Server TCP Multithread con operazioni CRUD

Gestione concorrente di movimenti bancari

Gabriel Piparo — 555736

Hands-On 14 — 2025

Indice

Scenario del problema

Sistema bancario per movimenti:

- Ogni movimento contiene: **ID, importo, causale**
- Accesso concorrente da:
  - Terminale bancario
  - App mobile
  - Sportello
  - E-banking
- Operazioni supportate: **ADD, DELETE, UPDATE, LIST**

Obiettivo: sviluppare un **Server TCP multithread** che gestisca più client contemporaneamente.

Motivazione

- Necessità di un accesso **simultaneo e sicuro** alla struttura dati
- Prevenzione inconsistenze tramite:
  - **mutex** per accesso esclusivo
  - **semaforo** per limitare i thread attivi (MAX_NUM)
- Architettura modulare:
  - `server_tcp.c`
  - `client_tcp.c`
  - `operazioni.c/h`

Lista concatenata dei movimenti

```
typedef struct movimento {
    int id;
    float importo;
    char* causale;
    struct movimento *next;
} movimento;

extern movimento *head;
```

- Ogni nodo rappresenta un movimento bancario
- head: testa della lista globale condivisa
- Accesso protetto tramite mutex

Funzioni principali

CRUD sulla lista:

- **inserisci_movimento()**
- **ricerca_movimento()**
- **modifica()**
- **elimina()**
- **stampa_movimenti()**

Tutte le modifiche sono protette da:

- `pthread_mutex_lock()`
- `pthread_mutex_unlock()`

Esempio — Inserimento

```
pthread_mutex_lock(&lock);

movimento *m = malloc(sizeof(movimento));
m->id = id;
m->importo = importo;
m->causale = strdup(causale);
m->next = head;
head = m;

pthread_mutex_unlock(&lock);
```

Accesso esclusivo alla struttura dati.

Architettura del server

- Socket TCP:
  - `socket()`
  - `bind()`
  - `listen()`
- Ogni client $\rightarrow$ **thread dedicato**
- Semaforo $\rightarrow$ max 10 connessioni
- Mutex $\rightarrow$ protezione struttura dati

Accettazione connessioni

```
sem_wait(&sem);

newsockfd = accept(sockfd, ...);

pthread_create(&thread_id, NULL,
               thread_function, &newsockfd);
```

- Il semaforo limita i thread attivi
- Ogni thread comunica col client

Thread Function

```
while (1) {
    n = read(sock, buffer, 1024);

    if (strcmp(buffer,"ADD\n")==0) { ... }
    if (strcmp(buffer,"UPDATE\n")==0) { ... }
    if (strcmp(buffer,"DELETE\n")==0) { ... }
    if (strcmp(buffer,"LIST\n")==0) { ... }
}
```

Il thread gestisce i comandi finché il client non invia `EXIT`.

Funzionamento del client

- Connessione TCP al server
- Menu operazioni:
  - ADD / DELETE / UPDATE / LIST / EXIT
- Comunicazione tramite `send()` e `read()`
- Loop finché non viene inviato `EXIT`

Invio dei comandi dal client

Il client:

- legge un comando,
- lo invia al server,
- attende la risposta,
- per richieste come ADD/UPDATE/DELETE invia poi i dati richiesti.

```
fgets(buffer, sizeof(buffer), stdin);    // Leggo comando
send(sockfd, buffer, strlen(buffer), 0);// Invio

n = read(sockfd, buffer, BUFFER_SIZE-1);
buffer[n] = '\0';
printf("%s", buffer);                    // Output server
```

Test effettuati (multi-client)

- Più client collegati contemporaneamente
- Lista condivisa sempre coerente (mutex)
- Max 10 sessioni attive (semaforo)
- Nessuna perdita di dati
- Risposte immediate e sincronizzate tra i client

Esempio sessione — Client 1

Client 1:

- ADD:
  - ID 1 — 100 euro — ricarica
  - ID 2 — 50 euro — prova
- LIST mostra:
  - ID 2 — 50.00 — prova
  - ID 1 — 100.00 — ricarica
- EXIT $\rightarrow$ disconnessione

Esempio sessione — Client 2

Client 2:

- LIST vede gli stessi dati del client 1
- DELETE: rimozione ID 2
- ADD: inserimento nuovo movimento
- LIST $\rightarrow$ lista aggiornata in tempo reale

Log del server

Il server mostra:

- Connessioni client
- Comandi ricevuti
- Output operazioni:
  - “OPERAZIONE ADD EFFETTUATA”
  - “OPERAZIONE LIST EFFETTUATA”
  - “NESSUN NODO ELIMINATO”
- Disconnessioni

Conclusioni

- Server TCP multithread stabile e funzionante
- Accesso concorrente corretto grazie al mutex
- Gestione limitata dei thread tramite semaforo
- CRUD pienamente operative
- Architettura modulare, espandibile e pulita