



Università degli Studi di Pisa

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Cybersecurity

Progetto Applied Cryptography

Piero Turri

Giacomo Viaggi

Giacomo Vitangeli

Anno Accademico 2021-2022

Indice

1	Introduzione	1
2	Scelte progettuali	2
2.1	Autenticazione client-server	2
2.2	Gestione del Nonce e dei counter Client-Server	5
2.3	Funzionalità dello Storage Cloud	5
2.3.1	List	5
2.3.2	Upload	6
2.3.3	Download	8
2.3.4	Rename	9
2.3.5	Delete	10
2.4	Logout client	12
3	Implementazione	13
3.1	Autenticazione	13
3.2	Costruzione del messaggio	13
3.2.1	Confidenzialità e Autenticazione dei pacchetti	13
3.3	Gestione delle funzionalità del Cloud Storage	13
3.3.1	Gestione upload e download dei file	14
3.3.2	Gestione Directory Traversal	15
4	Gestione degli utenti	15
5	Gestione degli errori	16

1 Introduzione

Nel seguente report, vengono delineate le scelte progettuali ad alto livello, comprese degli schemi temporali riguardo lo scambio dei messaggi e il loro formato dettagliato con la motivazione delle scelte prese. Secondariamente sono riportate le scelte implementative, la gestione degli errori e la gestione degli utenti.

Specifiche del progetto

Le specifiche del progetto prevedono la progettazione ed implementazione di un applicazione di tipo Client-Server che deve rispecchiare le principali funzionalità garantite da un Cloud Storage. In particolare, devono essere presi in considerazione gli aspetti relativi alla sicurezza dello stesso; tra le misure di sicurezza richieste vi è l'autenticazione tra Client e Server mediante l'uso di chiavi pubbliche condivise, precedentemente tra le due parti, e la negoziazione della chiave di sessione simmetrica, che dovrà garantire Perfect Forward Secrecy.

Inoltre, l'intera sessione deve essere cifrata, autenticata ed essere resistente ad attacchi di tipo replay.

Per l'utilizzo delle funzioni crittografiche viene utilizzata la libreria OpenSSL, una libreria open source che implementa le funzionalità disponibili nei protocolli SSL e TLS.

Il progetto è stato implementato in C, adottando i principi del secure coding, al fine di evitare la presenza di vulnerabilità date da un'implementazione non sicura.

2 Scelte progettuali

Per soddisfare i requisiti del progetto, sono state fatte delle scelte di progettazione ben definite.

Nei paragrafi seguenti vengono spiegate le scelte fatte analizzando il diagramma della connessione tra client e server delle funzioni di autenticazione, di upload, di download, di delete, di rename e di logout. Insieme al diagramma vengono analizzati anche tutti i messaggi scambiati.

Per rappresentare i diversi livelli di segretezza dei componenti di un messaggio, sono stati utilizzati colori differenti. Qui una legenda:

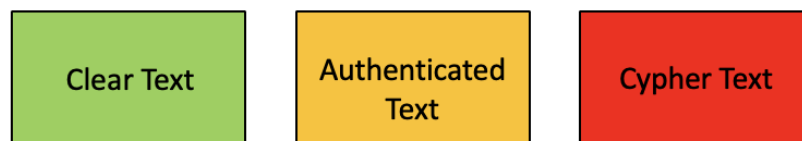


Figura 2.1: Legenda segretezza componenti all'interno di un messaggio.

2.1 Autenticazione client-server

L'autenticazione è l'atto di confermare la verità di un attributo di una singola parte di dato o di una informazione sostenuto vero da un'entità.

L'autenticazione tra client e server insieme con lo scambio della chiave avvengono tramite lo schema **RSA effimero**. In particolare, il modello di autenticazione che è stato implementato, è descritto dal seguente diagramma:

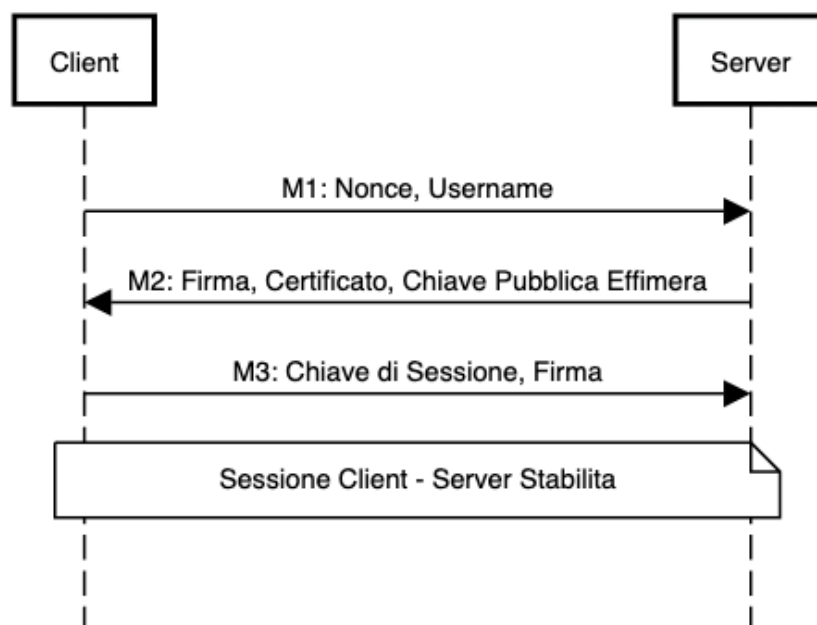


Figura 2.2: Schema dei messaggi per l'autenticazione tra client e server

Nel messaggio M1 (figura 2.3) il client invia al server un messaggio contenente il suo **username**, in modo da richiedere la fase di login/autenticazione, e quantità fresh, detta **nonce**, che gli servirà, leggendo la risposta del server, ad essere sicuro di non essere soggetto ad un attacco di tipo replay.

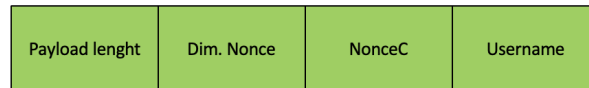


Figura 2.3: Messaggio M1, inviato dal client al server per l'autenticazione

Ricevuto il messaggio M1, il server genera una coppia di chiavi, una pubblica e una privata, dette effimere perché utili soltanto allo scambio della chiave di sessione ed eliminate non appena raggiunto lo scopo. La chiave pubblica effimera servirà al client per inviare in maniera confidenziale, nel messaggio in figura 2.5, la chiave di sessione per parlare con il server.

Nel messaggio M2 (figura 2.4) il server invia una quantità firmata con la propria chiave privata, composta dal nonce del client (*NonceC*) concatenato alla chiave pubblica effimera, e il suo certificato in modo da provare la sua autenticità al client. Inoltre, nel messaggio viene anche inviata la chiave pubblica effimera in chiaro, con un duplice scopo: permettere al client di verificare la validità della firma del server, e cifrare la chiave di sessione nel messaggio M3.

Il nonce viene inserito soltanto all'interno della quantità firmata, proprio perchè il client, avendolo generato, lo possiede.

Una mancata verifica della firma potrebbe essere segno di errore nella quantità nonce e perciò sintomo di un replay attack. In questo caso, il processo di autenticazione viene interrotto e il client dovrà ricominciare.

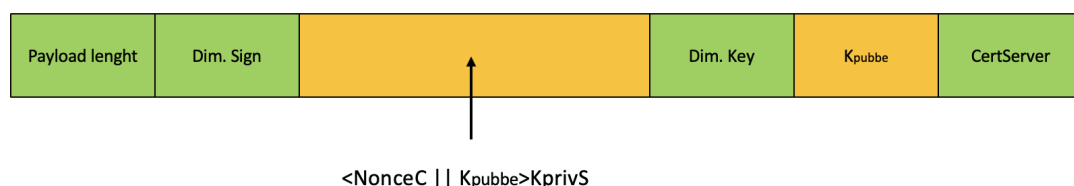


Figura 2.4: Messaggio M2, inviato dal server al client con la chiave pubblica effimera

Ricevuta la chiave pubblica effimera e verificata l'autenticità del server, il client genera la chiave di sessione e successivamente la invia al server. Generata la chiave, per garantire la sua confidenzialità, il client la cifrerà utilizzando la chiave pubblica effimera ricevuta dal server in modo che solo quest'ultimo possa decifrarla con la corrispondente chiave privata effimera.

Per proteggere il messaggio da attacchi di tipo replay, viene inserita nel messaggio una quantità così costruita:

$$\langle K_{pubbe} || \{SessionKey\}_{k_{pubbe}} \rangle_{k_{privC}}$$

Questa quantità viene utilizzata come quantità fresh poichè il server, avendo generato la coppia di chiavi effimere, può verificare la chiave pubblica effimera inserita nella quantità 2.1.

Oltre alla firma, il messaggio M3 conterrà la chiave di sessione cifrata ed alcuni parametri utili al server per decifrarla con il meccanismo della **Digital Envelope**. Firmando la quantità fresh, il client prova la sua autenticità al server, che essendo già in possesso delle chiavi pubbliche di ogni client, può quindi verificare la validità della firma prendendo la chiave pubblica corrispondente all'username che ha ricevuto nel messaggio M1.

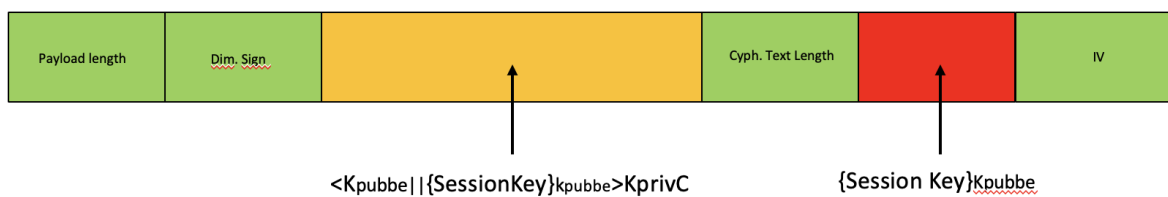


Figura 2.5: Messaggio M3, invio della chiave di sessione dal client al server

Il processo di autenticazione finisce con la decifrazione della chiave di sessione da parte del server. Inoltre, dopo che il server ha decifrato la chiave di sessione, vengono eliminate le chiavi effimere, mentre la chiave di sessione verrà eliminata con la procedura di logout da parte del client. Dopo il messaggio M3, il client e il server possono comunicare cifrando ogni messaggio tramite la chiave di sessione appena scambiata.

L'utilizzo della coppia di chiavi effimere garantisce la **Perfect Forward Secrecy**, ovvero la protezione delle sessioni passate in caso di compromissione della chiave privata "long term" del server.

I messaggi che seguono lo scambio della chiave di sessione sono tutti cifrati con **AES GCM 256**, che permette lo scambio di messaggi sia cifrati che autenticati. Tra le quantità inviate nei messaggi troviamo gli *AAD*, ovvero le quantità non confidenziali ma di cui si desidera provare l'autenticità, il *testo cifrato*, il *tag* (che serve all'algoritmo di decifratura per verificare l'autenticità del messaggio) e l'*IV*.

2.2 Gestione del Nonce e dei counter Client-Server

All'interno del progetto sono stati utilizzati *nonce* e *counter* al fine di evitare attacchi di tipo replay. In particolare, viene utilizzato un solo nonce, generato dal client per utilizzare una quantità "*fresh*" ed evitare replay attack.

In tutte le altre funzioni del Cloud Storage, viene utilizzato un counter progressivo e inizializzato a 0, uno per il server e uno per il client. Inoltre, sia il client che il server avranno una copia del counter dell'altro attore in modo da riuscire a verificare l'integrità dei messaggi scambiati. Nel caso in cui il counter non rispettasse la progressione, il client verrà immediatamente disconnesso dal server e sarà obbligato a ricompletare la procedura di autenticazione.

2.3 Funzionalità dello Storage Cloud

Come spiegato nell'introduzione il progetto consiste nello sviluppo di un Cloud Storage con delle funzionalità precise:

- List: consente di elencare i documenti presenti sul server;
- Upload: consente di caricare un file sul server;
- Download: consente di scaricare un file dal server;
- Rename: consente di rinominare un file sul server;
- Delete: consente di eliminare un file dal server;
- Logout: consente di effettuare il logout dal server.

Nei seguenti paragrafi viene illustrato lo scambio di messaggi tra client e server per ognuna delle funzioni sopra elencate.

2.3.1 List

La funzione **List** permette all'utente, che si collega al server, di vedere tutti i documenti presenti sul proprio storage cloud.

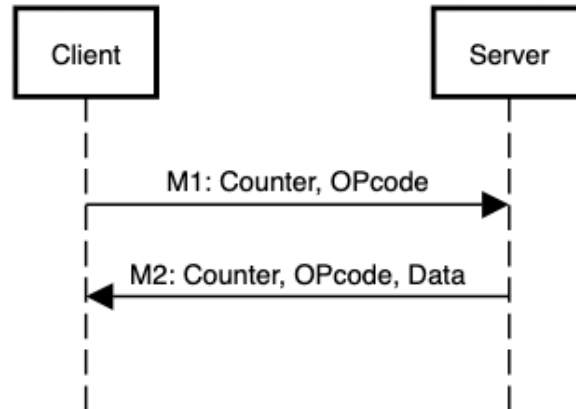


Figura 2.6: Diagramma scambio messaggio tra Client e Server, all'esecuzione del comando List

Tramite il messaggio M1 (figura 2.7) il client richiede al server di elencare i suoi documenti presenti online. Per richiedere ciò, invia un messaggio con il "codice operazione" (**OPCode**) della funzione List e un counter, entrambi autenticati.

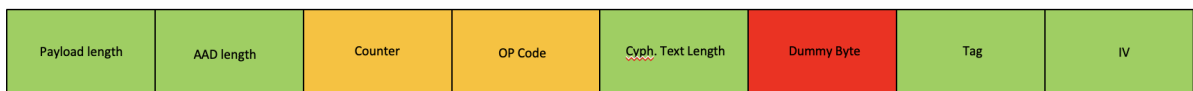


Figura 2.7: Messaggio di richiesta funzione List dal Client verso il Server

Il server, elaborata la richiesta, risponde con i documenti dell'utente che ha effettuato la richiesta. Il messaggio che invierà al client (figura 2.8), oltre ad avere il counter e l'OPCode autenticati, avrà anche un campo "Data", cifrato con la chiave di sessione, che conterrà una lista di tutti i documenti disponibili al client stesso.



Figura 2.8: Messaggio di risposta funzione List dal Server verso il Client

2.3.2 Upload

La funzione di **Upload** permette all'utente di caricare sul proprio spazio cloud un documento con qualsiasi estensione e dimensione massima di 4GB (4.294.967.296 byte).

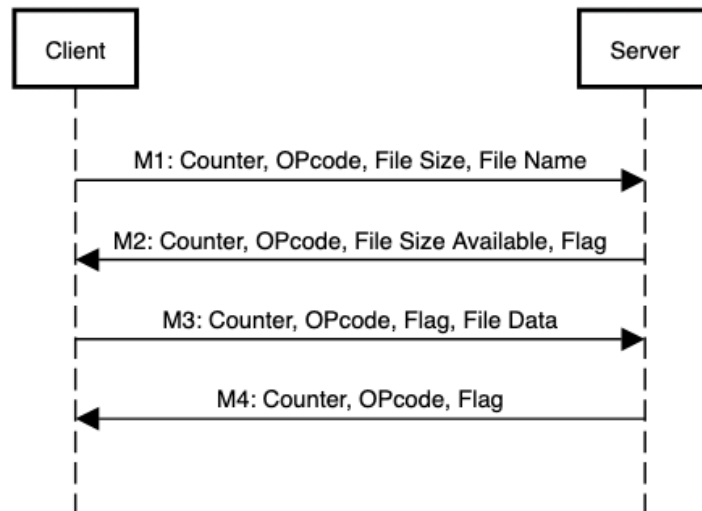


Figura 2.9: Scambio messaggi della funzione Upload tra Server e Client

Nel messaggio M1 (figura 2.10), il client richiede al server di poter caricare un file sul proprio storage cloud inviando nel messaggio l'OPcode specifico e la dimensione del file come campi autenticati, mentre il nome del file cifrato con la chiave di sessione.

Elaborata la richiesta e verificato che la dimensione del file non superi la soglia prefissata, il server risponde con il messaggio M2 (figura 2.11). Nel messaggio viene inserito un campo **flag** che viene utilizzato per far saper al client se l'operazione di Upload è stata autorizzata o se è stata negata:

- flag = 0: l'upload non è autorizzato (potrebbe significare che la dimensione del file eccede il limite massimo oppure che il file è già presente sullo storage cloud);
- flag = 1: l'upload è autorizzato.

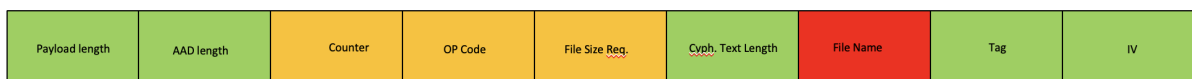


Figura 2.10: Messaggio di richiesta Upload dal Client verso il Server

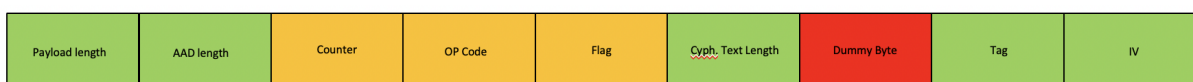


Figura 2.11: Messaggio di risposta dal Server al Client

Se l'Upload viene autorizzato il client inizia ad inviare messaggi al server con "chunk" del file scelto da caricare. In particolare ogni chunk ha dimensione pari a 1048576 Byte (~1MB), per trovare un compromesso tra velocità di cifratura e quantità di dati scambiati.

Il messaggio M3 (figura 2.12) rappresenta il messaggio tipo utilizzato dal client per inviare i chunk di dati al server. In questo messaggio il campo *flag* assume un valore differente. Infatti, con $flag = 0$ l'upload non è finito e il server dovrà aspettare ancora almeno un altro pacchetto prima di ricostruire il file; con $flag = 1$, l'upload è finito e il server può iniziare a ricostruire il file.

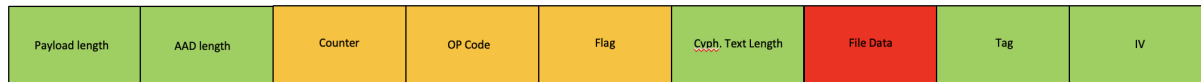


Figura 2.12: Messaggi di upload dal Client verso il Server

Una volta finito l'Upload il server ricostruisce il file mettendo insieme tutti i chunk che il client gli ha inviato e comunicherà l'esito di questa operazione al client tramite il messaggio M4 (figura 2.13). In questo messaggio il campo *flag* assume un terzo significato: nel caso $flag$ sia pari a 0, significa che l'Upload del file è fallito e il server non è riuscito a ricostruire il file; se invece il *flag* è pari a 1, l'Upload del file si è concluso con successo.

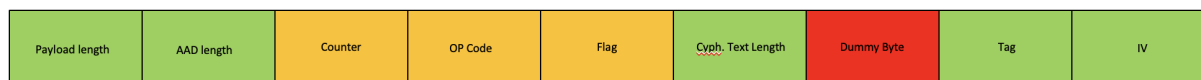


Figura 2.13: Messaggio Upload concluso dal Server verso il Client

2.3.3 Download

La funzione Download permette all'utente di scaricare un file dal server. È simile alla funzione di Upload, infatti il client richiede al server se un file preciso può essere scaricato. Nel caso lo fosse, il server invia tanti messaggi quanti sono i chunk di file e alla fine il client comunicherà al server l'esito dell'operazione.

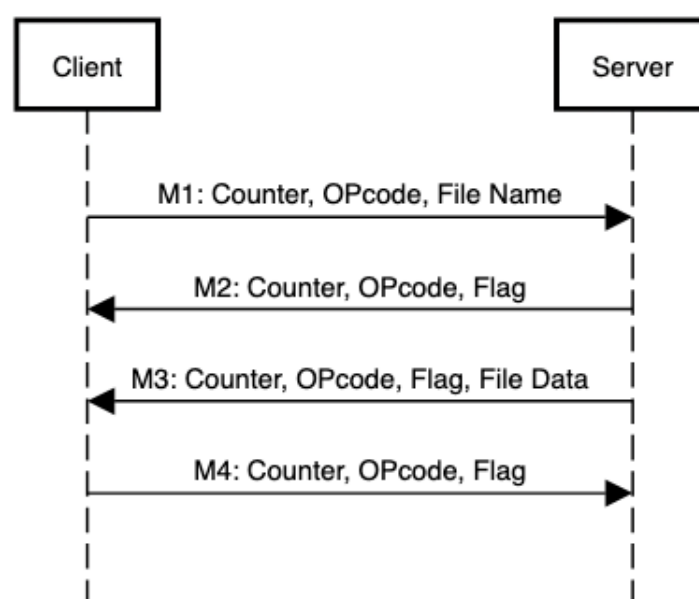


Figura 2.14: Scambio messaggi della funzione Download tra Server e Client

Nel messaggio M1 (figura 2.15) il client richiede di poter scaricare un file dal proprio storage sul server, inoltrando tramite il campo "File Name" il nome del file. Nel messaggio sono presenti anche un counter e l'OPcode dell'operazione di Download.

Come per l'Upload, all'interno del messaggio M2 (figura 2.16) il server utilizza il campo flag per poter autorizzare l'operazione (flag = 0 operazione non consentita, flag = 1 operazione consentita).



Figura 2.15: Messaggio di richiesta Download dal Client verso il Server



Figura 2.16: Messaggio di risposta dal Server al Client

Se il Download può essere eseguito allora il server inizia ad inviare i chunk di dati, appartenenti al file selezionato, al client stesso. Come per l'Upload anche qui l'utilizzo del campo flag è analogo ma gli attori sono invertiti: il client dovrà aspettare che flag sia pari a 1 per iniziare a ricomporre il file (figura 2.17).

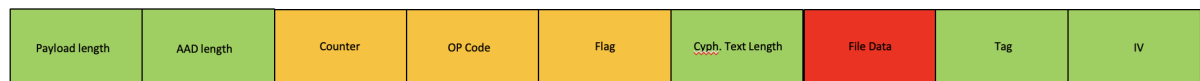


Figura 2.17: Messaggi di download dal Server verso il Client



Figura 2.18: Messaggio operazione conclusa dal Client verso il server

Una volta ricomposto il file, tramite il messaggio M4 (figura 2.18) il client comunica al server l'esito dell'operazione tramite l'uso del campo flag nuovamente: nel caso flag sia pari a 0, significa che il Download del file è fallito e il client non è riuscito a ricostruire il file; se invece il flag è pari a 1, il Download del file si è concluso con successo.

2.3.4 Rename

La Rename permette all'utente di modificare il nome di un file. Tramite il messaggio M1 (figura 2.20) il client comunica al server quale file sarà oggetto della procedura di rinomina.

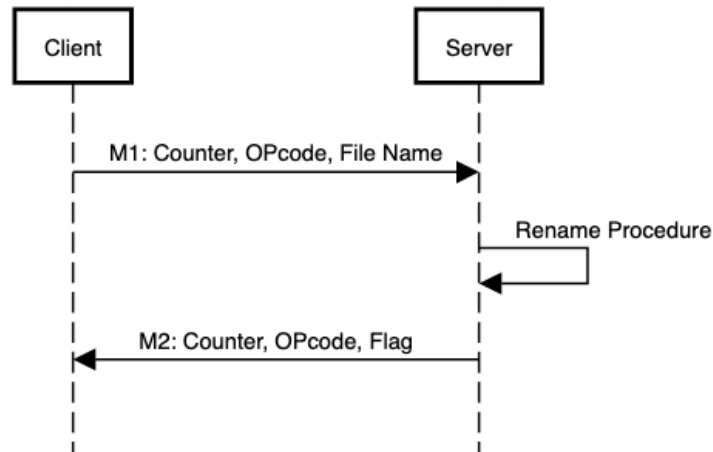


Figura 2.19: Schema funzione di Rename tra Client e Server



Figura 2.20: Messaggio di per rinominare un file dal Client verso il Server

Il server, ricevuta la richiesta di Rename, effettua controllo di blacklisting del nome del file ricevuto dal client. Operando in questo modo un attaccante non riuscirebbe ad effettuare attacchi di Directory Traversal. Controllato il File Name, il server effettua la procedura di Rename del file e risponde al client con il messaggio M2 (figura 2.21), dove il campo flag denota il successo dell'operazione (flag = 0 Rename fallito, flag = 1 Rename eseguito con successo).



Figura 2.21: Messaggio con l'esito della funzione Rename dal Server verso il Client

2.3.5 Delete

La funzione Delete consente ad un utente di eliminare un file presente sul sistema. Nella figura 2.22 è illustrato la sequenza dei messaggi scambiati.

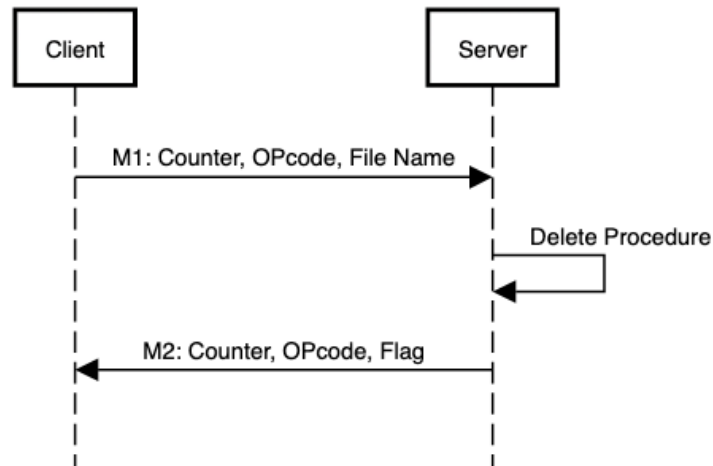


Figura 2.22: Schema funzione di Delete tra Client e Server

Come nella funzione Rename, il client tramite il messaggio M1 richiede che un determinato file venga eliminato dal server.



Figura 2.23: Messaggio di per eliminare un file dal Client verso il Server

Ricevuta la richiesta, il server effettua un controllo sul nome del file di whitelisting in modo che non siano possibili attacchi di Directory Traversal. Effettuati i controlli, esegue la procedura e comunica l'esito al client tramite il messaggio M2.

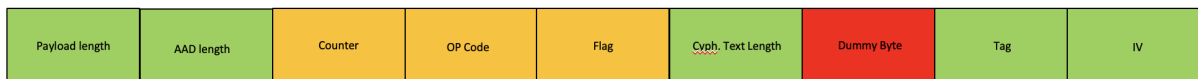


Figura 2.24: Messaggio con l'esito della funzione Delete dal Server verso il Client

2.4 Logout client

Finite le operazioni che l'utente desidera fare sul proprio storage, attraverso la procedura di Logout si può disconnettere dal server.

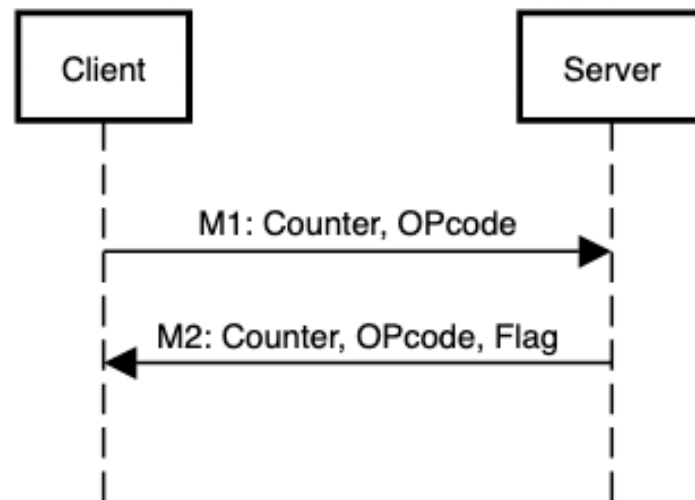


Figura 2.25: Diagramma funzione Logout tra Client e Server

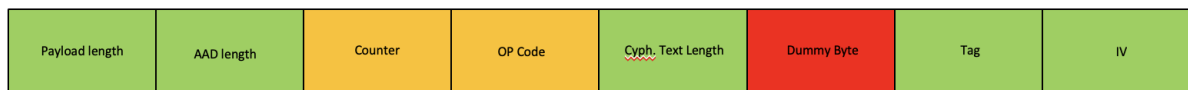


Figura 2.26: Messaggio di richiesta Logout



Figura 2.27: Messaggio contenente l'esito dell'operazione

Tramite il messaggio M1 (figura 2.26) il client si richiede di disconnettersi dal server. Ricevuta la richiesta dal client, il server notifica al client di aver ricevuto l'operazione richiesta (tramite il messaggio M2, in figura 2.27).

Successivamente, il server gestisce il logout "definitivo" del client, ovvero quello in cui il programma client termina, cancellando la chiave di sessione come previsto dallo schema della Perfect Forward Secrecy.

3 Implementazione

Dopo aver trattato la progettazione del Cloud Storage, prendiamo in considerazione le principali scelte implementative. In particolare, è stato implementato un server TCP in modo da avere comunicazioni più affidabili rispetto a UDP per un'applicazione di questo tipo.

3.1 Autenticazione

I principi seguiti per svolgere il login e l'autenticazione sono quelli adoperati per costruire il formato dei messaggi, spiegato nel paragrafo 2.1:

- Il server si autentica inviando il suo certificato. Il client leggerà il certificato ricevuto e, andando a controllare la sua validità (controllando che non sia stato revocato) potrà ricavare, se il certificato risulta valido, la chiave pubblica del server con la quale verificare la firma inviata nel messaggio in figura 2.4. La validità della firma implicherà l'autenticità del server.
- Il client si autentica inviando una quantità firmata al server il quale, avendo le chiavi pubbliche associate ad ogni username, può verificarne la validità.

3.2 Costruzione del messaggio

Ogni messaggio prima di essere inviato necessita dell'allocazione in memoria dei campi che il client e il server intendono scambiarsi all'interno del messaggio. Se l'allocazione delle porzioni di memoria richieste è andata a buon fine è possibile inizializzare tali variabili. Prima dell'invio del messaggio è importante serializzare i campi da inviare all'interno di un buffer, in cui vengono specificate, oltre ai valori, anche le dimensioni di vari campi, utili in fase di ricezione del messaggio. Infine viene usata la funzione *send* per l'invio del messaggio sulla socket.

3.2.1 Confidenzialità e Autenticazione dei pacchetti

I campi dei messaggi rispondono a diverse necessità in termine di sicurezza, come evidenziato in fase di progettazione. Pertanto, i campi che hanno bisogno di confidenzialità durante la trasmissione del messaggio sono cifrati con la chiave di sessione simmetrica, condivisa tra client e server in fase di autenticazione.

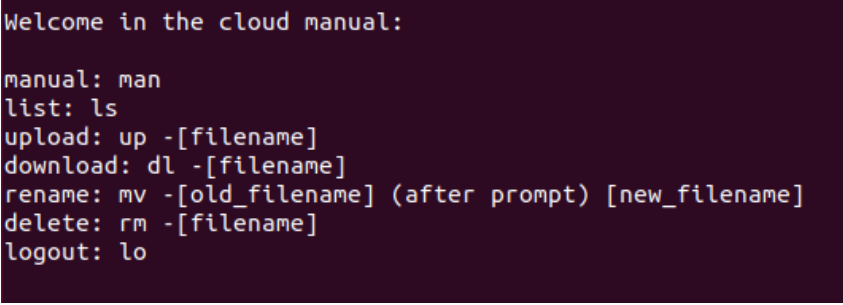
L'autenticazione dei campi viene effettuata mediante **AES 256 GCM** con la chiave simmetrica di sessione (*session key*).

3.3 Gestione delle funzionalità del Cloud Storage

Il Cloud Storage ha un OPCode specifico per ogni funzionalità, gestito dal campo autenticato "OPCode" nei messaggi scambiati.

Ogni funzionalità viene gestita all'interno di un costrutto *switch*, che regola il flusso delle operazioni tra client e server.

Attraverso il comando **man** l'utente invoca la funzione di help che elenca i comandi disponibili e la sintassi che i comandi devono avere effettuare correttamente le operazioni sul server:



```
Welcome in the cloud manual:

manual: man
list: ls
upload: up -[filename]
download: dl -[filename]
rename: mv -[old_filename] (after prompt) [new_filename]
delete: rm -[filename]
logout: lo
```

Figura 3.1: Screenshot del comando "man"

Come già anticipato, la gestione delle funzionalità viene gestita da due costrutti *switch*, uno per il client e uno per il server. Ogni funzionalità ha due parti comuni: la serializzazione del messaggio da inviare e la deserializzazione del messaggio ricevuto. Mentre le funzionalità sono implementate differentemente lato client e server per gestire la user experience e la messa a punto dell'operazione richiesta dal client.

3.3.1 Gestione upload e download dei file

Le funzioni di Upload e Download dei file operano in maniera simile. In particolare, entrambe le funzioni lavorano suddividendo il file interessato, dell'operazione di Upload o di Download, in "chunk" di dimensione pari a 1048576 Byte ($\sim 1\text{MB}$). Infatti, operando in questo modo si ottiene un compromesso tra velocità e quantità di informazioni scambiate tra cliente e server, andando ad ottimizzare le operazioni di Upload e di Download. Facendo un focus sul codice della funzione Upload:

```
1 if(file_size > CHUNK){
2     cout << "File greater than 1Mb - Proceeding to send chunks" << endl;
3     for(int i = 0; i < file_size - CHUNK && file_size > CHUNK; i += CHUNK){
4         memcpy(data_pt, &file_buffer[i], CHUNK);
5         [...]
6     }
```

Dal frammento di codice qui sopra possiamo vedere come viene fatta la suddivisione del file. Una volta suddiviso e inseriti i byte del chunk all'interno del file, il client (in questo caso, mentre il server nel caso di Download) invia il "pezzo di file". Questo meccanismo viene applicato fino al penultimo chunk. Infatti, l'ultimo chunk viene inviato in un messaggio con il campo flag settato a 1, in modo da segnalare al Server che il file è stato inviato completamente. Non appena il server avrà salvato correttamente il file nella directory del client, invierà al client una notifica, tramite il messaggio in figura 2.13, con l'esito dell'operazione di Upload.

In maniera speculare avviene l'operazione di Download come illustrato dalla figura 2.14.

3.3.2 Gestione Directory Traversal

Ogni operazione che prevede la gestione di file, quindi Upload, Download, Rename e Delete, ha una fase di controllo del nome del file in oggetto. Questa fase adopera una funzione di Blacklisting che controlla la presenza di alcuni caratteri speciali:

```
1 int blacklisting_cmd(string str) {
2     string check1 = "../";
3     string check2 = "..";
4     string check3 = "./";
5
6     if(str.find(check1) != string::npos || str.find(check2) != string::
7         npos || str.find(check3) != string::npos)
8         return -1;
9     return 0;
10 }
```

Operando in questo modo, si evita che il client possa accedere ad aree di memoria sul server non a lui dedicate, come lo spazio cloud di un altro utente.

È stato scelto di utilizzare la metodologia di Blacklist, invece che di Allowlist, per motivi di velocità di esecuzione e di controllo del filename. Infatti, scegliendo di implementare un controllo di whitelisting era necessario scegliere un dizionario composto da un set di caratteri definito (ad esempio l'alfabeto e i numeri da 0 a 9) e scansionare ogni singolo carattere del filename, controllando che il carattere sia tra quelli ammessi dalla funzione di whitelisting.

4 Gestione degli utenti

Siccome al server può accedere più di un utente, è stata implementata una struttura dati `_user` per fare in modo che il server riesca a gestire ogni connessione differente con il relativo utente. Di conseguenza, il server avrà la copia della chiave di sessione di ogni connessione, la copia del counter del client, e la socket di riferimento.

```
1 typedef struct _user {
2     int u_cl_socket;
3     int u_sv_socket;
4     unsigned int c_counter;
5     unsigned int s_counter;
6     char username[11];
7     unsigned char *session_key;
8
9     _user *next;
10 } user;
```

5 Gestione degli errori

Per fare in modo che gli errori siano gestiti correttamente è stato implementato un meccanismo che consente il fallimento sicuro dell'applicazione. In particolare, vengono utilizzati due buffer, uno per il client e uno per il server, che tengono traccia di tutti i puntatori allocati durante le operazioni effettuate dalle rispettive parti.

```
1 extern unsigned char *sv_free_buf[1024];
2 extern unsigned char *cl_free_buf[1024];
```

Non appena l'applicazione fallisce, viene chiamata la funzione *void free_var(int side)* che provvede a liberare la memoria occupata scorrendo il buffer che è stato selezionato e liberando un puntatore alla volta:

```
1 void free_var(int side){
2     int counter = 0;
3     if(side == 1){
4         counter = cl_index_free_buf;
5         for(int i = 0; i < counter - 1; i++){
6             if(cl_free_buf[i]){
7                 free((void*)cl_free_buf[i]);
8                 cl_free_buf[i] = NULL;
9             }
10        }
11        cl_index_free_buf = 0;
12    }
13    else if(side == 0){
14        counter = sv_index_free_buf;
15        for(int i = 0; i < counter - 1; i++){
16            if(sv_free_buf[i]){
17                free((void*)sv_free_buf[i]);
18                sv_free_buf[i] = NULL;
19            }
20        }
21        sv_index_free_buf = 0;
22    }
23    else{
24        cerr << "Panic! Critical error, shutting down program..." << endl;
25        exit(0);
26    }
27 }
```