Hands-On 14: midterm

Server TCP multithread

Sandi Russo
Corso di Laurea: Scienze Informatiche

30 maggio 2025

Indice

1 Introduzione

La seguente esercitazione descrive l'architettura e l'implementazione di un'applicazione client-server sviluppata per la gestione di movimenti su un conto corrente bancario. L'obiettivo del progetto è stato quello di creare un sistema robusto e concorrente, capace di gestire richieste simultanee provenienti da più punti di accesso, simulando un ambiente bancario realistico dove diversi canali (come sportelli, app e home banking) interagiscono con un'unica base dati.
Il cuore del sistema è un server TCP multithread, programmato in linguaggio C, che gestisce una struttura dati condivisa contenente le operazioni finanziarie. Per garantire l'integrità dei dati in un contesto concorrente, sono state implementate specifiche primitive di sincronizzazione. Il client, anch'esso sviluppato in C, fornisce un'interfaccia a riga di comando per consentire all'utente di interagire con il server ed eseguire le operazioni richieste. La soluzione è stata progettata in modo modulare per favorire la leggibilità, la manutenibilità e la potenziale espansione futura del codice.

2 Definizione del Problema

L'esercizio richiedeva la realizzazione di un sistema per il tracciamento dei movimenti di un conto corrente, identificati da un ID univoco, un importo e una causale. Il requisito fondamentale era la capacità del server di gestire l'accesso concorrente da parte di un massimo di 10 client simultanei. Questi client rappresentano i diversi canali attraverso cui un utente può operare sul proprio conto.

Le operazioni che il sistema deve supportare sono le quattro operazioni CRUD fondamentali:

- **ADD**: Aggiungere un nuovo movimento.
- **DELETE**: Cancellare un movimento esistente tramite il suo ID.
- **UPDATE**: Modificare i dati di un movimento esistente.
- **LIST**: Visualizzare l'elenco completo di tutti i movimenti registrati.

Il progetto doveva essere sviluppato in modo modulare, separando la logica di rete, la gestione del client e la manipolazione della struttura dati. Era inoltre richiesta la creazione di un `Makefile` per automatizzare il processo di compilazione e la stesura di una relazione tecnica e di una presentazione per illustrare il lavoro svolto.

3 Metodologia

La soluzione è stata implementata seguendo un'architettura client-server basata sul protocollo TCP, scelto per la sua natura affidabile e orientata alla connessione, essenziale per transazioni finanziarie. La modularità è stata ottenuta suddividendo il codice sorgente in più file, ciascuno con una responsabilità specifica.

3.1 Struttura del Progetto

Il codice è organizzato nei seguenti file:

- `server.c`: Contiene il punto di ingresso principale del server. Si occupa di creare il socket di ascolto, mettersi in attesa di connessioni e, per ogni nuova connessione accettata, avviare un thread dedicato alla sua gestione.
- `client.c`: Implementa il programma client che si connette al server e fornisce all'utente un'interfaccia per inviare comandi.
- `gestioneClient.c`: Contiene la funzione `gestisci_client`, eseguita da ogni thread del server. Questa funzione gestisce l'intero ciclo di vita della comunicazione con un singolo client, ricevendo i comandi, interpretandoli e invocando le funzioni appropriate per manipolare i dati.
- `gestioneLista.c`: Rappresenta il livello di accesso ai dati. Implementa la struttura dati (una lista concatenata) per memorizzare i movimenti e fornisce le funzioni per le operazioni di ADD, DELETE, UPDATE e LIST. Crucialmente, gestisce anche la sincronizzazione dell'accesso a questa risorsa condivisa.
- `header.h`: File di intestazione che contiene le definizioni delle strutture dati comuni (come `Movimento`), le costanti e i prototipi di tutte le funzioni globali, garantendo coerenza e coesione tra i vari moduli.

3.2 Gestione della Concorrenza e Sincronizzazione

La sfida principale del problema era gestire gli accessi concorrenti alla lista dei movimenti. Per risolvere questo problema, è stato adottato il modello "un thread per connessione" (*thread-per-connection*). In `server.c`, il thread principale entra in un ciclo infinito in cui la chiamata bloccante `accept()` attende nuove connessioni. Non appena un client si connette, `accept()` restituisce un nuovo descrittore di socket per quella specifica connessione. A questo punto, il server crea un nuovo thread utilizzando `pthread_create`, passando il socket del client come argomento alla funzione `gestisci_client`. Il thread viene creato in modalità *detached* (`pthread_detach`) in modo che le sue risorse vengano liberate automaticamente alla sua terminazione, senza che il thread principale debba attendere con una `pthread_join`.
La lista dei movimenti, definita come variabile statica globale in `gestioneLista.c`, è una risorsa condivisa tra tutti i thread. Per prevenire fenomeni di *race condition* e garantire la mutua esclusione, l'accesso a tale lista è protetto da un mutex (`pthread_mutex_t`). Il mutex, chiamato `lista_mutex`, viene inizializzato all'avvio del server tramite la funzione `init_lista()`. Ogni funzione che modifica o legge la lista (`add_operazione`, `delete_operazione`, `update_operazione` e `list_operazione`) acquisisce il lock sul mutex con `pthread_mutex_lock()` prima di accedere ai dati e lo rilascia con `pthread_mutex_unlock()` subito dopo. Questo meccanismo garantisce che solo un thread alla volta possa eseguire codice all'interno della sezione critica, preservando così la consistenza e l'integrità della struttura dati.

3.3 Flusso di Comunicazione

Il flusso operativo inizia con l'avvio del server, che si mette in ascolto sulla porta `8888`. Un client può quindi connettersi specificando l'indirizzo IP e la porta del server. Una volta stabilita la connessione, il thread server dedicato invia un messaggio di benvenuto con le istruzioni sui comandi disponibili.

Da quel momento, il client entra in un ciclo in cui:

1. Legge un comando dall'input dell'utente (`fgets`).
2. Invia il comando al server tramite `send()`.
3. Attende una risposta dal server tramite la chiamata bloccante `recv()`.
4. Stampa la risposta ricevuta a schermo.

Il server, nel frattempo, nel suo thread dedicato, attende i comandi con `recv()`. Una volta ricevuto un messaggio, ne esegue il parsing con `sscanf` per identificare il comando e i suoi parametri. A seconda del comando, invoca la funzione appropriata da `gestioneLista.c`, la quale, come descritto, opera sulla lista in modo thread-safe. Infine, il server invia una stringa di risposta al client per notificarne l'esito. Il ciclo termina quando il client invia il comando `EXIT`.

4 Presentazione dei risultati

4.1 header.h

Il file header definisce la struttura dati `Movimento`, le costanti condivise e i prototipi di funzione utilizzati in tutto il progetto, garantendo la modularità.

```c
#ifndef HEADER_H
#define HEADER_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

#define MAX_LEN_MESSAGGIO 256
#define MAX_LEN_CAUSALE 50

// La utilizzo per gestire il singolo movimento e il nodo della lista del movimento
typedef struct Movimento {
    int id_operazione;
    float importo;
    char causale[MAX_LEN_CAUSALE];
    struct Movimento *next; // punto al prossimo nodo, ovvero al prossimo movimento nella lista
} Movimento;

void* gestisci_client(void* arg); // la eseguira ogni client
```

```c
void init_lista(); // funzione per l'inizializzazione della lista prima dell'utilizzo effettivo
void add_operazione(float importo, char* causale); // aggiungo un movimento in testa alla lista (uso le LIFO)
int delete_operazione(int id_da_cancellare); // restituisce 1 in caso di successo, 0 in caso di errore
int update_operazione(int id_da_aggiornare, float nuovo_importo, char* nuova_causale); // restituisce 1 (successo), 0 (fallimento)
void list_operazione(int client_socket);
void pulizia();

#endif
```

Listing 1: File di intestazione `header.h`

4.2 server.c

Questo file contiene la logica principale del server: inizializzazione del socket, binding, ascolto e ciclo di accettazione delle connessioni dei client, con la creazione di un thread per ciascuna.

```c
#include "header.h"

#define PORTA 8888

int main () {
    // Creo la lista e la inizializzo
    init_lista();

    int server_socket;
    struct  sockaddr_in server_addr;

    server_socket = socket (AF_INET, SOCK_STREAM, 0);

    if (server_socket == -1 ) {
        perror("Errore nella creazione della socket");
        exit(EXIT_FAILURE);
    }

    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORTA);

    if (bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        perror("Bind fallito");
        exit(EXIT_FAILURE);
    }

    listen(server_socket, 10);
    printf("Server in ascolto sulla porta %d...\n", PORTA);


    while (1) {
        // La chiamata accept() e BLOCCANTE
        int client_socket = accept(server_socket, NULL, NULL);
        if (client_socket < 0) {
            perror("Accept fallito");
```

```
            continue;
        }

        printf("Nuova connessione accettata (socket %d)\n", client_socket);

        // Alloco memoria per passare il socket al thread in modo sicuro
        int* new_sock = malloc(sizeof(int));
        *new_sock = client_socket;

        pthread_t thread_id;
        if (pthread_create(&thread_id, NULL, gestisci_client, (void*)
    new_sock) < 0) {
            perror("Errore creazione thread");
            free(new_sock); // Libera la memoria se il thread non parte
        }
        pthread_detach(thread_id);
    }

    close(server_socket);
    pulizia();
    return 0;
}
```

Listing 2: Codice del server principale `server.c`

4.3 client.c

Il client stabilisce una connessione TCP con il server, invia i comandi inseriti dall'utente e visualizza le risposte del server.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#include "header.h"

#define SERVER_IP "127.0.0.1" // Indirizzo IP del server (localhost)
#define SERVER_PORT 8888      // Porta del server (deve corrispondere a
    quella del server.c)

int main() {
    int sock;
    struct sockaddr_in server_addr;
    char message[MAX_LEN_MESSAGGIO];
    char server_reply[MAX_LEN_MESSAGGIO * 10]; // Buffer piu grande per
    risposte lunghe del server

    // Creazione del socket TCP, AF_INET per IPv4, SOCK_STREAM per TCP, 0
    per il protocollo di default (TCP)
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("Errore creazione socket client");
        return 1;
    }
```

```
25
26      server_addr.sin_family = AF_INET;
27      server_addr.sin_port = htons(SERVER_PORT); // Converte la porta in
        network byte order
28      server_addr.sin_addr.s_addr = inet_addr(SERVER_IP);
29
30      // Questa e una chiamata BLOCCANTE. Il client attende che il server
        accetti la connessione.
31      if (connect(sock, (struct sockaddr*)&server_addr, sizeof(server_addr))
        < 0) {
32          perror("Connessione al server fallita");
33          close(sock);
34          return 1;
35      }
36      printf("Connesso al server %s sulla porta %d\n", SERVER_IP, SERVER_PORT
        );
37
38      // Ricezione del messaggio di benvenuto/istruzioni dal server
39      int read_size = recv(sock, server_reply, sizeof(server_reply) - 1, 0);
40      if (read_size > 0) {
41          server_reply[read_size] = '\0';
42          printf("%s", server_reply); // Il server invia gia i comandi e un
        newline
43      } else {
44          printf("Nessun messaggio di benvenuto ricevuto o errore.\n");
45          close(sock);
46          return 1;
47      }
48
49      // Ciclo di comunicazione
50      while (1) {
51          printf("> ");
52          if (fgets(message, sizeof(message), stdin) == NULL) {
53              break; // Errore o EOF
54          }
55          message[strcspn(message, "\n")] = 0; // Rimuove newline
56
57          if (strlen(message) == 0) {
58              continue;
59          }
60          // Invia il comando al server
61          if (send(sock, message, strlen(message), 0) < 0) {
62              perror("Invio fallito");
63              break;
64          }
65          if (strcasecmp(message, "EXIT") == 0) {
66              printf("Disconnessione...\n");
67              break;
68          }
69          // Ricevi la risposta dal server
70          read_size = recv(sock, server_reply, sizeof(server_reply) - 1, 0);
71          if (read_size > 0) {
72              server_reply[read_size] = '\0';
73              printf("Server: %s", server_reply);
74          } else if (read_size == 0) {
75              printf("Server ha chiuso la connessione.\n");
76              break;
77          } else {
```

```
            perror("Errore ricezione");
            break;
        }
    }

    // Chiusura del socket
    close(sock);
    printf("Socket client chiuso.\n");

    return 0;
}
```

Listing 3: Codice del client `client.c`

4.4 gestioneClient.c

Questa funzione, eseguita in un thread separato per ogni client, gestisce la comunicazione: riceve, interpreta ed esegue i comandi inviati dal client.

```
#include "header.h"

void* gestisci_client(void* arg) {
    int client_socket = *(int*)arg;
    free(arg); // libero la memoria allocata nel server.c

    char buffer[MAX_LEN_MESSAGGIO];
    int read_size;

    // Invia un messaggio di benvenuto/istruzioni al client
    char* welcome_msg = "Comandi: ADD <imp> <caus>, DEL <id>, UPD <id> <imp> <caus>, LIST, EXIT\n";
    send(client_socket, welcome_msg, strlen(welcome_msg), 0);

    // la chiamata recv e bloccante
    while ((read_size = recv(client_socket, buffer, MAX_LEN_MESSAGGIO, 0)) > 0) {
        buffer[read_size] = '\0';

        char comando[20] = {0}; // Aumentato per sicurezza, inizializzato
        int id;
        float importo;
        char causale[MAX_LEN_CAUSALE] = {0}; // Inizializzato

        // Estrai solo la prima "parola" come comando
        sscanf(buffer, "%19s", comando);

        if (strcasecmp(comando, "ADD") == 0) {
            // %*s salta la prima parola (il comando) che abbiamo gia letto con sscanf precedente
            if (sscanf(buffer, "%*s %f %s", &importo, causale) == 2 && strlen(causale) > 0) {
                add_operazione(importo, causale);
                send(client_socket, "OK: Movimento aggiunto.\n", strlen("OK: Movimento aggiunto.\n") ,0);
            } else {
                send(client_socket, "ERRORE: Formato ADD non valido. Usa: ADD <importo> <causale>\n", strlen("ERRORE: Formato ADD non valido. Usa: ADD <importo> <causale>\n"), 0);
```

```
            }
        } else if (strcasecmp(comando, "DEL") == 0) {
            if (sscanf(buffer, "%*s %d", &id) == 1) {
                if (delete_operazione(id)) {
                    send(client_socket, "OK: Movimento cancellato.\n",
    strlen("OK: Movimento cancellato.\n"), 0);
                } else {
                    send(client_socket, "ERRORE: ID non trovato per DEL.\n"
    , strlen("ERRORE: ID non trovato per DEL.\n"), 0);
                }
            } else {
                 send(client_socket, "ERRORE: Formato DEL non valido. Usa:
    DEL <id>\n", strlen("ERRORE: Formato DEL non valido. Usa: DEL <id>\n"),
    0);
            }
        } else if (strcasecmp(comando, "UPD") == 0) {
            if (sscanf(buffer, "%*s %d %f %s", &id, &importo, causale) == 3
     && strlen(causale) > 0) {
                if (update_operazione(id, importo, causale)) {
                    send(client_socket, "OK: Movimento aggiornato.\n",
    strlen("OK: Movimento aggiornato.\n"), 0);
                } else {
                    send(client_socket, "ERRORE: ID non trovato per UPD.\n"
    , strlen("ERRORE: ID non trovato per UPD.\n"), 0);
                }
            } else {
                send(client_socket, "ERRORE: Formato UPD non valido. Usa:
    UPD <id> <importo> <causale>\n", strlen("ERRORE: Formato UPD non valido.
     Usa: UPD <id> <importo> <causale>\n"), 0);
            }
        } else if (strcasecmp(comando, "LIST") == 0) {
            list_operazione(client_socket);
        } else if (strcasecmp(comando, "EXIT") == 0) {
            send(client_socket, "Disconnessione...\n", strlen("
    Disconnessione...\n"), 0);
            break;
        } else {
            if(strlen(comando) > 0){ // Invia errore solo se e stato
    ricevuto un comando non vuoto
                 send(client_socket, "ERRORE: Comando non riconosciuto.\n",
     strlen("ERRORE: Comando non riconosciuto.\n"), 0);
            }
        }
    }

    if (read_size == 0) {
        printf("Client (socket %d) ha chiuso la connessione.\n",
    client_socket);
    } else if (read_size < 0) {
        perror("Errore recv nel thread");
    } else { // Uscito per EXIT
         printf("Client (socket %d) disconnesso tramite EXIT.\n",
    client_socket);
    }

    close(client_socket);
    printf("Thread per client (socket %d) terminato.\n", client_socket);
    return NULL;
```

```
}
```

Listing 4: Gestione della logica del client lato server in `gestioneClient.c`

4.5 gestioneLista.c

Questo file è il cuore della gestione dati. Contiene le definizioni delle variabili globali per la lista e il mutex, e tutte le funzioni per manipolare la lista dei movimenti in modo sicuro dal punto di vista della concorrenza.

```c
#include "header.h"

static Movimento *testa_lista = NULL; // ovviamente, all'inzio la testa della lista non conterra assolutamente nulla
static pthread_mutex_t lista_mutex;
static int id_operazione_succ = 1;

void init_lista() {
    if (pthread_mutex_init(&lista_mutex, NULL) != 0) {
        perror("Non sono riuscito ad inizializzare il mutex");
        exit(EXIT_FAILURE);
    }
}
```

Listing 5: Variabili globali e inizializzazione del mutex in `gestioneLista.c`

```c
void add_operazione(float importo, char *causale) {
    pthread_mutex_lock(&lista_mutex); // vado a bloccare per la mutua esclusione
    Movimento *nuovo_movimento = (Movimento *) malloc(sizeof(Movimento)); // alloco lo spazio necessario per il nuvoo nodo

    if (nuovo_movimento) { // se riesce ad allocare lo spazio
        nuovo_movimento->id_operazione = id_operazione_succ++; // incremento il valore di id
        nuovo_movimento->importo = importo;
        strncpy(nuovo_movimento->causale, causale, MAX_LEN_CAUSALE - 1); // copio il valore di causale in movimento.causale e non mi permette di andare oltre alla lunghezza di massima di len
        nuovo_movimento->causale[MAX_LEN_CAUSALE - 1] = '\0'; // aggiungo carattere di terminazione di stringa alla fine della causale
        nuovo_movimento->next = testa_lista; // puntera al nodo che era primo (prima dell'inserimento), questo diventera il secondo nodo, mentre quello che ho appena creato, diventera il primo (testa della lista)
        testa_lista = nuovo_movimento; // la testa della lista e il nodo appena creato
        printf("Aggiungo il movimento ID: %d", nuovo_movimento->id_operazione);
    }
    pthread_mutex_unlock(&lista_mutex); // sblocco il mutex dopo aver inserito l'operazione
}
```

Listing 6: Funzione `add_operazione` per l'inserimento in testa alla lista

```c
void list_operazione (int client_socket) {
```

```
    char buffer[MAX_LEN_MESSAGGIO *10] = {0}; // Lo utilizzo per contenere l'intero messaggio di risposta da mandare al client, che vado a riempire ogni volta
    char riga[MAX_LEN_MESSAGGIO]; // lo utilizzo per stampe "semplici", quando stampo una riga alla volta

    pthread_mutex_lock(&lista_mutex); // blocco

    strcpy(buffer, "Lista dei movimenti:\n");
    Movimento* corrente = testa_lista; // Prendo l'ultima operazione inserita

    if (corrente == NULL) {
        strcat(buffer, "Nessun movimento presente \n"); // concateno le due stringhe (buffer (vuoto) + "")
    }else {
        while (corrente != NULL) {
            sprintf(riga, "ID: %d, importo: %.2f, causale: %s\n", corrente->id_operazione, corrente->importo, corrente->causale);
            strcat(buffer, riga);
            corrente = corrente->next; // vado a prendere il nodo successivo fino a quando non arrivo a NULL
        }
    }

    pthread_mutex_unlock(&lista_mutex); // sblocco il mutex dopo il calcolo di tutte le operazioni effettuate e salvate bel buffer
    strcat(buffer, "Fine lista movimenti\n");
    send(client_socket, buffer, strlen(buffer), 0); // invio al client tutto il buffer
}
```

Listing 7: Funzione `list_operazione` per l'invio della lista al client

```
int delete_operazione(int id_da_cancellare) {
    pthread_mutex_lock(&lista_mutex);

    Movimento* corrente = testa_lista; // imposto l'ultima operazione come testa della lista
    Movimento* precedente = NULL;
    int trovato = 0;

    while (corrente != NULL) {
        if (corrente->id_operazione == id_da_cancellare) {
            trovato = 1;
            break; // e inutile continuare il while
        }
        precedente = corrente; // controllo l'altro nodo
        corrente = corrente->next;
    }

    if (trovato) {
        if (precedente == NULL) {
            testa_lista = corrente->next; // se il nodo da cancellare e il primo, non c'e un precedente e la testa diventa il nodo successivo
        }
        else { // salto il nodo corrente e collego il precedente al successivo
```

```
            precedente->next = corrente->next;
        }
        free(corrente); // libero la memoria occupata dal nodo eliminato
        printf("Ho cancellato il movimento ID %d\n", id_da_cancellare);
    }

    pthread_mutex_unlock(&lista_mutex);
    return trovato;
}
```

Listing 8: Funzione `delete_operazione` per la cancellazione di un nodo

```
int update_operazione(int id_da_aggiornare, float nuovo_importo, char*
    nuova_causale) {
    pthread_mutex_lock(&lista_mutex);
    Movimento* corrente = testa_lista; // prendo l'ultima operazione
    int trovato = 0;

    while (corrente != NULL) {
        if (corrente->id_operazione == id_da_aggiornare) {
            corrente->importo = nuovo_importo;
            strncpy(corrente->causale, nuova_causale, MAX_LEN_CAUSALE - 1);
            corrente->causale[MAX_LEN_CAUSALE - 1] = '\0';
            trovato = 1;
            printf("Aggiornato movimento ID %d\n", id_da_aggiornare);
            break; // inutile continuare il ciclo
        }
        corrente = corrente->next;
    }

    pthread_mutex_unlock(&lista_mutex);
    return trovato;
}
```

Listing 9: Funzione `update_operazione` per l'aggiornamento di un nodo

```
void pulizia() {
    pthread_mutex_lock(&lista_mutex); // blocco il mutex
    Movimento* corrente = testa_lista; // prendo l'ultima operazione
    salvata
    Movimento* temp;

    while (corrente != NULL) { // effettuo lo swap per prendere il valore
    successivo ed eliminare quello attuale e cosi via fino ad arrivare a
    NULL
        temp = corrente;
        corrente = corrente->next;
        free(temp);
    }

    testa_lista = NULL; // imposto la testa a NULL
    pthread_mutex_unlock(&lista_mutex); // sblocco il mutex
    pthread_mutex_destroy(&lista_mutex); // rimuovo definitivamente il
    mutex
    printf("Ho eliminato la lista");
}
```

Listing 10: Funzione `pulizia` per la deallocazione della lista e del mutex

Il codice è stato compilato con successo utilizzando un `Makefile` (non mostrato, ma la cui creazione era parte dell'esercizio), che automatizza la compilazione dei singoli file `.c` e il loro collegamento per produrre due eseguibili: `server` e `client`.

Per testare il sistema, il server viene eseguito in un terminale:

```
$ ./server
Server in ascolto sulla porta 8888...
```

Successivamente, è possibile avviare una o più istanze del client in altri terminali. Ogni client si connette e riceve il prompt dei comandi. Il server nel frattempo logga ogni nuova connessione.

Esempio di sessione utente: Un client si connette e il server lo notifica:

```
Nuova connessione accettata (socket 4)
```

Il client riceve le istruzioni e può iniziare a operare:

```
$ ./client
Connesso al server 127.0.0.1 sulla porta 8888
Comandi: ADD <imp> <caus>, DEL <id>, UPD <id> <imp> <caus>, LIST, EXIT
> ADD 150.00 Stipendio
Server: OK: Movimento aggiunto.
> ADD 25.50 Spesa_Supermercato
Server: OK: Movimento aggiunto.
> LIST
Server: Lista dei movimenti:
ID: 2, importo: 25.50, causale: Spesa_Supermercato
ID: 1, importo: 150.00, causale: Stipendio
Fine lista movimenti
> DEL 1
Server: OK: Movimento cancellato.
> LIST
Server: Lista dei movimenti:
ID: 2, importo: 25.50, causale: Spesa_Supermercato
Fine lista movimenti
> EXIT
Disconnessione...
```

Durante queste operazioni, la console del server mostra i log interni, come l'aggiunta o la cancellazione di un movimento, e infine la disconnessione del client. Avviando più client contemporaneamente, si può verificare che le operazioni vengono accodate correttamente grazie alla protezione del mutex, senza che i dati vengano corrotti. Il sistema si è dimostrato stabile e rispondente ai requisiti del problema.

5 Conclusioni

Il progetto ha permesso di realizzare con successo un server TCP multithread per la gestione di transazioni bancarie, rispettando tutti i requisiti specificati. L'architettura modulare ha semplificato lo sviluppo e garantito una chiara separazione delle responsabilità tra i componenti del sistema. L'adozione del modello *thread-per-connection* e l'uso di mutex per la sincronizzazione si sono rivelati efficaci nel garantire la gestione sicura e concorrente delle operazioni sulla risorsa condivisa.

5.1 Sviluppi Futuri

Sebbene il sistema sia funzionale, esistono diversi ambiti di miglioramento per un'applicazione di livello produttivo:

- **Persistenza dei Dati**: Attualmente la lista dei movimenti risiede solo in memoria e viene persa alla chiusura del server. Si potrebbe implementare un meccanismo per salvare i dati su file o su un database per renderli persistenti.
- **Gestione di più Utenti**: Il sistema è stato progettato per un singolo correntista. Potrebbe essere esteso per gestire più conti correnti, richiedendo un meccanismo di autenticazione per i client.
- **Utilizzo di un Thread Pool**: Invece di creare un nuovo thread per ogni connessione, che può essere inefficiente in caso di un numero molto elevato di client, si potrebbe implementare un *thread pool* per riutilizzare un numero fisso di thread, migliorando le prestazioni e l'utilizzo delle risorse.
- **Handling Avanzato**: Migliorare la gestione degli errori, ad esempio nel parsing di input malformati da parte del client, per rendere il server ancora più robusto.