Hands-On 14: Server Bancario Multithread in C

Giacomo Lanza (542484)

MIFT - Università degli Studi di Messina
Scienze Informatiche, III Anno

14 Giugno 2025

Sommario

Contesto Bancario Moderno

- Accesso simultaneo ai dati tramite:
  - sportelli fisici
  - app mobile
  - portali e-banking
  - terminali bancari
- Requisiti fondamentali:
  - Sicurezza degli accessi
  - Consistenza dei dati
  - Gestione efficace della concorrenza

Obiettivi dell'Hands-On

Progettazione e Sviluppo

Realizzare un'applicazione modulare **Client/Server TCP** che consenta ad un singolo utente di gestire i propri movimenti bancari.

Funzionalità supportate:

- `ADD` – Inserimento di una nuova operazione
- `DELETE` – Eliminazione di un'operazione esistente
- `UPDATE` – Modifica dei dati di un'operazione
- `LIST` – Visualizzazione di tutte le operazioni

Caratteristiche tecniche:

- Server multithread
- Supporto per massimo 10 client concorrenti

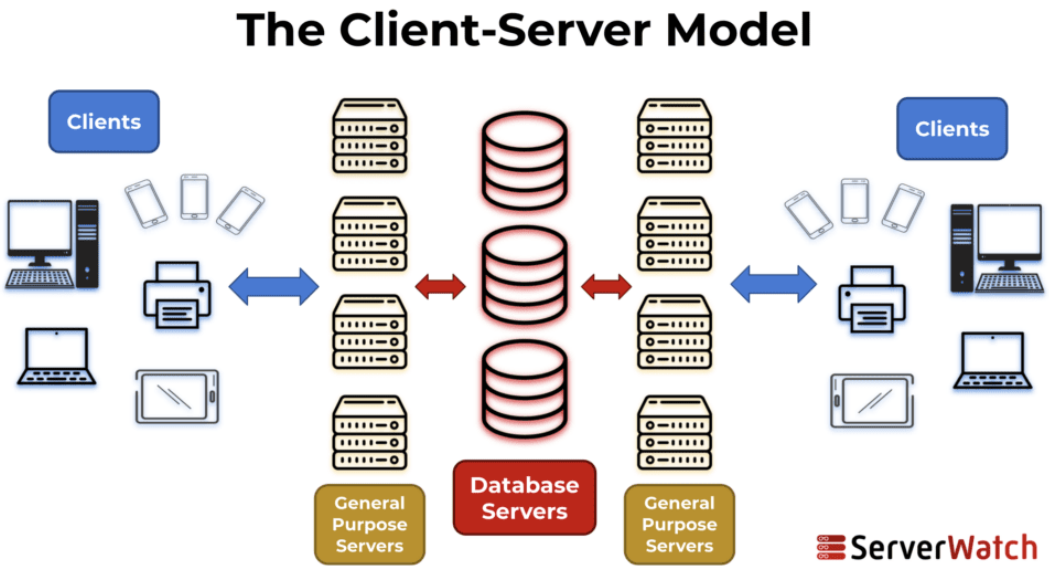


Figura: Descrizione sintetica

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 `pthread` 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 gestione dei thread. La sincronizzazione tramite `pthread_mutex_lock` e `pthread_mutex_unlock` è indispensabile quando più thread accedono e modificano strutture dati comuni.

Struttura dati condivisa

Le operazioni bancarie sono salvate in un array globale:

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

Listing 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`.

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`

Client TCP – Descrizione

Il file `client.c` implementa un semplice client TCP che:

- Si connette al server tramite socket.
- Legge i comandi da tastiera (es. `ADD`, `LIST`, `DELETE`, ecc.).
- Invia i comandi al server.
- Riceve ed eventualmente stampa la risposta del server.

Ogni comando corrisponde a un'operazione bancaria gestita dal server multithread.

Client TCP (1/3)

```c
#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");
    printf("Comandi disponibili:\n");
    printf("  ADD    <id> <importo> <causale>    - Aggiunge una nuova operazione\n");
    printf("  DELETE <id>                        - Elimina l'operazione con id specificato\n");
    printf("  UPDATE <id> <importo> <causale>    - Aggiorna l'operazione con id specificato\n");
    printf("  LIST                               - Visualizza tutte le operazioni\n");
    printf("  QUIT                               - Chiude la connessione\n");
}
```

Client TCP (2/3)

```c
int main() {
    int sock;
    struct sockaddr_in server_addr;
    char buffer[1024];
    char recvbuf[2048];
    int n;

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVER_PORT);

    if (inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr) <=
        0) {
        perror("inet_pton");
        close(sock);
        exit(EXIT_FAILURE);
    }
    if (connect(sock, (struct sockaddr *)&server_addr, sizeof(
        server_addr)) < 0) {
```

Client TCP (3/3)

```
        perror("connect");
        close(sock);
        exit(EXIT_FAILURE);
    }

    print_help();

    while (1) {
        printf("> ");
        if (!fgets(buffer, sizeof(buffer), stdin))
            break;
        if (strncmp(buffer, "QUIT", 4) == 0)
            break;
        send(sock, buffer, strlen(buffer), 0);
        n = recv(sock, recvbuf, sizeof(recvbuf)-1, 0);
        if (n > 0) {
            recvbuf[n] = '\0';
            printf("%s", recvbuf);
        }
    }
    close(sock);
    return 0;
}
```

Header comune – Descrizione

Il file `common.h` contiene:

- La definizione della struttura `Operazione`, che rappresenta una transazione bancaria.
- Le dichiarazioni di costanti, mutex e variabili condivise.
- Funzioni dichiarate che verranno implementate in `operazioni.c`.

Serve come file di interfaccia condiviso tra server, client e moduli interni.

Header comune (1/2)

```c
#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;
```

Header comune (2/2)

```
1 // Funzioni operazioni
2 void add_operazione(int client_sock, char *args);
3 void delete_operazione(int client_sock, int id);
4 void update_operazione(int client_sock, char *args);
5 void list_operazioni(int client_sock);
6
7 #endif
```

Operazioni bancarie – Descrizione

Il file `operazioni.c` contiene l'implementazione delle funzioni bancarie:

- `addOperazione`: aggiunge una nuova operazione.
- `deleteOperazione`: elimina una operazione esistente.
- `updateOperazione`: aggiorna un'operazione esistente.
- `listOperazioni`: elenca tutte le operazioni salvate.

Queste funzioni operano sull'array globale protetto da `mutex`.

Operazioni bancarie (1/6)

```c
#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;

void add_operazione(int client_sock, char *args) {
    pthread_mutex_lock(&lock);
    if (num_operazioni >= MAX_OP) {
        send(client_sock, "[SERVER] Archivio pieno\n", 25, 0);
```

Operazioni bancarie (2/)

```
    } else {
        Operazione op;
        int ret = sscanf(args, "%d %f %99s", &op.id, &op.importo
            , op.causale);
        if (ret != 3) {
            send(client_sock, "[SERVER] Formato ADD non valido\n
                ", 32, 0);
        } else {
            operazioni[num_operazioni++] = op;
            send(client_sock, "[SERVER] Operazione aggiunta\n",
                30, 0);
        }
    }
    pthread_mutex_unlock(&lock);
}
```

Operazioni bancarie (3/6)

```c
void delete_operazione(int client_sock, int id) {
    pthread_mutex_lock(&lock);
    int found = 0;
    for (int i = 0; i < num_operazioni; i++) {
        if (operazioni[i].id == id) {
            operazioni[i] = operazioni[--num_operazioni];
            found = 1;
            break;
        }
    }
    if (found)
        send(client_sock, "[SERVER] Operazione rimossa\n",30,0);
    else
        send(client_sock, "[SERVER] ID non trovato\n", 25, 0);
    pthread_mutex_unlock(&lock);
}
```

Operazioni bancarie (4/6)

```c
void update_operazione(int client_sock, char *args) {
    pthread_mutex_lock(&lock);
    int id;
    float importo;
    char causale[100];
    int ret = sscanf(args, "%d %f %99s", &id, &importo, causale)
        ;
    int found = 0;
    if (ret != 3) {
        send(client_sock, "[SERVER] Formato UPDATE non valido\n"
            , 34, 0);
    } else {
        for (int i = 0; i < num_operazioni; i++) {
            if (operazioni[i].id == id) {
                operazioni[i].importo = importo;
                strcpy(operazioni[i].causale, causale);
                found = 1;
                break;
            }
        }
```

Operazioni bancarie (5/6)

```
1           if (found)
2               send(client_sock, "[SERVER] Operazione aggiornata\n"
                    , 33, 0);
3           else
4               send(client_sock, "[SERVER] ID non trovato\n",25,0);
5       }
6       pthread_mutex_unlock(&lock);
7   }
```

Operazioni bancarie (6/6)

```
1  void list_operazioni(int client_sock) {
2      pthread_mutex_lock(&lock);
3      char response[2048] = "[SERVER] Operazioni:\n";
4      char temp[150];
5      for (int i = 0; i < num_operazioni; i++) {
6          snprintf(temp, sizeof(temp), "ID: %d, Importo: %.2f,
               Causale: %s\n",
7                   operazioni[i].id, operazioni[i].importo,
                        operazioni[i].causale);
8          if (strlen(response)+strlen(temp)<sizeof(response)-1)
9              strcat(response, temp);
10         else
11             break;
12     }
13     send(client_sock, response, strlen(response), 0);
14     pthread_mutex_unlock(&lock);
15 }
```

Makefile – Descrizione

Il file `Makefile` automatizza il processo di compilazione del progetto.

- Definisce le regole per compilare i file sorgente (`.c`) in eseguibili.
- Gestisce le dipendenze tra file, ad esempio tra `operazioni.c` e `common.h`.
- Include target come:
  - `all` – per costruire tutti gli eseguibili (client e server).
  - `clean` – per rimuovere file oggetto e binari.
- Facilita la ricompilazione evitando compilazioni inutili.

Makefile

```
CC = gcc
CFLAGS = -Wall -pthread

all: server client

server: server.c operazioni.c common.h
  $(CC) $(CFLAGS) server.c operazioni.c -o server

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

clean:
  rm -f server client *.o
```

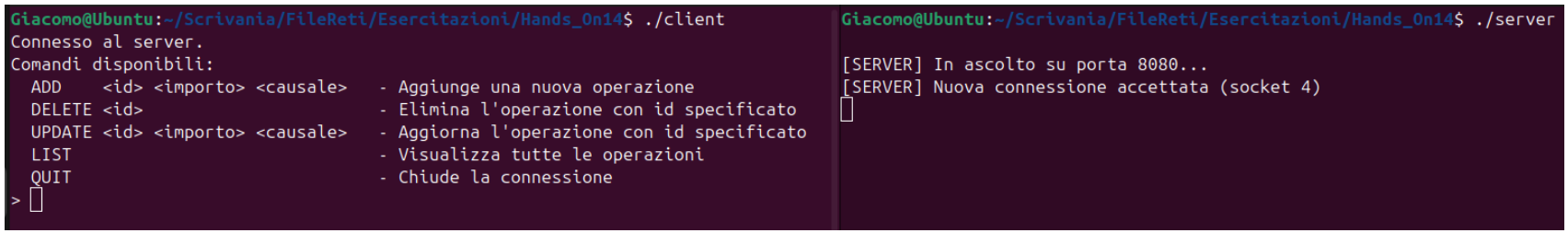
Validazione e Test

- Test tramite terminali multipli.
- Nessuna corruzione dei dati.
- Concorrenza gestita con successo.

Output: Esecuzione del Makefile

```
Giacomo@Ubuntu:~/Scrivania/FileReti/Esercitazioni/Hands_On14$ make
gcc -Wall -pthread server.c operazioni.c -o server
gcc -Wall -pthread client.c -o client
Giacomo@Ubuntu:~/Scrivania/FileReti/Esercitazioni/Hands_On14$
```

Output: Connessione Client/Server



Output: Test Completi

```
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)
```

Conclusioni

Il progetto ha evidenziato l'importanza della sincronizzazione nell'accesso concorrente a strutture dati condivise in ambienti multithread.

È stato realizzato con successo un server TCP robusto, capace di:

- Gestire richieste da più client in parallelo.
- Eseguire operazioni sulle transazioni bancarie in modo sicuro.

Sviluppi futuri

Possibili miglioramenti ed estensioni del progetto includono:

- Persistenza delle operazioni su file o database.
- Autenticazione e gestione degli utenti.
- Supporto per più correntisti con dati separati.
- Gestione avanzata di timeout e disconnessioni client.
- Sviluppo di un'interfaccia grafica per il client.