Reti e Sistemi Distribuiti Modulo B: Progetto Finale

Alessandro Guarnera - 556579 Gabriel Piparo - 555736

November 2025

1 INTRODUZIONE

L'obiettivo del progetto è quello di realizzare un server HTTP concorrente in linguaggio C.
Il server deve essere in grado di gestire le principali richieste HTTP quali: GET, POST, PUT e DELETE. Fornendo le corrispondenti risposte standard conformi alle specifiche definite nella RFC 2616 (200, 201, 204, 400, 401, ...).

2 PROBLEM STATEMENT

Il problema da affrontare riguarda lo sviluppo di server HTTP in linguaggio c che rispetto standard RFC 2616. Il protocollo HTTP definisce un insieme di regole per la comunicazione client-server dove:

- Il `Client` invia una richiesta testuale contente il metodo (GET, POST, PUT, ecc), il percorso della risorsa e gli eventuali header.
- Il `Server` ha il compito di elabora le richieste, accede alla risorsa e restituisce una risposta al client formattata secondo lo standard.

Le richieste che il nostro server dovrà gestire sono:

- `GET:` viene usato per richiedere una risorsa.
- `POST:` serve per inviare dati al server o creare nuove risorse nel nostro caso file.
- `PUT:` viene utilizzato per creare o aggiornare (sovrascrivere) una risorsa esistente.
- `DELETE:` serve per eliminare una risorsa dal server.

Le risposte che dovrà dare:

- `200 OK:` indica che la richiesta è stata eseguita correttamente.

- `201 Created:` viene restituita quando una nuova risorsa è stata creata o aggiornata con successo.
- `204 No Content:` indica che la richiesta è stata completata correttamente, ma non c'è alcun contenuto da restituire.
- `400 Bad Request` significa che la richiesta inviata dal client non è valida o non rispetta il formato previsto.
- `401 Unauthorized:` Viene restituita quando manca o è errato l'header di autenticazione..
- `404 Not Found:` indica che la risorsa richiesta non è presente sul server.
- `409 Conflict:` segnala un conflitto nella gestione della risorsa, ad esempio un errore durante la scrittura di un file già in uso o non accessibile. È usata nel metodo PUT quando l'operazione di aggiornamento fallisce per motivi legati al file system.
- `500 Internal Server Error:` viene inviata quando si verifica un errore interno non previsto dal programma

3 STATO DELL'ARTE

Nel contesto dei server di rete, uno degli obiettivi principali è gestire in modo efficiente più connessioni o richieste contemporaneamente. Abbiamo diverse opzioni per fare ciò, una potrebbe essere quella di usare un thread per ogni nuova connessione, tuttavia questo modello, pur permettendo l'elaborazione parallela, comporta un consumo elevato di risorse e un overhead notevole nella creazione e nella gestione dei thread.
Per risolvere questi limiti, nei sistemi UNIX/Linux sono stati introdotti meccanismi di I/O multiplexing; ovvero tecniche che permettono ad un singolo thread di monitorare più file descriptor, come select() e epoll(). Abbiamo visto però che la funzione select() sposta il monitoraggio dei FD da user-space a kernel-space. Ogni volta che si chiama select, il kernel itera su tutti gli FD nel set per verificare lo stato. Questo ha complessità O(n), quindi diventa lento con molti fd. Inoltre, c'‘e un limite al numero massimo di fd, di solito 1024. Qundi la nostra scelta è ricaduta sulla funzione epoll(), progettata per superare i limiti di select() e per monitorare efficientemente eventi di I/O su un gran numero di file descriptor (FD). A basso livello, funziona attraverso tre componenti principali:

- **strutture dati kernel**
- **notifiche asincrone**
- **ottimizzazioni per scalabilità**

Funzionamento di epoll()

```
int epoll_create1(int flags);
```

Con epoll_create1 si crea un'istanza, e con epoll_ctl si aggiungono/modificano gli fd.

```
int epoll_ctl (int epoll_fd, int op, int fd, struct epoll_event*
    event);
```

- `epoll fd`: Descrittore restituito da epoll_create1.
- `op`: Operazione da eseguire:
  - `EPOLL_CTL_ADD`: Aggiunge fd alla lista degli eventi monitorati con i settaggi specificati in event.
  - `EPOLL_CTL_MOD`: Modifica le condizioni di monitoraggio di fd.
  - `EPOLL_CTL_DEL`: Rimuove fd dalla lista.
- `fd`: File descriptor da gestire.
- `event`: Struttura di configurazione.

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

if(epoll_ctl(epoll_fd, EPOLL_CTL_ADD, sockfd, &event) == -1) {
    perror("epoll_ctl");
    exit(EXIT_FAILURE);
}
```

struct epoll_event definisce un evento monitorato da epoll e contiene due campi principali:

- `events`: Specifica quali eventi devono essere monitorati.
- `data` Contiene dati personalizzati che l'utente pu'o associare all'evento.

lista dei set possibili:

- `EPOLLIN`: Il file descriptor è pronto per la lettura.
- `EPOLLOUT`: Il file descriptor è pronto per la scrittura.
- `EPOLLET`: Modalità Edge-Triggered (Default: level triggered)

Il kernel mantiene una lista di interesse e una lista pronta. Quando chiami epoll_wait, il kernel restituisce gli fd pronti, senza doverli iterare tutti (perchè li conosce già). La complessità è O(1) per il retrieval di fd pronti, quindi è più scalabile.

Strutture kernel

Il kernel gestisce epoll() grazie a due strutture dati chiave:

- `Interest List (Tree):` Un albero che tiene traccia di tutti i FD registrati con epoll_ctl().
- `Ready List (Linked List):`Una lista collegata che mantiene sempre disponibile i FD pronti per operazioni di I/O e che viene aggiornata dal kernel quando si verificano eventi

Modalità di notifica

epoll supporta due modalità:

- `Level-Triggered (LT):` In questa modalità, il kernel notifica continuamente la presenza di un evento finché il file descriptor rimane in uno stato "pronto". Epoll_wait() continuerà a segnalare l'evento di lettura finché tutti i dati non saranno stati consumati.
- `Edge-Triggered (ET):`la notifica viene inviata solo nel momento in cui lo stato del file descriptor cambia; ad esempio quando arrivano nuovi dati in un buffer precedentemente vuoto o quando un socket diventa pronto per la scrittura. Il kernel quindi non ripete la notifica finché la condizione non cambia di nuovo.

Richiesta e risposta HTTP

Nel modello HTTP, la comunicazione tra client e server segue il paradigma request–response, in cui ogni richiesta del client genera una singola risposta del server.

Richiest

Una richiesta è un messaggio testuale contiene:

- Una riga con il metodo (GET, POST, PUT, DELETE), il percorso della risorsa e la versione del protocollo (HTTP/1.1).
- Una serie di header.
- Un eventuale corpo.

Il server deve analizzare il messaggio, verificare la validità del formato e interpretare correttamente il metodo per determinare l'azione da eseguire sulla risorsa. Dopo aver elaborato la richiesta, il server invia un messaggio di risposta composto da:

- Una status line, che specifica la versione del protocollo e il codice di stato (es. 200 OK, 404 Not Found, 500 Internal Server Error).

- Eventuali header di risposta (es. Content-Type, Content-Length).
- un corpo contenente la risorsa richiesta o un messaggio informativo.

Le risposte HTTP sono classificate in cinque categorie principali:

- `1xx`: Informational.
- `2xx`: Successo (200, 201, 204).
- `3xx`: Redirezioni.
- `4xx`: Errori lato client (400, 401, 404, 409).
- `5xx`: Errori lato server (500).

4 METODOLOGIA

La metodologia adottata per lo sviluppo del progetto ha seguito un approccio modulare e incrementale, con l'obiettivo di realizzare un server HTTP concorrente, efficiente e facilmente estendibile. Il sistema è stato interamente implementato in linguaggio `C`, con particolare attenzione alla gestione simultanea di più connessioni e alla corretta implementazione del protocollo HTTP/1.1, in conformità alla specifica `RFC 2616`.

4.1 Architettura generale

Il server è stato progettato come un'applicazione **multithreaded** che utilizza il meccanismo di **I/O multiplexing** fornito da `epoll`. Questo approccio consente di gestire in modo non bloccante numerose connessioni client, riducendo l'uso della CPU e migliorando la scalabilità del sistema.

Il funzionamento può essere descritto in due fasi principali:

1. **Gestione delle connessioni:**
   - Il server crea un socket di ascolto e lo configura come non bloccante.
   - Il socket principale viene registrato all'interno di un'istanza di `epoll`, che monitora gli eventi di lettura in ingresso.
   - Quando un nuovo client si connette, la connessione viene accettata e il relativo file descriptor viene aggiunto alla lista di `epoll`.
   - La chiamata `epoll_wait()` consente di identificare in modo efficiente i socket pronti per la lettura senza utilizzare cicli di polling continui.
2. **Gestione delle richieste HTTP:**
   - Per ogni socket che risulta pronta nella *ready list* di `epoll`, viene creato un **thread dedicato**.

- Il thread legge la richiesta dal socket, effettua il **parsing** del messaggio HTTP per identificare il metodo, il percorso e le intestazioni.
- Successivamente, invoca la funzione di elaborazione specifica in base al metodo richiesto (`GET`, `POST`, `PUT`, `DELETE`) e costruisce la risposta da inviare al client.
- Una volta completata la comunicazione, il thread chiude la connessione e termina, liberando le risorse utilizzate.

Questo modello ibrido, che combina l'uso di `epoll` per la gestione efficiente degli eventi I/O e la creazione di thread separati per l'elaborazione delle richieste, garantisce un'elevata concorrenza e tempi di risposta ridotti.

4.2 Gestione delle richieste e delle risposte HTTP

Il server è in grado di gestire i principali metodi previsti dallo standard HTTP/1.1:

- `GET`
- `POST`
- `PUT`
- `DELETE`

Sono stati implementati diversi codici di risposta, sia di successo che di errore, per una gestione più completa dei casi possibili:

- **200 OK** – richiesta elaborata correttamente;
- **201 Created** – risorsa creata con successo;
- **204 No Content** – richiesta elaborata senza corpo di risposta;
- **400 Bad Request** – richiesta malformata;
- **401 Unauthorized** – accesso non autorizzato;
- **404 Not Found** – risorsa richiesta non trovata;
- **409 Conflict** – conflitto nella gestione della risorsa;
- **500 Internal Server Error** – errore interno del server.

4.3 Gestione delle risorse e struttura del file system

Il server gestisce due aree principali nel file system:

- **/www** – directory pubblica accessibile a tutti i client, in cui sono contenute le risorse statiche (file HTML, immagini, CSS, ecc.);

- **/www/private** – directory privata, accessibile solo in presenza di credenziali o autorizzazioni specifiche, utilizzata per simulare risorse protette da autenticazione.

Durante l'elaborazione di una richiesta HTTP, il percorso richiesto viene concatenato alla directory di base corrispondente, in modo da restituire il contenuto corretto o il codice di errore appropriato nel caso in cui la risorsa non sia disponibile o l'accesso non sia consentito.

4.4 Struttura del codice

Il progetto è stato suddiviso in più moduli per garantire una chiara separazione delle responsabilità e facilitare la manutenzione:

- `inizializzazione_server.c/h` – gestione dell'avvio del server, creazione del socket e configurazione in modalità non bloccante;
- `gestione_client.c/h` – gestione dei thread e lettura delle richieste provenienti dai client;
- `http.c/h` – parsing delle richieste HTTP e costruzione delle risposte da inviare;

Questa organizzazione modulare consente di estendere facilmente il progetto, ad esempio aggiungendo nuovi metodi HTTP o meccanismi di autenticazione avanzata.

4.5 Header

4.5.1 Inclusione Librerie

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/epoll.h>
#include <pthread.h>
```

- `<stdio.h>`: Input/output standard (printf, scanf)
- `<stdlib.h>`: Memoria, conversioni, uscita programma
- `<string.h>`: Gestione stringhe e memoria
- `<unistd.h>`: Funzioni POSIX (read, write, fork)

- `<fcntl.h>`: Apertura e controllo file
- `<errno.h>`: Gestione errori di sistema
- `<sys/types.h>`: Tipi di dato di sistema
- `<sys/socket.h>`: Creazione e uso socket
- `<netinet/in.h>`: Strutture per rete Internet
- `<sys/epoll.h>`: Gestione eventi multipli (epoll)
- `<pthread.h>`: Thread POSIX (multithreading)

4.5.2 inizializzazione.h

```
#ifndef INIT_H
#define INIT_H

#define PORT 8080
#define BUFFER_SIZE 1024
#define MAX_EVENTS 512

void set_nonblocking(int sockfd);
int init_server();

#endif
```

- `#define PORT 8080`: Porta di ascolto del server
- `#define BUFFER_SIZE 1024`: Dimensione del buffer dati
- `#define MAX_EVENTS 512`: Numero massimo di eventi gestibili

4.5.3 gestione_client.h

```
#ifndef GESTIONE_CLIENT_H
#define GESTIONE_CLIENT_H
extern int epollfd;
void *thread_function(void* arg);

#endif
```

4.5.4 mutex.h

```
#ifndef MUTEX_H
#define MUTEX_H
#include <pthread.h>

extern pthread_mutex_t mutex_globale;

#endif
```

`extern pthread_mutex_t mutex_globale;` dichiara un mutex globale definito in un altro file.

4.5.5 http.h

```c
#ifndef HTTP_H
#define HTTP_H

#include <stddef.h>

#define MAX_HEADERS 10
#define MAX_HEADER_NAME 64
#define MAX_HEADER_VALUE 256
#define MAX_METHOD_LEN 8
#define MAX_PATH_LEN 256
#define MAX_VERSION_LEN 8
#define MAX_FILE_SIZE 1024

// Una coppia nome: valore per un header HTTP
typedef struct {
    char name[MAX_HEADER_NAME];
    char value[MAX_HEADER_VALUE];
} http_header_t;

// Rappresenta una richiesta HTTP completa
typedef struct {
    char method[MAX_METHOD_LEN];        // GET, POST, PUT, DELETE...
    char path[MAX_PATH_LEN];            // /index.html
    char version[MAX_VERSION_LEN];      // HTTP/1.1
    http_header_t headers[MAX_HEADERS];// array di header
    int header_count;                   // quanti header effettivi
    char *body;                         // puntatore al corpo (per
    POST/PUT)
    size_t body_length;                 // lunghezza del body
} http_request_t;

// Funzione da implementare in http.c richiamata da gestione_client
    .c
void gestione_request(int client_fd, const char *request);

#endif
```

- `#define MAX_HEADERS 10`: Numero massimo di intestazioni HTTP
- `#define MAX_HEADER_NAME 64`: Lunghezza massima del nome di un header
- `#define MAX_HEADER_VALUE 256`: Lunghezza massima del valore di un header
- `#define MAX_METHOD_LEN 8`: Lunghezza massima del metodo HTTP (es. GET, POST)
- `#define MAX_PATH_LEN 256`: Lunghezza massima del percorso richiesto
- `#define MAX_VERSION_LEN 8`: Lunghezza massima della versione HTTP
- `#define MAX_FILE_SIZE 1024`: Dimensione massima del file gestibile

Inoltre nel file sono presenti le struct principali dei messaggi HTTP per gestire le request dei client. Ovvero memorizziamo tutti valori ottenuti dal parsing del messaggio nella struct per manipolare la meglio i dati ottenuti e creare la response.

4.6 Inizializzazione del Server

L'implementazione del primo file `inizializzazione.c`, si occupa di:

- Creare un socket di ascolto.
- configurare il socker per funzionare in modalità non bloccante.
- effettuare il bind per legarlo ad una porta di rete (PORT).
- mettere il socket in ascolto di nuove connessioni in arrivo.
- infine restituisce il file descriptor del socket di ascolto per gestirlo nel main.

Rendiamo il socket non bloccante in modo le che chiamate come (accept(), recv(), o send()) non bloccano l'esecuzione del programma se non ci sono dati o connessioni disponibili ma ritornano subito permettendo l'esecuzione continua. Utile per server che utilizzano meccanismi come epoll.

```
void set_nonblocking(int sockfd) {
    int flags = fcntl(sockfd, F_GETFL, 0);
    fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);
}
```

- fcntl(): per ottenere i flag attuali del file descriptor (F_GETFL).
- Aggiunge il flag O_NONBLOCK, che rende il socket non bloccante.

La funzione principale che viene invocata nel main è `init_server()`, dentro la quale eseguiamo le seguenti operazioni:

```
    // Creazione del socket
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("Errore creazione socket");
        exit(EXIT_FAILURE);
    }
```

- AF_INET: usa il protocollo IPv4.
- SOCK_STREAM: socket TCP (connessione affidabile, orientata al flusso).
- Se fallisce, stampa un errore e termina.

```
// Configura l'indirizzo del server
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(PORT);
```

Dove `htons()` converte la porta da host byte order a network byte order, necessaria per la comunicazione in rete.

```
// Collega il socket alla porta
    if (bind(server_fd, (struct sockaddr *)&serv_addr, sizeof(
    serv_addr)) < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }
```

Collega il socket alla configurazione dell'indirizzo del server che continene anche la porta specificata. Se la porta è già in uso o non disponibile, fallisce.

```
    // Mette il socket in ascolto
    if (listen(server_fd, 128) < 0) {
        perror("listen failed");
        exit(EXIT_FAILURE);
    }
```

- Impostiamo il socket in modalità ascolto con il listen().
- 128 è la lunghezza massima della connessioni pendenti.

```
    printf("Server in ascolto sulla porta %d...\n", PORT);
    return server_fd;
```

Infine stampiamo un messaggio nel terminale e ritorniamo alla funzione main il server_fd utile per la gestione dell'accettazione delle nuove connessioni da parte di client. Anche esso verrà inserito nel set dell'epoll per controllare gli eventi di richiesta da parte dei client.

4.7 Thread Principale (main)

Il thread principale (main) nel file `server.c` è l'implementazione principale del nostro server multithread che si occupa di gestire gli eventi sui socket, (sia il socket principale per accettare le connessioni, che quelli associati ai vari client), con epoll. Inoltre per ogni evento di tipo richiesta da parte di un client, verrà gestito da un thread separato che sarà creato appositamente per gestire la richiesta.

```
...
#include "inizializzazione_server.h"
#include "gestione_client.h"
int epollfd;
...
    pthread_t thread_id;
    int sockfd, newsockfd, n, i;
    struct sockaddr_in cli_addr;
    socklen_t clilen_addrlen = sizeof(cli_addr);
    struct epoll_event ev, events[MAX_EVENTS];
```

- `thread_id` : ID del thread per pthread_create.

- `sockfd` : socket del server in ascolto per accettare nuove connessioni.
- `newsockfd` : socket del client appena connesso.
- `epollfd` : file descriptor dell'epoll dichiarato globale.
- `cli_addr` : struttura per salvare l'indirizzo del client.

```
    sockfd = init_server();
```

chiamiamo la funzione per inizializzare il server come visto prima.

```
    // Creazione dell'epoll instance
    if((epollfd=epoll_create1(0))==-1){
        perror("creazione istanza epoll fallita");
        exit(EXIT_FAILURE);
    }
```

- `epoll_create1(0)` : crea un epoll instance, usato per monitorare più socket contemporaneamente.
- Restituisce un `file descriptor epollfd` da usare con `epoll_ctl` e `epoll_wait`.

```
    // Creazione dell'epoll instance
    if((epollfd=epoll_create1(0))==-1){
        perror("creazione istanza epoll fallita");
        exit(EXIT_FAILURE);
    }
```

```

// Aggiungiamo il socket principale all'epoll
    ev.events = EPOLLIN; // Evento di lettura
    ev.data.fd = sockfd;
    if (epoll_ctl(epollfd, EPOLL_CTL_ADD, sockfd, &ev) == -1) {
        perror("epoll_ctl failed");
        exit(EXIT_FAILURE);
    }
```

- `EPOLLIN` : per controllare connessioni in arrivo
- `epoll_ctl()` : aggiunge il socket server all'epoll per monitorarlo per eventuali connessioni in arrivo.

4.7.1 Loop Principale

```

while (1) {
        // Aspettiamo che un evento accada su uno dei file
    descriptor
        n = epoll_wait(epollfd, events, MAX_EVENTS, -1);
        if (n == -1) {
            perror("epoll_wait failed");
            exit(EXIT_FAILURE);
        }
```

- `epoll_wait` : ci segnala gli eventi ready su tutti i file descriptor registrati, memorizzandoli nell'array di eventi `events`.
- `n` : numero di eventi ready.
- `-1` : attesa illimitata fino a quando non succede qualcosa.

```
1
2       // Gestiamo gli eventi
3       for (i = 0; i < n; i++) {
4           if (events[i].data.fd == sockfd) {
5               // Nuova connessione in arrivo
6               if ((newsockfd = accept(sockfd, (struct sockaddr *)
    &cli_addr, &clilen_addrlen)) < 0) {
7                   perror("accept failed");
8                   continue;
9               }
10
11              // Impostiamo il nuovo socket come non bloccante
12              set_nonblocking(newsockfd);
13
14              // Aggiungiamo il nuovo socket all'epoll
15              ev.events = EPOLLIN | EPOLLET; // Evento di lettura
    , modalita' edge-triggered
16              ev.data.fd = newsockfd;
17              if (epoll_ctl(epollfd, EPOLL_CTL_ADD, newsockfd, &
    ev) == -1) {
18                  perror("epoll_ctl failed");
19                  exit(EXIT_FAILURE);
20              }
21
22              printf("Nuova connessione, socket fd = %d\n",
    newsockfd);
23          } else {
24              int sd = events[i].data.fd;
25              int* arg = malloc(sizeof(int));
26              *arg = sd; //significa: vai nella zona di memoria
    puntata da arg e scrivi li dentro, Dentro la memoria puntata da
     arg, memorizza il numero sd
27              pthread_create(&thread_id, NULL, thread_function,
    arg);
28          }
29      }
30  }
```

Caso 1: l'evento riguarda il socket server (nuova connessione)

- accept accetta la connessione del client memorrizando l'fd del nuovo socket (newsockfd).
- Rende il socket del client non bloccante.
- Aggiunge il socket del client a epoll, con modalità edge-triggered (EPOLLET).
- Stampa messaggio di conferma.

Caso 2: l'evento rilevato riguarda un socket client:

- Alloca memoria per un intero (arg) e ci mette il socket descriptor.
- Crea un nuovo thread (pthread_create) per gestire la richiesta del client, passando arg.
- Il thread eseguirà la funzione thread_function
  (definita in `gestione_client.h/c`)

4.8 File gestione_client.c

Questo codice implementa la thread_function() che è la funzione principale che esegue ogni thread creato per gestire una specifica richiesta da parte di un client. in particolare la funzione

- Riceve come argomento arg, il socket del client.
- Legge continuamente i dati dal client, finchè sono presenti.
- richiama gestione_request() per elaborarli.
- Chiude il socket quando il client si disconnette o c'è un errore.
- Libera la memoria e termina il thread.

```
#include "gestione_client.h"
#include "http.h"
void *thread_function(void* arg) {
...
}
```

essa è una funzione pthread, quindi deve restituire void* e ricevere un void* come argomento. Ma arg è in realtà un puntatore a un intero, cioè il socket descriptor del client passato al thread per gestire la richiesta in modo da mandargli una response.
Recuperiamo il file descriptor nel seguente modo:

```
 int sd = *(int* )arg;   // recupera il file descriptor passato
```

Nello specifico alla creazione del thread, noi passiamo come argomento il contenuto di un puntatore ad int, che ha come valore un indirizzo di memoria allocato dinamicamente di 4 byte (int) dove il valore associato a tale indirizzo equivale al file descriptor in riferimento al client che effetua la richiesta (*arg = sd).

```
char request[BUFFER_SIZE];
int valore_letto;
```

request sarà il nostro buffer che salverà la richiesta ottenuta dal client, mentre valore_letto serve per memorizzare il numero di byte letto con read() per effettuare i controlli.

4.8.1 Lettura della richiesta

```
    if((valore_letto = read(sd, request, sizeof(request))) > 0) {
        request[valore_letto] = '\0';
        printf("Ricevuto da %d: %s\n", sd, request);
        gestione_request(sd, request);
}
```

Il thread prova a leggere dal socket tramite la chiamata read(). Se la lettura produce un valore positivo, significa che è stata ricevuta una richiesta HTTP. Il thread allora termina la stringa letta aggiungendo il carattere \0, in modo da poterla trattare come stringa C, successivamente, la richiesta viene passata alla funzione gestione_request(), che si occupa di analizzarla, interpretare il metodo HTTP e preparare la risposta.

4.8.2 Gestione della disconnessione del client

```
else if (valore_letto == 0) {
    printf("Client disconnesso, socket FD=%d\n", sd);
    epoll_ctl(epollfd, EPOLL_CTL_DEL, sd, NULL);
    close(sd);
}
```

La seconda condizione gestisce il caso in cui read() restituisca esattamente zero. Questo valore indica che il client ha chiuso la connessione in maniera ordinata. È importante sottolineare che il thread arriva in questo punto perché è stato epoll stesso a notificare l'evento, tipicamente con il flag EPOLLIN: quando la connessione viene chiusa dal client, epoll segnala comunque "dati disponibili", ma la lettura restituisce zero. In altre parole, è epoll a svegliare il thread, e read=0 conferma che l'evento era una chiusura del socket, non una richiesta valida.

4.8.3 Gestione dell'errore

Infine:

```
    else {
        perror("ERRORE");
    }
    free(arg);
    pthread_exit(NULL);
}
```

Se read() restituisce un valore negativo, significa che si è verificato un errore di comunicazione. Successivamente il socket descriptor che era stato allocato dinamicamente nel thread principale (con malloc) viene eliminato e il thread termina

4.9 File http.c

Il file http.c rappresenta la parte principale del server, dove vengono analizzate le richieste HTTP provenienti dai client e viene costruita la risposta da inviare.

4.9.1 Funzione gestione_request()

```c
void gestione_request(int client_fd, const char *request) {
    http_request_t req;
    if (parse_http_request(request, &req) < 0) {
        send_bad_request(client_fd);
        return;
    }
    // Controllo header Host (obbligatorio HTTP/1.1)
    const char *host = get_header_value(&req, "Host");
    if (!host) {
        send_bad_request(client_fd);
        return;
    }
    gestione_metodo(client_fd, &req);

}
```

Viene richiamata dal file gestione_client.c, all'interno del thread che si occupa della comunicazione con un client specifico. Il thread riceve la richiesta completa tramite recv() e poi passa la stringa a gestisci_http_request() per l'analisi e la risposta. Accetta in ingresso i parametri:

- `client_fd:` file descriptor della richiesta ricevuta dal client.
- `request:` è la stringa completa letta dal socket.

Viene poi dichiarata la variabile req di tipo http_request_t, questa struttura, definita in http.h, rappresenta il modello interno di una richiesta HTTP.
La prima operazione è il parsing della richiesta, tramite la funzione parse_http _request(); questa funzione analizza la stringa ricevuta e riempie la struttura req con i campi estratti se la richiesa non è corretta, quindi la funzione restituisce -1, il server risponde immediatamente al client con "400 Bad Request" tramite la funzione send_bad_request().
Se la richiesta è valida, viene passata alla funzione gestione_metodo(), che si occupa di instradare la richiesta verso la funzione corrispondente al metodo HTTP

4.9.2 Funzione parse_http_request()

Questa funzione si occupa del parsing della richiesta HTTP ricevuta dal client, cioè prende la stringa della rischiesta e la scompone nei vari campi: metodo, percorso, versione, header e body. Il risultato viene memorizzato nella struttura http_request_t passata per riferimento.

```c
int parse_http_request(const char *request, http_request_t *req){
    memset(req, 0, sizeof(http_request_t));                    //
    pulizia
    //Estrai la prima linea (es. "GET /index.html HTTP/1.1")
    //CONTROLLARE SE BAD REQUEST
    if (sscanf(request, "%s %s %s", req->method, req->path, req->
    version) != 3) return -1;

```

```
    //Controllo metodo valido in caso bad request
    if (strcmp(req->method, "GET") != 0 &&
        strcmp(req->method, "POST") != 0 &&
        strcmp(req->method, "PUT") != 0 &&
        strcmp(req->method, "DELETE") != 0) {
        return -1; // Metodo non supportato
    }

    //Controllo path valido
    if (req->path[0] != '/')
        return -1; // Path non valido


    //strstr restituisce l'indirizzo della prima posizione di
    inizio sotto-stringa
    const char *header_start = strstr(request, "\r\n");
    if (!header_start) return -1;
    //avanziamo di 2 per saltare la new line e l'andata a capo
    header_start += 2;
    const char *ptr = header_start;

    // Cicla sugli header fino a \r\n\r\n
    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;
    }


    // Cerca Content-Length (se esiste)
    int content_length = 0;
    const char *cl = get_header_value(req, "Content-Length");
    if (cl)
        content_length = atoi(cl); // sicuro, converte da stringa a
     int


    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';  // lo aggiungiamo noi
        req->body_length = content_length;
    }

    return 0;
}
```

Estrazione della prima riga della richiesta

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

La prima riga di una richiesta HTTP ha sempre questa forma:

```
GET /index.html HTTP/1.1
```

Con sscanf() vengono estratti i tre elementi principali:

- `req->method`: il metodo (GET, POST, PUT, DELETE).
- `req->path`: il percorso della risorsa richiesta.
- `req->version`: la versione del protocollo.

Se non vengono letti esattamente tre token, la richiesta è malformata quindi viene restituito -1, e il server risponderà con 400 Bad Request.

Validazione del metodo

```
if (strcmp(req->method, "GET") != 0 &&
    strcmp(req->method, "POST") != 0 &&
    strcmp(req->method, "PUT") != 0 &&
    strcmp(req->method, "DELETE") != 0) {
    return -1; // Metodo non supportato
}
```

Qui si controlla che il metodo estratto sia uno di quelli effettivamente gestiti dal server, altrimenti la funzione restituirà -1 e quindi il server risponderà con 400 Bad Request.

Validazione del percorso

```
if (req->path[0] != '/')
    return -1;
```

Un percorso HTTP valido deve sempre iniziare con uno slash (/).

Identificazione degli header e ciclo di lettura degli header

```
const char *header_start = strstr(request, "\r\n");
if (!header_start) return -1;
header_start += 2;
const char *ptr = header_start;

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;
}
```

La prima riga trova l'inizio della sezione degli header HTTP all'interno della stringa request.

- strstr(request, "\r\n"): Cerca nella stringa request la prima occorrenza di \r\n, che corrisponde alla fine della prima riga: "GET /index.html HTTP/1.1", serestituisce NULL; significa che la richiesta non contiene nemmeno un ritorno a capo: è malformata, quindi si ritorna -1.
- `header_start += 2:` Serve a spostare il puntatore avanti di due caratteri, saltando i simboli \r\n, header_start quindi punta esattamente all'inizio del primo header
- `const char *ptr = header_start:` crea un nuovo puntatore ptr che verrà usato nel ciclo successivo per scorrere uno a uno gli header presenti.

Il ciclo scorre tutta la sezione degli header fino a quando incontra una riga vuota (\r\n\r\n), che separa gli header dal corpo del messaggio.
Per ogni riga:

- `sscanf():` divide l'header in nome e valore.
- li copia nella struttura req-¿headers[].
- incrementa header_count.

Lettura del Content-Length e del corpo della richiesta

```
int content_length = 0;
const char *cl = get_header_value(req, "Content-Length");
if (cl)
    content_length = atoi(cl);
```

Dopo aver salvato tutti gli header, la funzione cerca l'header Content-Length, che specifica la lunghezza del corpo del messaggio. Se è presente, viene convertito in intero tramite atoi() e usato per sapere quanti byte leggere dopo gli header.

Lettura del body

```
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;
}
return 0;
```

strstr() trova la posizione in cui finisce la sezione header (`\r\n\r\n`) e da lì in poi inizia il body, se il body esiste:

- viene allocata dinamicamente la memoria necessaria
- il contenuto viene copiato dentro req-¿body
- viene aggiunto il terminatore `\0` per poterlo gestire come stringa
- viene salvata la lunghezza effettiva in req-¿body_length.

La funzione alla fine restituisce 0 se tutto il parsing è andato a buon fine

4.10 Funzione gestione_metodo()

Questa funzione viene richiamata alla fine di gestione_request(), dopo che la richiesta è stata validata e parsata correttamente. Il suo scopo è analizzare il metodo HTTP (GET, POST, PUT, DELETE) e chiamare il blocco di codice appropriato per elaborare la richiesta.

Inizio funzione e variabili locali

```
void gestione_metodo(int client_fd, http_request_t* req ){
    char filepath[MAX_PATH_LEN];
```

Qui viene dichiarato filepath, una stringa dove verrà costruito il percorso locale del file richiesto dal client, utilizzando la funzione build_path().
Fatto ciò la funzione controlla, tramite una serie di if, il tipo di metodo invocato

Gestione del metodo GET

```
if (strcmp(req->method, "GET") == 0) {
    build_path(filepath, sizeof(filepath), req->path);

    if (risorsa_protetta(req->path)) {
        if (!autorizzazione_valida(req)) {
            send_unauthorized(client_fd);
            return;
        }
    }

    char* content_body = get_html_file(filepath);
    send_ok(client_fd, content_body);
    free(content_body);
```

Dopo aver generato il percorso locale con build_path(), il server controlla se la risorsa richiesta si trova in una zona protetta, cioè nella directory /private/. In tal caso, viene verificata la presenza di un header di autenticazione valido,

tramite la funzione autorizzazione_valida(). Se il token non è corretto o l'header manca, il server interrompe l'elaborazione e risponde con 401 Unauthorized. Se invece la richiesta è autorizzata, la funzione richiama get_html_file() che legge il contenuto del file richiesto e lo restituisce al client, racchiuso in una risposta 200 OK.

Gestione del metodo POST

```
} else if (strcmp(req->method, "POST") == 0) {
    const char *ctype = get_header_value(req, "Content-Type");
    const char *cl = get_header_value(req, "Content-Length");
    if (!ctype || !cl) {
        send_bad_request(client_fd);
        return;
    }

    if (risorsa_protetta(req->path)) {
        if (!autorizzazione_valida(req)) {
            send_unauthorized(client_fd);
            return;
        }
    }

    build_path(filepath, sizeof(filepath), req->path);

    if (!strstr(filepath, ".txt")) {
        send_bad_request(client_fd);
        return;
    }

    if (access(filepath, F_OK) == 0) {
        send_conflict(client_fd);
        return;
    }

    int fd = open(filepath, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        perror("Errore apertura file POST");
        send_response(client_fd, "500 Internal Server Error", "text
    /plain", "Impossibile creare file\n");
        return;
    }

    write(fd, req->body, req->body_length);
    close(fd);
    send_created(client_fd);
```

Il metodo POST serve per creare una nuova risorsa sul server. Prima di procedere vengono controllati gli header Content-Type e Content-Length header perché sono obbligatori secondo lo standard HTTP/1.1 ogni volta che una richiesta contiene un corpo, se mancano il server manda 401.
Se il file richiesto è protetto, viene verificata l'autorizzazione. Vengono accettati solo file di tipo .txt e se il file esiste già, viene restituito 409 Conflict.
Altrimenti il file viene creato, scritto con i dati ricevuti nel body e, se tutto va

a buon fine, il server risponde con 201 Created. In caso di errore in apertura, viene inviato 500 Internal Server Error.

Gestione del metodo PUT

```
} else if (strcmp(req->method, "PUT") == 0) {
    pthread_mutex_lock(&mutex_globale);
    const char *ctype = get_header_value(req, "Content-Type");
    if (!ctype) {
        send_bad_request(client_fd);
        pthread_mutex_unlock(&mutex_globale);
        return;
    }

    if (risorsa_protetta(req->path)) {
        if (!autorizzazione_valida(req)) {
            send_unauthorized(client_fd);
            pthread_mutex_unlock(&mutex_globale);
            return;
        }
    }

    build_path(filepath, sizeof(filepath), req->path);

    if (!strstr(filepath, ".txt")) {
        send_bad_request(client_fd);
        pthread_mutex_unlock(&mutex_globale);
        return;
    }

    int scelta = 1;
    if (access(filepath, F_OK) == 0)
        scelta = 0;

    int fd = open(filepath, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        perror("Errore apertura file PUT");
        send_response(client_fd, "500 Internal Server Error", "text
    /plain", "Errore creazione o aggiornamento file\n");
        pthread_mutex_unlock(&mutex_globale);
        return;
    }

    ssize_t n = write(fd, req->body, req->body_length);
    if (n == -1) {
        perror("write failed");
        send_response(client_fd, "500 Internal Server Error", "text
    /plain", "Errore di scrittura nel file\n");
    } else {
        if (scelta == 1)
            send_created(client_fd);
        else
            send_response(client_fd, "200 OK", "text/plain", "File
    aggiornato con successo\n");
    }

```

```
49      pthread_mutex_unlock(&mutex_globale);
50      close(fd);
```

Il metodo PUT viene utilizzato per aggiornare o sovrascrivere una risorsa esistente. Essendo un'operazione di scrittura, viene utilizzato il mutex globale per evitare che più thread accedano contemporaneamente allo stesso file. Il server verifica se la risorsa esiste già:

- se non esiste il server restituisce 201 Created
- se esiste già il server restituisce 200 ok e lo sovrascrive.

In caso di errore di apertura o scrittura, il server restituisce 500 Internal Server Error.
Alla fine, il mutex viene sempre rilasciato con pthread_mutex_unlock().

Gestione del metodo DELETE

```
1  } else if (strcmp(req->method, "DELETE") == 0) {
2      pthread_mutex_lock(&mutex_globale);
3
4      if (risorsa_protetta(req->path)) {
5          if (!autorizzazione_valida(req)) {
6              send_unauthorized(client_fd);
7              pthread_mutex_unlock(&mutex_globale);
8              return;
9          }
10     }
11
12     build_path(filepath, sizeof(filepath), req->path);
13
14     if (unlink(filepath) == 0) {
15         send_no_content(client_fd);
16     } else {
17         perror("Errore DELETE");
18         send_response(client_fd, "404 Not Found", "text/plain", "
       File non trovato o impossibile da eliminare\n");
19     }
20
21     pthread_mutex_unlock(&mutex_globale);
```

Il metodo DELETE permette di eliminare una risorsa dal server, anche qui viene usato il mutex per garantire accesso esclusivo durante la cancellazione.
Se il file si trova in una directory protetta, viene richiesto il token di autorizzazione. Se la cancellazione va a buon fine, il server invia 204 No Content; in caso di errore o file mancante, risponde con 404 Not Found.

Metodo non riconosciuto

```
1  } else {
2      send_bad_request(client_fd);
3  }
```

Se il metodo specificato nella richiesta non corrisponde a nessuno dei quattro gestiti, la funzione risponde con 400 Bad Request.

4.11 Funzione get_header_value()

```
1
2 //restituisce l'header richiesto come parametro
3 const char *get_header_value(http_request_t *req, const char *
      header_name) {
4     for (int i = 0; i < req->header_count; i++) {
5         if (strcasecmp(req->headers[i].name, header_name) == 0) //
      ignora differenze tra maiuscolo e minuscolo (RFC lo dice
      chiaramente)
6             return req->headers[i].value;
7     }
8     return NULL; // non trovato
9 }
```

Questa funzione:

- Scorre tutti gli header della richiesta.
- Confronta ciascun nome con quello cercato, ignorando maiuscole/minuscole.
- Restituisce il valore del primo header che corrisponde.
- Se non trova nulla, ritorna NULL.

4.12 Funzione build_path()

```
1
2 void build_path(char *dest, size_t size, const char *req_path) {
3     if (strcmp(req_path, "/") == 0)
4         snprintf(dest, size, "./www/index.html");
5     else
6         snprintf(dest, size, "./www%s", req_path);
7 }
```

La funzione:

- Se la richiesta è "/", costruisce il percorso dell'index.
- Altrimenti, aggiunge req_path dopo "./www".
- Scrive il risultato in dest in modo sicuro.

4.13 Funzione risorsa_protetta()

```
1
2 //controlla se la risorsa e' protetta
3 int risorsa_protetta(const char *path) {
4     // Tutti i file nella cartella /private/ richiedono
      autorizzazione
5     if (strstr(path, "/private/") != NULL)
6         return 1;
7     return 0; // tutto il resto e' libero
8 }
```

Questa funzione Controlla se il percorso richiesto contiene "/private/". Nel caso lo contenga restituisce 1 per indicare risorsa proteta, altrimenti 0.

4.14 Funzione autorizzazione_valida()

```
1
2  // Controlla se l'header Authorization contiene il token corretto
3  int autorizzazione_valida(http_request_t *req) {
4      const char *auth = get_header_value(req, "Authorization");
5      if (!auth) return 0; // manca header
6
7      const char *prefix = "Bearer ";
8      size_t plen = strlen(prefix);
9
10     // Deve iniziare con "Bearer "
11     if (strncasecmp(auth, prefix, plen) != 0)
12         return 0;
13
14     // Estraggo il token vero e proprio
15     const char *token = auth + plen;
16
17     // Confronto col token che abbiamo deciso di accettare
18     if (strcmp(token, "secret123") == 0)
19         return 1; // valido
20     else
21         return 0; // sbagliato
22 }
```

- Legge l'header Authorization.
- Controlla che inizi con "Bearer ".
- Estrae il token dalla stringa.
- Confronta il token con quello corretto ("secret123").
- Restituisce 1 se valido, 0 altrimenti.

4.15 Funzione get_html_file

La funzione get_html_file() ha il compito di aprire un file presente sul server, leggerne il contenuto e restituirlo come stringa pronta per essere inviata al client.

```
1  char *get_html_file(const char* path) {
2      int fd;
3      char *body;
4      int bytes_read, total_read = 0;
5
6      fd = open(path, O_RDONLY);
```

Il file viene aperto in modalità sola lettura (O_RDONLY). Se l'apertura fallisce la funzione restituisce una stringa HTML che rappresenta una semplice pagina 404:

```
if (fd < 0) {
    return strdup("<html><body><h1>404 Not Found</h1></body></html>
    ");
}
```

Se invece il file viene aperto correttamente, viene allocata dinamicamente una porzione di memoria dove verrà copiato il contenuto. La dimensione massima è definita da MAX_FILE_SIZE, per evitare che un file troppo grande occupi troppa memoria.

```
    body = malloc(MAX_FILE_SIZE + 1);
```

A questo punto inizia la lettura del file: viene eseguito un ciclo che legge progressivamente blocchi di dati dal file fino a raggiungere la dimensione massima o fino a quando non ci sono più byte da leggere.

```
while (total_read < MAX_FILE_SIZE) {
    bytes_read = read(fd, body + total_read, MAX_FILE_SIZE -
    total_read);
    if (bytes_read <= 0) break;
    total_read += bytes_read;
}
close(fd);
```

Ogni lettura incrementa il contatore total_read, che tiene traccia del numero totale di byte effettivamente copiati. Al termine, il file viene chiuso con close(fd) per liberare la risorsa. Infine, viene aggiunto manualmente il terminatore di stringa '\0' per rendere il contenuto utilizzabile come normale stringa C:

```
    body[total_read] = '\0';
    return body;
```

La funzione restituisce il body

4.16 Invio Response

Adesso vediamo l'implementazione per mandare response HTTP ai client che hanno mandato una request.

4.16.1 Funzione Principale

```

void send_response(int client_fd, const char *status, const char *
    content_type, const char *body) {
    char response[1024];
    size_t body_len = body ? strlen(body) : 0;

    snprintf(response, sizeof(response),
             "HTTP/1.1 %s\r\n"
             "Content-Type: %s\r\n"
             "Content-Length: %zu\r\n"
             "Connection: close\r\n\r\n"
             "%s",
             status,
             content_type,
```

```
14              body_len,
15              body ? body : "");
16
17      send(client_fd, response, strlen(response), 0);
18  }
```

Questa funzione costruisce e invia una risposta HTTP completa al client sulla base dei parametri che gli arrivano.

- `client_fd`: il file descriptor della connessione TCP con il client.
- `status`: lo status code HTTP (es. "200 OK", "404 Not Found").
- `content_type`: tipo di contenuto della risposta (es. "text/plain", "text/html").
- `body`: corpo del messaggio, cioè il contenuto che vogliamo inviare.

4.16.2 Funzioni di Stato

Queste funzioni chiamano semplicemente `send_response()` con status code e messaggi standard, tranne il codice 200 che accetta un body specifico in base alla richiesta che è stata ricevuta:

```
1
2   void send_bad_request(int client_fd) {
3       send_response(client_fd, "400 Bad Request", "text/plain", "400
        Bad Request\n");
4   }
5
6   void send_unauthorized(int client_fd) {
7       send_response(client_fd, "401 Unauthorized", "text/plain", "401
         Unauthorized\n");
8   }
9
10  void send_forbidden(int client_fd) {
11      send_response(client_fd, "403 Forbidden", "text/plain", "403
        Forbidden\n");
12  }
13
14  void send_created(int client_fd) {
15      send_response(client_fd, "201 Created", "text/plain", "201
        Created\n");
16  }
17
18  void send_ok(int client_fd, const char *body) {
19      send_response(client_fd, "200 OK", "text/html", body);
20  }
21
22  void send_no_content(int client_fd) {
23      send_response(client_fd, "204 No Content", "text/plain", "");
24  }
25
26  void send_conflict(int client_fd) {
27      send_response(client_fd, "409 Conflict", "text/plain", "409
        Conflict: resource already exists\n");
28  }
```

- `send_response`: è la funzione principale che costruisce e invia la response HTTP.
- Le altre funzioni come (send_bad_request, send_ok, ecc.) sono scorciatoie per inviare risposte comuni senza riscrivere sempre tutti i dettagli.

5 PRESENTAZIONE RISULTATI

I risultati ottenuti sono i seguenti:

5.1 Risultati Ottenuti Contattando il Server Tramite Browser

Abbiamo effettuato due richieste GET tramite i seguenti link:
http://127.0.0.1:8080/

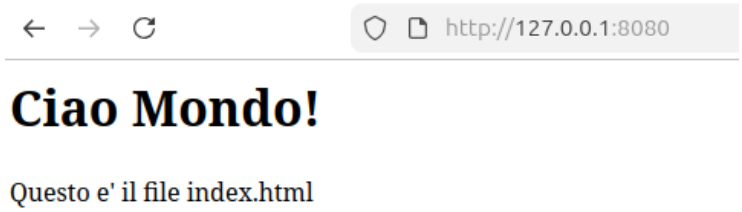

Figure 1: Risultato ottenuto

Lato Server:

```
Server in ascolto sulla porta 8080...
Nuova connessione, socket fd = 5
Ricevuto da 5: GET / HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:145.0) Gecko
    /20100101 Firefox/145.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q
    =0.8
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br, zstd
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
Priority: u=0, i
```

```
16
17
18 Client disconnesso, socket FD=5
```

http://127.0.0.1:8080/private/segreto.txt

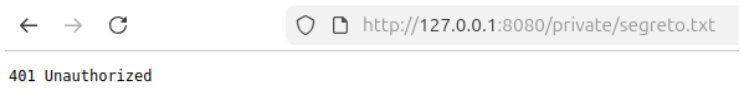

Figure 2: Risultato ottenuto

Lato Server:

```
1  Nuova connessione, socket fd = 5
2  Ricevuto da 5: GET /private/segreto.txt HTTP/1.1
3  Host: 127.0.0.1:8080
4  User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:145.0) Gecko
       /20100101 Firefox/145.0
5  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q
       =0.8
6  Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
7  Accept-Encoding: gzip, deflate, br, zstd
8  Connection: keep-alive
9  Upgrade-Insecure-Requests: 1
10 Sec-Fetch-Dest: document
11 Sec-Fetch-Mode: navigate
12 Sec-Fetch-Site: none
13 Sec-Fetch-User: ?1
14 Priority: u=0, i
15
16
17 Client disconnesso, socket FD=5
```

Proviamo con una risorsa inesistente

Figure 3: Risultato ottenuto

5.2 Risultati Ottenuti Utilizzando curl

cURL (client URL), è uno strumento a riga di comando che permetter di trasferire dati al server usando vari protocolli di rete.
Nel nostro caso utilizziamo HTTP.

curl metodo GET

Proviamo a mandare una richiesta GET per ottenere il file index.

```
curl http://127.0.0.1:8080/
<h1>Ciao Mondo!</h1>
```

Lato Server:

```
Nuova connessione, socket fd = 5
Ricevuto da 5: GET / HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: curl/8.5.0
Accept: */*


Client disconnesso, socket FD=5
```

I risultati mostrano il corretto funzionamento dell'operazione.

curl metodo POST

Mandiamo una richiesta POST per la creazione di un file

```
curl -X POST http://localhost:8080/nuovofile.txt \
     -H "Host: localhost" \
     -H "Content-Type: text/plain" \
     -d "Questo e' il contenuto del nuovo file creato via POST."
201 Created
```

Lato Server:

```
Nuova connessione, socket fd = 5
Ricevuto da 5: POST /nuovofile.txt HTTP/1.1
Host: localhostIl progetto ha permesso di realizzare con successo
    un Server HTTP concor-
rente in linguaggio C, in grado di gestire le principali operazioni
     previste dal
protocollo HTTP - GET, POST, PUT e DELETE
User-Agent: curl/8.5.0
Accept: */*
Content-Type: text/plain
Content-Length: 54

Questo e' il contenuto del nuovo file creato via POST.
Client disconnesso, socket FD=5
```

Andando a verificare nella cartella www, il file è stato correttamentre creato con il contenuto del body. I risultati mostrano il corretto funzionamento dell'operazione.
Proviamo a mandare una richiesta POST su un file già esistente.

```
curl -X POST http://localhost:8080/nuovofile.txt \
     -H "Host: localhost" \
     -H "Content-Type: text/plain" \
     -d "prova POST su un fie gia' esistente"
409 Conflict: resource already exists
```

Lato Server:

```
Nuova connessione, socket fd = 5
Ricevuto da 5: POST /nuovofile.txt HTTP/1.1
Host: localhost
User-Agent: curl/8.5.0
Accept: */*
Content-Type: text/plain
Content-Length: 35

prova POST su un fie gia' esistente
Client disconnesso, socket FD=5
```

I risultati mostrano il successo dell'operazione. Il server non permette l'operazione con POST per la modifica/sovrascrittura.

curl metodo PUT

Adesso utilizziamo PUT per sovrascrivere/modificare il file esistente.

```
curl -X PUT http://localhost:8080/nuovofile.txt \
     -H "Host: localhost" \
     -H "Content-Type: text/plain" \
     -d "Questo file e' stato creato o aggiornato con PUT"
File aggiornato con successo
```

Lato Server:

```
Nuova connessione, socket fd = 5
Ricevuto da 5: PUT /nuovofile.txt HTTP/1.1
Host: localhost
User-Agent: curl/8.5.0
```

```
Accept: */*
Content-Type: text/plain
Content-Length: 48

Questo file e' stato creato o aggiornato con PUT
Client disconnesso, socket FD=5
```

I risultati mostrano il successo dell'operazione. Il file aggiornato è presente nella cartella www.

curl metodo DELETE

Mandiamo una richiesta di tipo DELETE per eliminare il file:

```
curl -X DELETE http://localhost:8080/nuovofile.txt \
     -H "Host: localhost"
```

Lato server:

```
Nuova connessione, socket fd = 5
Ricevuto da 5: DELETE /nuovofile.txt HTTP/1.1
Host: localhost
User-Agent: curl/8.5.0
Accept: */*
Authorization: Bearer secret123


Client disconnesso, socket FD=5
```

Il file è stato eliminato con successo, adesso riproviamo ad eliminare il file non più presente:

```
curl -X DELETE http://localhost:8080/nuovofile.txt \
     -H "Host: localhost"
File non trovato o impossibile da eliminare
```

Lato server:

```
Nuova connessione, socket fd = 5
Ricevuto da 5: DELETE /nuovofile.txt HTTP/1.1
Host: localhost
User-Agent: curl/8.5.0
Accept: */*


Errore DELETE: No such file or directory
Client disconnesso, socket FD=5
```

5.3 comandi curl su file privati

Adesso lanciamo i comandi curl su risorse private per analizzare il sistema di autenticazione.

curl metodo GET senza secret-key

```
curl http://localhost:8080/private/segreto.txt
401 Unauthorized
```

Lato Server:

```
Nuova connessione, socket fd = 5
Ricevuto da 5: GET /private/segreto.txt HTTP/1.1
Host: localhost:8080
User-Agent: curl/8.5.0
Accept: */*


Client disconnesso, socket FD=5
```

come previsto il server non ci restituisce la risorsa.

curl metodo GET con secret-key sbagliata

```
curl http://localhost:8080/private/segreto.txt \
     -H "Authorization: Bearer sbagliato"
401 Unauthorized
```

Lato Server

```
Nuova connessione, socket fd = 5
Ricevuto da 5: GET /private/segreto.txt HTTP/1.1
Host: localhost:8080
User-Agent: curl/8.5.0
Accept: */*
Authorization: Bearer sbagliato


Client disconnesso, socket FD=5
```

Anche in questo caso il server non restituisce la risorsa come previsto.

curl metodo GET con secret-key corretta

Adesso richiediamo il file con la secret key.

```
curl http://localhost:8080/private/segreto.txt \
     -H "Authorization: Bearer secret123"
questo e' il contenuto del file e' privato.
```

Lato Server:

```
Nuova connessione, socket fd = 5
Ricevuto da 5: GET /private/segreto.txt HTTP/1.1
Host: localhost:8080
User-Agent: curl/8.5.0
Accept: */*
Authorization: Bearer secret123


Client disconnesso, socket FD=5
```

Siamo riusciti ad ottenerlo correttamente.

curl metodo POST senza secret-key

Proviamo a creare un file nella directory private.

```
    curl -X POST http://localhost:8080/private/creazioneFilePrivato
    .txt \
     -H "Content-Type: text/plain" \
     -H "Content-Length: 12" \
     --data "file segreto"
401 Unauthorized
```

Lato Server:

```
Nuova connessione, socket fd = 5
Ricevuto da 5: POST /private/creazioneFilePrivato.txt HTTP/1.1
Host: localhost:8080
User-Agent: curl/8.5.0
Accept: */*
Content-Type: text/plain
Content-Length: 12

file segreto
Client disconnesso, socket FD=5
```

Il server non permette la creazione senza autorizzazione.

curl metodo POST con secret-key

Adesso proviamo con la secret-key corretta

```
curl -X POST http://localhost:8080/private/creazioneFilePrivato.txt
     \
     -H "Content-Type: text/plain" \
     -H "Content-Length: 12" \
     -H "Authorization: Bearer secret123" \
     --data "file segreto"
201 Created
```

Lato Server:

```
Nuova connessione, socket fd = 5
Ricevuto da 5: POST /private/creazioneFilePrivato.txt HTTP/1.1
Host: localhost:8080
User-Agent: curl/8.5.0
Accept: */*
Content-Type: text/plain
Content-Length: 12
Authorization: Bearer secret123

file segreto
Client disconnesso, socket FD=5
```

Il file è stato correttamente creato nella directory private.
Pertanto abbiamo visto il correttamento funzionamento del nostro server HTTP.

6 Test di Carico

Per testare il corretto funzionamento e le prestazioni del nostro server HTTP, sono stati eseguiti dei test di carico con l'utilizzo del tool `wrk`. uno strumento per stressare ed ottenere un'analisi delle prestazione del nostro servizio. A differenza di strumenti più semplici come `ab`, `wrk` permette di simulare un numero molto elevato di connessioni simultanee grazie al supporto nativo a:

- multithreading;
- I/O non bloccante;
- gestione ottimizzata di socket e connessioni persistenti;
- polling asincrono tramite `epoll`.

sintassi del comando per eseguire il test:

```
wrk -t <threads> -c <connessioni> -d <durata> <URL>
```

dove:

- **-t**: numero di thread del tool che generano richieste;
- **-c**: numero di connessioni HTTP simultanee;
- **-d**: durata complessiva del test.

Il tool fornisce inoltre metriche fondamentali quali latenza media, massima e deviazione standard, throughput (Requests/sec), dati trasferiti e numero di errori di connessione. Tali parametri si sono rilevati fondamentali per valutare la scalabilità, la stabilità e il comportamento del server sotto carico crescente.

6.1 Test 1: Carico Moderato (50 connessioni)

Il primo test è stato eseguito con 4 thread, 50 connessioni simultanee e durata pari a 10 secondi:

```
wrk -t4 -c50 -d10s http://localhost:8080/index.html
```

Risultati ottenuti:

```
Running 10s test @ http://localhost:8080/index.html
  4 threads and 50 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     6.92ms    1.63ms  45.52ms   97.59%
    Req/Sec     1.73k    98.71     1.92k    92.50%
  69072 requests in 10.01s, 9.16MB read
Requests/sec:   6902.68
Transfer/sec:      0.92MB
```

Risultati principali:

- 69072 richieste servite in 10 secondi;

- throughput: **6902.68 req/s**;
- latenza media: **6.92 ms**;
- deviazione standard: 1.63 ms;
- latenza massima: 45.52 ms;
- nessun errore di connessione.

Analisi: Il server dai valori ottenuti mostra un comportamento stabile sotto un carico moderato, dove la latenza rimane contenuta (circa 7 ms), la deviazione standard è bassa e non si registrano errori di comunicazione. Il throughput vicino alle 7000 richieste al secondo dimostra l'efficacia della gestione concorrente tramite `epoll` e thread. Questo test rappresenta il comportamento del sistema in condizioni operative normali, evidenziando una gestione efficiente delle richieste GET.

6.2 Test 2: Carico Estremo (500 connessioni)

Per testare il comportamento del server in condizione di stress alto, è stato eseguito un secondo test con 12 thread e 500 connessioni simultanee con la durata 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
```

Risultati principali:

- 161801 richieste servite;
- throughput: **5385.76 req/s**;
- latenza media: **32.71 ms**;
- deviazione standard: 82.74 ms;
- latenza massima: 1.79 s;
- errori socket: 70 read errors, 38 timeouts.

Analisi: In condizioni di carico estremo la latenza media aumenta sensibilmente rispetto al test precedente e la deviazione standard cresce in modo significativo, indicando risposte meno uniformi. La latenza massima di circa 1.79 secondi.
Gli errori di lettura e i timeout registrati da `wrk` sono normali in condizioni di stress molto elevato, causati dalla chiusura anticipata delle connessioni da parte del client quando i tempi di risposta superano determinate soglie.
Nonostante ciò, il server continua a servire oltre 5300 richieste al secondo senza mai andare in crash, confermando un buon livello di robustezza anche in condizioni estreme.

6.3 Test del metodo PUT

Per valutare le prestazioni del server relativamente alle operazioni di scrittura dove vi è il mutex globale, è stato eseguito un test di carico sul metodo `PUT`.
Il tool `wrk`, utilizzato anche per i test precedenti, non supporta nativamente metodi diversi da `GET`, pertanto è stato necessario creare uno script Lua dedicato. Questo script consente di impostare il metodo HTTP, definire il body della richiesta e configurare i relativi header.

Script Lua utilizzato

```
wrk.method = "PUT"
wrk.body   = "Contenuto di test per il metodo PUT"
wrk.headers["Content-Type"] = "text/plain"
```

Lo script definisce una semplice richiesta PUT con un body testuale e l'header `Content-Type`. Il server, una volta ricevuta la richiesta, procede alla creazione (o sovrascrittura) del file di destinazione nella directory `www/`.

Configurazione del test Il test è stato eseguito con il comando:

```
wrk -t4 -c50 -d10s -s put.lua http://localhost:8080/ciao.txt
```

Risultati ottenuti:

```
    Running 10s test @ http://localhost:8080/ciao.txt
  4 threads and 50 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     6.59ms    3.80ms  43.06ms   66.99%
    Req/Sec     1.85k    96.92     2.16k    75.25%
  73726 requests in 10.00s, 7.95MB read
Requests/sec:   7369.07
Transfer/sec:    813.19KB
```

- **73726**: richieste servite in 10 secondi;
- throughput: **7369.07 req/s**;
- latenza media: **6.59 ms**;

- deviazione standard: 3.80 ms;
- latenza massima: 43.06 ms;
- nessun errore di socket o timeout.

Analisi dei risultati Il comportamento del server in risposta a richieste PUT è estremamente positivo.

- parsing dell'header e del body;
- apertura del file di destinazione;
- eventuale creazione o sovrascrittura del contenuto;
- sincronizzazione tramite `mutex` globale per evitare race condition.

Il throughput di oltre 7300 richieste al secondo risulta addirittura superiore a quello ottenuto nel test GET a 50 connessioni; ciò è attribuibile alla rapidità delle operazioni di scrittura su SSD e al buffering efficiente del kernel Linux.
Inoltre, l'assenza di errori o timeout conferma la stabilità del sistema anche sotto carico moderato di scritture concorrenti.
In sintesi:

- latenza contenuta e stabile;
- throughput elevato e costante;
- assenza di errori di rete o di gestione delle connessioni;
- piena stabilità del sistema nella gestione simultanea di 50 operazioni di scrittura.

6.4 Conclusioni

I test di carico hanno mostrato che il server:

- gestisce in modo eccellente carichi moderati fino a 50 connessioni, con bassa latenza e nessun errore;
- scala fino a centinaia di connessioni mantenendo un throughput elevato;
- rimane stabile anche sotto 500 connessioni simultanee, pur mostrando un aumento della latenza e qualche errore di timeout;
- non presenta deadlock o crash durante i test.

Nel complesso, i risultati evidenziano una buona scalabilità e robustezza dell'implementazione, considerando che il server è stato progettato e sviluppato interamente in linguaggio C senza l'ausilio di framework o ottimizzazioni avanzate.

7 CONCLUSIONE

Il progetto ha permesso di realizzare con successo un Server HTTP concorrente in linguaggio C, in grado di gestire importanti operazioni previste dal protocollo HTTP (GET, POST, PUT e DELETE) e di fornire le risposte corrispondenti secondo quanto stabilito dalla RFC 2616.
Durante la realizzazione del progetto, sono stati affrontati diversi aspetti fondamentali della programmazione di rete, tra cui la gestione dei socket TCP/IP, la sincronizzazione tra thread tramite mutex, la gestione delle code di eventi tramite epoll e la corretta interpretazione dei messaggi HTTP.
In conclusione, il progetto ha raggiunto pienamente gli obiettivi prefissati, fornendo una piattaforma solida e funzionale per la gestione di richieste HTTP concorrenti.
Sebbene il server implementi le funzionalità principali richieste, esistono diverse direzioni di sviluppo futuri, che potrebbero migliorare ulteriormente il sistema o renderlo idoneo a contesti più avanzati come:

- **Supporto HTTPS**: integrazione del layer di sicurezza (OpenSSL) per abilitare connessioni sicure e cifrate.
- **Thread Pool**: sostituzione della creazione dinamica dei thread con un pool predefinito, così da ridurre l'overhead e migliorare la scalabilità.