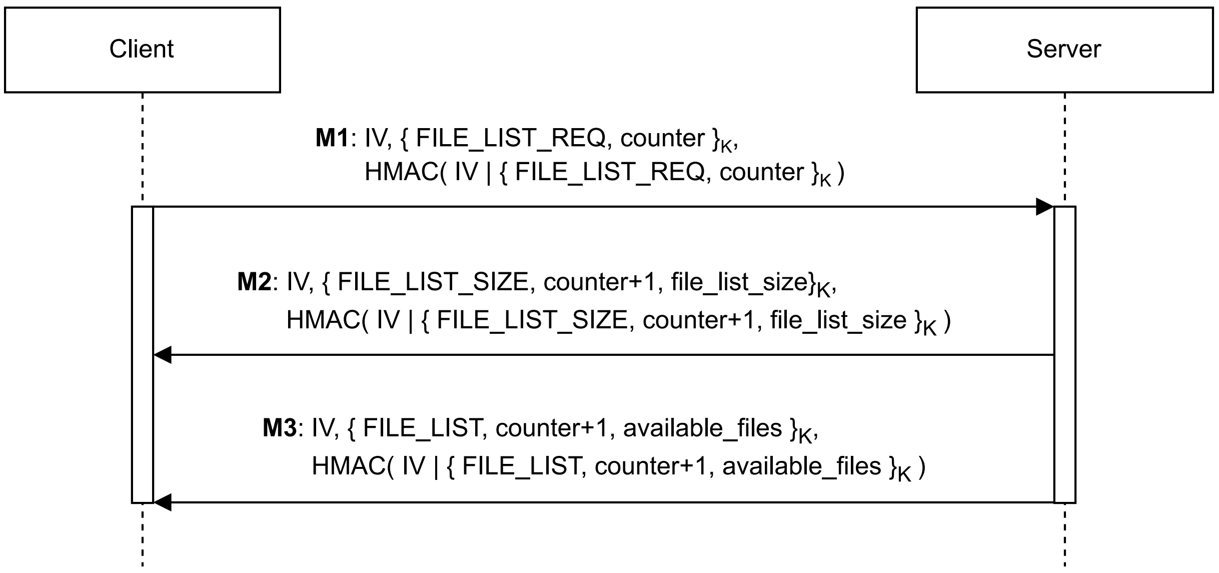


Then, the client calculates the session keys, **decrypts** and **verifies** the received signature and replies with its version of ephemeral keys, signing them with its long term private key and encrypting them with the newly generated session key.



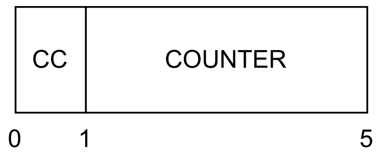
If any of these steps fails, the connection will be closed and a new login phase must be **started again**.

3.4 List

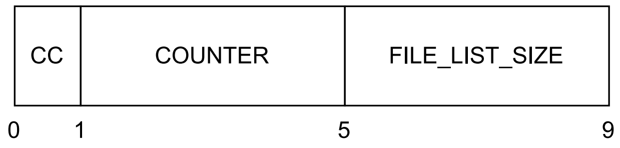


Client
Server
M1: IV, { FILE_LIST_REQ, counter }$_K$,
HMAC(IV | { FILE_LIST_REQ, counter }$_K$)
M2: IV, { FILE_LIST_SIZE, counter+1, file_list_size}$_K$,
HMAC(IV | { FILE_LIST_SIZE, counter+1, file_list_size }$_K$)
M3: IV, { FILE_LIST, counter+1, available_files }$_K$,
HMAC(IV | { FILE_LIST, counter+1, available_files }$_K$)

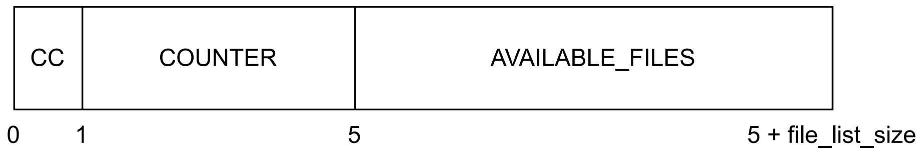
The packet that starts the list protocol contains the **COMMAND_CODE** equals to *FILE_LIST_REQ* and the **COUNTER**.



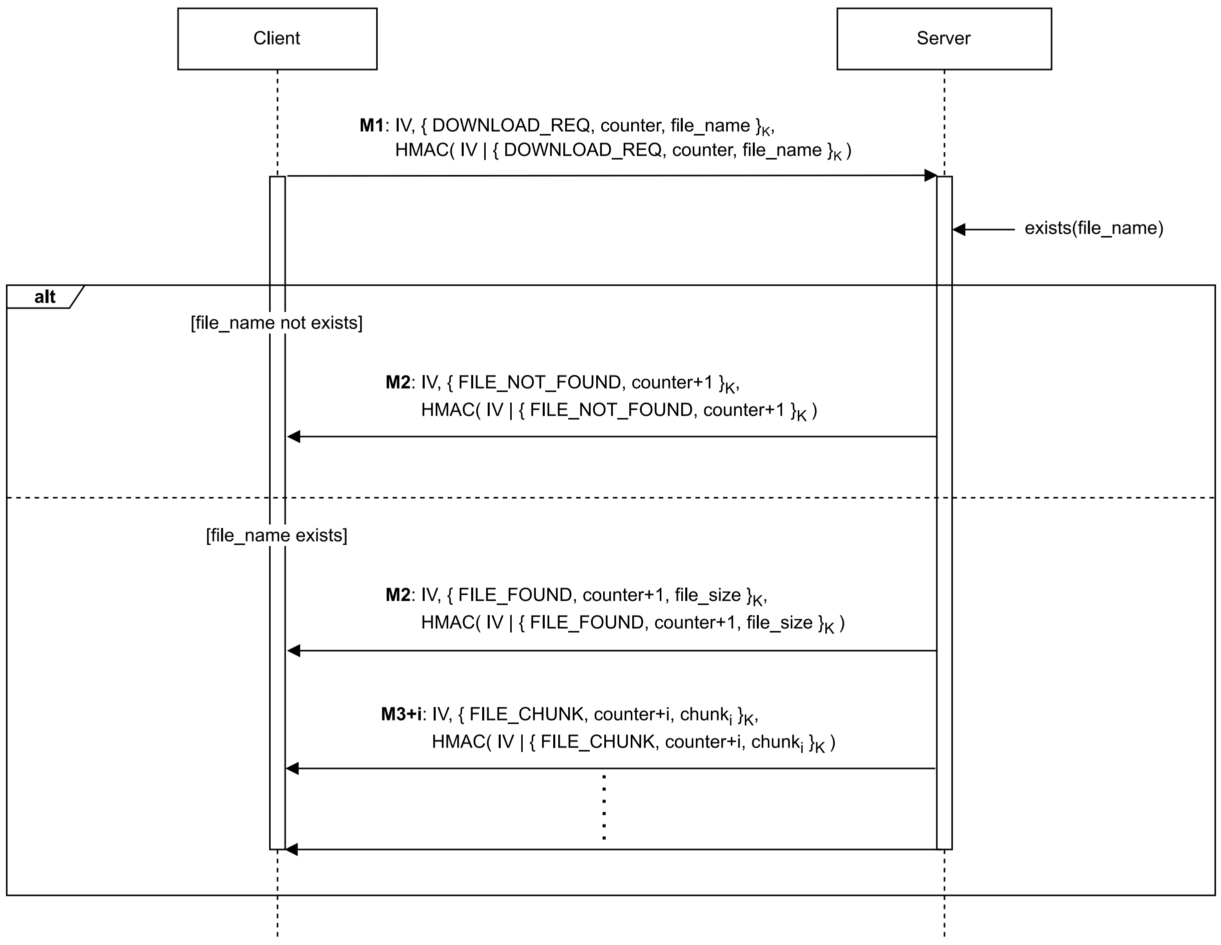
Since the file list size is **not predictable** a priori, the first server reply, with **COMMAND_CODE** equals to *FILE_LIST_SIZE*, has to contain this information.



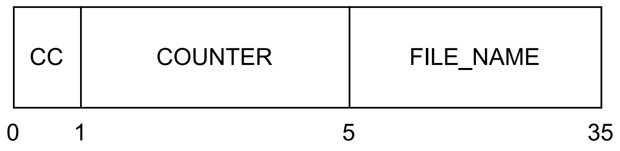
After that, the server sends a packet containing the list of the file names, divided by the '|' character. In this case, the **COMMAND_CODE** is *FILE_LIST*.



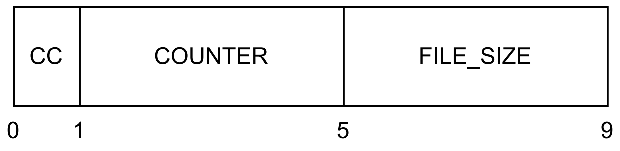
3.5 Download


Client
Server
M1: IV, { DOWNLOAD_REQ, counter, file_name }$_K$,
HMAC(IV | { DOWNLOAD_REQ, counter, file_name }$_K$)
exists(file_name)
alt
[file_name not exists]
M2: IV, { FILE_NOT_FOUND, counter+1 }$_K$,
HMAC(IV | { FILE_NOT_FOUND, counter+1 }$_K$)
[file_name exists]
M2: IV, { FILE_FOUND, counter+1, file_size }$_K$,
HMAC(IV | { FILE_FOUND, counter+1, file_size }$_K$)
M3+i: IV, { FILE_CHUNK, counter+i, chunk$_i$ }$_K$,
HMAC(IV | { FILE_CHUNK, counter+i, chunk$_i$ }$_K$)

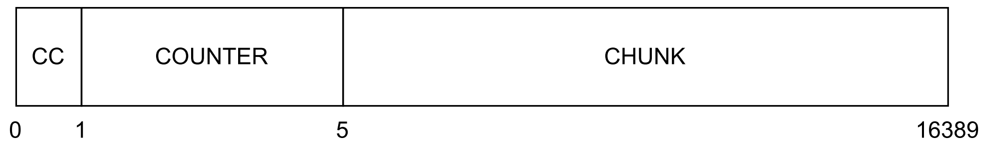
The client starts the download protocol sending to the server the name of the file to download. The **COMMAND_CODE** contained in the packet is *DOWNLOAD_REQ*.



Then, the server checks if the file is on the user's cloud and provides the operation result to the client. If the file exists, the **COMMAND_CODE** will be equal to *FILE_FOUND* and the packet will contain the file size, otherwise the **COMMAND_CODE** will be equal to *FILE_NOT_FOUND* and the file size field will be filled with **random bytes**.

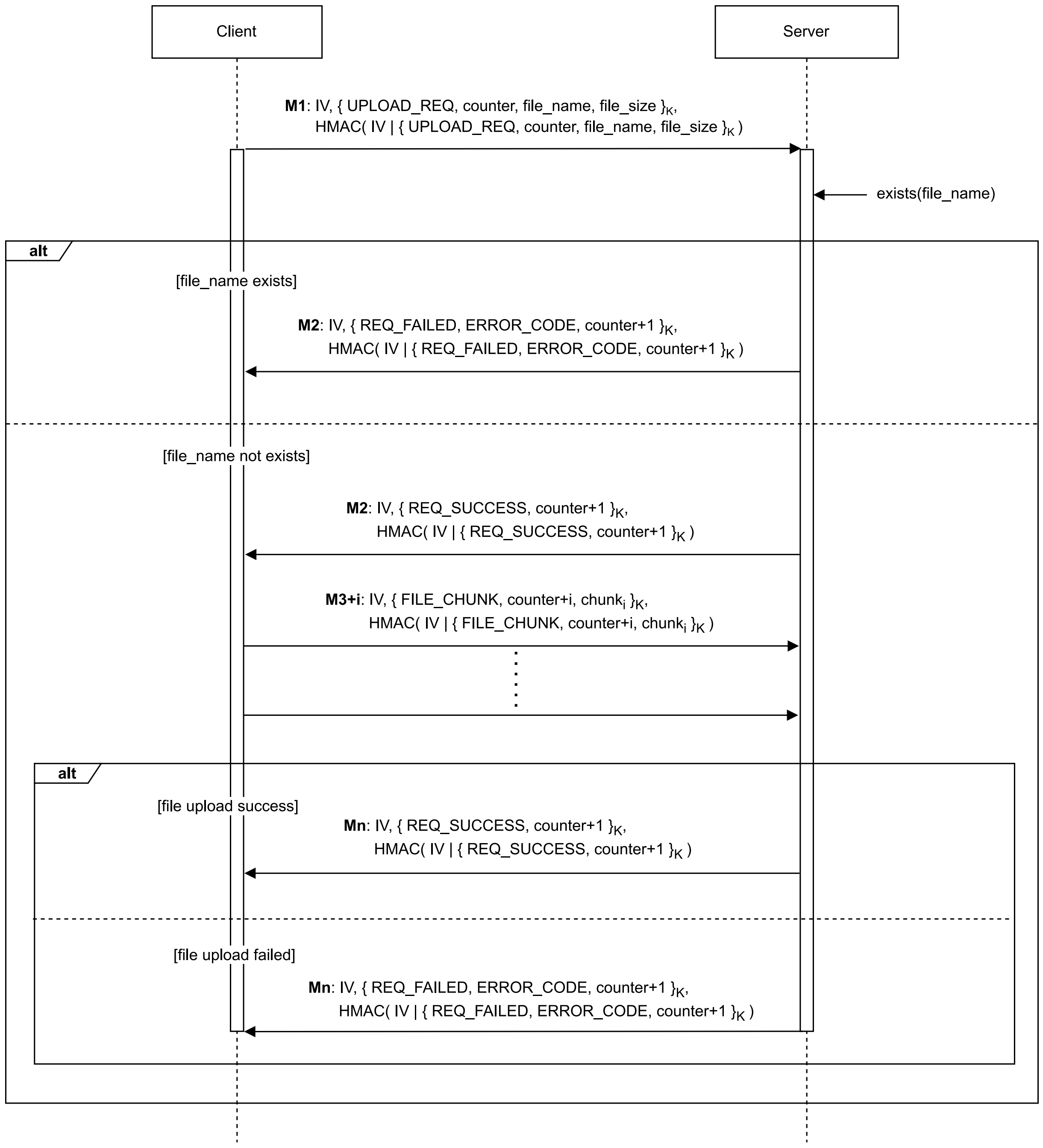


From this point on, the server starts sending the file divided into **chunks** having a **fixed size**. The chosen size is *16KiB* and it is known a priori by both client and server. Since the client has received the file size previously, it is **aware** of the number of chunks that it will receive. These packets will have *FILE_CHUNK* as **COMMAND_CODE**.

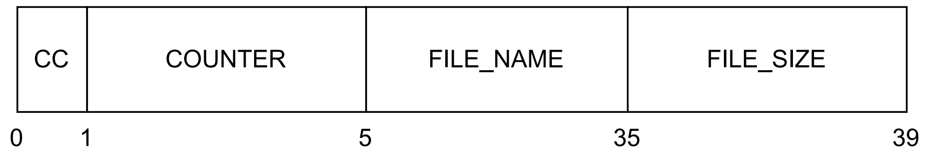


If the download fails, **no message** will be sent to the server and the user will have to request a **new download operation**.

3.6 Upload

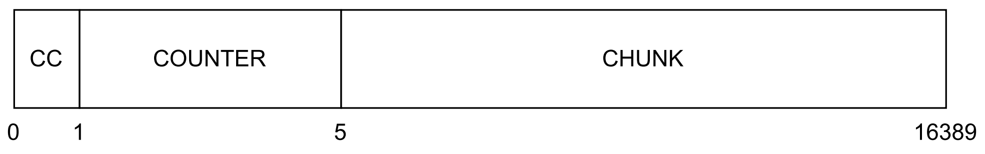

Client
Server
M1: IV, { UPLOAD_REQ, counter, file_name, file_size }K, HMAC(IV | { UPLOAD_REQ, counter, file_name, file_size }K)
exists(file_name)
alt
[file_name exists]
M2: IV, { REQ_FAILED, ERROR_CODE, counter+1 }K, HMAC(IV | { REQ_FAILED, ERROR_CODE, counter+1 }K)
[file_name not exists]
M2: IV, { REQ_SUCCESS, counter+1 }K, HMAC(IV | { REQ_SUCCESS, counter+1 }K)
M3+i: IV, { FILE_CHUNK, counter+i, chunki }K, HMAC(IV | { FILE_CHUNK, counter+i, chunki }K)
alt
[file upload success]
Mn: IV, { REQ_SUCCESS, counter+1 }K, HMAC(IV | { REQ_SUCCESS, counter+1 }K)
[file upload failed]
Mn: IV, { REQ_FAILED, ERROR_CODE, counter+1 }K, HMAC(IV | { REQ_FAILED, ERROR_CODE, counter+1 }K)

The client starts the upload protocol sending to the server the name of the file to upload and its size. The **COMMAND_CODE** contained in the packet is *UPLOAD_REQ*. Since the maximum allowed size is *4GiB*, the file size is **checked** on **client-side**, before starting the protocol.



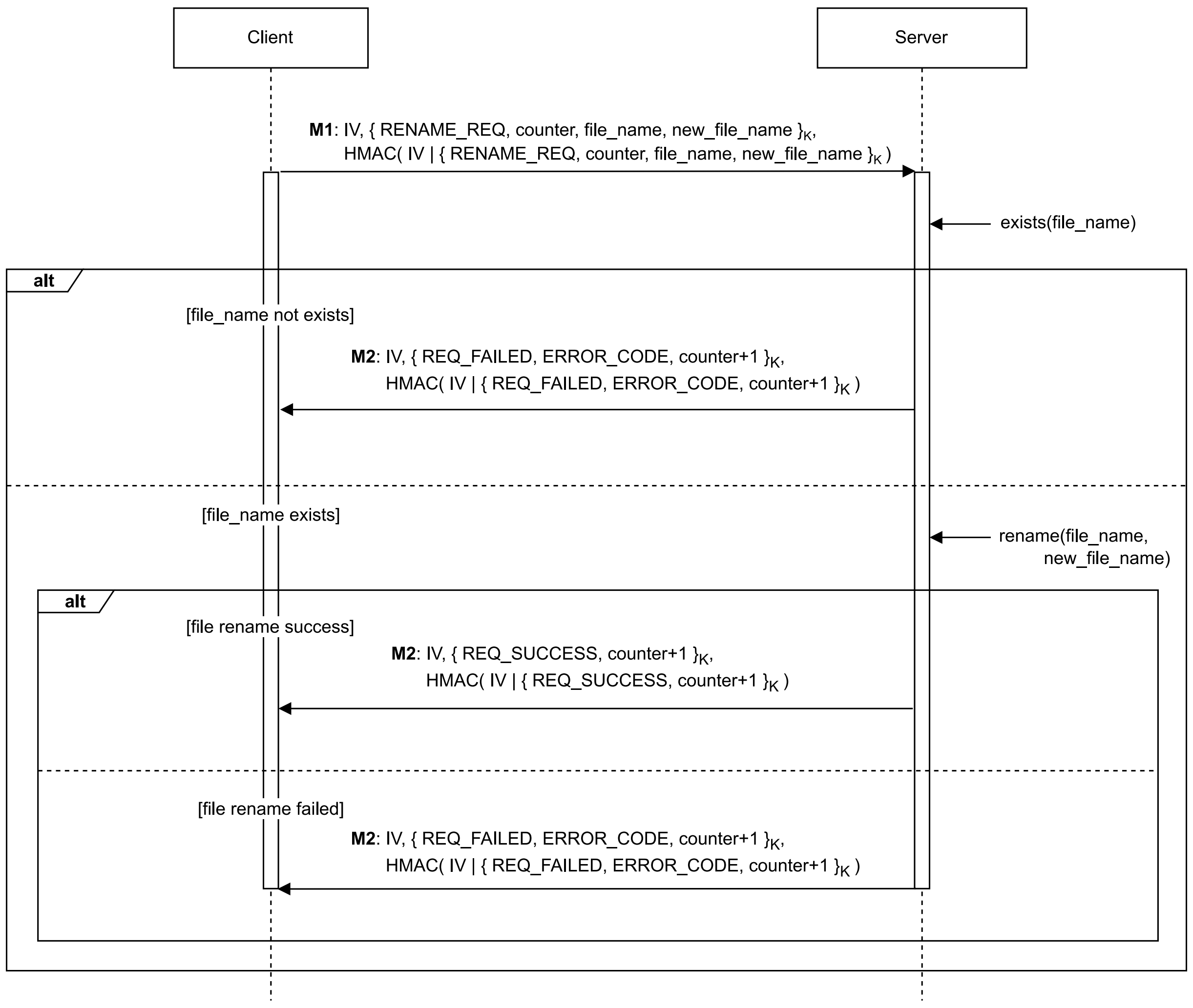
The server has to check if there is another file with the same name. The result of this operation is sent to the client using a **Result** packet (introduced in Section 3.2).

At this point, the client starts sending the file divided into **chunks** with **fixed size** (as already described for the download operation in Section 3.5). In this case the **COMMAND_CODE** is *FILE_CHUNK*.



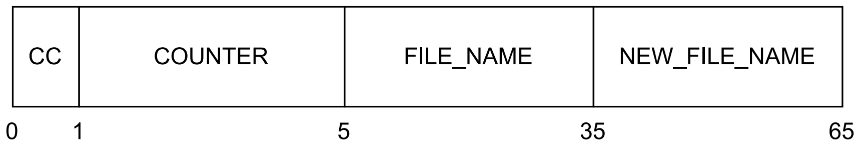
Finally, the server sends a **Result** packet representing the result of the upload operation.

3.7 Rename



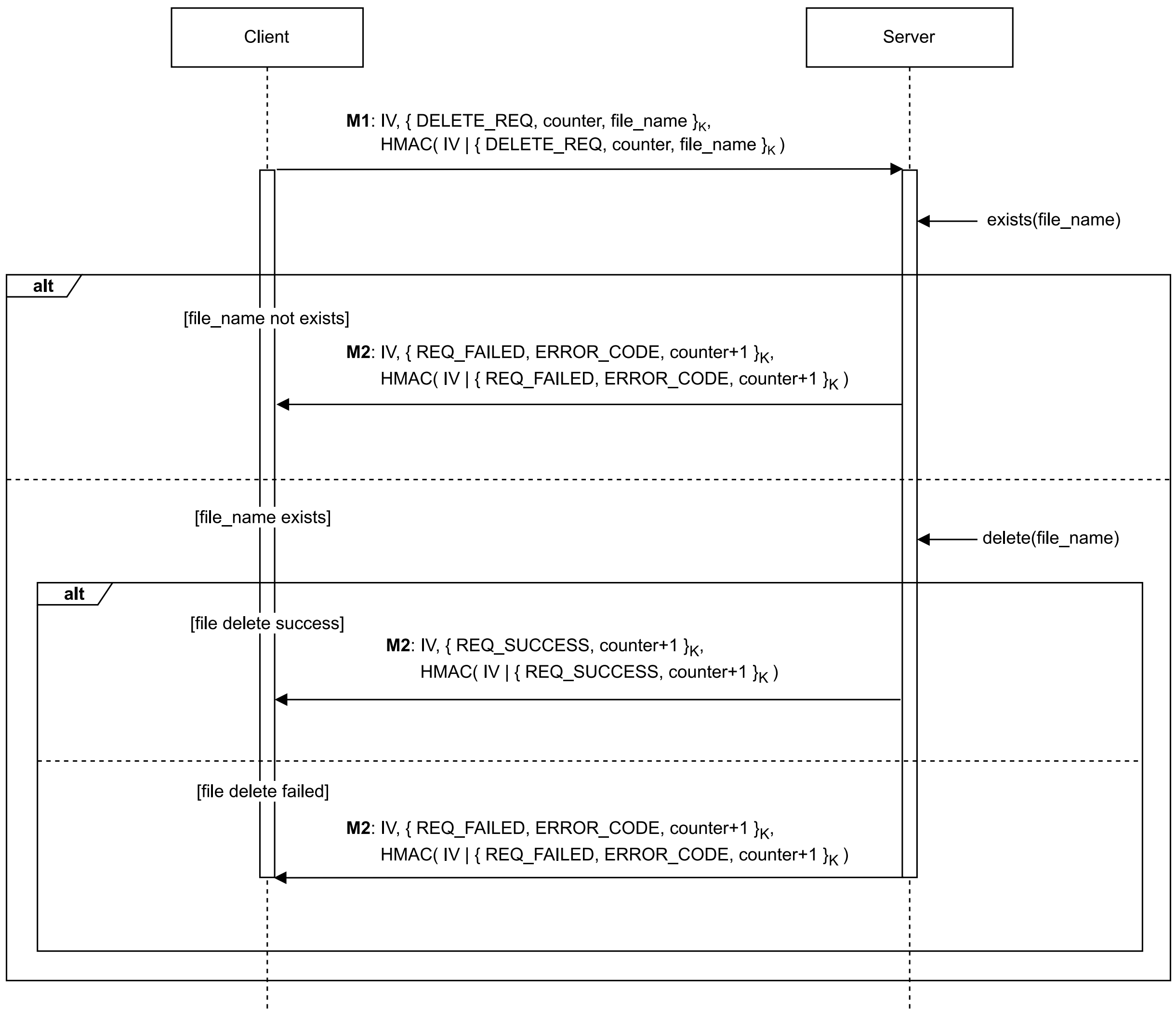
Client
Server
M1: IV, { RENAME_REQ, counter, file_name, new_file_name }K, HMAC(IV | { RENAME_REQ, counter, file_name, new_file_name }K)
exists(file_name)
alt
[file_name not exists]
M2: IV, { REQ_FAILED, ERROR_CODE, counter+1 }K, HMAC(IV | { REQ_FAILED, ERROR_CODE, counter+1 }K)
[file_name exists]
rename(file_name, new_file_name)
alt
[file rename success]
M2: IV, { REQ_SUCCESS, counter+1 }K, HMAC(IV | { REQ_SUCCESS, counter+1 }K)
[file rename failed]
M2: IV, { REQ_FAILED, ERROR_CODE, counter+1 }K, HMAC(IV | { REQ_FAILED, ERROR_CODE, counter+1 }K)

The client starts the rename protocol sending to the server the name of the file to rename and the new file name. The **COMMAND_CODE** contained in the packet is *RENAME_REQ*.



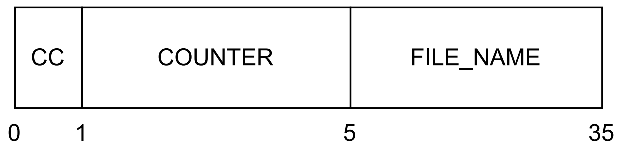
The operation result is communicated using a **Result** packet.

3.8 Delete



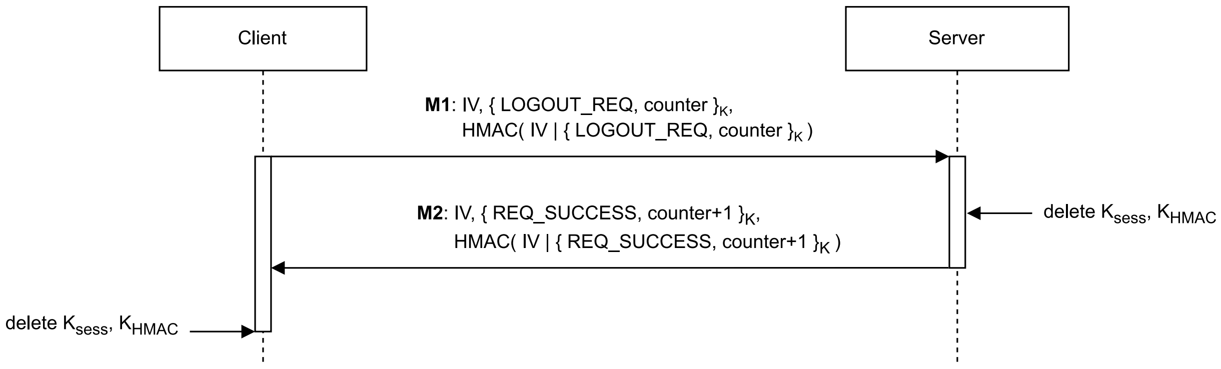
Client
Server
M1: IV, { DELETE_REQ, counter, file_name }K,
HMAC(IV | { DELETE_REQ, counter, file_name }K)
exists(file_name)
alt
[file_name not exists]
M2: IV, { REQ_FAILED, ERROR_CODE, counter+1 }K,
HMAC(IV | { REQ_FAILED, ERROR_CODE, counter+1 }K)
[file_name exists]
delete(file_name)
alt
[file delete success]
M2: IV, { REQ_SUCCESS, counter+1 }K,
HMAC(IV | { REQ_SUCCESS, counter+1 }K)
[file delete failed]
M2: IV, { REQ_FAILED, ERROR_CODE, counter+1 }K,
HMAC(IV | { REQ_FAILED, ERROR_CODE, counter+1 }K)

The client starts the delete protocol sending to the server the name of the file to delete. The **COMMAND_CODE** contained in the packet is *DELETE_REQ*.

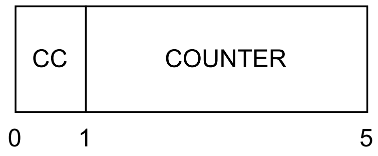


The operation result is communicated using a **Result** packet.

3.9 Logout


Client
Server
M1: IV, { LOGOUT_REQ, counter }$_K$,
HMAC(IV | { LOGOUT_REQ, counter }$_K$)
M2: IV, { REQ_SUCCESS, counter+1 }$_K$,
HMAC(IV | { REQ_SUCCESS, counter+1 }$_K$)
delete K_{sess}, K_{HMAC}
delete K_{sess}, K_{HMAC}

The logout operation starts when the client sends to the server a packet containing the *LOGOUT_REQ* as **COMMAND_CODE**.



Then, the server **deletes** the **session keys** and send a **Result** packet to the client. Once received, similarly the client **deletes** the **session keys**.