Report progetto

Andrea Caltagirone, Matricola 542666

17 febbraio 2026

1 Introduzione

Per creare un server concorrente in C che gestisca almeno le richieste GET, POST, PUT, DELETE, sono stati utilizzati vari elementi affrontati durante il corso per poter utilizzare strumenti come i socket ed i thread al meglio.

Socket

L'interfaccia socket in linux è un pilastro della programmazione di rete essi vengono utilizzati per stabilire una comunicazione bidirezionale tra due endpoint. La funzione socket è utilizza per creare un nuovo socket, attraverso di essa si può specificare se la comunicazione verrà effetuata attraverso IPv4 o v6,, oppure il tipo di socket se per esempio sarà orientato ai byte e affidabile (es. TCP) , se sarà basato sui messaggi, non affidabile (es. UDP), se sarà un socket RAW o altri ancora. Una volta creato il socket si usa la funzione bind per associarlo ad un indirizzo ed a una porta specifica.
Altre funzioni utiliti nell'utilizzo dei socket sono:

- a funzione listen per mettere per esempio il socket di un server in ascolta per possibili connesioni di client
- la funzione accept per accetare le connessioni in entrata una volta che il socket è stato messo in ascolto attraverso listen
- La funzione connect per connettere un socket ad un server remoto
- le funzioni send e recv che vengono utilizzati per i socket orientati alle connessioni per inviare e ricevere dati
- Le funzioni sendto e recvfrom invece vengono utilizzate per socket senza connessi'one sempre per inviare e ricevere dati (a differenza di send e recv bisognerà specificare l'indirizzo della destinazione o della sorgente tra i parametri)

Thread

Nei sistemi moderni un processo può essere costituito da più unità di esecuzione, chiamate thread, ciascuna delle quali viene eseguita nel contesto del processo e condivide lo stesso codice e gli stessi dati globali. Alcuni dei vantaggi nell'utilizzo dei thread sono la maggiore facilità nel condividere informazioni tra di essi piuttosto che tra i processi ed inoltre il multi-threading è un modo per rendere i programmi più veloci quando sono presenti più processori/core.

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_-routine) (void *), void *arg); è il prototipo della funzione pthread_create che permette di creare un thread e fargli avviare l'esecuzione della funzione "start_-routine".
altre funzioni utili nella gestione dei thread posso essere per esempio:

- pthread_join che permette di attendere la terminazione di un thread specificato
- pthread_exit che termina il thread corrente

2 Definizione del problema

L'obiettivo del progetto è quello di realizzare un server http concorrente in C che gestisca le richieste GET, POST, PUT, DELETE e le risposte principali (200, 201, 204, 400, 401), per fare questo sono stati utilizzate i socket per la comunicazione con il server, ed ogni richiesta viene gestita da un thread diverso per implementare l'esecuzione concorrente

3 Metodologia

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <pthread.h>

#include "header.h"

#define BUFFER_SIZE 4096
#define PORT 8080
#define BACKLOG 10

regex_t re_http;

int main() {
    int server_fd, *new_sock;
    struct  sockaddr_in address;
    int addrlen = sizeof(address);

    pthread_t thread_id;

    regcomp(&re_http, "^(GET|POST|PUT|DELETE) /([^ ]*) HTTP/1", REG_EXTENDED);

    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket_failed");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0 )
    {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    if (listen(server_fd, BACKLOG) < 0)
```

```
    {
        perror("listen");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    while (1)
    {
        new_sock = malloc(sizeof(int));

        if ((*new_sock = accept(server_fd, (struct sockaddr *)&address, (socklen_t*)&addrlen)) >= 0)
        {
            if (pthread_create(&thread_id, NULL, handle_client, (void*)new_sock) != 0)
            {
                perror("pthread_create failed");
                close(*new_sock);
                free(new_sock);
                continue;
            }
            pthread_detach(thread_id);
        }else{
            free(new_sock);
        }
    }
}
```

Questo è il codice riguardante il funzionamento del server

```
include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <pthread.h>

#include "header.h"

#define BUFFER_SIZE 4096
#define PORT 8080
#define BACKLOG 10

regex_t re_http;
```

Qui includo:

- Le librerie di cui il server ha bisogno
- il file header.h nel quale vengono dichiarati elementi che poi verranno richiamati dal server
- definisco dei valori costanti quali la dimensione del buffer, la porta del server ed il backlog per le connessioni in attesa col server
- in fine dichiaro la variabile globale re_http;

```
#ifndef HEADER_H
#define HEADER_H


#include <regex.h>

extern regex_t re_http;

void *handle_client(void *arg);

char* get_extension(const char *path);
char* get_mime_type(const char* ext);
void send_response(int client_fd,
int status_code, const char *content_type,const char *body, size_t body_len);
```

```
int check_auth(const char *auth_header);

void handle_get(int client_fd, const char *file_name);
void handle_post(int client_fd, const char *path, const char *body);
void handle_put(int client_fd, const char *path, const char *body);
void handle_delete(int client_fd, const char *path);

#endif
```

Questo è il file header.h nel quale vengono definite le varie funmzioni e variabili che poi saranno richiamati da vari moduli dell'applicativo

```
int server_fd, *new_sock;
struct  sockaddr_in address;
int addrlen = sizeof(address);
pthread_t thread_id;
regcomp(&re_http, "^(GET|POST|PUT|DELETE) /([^ ]*) HTTP/1", REG_EXTENDED);
```

Qui dichiaro le variabili:

- int server_fd, *new_sock che conterranno il riferimento al file descriptor del socket del server e dei socket dei client che si connetterrano
- struct sockaddr_in address che contiene la struttura dell'indirizzo del server
- int addrlen = sizeof(address) che contiene la lunghezza dell'indirizzo
- pthread_t thread_id che contiene l'identificatore di un thread
- in fine compilo la regex re_http per gestire le varie richieste http

```
if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket_failed");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0 )
    {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    if (listen(server_fd, BACKLOG) < 0)
    {
        perror("listen");
        close(server_fd);
        exit(EXIT_FAILURE);
    }
```

Qui nell'ordine

1. ottengo il file descriptor che fa riferimento alla socket del server
2. configuro la struttura sockaddr_in definendo la famiglia degli indirizzi IPv4, che accetta da qualsiasi interfaccia di rete ed imposta la porta del server convertita nel formato di rete
3. Con bind associo il socket del server con la struttura sockaddr

4. in fine imposto il socket in ascolto delle connessioni dei client con listen

```
while (1)
    {
        new_sock = malloc(sizeof(int));

        if ((*new_sock = accept(server_fd, (struct sockaddr *)&address, (socklen_t*)&addrlen)) >= 0)
        {
            if (pthread_create(&thread_id, NULL, handle_client, (void*)new_sock) != 0)
            {
                perror("pthread_create failed");
                close(*new_sock);
                free(new_sock);
                continue;
            }
            pthread_detach(thread_id);
        }else{
            free(new_sock);
        }
    }
```

viene creato dinamicamente l'allocazione di memoria per il file descriptor del client che si collegherà, ed una volta che un client verrà accettato dalla funzione accept il server crearà un thread che lancerà la funzione handle_client che si occuperà di gestire la specifica richiesta http

```
void *handle_client(void *arg) {

    int client_fd = *(int *)arg;
    free (arg);

    char buffer[BUFFER_SIZE] = {0};

    regmatch_t matches[3];

    char method[16] = {0};
    char path[256] = {0};

    ssize_t byte_rcv = read(client_fd, buffer, BUFFER_SIZE - 1);
    if (byte_rcv <= 0)
    {
        close(client_fd);
        return NULL;
    }

    buffer[byte_rcv] = '\0';

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

        //metodo
        int len = matches[1].rm_eo - matches[1].rm_so;
        strncpy(method, buffer + matches[1].rm_so, len);
        method[len] = '\0';

        //path
        len = matches[2].rm_eo - matches[2].rm_so;
        strncpy(path, buffer + matches[2].rm_so, len);
        path[len] = '\0';

        if (strstr(path, "admin") != NULL)
        {
            char *auth_line = strstr(buffer, "Authorization:");

            if(auth_line == NULL || !check_auth(auth_line)) {
                send_response(client_fd, 401, "text/plain", "Unauthorized", 12);
                close(client_fd);
                return NULL;
            }
        }
```

```
        if ((strcmp(method, "GET")) == 0)
        {
            handle_get(client_fd, path);

        }else if ((strcmp(method, "DELETE")) == 0)
        {
            handle_delete(client_fd, path);

        }else if ((strcmp(method, "PUT")) == 0)
        {
            char *body = strstr(buffer, "\r\n\r\n");
            if (body != NULL) {
                body += 4; // Salta "\r\n\r\n"
            }
            handle_put(client_fd, path, body);

        }else if ((strcmp(method, "POST")) == 0)
        {
            char *body = strstr(buffer, "\r\n\r\n");
            if (body != NULL) {
                body += 4; // Salta "\r\n\r\n"
            }
            handle_post(client_fd, path, body);

        }else{
            send_response(client_fd, 400, "text/plain", "Bad Request: Metodo non supportato", 34);
        }
    }else{
        send_response(client_fd, 400, "text/plain", "Bad Request: Formato richiesta invalido", 40);
    }

    close(client_fd);
    return NULL;

}
```

Questo è la funzione handle_client che viene lanciata da ogni thread

```
int client_fd = *(int *)arg;
free (arg);

char buffer[BUFFER_SIZE] = {0};

regmatch_t matches[3];

char method[16] = {0};
char path[256] = {0};
```

viene passato il parametro contenente il file descriptor in int client_fd, con free libero lo spazio che è stato allocato in memoria per il parametro arg, viene inizializzato il buffer l'array che conterrà il metodo usato dalla richiesta e quella che conterrà il path infine viene dichiarato un array di tre strutture regmatch_t

```
ssize_t byte_rcv = read(client_fd, buffer, BUFFER_SIZE - 1);
    if (byte_rcv <= 0)
    {
        close(client_fd);
        return NULL;
    }

    buffer[byte_rcv] = '\0';
```

viene lette e passato in byte_rcv il messaggio inviato dal client

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

        //metodo
        int len = matches[1].rm_eo - matches[1].rm_so;
        strncpy(method, buffer + matches[1].rm_so, len);
        method[len] = '\0';
```

```
        //path
        len = matches[2].rm_eo - matches[2].rm_so;
        strncpy(path, buffer + matches[2].rm_so, len);
        path[len] = '\0';

        if (strstr(path, "admin") != NULL)
        {
            char *auth_line = strstr(buffer, "Authorization:");

            if(auth_line == NULL || !check_auth(auth_line)) {
                send_response(client_fd, 401, "text/plain", "Unauthorized", 12);
                close(client_fd);
                return NULL;
            }
        }
```

In questa sezione del codice viene effetuato il parsing della richiesta http, attraverso l'utilizzo delle regex, ciò viene fatto con questi passaggi:

1. inizialmente si utilizza regexec per verificare se il messaggio contenuto nel buffer match con re_http ed in caso positivo passare i valori corretti a matches

2. verificato il riscontro inizialmente estraiamo il valore del metodo della richiesta (GET POST PUT DELETE)

3. in secondo luogo invece con un passaggio analogo estraiamo il percorso del file a cui il client vuole accedere

4. viene verificata la presenza della stringa "admin" per controllare se il client stia cercando di accedere a dei file protetti ed in caso positivo vengono estratte le credenziali con cui il client sta cercando di autenticarsi, le quali verranno passate alla funzione check_auth per effetuare il controllo. In caso non siano presenti le credenziali o se la verifica delle stesse avrà esito negativo verrà usata la funzione send_response per inviare il messaggio di errore 401

```
if ((strcmp(method, "GET")) == 0)
        {
            handle_get(client_fd, path);

        }else if ((strcmp(method, "DELETE")) == 0)
        {
            handle_delete(client_fd, path);

        }else if ((strcmp(method, "PUT")) == 0)
        {
            char *body = strstr(buffer, "\r\n\r\n");
            if (body != NULL) {
                body += 4; // Salta "\r\n\r\n"
            }
            handle_put(client_fd, path, body);

        }else if ((strcmp(method, "POST")) == 0)
        {
            char *body = strstr(buffer, "\r\n\r\n");
            if (body != NULL) {
                body += 4; // Salta "\r\n\r\n"
            }
            handle_post(client_fd, path, body);

        }else{
            send_response(client_fd, 400, "text/plain", "Bad Request: Metodo non supportato", 34);
        }
    }else{
```

```
        send_response(client_fd, 400, "text/plain", "Bad Request: Formato richiesta invalido", 40);
    }

    close(client_fd);
    return NULL;
```

In questa serie di if viene verificato il valore di method in modo da chiamare la funzione correttà che poi gestirà la richiesta http specifica

```
int base64_decode(const char *input, char *output, size_t output_size) {
    int val = 0, valb = -8;
    size_t out_len = 0;


    for (const char *p = input; *p && *p != '='; p++) {
        const char *pos = strchr(base64_table, *p);
        if (pos == NULL) {
            fprintf(stderr, "Invalid base64 character: '%c' (0x%02x)\n", *p, (unsigned char)*p);
            return -1;
        }

        val = (val << 6) + (pos - base64_table);
        valb += 6;

        if (valb >= 0) {
            if (out_len >= output_size - 1) {
                fprintf(stderr, "Output buffer too small\n");
                return -1;
            }
            output[out_len++] = (char)((val >> valb) & 0xFF);
            valb -= 8;
        }
    }

    output[out_len] = '\0';
    fprintf(stderr, "Decoded: '%s' (length: %zu)\n", output, out_len);
    return (int)out_len;
}

int check_auth(const char *auth_header) {

    const char *valid_user = "admin";
    const char *valid_password = "123";

    const char *basic = strstr(auth_header, "Basic ");
    fprintf(stderr, "basic pointer: %p\n", (void*)basic);

    if (basic == NULL) {
        fprintf(stderr, "Basic NOT found\n");
        return 0;
    }

    basic += 6;

    char base64_str[256];
    int i = 0;
    while (basic[i] && basic[i] != '\r' && basic[i] != '\n' && basic[i] != ' ' && i < 255) {
        base64_str[i] = basic[i];
        i++;
    }
    base64_str[i] = '\0';

    fprintf(stderr, "base64_str extracted: '%s'\n", base64_str);
    fprintf(stderr, "base64_str length: %d\n", i);

    char decoded[256];
    int decode_result = base64_decode(base64_str, decoded, sizeof(decoded));
    fprintf(stderr, "base64_decode returned: %d\n", decode_result);

    if (decode_result < 0) {
        fprintf(stderr, "base64_decode failed\n");
        return 0;
    }
```

```
    char *colon = strchr(decoded, ':');
    if (colon == NULL) {
        fprintf(stderr, "No colon found in decoded string\n");
        return 0;
    }

    *colon = '\0';
    char *username = decoded;
    char *password = colon + 1;

    fprintf(stderr, "username: '%s', password: '%s'\n", username, password);
    fprintf(stderr, "Comparing with valid_user: '%s',
    valid_password: '%s'\n", valid_user, valid_password);

    if (strcmp(username, valid_user) == 0 && strcmp(password, valid_password) == 0) {
        fprintf(stderr, " AUTH SUCCESS\n");
        return 1;
    }

    fprintf(stderr, " AUTH FAILED\n");
    return 0;
}
```

così viene implementata la decodifica in base64 delle credenziali di autorizzazioni che vengono passate dal client

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

#include "header.h"

#define BUFFER_SIZE 4096
#define WEB_ROOT "./www/"

void handle_get(int client_fd, const char *path){

    char file_path[512];
    int file_fd;
    snprintf(file_path, sizeof(file_path), "%s%s", WEB_ROOT, path);

    char *ext =get_extension(path);
    char *mime_type =get_mime_type(ext);

    file_fd = open(file_path, O_RDONLY);
    if (file_fd == -1){
        send_response(client_fd, 404, "text/plain", "404 Not Found", 13);
        return;
    }

    struct stat file_stat;
    fstat(file_fd, &file_stat);
    off_t file_size = file_stat.st_size;

    char *file_content = malloc(file_size);
    if ((file_content == NULL))
    {
        close(file_fd);
        send_response(client_fd, 500, "text/plain", "Internal Server Error", 21);
        return;
    }

    ssize_t bytes_read = read(file_fd, file_content, file_size);
    close(file_fd);

    if (bytes_read != file_size) {
        free(file_content);
        send_response(client_fd, 500, "text/plain", "Internal Server Error", 21);
        return;
```

```
    }

    send_response(client_fd, 200, mime_type, file_content, file_size);

    free(file_content);

}

void handle_post(int client_fd, const char *path, const char *body){

    char file_path[512];

    snprintf(file_path, sizeof(file_path), "%s%s", WEB_ROOT, path);

    int file_fd = open(file_path, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (file_fd == -1) {
        send_response(client_fd, 400, "text/plain", "Bad Request", 11);
        return;
    }

    if (body != NULL) {
        write(file_fd, body, strlen(body));
    }
    close(file_fd);

    send_response(client_fd, 201, "text/plain", "Created", 7);
}

void handle_put(int client_fd, const char *path, const char *body) {

    char file_path[512];

    snprintf(file_path, sizeof(file_path), "%s%s", WEB_ROOT, path);

    int file_fd = open(file_path, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (file_fd == -1) {
        send_response(client_fd, 400, "text/plain", "Bad Request", 11);
        return;
    }

    if (body != NULL) {
        write(file_fd, body, strlen(body));
    }
    close(file_fd);

    send_response(client_fd, 204, "text/plain", "", 0);
}

void handle_delete(int client_fd, const char *path) {

    char file_path[512];

    snprintf(file_path, sizeof(file_path), "%s%s", WEB_ROOT, path);

    if (unlink(file_path) == -1) {
        send_response(client_fd, 404, "text/plain", "Not Found", 9);
        return;
    }

    send_response(client_fd, 204, "text/plain", "", 0);
}
```

Queste sono le varie funzioni che si occupano di gestire le diverse richieste http

```
void handle_get(int client_fd, const char *path){

    char file_path[512];
    int file_fd;
    snprintf(file_path, sizeof(file_path), "%s%s", WEB_ROOT, path);

    char *ext =get_extension(path);
    char *mime_type =get_mime_type(ext);

    file_fd = open(file_path, O_RDONLY);
```

```
    if (file_fd == -1){
        send_response(client_fd, 404, "text/plain", "404 Not Found", 13);
        return;
    }

    struct stat file_stat;
    fstat(file_fd, &file_stat);
    off_t file_size = file_stat.st_size;

    char *file_content = malloc(file_size);
    if ((file_content == NULL))
    {
        close(file_fd);
        send_response(client_fd, 500, "text/plain", "Internal Server Error", 21);
        return;
    }

    ssize_t bytes_read = read(file_fd, file_content, file_size);
    close(file_fd);

    if (bytes_read != file_size) {
        free(file_content);
        send_response(client_fd, 500, "text/plain", "Internal Server Error", 21);
        return;
    }

    send_response(client_fd, 200, mime_type, file_content, file_size);

    free(file_content);

}
```

Questa è la funzione che si occupa di gestire le richieste GET ed in ordine effetua queste operazioni:

1. Riceve come parametri il file_descriptor del client ed il percorso del file a cui il client vuole accedere
2. Definisce un file descriptor che verrà utilizzato per aprire il file richiesto ed una stringa che conterrà il percorso del file, la quale viene inizializzata con snprintf,
3. rispettivamente con le funzioni get_extension e get_mime_type estrae l'estensione del file a cui il client sta cercado di accedere ed il mime_type dello stesso
4. Apre in modalità lettura il file richiesto
5. viene dichiarata una struttura file stat nella quale saranno contenute varie informazioni del file richiesto, nel nostro caso ci servirà per ricavare la grandezza del file e scriverla in file_size, per poter allocare dinamicamente uno spazio della dimensione del file
6. Viene effetuata la lettura del contenuto del file
7. tramite la funzione send_response verrà inoltrata la risposta al client

```
void handle_post(int client_fd, const char *path, const char *body){

    char file_path[512];

    snprintf(file_path, sizeof(file_path), "%s%s", WEB_ROOT, path);

    int file_fd = open(file_path, O_WRONLY | O_CREAT | O_EXCL, 0644);
    if (file_fd == -1) {
```

```
        send_response(client_fd, 400, "text/plain", "Bad Request", 11);
        return;
    }

    if (body != NULL) {
        write(file_fd, body, strlen(body));
    }
    close(file_fd);

    send_response(client_fd, 201, "text/plain", "Created", 7);
}
```

Questa è la funzione che si occupa di gestire le richieste POST ed in ordine effetua queste operazioni:

1. innazitutto come parametri riceverà anche il body della richiesta
2. Viene costruito il percorso del file che il client vuole aggiungere
3. Viene creato il file in modalità scrittura
4. viene scritto il body della richiesta http nel file aperto
5. viene inviata la risposta al client con send_response

```
void handle_put(int client_fd, const char *path, const char *body) {

    char file_path[512];

    snprintf(file_path, sizeof(file_path), "%s%s", WEB_ROOT, path);

    int file_fd = open(file_path, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (file_fd == -1) {
        send_response(client_fd, 400, "text/plain", "Bad Request", 11);
        return;
    }

    if (body != NULL) {
        write(file_fd, body, strlen(body));
    }
    close(file_fd);

    send_response(client_fd, 204, "text/plain", "", 0);
}
```

Il contenuto della funzione PUT è simile a quello della funzione handle_post solo che che i file che il client vorrà creare verranno sovrascritti qualora già esistenti ed inoltre con send_response verrà inoltrata al client la risposta 204

```
void handle_delete(int client_fd, const char *path) {

    char file_path[512];

    snprintf(file_path, sizeof(file_path), "%s%s", WEB_ROOT, path);

    if (unlink(file_path) == -1) {
        send_response(client_fd, 404, "text/plain", "Not Found", 9);
        return;
    }

    send_response(client_fd, 204, "text/plain", "", 0);
}
```

QUesta è la funzione che si occupa di gestire la richiesta DELETE e fa ciò eseguendo la funzione unlink sul file specificato dal client

```
char* get_extension(const char *path) {
    char *ext = strrchr(path, '.');
    return (ext == NULL) ? "" : ext + 1;
}
```

questa è la funzione che permette di estrarre l'estensione dei file richiesti dal client

```
char* get_mime_type(const char *ext) {
    if (strcmp(ext, "html") == 0) return "text/html";
    if (strcmp(ext, "css") == 0)  return "text/css";
    if (strcmp(ext, "js") == 0)   return "application/javascript";
    if (strcmp(ext, "txt") == 0)  return "text/plain";
    if (strcmp(ext, "json") == 0) return "application/json";
    if (strcmp(ext, "jpg") == 0 || strcmp(ext, "jpeg") == 0) return "image/jpeg";
    if (strcmp(ext, "png") == 0)  return "image/png";
    return "application/octet-stream";
}
```

QUesta è la funzione che permette di ricavare il mime__typee di un file e fa ciò associando a varie estensioni dei file a cui un client potrebbe chiedere l'accesso il loro mime_tyoe associato

```
void send_response(int client_fd, int status_code, const char *content_type,
                   const char *body, size_t body_len) {
    char header[512];
    const char *status_text;

    // Determina il testo dello status
    switch (status_code) {
        case 200: status_text = "OK"; break;
        case 201: status_text = "Created"; break;
        case 204: status_text = "No Content"; break;
        case 400: status_text = "Bad Request"; break;
        case 401: status_text = "Unauthorized"; break;
        case 404: status_text = "Not Found"; break;
        case 500: status_text = "Internal Server Error"; break;
        default: status_text = "Unknown"; break;
    }

    // Costruisci header
    int header_len = snprintf(header, sizeof(header),
        "HTTP/1.1 %d %s\r\n"
        "Content-Type: %s\r\n"
        "Content-Length: %zu\r\n"
        "%s"
        "Connection: close\r\n"
        "\r\n",
        status_code, status_text, content_type, body_len,
        (status_code == 401 ? "WWW-Authenticate: Basic realm=\"Protected Area\"\r\n" : ""));

    // Invia header
    write(client_fd, header, header_len);

    // Invia body (se presente)
    if (body_len > 0 && body != NULL) {
        write(client_fd, body, body_len);
    }
}
```

Questa è la funzione che permette di inviare la risposta correttà al client in base alla richiesta http che è stata effetuata, per fare ciò innanzitutto

1. determina lo status text in base al codice della risposta
2. costruisci l'header della risposta inserendo lo status_code, lo status_text, il content_type ed il body_len

3. In fine invia l'header della risposta e se presente il body anche il body della stessa

```
CC = gcc
CFLAGS = -Wall -Wextra -pthread -std=c11
LDFLAGS = -pthread

# File sorgenti
SRCS = server.c handle_client.c method.c utility.c auth.c
OBJS = $(SRCS:.c=.o)
TARGET = server

# Target principale
all: $(TARGET)

$(TARGET): $(OBJS)
        $(CC) $(OBJS) -o $(TARGET) $(LDFLAGS)

# Compilazione oggetti
%.o: %.c header.h
        $(CC) $(CFLAGS) -c $< -o $@

# Pulizia
clean:
        rm -f $(OBJS) $(TARGET)

# Esecuzione
run: $(TARGET)
        ./$(TARGET)
```

Questo è il makefile che permette la compilazione modulare del server http, compilando i vari file .c in file .o per poi effetuare il linking finale per generare l'eseguibile

```
CC = gcc
CFLAGS = -Wall -Wextra -pthread -std=c11
LDFLAGS = -pthread
```

Qui vengono definite tre variabili che specificano il compilatore da usare e le opzioni di compilazione e linking che verranno applicate durante la costruzione del progetto.

```
SRCS = server.c handle_client.c method.c utility.c auth.c
OBJS = $(SRCS:.c=.o)
TARGET = server
```

Viene definita una variabile che contiene tutti i file sorgente del programma; con OBJS si specifica che ogni file .c deve essere compilato nel corrispondente .o, mentre con TARGET si indica che l'eseguibile finale si chiamerà server.

```
all: $(TARGET)

$(TARGET): $(OBJS)
        $(CC) $(OBJS) -o $(TARGET) $(LDFLAGS)
```

Qui viene definito il target principale del Makefile. Il target all richiede la creazione dell'eseguibile ($(TARGET)), che a sua volta dipende da tutti i file oggetto ($(OBJS)). Se uno dei file .o cambia, make ricostruisce automaticamente l'eseguibile eseguendo il comando di linking, che unisce tutti gli oggetti in un unico programma finale.

```
%.o: %.c header.h
        $(CC) $(CFLAGS) -c $< -o $@
```

Questa regola indica a make come compilare ogni file .c nel corrispondente file oggetto .o. La dipendenza da header.h fa sì che, se l'header viene modificato,

tutti i file .o vengano ricompilati. Il comando usa il compilatore ($(CC)) con i flag definiti ($(CFLAGS)), compilando il file sorgente $< e producendo il file oggetto $@.

4 Risultati

Questi sono le risposte che ho ricevuto dal server una volta effetuate le richeiste GET POST PUT e DELETE: Da qui si può vedere il funzionamento della richiesta GET

```
vboxuser@Ubuntu:~/Desktop/Laboratorio$ curl  http://localhost:8080/index.html
<h1>Homepage</h1>
```

Questo invece è il messaggio di risposta alla richiesta POST che porta alla creazione del file test.txt

```
vboxuser@Ubuntu:~/Desktop/Laboratorio$ curl -X POST -d "test data" http://localhost:8080/test.txt
Created
```

Questo è il risultato della richiesta PUT che sovrascrive il file test.txt e come si può vedere non scrive un messaggio al client in quanto la risposta con codice 204 non ha un body

```
vboxuser@Ubuntu:~/Desktop/Laboratorio$ curl -X PUT -d "Updated data" http://localhost:8080/test.txt
```

Questo è il risultato della richiesta DELETE che elimina il file test.txt

```
vboxuser@Ubuntu:~/Desktop/Laboratorio$ curl -X DELETE http://localhost:8080/test.txt
```

5 Conclusioni

ConcQuesto progetto ha portato all'implementazione di un server HTTP concorrente che gestisce correttamente le richieste GET, POST, PUT, DELETE e le risposte principali 200, 201, 204, 400 e 401, per fare ciò come si è visto è stata utilizzata l'interfaccia socke per implementare la comunicazione client server e l'utilizzo della libreria pthread.h per implementare la concorrenza attraverso una gestione in thread indipendenti di ogni richiesta http ricevuta dal server