Hands-On 14: Midterm

Server TCP Multithread per Gestione Movimenti Bancari

Sandi Russo

30 maggio 2025

Indice dei Contenuti

1. Introduzione
2. Definizione del Problema
3. Metodologia
4. Presentazione dei Risultati
5. Conclusioni

Introduzione: Panoramica del Progetto

- **Obiettivo:** Realizzare un sistema software per la gestione di movimenti su un conto corrente bancario.
- **Focus:** Capacità di gestire accessi concorrenti da più client, simulando diversi canali di interazione (ATM, App, E-banking).
- **Tecnologie:**
  - Linguaggio: C
  - Protocollo: TCP/IP per comunicazione client-server
  - Concorrenza: Libreria Pthreads
- **Architettura:** Client-Server multithread.

Definizione del Problema: Requisiti Chiave

- **Sistema di Tracciamento:**
  - Movimenti identificati da: ID operazione, Importo, Causale.
  - Relativo ad un unico utente/correntista.

- **Accesso Concorrente:**
  - Il server deve gestire fino a `MAX_NUM_CLIENTS` (10) thread client simultaneamente.
  - Simulazione di accesso da: Terminale Bancario, App, Sportello, E-banking.

- **Operazioni Richieste:**
  - `ADD`: Aggiungere un nuovo movimento.
  - `DELETE`: Cancellare un movimento (per ID).
  - `UPDATE`: Modificare un movimento (per ID).
  - `LIST`: Visualizzare tutti i movimenti.
- **Altri Requisiti:** Programma modulare, Makefile.

Metodologia: Architettura Client-Server TCP

Server TCP Multithread:

- **Gestore centrale:** Riceve richieste, elabora logica, mantiene consistenza dati.
- **Dati condivisi:** Lista movimenti.
- **Concorrenza:** Thread dedicati per client (`pthread_create`), Mutex (`pthread_mutex_t`) per sincronizzazione.
- **Interfaccia:** Riga di comando.
- **Connessione:** `socket()`, `connect()`.
- **Comunicazione:** Invia comandi (`send()`), riceve risposte (`recv()`).
- **Terminazione:** `close()` (con comando `EXIT`).

Protocollo TCP: Scelto per comunicazioni affidabili.

Metodologia: Struttura Modulare del Codice

- `header.h`: Definizioni globali e prototipi.

```
1 // header.h - Snippet
2 typedef struct Movimento { /* ... */ } Movimento;
3 void* gestisci_client(void* arg);
4 void add_operazione(float importo, char* causale);
5 // ... altri prototipi ...
6
```

- `server.c`: Avvio server, accettazione connessioni, creazione thread.
- `client.c`: Logica del client per interazione utente e comunicazione.
- `gestioneClient.c`: Funzione `gestisci_client` eseguita da ogni thread server.
- `gestioneLista.c`: Gestione della lista movimenti e sincronizzazione.

Metodologia: Gestione Concorrenza (Server)

Modello "Thread-per-Client" in `server.c`:

- Il server principale attende connessioni sulla `accept()`.
- Per ogni nuova connessione, crea un thread dedicato.

```
1 client_socket = accept(server_socket, NULL, NULL);
2 // ... error check ...
3 int* new_sock = malloc(sizeof(int));
4 *new_sock = client_socket;
5 pthread_create(&thread_id, NULL, gestisci_client, (void*)new_sock);
6 pthread_detach(thread_id);
7
```

Listing 1: Creazione Thread in `server.c`

Metodologia: Struttura Dati e Sincronizzazione

Dati Condivisi: Lista concatenata (`Movimento *testa_lista`).
Problema: Rischio di *race conditions*. **Soluzione: Mutua Esclusione**

- `pthread_mutex_t lista_mutex`.
- Inizializzato in `init_lista()`.
- Protegge le sezioni critiche.

```
1 void add_operazione(float importo, char *causale) {
2     pthread_mutex_lock(&lista_mutex); // Inizio Sezione Critica
3     // ... logica di aggiunta alla lista ...
4     testa_lista = nuovo_movimento;
5     pthread_mutex_unlock(&lista_mutex); // Fine Sezione Critica
6 }
7
```

Listing 2: Uso del Mutex in `gestioneLista.c` (es. aggiunta)

Metodologia: gestisci_client.c

Ogni thread esegue gestisci_client:

- Riceve messaggi (recv), interpreta (sscanf).
- Invoca funzioni di gestioneLista.c.
- Invia risposte (send).

```
1 sscanf(buffer, "%19s", comando);
2 if (strcasecmp(comando, "ADD") == 0) {
3     if (sscanf(buffer, "%*s %f %s", &importo, causale) == 2) {
4         add_operazione(importo, causale);
5         send(client_socket, "OK: Movimento aggiunto.\n", ...);
6     } // ... else gestione errore ...
7 } // ... else if per altri comandi ...
8
```

Listing 3: Parsing Comandi in gestisci_client.c

Risultati: Compilazione ed Esecuzione

- **Compilazione:** `Makefile -> server, client`.
- **Esecuzione:** Avviare `./server`, poi uno o più `./client`.

Console del Server (estratto):

```
Server in ascolto sulla porta 8888...
Nuova connessione accettata (socket 4)
Aggiungo il movimento ID: 1
Client (socket 4) disconnesso tramite EXIT.
```

Risultati: Esempio Sessione Client

```
$ ./client
Connesso al server 127.0.0.1 sulla porta 8888
Comandi: ADD <imp> <caus>, DEL <id>, ...
> ADD 250.75 Stipendio
Server: OK: Movimento aggiunto.
> LIST
Server: Lista dei movimenti:
ID: 1, importo: 250.75, causale: Stipendio
Fine lista movimenti
> EXIT
Disconnessione...
```

- Sistema reattivo e corretto.
- Integrità dati mantenuta.

Conclusioni e Sviluppi Futuri

Riepilogo:

- Realizzato server TCP multithread funzionale.
- Rispettati requisiti di concorrenza e operazioni CRUD.
- Architettura modulare, sincronizzazione con Mutex.
- Persistenza Dati (file/database).
- Gestione Multi-Utente + Autenticazione.
- Thread Pool per ottimizzare risorse.