Realizzazione di un Server HTTP concorrente in C

Gestione GET, POST, PUT, DELETE – RFC 2616

Alessandro Guarnera
- Gabriel Piparo

Novembre 2025

Indice

Obiettivo del progetto

- Implementare un **server HTTP concorrente** in linguaggio C.
- Gestione metodi: **GET, POST, PUT, DELETE**.
- Risposte conformi a **RFC 2616**: 200, 201, 204, 400, 401, 404, 409, 500.
- Supporto a risorse pubbliche e private con autenticazione.

Modello Client–Server

Il client invia:

- metodo HTTP
- percorso risorsa
- header
- eventuale body

Il server risponde con:

- status code
- header di risposta
- contenuto della risorsa

HTTP secondo la RFC 2616

HTTP (HyperText Transfer Protocol) è un protocollo di comunicazione **client–server** basato sul modello **request/response**.

Principi fondamentali della RFC 2616:

- Protocollo **stateless**: ogni richiesta è indipendente.
- Comunicazione orientata al testo (**human-readable**).
- Ogni messaggio **Request** si compone di:
  - **Request line**: metodo, path, versione (es. GET /index.html HTTP/1.1)
  - **Header**: informazioni aggiuntive (Host, Content-Type, ecc.)
  - **Body** (opzionale): presente in POST/PUT

Response HTTP

- Inviata dal server dopo aver elaborato la richiesta
- Struttura:
  - **Status Line**: `HTTP/1.1 200 OK`
  - **Header di risposta**: Content-Type, Content-Length, ecc.
  - **Body**: contenuto della risorsa (HTML, testo, JSON...)

Gli status code sono divisi in categorie:

- 1xx — Informational
- 2xx — Successo (200, 201, 204)
- 3xx — Redirect
- 4xx — Errori client (400, 401, 404, 409)
- 5xx — Errori server (500)

RFC ufficiale: `https://datatracker.ietf.org/doc/html/rfc2616`

Limitazioni di select()

- Iterazione completa ad ogni chiamata - Complessità O(n)
- Limite massimo 1024 file descriptor
- Elevato overhead con molti client

Perché epoll()?

- Progettato per gestire **tantissime connessioni**
- Notifiche basate su eventi (**ready list**)
- Complessità O(1)
- Supporto a **edge-triggered**

Esempio epoll

```
struct epoll_event ev;
ev.events = EPOLLIN | EPOLLET;
ev.data.fd = sockfd;

epoll_ctl(epollfd, EPOLL_CTL_ADD, sockfd, &ev);
```

Struttura generale

- Applicazione C **multithreaded**
- epoll per gestione non-bloccante dei socket
- Un thread per ogni richiesta in arrivo
- Mutex per le operazioni di scrittura

Inizializzazione server

```
int server_fd = socket(AF_INET, SOCK_STREAM, 0);
set_nonblocking(server_fd);
bind(server_fd, ...);
listen(server_fd, 128);
```

Struttura del file system

- **/www** — risorse pubbliche
- **/www/private** — risorse protette (token: secret123)

Parsing HTTP — Request Line

```
if (sscanf(request, "%s␣%s␣%s",
           req->method,
           req->path,
           req->version) != 3)
    return -1;
```

Spiegazione:

- La request line è la prima riga della richiesta HTTP.
- Esempio: `GET /index.html HTTP/1.1`
- `sscanf()` estrae:
  - **method** — GET, POST, PUT, DELETE
  - **path** — risorsa richiesta (/index.html)
  - **version** — HTTP/1.1
- Se non vengono estratti esattamente 3 campi → **400 Bad Request**

Parsing HTTP — Header

```
ptr = strstr(request, "\r\n") + 2;

while (req->header_count < MAX_HEADERS && strncmp(ptr, "\r\n
    ", 2) != 0) {
        char name[MAX_HEADER_NAME];
        char value[MAX_HEADER_VALUE];
        if (sscanf(ptr, "%[^:]: %[^\r\n]", name, value) ==
            2) {
            strcpy(req->headers[req->header_count].name,
                name);
            strcpy(req->headers[req->header_count].value,
                value);
            req->header_count++;
        }
        ptr = strstr(ptr, "\r\n");
        if (!ptr) break;
        ptr += 2;
    }
```

Parsing HTTP — Header

Spiegazione:

- Itera riga per riga fino alla riga vuota
- Divide ogni header in **nome** e **valore**
- Memorizza gli header nella struct

- Gli header iniziano dopo la request line.
- Vengono letti fino alla riga vuota: `\r\n\r\n`
- Ogni header è nella forma: `Nome:  Valore`
- Gli header vengono salvati in: `req->headers[]`
- **MAX_HEADERS = 10**

Parsing HTTP — Body (POST/PUT)

```
const char *body_start = strstr(request, "\r\n\r\n");
if (body_start) {
    body_start += 4;

    req->body = malloc(content_length + 1);
    memcpy(req->body, body_start, content_length);
    req->body[content_length] = '\0';
    req->body_length = content_length;
}
```

Parsing HTTP — Body (POST/PUT)

Spiegazione:

- Il body esiste solo in **POST** e **PUT**.
- Il server lo trova dopo la sequenza: `\r\n\r\n`
- Prima deve essere letto l'header `Content-Length`.
- Il contenuto viene copiato in **req→body**.
- Il body viene usato per:
  - creare file (POST)
  - sovrascrivere risorsa (PUT)

Metodo GET

- Restituisce contenuto risorsa
- Se path contiene /private/ $\rightarrow$ richiede token
- Risposte possibili:
  - 200 OK
  - 401 Unauthorized
  - 404 Not Found
  - 400 Bad Request
  - 500 Internal Server Error

Metodo POST

- Crea un nuovo file .txt
- Richiede Content-Type e Content-Length
- Risposta:
  - 201 Created
- Errori possibili:
  - 409 Conflict (file esistente)
  - 401 Unauthorized
  - 400 Bad Request
  - 500 Internal Server Error

Metodo PUT

- Aggiorna o crea risorsa
- Uso del **mutex** globale
- Risposte:
  - 201 Created
  - 200 OK
  - 401 Unauthorized
  - 400 Bad Request
  - 500 Internal Server Error

Metodo DELETE

- Elimina file richiesto
- Richiede autenticazione se privato
- Risposte:
  - 204 No Content
  - 404 Not Found
  - 400 Bad Request
  - 401 Unauthorized
  - 500 Internal Server Error

Test con browser

- GET su risorse pubbliche $\rightarrow$ 200 OK
- GET su risorse protette senza token $\rightarrow$ 401 Unauthorized
- Risorsa inesistente $\rightarrow$ 404 Not Found

Test con cURL

- POST crea file nuovo $\rightarrow$ 201 Created
- POST su file esistente $\rightarrow$ 409 Conflict
- PUT aggiorna correttamente il file
- DELETE elimina la risorsa $\rightarrow$ 204 No Content

Test di Carico (wrk)

GET — 50 connessioni

- Throughput: 6902 req/s
- Latenza media: 6.92 ms

PUT — 50 connessioni

- 7369 req/s
- Latenza media: 6.59 ms

Analisi dei risultati:

- Il server mantiene un throughput elevato (circa 7000 richieste al secondo) anche con 50 connessioni simultanee.
- La latenza rimane molto bassa (6–7 ms), indice di una gestione efficiente dell'I/O tramite `epoll`.
- Le prestazioni del metodo **PUT** risultano addirittura leggermente migliori del **GET**, grazie alla rapidità delle operazioni di scrittura su file.
- Nessun errore di competizione o stallo: il server risponde in modo stabile e costante.

Test di Carico: Carico Estremo (500 connessioni)

Per valutare il comportamento del server in condizioni di carico molto elevato, è stato eseguito un secondo test utilizzando 12 thread, 500 connessioni simultanee e una durata complessiva di 30 secondi:

```
wrk -t12 -c500 -d30s http://localhost:8080/index.html
```

Risultati ottenuti:

```
Running 30s test @ http://localhost:8080/index.html
  12 threads and 500 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    32.71ms   82.74ms   1.79s    97.14%
    Req/Sec   494.05    330.72     1.44k    63.47%
  161801 requests in 30.04s, 21.45MB read
  Socket errors: connect 0, read 70, write 0, timeout 38
Requests/sec:   5385.76
Transfer/sec:    731.07KB
```

Analisi

- Il throughput rimane elevato: **5385.76 req/s**, segno che il server riesce a mantenere prestazioni stabili anche sotto forte stress.
- La latenza media aumenta a **32.71 ms**, dovuta all'elevato numero di richieste concorrenti.
- Il valore di latenza massima (**1.79 s**) evidenzia che alcune connessioni hanno subito ritardi importanti durante i picchi di carico.
- Si osservano errori di lettura (70 read errors) e timeout (38).

Considerazioni

Nonostante la pressione elevata:

- il server **non è mai andato in crash**;
- ha continuato a servire richieste durante tutta la durata del test;
- ha mantenuto un throughput superiore a 5000 richieste al secondo.

Questi risultati mostrano che il server è in grado di sostenere carichi estremamente elevati, confermando la solidità dell'implementazione e l'efficacia della gestione con `epoll` e threading concorrente.

Conclusioni

- Server stabile e performante
- Scalabile fino a centinaia di connessioni
- Nessun deadlock riscontrato

Sviluppi Futuri

- Supporto HTTPS (OpenSSL)
- Thread pool per evitare overhead di creazione

Grazie per l'attenzione!