

Progetto di Fine Corso 2025 - Server HTTP

Davide Frodà

Corso di laurea: Scienze informatiche

Giugno 2025

Contents

1	Introduzione	3
2	Definizione del Problema	3
3	Stato dell'Arte	4
4	Metodologia	5
5	Implementazione	6
5.1	header	6
5.2	server	11
5.3	socket_functions	12
5.4	epoll_functions	15
5.5	thread_functions	16
5.6	parsing_functions	18
5.7	http_methods	24
5.7.1	GET	27
5.7.2	POST	28
5.7.3	PUT	30
5.7.4	DELETE	32
5.8	file_functions	33
5.9	sending_functions	38
6	Risultati sperimentali	40
7	Conclusione e sviluppi futuri	42

1 Introduzione

Questo report si propone di illustrare l'implementazione di un server HTTP, multicient e concorrente, in linguaggio C. Il server andrà a gestire le richieste GET, POST, PUT e DELETE, rispondendo secondo come descritto dallo standard RFC 2616¹.

2 Definizione del Problema

Il problema da affrontare consiste nella realizzazione di un server HTTP in linguaggio C che rispetti lo standard RFC 2616 sia nell'elaborazione delle richieste che nella formulazione delle risposte. Il server dovrà inoltre essere capace di gestire più client connessi simultaneamente seguendo i principi della programmazione concorrente.

Di seguito i principali codici di risposta che il server dovrà essere in grado di utilizzare:

- 200 (OK): l'operazione è stata eseguita con successo.
- 201 (Created): l'operazione ha comportato la creazione di un file.
- 204 (No Content): l'operazione è stata eseguita senza errori ma la risposta non possiede un contenuto da restituire.
- 400 (Bad Request): la richiesta non è formulata in maniera corretta, non rispetta la sintassi dell'RFC 2616.
- 401 (Unauthorized): la richiesta deve contenere un campo Authorization valido per essere eseguita.

A questi si aggiungono altri codici di risposta che ho deciso di utilizzare nell'implementazione:

- 202 (Accepted): la richiesta è stata accettata ma non ancora eseguita, è utilizzato per le operazioni asincrone.
- 404 (Not Found): la richiesta tenta di accedere ad un file non esistente.
- 405 (Method Not Allowed): l'operazione non è consentita su l'URI specificata.
- 414 (Request-URI Too Large): la richiesta presenta un URI troppo lungo che non può essere elaborato dal server.
- 500 (Internal Server Error): il server ha riscontrato un problema inatteso e non può portare a termine la richiesta.
- 501 (Not Implemented): il server non supporta tutte le funzionalità per portare a termine la richiesta.

¹fonte: <https://datatracker.ietf.org/doc/html/rfc2616>

Le sfide principali includono:

- Realizzazione di un server multicient.
- Gestione concorrente delle richieste dei client.
- Parsing e validazione delle richieste HTTP seguendo l’RFC 2616.
- Formulazione corretta e specifica delle risposte alle richieste del client, differenziando anche gli errori tramite i codici di stato.
- Sincronizzazione dei threads nell’accesso alle risorse condivise.
- Esecuzione di test di carico sul server.

3 Stato dell’Arte

Le implementazioni di server multicient vengono attualmente realizzate tramite tre principali metodologie di I/O multiplexing:

- select: chiamata di sistema utilizzata per monitorare più file descriptor.
 - Multiplatforma: disponibile sia su linux che su windows.
 - Poco scalabile: possiede buone prestazioni con pochi client ma le prestazioni si degradano relativamente velocemente all’aumentare dei client.
- epoll: API Linux con un meccanismo basato su eventi.
 - Possiede prestazioni elevate anche con migliaia di clients.
- thread-per-client: associa un thread ad ogni client che viene quindi gestito indipendentemente.
 - Poco scalabile, quando si connettono migliaia di client il server dovrebbe gestire migliaia di threads.

Dovendo scegliere tra queste tre possibilità l’implementazione è stata realizzata utilizzando `epoll()`.

4 Metodologia

La realizzazione del server HTTP in linguaggio C ha richiesto numerose scelte di implementazione, di seguito la metodologia seguita:

- Per la gestione dell'aspetto multiclient del server, come anticipato, sono state utilizzate le API `epoll`.
- Per la gestione della concorrenza tra client è stato implementato un meccanismo che associ ad ogni evento un thread. Questa scelta è dovuta alla priorità che dona questa implementazione al numero di connessioni che è possibile gestire: il quale è maggiore gestendo come threads i singoli eventi piuttosto che i singoli clients. Tuttavia va sottolineato questa implementazione è stata scelta nonostante causi un aumento della latenza.
- Ogni richiesta, come vedremo nel dettaglio, effettua anzitutto una fase di parsing, la quale controlla che la richiesta sia formulata in maniera coerente all'RFC 2616.
- Dopo la fase di parsing, ogni richiesta viene elaborata in base al suo metodo.
- Vi sono due principali sistemi di sincronizzazione all'interno del server:
 - `pthread_mutex`: utilizzato per evitare race condition nella fase di accettazione dei clients
 - `flock`: per la sincronizzazione dell'accesso ai file da parte di richieste simultanee. `flock` è stato ritenuto il meccanismo più efficace in quanto possiede una distinzione intrinseca tra il lock in lettura di un file, che non permette di modificarlo ma consente al altri threads di accedervi anche loro in lettura, e il lock in scrittura, che non consente a nessun altro thread di accedere al file nè in lettura nè in scrittura.
- Una scelta importante nella realizzazione del server HTTP è stata anche la realizzazione dell'operazione DELETE come operazione asincrona in quanto la funzione `unlink()` non cancella effettivamente il file finché non vengono chiusi tutti i file descriptor ad esso associati.
- L'implementazione del programma è, ovviamente, stata realizzata modularmente ed è stato creato un apposito `makefile` per la compilazione.

5 Implementazione

L'implementazione della soluzione è suddivisa in diversi file, i quali verranno analizzati singolarmente.

5.1 header

Listing 1: include

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

- `stdio.h`: Funzioni di input/output standard.
- `stdlib.h`: Funzioni standard di C.
- `unistd.h`: Funzioni standard in UNIX.
- `fcntl.h`: Funzioni per operazioni su file descriptor.
- `string.h`: Funzioni per la manipolazione delle stringhe.
- `errno.h`: Funzioni per la gestione degli errori.
- `pthread.h`: Funzioni per la gestione dei threads.
- `sys/types.h`: Tipi del sistema come `pid_t`, `size_t` e molti altri.
- `sys/socket.h`: Funzioni per la gestione dei socket.
- `netinet/in.h`: Strutture e macro per indirizzi IP.
- `sys/epoll.h`: API epoll per I/O multiplexing ad alte prestazioni.

Listing 2: define

```
#define BUFFER_SIZE 10000
#define PORT 8080
#define MAX_EVENTS 1000
#define MAX_METHOD_LEN 7
#define MAX_URI_LEN 2000
#define MAX_REASON_PHRASE_LEN 22
#define MAX_CONNECTION_LEN 11
#define MAX_HEADER_LEN 4000
#define MAX_REALM_LEN 8
#define MAX_ALLOW_LEN 23
#define HTTP_VERSION 1.1
#define AUTH_SCHEME "Basic"
#define MAX_AUTH_LEN 100
#define MAX_AUTH_SCHEME_LEN 7

#define MESSAGE_400 "<html><head><title>400_Bad_Request</title></head><body><h1>400_Bad_Request</h1><p>Your_browser_sent_a_request_that_this_server_could_not_understand.</p></body></html>"
#define MESSAGE_400_LEN 167
#define MESSAGE_401 "<html><head><title>401_Unauthorized</title></head><body><h1>401_Unauthorized</h1><p>Permission_is_required.</p></body></html>"
#define MESSAGE_401_LEN 126
#define MESSAGE_404 "<html><head><title>404_Not_Found</title></head><body><h1>Not_Found</h1><p>The_requested_URL_was_not_found_on_this_server.</p></body></html>"
#define MESSAGE_404_LEN 140
#define MESSAGE_414 "<html><head><title>414_Request_URI_Too_Large</title></head><body><h1>414_Request_URI_Too_Large</h1><p>The_requested_URI_is_too_long_for_this_server_to_process.</p></body></html>"
#define MESSAGE_414_LEN 178
#define MESSAGE_500 "<html><head><title>500_Internal_Server_Error</title></head><body><h1>500_Internal_Server_Error</h1><p>An_unexpected_error_occurred.</p></body></html>"
#define MESSAGE_500_LEN 150
```

- BUFFER_SIZE: Dimensione massima del buffer per ricevere o inviare dati.
- PORT: Porta su cui il server ascolta le connessioni.
- MAX_EVENTS: Numero massimo di eventi gestibili simultaneamente da `epoll()`.
- MAX_METHOD_LEN: Lunghezza massima del metodo HTTP.
- MAX_URI_LEN: Lunghezza massima dell'URI nella richiesta HTTP.
- MAX_REASON_PHRASE_LEN: Lunghezza massima della frase di stato HTTP.
- MAX_CONNECTION_LEN: Lunghezza massima per l'header Connection.
- MAX_HEADER_LEN: Lunghezza massima consentita per gli header HTTP.
- MAX_REALM_LEN: Lunghezza massima del campo `realm` nell'autenticazione.

- `MAX_ALLOW_LEN`: Lunghezza massima dell'header Allow.
- `HTTP_VERSION`: Versione del protocollo HTTP.
- `AUTH_SCHEME`: Tipo di schema di autenticazione utilizzato.
- `MAX_AUTH_LEN`: Lunghezza massima del campo Authorization.
- `MAX_AUTH_SCHEME_LEN`: Lunghezza massima per lo schema di autenticazione nelle richieste.
- `MESSAGE_400`: HTML di risposta per errore 400 (Bad Request).
- `MESSAGE_400_LEN`: Lunghezza del messaggio HTML per errore 400.
- `MESSAGE_401`: HTML di risposta per errore 401 (Unauthorized).
- `MESSAGE_401_LEN`: Lunghezza del messaggio HTML per errore 401.
- `MESSAGE_404`: HTML di risposta per errore 404 (Not Found).
- `MESSAGE_404_LEN`: Lunghezza del messaggio HTML per errore 404.
- `MESSAGE_414`: HTML di risposta per errore 414 (Request-URI Too Long).
- `MESSAGE_414_LEN`: Lunghezza del messaggio HTML per errore 414.
- `MESSAGE_500`: HTML di risposta per errore 500 (Internal Server Error).
- `MESSAGE_500_LEN`: Lunghezza del messaggio HTML per errore 500.

Listing 3: definizione strutture

```
#ifndef TYPES
#define TYPES
typedef struct request{
    char method[MAX_METHOD_LEN];
    char uri[MAX_URI_LEN];
    char connection[MAX_CONNECTION_LEN];
    long int content_length;
    char auth_scheme[MAX_AUTH_SCHEME_LEN];
    char authorization[MAX_AUTH_LEN];
    char *content;
} request;

typedef struct response{
    int status_code;
    char reason_phrase[MAX_REASON_PHRASE_LEN];
    long int content_length;
    char location[MAX_URI_LEN];
    char connection[MAX_CONNECTION_LEN];
    char allow[MAX_ALLOW_LEN];
    char WWW_Authenticate[MAX_REALM_LEN];
    char *content;
} response;

typedef struct event_t{
    int epoll_fd;
    int socket_fd;
} event_t;
#endif
```

- **request:** Rappresenta una richiesta HTTP:
 - **method:** stringa contenente il metodo HTTP;
 - **uri:** stringa contenente l'URI richiesto;
 - **connection:** stringa che contiene il valore dell'header Connection;
 - **content_length:** lunghezza del corpo della richiesta;
 - **auth_scheme:** schema di autenticazione usato;
 - **authorization:** stringa che conterrà le credenziali dell'header Authorization;
 - **content:** puntatore al corpo della richiesta.
- **response:** Rappresenta una risposta HTTP:
 - **status_code:** codice di stato HTTP;
 - **reason_phrase:** frase associata al codice di stato;
 - **content_length:** lunghezza del corpo della risposta;
 - **location:** contiene l'URI di una risorsa.
 - **connection:** valore dell'header Connection;

- **allow**: contiene i metodi consentiti su una specifica risorsa;
 - **WWW_Authenticate**: indica al client che deve autenticarsi;
 - **content**: puntatore al corpo della risposta.
- **event_t**: Rappresenta un evento epoll, legato a un socket, che deve essere elaborato:
 - **epoll_fd**: file descriptor dell'istanza epoll;
 - **socket_fd**: file descriptor della **socket** client associata all'evento.

A queste definizioni segue la lista dei prototipi delle funzioni dei vari moduli, che tuttavia, in quanto solo prototipi, non è necessario approfondire.

5.2 server

Il modulo server contiene la logica di gestione delle connessioni dei client e la ricezione e distribuzione degli eventi da elaborare.

Listing 4: variabili globali

```
#include "header.h"
struct sockaddr_in addr;
socklen_t addr_len = sizeof(addr);
```

- `addr`: Indirizzo a cui il server verrà associato.
- `addr_len`: Dimensione della struttura che contiene un indirizzo IP.

Listing 5: `main()`

```
int main(){
    int master_fd = listening_socket(&addr, addr_len);
    int epoll_fd = epoll_initialize();
    struct epoll_event events[MAX_EVENTS];
    if(epoll_add(master_fd, epoll_fd, 0)){
        close(master_fd);
        exit(EXIT_FAILURE);
    }
    int timeout;
    int num_events = 0;
    while (1){
        timeout = 60000;
        num_events = epoll_wait(epoll_fd, events, MAX_EVENTS,
                                timeout);
        if(num_events > 0){
            for (int i = 0; i < num_events; i++) {
                if(events[i].events & EPOLLIN){
                    if(events[i].data.fd == master_fd){
                        thread_manager(epoll_fd, events[i].data.fd,
                                      (void *) acceptance_thread);
                    }
                    else{
                        thread_manager(epoll_fd, events[i].data.fd,
                                      (void *) task_thread);
                    }
                }
            }
        }
        else if (num_events < 0){
            perror("epoll_wait failed");
            close(master_fd);
            break;
        }
    }
}
```

La funzione `main()` è il punto d'ingresso dell'esecuzione del programma. Come anticipato verranno gestite le connessioni in ingresso e gli eventi tramite `epoll`, dopodiché ad ogni evento sarà elaborato da un thread indipendente.

Nel dettaglio, nel codice troviamo:

- `master_fd`: `socket` del server, viene inizializzato, posto in ascolto e restituito dalla funzione `listening_socket()`.
- `epoll_fd`: File descriptor di `epoll`, viene inizializzato dalla funzione `epoll_initialize()`.
- `events[]`: Conterrà gli eventi gestiti da `epoll`.
- `epoll_add()`: Aggiunge `master_fd` ai `socket` monitorati da `epoll`.
- `timeout`: Conterrà il tempo limite di attesa di `epoll_wait()`.
- `num_events`: Conterrà il numero di eventi verificati sui `socket` monitorati da `epoll`.
- `epoll_wait()`: Aspetta che si verifichino eventi sui `socket` monitorati.
- `if(num_events > 0)`: Controlla la presenza di nuovi eventi.
- `for(...)`: Itera per tutti gli eventi.
- `if(events[i].events & EPOLLIN)`: Controlla se l'evento è un evento di input.
- `if(events[i].data.fd == master_fd)`: Se l'evento di scrittura avviene sul `master_fd`, allora un client sta provando a connettersi. Quindi:
 - `thread_manager(...)`: Richiede a `thread_manager()` di eseguire in un thread a parte la funzione `acceptance_thread()`.
- Altrimenti:
 - `thread_manager(...)`: Richiede a `thread_manager()` di eseguire in un thread a parte la funzione `task_thread()`.
- `else if (num_events < 0)`: Se il numero di eventi è negativo si è verificato un errore.

5.3 socket_functions

Il modulo `socket_functions` contiene tutte le funzioni per la gestione dei `socket`, dall'inizializzazione del `socket` di accettazione fino all'accettazione dei nuovi `socket`.

Listing 6: socket_initialize()

```
int socket_initialize(){
    int socket_fd;

    if((socket_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        perror("socket_creation_failed");
        exit(EXIT_FAILURE);
    }

    return socket_fd;
}
```

La funzione `socket_initialize()` crea un `socket` TCP utilizzando il dominio IPv4 e ne restituisce il file descriptor. Se la creazione del `socket` fallisce, viene stampato un messaggio d'errore e il processo termina con codice di errore.

Listing 7: socket_bind()

```
void socket_bind(int socket_fd, struct sockaddr_in *addr, socklen_t
    addr_len){
    addr->sin_family = AF_INET;
    addr->sin_addr.s_addr = INADDR_ANY;
    addr->sin_port = htons(PORT);

    if(bind(socket_fd, (struct sockaddr *) addr, addr_len) < 0){
        perror("binding_failed");
        close(socket_fd);
        exit(EXIT_FAILURE);
    }
}
```

La procedura `socket_bind()` inizializza la struttura `sockaddr_in` passata in ingresso, configurandola per accettare connessioni da qualsiasi indirizzo IP locale sulla porta specificata tramite `PORT`. Successivamente, associa questa struttura al `socket` identificato da `socket_fd`. In caso di errore nel *binding*, la funzione stampa un messaggio diagnostico, chiude il `socket` e termina il processo con codice di errore.

Listing 8: socket_listen()

```
void socket_listen(int socket_fd){
    if(listen(socket_fd, MAX_EVENTS) < 0){
        perror("listening_failed");
        close(socket_fd);
        exit(EXIT_FAILURE);
    }
}
```

La procedura `socket_listen()` imposta il `socket`, il cui file descriptor è passato come parametro in ingresso, in stato di ascolto. Se si verifica un errore, stampa un messaggio di errore, chiude il file descriptor e termina il processo con un valore di errore.

Listing 9: set_nonblocking()

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

La procedura `set_nonblocking()` imposta il `socket` in modalità non bloccante.

Listing 10: listening_socket()

```
int listening_socket(struct sockaddr_in *addr, socklen_t addr_len){
    int socket_fd = socket_initialize();
    socket_bind(socket_fd, addr, addr_len);
    socket_listen(socket_fd);
    set_nonblocking(socket_fd);
    return socket_fd;
}
```

La funzione `listening_socket()` prende in ingresso una struttura `sockaddr_in` che conterrà un indirizzo IP e una variabile `socklen_t` che contiene la grandezza della struttura. Restituisce il file descriptor di un nuovo `socket`, associato all'indirizzo `INADDR_ANY`, impostato come non bloccante e in stato di ascolto.

Listing 11: socket_accept()

```
int socket_accept(int server_fd, struct sockaddr * addr, socklen_t
* addr_len){
    int new_socket;
    if((new_socket = accept(server_fd, addr, addr_len)) == -1){
        if(errno != EWOULDBLOCK && errno != EAGAIN){
            perror("accept failed");
            return -1;
        }
    }
    set_nonblocking(new_socket);
    return new_socket;
}
```

La funzione `socket_accept()` accetta una connessione in ingresso sul `socket server_fd` e ne restituisce il file descriptor dopo aver reso il `socket` non bloccante. Se si verifica un errore nell'accettazione e tale errore non è né `EWOULDBLOCK` né `EAGAIN`, che sono errori che possono normalmente verificarsi utilizzando un `socket` non bloccante, viene stampato un messaggio di errore e restituito un valore anomalo.

5.4 epoll_functions

Il modulo `epoll_functions` contiene tutte le funzioni per la gestione di `epoll`, dall'inizializzazione fino all'aggiunta e rimozione di nuovi socket da monitorare.

Listing 12: `epoll_initialize()`

```
int epoll_initialize(){
    int epoll_fd;
    if((epoll_fd = epoll_create1(0)) == -1){
        perror("epoll_create1 failed");
        exit(EXIT_FAILURE);
    }
    return epoll_fd;
}
```

La funzione `epoll_initialize()` crea un'istanza di `epoll()` e ne restituisce il file descriptor. Se la creazione fallisce, viene stampato un messaggio d'errore e il processo termina con codice di errore.

Listing 13: `epoll_add()`

```
int epoll_add(int socket_fd, int epoll_fd, int et){
    struct epoll_event event;
    if(et)
        event.events = EPOLLIN | EPOLLET;
    else
        event.events = EPOLLIN;
    event.data.fd = socket_fd;
    if(epoll_ctl(epoll_fd, EPOLL_CTL_ADD, socket_fd, &event) == -1)
    {
        perror("epoll_ctl failed");
        return 1;
    }
    return 0;
}
```

La funzione `epoll_add()` prende in ingresso i file descriptor di un `socket` e di un'istanza di `epoll()` e aggiunge il `socket` alla lista dei sockets monitorati da `epoll` sugli eventi di scrittura, che possono essere edge triggered o no in base a valore dle parametro `et`. Se l'operazione fallisce, viene stampato un messaggio d'errore e il restituito il valore 1.

Listing 14: `epoll_remove()`

```
void epoll_remove(int epoll_fd, int fd){
    epoll_ctl(epoll_fd, EPOLL_CTL_DEL, fd, NULL);
    close(fd);
}
```

La procedura `epoll_remove()` prende in ingresso i file descriptor di un `socket` e di un'istanza di `epoll()` e rimuove il `socket` alla lista dei sockets monitorati da `epoll` per poi chiuderne il file descriptor.

5.5 thread functions

Il modulo `thread_functions` contiene tutte le funzioni per la gestione delle task assegnate ai threads, dall'accettazione di nuovi socket fino all'elaborazione di richieste HTTP.

Listing 15: `thread_manager()`

```
void thread_manager(int epoll_fd, int socket_fd, void *function){
    pthread_t thread;
    event_t *arg = malloc(sizeof(event_t));
    arg->epoll_fd = epoll_fd;
    arg->socket_fd = socket_fd;
    pthread_create(&thread, NULL, (void *) function, arg);
    pthread_detach(thread);
}
```

La procedura `thread_manager()` prende in ingresso un puntatore a funzione `function` e i file descriptor di un socket e un'istanza di `epoll()`.

- Inizializza con `malloc()` un puntatore ad una struttura `event_t`.
- I campi di tale struttura vengono inizializzati con i valori dei due file descriptor.
- Viene creato tramite `pthread_create()` un nuovo thread che esegue la funzione puntata da `function` e prende come argomento il puntatore alla struttura sopra citata.
- Il nuovo thread viene infine scollegato dal thread principale.

Listing 16: `acceptance_thread()`

```
void *acceptance_thread(event_t *arg){
    pthread_mutex_lock(&accept_mutex);
    int new_socket;
    if((new_socket = socket_accept(arg->socket_fd, (struct
        sockaddr *) &addr, &addr_len)) > 0){
        if(epoll_add(new_socket, arg->epoll_fd, 1)){
            close(new_socket);
        }
    }
    free(arg);
    pthread_mutex_unlock(&accept_mutex);
    pthread_exit(NULL);
}
```

La procedura `acceptance_thread()` prende in ingresso un puntatore a una struttura `event_t`.

- Acquisisce un mutex di accettazione, per evitare race conditions su `addr` e `addrlen`.
- Accetta la nuova connessione e la aggiunge alla lista di connessioni monitorate da `epoll`.

- Se si verificano errori viene chiuso il file descriptor del `socket`.
- Vengono liberate le risorse non più utili e viene rilasciato il lock sul `mutex`.

Listing 17: `task_thread()`

```
void *task_thread(event_t *arg){
    char buffer[BUFFER_SIZE];
    response res;
    request req;
    int n;
    while((n = recv(arg->socket_fd, buffer, BUFFER_SIZE - 1, 0)) >
0){
        buffer[n] = '\0';
        char *message = buffer;
        while(*message != '\0'){
            if(!parse(&message, &req, &res)){
                elaborate_request(&req, &res);
            }
            http_send(&res, arg->socket_fd);
            if((strcmp(res.connection, "close") == 0) || (strcmp(req
            .connection, "close") == 0)){
                epoll_remove(arg->epoll_fd, arg->socket_fd);
                free(arg);
                pthread_exit(NULL);
            }
        }
    }
    if(errno != EWOULDBLOCK && errno != EAGAIN){
        epoll_remove(arg->epoll_fd, arg->socket_fd);
    }
    free(arg);
    pthread_exit(NULL);
}
```

La procedura `task_thread()` prende in ingresso un puntatore a una struttura `event_t`. Riceve in input una o più richieste da parte del client, effettua il parsing delle richieste una ad una e le elabora per formulare una risposta da inviare. Nel dettaglio:

- Receve una o più richieste in input.
- Entra in un ciclo che continua finché continuano ad arrivare richieste.
- Entra in un ulteriore ciclo che gli impedisce di ricevere le nuove richieste finché non ha elaborato quelle precedenti.
- Effettua il parsing della prima richiesta tramite la funzione `parse()`.
- Se la richiesta non presenta errori di sintassi viene elaborata tramite `elaborate_request()`.
- Dopo l'elaborazione viene subito inviata una risposta al client tramite `http_send()`.
- Se è necessario, chiude la connessione.

- Quando il ciclo finisce, se si sono verificati errori di connessione il file descriptor del `socket` viene rimosso da `epoll` e chiuso.
- Infine libera le risorse usate e termina il thread.

5.6 parsing_functions

Listing 18: `parse()`

```
int parse(char **message, request *req, response *res){
    int i = get_method(*message, req, res);
    if(i == -1){
        return 1;
    }
    if(get_uri(*message, i, req, res)){
        return 1;
    }
    if(find_newline(*message) || find_host(*message) ||
       get_connection(*message, req)){
        res->status_code = 400;
        return 1;
    }
    char *found;
    found = strstr(*message, "\r\n\r\n");
    if(found == NULL){
        res->status_code = 400;
        return 1;
    }
    found += strlen("\r\n\r\n");
    if(req->method[0] == 'P'){
        if(get_body(*message, req)){
            res->status_code = 400;
            return 1;
        }
        *message = found + req->content_length;
        if(!strcmp(req->method, "PUT")){
            if((strstr(*message, "Content-Encoding") != NULL) || (
                strstr(*message, "Content-Location") != NULL) || (
                strstr(*message, "Content-Range") != NULL)){
                res->status_code = 501;
                return 1;
            }
        }
    }
    else{
        *message = found;
    }
    if(get_authorization(*message, req)){
        res->status_code = 400;
        return 1;
    }
    return 0;
}
```

La funzione `parse()` effettua il parsing del messaggio e determina se è conforme allo standard RFC 2616. Prende in ingresso un puntatore a puntatore di caratteri, un puntatore alla struttura `request` che rappresenta la richiesta e un

puntatore alla struttura **response** che rappresenta la risposta del server. Nel dettaglio la funzione:

- Copia il metodo della richiesta dal messaggio al campo **method** della struttura **request** tramite la funzione **get_method()**.
- Copia l'URI della richiesta dal messaggio al campo **uri** della struttura **request** tramite la funzione **get_uri()**.
- Controlla la presenza di "new line" composte da `\r\n\r\n` tramite il metodo **find_newline()**.
- Verifica la presenza dell'header Host tramite **fin_host()**.
- Verifica la presenza della riga vuota di terminazione dell'header.
- Se il metodo è PUT o POST, deve necessariamente avere un corpo del messaggio quindi:
 - Copia il contenuto della richiesta dal messaggio al campo **content** della struttura **request** tramite la funzione **get_body()**.
 - Fa avanzare il puntatore ***message** per segnalare che la richiesta è stata elaborata.
 - PUT richiede di non ignorare alcun header del tipo Content-*, quindi se una funzione non è implementata restituisce quindi 501.
- Infine copia il contenuto dell'header Authorization dal messaggio alla struttura **request** tramite **get_authorization()**.

Listing 19: **get_method()**

```
int get_method(char message[], request *req, response *res){
    int i = 0;
    while(message[i] != '\0'){
        if(i >= MAX_METHOD_LEN){
            res->status_code = 400;
            return -1;
        }
        req->method[i] = message[i];
        i++;
    }
    req->method[i] = '\0';
    i++;
    if((strcmp(req->method,"GET") != 0) && (strcmp(req->method,"
    POST") != 0) && (strcmp(req->method,"PUT") != 0) && (strcmp
    (req->method,"DELETE") != 0)){
        res->status_code = 400;
        return -1;
    }
    return i;
}
```

La funzione `get_method()` riceve in input una stringa `message`, un puntatore a una struttura `request` e un puntatore a una struttura `response`. Copia il metodo dalla richiesta HTTP all'apposito campo della struttura `request`. Se il metodo risultasse, già durante la fase di copia, troppo lungo o se non corrispondesse a nessuno dei metodi accettati verrebbe impostato come codice di stato di `response` 400 e la funzione restituirebbe -1.

Listing 20: `get_uri()`

```
int get_uri(char message[], int i, request *req, response *res){
    int j = 0;
    while(message[i] != '\0'){
        if(j >= MAX_URI_LEN){
            res->status_code = 414;
            return 1;
        }
        req->uri[j] = message[i];
        i++;
        j++;
    }
    req->uri[j] = '\0';
    if(strlen(req->uri) == 0){
        res->status_code = 400;
        return 1;
    }
    return 0;
}
```

La funzione `get_uri()` riceve in input una stringa `message`, un intero `i`, un puntatore a una struttura `request` e un puntatore a una struttura `response`. Copia l'URI dalla richiesta HTTP all'apposito campo della struttura `request`.

- Se l'URI risultasse, già durante la fase di copia, troppo lungo verrebbe impostato come codice di stato di `response` 414 e la funzione restituirebbe 1.
- Se l'URI non fosse presente verrebbe impostato come codice di stato di `response` 400 e la funzione restituirebbe 1.

Listing 21: `find_newline()`

```
int find_newline(char message[]){
    char *found;
    if((found = strstr(message, "\r\n")) == NULL){
        return 1;
    }
    return 0;
}
```

La funzione `find_newline()` riceve in input una stringa contenente un messaggio HTTP e verifica la presenza dei caratteri di fine linea, obbligatori nei pacchetti HTTP. In caso non fosse presente nemmeno un newline, la funzione ritornerebbe 1.

Listing 22: find_host()

```
int find_host(char message[]){
    char *found;
    if((found = strstr(message, "Host:")) == NULL){
        return 1;
    }
    return 0;
}
```

La funzione `find_host()` riceve in input una stringa contenente un messaggio HTTP e verifica la presenza dell'header `Host`, obbligatoria secondo l'RFC 2616.

Listing 23: get_connection()

```
int get_connection(char message[], request *req){
    char *found;
    int i;
    found = strstr(message, "Connection:");
    if(found != NULL){
        found += strlen("Connection:");
        while(found[0] == ' '){
            found++;
        }
        i = 0;
        while(found[i] != '\r' && found[i] != '\n' && found[i] != '\n'){
            req->connection[i] = found[i];
            i++;
        }
        req->connection[i] = '\0';
        if((strcmp(req->connection, "keep-alive") != 0) && (strcmp(
            req->connection, "close") != 0)){
            return 1;
        }
    }
    else{
        strcpy(req->connection, "keep-alive");
    }
    return 0;
}
```

La funzione `get_connection()` riceve in input una stringa `message` e un puntatore a una struttura `request`. Copia l'header `Connection` dalla richiesta HTTP all'apposito campo della struttura `request`, se non presente il valore assegnato di default è `"keep-alive"`. Se però l'header `Connection` è presente deve essere necessariamente `"close"` oppure `"keep-alive"`.

Listing 24: get_body()

```
int get_body(char message[], request *req){
    char *found;
    found = strstr(message, "Content-Length:");
    if(found == NULL){
        return 1;
    }
    found += strlen("Content-Length:");
    req->content_length = atoi(found);
    if(req->content_length <= 0){
        return 1;
    }
    found = strstr(message, "\r\n\r\n");
    if(found == NULL){
        return 1;
    }
    found += strlen("\r\n\r\n");
    req->content = malloc(req->content_length);
    for(int i = 0; i < req->content_length; i++){
        req->content[i] = found[i];
    }
    return 0;
}
```

La funzione `get_body()` riceve in input una stringa `message` rappresentante una richiesta HTTP e un puntatore a una struttura `request`. Copia il contenuto dell'header `Content-Lenght` all'interno dell'apposito campo della struttura `request`. Dopodiché si reca alla fine dell'header e copia `content_length` byte dal messaggio all'apposito campo della struttura `request`.

Listing 25: get_authorization()

```
int get_authorization(char message[], request *req){
    char *found;
    if((found = strstr(message, "Authorization:")) != NULL){
        found += strlen("Authorization:");
        while(*found == ' '){
            found++;
        }
        int i = 0;
        while(found[i] != '\r' && found[i] != '\n' && found[i] != '\0'){
            if(i >= MAX_AUTH_LEN){
                return 1;
            }
            req->auth_scheme[i] = found[i];
            i++;
        }
        req->auth_scheme[i] = '\0';
        found += strlen(req->auth_scheme);
        while(*found == ' '){
            found++;
        }

        i = 0;
        while(found[i] != '\r' && found[i] != '\n' && found[i] != '\0'){
            if(i >= MAX_AUTH_LEN){
                return 1;
            }
            req->authorization[i] = found[i];
            i++;
        }
        req->authorization[i] = '\0';
    }
    else{
        req->auth_scheme[0] = '\0';
        req->authorization[0] = '\0';
    }
    return 0;
}
```

La funzione `get_authorization()` riceve in input una stringa `message` rappresentante una richiesta HTTP e un puntatore a una struttura `request`. Copia la prima parola dell'header Authorization all'interno del campo `auth_scheme` della struttura `request`, dopodiché copia la seconda parola dell'header Authorization all'interno del campo `authorization` della struttura `request`.

5.7 http_methods

Il modulo `http_methods` contiene l'implementazione di tutti i metodi HTTP accettati dal server: GET, POST, PUT e DELETE.

Listing 26: `elaborate_request()`

```
int elaborate_request(request *req, response *res){
    char path[MAX_URI_LEN + strlen("files") + strlen("index.html")
              + 1];
    sprintf(path, "files");
    if(build_path(req->uri, path)){
        res->status_code = 404;
        return 1;
    }
    if(!is_authorized(path, req, res)){
        return 1;
    }
    if(!strcmp(req->method, "GET")){
        return get(path, req, res);
    }
    else if(!strcmp(req->method, "POST")){
        return post(path, req, res);
    }
    else if(!strcmp(req->method, "PUT")){
        return put(path, req, res);
    }
    else if(!strcmp(req->method, "DELETE")){
        return delete(path, req, res);
    }
    res->status_code = 400;
    return 1;
}
```

La funzione `elaborate_request()` riceve in input un puntatore a una struttura `request` e un puntatore a una struttura `response`. La funzione trasforma l'uri presente nella richiesta in un percorso file valido tramite `build_path()` e verifica tramite `is_authorized()` se l'operazione sul file è un'operazione autorizzata. Infine richiama la funzione corrispondente al metodo della richiesta.

Listing 27: build_path()

```
int build_path(char uri[], char path[]){
    if (strstr(uri, "..") != NULL || strstr(uri, "//") != NULL) {
        return 1;
    }
    else{
        char tmp[MAX_URI_LEN + strlen("files") + strlen("index.html") + 1];
        strcpy(tmp, path);
        snprintf(path, MAX_URI_LEN + strlen("files") + strlen("index.html") + 1, "%s%s", tmp, uri);
        if(path[strlen(path)-1] == '/'){
            strcpy(tmp, path);
            snprintf(path, MAX_URI_LEN + strlen("files") + strlen("index.html") + 1, "%sindex.html", tmp);
        }
        return 0;
    }
}
```

La funzione `build_path()` riceve in input una stringa rappresentante l'uri della richiesta HTTP e una stringa `path`:

- Controlla la presenza di sequenze potenzialmente pericolose come `".."` oppure `"//"` e in caso, restituisce 1.
- Costruisce, a partire dall'URI, il vero percorso del file, salvando dentro `path` l'uri della richiesta, preceduto da `"files"`.
- Se l'uri termina con `"/"` inoltre verrà concatenata al `path` la stringa `"index.html"`.

Listing 28: check_index()

```
int check_index(char path[], response *res){
    if(!strcmp(path, "files/index.html")){
        res->status_code = 405;
        strcpy(res->allow, "GET");
        return 1;
    }
    return 0;
}
```

La funzione `check_index()` riceve in input una stringa rappresentante il percorso della risorsa richiesta e un puntatore alla struttura `response`: impedisce alle richieste diverse da GET di accedere a `"files/index.html"`.

Listing 29: is_authorized()

```
int is_authorized(char path[], request *req, response *res){
    int authorized = 0;
    char *found;
    if((found = strstr(path, "/private/")) != NULL){
        sprintf(res->WWW_Authenticate, "private");
        if(strcmp(req->auth_scheme, AUTH_SCHEME) != 0){
            res->status_code = 401;
            return authorized;
        }
        if(!read_with_lock("configuration/config.txt", req, res)){
            if(!strcmp(req->authorization, res->content)){
                authorized = 1;
            }
            else{
                res->status_code = 401;
            }
            free(res->content);
        }
    }
    else{
        authorized = 1;
    }
    return authorized;
}
```

La funzione `is_authorized()` riceve in input una stringa rappresentante il percorso della risorsa richiesta, un puntatore a una struttura `request` e un puntatore a una struttura `response`:

- Controlla se l'accesso alla risorsa necessita di autorizzazione.
- In caso sia necessaria, confronta lo schema di crittografia della richiesta con lo schema di crittografia del server, se non corrispondono la richiesta viene respinta, altrimenti la funzione prosegue.
- Legge le credenziali per l'autorizzazione dal file di configurazione tramite la funzione `read_with_lock()` e le confronta con le credenziali della richiesta. In caso combacino l'operazione può proseguire, altrimenti verrà inviato un messaggio di errore.
- In entrambi i controlli il messaggio di errore restituito è il messaggio 401 (Unauthorized).

5.7.1 GET

Il metodo GET (figure 1) è utilizzato per richiedere al server una informazione identificata dall'URI della richiesta. In questa implementazione ogni messaggio GET viene interpretato come la richiesta di leggere il contenuto del file identificato dall'URI.

"GET"	SP	URI	SP	"HTTP/1.1"	CRLF
"Host:"	1*LWS	host [":" port]			CRLF
CRLF					

SP = space = ' '

LWS = linear white space = ' ' | '\t'

CRLF= carriage return line feed = "\r\n"

Figure 1: Messaggio GET HTTP 1.1

Il metodo GET non possiede header obbligatori oltre all'header **Host** che è obbligatorio per tutti i metodi.

Il metodo GET può ricevere una sola risposta se non si verificano errori:

- 200 OK: L'operazione è stata eseguita con successo. La risposta conterrà nel proprio entity-body il contenuto della risorsa richiesta.

Listing 30: get()

```
int get(char path[], request *req, response *res){
    res->status_code = 200;
    return read_with_lock(path, req, res);
}
```

La funzione `get()` riceve in input una stringa rappresentante il percorso della risorsa richiesta, un puntatore a una struttura `request` e un puntatore a una struttura `response`:

- Imposta il valore standard della risposta (200).
- Richiama la funzione `read_with_lock()` per leggere il contenuto della risorsa richiesta.

5.7.2 POST

Il metodo POST (figure 2) viene utilizzato per richiedere al server di accettare l'entità inclusa nella richiesta come nuova entità, subordinata alla risorsa identificata dall'URI della richiesta. In questa implementazione ogni messaggio POST viene interpretato come la richiesta di scrivere in modalità append sul file identificato dall'URI.

"POST"	SP	URI	SP	"HTTP/1.1"	CRLF
"Host:"	1*LWS	host [":" port]			CRLF
"Content-Length:"		1*LWS	1*DIGIT		CRLF
CRLF					
entity-body					

SP = space = ' '

LWS = linear white space = ' ' | '\t'

CRLF= carriage return line feed = "\r\n"

Figure 2: Messaggio POST HTTP 1.1

Il metodo POST possiede un header obbligatorio:

- **Content-Length**: indica la dimensione, in byte, dell'**entity-body**.

Il metodo POST può ricevere due diverse risposte se non si verificano errori:

- **200 OK**: L'operazione è stata eseguita con successo. La risposta conterrà nel proprio **entity-body** il contenuto della risorsa modificata.
- **201 Created**: Il file identificato dall'URI non esisteva ed è quindi stato creato. La risposta conterrà, nel proprio header **Location**, l'URI della risorsa e nel proprio **entity-body** il contenuto della risorsa creata.

Listing 31: post()

```
int post(char path[], request *req, response *res){
    if(check_index(path,res)){
        return 1;
    }
    if(!access(path, F_OK)){
        res->status_code = 200;
    }
    else{
        strcpy(res->location, req->uri);
        res->status_code = 201;
    }
    return write_read_with_lock(path, req, res);
}
```

La funzione `post()` riceve in input una stringa rappresentante il percorso della risorsa richiesta, un puntatore a una struttura `request` e un puntatore a una struttura `response`:

- Effettua un controllo per evitare di modificare `index.html`.
- Imposta il valore standard della risposta:
 - 200 (OK) se la risorsa esiste già;
 - 201 (Created) se la risorsa deve essere creata.
- Richiama la funzione `write_read_with_lock()` per modificare e leggere il contenuto della risorsa indicata.

5.7.3 PUT

Il metodo PUT (figure 3) richiede che l'entità racchiusa venga memorizzata nell'URI fornito nella richiesta. In questa implementazione ogni messaggio PUT viene interpretato come la richiesta di creare o sovrascrivere il file identificato dall'URI.

"PUT"	SP	URI	SP	"HTTP/1.1"	CRLF
"Host:"	1*LWS	host [":" port]			CRLF
"Content-Length:"		1*LWS	1*DIGIT		CRLF
CRLF					
entity-body					

SP = space = ' '

LWS = linear white space = ' ' | '\t'

CRLF= carriage return line feed = "\r\n"

Figure 3: Messaggio PUT HTTP 1.1

Il metodo PUT possiede un header obbligatorio:

- **Content-Length**: indica la dimensione, in byte, dell'entity-body.

Il metodo PUT può contenere anche altri header "**Content-***" e se il server non implementa quelle funzioni dovrà necessariamente respingere la richiesta. Il metodo PUT può ricevere due diverse risposte se non si verificano errori:

- **201 Created**: Il file identificato dall'URI non esisteva ed è quindi stato creato. La risposta conterrà, nel proprio header **Location**, l'URI della risorsa creata.
- **204 No Content**: Il file identificato dall'URI esisteva ed è quindi sovrascritto.

Listing 32: put()

```
int put(char path[], request *req, response *res){
    if(check_index(path,res)){
        return 1;
    }
    strcpy(res->location, req->uri);
    if(!access(path, F_OK)){
        res->status_code = 204;
    }
    else{
        res->status_code = 201;
    }

    res->content_length = 0;
    res->content = NULL;

    return write_with_lock(path, req, res);
}
```

La funzione `put()` riceve in input una stringa rappresentante il percorso della risorsa richiesta, un puntatore a una struttura `request` e un puntatore a una struttura `response`:

- Effettua un controllo per evitare di sovrascrivere `index.html`.
- Salva in `res->location` l'URI della risorsa che verrà creata.
- Imposta il valore standard della risposta:
 - 204 (No Content) se la risorsa esiste già;
 - 201 (Created) se la risorsa deve essere creata.
- Imposta il contenuto della risposta come vuoto.
- Richiama la funzione `write_with_lock()` per creare o sovrascrivere la risorsa indicata.

5.7.4 DELETE

Il metodo DELETE (figure 4) richiede che il server elimini la risorsa indicata nell'URI fornito nella richiesta. In questa implementazione ogni messaggio DELETE viene interpretato come la richiesta di cancellare il file identificato dall'URI.

"DELETE"	SP	URI	SP	"HTTP/1.1"	CRLF
"Host:"	1*LWS	host [":" port]			CRLF
CRLF					

SP = space = ' '

LWS = linear white space = ' ' | '\t'

CRLF= carriage return line feed = "\r\n"

Figure 4: Messaggio DELETE HTTP 1.1

Il metodo DELETE non possiede header obbligatori oltre all'header `Host` che è obbligatorio per tutti i metodi.

Il metodo DELETE può ricevere una sola risposta se non si verificano errori:

- **202 Accepted:** La richiesta è stata accettata ma non ancora eseguita.

Questo codice di stato indica un'operazione che è stata avviata ma verrà conclusa in modalità asincrona: l'eliminazione del file difatti non avviene finché non vengono chiusi tutti i file descriptor del file.

Listing 33: delete()

```
int delete(char path[], request *req, response *res){
    if(check_index(path,res)){
        return 1;
    }
    res->status_code = 202;
    return unlink_with_lock(path, req, res);
}
```

La funzione `delete()` riceve in input una stringa rappresentante il percorso della risorsa richiesta, un puntatore a una struttura `request` e un puntatore a una struttura `response`:

- Effettua un controllo per evitare di eliminare `index.html`.
- Imposta il valore standard della risposta (202).
- Richiama la funzione `unlink_with_lock()` per cancellare la risorsa indicata.

5.8 file_functions

Il modulo `file_functions` contiene tutte le funzioni per la gestione dei files nel programma.

Listing 34: `open_file()`

```
int open_file(char path[], char method[]){
    int file_fd;
    char *found;
    if((found = strstr(path,"configuration/")) == NULL){
        if(!strcmp(method, "GET")){
            file_fd = open(path, O_RDONLY, 0644);
        }
        else if(!strcmp(method, "POST")){
            file_fd = open(path, O_RDWR | O_APPEND | O_CREAT, 0644);
        }
        else if(!strcmp(method, "PUT")){
            file_fd = open(path, O_WRONLY | O_CREAT | O_TRUNC,
                           0644);
        }
        else if(!strcmp(method, "DELETE")){
            file_fd = open(path, O_RDWR, 0644);
        }
    }
    else{
        file_fd = open(path, O_RDONLY, 0644);
    }
    return file_fd;
}
```

La funzione `open_file()` riceve in input una stringa rappresentante il percorso di un file, una stringa rappresentante il metodo HTTP che vuole accedere al file e restituisce il file descriptor del file:

- Se il file è un file di configurazione viene aperto solo in lettura, altrimenti:
 - Se il metodo è GET il file viene aperto solo in lettura.
 - Se il metodo è POST il file viene aperto solo in scrittura, in modalità append.
 - Se il metodo è PUT il file viene aperto solo in scrittura, con il tag `O_TRUNC` che svuota il file prima di scrivervi.
 - Se il metodo è DELETE il file viene aperto sia in lettura che in scrittura.

Listing 35: lock_file()

```
int lock_file(struct flock *lock, int fd){
    lock->l_whence = SEEK_SET;
    lock->l_start = 0;
    lock->l_len = 0;
    if(fcntl(fd, F_SETLKW ,lock) == -1){
        close(fd);
        return 1;
    }
    return 0;
}
```

La funzione `lock_file()` riceve in input un puntatore ad una struttura `flock`, il file descriptor di un file aperto e effettua il lock sul file:

- Grazie al flag `F_SETLKW` se il file è già sottoposto ad un lock, il thread aspetta che lock venga rilasciato.
- Se il lock avviene in un file aperto in lettura, tutti i threads possono leggere il file senza problemi ma nessun thread può modificarlo.
- Se il lock avviene in un file aperto in scrittura, nessun altro thread può leggere o scrivere sul file.

Listing 36: unlock_file()

```
int unlock_file(struct flock *lock, int fd){
    lock->l_type = F_UNLCK;
    int ret = (fcntl(fd, F_SETLKW ,lock) == -1);
    close(fd);
    return ret;
}
```

La funzione `unlock_file()` riceve in input un puntatore ad una struttura `flock`, il file descriptor di un file aperto e rilascia il lock posto sul file.

Listing 37: write_read_with_lock()

```
int write_read_with_lock(char path[], request *req, response *res){
    if(write_with_lock(path, req, res)){
        return 1;
    }
    return read_with_lock(path, req, res);
}
```

La funzione `write_read_with_lock()` riceve in input una stringa rappresentante il percorso di un file, un puntatore a una struttura `request` e un puntatore a una struttura `response`. Scrive il `req->content` sul file identificato da `path` per poi leggere tutto il contenuto del file su `res->content`.

Listing 38: write_with_lock()

```
int write_with_lock(char path[], request *req, response *res){
    int file_fd = open_file(path, req->method);
    if(file_fd < 0){
        res->status_code = 404;
        return 1;
    }

    struct flock file_lock;
    file_lock.l_type = F_WRLCK;
    if(lock_file(&file_lock, file_fd)){
        res->status_code = 500;
        return 1;
    }

    write(file_fd, req->content, req->content_length);
    fsync(file_fd);
    if(unlock_file(&file_lock, file_fd)){
        res->status_code = 500;
        return 1;
    }

    return 0;
}
```

La funzione `write_with_lock()` riceve in input una stringa rappresentante il percorso di un file, un puntatore a una struttura `request` e un puntatore a una struttura `response`. Lo scopo della funzione è:

- aprire il file indicato da `path` tramite la funzione `open_file()`,
- acquisire il lock esclusivo sul file tramite la funzione `lock_file()`,
- scrivere il contenuto di `req->content` sul file tramite `write()` e `fsync()`,
- rilasciare il lock sul file.

Listing 39: read_with_lock()

```
int read_with_lock(char path[], request *req, response *res){
    int file_fd = open_file(path, req->method);
    if(file_fd < 0){
        res->status_code = 404;
        return 1;
    }

    struct flock file_lock;
    file_lock.l_type = F_RDLCK;
    if(lock_file(&file_lock, file_fd)){
        res->status_code = 500;
        return 1;
    }

    res->content_length = lseek(file_fd, 0, SEEK_END);
    lseek(file_fd, 0, SEEK_SET);
    res->content = malloc(res->content_length+1);
    if(res->content == NULL){
        res->status_code = 500;
        file_lock.l_type = F_UNLCK;
        fcntl(file_fd, F_SETLKW, &file_lock);
        close(file_fd);
        return 1;
    }

    read(file_fd, res->content, res->content_length);
    res->content[res->content_length] = '\0';
    if(unlock_file(&file_lock, file_fd)){
        res->status_code = 500;
        return 1;
    }

    return 0;
}
```

La funzione `read_with_lock()` riceve in input una stringa rappresentante il percorso di un file, un puntatore a una struttura `request` e un puntatore a una struttura `response`. Lo scopo della funzione è:

- aprire il file indicato da `path` tramite la funzione `open_file()`,
- acquisire il lock in lettura del file tramite la funzione `lock_file()`,
- il lock in lettura permette a tutti i threads di accedervi in lettura ma a nessun thread di accedervi in scrittura,
- leggere il contenuto del file e copiarlo su `res->content` tramite `read()`,
- rilasciare il lock sul file.

Listing 40: unlink_with_lock()

```
int unlink_with_lock(char path[], request *req, response * res){
    int file_fd = open_file(path, req->method);
    if(file_fd < 0){
        res->status_code = 404;
        return 1;
    }
    struct flock file_lock;
    file_lock.l_type = F_WRLCK;
    if(lock_file(&file_lock, file_fd)){
        res->status_code = 500;
        return 1;
    }
    if(!access(path, F_OK)){
        unlink(path);
    }
    if(unlock_file(&file_lock, file_fd)){
        res->status_code = 500;
        return 1;
    }
    return 0;
}
```

La funzione `unlink_with_lock()` riceve in input una stringa rappresentante il percorso di un file, un puntatore a una struttura `request` e un puntatore a una struttura `response`. Lo scopo della funzione è:

- aprire il file indicato da `path` tramite la funzione `open_file()`,
- acquisire il lock esclusivo sul file tramite la funzione `lock_file()`,
- se il file è presente nel filesystem quando la funzione acquisisce il lock, rimuovere il file con la funzione `unlink()`,
- rilasciare il lock sul file.

5.9 sending_functions

Il modulo `sending_functions` contiene tutte le funzioni per la gestione dell'invio delle risposte ai clients.

Listing 41: `http_send()`

```
void http_send(response *res, int fd){
    char *message = malloc(MAX_HEADER_LEN + (res->content ? res->
        content_length : 0 ));
    if(message == NULL){
        message = malloc(MESSAGE_500_LEN);
        res->status_code = 500;
    }
    resolve_status_code(res);
    switch(res->status_code){
        case 201:
            sprintf(message, "HTTP/%1.1f_%d_%s\r\nConnection:_%s\r\
                nContent-Length:_%ld\r\nLocation:_%s\r\n\r\n%s",
                HTTP_VERSION, res->status_code, res->reason_phrase,
                res->connection, res->content_length, res->
                location, (res->content ? res->content : "\0"));
            break;
        case 204:
            sprintf(message, "HTTP/%1.1f_%d_%s\r\nConnection:_%s\r\
                nLocation:_%s\r\n\r\n", HTTP_VERSION, res->
                status_code, res->reason_phrase, res->connection,
                res->location);
            break;
        ...
        default:
            sprintf(message, "HTTP/%1.1f_%d_%s\r\nConnection:_%s\r\
                nContent-Type:_%text/html\r\nContent-Length:_%ld\r\n\
                \r\n%s", HTTP_VERSION, res->status_code, res->
                reason_phrase, res->connection, res->content_length
                , res->content);
    }
    send(fd, message, strlen(message), 0);
    free(message);
    if(res->content != NULL){
        free(res->content);
    }
}
```

La funzione `http_send()`, dati in input un puntatore a una struttura `response` e il file descriptor di un socket, invia al client un messaggio di risposta contenente le informazioni salvate in `res`. Nel dettaglio:

- alloca la memoria necessaria a contenere il messaggio;
- completa il messaggio HTTP di risposta tramite `resolve_status_code()`;
- formatta il messaggio HTTP di risposta tramite un costrutto `switch-case`;
- invia il messaggio HTTP di risposta tramite `send()`;
- libera le risorse utilizzate

Listing 42: resolve_status_code()

```
void resolve_status_code(response *res){
    switch (res->status_code){
        case 200:
            strcpy(res->reason_phrase, "OK");
            strcpy(res->connection, "keep-alive");
            break;
        case 201:
            strcpy(res->reason_phrase, "Created");
            strcpy(res->connection, "keep-alive");
            break;
        ...
    }
```

La funzione `resolve_status_code()` riceve in input un puntatore a una struttura `response` e completa i campi necessari alla corretta formulazione della risposta tramite un costrutto switch-case che effettua una selezione basandosi sul valore di `res->status_code`.

6 Risultati sperimentali

Il programma sviluppato è un'implementazione di un server HTTP 1.1 in C, realizzato tenendo come testo di riferimento l'RFC 2616. L'obiettivo del server è ricevere, elaborare e rispondere in maniera coerente alle richieste HTTP inviategli. Il programma deve, inoltre, essere in grado di gestire numerosi client contemporaneamente e mantenere con essi connessioni TCP stabili, ad eccezione del verificarsi di gravi errori di formulazione delle richieste o gravi errori interni al server.

Di seguito si riportano i risultati sperimentali ottenuti:

- **Fase di testing con il browser (Firefox):**

- Il server riesce a ricevere, elaborare e rispondere correttamente alle semplici richieste di GET.
- Per testare attraverso il browser anche le funzioni di POST, PUT e DELETE è stata utilizzata l'estensione RESTer², il server è riuscito a gestire correttamente tutti i casi di POST, PUT e DELETE, rispondendo e segnalando anche gli errori con i giusti codici di stato.

Una volta constatato il corretto comportamento del server con una o due connessioni sono stati effettuati dei test con connessioni progressivamente sempre maggiori sul server per osservarne il comportamento.

- **Esecuzione dei test di carico:**

- I test di carico sono stati effettuati con lo strumento di benchmarking wrk³, sono state eseguite 20 ripetizioni per ogni test di carico, rispettivamente con 100, 1000 e 10000 connessioni, attraverso uno script bash (Listing 43).

Listing 43: wrk.sh

```
#!/bin/bash
for i in {1..20}; do
    wrk -t4 -c100 -d30s http://localhost:8080/ >> wrk_get.text
    echo {$i}
done
for i in {1..20}; do
    wrk -t4 -c1000 -d30s http://localhost:8080/ >> wrk_get.text
    echo {$i}
done
for i in {1..20}; do
    wrk -t4 -c10000 -d30s http://localhost:8080/ >> wrk_get.text
    echo {$i}
done
```

²<https://kuehle.me/projects/rester/>

³<https://github.com/wg/wrk>

- La media dei test da 30 secondi con 100 connessioni riportano questi risultati:
 - **Richieste totali:** 2338631
 - **Latenza media:** 1.29 ms
 - **Richieste/sec:** 77710.86
 - **Dati trasferiti/sec:** 10.30 MB
 - **Nessun errore**
- La media dei test da 30 secondi con 1000 connessioni riportano questi risultati:
 - **Richieste totali:** 2278945
 - **Latenza media:** 16.74 ms
 - **Richieste/sec:** 73842.39
 - **Dati trasferiti/sec:** 9.80 MB
 - **Errori:** timeout 142
- La media dei test da 30 secondi con 10000 connessioni riportano questi risultati:
 - **Richieste totali:** 2113477
 - **Latenza media:** 63.51 ms
 - **Richieste/sec:** 70284.15
 - **Dati trasferiti/sec:** 9.31 MB
 - **Errori:** connect 7, read 432, timeout 568

7 Conclusione e sviluppi futuri

I risultati ottenuti attraverso i test di performance rappresentano un quadro incoraggiante delle capacità del server sviluppato. L'analisi dei dati mostra come il programma riesca a mantenere prestazioni eccellenti nel scenario con 100 connessioni concorrenti, raggiungendo oltre 77.000 richieste elaborate al secondo con una latenza media di soli 1.29 ms e l'assenza totale di errori. Questi valori testimoniano l'efficacia delle scelte architetturali adottate per carichi di lavoro moderati.

L'incremento del carico a 1000 connessioni simultanee ha evidenziato un comportamento del sistema ancora soddisfacente, seppur con una naturale degradazione delle performance. L'aumento della latenza media e la presenza di timeout rappresentano fenomeni attesi in condizioni di stress crescente, mentre il mantenimento di un throughput elevato dimostra la capacità del sistema di adattarsi efficacemente anche sotto pressione significativa.

Il test più impegnativo, condotto con 10.000 connessioni, ha posto il sistema di fronte a condizioni estreme di stress che hanno inevitabilmente comportato un incremento degli errori e della latenza. Tuttavia, la capacità di continuare a processare oltre 70.000 richieste al secondo anche in questo scenario critico evidenzia la solidità dell'infrastruttura realizzata e la sua capacità di degradare gradualmente piuttosto che collassare improvvisamente. In conclusione, i test condotti confermano che il sistema sviluppato possiede le caratteristiche necessarie per operare efficacemente negli scenari d'uso previsti, dimostrando inoltre una robustezza strutturale che lascia intravedere interessanti prospettive per futuri sviluppi e ottimizzazioni.