Relazione Progetto Applied Cryptography

Cortis Salvatore, Del Castello Diego

Introduzione

Nel seguente report, viene riportata la documentazione per un progetto universitario di Applied Cryptography del corso di laurea magistrale in Cybersecurity dell'Università degli Studi di Pisa. Nel dettaglio sono riportati in primo luogo la presentazione dello scenario, 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. Secondariamente sono riportate le scelte implementative e le linee guida per l'uso.

1 Presentazione dello scenario

Lo scenario è quello di un'applicazione Client-Server che possa fungere da Cloud Storage. Si pensi che ad una situazione con diversi utenti, ognuno con il suo spazio riservato, al quale gli altri utenti non possono accedere. Si vuole che ogni utente possa eseguire i comandi: Update, Download, Delete, List, Rename, LogOut, meglio analizzati nella sessione 3. È richiesto che venga prima negoziata una chiave di sessione che rispetti i canoni della perfect forward secrecy e che vengano sempre garantite autenticazione, integrità, riservatezza e protezione dagli attacchi di tipo replay.

2 Scelte progettuali

2.1 Diagramma di sequenza

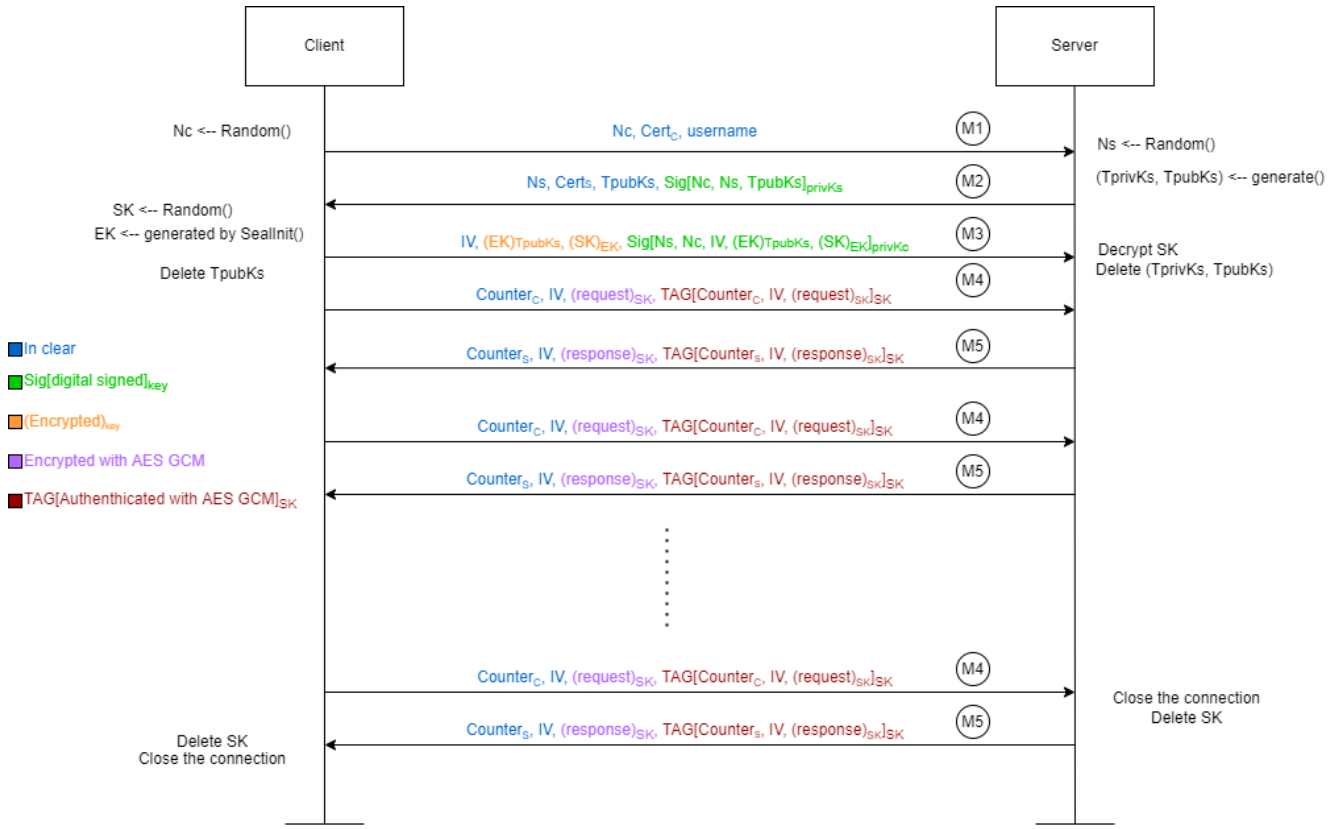

Figure 1: Diagramma di sequenza

2.2 Autenticazione client-server

In questa sezione vedremo le fasi di autenticazione e scambio della chiave che avvengono tra client e server, prima iniziare lo scambio comando del client - risposta del server. L'autenticazione tra client e server insieme con lo scambio della chiave avvengono tramite lo schema **RSA effimero**.

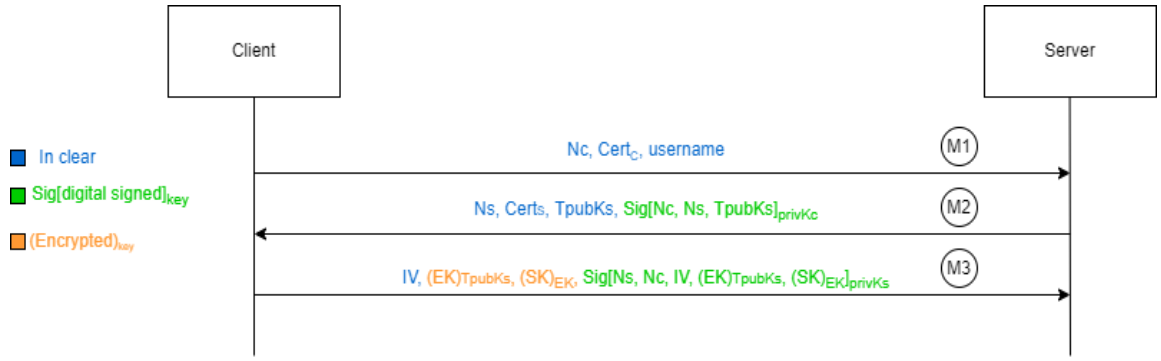

Figure 2: Schema dei messaggi tra client e server per la reciproca autenticazione

Nel messaggio M1 il client invia al server un messaggio contenente una quantità fresh, detta *nonce*, che servirà al server nella sua fase di autenticazione per dimostrare la freschezza del messaggio M2. Invia inoltre il proprio certificato in modo da richiedere la fase di autenticazione e il suo *username*, che sarà poi utile durante l'esecuzione dei comandi. Abbiamo scelto di far inviare il certificato anche al client perché dalle specifiche risulta che il server possiede la chiave pubblica di ogni utente, ma non il suo certificato. Di conseguenza per effettuare la verifica della validità di tale chiave è necessario inviare dapprima l'intero certificato.

Il server infatti, ricevuto il messaggio M1, come prima cosa verifica se la chiave estratta dal certificato ricevuto è la stessa della copia in suo possesso. Se ciò viene confermato, passa poi a controllarne la validità interrogando la CRL e solo se tutti i controlli vengono superati passa a memorizzare il *nonce* ricevuto dal client e il suo username.

Dimensione del *nonce*	*Nonce* del client	Dimensione del certificato del client	Certificato del client	Dimensione dell'username	Username

Figure 3: Formato del messaggio M1

Il server a questo punto genera una coppia di chiavi, pubblica e 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 successivo, la chiave di sessione per parlare con il server. Nel messaggio M2 il server invia in chiaro il suo *nonce*, il suo *certificato* e la *chiave pubblica effimera*. Il tutto è concatenato alla firma del *nonce* del client che aveva estratto dal messaggio M1, del proprio *nonce* e della chiave effimera, in modo che se ne possa garantire l'integrità e l'autenticità. Il certificato servirà in più a permettere al client di autenticare il server. Il *nonce* del client viene inserito solo nella firma perché già

a disposizione di esso.
Il client dunque, una volta ricevuto il messaggio M2, estrae la chiave pubblica dal certificato del server e la usa per verificare la firma digitale. Un successo significa non solo autenticità del messaggio, ma anche freschezza dovuta alla correttezza del *nonce* inserito nella firma. Si procede quindi con la verifica della validità del certificato. Superato anche questo controllo procede dunque a memorizzare il *nonce* del server e la chiave effimera pubblica. Utilizzare la coppia di chiavi effimere garantisce il **Perfect Forward Secrecy**, ovvero la protezione delle sessioni passate in caso di compromissione della chiave privata *"long term"* del server.

Dimensione del *nonce* del server	*Nonce* del server	Dimensione del certificato del server	Certificato del server	Dimensione della chiave effimera pubblica	Chiave effimera pubblica	Dimensione della firma	Firma digitale

Figure 4: Formato del messaggio M2

A questo punto il client deve generare la *chiave di sessione* per inviarla al server. Una volta generata, per garantire la sua confidenzialità, la cifrerà con la chiave pubblica effimera appena ricevuta dal server in modo che solo quest'ultimo possa decifrarla con la corrispondente chiave privata effimera. Il messaggio M3 viene quindi preparato con la chiave di sessione cifrata con la chiave pubblica effimera. A questa si aggiunge la sua firma digitale, preceduta da quella dei due *nonce* precedentemente scambiati. Il messaggio conterrà inoltre alcuni parametri utili al server per decifrare la chiave di sessione con il meccanismo della **Digital Envelope**.

SK è la Session Key che verrà utilizzata durante la sessione. Prima di essere inserita nel messaggio M3 viene cifrata con EK. EK è l'Encrypted Key, ossia la chiave cifrata dalle primitive del Digital Envelope, utilizzando TpubKs, che è la chiave effimera pubblica generata dal server.

IV	Dimensione della chiave del Dig. Env.	Chiave del Dig. Env.	Dimensione della chiave simmetrica	Chiave Simmetrica	Dimensione della firma	Firma digitale

Figure 5: Formato del messaggio M3

Per garantire la proprietà Perfect Forward Secrecy, le chiavi effimere vengono eliminate subito dopo la decifratura della chiave di sessione, mentre la chiave di sessione verrà eliminata quando il client farà logout dal server.

2.3 Autenticità e confidenzialità della sessione

Dopo il messaggio M3, il client e il server possono comunicare cifrando ogni messaggio tramite la chiave di sessione appena scambiata. I messaggi che seguono lo scambio della chiave di sessione sono tutti cifrati con **AES GCM 128** che permette a due utenti di inviarsi 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*. L'utilizzo di un metodo di *Authenticated Encryption* ci permette di avere lo scambio di un'unica chiave di sessione che verrà utilizzata sia per cifrare che per autenticare i messaggi. È dunque importante che essa non venga in alcun modo compromessa perché altrimenti tutto il sistema risulterebbe compromesso a sua volta.

Inoltre non è possibile separare le due operazioni. Ogni messaggio viene cifrato e autenticato, anche laddove non fosse strettamente necessaria l'autenticazione.

2.4 Messaggi di sessione

Secondo il nostro protocollo, questo è lo schema dei messaggi che seguono M3:

Figure 6: Schema dei messaggi scambiati tra client e server dopo l'autenticazione per inviare comandi e ricevere risposte.

Una volta stabilita la sessione, il client può procedere all'invio dei comandi. I messaggi presentano tutti la stessa struttura composta dai parametri tipici di AES GCM: *AAD*, *messaggio cifrato*, *tag*. All'interno dell'AAD, come parametri che non necessitano confidenzialità troviamo un contatore e l'IV. La chiave utilizzata per cifrare il messaggio e per calcolare il TAG è la chiave di sessione SK che era stata scambiata nel corso del messaggio M3. I possibili valori assumibili dal messaggio cifrato vengono analizzati nel dettaglio nella sessione 3, ma in via generica corrispondono sempre ad una *request* inviata da parte del client o ad una *response* del server. Altra differenza tra i messaggi M4 e M5 sta nel tipo di contatore che viene inviato: rispettivamente $Counter_{client}$ nel caso di messaggi da client a server, $Counter_{server}$ nel caso di messaggi da server a client.

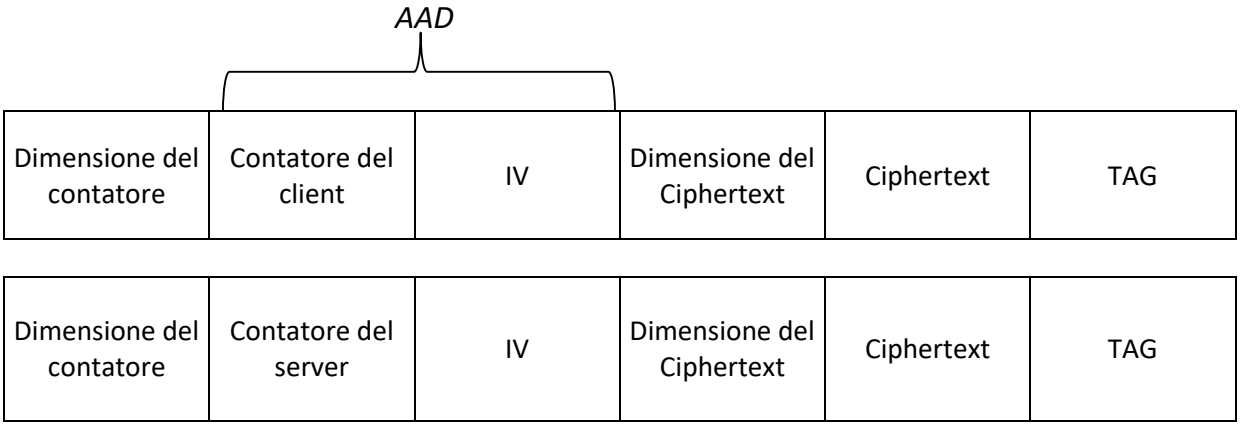

Figure 7: Formato dei messaggi M4 e M5

I comandi tuttavia seguono procedure di scambio di messaggi preimpostate e approfondite nella sessione 3.

Nonce e contatori client-server

Per proteggersi dai replay attack, ogni messaggio deve contenere una quantità fresh detta *nonce*, che provi a chi legge ogni messaggio, che esso sia recente e non frutto di una replica.

Nei primi tre messaggi, la soluzione è quella di utilizzare gli stessi *nonce* utilizzati nella fase di autenticazione. Poiché bisogna ancora stabilire una chiave di sessione, essendo inviati in chiaro, per garantire l'integrità e l'autenticità dei *nonce*, essi vengono anche firmati digitalmente. Una volta stabilita la chiave di sessione invece, la nostra scelta è stata quella di usare un contatore per il client ed uno per il server. Ad ogni sessione i contatori di client e server vengono inizializzati a 0. Ogni volta che il client deve inviare un messaggio manda come primo campo il proprio *counter* incrementato di 1. Dal lato ricezione, il server, ad ogni messaggio che viene ricevuto, controlla che il primo campo corrisponda esattamente al *counter* precedentemente memorizzato, incrementato di 1. Se il controllo fallisce si presuppone che sia in atto un replay attack, o un generico problema di integrità, il messaggio viene scartato. In caso contrario il contatore viene aggiornato e si prosegue all'analisi del resto del messaggio. Il tutto succede in maniera complementare anche quando le parti sono invertite. Utilizzando AES GCM 128 durante la sessione, i contatori fanno parte, insieme all'IV, degli AAD, e la loro integrità ed autenticità è garantita dal TAG.

3 Comandi

I possibili comandi sono: *Upload, Download, Delete, List, Rename* e *Logout.* Di volta in volta il client è chiamato ad inserire uno di questi. Nessun messaggio viene inviato al server fino all'inserimento corretto di un comando.

3.1 Upload

Upload è il comando tramite il quale un utente può caricare un file nel suo spazio dedicato nel cloud storage. Una volta ricevuto il comando, il server si pone in attesa di ricevere anche il nome del file che si vuole caricare, e la sua dimensione. Quando il client invia il *filename*, il server esegue un controllo sul formato del nome per evitare possibili *code injection.* Se il nome del file non corrisponde a nessun file presente nella cartella di esecuzione del client viene inviato un messaggio di errore al server e l'operazione viene interrotta. Se la dimensione del file supera quella massima di 4GB (4.294.967.296 byte), è il server ad inviare un messaggio di errore al client. In assenza di errori invece, inizia un ciclo di messaggi con una dimensione predefinita del plaintext di 32KB (32768 byte), in cui viene inviato tutto il contenuto del file dal client al server e ogni volta il

server notifica se c'è stato un errore per farsi rinviare il messaggio.

Se il file che si vuole caricare è già presente nel cloud storage, il server invia al client un messaggio in cui offre la possibilità di rinominare il file già presente nello storage. In caso negativo l'upload fallisce. In caso affermativo invece, l'operazione di *upload* viene sospesa e si passa a seguire le procedure che intercorrono durante il comando *Rename* analizzato nella sezione 3.5. In assenza di errori viene poi conclusa l'upload.

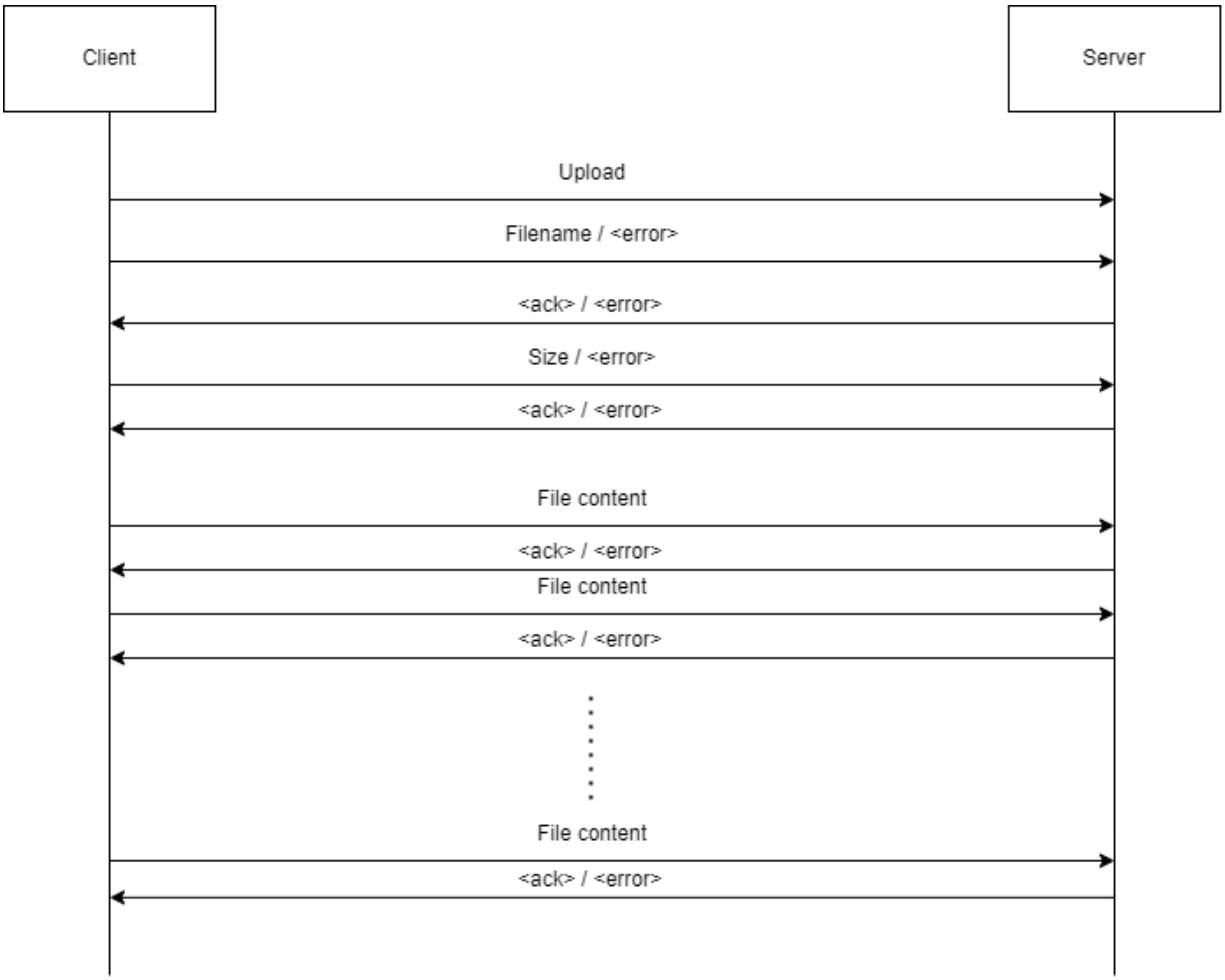

Figure 8: Scambio di messaggi per il comando *Upload.* (Parametri *request* e *response* dei messaggi M4 e M5)

3.2 Download

Download è il comando tramite il quale un utente può scaricare un file dal suo spazio dedicato nel cloud storage. Una volta ricevuto il comando, il server si pone in attesa di ricevere anche il nome del file che si desidera scaricare. Quando il client invia il *filename*, il server esegue un controllo sul formato del nome per evitare possibili *code injection.* Se il nome del file corrisponde ad un file già presente all'interno della cartella di esecuzione del client viene inviato un messaggio di errore al server e l'operazione viene interrotta. Se il file che si vuole scaricare non è presente nel cloud storage invece, è il server ad inviare un messaggio di errore al client. In assenza di errori invece, inizia un ciclo di messaggi con una dimensione predefinita del plaintext di 32KB (32768 byte), in cui viene inviato tutto il contenuto del file dal server al client e ogni volta il client notifica se c'è stato un errore per farsi rinviare il messaggio.

Se il file che si vuole scaricare è già presente nella cartella di esecuzione del client, il file viene sovrascritto.

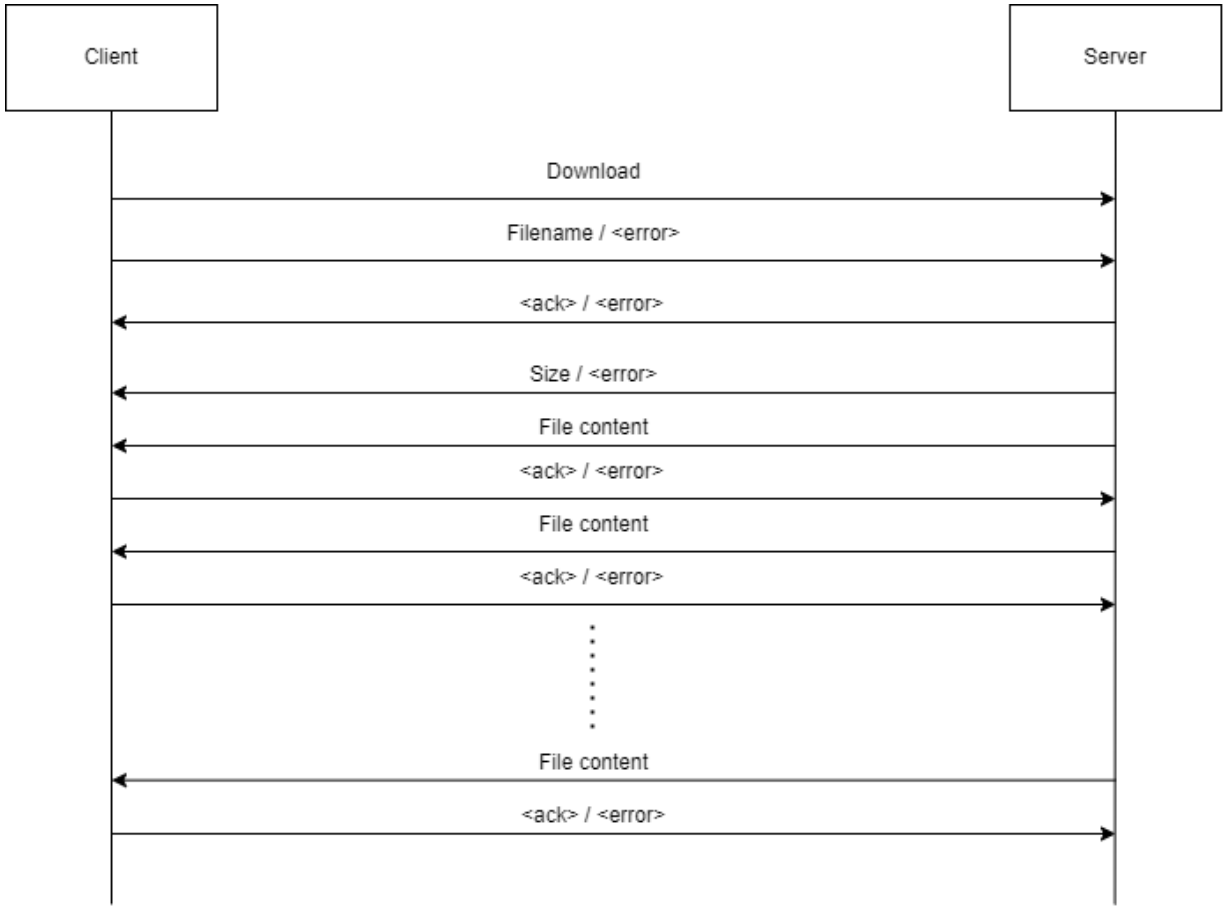

Figure 9: Scambio di messaggi per il comando *Download.* (Parametri *request* e *response* dei messaggi M4 e M5)

3.3 Delete

Delete è il comando tramite il quale un utente può eliminare un file dal suo spazio dedicato nel cloud storage. Una volta ricevuto il comando, il server si pone in attesa di ricevere anche il nome del file che si desidera eliminare. Quando il client invia il *filename*, il server esegue un controllo sul formato del nome per evitare possibili *code injection.* Se il nome del file non corrisponde a nessun file presente nel cloud storage dedicato all'utente, il server invia un messaggio di errore al client e l'operazione viene interrotta. In assenza di errori invece, il server chiede al client conferma dell'operazione e in caso affermativo elimina il file.

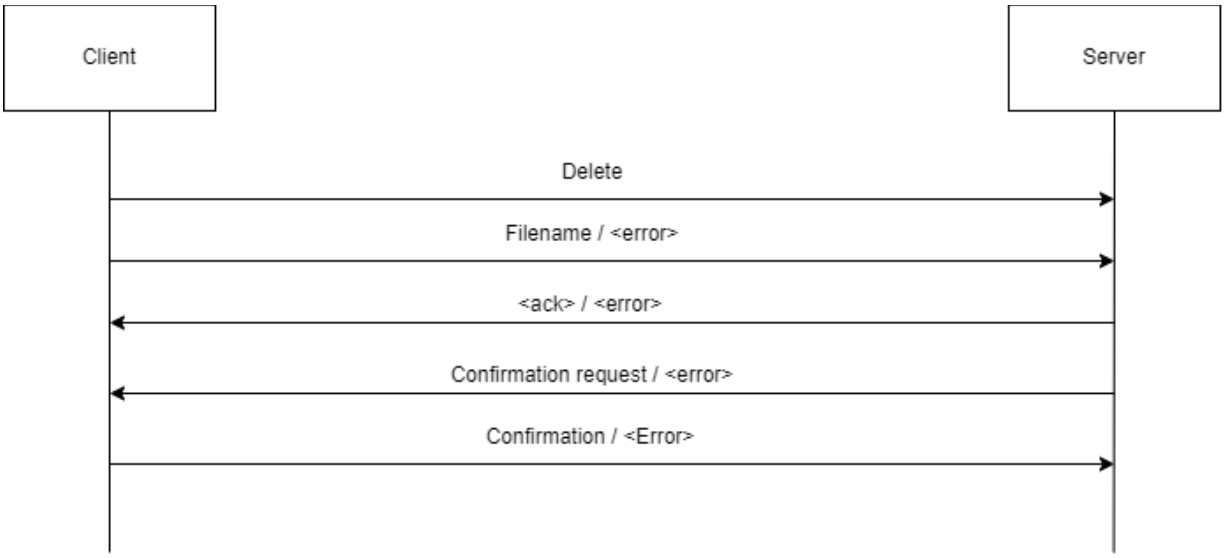

Figure 10: Scambio di messaggi per il comando *Delete.* (Parametri *request* e *response* dei messaggi M4 e M5)

3.4 List

List è il comando tramite il quale un utente può visualizzare l'elenco dei file presenti all'interno del suo spazio dedicato nel cloud storage. Una volta ricevuto il comando, il server invia l'elenco dei file.

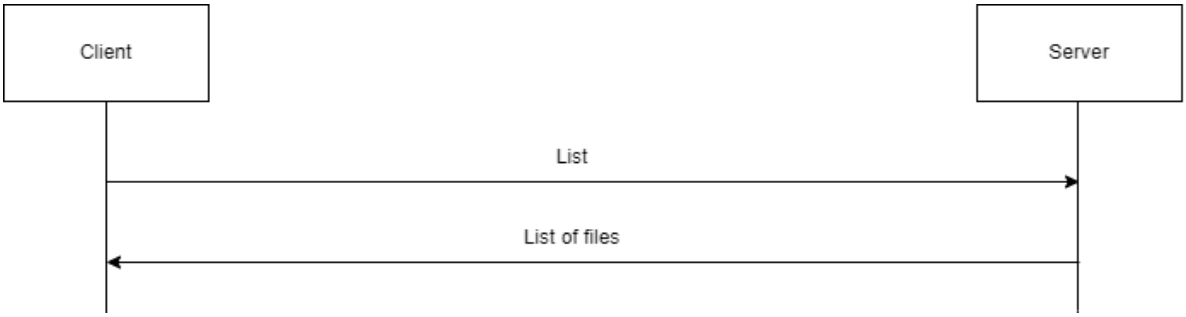

Figure 11: Scambio di messaggi per il comando *List.* (Parametri *request* e *response* dei messaggi M4 e M5)

3.5 Rename

Rename è il comando tramite il quale un utente può rinominare un file presente all'interno del suo spazio dedicato nel cloud storage. Una volta ricevuto il comando, il server si pone in attesa di ricevere anche il nome del file che si desidera rinominare. Quando il client invia il *filename*, il server esegue un controllo sul formato del nome per evitare possibili *code injection.* Se il nome del file non corrisponde a nessun file presente nel cloud storage dedicato all'utente, il server invia un messaggio di errore al client e l'operazione viene interrotta. In assenza di errori invece, il server chiede al client di inviare anche il nuovo nome che si vuole dare al file. Anche in questo caso viene eseguito un controllo sul nome per evitare *code injection* e se è già presente un file nel cloud storage con lo stesso nome, viene inviato un messaggio d'errore. In assenza di errori invece, il server procede nell'esecuzione dell'operazione.

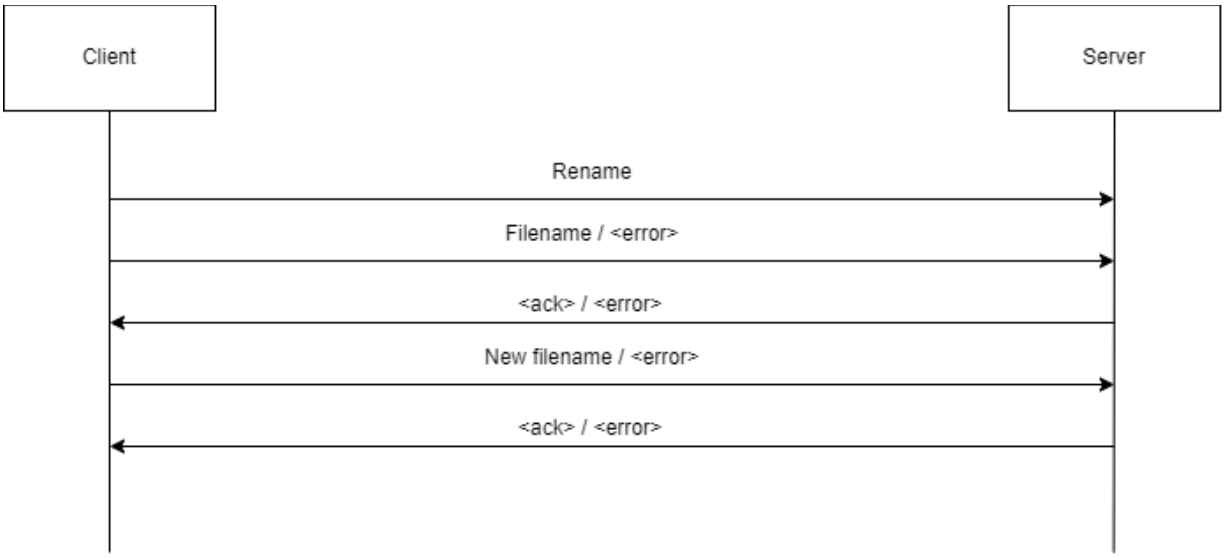

Figure 12: Scambio di messaggi per il comando *Rename.* (Parametri *request* e *response* dei messaggi M4 e M5)

3.6 LogOut

LogOut è infine il comando tramite il quale un utente chiude la sessione. Una volta ricevuto il comando, il server invia al client un messaggio di avvenuta operazione, elimina la chiave di sessione ed esce. Il client riceve il messaggio e a sua volta, dopo aver eliminato la chiave di sessione, esce.

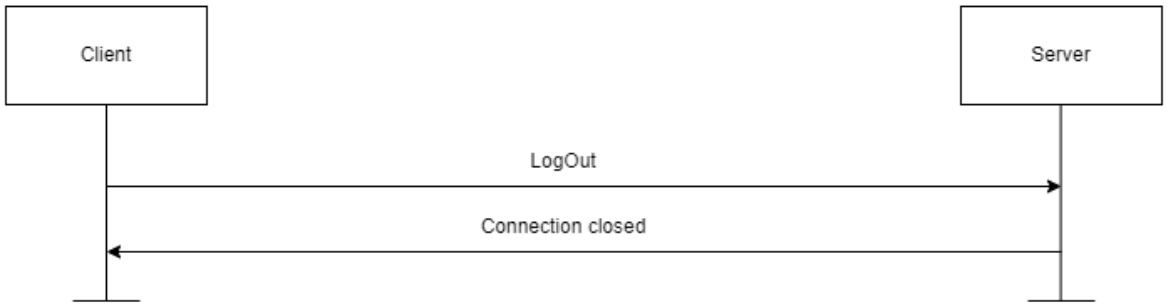

Figure 13: Scambio di messaggi per il comando *LogOut.* (Parametri *request* e *response* dei messaggi M4 e M5)

3.7 Help

Abbiamo anche implementato un comando *Help* tramite il quale un utente visualizza nuovamente i comandi. Tale comando avviene solo lato client. Nello specifico, se l'utente nel momento in cui digita un comando ed

esso sia per l'appunto "Help", il client stampa tutti i comandi disponibili senza inviare nulla al server.

4 Implementazione della connessione

Abbiamo implementato un server TCP in modo da avere comunicazioni più affidabili rispetto a UDP per un'applicazione di questo tipo.

Il server ha un'architettura multi processo, ciò significa che viene creato un processo per ogni connessione ricevuta da un client. Il processo gestirà fino al momento in cui desidera fare logout. Una volta avviato il programma server, si entra in un ciclo infinito nel quale si attendono connessioni: quando ne arriva una, viene lanciato il processo che gestirà con il client le comunicazioni tramite la socket con cui si è connesso.

Lo scambio dei messaggi avviene attraverso la socket tra client e server per tutte le fasi del programma in quanto ogni processo si occupa di un client.
I messaggi vengono scritti e letti dalla socket tramite due funzioni *send* e *read*. In generale, la lettura del buffer dalla socket avviene grazie al formato fisso dei messaggi ad ogni fase. Ogni elemento inviato nel messaggio (es. chiavi, ciphertext...) viene preceduto dalla sua lunghezza in byte che, una volta convertita in intero, permette di fare la lettura dell'elemento in questione dalla socket specificando il numero di byte da leggere corretto.