Chatty

Federico Bernacca, Kostantino Prifti

May 2021

Contents

1 Introduzione **2**

2 Struttura del progetto **2**

3 Struttura del server **3**
3.1 Architettura . . . 3
3.2 Servizi . . . 3

4 Struttura del client **3**
4.1 Architettura . . . 3

5 Protocollo **4**
5.1 Messaggi . . . 4
5.2 Quantità . . . 4
5.3 Cifrari . . . 4
5.4 Replay attack . . . 4
5.5 Operazioni . . . 4
5.6 Negoziazione chiave simmetrica client-server . . . 5
5.7 Sessione client-server . . . 6
5.7.1 Formato dei messaggi . . . 6
5.8 Negoziazione chiave simmetrica client-client . . . 7
5.8.1 Formato dei messaggi . . . 7
5.9 Chat client-client . . . 8
5.9.1 Formato dei messaggi . . . 8

1 Introduzione

Durante la stesura del seguente report, abbiamo deciso di seguire un approccio top down per la descrizione delle scelte implementative, partendo da una descrizione (informale) ad alto livello delle per poi scendere più nello specifico, dettagliando accuratamente. In particolare, seguiremo la seguente scaletta:

- **Struttura fisica del progetto**.
- **Struttura del server**.
- **Struttura del client**.
- **Struttura del protocollo**.

2 Struttura del progetto

Durante il corso, abbiamo implementato diverse classi, una per ogni argomento visto a lezione, sfruttando le funzioni di openSSL, cercando di costruire una sorta di libreria ad alto livello, così da arrivare ad implementare il progetto con una base solida di partenza.
Tutti i files si trovano nella directory **chatty**, che ha la seguente struttura.

- **chatty**
  - **server**
    - **server.cc** (implementazione del server)
    - **Server_cert.pem** (certificato del server generato da *Simple Authority*)
    - **Server_key.pem** (chiave privata del server)
    - **bob.pem** (chiave pubblica dell'utente *bob*)
    - **fede.pem** (chiave pubblica dell'utente *fede*)
    - **kosta.pem** (chiave pubblica dell'utente *kosta*)
  - **client**
    - **client.cc** (implementazione del client)
    - **bob_private_key.pem**, **bob_public_key.pem** (chiave privata/pubblica dell'utente *bob*)
    - **fede_private_key.pem**, **fede_public_key.pem** (chiave privata/pubblica dell'utente *fede*)
    - **kosta_private_key.pem**, **kosta_public_key.pem** (chiave privata/pubblica dell'utente *kosta*)
    - **CA_cert.pem**, **CA_crl.pem** certificato della CA (Simple Authority) / certificate revocation list
  - **asymmetric_encryption**
    - **AsymmetricEncryption.h** (dichiarazione classe per la cifratura a chiave pubblica)
    - **AsymmetricEncryption.cc** (implementazione)
  - **symmetric_encryption**
    - **SymmetricEncryption.h** (dichiarazione classe per la cifratura simmetrica)
    - **SymmetricEncryption.cc** (implementazione)
  - **ca**
    - **Certificate.h** (dichiarazione classe per la verifica dei certificati)
    - **Certificate.cc** (implementazione)
  - **digital_signature**
    - **DigitalSignature.h** (dichiarazione classe per firmare/verificare una firma)
    - **DigitalSignature.cc** (implementazione
  - **user**
    - **User.h** (dichiarazione classe rappresentante un utente online)
    - **User.cc** (implementazione)
  - **Message**
    - **Message.h** (dichiarazione classe rappresentante un messaggio di tipo generico)
    - **Message.cc** (implementazione)
  - **utils**
    - **utils.h** (dichiarazione helper class per funzioni generiche)
    - **utils.cc** (implementazione)
  - **ssl_utils**
    - **ssl_utils.h** (dichiarazione helper class per funzioni openSSL)
    - **ssl_utils.cc** (implementazione)

3 Struttura del server

Abbiamo optato per implementare un server TCP, ritenendo questa scelta più affidabile di un server UDP, essendo quest'ultimo connectionless, non gestisce la ritrasmissione dei pacchetti persi, quindi lo abbiamo ritenuto meno affidabile e meno opportuno per questo tipo di applicazione. Inoltre abbiamo scartato anche l'opzione server AF_UNIX perché le comunicazioni sarebbero state possibili solo tra processi appartenenti alla stessa macchina.

3.1 Architettura

Il server ha un'architettura multithread, viene creato un thread per ogni connessione ricevuta da un client. Questo thread gestisce il client fino alla sua disconnessione.
Una volta avviato il programma server, si entra in un ciclo infinito nel quale si attendono connessioni; quando ne arriva una, viene creato il suddetto thread.

3.2 Servizi

Il server offre due diverse connessioni TCP; si avranno dunque due socket per ogni client connesso.
La prima, chiamata *main_socket* è quella principale con cui comunicano client e server; la seconda, chiamata *request_socket*, serve solo ad inoltrare, da parte del thread gestore di un client X, una richiesta di chat ad un altro client Y.
Questa richiesta viene ricevuta da un apposito thread, lato client, in ascolto su *chat_socket*, cosicché questa richiesta non si sovrapponga all'input del client Y. Essa viene accodata in un vettore di richieste di chat.
In questo modo, il client Y può interrogare questo vettore quando vuole, così da non avere una sovrapposizione nell'input nel caso in cui dovessero arrivargli una o più richieste di chat.

4 Struttura del client

Una volta avviato il programma client, esso si connette ai due servizi attivi messi a disposizione dal server.

4.1 Architettura

Il client, come il server, presenta un'architettura multithread; viene creato un primo thread dopo la fase di autenticazione dell'utente. Questo thread rimane attivo fino alla disconnessione o fino all'ingresso in chat; è in ascolto sulla socket *chat_socket*, descritta nella sezione **3.2**, sulla quale vengono inviate le richieste di chat da parte degli altri utenti.

Quando arriva una richiesta, questa non viene immediatamente mostrata all'utente ma viene accodata in un vettore, interrogabile in qualsiasi momento.

Negoziata la chiave simmetrica tra i due client, prima di iniziare la chat, viene creato un thread **reader** che entra in un loop e rimane in attesa di ricevere i messaggi dell'altro peer, inoltrati dal server. Quando li riceve, vengono decifrati, verificati e stampati a video.

Il thread principale invece funziona da **writer**; quindi attende l'input del peer, lo cifra, e lo invia al server, che lo inoltrerà a sua volta al corrispettivo utente.

5 Protocollo

Di seguito illustriamo le scelte implementative che realizzano il nostro protocollo.

5.1 Messaggi

Ogni messaggio inviato sulle socket è così strutturato:

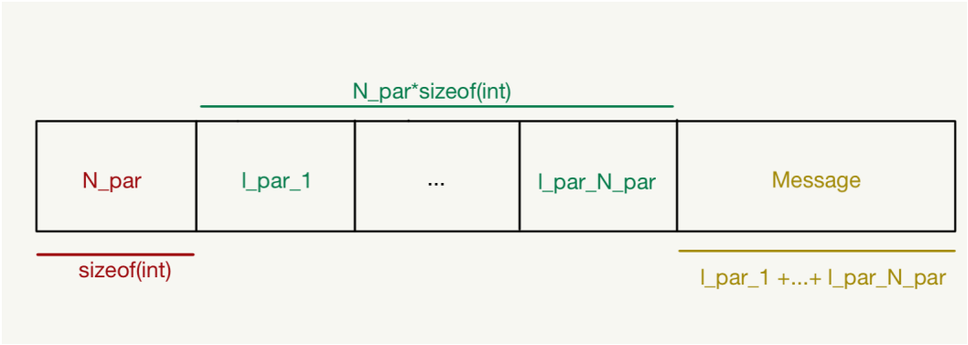

Figure 1: Struttura di un messaggio

dove:

- **N_par** indica il numero di diversi messaggi concatenati nel **Message**.
- **I_par**$_i$ indica la lunghezza del messaggio in posizione i.
- **Message** rappresenta tutto il payload del messaggio.

5.2 Quantità

Abbiamo utilizzato chiavi **RSA** da **2048 bit**; la dimensione della firma digitale risulta essere di 256 bytes: **SIGN_SIZE 256**.
Abbiamo fissato una dimensione massima per ogni messaggio scambiato in chat: **MAX_SIZE 10000**.
Infine, abbiamo definito un'ulteriore quantità fissata: **CHUNK 256**, utilizzata per la generazione dei vari nonce.

5.3 Cifrari

Abbiamo utilizzato la crittografia a chiave pubblica per la negoziazione della chiave simmetrica di sessione, col cifrario **EVP_aes_256_cbc**.
Abbiamo invece utilizzato la crittografia simmetrica nelle sessioni, utilizzando **Authenticated encryption with associated data (AEAD)** col cifrario **EVP_aes_256_gcm** per soddisfare i requisiti di segretezza e autenticità.

5.4 Replay attack

Per evitare i replay attack in sessione, abbiamo utilizzato 4 diversi contatori.

```
unsigned int server_counter_send;
unsigned int server_counter_recv;
unsigned int peer_counter_send;
unsigned int peer_counter_recv;
```

I primi due sono utilizzati per la sessione tra client e server, dopo la generazione della chiave simmetrica di sessione.
Gli altri due sono utilizzati per la sessione tra due utenti, dopo la generazione della chiave simmetrica di sessione, quando iniziano la chat.

In particolare, ogni volta che viene inviato un messaggio da parte di X, viene incrementato $X\_counter\_send$; quando raggiunge la destinazione Y, questa controlla che il suo contatore di ricezione sia:

$$Y\_counter\_recv == X\_counter\_send - 1$$

5.5 Operazioni

Di seguito elenchiamo le operazioni che vengono fatte tra client e server quando quest'ultimo riceve una connessione da parte di un client, illustrando il formato dei messaggi per ogni operazione. Scriveremo poi una sezione più approfondita per ogni operazione.

1. **Negoziazione chiave simmetrica client-server**: client e server concordano una chiave simmetrica da usare per la loro sessione, autenticandosi reciprocamente.
2. **Sessione client-server**: scambio di messaggi cifrati tra client e server.

3. **Negoziazione chiave simmetrica client-client**: prima di iniziare una conversazione, i due client stabiliscono una chiave simmetrica da utilizzare in sessione.

4. **Chat client-client** scambio di messaggi cifrati tra due client.

5.6 Negoziazione chiave simmetrica client-server

Client e server stabiliscono una chiave simmetrica da utilizzare in sessione, autenticandosi reciprocamente. Abbiamo deciso di utilizzare lo schema **Ephemeral RSA (RSAE)** per questa negoziazione e per garantire il *perfect forward secrecy*.

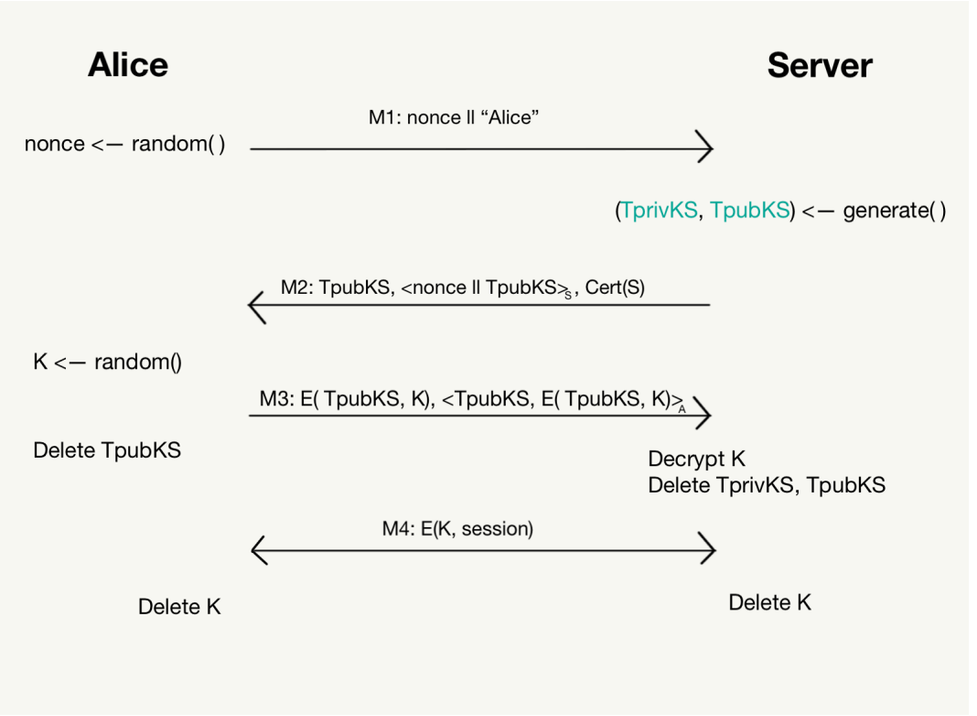

Figure 2: Ephemeral RSA (RSAE)

Il protocollo è stato così realizzato:

1. Il client genera un nonce random di dimensione fissa **CHUNK**, e salva il suo username.
2. Il client invia al server il seguente messaggio:

   `[nonce, username]`

3. Il server riceve il messaggio e lo divide.
4. Il server genera una coppia di chiavi RSA (privata/pubblica) temporanee.
5. Il server invia al client il seguente messaggio:

   `[<[R, temp_pub_Key]>signed, temp_pub_key, certificate]`

   firmando con la chiave privata a lungo termine.
6. Il client riceve il messaggio, lo divide, ed estrae il certificato.
7. Il client Estrae il certificato dal messaggio ricevuto.
8. Il client verifica che il certificato sia valido.
9. Il client estrae la chiave pubblica dal certificato.
10. Il client estrae la chiave pubblica temporanea e la firma dal messaggio ricevuto.
11. Il client verifica la firma utilizzando la chiave pubblica estratta.
12. Il client genera la chiave simmetrica di sessione da utilizzare col server.
13. Il client cifra, utilizzando la tecnica della *Digital Envelope*, la chiave di sessione usando la chiave pubblica effimera inviata dal server al punto 6.
14. Il client firma la seguente quantità:

    `<[E(k), TpubKey]>`

15. Il client invia invia al server il seguente messaggio:

    `[E(k), iv, iv <[E(k), TpubKey]>, ct (session key)]`

    dove *E(k)*, *iv*, *ct* sono generati dalla Digital Envelope.

16. Il server riceve il messaggio, lo divide, e verifica la firma.

17. Il server decifra la chiave simmetrica di sessione, e la salva.

18. Client e server eliminano le chiavi effimere.

19. Client e server utilizzano questa chiave di sessione per comunicare.

5.7 Sessione client-server

Negoziata la chiave simmetrica di sessione al punto precedente, si entra in un ciclo in cui client e sever scambiano messaggi cifrati con questa chiave simmetrica. Come prima operazione, il server inoltra al client gli utenti online.
Il client ha a disposizione le seguenti operazioni:

- **"reload"**: per chiedere al server di inviare la lista aggiornata di utenti online.
- **"username"**: per chiedere al server di inoltrare la richiesta di chat all'utente *username*; il client viene messo in attesa di risposta.
- **"requests"**: per interrogare le richieste di chat arrivate al thread nel client che si occupa di riceverle sull'apposita socket e di accodarle in un vettore; a questo punto può accettare o rifiutare una richiesta di chat.

5.7.1 Formato dei messaggi

Ogni messaggio scambiato tra client e server durante la sessione è di tipo *AEAD* ed ha la seguente forma:

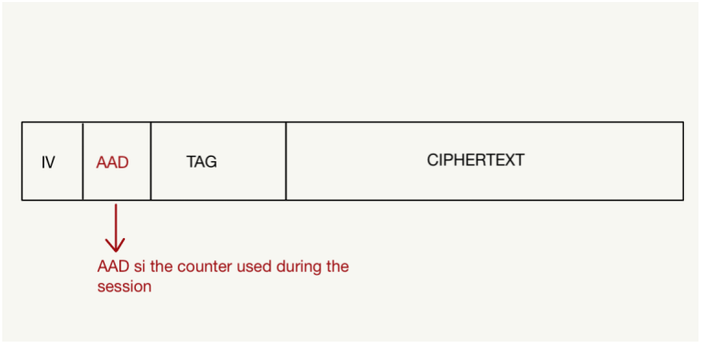

Figure 3: AEAD message

5.8 Negoziazione chiave simmetrica client-client

Quando due peer sono d'accordo sull'iniziare una chat, il server inoltra ad essi le rispettive chiavi pubbliche; così inizia la negoziazione della chiave simmetrica di sessione tra i due peer.
La negoziazione segue lo schema **RSAE** in questo modo:

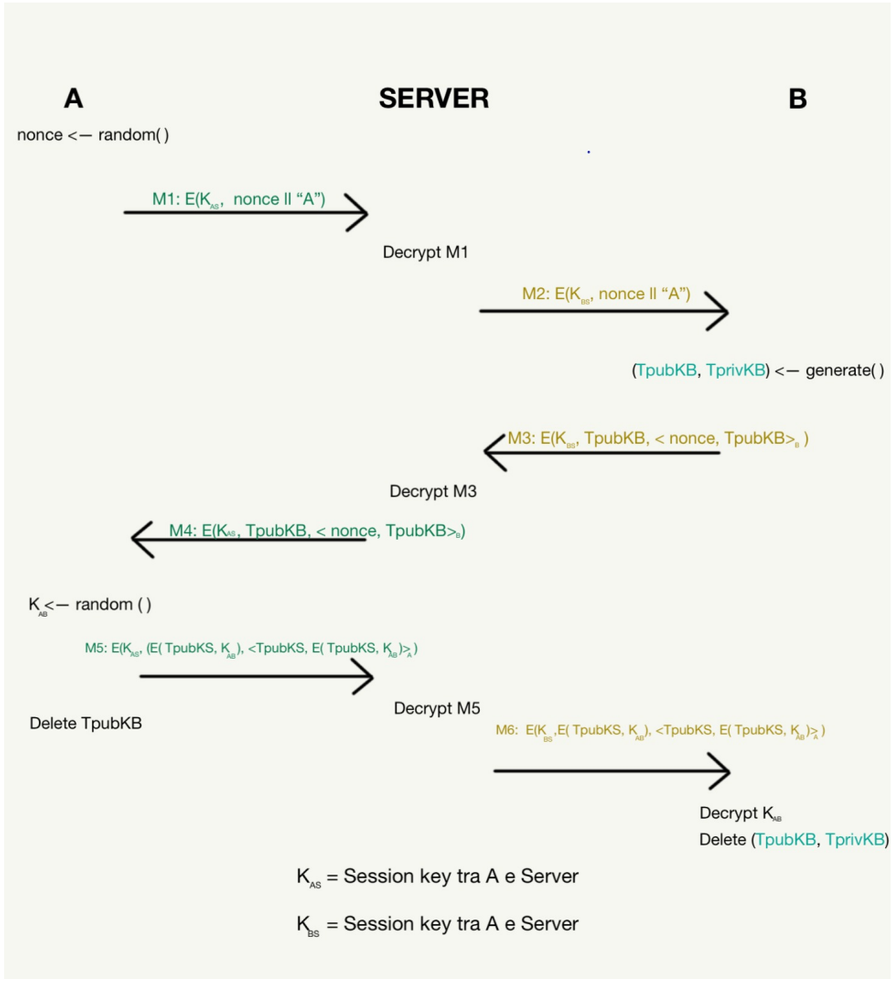

Figure 4: Ephemeral RSA (RSAE)

In particolare, ogni messaggio in questa fase passa dal server, che lo inoltra al corrispettivo peer. Non c'è mai un'interazione p2p.

5.8.1 Formato dei messaggi

Durante questa fase, ogni messaggio che arriva al server, ed inoltra il server, viene cifrato/decifrato con le chiavi di sessione che i due peer hanno col server. In questo modo vengono garantite l'autenticità e la freschezza di ogni messaggio:

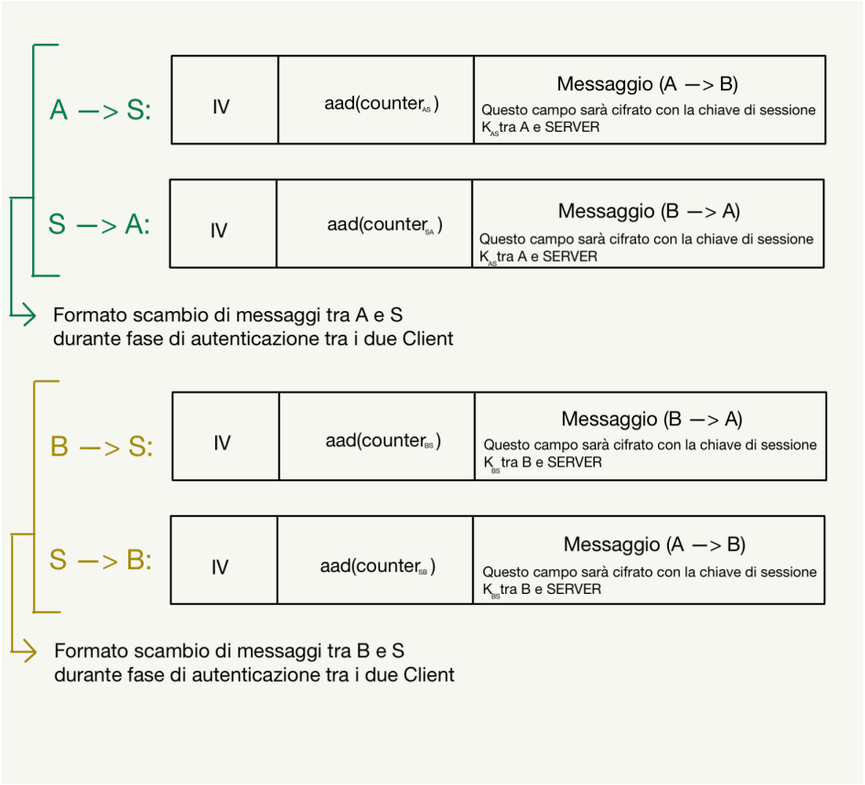

Figure 5: Formato dei messaggi

```
A -> S: [iv, aad(counter AS), E(Kas, messageAB)]
```

Il server verifica il messaggio con la chiave di sessione che mantiene con **A** e ricifra con la chiave di **B**

```
S -> B: [iv, aad(counter SB), E(Kbs, messageAB)]
```

5.9 Chat client-client

Come già osservato nella sezione **4.1**, abbiamo implementato un thread lettore e uno scritto nel client, in questo modo i peer hanno la possibilità di inviare e ricevere consecutivamente un numero arbitrario di messaggi.

5.9.1 Formato dei messaggi

Ogni messaggio scambiato tra i due peer durante la sessione è di tipo **AEAD**, ed è così strutturato:

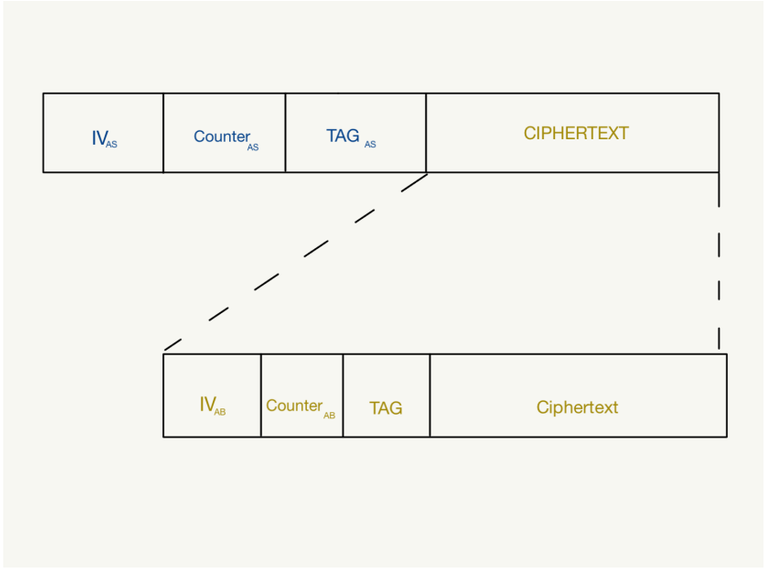

Figure 6: Formato dei messaggi

dove: CIPHERTEXT è cifrato con la chiave di sessione tra **A** ed **S**; viene decifrato da **S**, ricifrato con la chiave simmetrica condivisa tra **S** e **B** ed inoltrato a **B** tramite la tecnica **AAD**, il suo contenuto è quello illustrato nel secondo messaggio, dove Ciphertext è cifrato con la chiave di sessione tra **A** e **B**.