Università degli Studi di Napoli
‘Parthenope’

Dipartimento di Scienze e Tecnologie

Relazione del progetto di “Reti di Calcolatori”

Gestione di tessere bibliotecarie

Proponenti:

D’Aniello Eleonora

Parmendola Salvatore

Nazzaro Flavio

Matricole:

0124002790

0124002933

0124002946

Professori:

Scafuri Umberto

Data di consegna:

14/01/2026

Anno accademico: 2024/2025

Indice

Indice delle figure

Traccia

Realizzare un sistema per gestire tessere di abbonamento a una biblioteca. Un utente può registrare la sua tessera al sistema tramite un client che si connette alla biblioteca. La biblioteca comunica al **`ServerB`** i dettagli della tessera e la sua durata. Un **`ClientV`** verifica se una tessera è valida interagendo con il **`ServerG`** che richiede al **`ServerB`** il controllo. Un **`ClientU`** può sospendere o ripristinare una tessera a seguito di eventi come mancata restituzione di libri o rinnovo.

Descrizione del progetto

Il progetto è un sistema software sviluppato in **linguaggio C** che implementa la gestione di tessere bibliotecarie tramite un'**architettura client-server**. La comunicazione tra i vari componenti del sistema avviene tramite l'uso di **socket**, che permettono lo scambio di messaggi tra processi e thread concorrenti. Il sistema è progettato per operare in un ambiente "Unix-like" ed è stato sviluppato ed eseguito all'interno di una **virtual machine** basata su Linux Ubuntu, in particolare **Linux Mint**. Questa scelta garantisce la compatibilità con le API di sistema POSIX, fondamentali per la gestione di socket, thread e sincronizzazione. Lo sviluppo del progetto è stato effettuato utilizzando l'IDE **Visual Studio Code**.

Schemi dell'architettura

- `client`: usato dall'utente per collegarsi alla biblioteca e per registrare la tessera bibliotecaria;
- `biblioteca`: comunica al `serverB` i dettagli della tessera: codice della tessera (una stringa di dieci caratteri), la data di emissione, la data di scadenza e lo stato (attiva/inattiva);
- `serverB`: riceve la tessera bibliotecaria e comunica con il `serverG`, così da permettere operazioni di controllo e attivazione/disattivazione della tessera;
- `serverG`: fa da tramite tra il `serverB` e i `client U` e `V`;
- `clientV`: verifica se la tessera è valida (se non è scaduta o se non è stata sospesa) tramite il `serverG`;
- `clientU`: può sospendere (per mancato rinnovo) o ripristinare una tessera tramite il `serverG`.

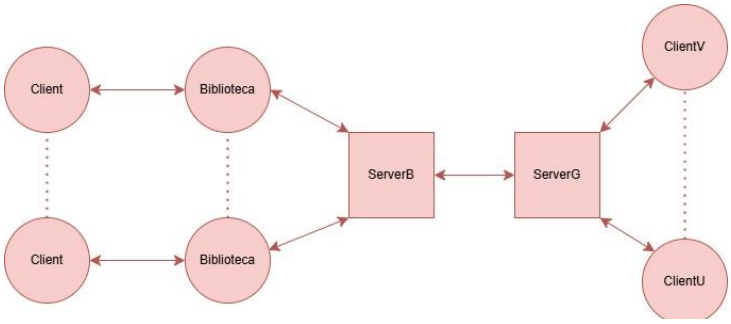

Figura 1: Schema dell'architettura del progetto

Schemi del protocollo applicazione

In questo progetto, la comunicazione tra i client e i server avviene grazie al protocollo **TCP** (Trasmission Control Protocol), scelto per garantire affidabilità e correttezza nello scambio dei dati. È un protocollo orientato alla connessione, che assicura che i messaggi inviati dal client arrivino al server:

- senza perdere i dati;
- nell'ordine corretto: evita situazioni critiche (ad esempio una sospensione eseguita prima di una verifica) perché le richieste e le risposte vengono elaborate nello stesso ordine in cui sono state inviate.
- senza duplicazioni.

Il TCP risulta essere appropriato anche per altri motivi, come:

- il **controllo della connessione**: la comunicazione avviene tramite una connessione stabile instaurata con le primitive `socket()`, `bind()`, `listen()`, `accept()` lato server e `connect()` lato client, consentendo un dialogo strutturato tra le parti;
- l'**integrazione con il multithreading**: ogni connessione TCP può essere gestita da un thread dedicato o inserita in una coda di task, permettendo al server di servire più client contemporaneamente.

I server vengono attivati all'indirizzo `localhost` (127.0.0.1) e le porte utilizzate da ogni server sono le seguenti:

- `biblioteca`: 55000;
- `serverG`: 50000;
- `serverB`: 60000;

Implementazione

Tutte le tessere bibliotecarie vengono salvate all'interno del file `tessere_bibliotecarie.txt`.

`tessera_bibliotecaria.h`

Questo file definisce la struttura dati principale utilizzata nel sistema di gestione della biblioteca. Questa struttura rappresenta una tessera bibliotecaria ed è condivisa tra client e server, fungendo da formato standard dei messaggi scambiati tramite socket TCP. L'uso di un file header comune garantisce coerenza nella rappresentazione dei dati e semplifica la comunicazione tra i diversi componenti del sistema.

```
#define TESSERA_LUNG 10
```

Definisce la lunghezza fissa del codice identificativo della tessera bibliotecaria. Questo vincolo consente:

- validazione immediata dell'input lato client;
- dimensione nota e costante dei messaggi scambiati;

```
#define ANNO 31536000
```

Rappresenta il numero di secondi corrispondente a un anno. Viene utilizzato per calcolare la data di scadenza della tessera a partire dalla data di emissione, sfruttando il tipo `time_t`.

Struttura `TesseraBibliotecaria`

```
struct TesseraBibliotecaria {

    char codice_tb[TESSERA_LUNG+1]; //codice della tessera bibliotecaria di tipo "char"

    time_t data_emissione; //data emissione della tessera bibliotecaria

    time_t data_scadenza; //data di scadenza della tessera bibliotecaria

    int servizio; //0: inserimento tessera, 1: controllo validita', 2: sospende/abilita la tessera

    int stato; // 1: attiva, 0: sospesa

};
```

La struttura contiene tutte le informazioni necessarie per la gestione di una tessera bibliotecaria:

- `codice_tb`: campo di tipo `char[]` che memorizza il codice identificativo della tessera. La dimensione `TESSERA_LUNG+1` consente di includere il terminatore di stringa '\0'. Il codice viene standardizzato in maiuscolo per evitare ambiguità;
- `data_emissione`: campo di tipo `time_t` che rappresenta la data di emissione della tessera. L'uso di `time_t` permette una gestione semplice e portabile delle date in ambiente Unix-like;
- `data_scadenza`: campo di tipo `time_t` che indica la data di scadenza della tessera. Viene calcolata aggiungendo la costante `ANNO` alla data di emissione, permettendo una gestione automatica della validità temporale.
- `servizio`: campo intero che identifica il tipo di operazione richiesta dal client:
    - 0: inserimento di una nuova tessera;
    - 1: controllo della validità di una tessera;
    - 2: sospensione o riattivazione della tessera

  Questo campo consente al server di distinguere le richieste e instradarle verso la logica appropriata;

- `stato`: campo intero che rappresenta lo stato corrente della tessera:
    - 1: tessera attiva;
    - 0: tessera sospesa.

  Viene utilizzato dal server per determinare se una tessera è utilizzabile o meno.

settings_Client_Server.h

Contiene le costanti di configurazione e le strutture di supporto utilizzate dai componenti client e server del sistema di gestione della biblioteca. Questo file centralizza le impostazioni di rete e di concorrenza, garantendo coerenza e manutenibilità del progetto. Vengono definite le porte dei vari server e l'indirizzo IP di loopback (indirizzo `LOCALHOST`) utilizzato per la comunicazione locale tra i processi all'interno della virtual machine.

```
#define THREAD_NUM 10
```

Specifica il numero di thread presenti nel thread pool del server backend. Questo valore consente di limitare il numero massimo di thread concorrenti, migliorando la gestione delle risorse e la scalabilità del sistema.

Struttura `Task`

```
/*rappresenta un compito da eseguire all'interno del threadpool

ogni task e' composta da:*/

typedef struct Task{

    void (*taskFunction)(void*); //puntatore a funzione che accetta un argomento generico (void*) e non restituisce nulla. Sara' la funzione da eseguire

    void* arg;

}Task;

/*questo schema consente di avere una coda di task eterogenee ma gestibili in modo uniforme dai thread del threadpool*/
```

La struttura `Task` rappresenta un compito da eseguire all'interno del thread pool:

- `taskFunction`: è un puntatore a funzione che:
  - accetta un argomento generico di tipo `void*`;
  - non restituisce alcun valore.

  Questo approccio permette di assegnare al thread pool funzioni diverse, mantenendo un'interfaccia uniforme;

- `arg`: è un puntatore generico utilizzato per passare i dati necessari all'esecuzione del task. La responsabilità della conversione del tipo (cast) è demandata alla funzione eseguita.

Questa struttura consente di implementare una coda di task eterogenei, che possono essere:

- inseriti nella coda dal thread principale del server;
- estratti ed eseguiti dai thread worker del thread pool.

Questo schema permette di:

- separare l'accettazione delle connessioni dall'elaborazione delle richieste;

- migliorare la concorrenza e le prestazioni del server;
- evitare la creazione di un thread per ogni client.

Dettagli implementativi dei client

Tutti i client hanno delle operazioni in comune:

- Il client verifica, innanzitutto, che il numero di argomenti passati da linea di comando sia corretto e che il codice della tessera abbia la lunghezza prevista (`TESSERA_LUNG`, constante presente all'interno del file "`tessera_bibliotecaria.h`"). Questa fase permette di intercettare errori di input lato client, evitando comunicazioni inutili con il server;
- **creazione della socket TCP**: ogni client crea una socket(utilizzando una funzione che restituisce una variabile di tipo `int`) utilizzando come parametri:
  - dominio `AF_INET`(IPV4);
  - tipo `SOCK_STREAM`;
  - protocollo TCP.

  ISTRUZIONE COMPLETA: `nome_variabile = socket(AF_INET, SOCK_STREAM, 0);`

- **impostazione dell'indirizzo del server**: viene inizializzata una struttura `sockaddr_in` contenente:
  - indirizzo IP del server (il `LOCALHOST` in questo caso);
  - porta del server;
  - famiglia di indirizzi IPV4.
- **connessione al server**: il client stabilisce la connessione TCP al server tramite la chiamata: `connect(nome_socket, (struct sockaddr*)&ind, sizeof(ind));`
  In caso di errore, l'esecuzione viene interrotta, garantendo che le operazioni successive avvengano solo in presenza di una connessione valida;
- **invio dati al server**: avviene tramite la chiamata al funzione `send()`.
- **chiusura della connessione**: al termine della comunicazione, il client chiude correttamente la socket tramite `close()`, liberando le risorse di sistema.

`client`

È il client principale perché si occupa di comunicare al server `biblioteca` il codice della tessera bibliotecaria (come argomento da linea di comando) tramite una connessione TCP. Viene verificato l'input da linea di comando, poi creata la socket TCP, impostato l'indirizzo del server (in questo caso consideriamo la porta del server `biblioteca`) e richiesta la connessione al server. Il client inizializza una struttura `TesseraBibliotecaria`:

- azzera la memoria per evitare valori non inizializzati;
- copia il codice della tessera ricevuto da linea di comando;
- converte il codice in maiuscolo per garantire una rappresentazione standardizzata dei dati.

Questa struttura rappresenta il messaggio di richiesta inviato al server. La struttura contenente i dati della tessera viene inviata al server tramite la chiamata `send()`. Il client attende la risposta del server tramite `recv()`, che consiste in un valore intero (`risposta`) indicante l'esito dell'operazione (rappresentato da una variabile di tipo `int`). In base al valore ricevuto:

- 0: tessera già inserita;
- valore diverso da 0: tessera valida.

Il risultato viene mostrato all'utente. Al termine dell'operazione, il client chiude la socket.

`clientU`

È un client dedicato alla gestione dello stato delle tessere bibliotecarie, in particolare alle operazioni di sospensione e riattivazione. Il codice della tessera viene passato come argomento da linea di comando e inviato al serverG, che si occupa di inoltrare la richiesta al `serverB`. Viene verificato l'input da linea di comando, poi creata la socket TCP, impostato l'indirizzo del server (in questo caso consideriamo la porta del `serverG`) e richiesta la connessione al server. Il client inizializza una struttura `TesseraBibliotecaria`:

- azzerando la memoria con `memset`;
- copiando il codice della tessera;
- convertendo il codice in maiuscolo per standardizzare il formato;
- impostando il campo `servizio = 2` che identifica l'operazione di sospensione/riattivazione della tessera (vedi "`tessera_bibliotecaria.h`").

La struttura rappresenta il messaggio di richiesta inviato al server (tramite la chiamata `send()`). Il `serverG` interpreta il campo servizio e instrada la richiesta al componente appropriato. Il client riceve un valore intero che rappresenta l'esito dell'operazione:

- 0: tessera invalidata;
- -2: tessera non trovata;
- altro valore: tessera valida o riattivata.

Il risultato viene comunicato all'utente tramite messaggi testuali. Al termine dell'operazione, il client chiude la socket.

`clientV`

È il client dedicato al controllo della validità di una tessera bibliotecaria.

Consente all'utente di verificare se una tessera:

- esiste nel sistema;
- è attiva,
- non è scaduta.

Il client comunica tramite protocollo TCP con il server intermedio (`serverG`), al quale invia una richiesta strutturata che verrà poi inoltrata al `serverB`. Il codice della tessera viene passato come argomento da linea di comando. Viene verificato l'input da linea di comando, poi creata la socket TCP, impostato l'indirizzo del server (in questo caso consideriamo la porta del `serverG`) e richiesta la connessione al server. Viene inizializzata una struttura `TesseraBibliotecaria`:

- il codice della tessera viene copiato dalla linea di comando;
- il codice viene convertito in maiuscolo per garantire uniformità;
- il campo servizio viene impostato a 1, che identifica il servizio di verifica della validità.

La struttura `TesseraBibliotecaria` viene inviata al server tramite `send()`.

Il client riceve dal server un valore intero che rappresenta l'esito della verifica:

- 1: tessera valida;
- 0: tessera non valida (scaduta o sospesa);
- -2: tessera non trovata.

In base al valore ricevuto, viene mostrato un messaggio appropriato all'utente. Al termine, la socket viene chiusa.

Dettagli implementativi dei server

`biblioteca`

È un server TCP concorrente che gestisce le richieste di inserimento di nuove tessere bibliotecarie. Il server è progettato secondo il modello **thread pool + coda di task**, in modo da gestire più connessioni simultanee in maniera efficiente e controllata. Il server si occupa di:

- accettare le connessioni dei client;
- delegare la gestione delle richieste ai thread del pool;
- comunicare con il `serverB` per la memorizzazione persistente delle tessere;
- inoltrare la risposta al client.

Il server utilizza:

- un **thread principale** che accetta le connessioni in ingresso;
- un **thread pool** di dimensione fissa (`THREAD_NUM`);
- una **coda di task condivisa**, protetta da **mutex** e **variabile di condizione**.

Ogni connessione client viene incapsulata in una task ed eseguita da uno dei thread worker.

All'invio, il server:

- installa un signal handler per la gestione del segnale `SIGINT`;
- inizializza mutex e variabile di condizione;
- crea la socket TCP IPV4;
- associa il socket alla porta `BIBLIOTECA`;
- pone il socket in ascolto tramite `listen()`.

Il server crea un numero fisso di thread e ogni thread esegue la funzione `routine`, che resta in attesa di task da elaborare. Il thread principale esegue un ciclo `while(running)` in cui:

- accetta nuove connessioni tramite `accept()`;
- alloca dinamicamente il socket del client;
- crea una task associata alla funzione `connessioni`;
- inserisce la task nella coda tramite `submitTask()`.

La funzione `connessioni` (per ogni connessione):

- riceve dal client una struttura `TesseraBibliotecaria`;
- inizializza i campi:
  - `data_emissione`;
  - `data_scadenza`;
  - `servizio = 0` (inserimento);
  - `stato = 1` (attiva).
- crea una nuova connessione TCP verso il `serverB`;
- invia la struttura al `serverB`;
- riceve la risposta dal `serverB`;
- inoltra la risposta al client originale;
- chiude tutte le socket e libera la memoria allocata.

Il server `Biblioteca` agisce quindi come intermediario tra client e backend.

La concorrenza è gestita tramite:

- `pthread_mutex_t`: per proteggere l'accesso alla coda di task;
- `pthread_cond_t`: per sincronizzare thread produttore(main) e consumatore(worker);
- un flag `running` per consentire una terminazione ordinata.

Questo approccio evita race condition e garantisce un uso efficiente delle risorse.

Alla ricezione del segnale `SIGINT`:

- Il flag `running` viene impostato a 0;
- La socket principale viene chiusa per sbloccare `accept()`;
- Tutti i thread in attesa vengono risvegliati tramite `pthread_cond_broadcast()`.

Questo consente una terminazione pulita del server.

serverB

È il server backend del sistema di gestione della biblioteca. È un server TCP concorrente, basato su thread pool, responsabile della gestione persistente delle tessere bibliotecarie e dell'elaborazione dei servizi richiesti dai client tramite i server intermedi. Il `serverB` rappresenta il livello più basso dell'architettura ed è l'unico componente che accede direttamente al file di persistenza `tessere_bibliotecarie.txt`.

Il server backend:

- riceve richieste strutturate di tipo `TesseraBibliotecaria`;
- interpreta il campo `servizio` per determinare l'operazione da eseguire;
- gestisce l'accesso concorrente al file delle tessere;
- invia al chiamante una risposta che indica l'esito dell'operazione.

Il `serverB` utilizza:

- un thread principale per l'accettazione delle connessioni TCP;
- un thread pool di dimensione fissa;
- una coda di task condivisa;
- mutex e variabili di condizione per la sincronizzazione;
- un mutex dedicato (`mutex_file`) per proteggere l'accesso al file.

All'avvio, il server:

- registra un gestore per il segnale `SIGINT`;
- inizializza mutex e variabili di condizione;
- crea e configura una socket TCP IPv4;
- associa la socket alla porta `SERVERB`;
- pone il socket in ascolto tramite `listen()`.

Il server crea un numero fisso di thread worker che eseguono la funzione `routine`. Ogni thread resta in attesa di task da eseguire. Il thread principale accetta nuove connessioni tramite `accept()`. Ogni connessione viene incapsulata in una task e inserita nella coda tramite `submitTask()`. Il server riceve una struttura `TesseraBibliotecaria` contenente il codice della tessera e il tipo di servizio richiesto. L'apertura e la modifica del file `tessere_bibliotecarie.txt` avvengono all'interno di una sezione critica protetta da `mutex_file`, garantendo sicurezza in presenza di accessi concorrenti.

Servizio 0 – Inserimento tessera

- verifica la presenza di duplicati;
    - in caso positivo, inserisce una nuova riga nel file;
- restituisce 1 se l'inserimento è avvenuto correttamente, 0 in caso di duplicato.

Servizio 1 – Verifica validità

- ricerca il codice nel file;
- controlla stato e data di scadenza;
- restituisce:
    - 1: tessera valida,
    - 0: tessera non valida,
    - 2: tessera non trovata.

Servizio 2 – Sospensione/Riattivazione

- individua la riga corrispondente alla tessera;
- modifica lo stato della tessera;
- aggiorna il file in-place;
- restituisce il nuovo stato oppure -2 se non trovata.

Al termine dell'elaborazione, il server invia al client un valore intero che rappresenta l'esito dell'operazione richiesta.

Alla ricezione di SIGINT:

- il flag `running` viene impostato a 0;
- la socket principale viene chiusa;
- i thread in attesa vengono risvegliati per terminare correttamente.

`serverG`

Il serverG è il server gateway del sistema client-server. Agisce come livello di intermediazione tra i `client`(`U` e `V`) e il `serverB` occupandosi dell'inoltro delle richieste e delle risposte senza accedere direttamente ai dati persistenti. Il `serverG` implementa un server TCP concorrente basato su thread pool e una coda di task, consentendo la gestione efficiente di più richieste simultanee. Il `serverG`:

- riceve richieste dai client;
- incapsula le richieste in task concorrenti;
- inoltra la richiesta al `serverB`;
- riceve la risposta dal `serverB`;

- restituisce la risposta al client chiamante.

Non contiene logica applicativa né accesso diretto al file.

Il `serverG` utilizza:

- un thread principale per l'accettazione delle connessioni TCP;
- un thread pool di dimensione fissa;
- una coda di task FIFO condivisa;
- mutex e variabili di condizione per la sincronizzazione;
- un flag globale (`running`) per la terminazione controllata.

All'avvio il server:

- associa un gestore al segnale SIGINT;
- inizializza mutex e variabile di condizione;
- crea una socket TCP IPv4;
- effettua il `bind`() sulla porta `serverG`;
- pone il socket in ascolto con `listen()`.

Vengono creati `THREAD_NUM` thread worker che eseguono la funzione `routine`. Ogni thread resta in attesa di task da elaborare. Il thread principale:

- accetta nuove connessioni dai client;
- alloca dinamicamente il descrittore della socket;
- crea una task associata alla funzione `connessioni`;
- inserisce la task nella coda condivisa tramite `submitTask()`.

Il server riceve una struttura `TesseraBibliotecaria` contenente:

- codice della tessera;
- tipo di servizio richiesto (verifica, sospensione, inserimento).

Per ogni richiesta:

- viene creata una nuova socket TCP;
- il `serverG` si connette al `serverB` sulla porta `SERVERB`;
- la struttura ricevuta dal client viene inoltrata al backend.

Il `serverG`:

- riceve un valore intero da `serverB` che rappresenta l'esito dell'operazione;
- inoltra la stessa risposta al client senza modificarla.

Al termine:

- vengono chiuse le socket verso client e `serverB`;
- viene liberata la memoria allocata dinamicamente.

- la coda di task è protetta da un mutex (`mutexQueue`).
- i thread in attesa vengono sospesi tramite `pthread_cond_wait`.
- l'arrivo di nuove task viene segnalato con `pthread_cond_signal`.

Questo garantisce:

- assenza di race condition;
- corretto bilanciamento del carico;
- utilizzo efficiente delle risorse.

Alla ricezione del segnale `SIGINT`:

- il flag `running` viene impostato a 0;
- la socket di ascolto viene chiusa per forzare l'uscita da `accept()`;
- tutti i thread in attesa vengono risvegliati tramite `pthread_cond_broadcast()`.

Istruzioni per la compilazione

Per la compilazione dei file, i comandi sono i seguenti:

```
gcc -o biblioteca biblioteca.c

gcc -o serverB serverB.c

gcc -o serverG serverG.c

gcc -o clientV clientV.c

gcc -o clientU clientU.c

gcc -o client client.c
```

Per facilitare la compilazione, abbiamo creato un `Makefile`:

```
# compilatore

CC = gcc

# abilita i warnings

CFLAGS = -g -Wall


all: client clientU clientV biblioteca serverB serverG


client: client.c tessera_bibliotecaria.h settings_Client_Server.h

  $(CC) $(CFLAGS) client.c -o client
```

```
clientU: clientU.c tessera_bibliotecaria.h
settings_Client_Server.h
  $(CC) $(CFLAGS) clientU.c -o clientU


clientV: clientV.c tessera_bibliotecaria.h
settings_Client_Server.h
  $(CC) $(CFLAGS) clientV.c -o clientV


biblioteca: biblioteca.c tessera_bibliotecaria.h
settings_Client_Server.h
  $(CC) $(CFLAGS) biblioteca.c -o biblioteca


serverB: serverB.c tessera_bibliotecaria.h
settings_Client_Server.h
  $(CC) $(CFLAGS) serverB.c -o serverB


serverG: serverG.c tessera_bibliotecaria.h
settings_Client_Server.h
  $(CC) $(CFLAGS) serverG.c -o serverG


clean:
  rm -f *.o
  rm -f clientU clientV client biblioteca serverG serverB
```

Nella radice del progetto, richiamiamo in linea di comando "`make`" (per compilare tutto) e poi, terminato l'utilizzo del programma, richiamiamo "`make clean`" da terminale per eliminare gli eseguibili.

Istruzioni per l'esecuzione

Per poter eseguire (senza problemi) il programma, è necessario eseguire questi comandi in ordine:

```
./biblioteca
```

`./serverB`

`./serverG`

Dopodiché, possiamo eseguire i singoli client:

`./client <codice_tessera>`

`./clientV <codice_tessera>`

`./clientU <codice_tessera>`

Simulazione dell'applicazione

Inserimento

Usando la sintassi `./client <codice_tessera>`, otteniamo:

```
eleonora@eleonoradan:~/Scrivania/Gestione-Biblioteca$ ./client 12wer3t4yu
La tessera bibliotecaria e' valida.
```

Figura 2: codice con lunghezza fissa corretta e tessera non ancora presente nel file

```
eleonora@eleonoradan:~/Scrivania/Gestione-Biblioteca$ ./client 12wer3t4yu
La tessera bibliotecaria e' gia' stata inserita.
```

Figura 3: codice con lunghezza fissa corretta, ma già presente nel file

```
eleonora@eleonoradan:~/Scrivania/Gestione-Biblioteca$ ./client 23456w
Errore: il codice deve essere lungo 10 caratteri.
```

Figura 4: codice con lunghezza fissa non corretta

```
1  12WER3T4YU, emissione: 14-01-2026, scadenza: 14-01-2027, stato: 1
```

Figura 5: file di testo con una nuova tessera inserita

Controllo validità

Usando la sintassi `./clientV <codice_tessera>`, otteniamo:

```
eleonora@eleonoradan:~/Scrivania/Gestione-Biblioteca$ ./clientV 12wer3t4yu
La tessera bibliotecaria e' valida.
```

Figura 6: tessera presente nel file e ancora attiva

```
eleonora@eleonoradan:~/Scrivania/Gestione-Biblioteca$ ./clientV 12wer3t4y
Errore: il codice deve essere lungo 10 caratteri.
```

Figura 7: codice con lunghezza fissa errata

```
eleonora@eleonoradan:~/Scrivania/Gestione-Biblioteca$ ./clientV 12wer3t4yy
La tessera non e' stata trovata.
```

Figura 8: tessera non presente nel file

```
eleonora@eleonoradan:~/Scrivania/Gestione-Biblioteca$ ./clientV 12wer3t4yu
La tessera bibliotecaria non e' valida (scaduta o mancato rinnovo).
```

Figura 9: tessera presente nel file, ma sospesa

Attivazione/sospensione tessera

Usando la sintassi `./clientU <codice_tessera>`, otteniamo:

```
eleonora@eleonoradan:~/Scrivania/Gestione-Biblioteca$ ./clientU 12wer3t4yu
La tessera bibliotecaria e' stata invalidata.
```

Figura 10: tessera presente nel file e attiva

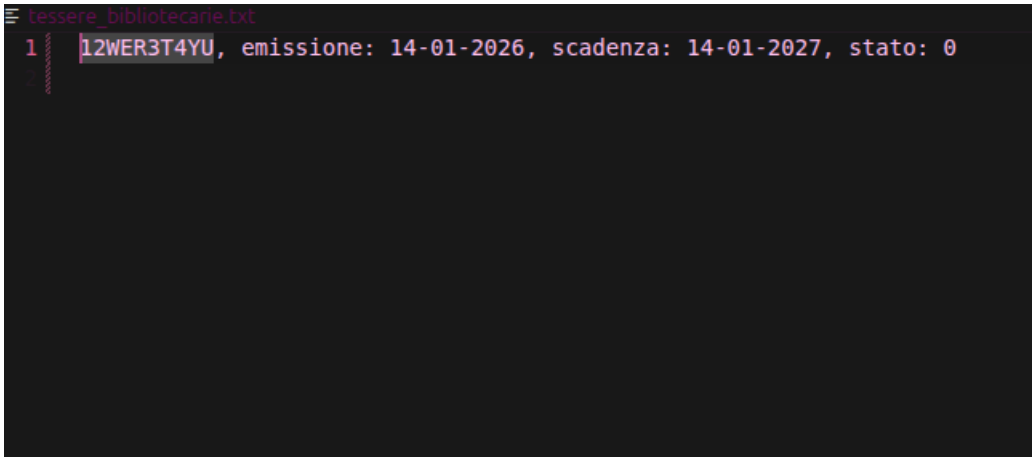

Figura 11: file txt dopo aver utilizzato il `clientU` con una tessera inizialmente attiva

```
eleonora@eleonoradan:~/Scrivania/Gestione-Biblioteca$ ./clientU 12wer3t4yu
La tessera bibliotecaria e' valida.
```

Figura 12: tessera inizialmente sospesa

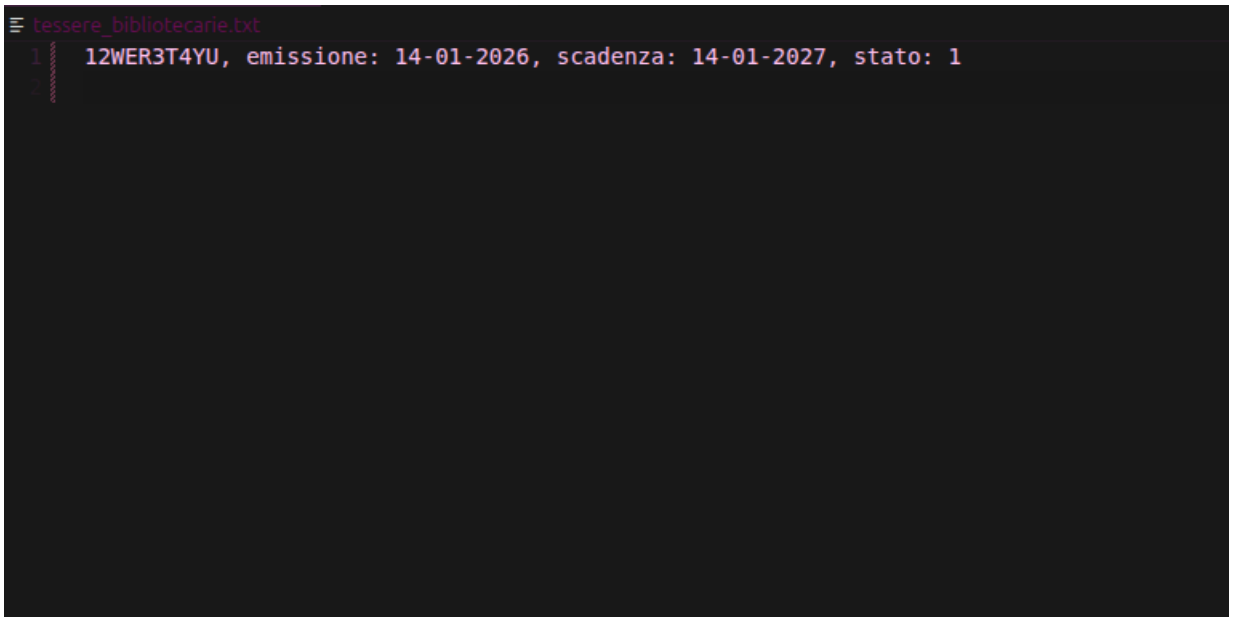

Figura 13: file txt dopo che una tessera inizialmente sospesa è stata riattivata

```
eleonora@eleonoradan:~/Scrivania/Gestione-Biblioteca$ ./clientU 12wer3t4yy
La tessera bibliotecaria non e' stata trovata.
```

Figura 14: tessera non presente nel file txt

```
eleonora@eleonoradan:~/Scrivania/Gestione-Biblioteca$ ./clientU 12wer3t4y
Errore: il codice deve essere lungo 10 caratteri.
```

Figura 15: codice di lunghezza fissa errata