Hands-On 14

Giacomo Lanza [542484]
MIFT, Scienze Informatiche, III Anno

14 Giugno 2025

Indice

Capitolo 1

Introduzione al problema

In un contesto bancario moderno, è fondamentale offrire accesso simultaneo e sicuro ai dati dei correntisti attraverso molteplici punti d'ingresso: sportello fisico, app mobile, e-banking e terminale bancario. Il sistema deve garantire la consistenza dei dati anche in presenza di accessi concorrenti, evitando corruzioni e garantendo l'affidabilità delle operazioni.

Questo hands-on si propone di progettare e realizzare un'applicazione modulare Client/Server basata su TCP che permetta ad un singolo utente/correntista di registrare e gestire i propri movimenti bancari attraverso operazioni di `ADD`, `DELETE`, `UPDATE` e `LIST`. Il server deve essere multithread e supportare fino a 10 connessioni concorrenti.

Capitolo 2

Stato dell'arte

L'architettura Client/Server basata su socket TCP è un paradigma consolidato per la realizzazione di applicazioni di rete affidabili. Nella programmazione concorrente in C, l'uso dei thread (pthreads) è una pratica diffusa per gestire più client simultaneamente. Tuttavia, l'accesso concorrente a risorse condivise richiede l'uso di meccanismi di sincronizzazione come i mutex per evitare condizioni di race.

La libreria `pthread.h` fornisce un set di strumenti per la creazione e la gestione di thread. La sincronizzazione tramite `pthread_mutex_lock` e `pthread_mutex_unlock` è indispensabile quando più thread accedono e modificano strutture dati comuni.

Capitolo 3

Metodologia usata

3.1 Struttura dati condivisa

Le operazioni bancarie sono salvate in un array globale:

```
typedef struct {
    int id;
    float importo;
    char causale[100];
} Operazione;
```

Listing 3.1: Struttura dati Operazione

Questo array, definito in `common.h`, è condiviso tra tutti i thread. Per evitare accessi concorrenti inconsistenti, è protetto da un `pthread_mutex_t lock`.

3.2 Architettura client-server

Il server utilizza la libreria `pthread` per creare un nuovo thread per ogni client connesso. I comandi accettati dal server sono:

- ADD <id> <importo> <causale>
- DELETE <id>
- UPDATE <id> <importo> <causale>
- LIST

3.2.1 Client TCP

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

#define SERVER_IP "127.0.0.1"
#define SERVER_PORT 8080

void print_help() {
    printf("Connesso al server.\n");
```

```
12      printf("Comandi disponibili:\n");
13      printf("  ADD    <id> <importo> <causale>   - Aggiunge una
    nuova operazione\n");
14      printf("  DELETE <id>                       - Elimina l'
    operazione con id specificato\n");
15      printf("  UPDATE <id> <importo> <causale>   - Aggiorna l'
    operazione con id specificato\n");
16      printf("  LIST                              - Visualizza
    tutte le operazioni\n");
17      printf("  QUIT                              - Chiude la
    connessione\n");
18  }
19
20  int main() {
21      int sock;
22      struct sockaddr_in server_addr;
23      char buffer[1024];
24      char recvbuf[2048];
25      int n;
26
27      if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
28          perror("socket");
29          exit(EXIT_FAILURE);
30      }
31      server_addr.sin_family = AF_INET;
32      server_addr.sin_port = htons(SERVER_PORT);
33
34      if (inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr) <=
    0) {
35          perror("inet_pton");
36          close(sock);
37          exit(EXIT_FAILURE);
38      }
39      if (connect(sock, (struct sockaddr *)&server_addr, sizeof(
    server_addr)) < 0) {
40          perror("connect");
41          close(sock);
42          exit(EXIT_FAILURE);
43      }
44
45      print_help();
46
47      while (1) {
48          printf("> ");
49          if (!fgets(buffer, sizeof(buffer), stdin))
50              break;
51          if (strncmp(buffer, "QUIT", 4) == 0)
52              break;
53          send(sock, buffer, strlen(buffer), 0);
54          n = recv(sock, recvbuf, sizeof(recvbuf)-1, 0);
55          if (n > 0) {
```

```
            recvbuf[n] = '\0';
            printf("%s", recvbuf);
        }
    }
    close(sock);
    return 0;
}
```

Listing 3.2: client.c

3.2.2 Header comune

```
#ifndef COMMON_H
#define COMMON_H

#include <pthread.h>

#define MAX_OP 100
#define PORT 8080
#define MAX_THREADS 10

typedef struct {
    int id;
    float importo;
    char causale[100];
} Operazione;

extern Operazione operazioni[MAX_OP];
extern int num_operazioni;
extern pthread_mutex_t lock;

// Funzioni operazioni
void add_operazione(int client_sock, char *args);
void delete_operazione(int client_sock, int id);
void update_operazione(int client_sock, char *args);
void list_operazioni(int client_sock);

#endif
```

Listing 3.3: common.h

3.2.3 Operazioni bancarie

```
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/socket.h>
#include "common.h"

Operazione operazioni[MAX_OP];
int num_operazioni = 0;
extern pthread_mutex_t lock;
```

```
11
12 void add_operazione(int client_sock, char *args) {
13     pthread_mutex_lock(&lock);
14     if (num_operazioni >= MAX_OP) {
15         send(client_sock, "[SERVER] Archivio pieno\n", 25, 0);
16     } else {
17         Operazione op;
18         int ret = sscanf(args, "%d %f %99s", &op.id, &op.importo,
    op.causale);
19         if (ret != 3) {
20             send(client_sock, "[SERVER] Formato ADD non valido\n"
   , 32, 0);
21         } else {
22             operazioni[num_operazioni++] = op;
23             send(client_sock, "[SERVER] Operazione aggiunta\n",
   30, 0);
24         }
25     }
26     pthread_mutex_unlock(&lock);
27 }
28
29 void delete_operazione(int client_sock, int id) {
30     pthread_mutex_lock(&lock);
31     int found = 0;
32     for (int i = 0; i < num_operazioni; i++) {
33         if (operazioni[i].id == id) {
34             operazioni[i] = operazioni[--num_operazioni];
35             found = 1;
36             break;
37         }
38     }
39     if (found)
40         send(client_sock, "[SERVER] Operazione rimossa\n",30,0);
41     else
42         send(client_sock, "[SERVER] ID non trovato\n", 25, 0);
43     pthread_mutex_unlock(&lock);
44 }
45
46 void update_operazione(int client_sock, char *args) {
47     pthread_mutex_lock(&lock);
48     int id;
49     float importo;
50     char causale[100];
51     int ret = sscanf(args, "%d %f %99s", &id, &importo, causale);
52     int found = 0;
53     if (ret != 3) {
54         send(client_sock, "[SERVER] Formato UPDATE non valido\n",
    34, 0);
55     } else {
56         for (int i = 0; i < num_operazioni; i++) {
57             if (operazioni[i].id == id) {
```

```
58                  operazioni[i].importo = importo;
59                  strcpy(operazioni[i].causale, causale);
60                  found = 1;
61                  break;
62              }
63          }
64          if (found)
65              send(client_sock, "[SERVER] Operazione aggiornata\n",
    33, 0);
66          else
67              send(client_sock, "[SERVER] ID non trovato\n",25,0);
68      }
69      pthread_mutex_unlock(&lock);
70  }
71
72  void list_operazioni(int client_sock) {
73      pthread_mutex_lock(&lock);
74      char response[2048] = "[SERVER] Operazioni:\n";
75      char temp[150];
76      for (int i = 0; i < num_operazioni; i++) {
77          snprintf(temp, sizeof(temp), "ID: %d, Importo: %.2f,
   Causale: %s\n",
78                   operazioni[i].id, operazioni[i].importo,
   operazioni[i].causale);
79          if (strlen(response)+strlen(temp)<sizeof(response)-1)
80              strcat(response, temp);
81          else
82              break;
83      }
84      send(client_sock, response, strlen(response), 0);
85      pthread_mutex_unlock(&lock);
86  }
```

Listing 3.4: operazioni.c

3.2.4 Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -pthread
3
4  all: server client
5
6  server: server.c operazioni.c common.h
7    $(CC) $(CFLAGS) server.c operazioni.c -o server
8
9  client: client.c
10   $(CC) $(CFLAGS) client.c -o client
11
12 clean:
13   rm -f server client *.o
```

Listing 3.5: Makefile

Capitolo 4

Risultati sperimentali

I test sono stati effettuati simulando l'accesso concorrente da più client tramite terminali multipli. Il sistema si è dimostrato stabile e consistente: ogni operazione è stata eseguita correttamente, senza corruzione dei dati, grazie all'uso corretto del mutex.

4.1 Esempio di output

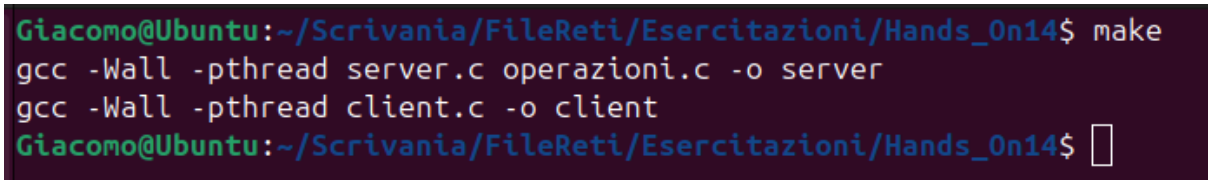

Figura 4.1

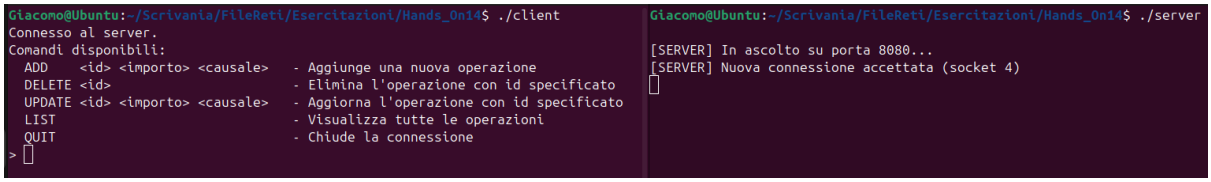

Figura 4.2

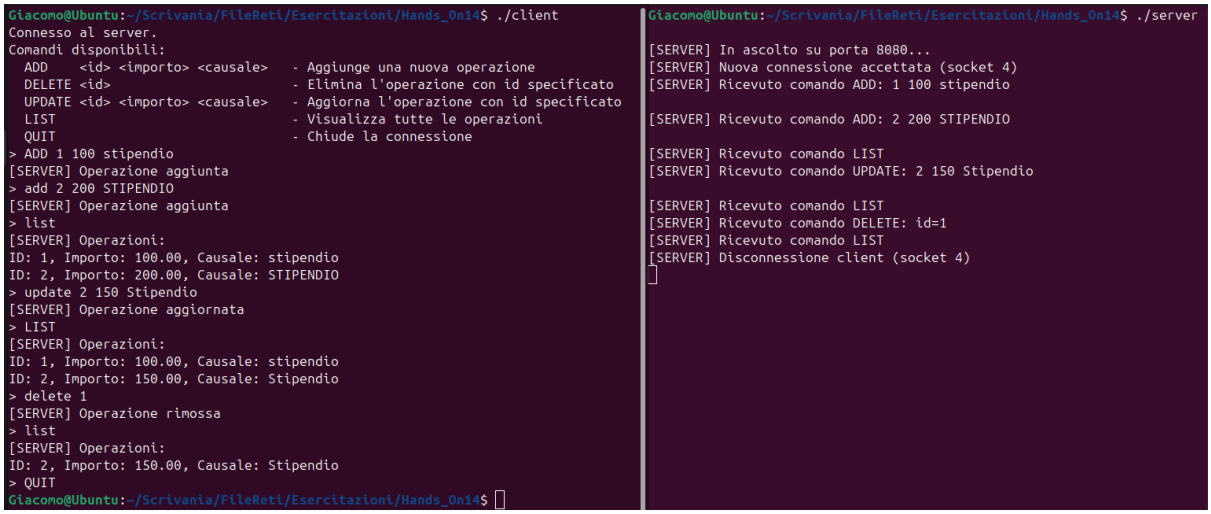
Giacomo@Ubuntu:~/Scrivania/FileReti/Esercitazioni/Hands_On14$ ./client
Connesso al server.
Comandi disponibili:
ADD <id> <importo> <causale> - Aggiunge una nuova operazione
DELETE <id> - Elimina l'operazione con id specificato
UPDATE <id> <importo> <causale> - Aggiorna l'operazione con id specificato
LIST - Visualizza tutte le operazioni
QUIT - Chiude la connessione
> ADD 1 100 stipendio
[SERVER] Operazione aggiunta
> add 2 200 STIPENDIO
[SERVER] Operazione aggiunta
> list
[SERVER] Operazioni:
ID: 1, Importo: 100.00, Causale: stipendio
ID: 2, Importo: 200.00, Causale: STIPENDIO
> update 2 150 Stipendio
[SERVER] Operazione aggiornata
> LIST
[SERVER] Operazioni:
ID: 1, Importo: 100.00, Causale: stipendio
ID: 2, Importo: 150.00, Causale: Stipendio
> delete 1
[SERVER] Operazione rimossa
> list
[SERVER] Operazioni:
ID: 2, Importo: 150.00, Causale: Stipendio
> QUIT
Giacomo@Ubuntu:~/Scrivania/FileReti/Esercitazioni/Hands_On14$
Giacomo@Ubuntu:~/Scrivania/FileReti/Esercitazioni/Hands_On14$ ./server
[SERVER] In ascolto su porta 8080...
[SERVER] Nuova connessione accettata (socket 4)
[SERVER] Ricevuto comando ADD: 1 100 stipendio
[SERVER] Ricevuto comando ADD: 2 200 STIPENDIO
[SERVER] Ricevuto comando LIST
[SERVER] Ricevuto comando UPDATE: 2 150 Stipendio
[SERVER] Ricevuto comando LIST
[SERVER] Ricevuto comando DELETE: id=1
[SERVER] Ricevuto comando LIST
[SERVER] Disconnessione client (socket 4)

Figura 4.3

Capitolo 5

Conclusione e sviluppi futuri

Il progetto ha dimostrato l'importanza della sincronizzazione nell'accesso concorrente a strutture dati condivise in ambiente multithread. È stato implementato con successo un server TCP robusto che gestisce richieste da client multipli in parallelo, con operazioni di gestione delle transazioni bancarie.

5.1 Sviluppi futuri

Tra gli sviluppi futuri si possono considerare:

- Persistenza dei dati su file o database.
- Autenticazione dell'utente.
- Estensione a più correntisti.
- Gestione di timeout e disconnessioni.
- Interfaccia grafica per il client.