Implementazione di un server REST concorrente in C

Fabiana Presti
Matricola: 554563

6 luglio 2025

Indice

Contesto del progetto

- Sviluppo di un server REST in linguaggio C
- Gestione concorrente delle richieste tramite thread
- Interazione con un database SQLite
- Frontend dinamico con HTML, CSS e JavaScript

Architettura REST

REST (REpresentational State Transfer) è uno stile architetturale per la progettazione di sistemi distribuiti, in particolare servizi web.

Introdotto da *Roy Fielding* nella sua tesi di dottorato (2000), REST definisce un insieme di vincoli per sistemi che:

- comunicano via HTTP,
- utilizzano URL univoci per identificare risorse,
- manipolano le risorse tramite metodi standard: `GET`, `POST`, `PUT`, `DELETE`.

REST e server web

Un **server REST** espone un'interfaccia HTTP e risponde a richieste client secondo le convenzioni REST. Le caratteristiche principali sono:

- **Statelessness**: ogni richiesta è indipendente.
- **Scalabilità**: facilmente distribuibile e replicabile.
- **Rappresentazioni**: le risorse sono restituite come JSON, XML, ecc.

È una delle soluzioni più adottate per sviluppare API web moderne e interoperabili. (Architectural Styles and the Design of Network-based Software Architectures, chapter 5)

Esempio di chiamata REST

Esempio

`GET /enti` → Restituisce un elenco di enti in formato JSON
`POST /donazioni` → Inserisce una nuova donazione

Le API REST sono facilmente testabili e leggibili anche da client leggeri come browser, ‘curl‘ o JavaScript.

Il server utilizza TCP

Il server ShareCare utilizza il protocollo **TCP**, come si nota dalla creazione del socket:

```
int server_sock = socket(AF_INET, SOCK_STREAM, 0);
```

Listing 1: Creazione socket TCP

`SOCK_STREAM` indica una comunicazione **orientata alla connessione**, tipica del protocollo TCP. Questo è fondamentale per applicazioni RESTful basate su HTTP.

Differenza tra TCP e UDP

Caratteristica	**TCP**	**UDP**
Connessione	Sì (handshake 3-way)	No
Affidabilità	Alta (ritrasmissione)	Nessuna garanzia
Ordine dei pacchetti	Garantito	Non garantito
Velocità	Più lento (overhead)	Più veloce
Uso tipico	HTTP, FTP, SMTP	VoIP, DNS, streaming

Il protocollo **TCP** è più adatto per applicazioni web che richiedono *affidabilità* e *ordine*, come i server REST.

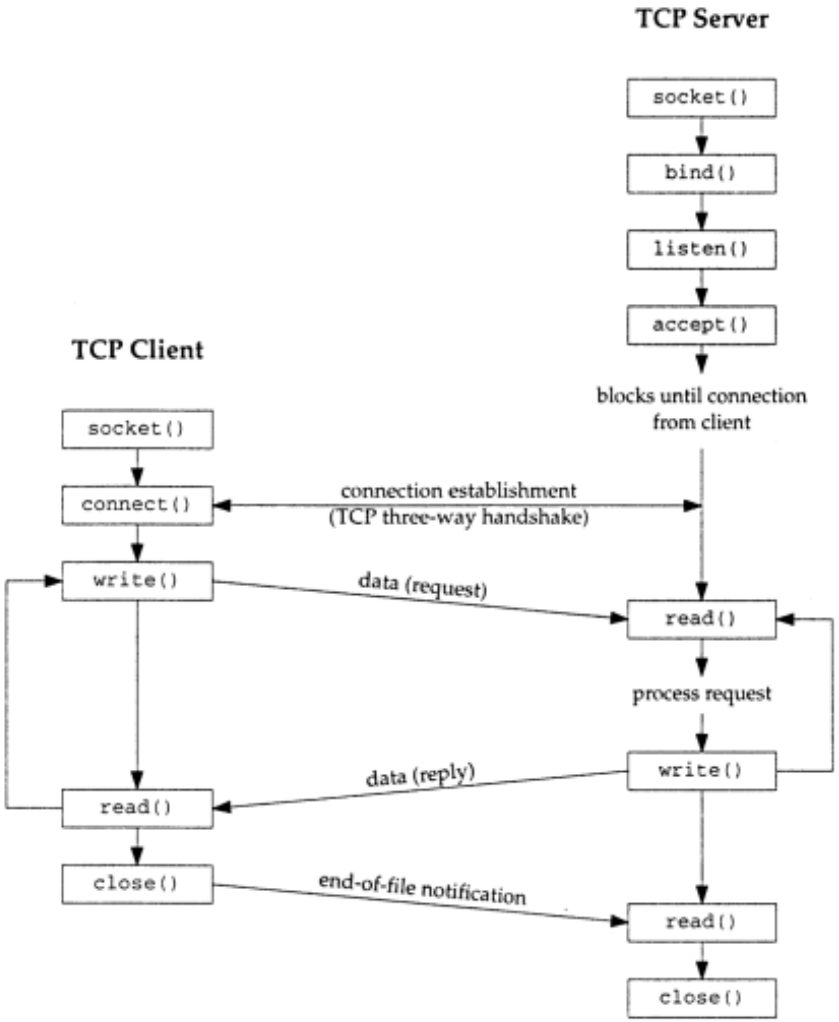

Figura 1: Modello TCP

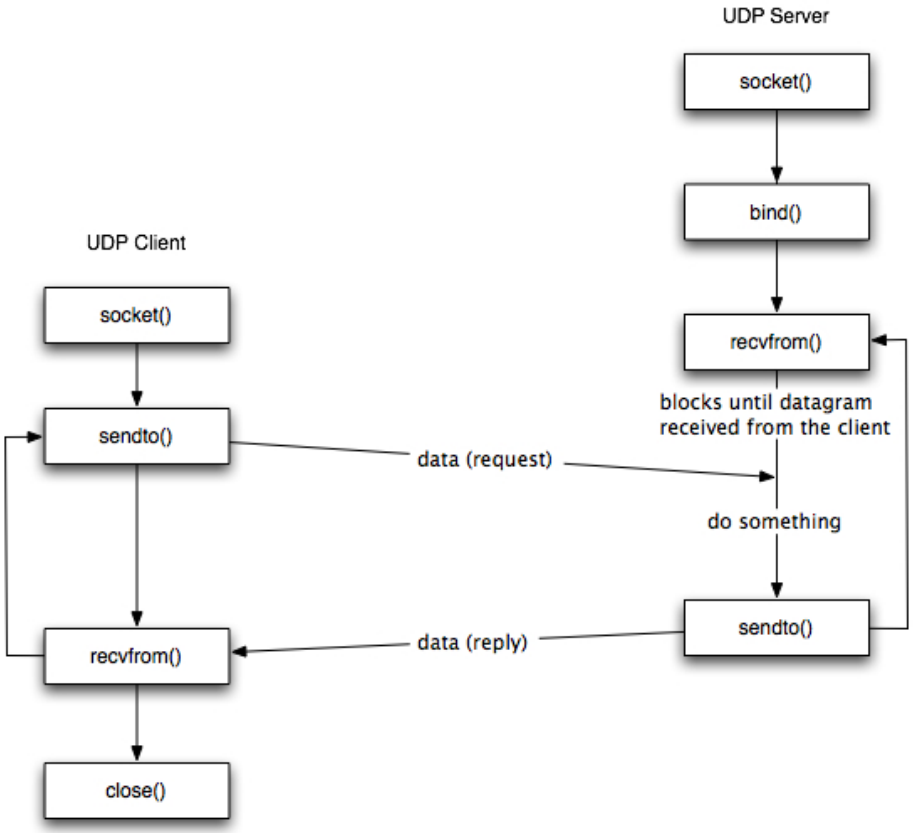

Figura 2: Modello UDP

HTTP e modello client-server

Il protocollo **HTTP (HyperText Transfer Protocol)** è alla base della comunicazione Web.

- Utilizza una architettura **client-server**.
- I client (es. browser) inviano richieste, i server rispondono con risorse.
- Il server REST implementato in C riceve richieste HTTP da client tramite socket TCP.

Socket TCP

La comunicazione avviene tramite `socket`, ovvero endpoint bidirezionali.

- `socket()` crea un socket.
- `bind()` collega il socket a un indirizzo IP e una porta.
- `listen()` mette il socket in attesa.
- `accept()` accetta una connessione in arrivo.

Esempio

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
```

Indirizzi IP e Maschere di rete

- Ogni host è identificato da un indirizzo IP.
- La **maschera di rete** (subnet mask) distingue la parte di rete da quella host.
- Esempio: `IP: 192.168.1.10, Maschera: 255.255.255.0`
- Risultato: `Network ID: 192.168.1.0, Host ID: .10`

Spazio degli indirizzi IPv4

- Lo spazio IP è diviso in:
  1. Indirizzi pubblici
  2. Indirizzi privati (es. 192.168.0.0/16)
  3. Indirizzi riservati (loopback, multicast, ecc.)
- Con l'uso delle maschere, è possibile suddividere una rete in **sottoreti**.

Bloccante + Multithread

Il server REST di **ShareCare** è stato implementato in C con un modello **bloccante** + **multithread**.
Ogni thread rimane in attesa (*blocking*) su chiamate di sistema come: `accept()`, `recv()`, `read()` finché l'evento desiderato non si verifica.

Snippet accept() bloccante

```
while (1) {
        int client_sock = accept(server_sock, (struct
            sockaddr *)&client_addr, &client_len);
        if (client_sock < 0) {
            perror("accept");
            continue;
        }
```

Listing 2: accept() nel thread principale

Snippet `recv()` bloccante

```
static void gestisci_client(int client_sock, sqlite3
    *db) {
    char buffer[BUF_SIZE];
    int bytes = recv(client_sock, buffer,
        sizeof(buffer) - 1, 0);
    if (bytes <= 0) return;
    buffer[bytes] = '\0';
```

Listing 3: recv() nel thread del client

Cos'è una *blocking call*

Definizione: *Una chiamata* ***bloccante*** *è una funzione o operazione che sospende l'esecuzione del programma finché non viene completata. Durante l'attesa, il thread che ha invocato la chiamata rimane dormiente, senza consumare cicli di CPU.*

Come funziona:

1. **Invocazione:** chiamata a `accept()`, `read()`, ecc.
2. **Attesa:** il kernel sospende il thread se la risorsa non è pronta.
3. **Ripresa:** il thread viene riattivato al verificarsi dell'evento.

Vantaggi e svantaggi

Vantaggi:

- *Semplicità:* codice lineare e facile da mantenere.
- *Efficienza CPU:* il thread dorme, no busy-waiting.

Svantaggi:

- *Scalabilità limitata:* un thread per client → consumo memoria.
- *Ritardi potenziali:* un thread bloccato può rallentare tutto.

Confronto: bloccante vs non-bloccante

Codice bloccante

```
1 int client = accept(sock, ...); //si
      blocca
```

Codice non-bloccante

```
1 fcntl(sock, F_SETFL, O_NONBLOCK);
2 for (;;) {
3   int client = accept(sock, ...);
4   if (client == -1 && errno ==
        EWOULDBLOCK)
5     continue;
6 }
```

Conclusione tecnica

Ho preferito il modello **bloccante** + **multithread** per risparmiare cicli di clock e perché è adeguato per il carico didattico.
In ambienti reali ad alta concorrenza si può valutare I/O non-bloccante con `select()`, `poll()`, `epoll()`;

Modello multithread

Il server **ShareCare** sfrutta il multithreading per gestire più client in parallelo. Ad ogni nuova connessione accettata, viene creato un **nuovo thread** che gestisce in autonomia quella sessione. Questo evita che un client in attesa blocchi il resto del server.

Snippet: creazione del thread

```
pthread_t tid;
struct ThreadArgs *args = malloc(sizeof(struct
    ThreadArgs));
args->sock = client_sock;

pthread_create(&tid, NULL, thread_client, args);
pthread_detach(tid);   //il thread si autodistrugge
    a fine lavoro
```

Listing 4: Creazione e detach di un thread

Viene passato un argomento personalizzato (il socket client), e il thread viene scollegato dal main thread per liberare le risorse automaticamente.

Cos'è un thread?

Definizione

Sebbene normalmente si pensi a un processo come a un unico flusso di controllo, nei sistemi moderni un processo può essere costituito da più **unità di esecuzione**, chiamate **thread**, ciascuna delle quali:

- è eseguita nel contesto del processo padre;
- condivide lo stesso codice, heap e dati globali.

- È più facile condividere informazioni tra **thread** che tra processi.
- I **thread** sono più leggeri da creare rispetto ai processi.
- Il multithreading è utile per sfruttare più core/CPU.

Il server è concorrente

Il server REST **ShareCare** è stato progettato per essere **concorrente**: è in grado di gestire più client nello stesso intervallo di tempo. Questo è possibile grazie alla creazione di un thread dedicato per ogni connessione.

Anche su una macchina **single-core**, la concorrenza viene ottenuta tramite **interleaving** temporale (*thread switching*) gestito dal sistema operativo.

Definizione generale di concorrenza

Definizione

Il termine **concorrenza** si riferisce al concetto generale di un sistema con attività multiple e simultanee che concorrono a sfruttare risorse condivise.
Il termine **parallelismo**, invece, si riferisce all'uso della concorrenza per rendere un sistema più veloce su più processori.

Attenzione:

- Un'esecuzione concorrente **non è necessariamente** parallela (es: su CPU single-core).
- Un'esecuzione parallela **è sempre** anche concorrente.

Tipi di concorrenza

Concorrenza può essere ottenuta in diversi contesti:

- **Simulata** (su processori singoli): basata su *thread switching* gestito dal kernel.
- **Multiprocessore reale (multicore)**: ogni thread può girare fisicamente in parallelo.
- **Multiprocessore avanzato**: con supporto a *hyperthreading* e *instruction pipelining*.

Conclusioni tecniche

Il progetto **ShareCare** ha dimostrato che è possibile realizzare un **server REST concorrente** in linguaggio C, capace di gestire più client in parallelo con l'uso di **socket TCP** e `pthread`.

Risultati raggiunti:

- Gestione corretta di connessioni HTTP tramite `socket` bloccanti.
- Supporto multithread: un thread per ogni client.
- Persistenza dati tramite `SQLite`.
- Interfaccia web semplice, con form HTML e stile CSS.
- API REST funzionante per enti, articoli, donazioni e blog.

Riassunto didattico

Il progetto consolida concetti fondamentali di:

- Networking (protocollo TCP, porta, indirizzo IP).
- Programmazione concorrente
- Sistemi distribuiti e comunicazione client/server.
- Interazione con database relazionali (SQL).
- Architettura REST semplificata.

Sviluppi futuri

Possibili miglioramenti e estensioni:

- Passaggio a **I/O non bloccante** con `select()`, `poll()`, `epoll()`.
- Aggiunta di **autenticazione reale**
- Validazione e sanitizzazione input utente