LabReSiD25: HTTP Server

Marco Iaria, Matricola 557295

November 2025

1 Introduzione

E' possibile unire la tecnologia dei **socket** con quella dei **thread** per programmare un server HTTP concorrente. La presentazione dimostra come sia possibile crearlo con il linguaggio C usando la **programmazione modulare** e `make`.

1.1 Protocollo HTTP

L'**HTTP (Hypertext Transfer Protocol)** è il protocollo applicativo fondamentale su cui si basa la comunicazione dati del World Wide Web. Questo stabilisce le regole per lo scambio di informazioni tra un client (come un browser web) e un server.
Il protocollo opera secondo un modello client-server e utilizza un meccanismo semplice di tipo **richiesta-risposta**: il client invia una richiesta per una risorsa e il server risponde fornendo tale risorsa o un messaggio d'errore. È un protocollo stateless e opera tipicamente sopra il protocollo **TCP/IP**.

2 Definizione del problema

Viene richiesto un **Server HTTP concorrente**, sviluppato in linguaggio C, e conforme ad RFC 2616. Il server deve essere in grado di gestire almeno le richieste `GET, POST, PUT, DELETE` e le risposte principali `200, 201, 204, 400, 401`.
Il server quindi deve essere in grado di accettare connessioni multiple dai client, mantenere le connessioni persistenti, e comprendere le richieste inviate dai client. Un esempio di richiesta HTTP è `GET /index.html HTTP/1.1`.

3 Metodologia

Per garantire una maggiore chiarezza e agevolare lo sviluppo futuro, il codice è stato diviso in 9 file:

- `http_server.h`: File header con librerie e funzioni utilizzate dal server;
- `server.c`: Inizializzazione Socket, gestione client e creazione dei thread;
- `threadpool.h`: File header con librerie, funzioni e strutture utilizzate per gestire il thread pool;
- `threadpool.c`: File contenente le funzioni definite in `threadpool.h`;
- `parse_request.c`: Funzione `parse_request` eseguita da ogni thread per gestire le connessioni di un client;
- `methods.c`: Definizione dei metodi HTTP;
- `status_codes.c`: Definizione di risposte HTTP;
- `utils.c`: Definizione di funzioni utilizzate dai metodi HTTP;
- `authentication.c`: Definizione di funzioni utilizzate per controllare l'autenticazione di un client.

3.1 HTTP Server

Il file `http_server.h` comprende le funzioni e gli status codes definiti dal server, una costante `BUFFER_SIZE` e le librerie essenziali per la corretta esecuzione del codice.

```
1 #include <stdlib.h>
2 #include <string.h>
3 #include <fcntl.h>
4 #include <sys/stat.h>
5 #include <unistd.h>
6 #include <stdio.h>
7 #include <sys/socket.h>
8
9 #define BUFFER_SIZE 104857
```

Le librerie inserite non sono le uniche utilizzate, alcune, come `pthread.h`, sono definite altrove per mantenere il codice più ordinato.

3.2 Threadpool.h

Comprende la struttura `thread_pool_t`, le costanti `MAX_THREADS e MAX_QUEUE_SIZE`, e le intestazioni delle funzioni utilizzate per creare e distruggere un thread pool, e aggiungere un task alla coda.

```
#define MAX_THREADS 4
#define MAX_QUEUE_SIZE 256

// Struttura del Thread Pool
typedef struct {
    pthread_t *threads; // Array di thread
    int *queue[MAX_QUEUE_SIZE]; // Coda dei lavori
    int head; // Testa della coda
    int tail; // Coda della coda
    int task_count; // Numero di lavori in coda
    pthread_mutex_t lock; // Mutex per proteggere la coda
    pthread_cond_t notify; // Variabile di condizione per svegliare
        i thread
    int shutdown; // Flag per terminare il pool
} thread_pool_t;
```

La struttura `thread_pool_t` contiene il riferimento del file descriptor del client all'interno di un array di puntatori `queue`. Contiene anche tutte le informazioni necessarie per la sua corretta implementazione, come specificato dai commenti del codice.
Infine contiene le funzioni specificate precedentemente:

```
// Creazione del Thread Pool
thread_pool_t *thread_pool_create();

// Aggiunta di un task al Thread Pool
int thread_pool_add_task(thread_pool_t *pool, int *client_fd);

// Distruzione del Thread Pool
int thread_pool_destroy(thread_pool_t *pool);

// Funzione thread pool
void parse_request(int client_fd);
```

3.3 Server

Il file `server.c` si occupa dell'inizializzazione del **Thread Pool** e di un **Socket TCP** con dominio IPv4, indirizzo `localhost (127.0.0.1)` e porta 8080.

```
// Inizializzazione Thread Pool
thread_pool_t *pool = thread_pool_create();
if (pool == NULL) {
    perror("Creazione thread pool fallita");
    exit(EXIT_FAILURE);
}

// Inizializzazione server socket
if((server_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
    perror("Creazione socket fallita");
    exit(EXIT_FAILURE);
}

// Configurazione indirizzo
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(PORT);
```

Al socket vengono applicate l'impostazione `SO_REUSEADDR`, che rispettivamente dicono al socket di utilizzare la coppia di indirizzo e porta anche se sono già utilizzati da altri processi.
Questo è necessario per risolvere un bug ricorrente in fase di testing che non permetteva l'avvio e l'arresto repentino del server.

```
setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &(int){1}, sizeof(
    int));
```

3.3.1 Gestione Connessioni

Dopo aver messo il socket in ascolto, il server inizializza epoll, impostando gli `events` in modalità di sola lettura (`EPOLLIN`), e aggiungendo il `file_descriptor` del server a epoll.

```
// Inizializzazione Epoll
if ((epoll_fd = epoll_create1(0)) < 0) {
    perror("Epoll create fallito");
    close(server_fd);
    exit(EXIT_FAILURE);
}

// Aggiungi il server_fd a epoll
ev.events = EPOLLIN;
ev.data.fd = server_fd;
if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, server_fd, &ev) < 0) {
    perror("Epoll ctl (server_fd) fallito");
    close(server_fd);
    close(epoll_fd);
    exit(EXIT_FAILURE);
}
```

Gli eventi ricevuti da epoll vengono gestiti attraverso un loop infinito while. Gli eventi vengono inseriti all'interno di un array `events`, dalla funzione `epoll_wait`, che metterà in attesa il server fino a quando non riceverà una richiesta.

```
while (1){
    int n_events = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
        if (n_events < 0) {
            perror("Epoll wait fallito");
            continue;
        }
```

Ricevuti gli eventi, si usa un ciclo for per iterare su di essi, controllando quale di questi è una nuova connessione.

```
 for (int i = 0; i < n_events; i++) {
        if (events[i].data.fd == server_fd) {
```

Se l'evento è accaduto sul socket (`server_fd`), allora un client sta richiedendo una connessione.

```c
int client_fd = accept(server_fd, (struct sockaddr *)&client_addr,
    &client_addr_len);
if (client_fd < 0) {
    perror("Accept fallito");
    continue;
}
```

Viene accettata la connessione e viene aggiunto il nuovo client a epoll, esattamente come fatto con il server.

```c
ev.events = EPOLLIN;
ev.data.fd = client_fd;
if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, client_fd, &ev) < 0) {
    perror("Epoll ctl (client_fd) fallito");
    close(client_fd);
}
```

Tuttavia, se l'evento avviene su un file descriptor che non è del socket, una richiesta HTTP è pronta per essere elaborata.

```c
 // Recuperiamo il file descriptor del client
int client_fd = events[i].data.fd;

// Per evitare che questo evento venga processato piu volte lo
    rimuoviamo da epoll.
epoll_ctl(epoll_fd, EPOLL_CTL_DEL, client_fd, NULL);

// Allochiamo memoria per passare il client_fd al worker
int *client_fd_ptr = malloc(sizeof(int));
if (client_fd_ptr == NULL) {
    perror("malloc per client_fd_ptr fallito");
    close(client_fd);
    continue;
}
*client_fd_ptr = client_fd;

// Aggiungiamo il lavoro al pool.
thread_pool_add_task(pool, client_fd_ptr);
```

Recuperiamo il file descriptor pronto per essere letto, lo rimuoviamo dal monitoraggio di epoll, in modo che la funzione `epoll_wait` non rilegga di nuovo lo stesso dato, e aggiungiamo il file descriptor alla coda del thread pool.

3.4 Thread Pool

Per evitare di creare e distruggere un nuovo thread per ogni richiesta, creiamo un numero fisso di **thread worker**, che rimangono in attesa di lavoro e non terminano mai. I thread worker accederanno ad una variabile condivisa `queue` per recuperare i file descriptor che necessitano di essere elaborati.

3.4.1 thread_pool_create

La funzione `thread_pool_create` inizializza un **thread pool** di `MAX_THREADS` dimensione con dimensione massima della coda di `MAX_QUEUE_SIZE`.

```
// Creazione del Thread Pool
thread_pool_t *thread_pool_create() {
    // Se i parametri non sono validi, ritorna NULL
    if (MAX_THREADS <= 0 || MAX_QUEUE_SIZE <= 0) return NULL;

    thread_pool_t *pool = (thread_pool_t *)malloc(sizeof(
        thread_pool_t));
    if (pool == NULL) return NULL;

    pool->head = pool->tail = pool->task_count = 0;
    pool->shutdown = 0;
```

Si alloca memoria per la struttura principale che conterrà tutte le informazioni (elenco dei thread, coda dei lavori, mutex ecc.).

```
pool->threads = (pthread_t *)malloc(sizeof(pthread_t) *
    thread_count);
```

Vengono inizializzati il mutex e la condition variable, necessari per gestire la concorrenza dei thread per l'accesso alla coda e per il risveglio di questi quando un nuovo task viene aggiunto.

```
// Inizializza mutex e variabile di condizione
pthread_mutex_init(&(pool->lock), NULL);
pthread_cond_init(&(pool->notify), NULL);
```

Infine vengono creati i thread e posti in stato di `detach` per semplificare la gestione della chiusura.

```
// Avvia i thread worker
for (int i = 0; i < thread_count; i++) {
```

```
    pthread_create(&(pool->threads[i]), NULL, worker_thread, (void
        *)pool);
    pthread_detach(pool->threads[i]); // Li avviamo gia' detached
}
```

3.4.2 worker_thread

Ogni thread creato esegue la funzione `worker_thread` che lo pone in un ciclo infinito di controllo sulla coda.

```
while (1) {
    pthread_mutex_lock(&(pool->lock));

    // Aspetta finche' non c'e' un lavoro o il pool non viene chiuso
    while (pool->task_count == 0 && !pool->shutdown) {
        pthread_cond_wait(&(pool->notify), &(pool->lock));
    }
```

Il thread acquisisce il `lock` e controlla la coda dei task. Se questa è vuota, ma il server non sta terminando la propria esecuzione, il thread chiama `pthread_cond_wait`, che rilascia il `lock` e pone il thread in attesa di essere "svegliato". Una volta risvegliato acquisirà il `lock` in automatico.
Se `shutdown` è uguale ad 1, il thread termina il proprio ciclo di vita.

```
if (pool->shutdown) {
    pthread_mutex_unlock(&(pool->lock));
    pthread_exit(NULL);
}
```

Se il thread trova qualcosa all'interno della coda, recupera il contenuto ed esegue la funzione `parse_request`.

```
// Prendi un lavoro dalla coda
int *client_fd = pool->queue[pool->head];
pool->head = (pool->head + 1) % MAX_QUEUE_SIZE;
pool->task_count--;

pthread_mutex_unlock(&(pool->lock));

// Esegui il lavoro
parse_request(*client_fd);
```

3.4.3 thread_pool_add_task

Per aggiungere un task alla coda del **thread pool**, il server chiama la funzione `thread_pool_add_task`. Queste recupera il riferimento in memoria del file descriptor del client, acquisisce il `lock`, controlla se la coda é piena, aggiunge il `fd` alla coda e sveglia un worker.

```
// Lock della coda
pthread_mutex_lock(&(pool->lock));

if (pool->task_count == MAX_QUEUE_SIZE) {
    // Coda piena
    pthread_mutex_unlock(&(pool->lock));
    return -1;
}

// Aggiungi il task alla coda
    pool->queue[pool->tail] = client_fd;
    pool->tail = (pool->tail + 1) % MAX_QUEUE_SIZE;
    pool->task_count++;

// Sveglia un worker
pthread_mutex_unlock(&(pool->lock));
pthread_cond_signal(&(pool->notify));
```

3.4.4 thread_pool_destroy

Questa funzione indica ai thread di terminare la propria esecuzione e libera la memoria. Per farlo imposta la variabile `shutdown` del pool a 1 e sveglia tutti i thread con `pthread_cond_broadcast`.

```
pthread_mutex_lock(&(pool->lock));
pool->shutdown = 1;
pthread_cond_broadcast(&(pool->notify)); // Sveglia tutti
pthread_mutex_unlock(&(pool->lock));
```

3.5 Parse Request

La funzione `parse_request` imposta un timeout di 5 secondi e alloca un buffer di dimensione `BUFFER_SIZE` per ricevere dati dal client.

```c
void parse_request(int client_fd) {
    // Imposta un Timeout di 5 secondi
    struct timeval tv;
    tv.tv_sec = 5;
    tv.tv_usec = 0;
    setsockopt(client_fd, SOL_SOCKET, SO_RCVTIMEO, (const char*)&tv,
        sizeof tv);

    char *buffer = (char *) malloc(BUFFER_SIZE * sizeof(char));
```

Imposta il keep alive ad 1, come di default in HTTP 1.1, e avvia un loop per ricevere costantemente dati.

```c
int keep_alive = 1; // HTTP 1.1 e' keep-alive di default

// Loop di connessione persistente
while (keep_alive) {
```

Recupera i dati ricevuti sul file descriptor del client e controlla se vuole chiudere la connessione

```c
ssize_t bytes_received = recv(client_fd, buffer, BUFFER_SIZE - 1,
    0);

// Controlla se il client vuole chiudere (Connection: close)
if (strstr(buffer, "Connection: close") != NULL) {
    keep_alive = 0;
}
```

Le richieste vengono analizzate usando una **Regular Expression**.

```c
// Riceve dati dal client
if(recv(client_fd, buffer, BUFFER_SIZE, 0) > 0) {
    // Controlla il metodo HTTP richiesto usando regex
    regex_t regex;
    regcomp(&regex, "([A-Z]+) /([^ ]*)", REG_EXTENDED);
    regmatch_t matches[3];
```

La regex compilata è composta da due gruppi:

- `([A-Z]+)`: Cerca una o più lettere maiuscole (es. `GET`, `POST`, `PUT`, ...);
- `([^ ]*)`: Cerca una sequenza di zero o più caratteri che non sono spazi (es.`index.html`).

Inoltre tra i due gruppi è presente " /" che cerca un carattere letterale spazio e un carattere slash. La regex è composta per analizzare la prima riga di una richiesta HTTP come: `GET /index.html HTTP/1.1`.
La regex una volta compilata viene eseguita all'interno di un `if` per controllare la validità dei dati ricevuti dal client.

```
if (regexec(&regex, buffer, 3, matches, 0) == 0){
```

La funzione `regexec` usa la regex compilata in precedenza per cercare delle corrispondenze in `buffer`. Se trovate vengono salvate massimo 3 corrispondenze:

1. `matches[0]`: Corrispondenza intera (es. `GET /index.html`);
2. `matches[1]`: Il primo gruppo (es. `GET`);
3. `matches[2]`: Il secondo gruppo (es. `/index.html`).

Il metodo HTTP viene estratto dall'array `matches` nella posizione 1.

```
// Estrae il metodo HTTP
size_t method_len = matches[1].rm_eo - matches[1].rm_so;
char method[method_len + 1];
strncpy(method, buffer + matches[1].rm_so, method_len);
method[method_len] = '\0';
```

Viene estratta la lunghezza della stringa trovata attraverso gli **Offset** `.rm_so` (`Regular Match Start Offset`) e `.rm_eo` (`Regular Match End Offset`). La stringa viene copiata dal buffer verso una array di caratteri `method`, indicando come sorgente il buffer + offset di inizio del metodo.
Infine si aggiunge il terminatore `\0` per terminare correttamente la stringa e renderla valida. Il nome del file viene estratto similmente al metodo dall'array `matches` nella posizione 2.

```
// Estrae il nome del file dalla request
size_t path_len = matches[2].rm_eo - matches[2].rm_so;
char url_encoded_file_name[path_len + 1];
strncpy(url_encoded_file_name, buffer + matches[2].rm_so, path_len)
    ;
url_encoded_file_name[path_len] = '\0';

char *file_name = url_decode(url_encoded_file_name);
```

Questo blocco utilizza la stessa logica del precedente, con gli **Offset** utilizzati per estrarre la lunghezza della stringa e l'inizio del nome del file. Questo blocco tuttavia si differenzia dalla chiamata a funzione `url_decode`, definita in `utils.c`, che decodifica eventuali **caratteri speciali** presenti all'interno del URL. Il metodo recuperato è posto all'interno di diversi if/else per chiamare la funzione specifica della richiesta. Se la richiesta non è compresa tra i quattro metodi sviluppati (`GET, POST, PUT, DELETE`), il server restituisce un errore di tipo **501 Method Not Implemented**.

```
// GET request
if (strcmp(method, "GET") == 0) get(buffer, client_fd, file_name);
// POST request
else if(strcmp(method, "POST") == 0) post(buffer, client_fd,
    file_name);
// PUT request
else if(strcmp(method, "PUT") == 0) put(buffer, client_fd,
    file_name);
// DELETE request
else if(strcmp(method, "DELETE") == 0) delete(buffer, client_fd,
    file_name);
else {
    // Metodo non implementato
    char response[BUFFER_SIZE];
    size_t response_size;
    not_implemented_501(response, &response_size);
    send(client_fd, response, response_size, 0);
}
```

Il ciclo termina con un controllo sulla variabile `keep_alive`.

```
if (!keep_alive) {
        break;
    }
```

3.6 Methods

Ognuno dei quattro metodi effettua operazioni diverse, pertanto sono state definite quattro diverse funzioni:

- `get()`: Recupera il file richiesto e lo invia al client;
- `post()`: Crea o *aggiunge* informazioni presenti nel body della request;
- `put()`: Crea o *sostituisce* informazioni presenti nel body della request;
- `delete()`: Rimuove il file richiesto.

E' stata inserita una cartella privilegiata **root**, accessibile soltanto tramite login con credenziali. Il metodo `GET` è l'unico utilizzabile senza credenziali, qualsiasi altro metodo richiede un'autenticazione anche per operazioni fuori da **root**. Pertanto per ogni metodo verrà chiamata la funzione `check_authentication`:

```
if (check_authentication(client_fd, buffer, file_name, 0) == -1){
    // Non autorizzato, invia 401 Unauthorized
    unauthorized_401(response, &response_size);
    send(client_fd, response, response_size, 0);

    free(response);
    free(file_name);
    return;
}
```

Questa controlla il path della richiesta e la validità delle credenziali. La funzione `GET` è l'unica con **flag 0** come specificato in precedenza.

3.6.1 GET

Il metodo `GET` controlla se il file richiesto si trova dentro la cartella protetta **root** e recupera il file descriptor del file richiesto in modalità di sola lettura. Se la funzione non riesce ad aprire il file questa elabora un errore **404 Not Found** e ritorna `-1` (valore di default di un file descriptor non aperto).

```
// Ottiene il file descriptor del file richiesto
int file_fd;
if ((file_fd = get_file_fd(file_name, response, &response_size)) ==
     -1) {
     // Invia risposta 404 al client
    send(client_fd, response, response_size, 0);

    free(response);
    free(file_name);
    return;
}
```

Se l'operazione è andata a buon fine, copia il contenuto del file all'interno di un buffer temporaneo `body` e invia una risposta **OK 200** al client.

```
// Copia il contenuto del file
char *body = (char *) malloc(BUFFER_SIZE * sizeof(char));
ssize_t bytes_read = read(file_fd, body, BUFFER_SIZE);
if (bytes_read < 0) bytes_read = 0;
response_size = bytes_read;

// Elabora risposta 200 OK
char *file_ext = get_file_extension(file_name);
ok_200(response, &response_size, file_ext, body);
close(file_fd);
```

La struttura `ssize_t` tiene conto dei byte letti dalla funzione read, la funzione read leggerà il contenuto di `file_fd` e lo scriverà in memoria in `body`.

3.6.2 POST

Il metodo post estrae il contenuto del body della request, inviando un errore **400 BAD REQUEST** se vuoto.

```
 // Estrae il body della request
char *body_data;
if((body_data = get_body_data(buffer)) == NULL){
    // Request malformata, invia 400 Bad Request
    bad_request_400(response, &response_size);

    free(response);
    free(file_name);
    return;
}
```

Successivamente ottiene il file descriptor del file richiesto nell'header della richiesta, e lo apre in modalità di scrittura (`O_RDONLY`) con flag `O_CREAT, O_APPEND` per creare il file se non esistente e aprirlo in `append mode`. Il file viene creato con permessi di **lettura/scrittura** per il proprietario e solo **lettura** per gli altri utenti di sistema (`0644`). Se il file descriptor non viene aperto, il server invia un errore **500 INTERNAL SERVER ERROR**.

```
// Ottiene il file descriptor del file richiesto
int file_fd = open(file_name, O_WRONLY | O_CREAT | O_APPEND, 0644);
if (file_fd == -1){
    // Errore nell'aprire/creare il file
    internal_server_error_500(response, &response_size);
    send(client_fd, response, response_size, 0);
}
```

Il contenuto recuperato dal body viene scritto all'interno del file descriptor. Se le operazioni vengono concluse a buon fine, il server elabora una risposta **OK 200**, in caso contrario elabora una risposta **INTERNAL SERVER ERROR 500**.

```
// Scrive / Appendo il body nel file
ssize_t content_length = get_content_length(buffer);
if(write(file_fd, body_data, content_length) == -1)
    internal_server_error_500(response, &response_size);
else {
    char *file_ext = get_file_extension(file_name);
    ok_200(client_fd, file_ext, body_data, content_length);}
```

3.6.3 PUT

Il metodo `PUT` ha un comportamento simile al metodo `POST`, recupera il `body_data` in modo identico, ma prima ancora di aprire il file descriptor del file richiesto ne controlla la possibile esistenza. Se il file esiste ed è identico, la funzione non effettua nessuna operazione, inviando al client una risposta di tipo **204 No Content**. Per gestire questa possibilità sono stati effettuati dei controlli sul nome, dimensione e contenuto del file, e usato un flag `flag_different` per inviare lo `status_code` corretto al client.

```
1  // Controlla se il file esiste gia'
2 int existing_file_fd = get_file_fd(file_name, response, &
     response_size);
3 int flag_different = 0;
4
5 if (existing_file_fd != -1){
```

Il primo confronto avviene sul nome del file, se il file non esiste, la funzione `get_file_fd` restituirà come output `-1` e i successivi controlli verranno saltati immediatamente.
Il secondo confronto avviene sulle dimensioni del file, se queste sono diverse il contenuto sarà di conseguenza differente.

```
1 // Il file esiste, controlla dimensioni del file
2 struct stat file_stat;
3 fstat(existing_file_fd, &file_stat);
4 int content_length = get_content_length(buffer);
5
6 if (file_stat.st_size == content_length){
```

Utilizzando la struttura `stat` e la funzione `fstat()` recuperiamo informazioni sul file, in particolare la dimensione in byte di quest'ultimo, e usando una funzione personalizzata `get_content_length` recuperiamo il numero presente nel campo `Content-Length` all'interno del **header HTTP**.
Se il file ha le stesse dimensioni del contenuto della richiesta, vengono confrontati il contenuto del file e il `body_data`.

```
char *file_ext_data = (char *) malloc(content_length);
read(existing_file_fd, file_ext_data, content_length);

// Confronta il contenuto del file con il body della request
if (memcmp(file_ext_data, body_data, content_length) == 0)
    flag_different = 1; // I file sono identici
```

Il blocco termina dopo aver comparato le due aree di memoria con la funzione `memcmp`. Viene controllato il flag creato in precedenza:

```
if (flag_different){
    // I file sono identici, invia 204 No Content
    no_content_204(response, &response_size);
    send(client_fd, response, response_size, 0);

    free(body_data);
    free(response);
    free(file_name);
    return;
}
int file_fd = open(file_name, O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

Se il file è diverso, quel blocco verrà saltato, aprendo il file descriptor in modalità di scrittura con flag `O_CREAT, O_TRUNC`, che azzererà il contenuto del file prima di scriverci nuovi dati al suo interno.

```
int file_fd = open(file_name, O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (file_fd == -1){
    // Errore nell'aprire/creare il file
    internal_server_error_500(response, &response_size);
    send(client_fd, response, response_size, 0);

    free(body_data);
    free(response);
    free(file_name);
    return;
}
```

La funzione termina esattamente come `POST`, scrivendo all'interno del file descriptor e inviando una risposta **201**.

3.6.4 DELETE

Il metodo `DELETE` controlla l'esistenza del file presente nel `URL` della richiesta, lo elimina con `remove()` e invia una risposta **204 NO CONTENT** al client. Se il file specificato non esiste, invia la risposta **404 Not Found**; e se non riesce a rimuoverlo, invia **500 Internal Server Error**.

```
// Controlla se il file esiste
if(get_file_fd(file_name, response, &response_size) == -1) {
    send(client_fd, response, response_size, 0);

    free(response);
    free(file_name);
    return;
}

// Prova a eliminare il file
if (remove(file_name) != 0){
    // Se fallisce, elabora risposta Internal Server Error 500
    internal_server_error_500(response, &response_size);
else{
    // Elabora risposta 204 No Content
    char *file_ext = get_file_extension(file_name);
    no_content_204(response, &response_size);
}
send(client_fd, response, response_size, 0);
```

La funzione `get_file_fd` viene utilizzata con l'unico obiettivo di controllare se il file esiste. Il file descriptor non viene recuperato e non è necessario per rimuovere il file.

3.7 Utils

Il file `utils.c` contiene funzioni utili, necessarie per la corretta implementazione dei metodi HTTP.

3.7.1 url_decode

La funzione `url_decode` viene utilizzata dal server per decodificare eventuali caratteri speciali presenti all'interno dell'URL di una richiesta HTTP. La funzione cerca all'interno della stringa una sequenza formata da `"% + due caratteri"`, legge i due caratteri dopo il `%` e li converte da esadecimale a numero decimale. Il numero decimale recuperato viene convertito nel carattere ASCII corrispondente e aggiunto alla stringa `decoded`. Il contatore effettua un `+= 2` per saltare i due caratteri convertiti

```
// Decodifica da %2x a esadecimale
for(size_t i = 0; i < src_len; i++){
    if(src[i] == '%' && i + 2 < src_len){
        int hex_val;
        sscanf(src + i + 1, "%2x", &hex_val);
        decoded[decoded_len++] = hex_val;
        i += 2;
    }else decoded[decoded_len++] = src[i];
}
```

3.7.2 get_file_fd

La funzione `get_file_fd` ritorna il `file_descriptor` del file richiesto in modalità di sola lettura `O_RDONLY`. Se l'operazione non viene effettuata correttamente elabora una risposta **404 NOT FOUND**.

```
int file_fd = open(file_name, O_RDONLY);
if(file_fd == -1){
    // Se il file non esiste, prepara la risposta 404
    not_found_404(response, response_size);
}

return file_fd;
```

3.7.3 get_content_length

Questa funzione cerca l'occorrenza `"Content-Length: "`, recupera la posizione del valore di `Content-Length` e lo converte in un numero intero con la funzione `atoi()`.

```
const char *header_key = "Content-Length: ";
char *header_line = strstr(buffer, header_key);

if(header_line){
    const char *value_start = header_line + strlen(header_key);
    return atoi(value_start);
}
return -1;
```

3.7.4 find_body_start

Questa funzione cerca il separatore `"\r\n\r\n"`, presente di default in una request HTTP, che separa il contenuto del **Header** dal contenuto del **Body** e ritorna un puntatore all'inizio di questo.

```
// Cerca la sequenza di fine header HTTP
const char* separator = "\r\n\r\n";
const char* body_start = strstr(buffer, separator);

// Restituisce il puntatore all'inizio del corpo della richiesta
if (body_start) {
    return body_start + 4;
}
return NULL;
```

3.7.5 get_body_data

Questa funzione utilizza le precedenti funzioni `get_content_length` e `find_body_start` per recuperare il contenuto del body di una request. Recuperato l'inizio del body e la sua dimensione, chiama la funzione `strncpy` per copiare il contenuto della richiesta in una variabile apposita e ritorna il puntatore ad essa.

```
// Trova l'inizio del body nella request
char *body_start = find_body_start(buffer);
if (body_start == NULL) {
    // Request malformata
    return NULL;
}

// Trova la lunghezza del body
int content_length = get_content_length(buffer);
if (content_length == -1){
    // Request malformata
    return NULL;
}

// Estrae il body della request
char *body_data = (char *) malloc(content_length + 1);
strncpy(body_data, body_start, content_length);
body_data[content_length] = '\0';

return body_data;
```

3.8 Authentication

Il file `authentication.c` contiene la definizione della funzione `check_authentication` e la funzione di decodifica `base64_decode`. Queste due funzioni permettono l'autenticazione necessaria per i metodi `POST, PUT e DELETE`, e per l'accesso alla cartella root.

3.8.1 check_authentication

Questa funzione richiede un flag che serve a marcare i metodi che richiedono autenticazione per accedere a qualsiasi file:

- `1` - Autenticazione obbligatoria;
- `0` - Autenticazione necessaria per accedere alla cartella `/root`.

```
1 if (strncmp(file_name, "root/", 5) != 0 && flag == 0) return 1; //
    Cartella non protetta
2
3 const char *auth_header = strcasestr(buffer, "Authorization:␣Basic␣
    ");
4 if (auth_header == NULL) return -1;
5
6 if (auth_header != NULL){
```

Se il metodo richiesto tenta di accedere alla cartella **root**, oppure richiede credenziali per essere utilizzato, cerca la stringa `Authorization: Basic` contenente le informazioni codificate in **Base64**. Se la stringa è presente vengono effettuate le operazioni di recupero e decodifica delle credenziali.

```
1 const char *base64_creds = auth_header + strlen("Authorization:␣
    Basic␣");
2 char decoded_creds[128];
3
4 char base64_string[128];
5 int i = 0;
6 while (base64_creds[i] != '\r' && base64_creds[i] != '\n' &&
    base64_creds[i] != '\0' && i < 127) {
7     base64_string[i] = base64_creds[i];
8     i++;
9 }
10 base64_string[i] = '\0';
```

Le credenziali vengono recuperate addizionando la lunghezza della stringa `Authorization: Basic` al puntatore `auth_header`, contenente l'indirizzo di inizio riga. Si utilizza un ciclo `while` per iterare sulle credenziali e recuperarle per intero. Si procede chiamando la funzione di decodifica `base64_decode()` e si termina il blocco verificando la validità delle credenziali.

```
if (base64_decode(base64_string, decoded_creds) == -1) return -1;
char expected_creds[128];
snprintf(expected_creds, sizeof(expected_creds), "%s:%s", "root", "
    toor");
if (strcmp(decoded_creds, expected_creds) == 0) return 1;
```

3.8.2 base64_decode

La funzione recupera **4 caratteri Base64**, che utilizzano `24bit (4 * 6)`, e li converte recuperando **3 byte di dati** `(3 * 8)`. Per farlo la funzione dichiara due array, utilizzati per contenere 4 caratteri Base 64 e 3 caratteri decodificati.

```
// Calcola la lunghezza della stringa di input
int len = strlen(src);
int i = 0, j = 0, in_ = 0;
unsigned char char_array_4[4], char_array_3[3];
```

Si utilizza un ciclo while che itera sulla lunghezza della stringa codificata per recuperare e decodificare gruppi di 4 caratteri. Vengono controllati anche eventuali padding (=) e caratteri non validi (`+ o /`).

```
while (len-- && (src[in_] != '=') && (isalnum(src[in_]) || src[in_]
     == '+' || src[in_] == '/')) {
    // Riempe il buffer di 4 caratteri
    char_array_4[i++] = src[in_];
    in_++;
```

Formato un gruppo di 4 caratteri, questi vengono convertiti nei loro valori a 6 bit usando la tabella ASCII.

```
if (i == 4) {
    for (i = 0; i < 4; i++) char_array_4[i] = base64_decode_table[(
        int)char_array_4[i]];
```

Attraverso operazioni di shifting e somma binaria, i quattro valori numerici recuperati diventano 3 caratteri decodificati.

Per costruire il primo byte sono stati usati i 6 bit del primo carattere e i primi due bit del secondo. Per costruire il secondo byte sono stati usati i 4 bit inferiori del secondo carattere e i 4 bit superiori del terzo. Per costruire il terzo byte sono stati utilizzati i 2 bit inferiori del terzo carattere e i restanti 6 bit del quarto.

```
// Decodifica i 3 byte
char_array_3[0] = (char_array_4[0] << 2) + ((char_array_4[1] & 0x30
    ) >> 4);
char_array_3[1] = ((char_array_4[1] & 0xf) << 4) + ((char_array_4
    [2] & 0x3c) >> 2);
char_array_3[2] = ((char_array_4[2] & 0x3) << 6) + char_array_4[3];

// Scrive i 3 byte decodificati nell'output
for (i = 0; (i < 3); i++) out_buffer[j++] = char_array_3[i];
i = 0;
```

Se la stringa non e' un multiplo perfetto di 4 caratteri, la funzione esegue un blocco successivo dopo il `while`, che riempe gli slot incompleti con il valore 0 ed esegue le medesime operazioni di shifting e somma binaria. Si controlla che il contatore del buffer temporaneo `char_array_4` sia diverso da 0.
Se `i` non è 0, significa che il loop si è interrotto prima di completare un blocco di 4 caratteri.

```
if (i) {
   for (int k = i; k < 4; k++) char_array_4[k] = 0;

   // Mappa caratteri Base64
   for (int k = 0; k < 4; k++) char_array_4[k] =
       base64_decode_table[(int)char_array_4[k]];

   // Operazioni di shifting binario
   ...

   for (int k = 0; (k < i - 1); k++) out_buffer[j++] = char_array_3
       [k];
}
out_buffer[j] = '\0';
return j;
```

Terminata la decodifica, la funzione termina la stringa decodificata `out_buffer` e ritorna la sua lunghezza `j`.

3.9 Status Codes

Il file `status_codes.c` contiene le funzioni che definiscono gli **Status Codes** restituiti dal server. Gli status codes che restituiscono un corpo (`200 e 201`) sviluppati sono formati da tre righe:

1. **Status-Line**, contiene versione protocollo HTTP, lo Status-Code e la Reason-Phrase;
2. **Content Type**, indica il tipo di media contenuto nel `entity-body`;
3. **Content Lenght**, indica la dimensione del body presente nel messaggio di risposta.

Esempio di blocco status code `ok_200`:

```
/*************Successful 2xx***************/
void ok_200(int client_fd, const char *file_ext, const char *body,
    ssize_t body_len) {
    char header[BUFFER_SIZE];
    const char *mime_type = get_mime_type(file_ext);
    int header_len = snprintf(header, BUFFER_SIZE,
        "HTTP/1.1␣200␣OK\r\n"
        "Content-Type:␣%s\r\n"
        "Content-Length:␣%zu\r\n"
        "\r\n",
        mime_type, body_len);

    send(client_fd, header, header_len, 0);
    send(client_fd, body, body_len, 0);
}
```

Gli status codes che non restituiscono un corpo hanno la stessa implementazione dell'header, ma senza le due righe `Content-type e Content-lenght`.

```
void no_content_204 (char *response, size_t *response_size){
    snprintf(response, BUFFER_SIZE,
        "HTTP/1.1␣204␣No␣Content\r\n");
    *response_size = strlen(response);
}
```

4 Risultati

Lo scenario di test analizzato comprende richieste multiple per testare le connessioni persistenti. Lo strumento utilizzato per effettuare i test e' cURL. Sono stati testati tutti i metodi implementati, comprendendo anche casi di mancata autenticazione. L'ordine dei metodi utilizzati durante il test è:

1. `PUT`: inserito un file con e senza credenziali, e provato inserimento con file già presente;
2. `GET`: recupero file su cartella protetta e non protetta, richieste multiple su connessione persistente;
3. `POST`: aggiunta dati su file già esistente;
4. `DELETE`: rimozione file da cartella protetta;

4.1 PUT

Chiamata put con credenziali corrette:

```
marco@marco-swiftsf31443:/$ curl -X PUT -i --user "root:toor" http://localhost:8080/file.txt
 -T file.txt
HTTP/1.1 201 Created
Content-type: application/octet-stream
```

Chiamata put con credenziali errate:

```
marco@marco-swiftsf31443:/$ curl -X PUT -i --user "root:wrong_credentials" http://localhost:8080/file.txt
 -T file.txt
HTTP/1.1 401 Unauthorized
Content-type: text/plain
```

Chiamata put specificando un file già esistente:

```
marco@marco-swiftsf31443:/$ curl -X PUT -i --user "root:toor" http://localhost:8080/file.txt -T file.txt
HTTP/1.1 204 No Content
```

4.2 GET

Chiamata get su cartella non protetta:

```
marco@marco-swiftsf31443:/$ curl -X GET -i http://localhost:8080/file.txt
HTTP/1.1 200 OK
Content-type: application/octet-stream

ciao sono un file di prova
```

Chiamata get su cartella protetta specificando credenziali errate:

```
marco@marco-swiftsf31443:/$ curl -i  http://localhost:8080/root/root_file.txt --user "root:wrong_pwd"
HTTP/1.1 401 Unauthorized
```

Chiamata get su cartella protetta specificando le credenziali corrette:

```
marco@marco-swiftsf31443:/$ curl -X GET -i --user "root:toor" http://localhost:8080/root/root_file.txt
HTTP/1.1 200 OK
Content-type: application/octet-stream

file protetto!marco@marco-swiftsf31443:/$
```

Chiamate multiple get su connessione persistente:

```
marco@marco-swiftsf31443:/$ curl http://localhost:8080/file.txt http://localhost:8080/file2.txt
ciao sono un file di prova
ciao sono un file di prova
sono un file di prova 2
```

Server su chiamate get multiple:

```
marco@marco-swiftsf31443:~/Desktop/Laboratorio Reti/Progetto$ ./server
Server in ascolto sulla porta 8080
Nuova connessione accettata: fd 5
Client fd 5 disconnesso.
Chiusura connessione per fd 5
```

4.3 POST

Chiamata post su file già esistente:

```
marco@marco-swiftsf31443:/$ curl -i -X PUT http://localhost:8080/file.txt -d "Ciao " --user "root:toor"
HTTP/1.1 201 Created
Content-Type: application/octet-stream
Content-Length: 5

Ciao marco@marco-swiftsf31443:/$
```

4.4 DELETE

Chiamata delete su cartella protetta:

```
marco@marco-swiftsf31443:/$ curl -X DELETE -i --user "root:toor" http://localhost:8080/root/root_file.txt
HTTP/1.1 204 No Content
```

5 Conclusioni

Il server e' quindi in grado di gestire le richieste HTTP più comuni in modo funzionale e conforme a RFC 2616, garantendo la persistenza delle connessioni.