

Laboratorio Di Reti e Sistemi Distribuiti

Ilacqua Giovanni

6 luglio 2025

1 Introduzione al problema

Il progetto consiste nello sviluppo di un server HTTP concorrente, scritto in linguaggio C, che si attiene allo standard HTTP/1.1. Il server deve essere capace di gestire alcuni tipi di richiesta: **GET**, **POST**, **PUT** e **DELETE**. Il server deve anche essere capace di gestire i tipi di risposta principale: 200, 201, 204, 400.

Si cerca quindi di creare un server HTTP minimo capace di gestire le funzioni più base, in questo caso, l'invio di documenti HTML salvati in una cartella sul server, e la registrazione di coppie chiave-valore nel server. Il server deve essere capace di eseguire le proprie funzioni in maniera concorrente, ovvero deve poter gestire più connessioni alla volta contemporaneamente.

2 Stato dell'Arte

2.1 HTTP

HTTP (Hypertext Transfer Protocol) è un protocollo appartenente al livello Applicazione, ed è il protocollo che definisce il Web.

HTTP viene principalmente utilizzato per lo scambio di documenti ipertestuali scritti in HTML. Tutta via, le richieste e risposte definite da HTTP lo rendono versatile, è può essere usato anche per altri scopi.

HTTP è basato su un architettura client-server, dove il client è molto spesso un Browser Web. Il client invia richieste al server, specificando il metodo e la risorsa, vari header, e un eventuale corpo. Il server risponde specificando uno stato, vari header, e un eventuale corpo.

HTTP è stateless, non mantiene lo stato tra due richieste successive. Per tenere conto di una sessione, è necessario introdurre ulteriori meccanismi.

HTTP utilizza TCP come protocollo al livello trasporto, esso garantisce l'affidabilità delle comunicazioni, nonostante sia più lento rispetto alla sua controparte non affidabile (UDP).

Esistono due porte riservate per il protocollo HTTP:

- 80: Viene usata per il protocollo HTTP di default
- 443: Viene utilizzata per il protocollo HTTPS

HTTPS è la versione più sicura di HTTP, utilizza SSL/TLS per aggiungere sicurezza alle comunicazioni.

2.2 Messaggi HTTP

Un messaggio HTTP appartiene a una di due categorie principali: richieste e risposte

2.2.1 Richieste

Le richieste HTTP sono composte da 3 parti:

- **Request line:** Contiene il metodo richiesto, la risorsa richiesta, e la versione di HTTP
- **Header:** Contiene una serie di impostazioni e metadati come l'host, l'user-agent, etc. Ogni linea rappresenta un meta-dato
- **Body:** Contiene i dati specifici al metodo richiesto, per esempio, i dati da inviare al server per il POST o PUT. Viene separato dal header con una linea vuota.

Esempio:

```
1 GET / HTTP/1.1
2 Host: localhost:8090
3 User-Agent: Mozilla/5.0
4 Accept: text/html,application/xhtml+xml
```

2.2.2 Risposte

Analogamente alle richieste, le risposte HTTP sono composte da 3 parti:

- **Response line:** Contiene la versione HTTP, il codice di stato,
- **Header:** Contiene impostazioni e metadati, come il content-type e content-length.
- **Body:** Contiene la risorsa richiesta, ad esempio, un documento HTML, un'immagine o dei dati in formato JSON.

Esempio:

```
1 HTTP/1.1 200 OK
2 Content-Length: 662
3 Content-Type: text/html
4 Connection: close
5
6 <!DOCTYPE html>
7 ...
```

2.3 Metodi e Risposte HTTP

Ogni metodo in HTTP rappresenta un'operazione che il server deve eseguire per attenersi allo standard. In questo progetto, le operazioni di interesse sono: GET, POST, PUT, DELETE

- **GET:** Richiedi una risorsa dal server, ad esempio una pagina HTML, o dei dati da un API
- **POST:** Manda dei dati al server per creare una risorsa, ad esempio manda i dati di un form
- **PUT:** Manda dei dati al server per modificare una risorsa, o crearla se non esiste
- **DELETE:** Elimina dai dati dal server

Le risposte HTTP rappresentano lo stato del server dopo aver ricevuto la risposta. In questo progetto, le risposte di interesse sono: 200, 201, 204, 400.

- **200:** OK, la richiesta è stata processata correttamente.
- **201:** CREATED, la risorsa inviata è stata creata, e quindi aggiunta al server.
- **204:** NO CONTENT, la richiesta è stata processata correttamente, ma non c'è un body nella risposta
- **400:** BAD REQUEST, la richiesta inviata non è scritta correttamente, è malformata oppure ha dati incorretti.

2.4 Server HTTP

Un server HTTP è un software che si occupa di gestire le richieste HTTP ricevute da dei client (di solito, i Web Browser).

I server HTTP vengono usati per vari scopi:

- Hosting di siti web, permettere l'accesso alle pagine HTML di un sito web.
- Gestione di API attraverso servizi RESTful, ovvero permettere ai client di inviare richieste alle API attraverso HTTP.
- Hosting di applicazioni web, permettere l'accesso a delle applicazioni eseguibili tramite browser, inviando il codice necessario (e.g. javascript).

Lo scopo principale di un server HTTP è quindi quello di consegnare risorse attraverso il web.

3 Metodologia ed Implementazione

Il server è stato sviluppato su un ambiente Linux Ubuntu, utilizzando il linguaggio C e utilizzando Git come software di version control. Il programma è stato diviso in moduli, e viene compilato attraverso Make.

3.1 Architettura del programma

Il programma è suddiviso in quattro moduli: Main, response, request, keyvalue. Oltre ai moduli, il server contiene una cartella, **www**, all'interno della quale vengono inseriti i documenti HTML da hostare, e un makefile basilare per la compilazione del programma.

3.1.1 Moduli

Ogni modulo ha un rispettivo file header (.h) contenente la definizione dell'interfaccia del modulo (interfaccia delle funzioni, definizioni di strutture dati), e un rispettivo file di codice C (.c) contenente l'implementazione del codice (ad eccezione del main, che non ha un header).

- **main.c**: Gestisce la configurazione della connessione TCP, accetta le connessioni in entrata e crea un thread per ogni connessione.
- **keyvalue.h/keyvalue.c**: Gestisce le coppie chiave-valore (aggiunta, cancellazione), contiene un mutex lock per evitare race condition.
- **request.h/request.c**: Gestisce il parsing delle richieste in un apposita struttura dati.
- **response.h/response.c**: Gestisce la creazione delle risposte e l'esecuzione delle operazioni necessarie a soddisfare le richieste.

3.1.2 Makefile

Il makefile utilizzato è molto basilare, consiste semplicemente nel compilare il programma con gcc, assieme ad alcuni parametri, e una macro *clear* usata per cancellare il file eseguibile.

```
1 Makefile:
2 CC = gcc
3 CFLAGS = -g -Wall
4
5 TARGET = httpserver
6 SRCS = main.c request.c keyvalue.c response.c
7
8 $(TARGET):
9     $(CC) $(CFLAGS) -o $(TARGET) $(SRCS)
10
11
12 clean:
13     rm httpserver
```

3.2 Implementazione del codice

3.2.1 Main

Il programma inizia con la configurazione della connessione, viene creata una socket sulla porta *8090*, e il server si mette in ascolto per connessioni. Nel momento in cui si riceve una connessione, viene accettata, e un thread viene creato per gestirla.

La funzione *handle_client* viene usata per la gestione del client, la funzione gestisce una richiesta, per poi chiudere la connessione. Vengono usate 2 funzioni per gestire la richiesta, definite negli altri moduli: *parse_http_request* e *handle_request*

```
1 void* handle_client(void *arg) {
2
3     //gestisci la richiesta di un client
4     int client_fd = *(int *)arg;
5     free(arg);
6
7     //leggi il messaggio del client ed inseriscilo nel
8     buffer
9     char buffer[BUFFER_SIZE];
10
11     ssize_t total = 0, n;
12     while ((n = read(client_fd, buffer + total,
13         BUFFER_SIZE - 1 - total)) > 0) {
14         total += n;
15         if (strstr(buffer, "\r\n\r\n")) break;
16     }
17     buffer[total] = '\0';
18
19     printf("---_RAW_-----_n%s\n", buffer);
20
21     //parsing della richiesta
22     HttpRequest nreq;
```

```

22     parse_http_request(buffer,&nreq);
23     printf("---_REQUEST-----_\n%s\n%s\n%s\n%s\n%
        s\n",nreq.method,nreq.path,nreq.version,nreq.
        headers,nreq.body);

24
25     //gestisci la richiesta e scrivi la risposta
26     char* resp = handle_request(&nreq);
27     printf("---_RESPONSE_\n%s\n",resp);
28
29     //invia la risposta al client
30     if (send(client_fd,resp,strlen(resp),0) == -1) {
31         perror("Response_send_error");
32         close(client_fd);
33         exit(EXIT_FAILURE);
34     }
35
36     //chiudi la connessione
37     close(client_fd);
38     pthread_exit(NULL);
39     printf("Response sent, connection closed\n");
40
41     return NULL;
42 }

```

3.2.2 Request

Per gestire il parsing delle richieste, viene usata una struttura dati che contiene le varie parti della richiesta:

```

1 typedef struct {
2     char method[8];
3     char path[256];
4     char version[16];
5     char headers[1024];
6     char body[2048];
7 } HttpRequest;

```

Il compito della funzione *parse_http_request* è di inserire le varie parti della richiesta nelle rispettive stringhe dello struct. La funzione trova l'inizio del header e del body, e li usa per calcolare le dimensioni del header, per poi determinare la dimensione del body attraverso il parametro *content-length*.

```

1 void parse_http_request(const char *raw_request, HttpRequest
    *req) {
2
3     //inizializza la struttura dati
4     memset(req, 0, sizeof(HttpRequest));
5
6     //inserisci la request line nei rispettivi campi
        della struttura
7     int parsed = sscanf(raw_request, "%7s_%255s_%15s",req
        ->method, req->path, req->version);
8

```

```

9      // Se non abbiamo tutte e tre le parti, inserisci
      delle linee di default
10     if (parsed != 3) {
11         strcpy(req->method, "BAD");
12         strcpy(req->path, "/");
13         strcpy(req->version, "HTTP/1.0");
14     }
15
16     //trova l'inizio dell'header
17     const char *header_start = strstr(raw_request, "\r\n"
      );
18     if (!header_start) return;
19     header_start += 2;
20
21     //trova l'inizio del body, se presente.
22     const char *body_start = strstr(raw_request, "\r\n\r\n");
23
24
25     if (body_start && (body_start > header_start)) {
26
27         //se il body e presente, inserisci l'header e
          il body
28
29         //determina la dimensione dell'header
30         int header_len = (int)(body_start -
          header_start);
31         if (header_len > 1023) header_len = 1023;
32
33         //inserisci l'header
34         strncpy(req->headers, header_start,
          header_len);
35         req->headers[header_len] = '\0';
36
37         //determina la dimensione del body, in base
          al tag "Content-Length" dell'header.
38         int content_length = 0;
39         const char *cl = strstr(req->headers, "
          Content-Length:");
40         if (cl) {
41
42             cl += strlen("Content-Length:");
43             while (*cl == ' ') cl++;
44             content_length = atoi(cl);
45
46         }
47
48         if (content_length > 0 && content_length <
          sizeof(req->body)) {
49
50             strncpy(req->body, body_start + 4,
          content_length);
51             req->body[content_length] = '\0';

```

```

52         } else {
53             req->body[0] = '\0';
54         }
55     } else {
56
57         //se il body non e presente, inserisci solo 1
58         //header.
59         strncpy(req->headers, header_start, 1023);
60         req->headers[1023] = '\0';
61         req->body[0] = '\0';
62     }
63 }
64
65 }

```

3.2.3 Keyvalue

Lo struct *KeyValue* viene usato per rappresentare le coppie chiave-valore, che vengono poi inserite in un array. Per assicurarsi che le funzioni di questo modulo non creino race-conditions, viene usato un mutex lock.

```

1 struct KeyValue {
2     char key[64];
3     char value[256];
4 };
5
6 extern struct KeyValue store[MAX_KV_PAIRS];
7 extern int store_count;
8
9 extern pthread_mutex_t store_mutex;

```

Sono definite 3 funzioni che operano sulle coppie chiave-valore:

- *add_key_value*: aggiunge una coppia
- *is_kv_stored*: determina se una coppia è presente, ritornando 0 o 1.
- *delete_kv_pair*: trova una coppia, e muove le coppie che vengono dopo verso sinistra per cancellarla.

La funzione *append_kv_to_html* serve ad aggiungere la lista delle coppie a un documento HTML, in maniera tale da poterla visualizzare. Costruisce la lista di coppie in formato HTML nella stringa *kv_html*, per poi unirla al documento HTML.

```

1 char *append_kv_to_html(const char *html) {
2
3     //aggiungi a un documento html una lista che mostra le
4     //coppie chiave-valore
5
6     char kv_html[2048];
7     strcpy(kv_html, "<h2>Saved_Pairs:</h2><ul>");
8
9     pthread_mutex_lock(&store_mutex);

```



```

10 //leggi la lista e aggiungi ogni coppia
11     for (int i = 0; i < store_count; i++) {
12
13         strcat(kv_html, "<li>");
14         strcat(kv_html, store[i].key);
15         strcat(kv_html, ": ");
16         strcat(kv_html, store[i].value);
17         strcat(kv_html, "</li>");
18
19     }
20
21     pthread_mutex_unlock(&store_mutex);
22
23     strcat(kv_html, "</ul>");
24
25 //genera il documento finalizzato
26     char *final = malloc(strlen(html) + strlen(kv_html) +
27         1);
28     strcpy(final, html);
29     strcat(final, kv_html);
30
31     return final;

```

3.2.4 Response

La funzione principale del modulo response è *handle_request*, che si occupa di soddisfare una richiesta e di scrivere una risposta.

```

1 char *handle_request(HttpRequest *req) {
2
3     char *body = NULL;
4     static char response[MAX_RESPONSE_SIZE];
5
6     //determina quale metodo e stato richiesto, e chiama
7     //la corrispondente funzione che lo gestisce
8     //poi scrivi una risposta
9     if (strcmp(req->method, "GET") == 0) {
10
11         body = handle_get(req);
12         write_http_response(response, 200, "text/html",
13             ,body);
14
15     } else if (strcmp(req->method, "POST") == 0) {
16
17         body = handle_post(req);
18         write_http_response(response, 201, "text/html",
19             ,body);
20
21     } else if (strcmp(req->method, "PUT") == 0) {
22
23         int flg;
24
25         body = handle_put(req, &flg);

```

```

23         write_http_response(response, flg ? 201 :
24                               204, "text/html", body);
25
26     } else if (strcmp(req->method, "DELETE") == 0) {
27
28         int flg;
29
30         body = handle_delete(req, &flg);
31         write_http_response(response, flg ? 201 :
32                               204, "text/html", body);
33
34     } else {
35
36         body = strdup("<html><body><h1>400_Bad_
37                       Request</h1></body></html>");
38         write_http_response(response, 400, "text/html"
39                               , body);
40
41     }
42
43     if (body != NULL) {
44         free(body);
45     }
46
47     return response;
48 }

```

La funzione *handle_request* gestisce le richieste chiamando, per ogni tipo di richiesta, una corrispondente funzione che la gestisce, e scrive il body della risposta.

- **GET:** la funzione *handle_get* trova il documento HTML corrispondente al path della richiesta, e lo ritorna. Il codice inviato nella risposta sarà 200 (OK). Ogni metodo implementa questo comportamento.
- **POST:** la funzione *handle_post* aggiunge una coppia chiave-valore, e ritorna il documento HTML richiesto con l'aggiunta delle coppie. Il codice inviato nella risposta sarà 201 (CREATED).
- **PUT:** La funzione *handle_put* aggiunge una coppia chiave-valore, ma solo se non è già presente, e ritorna il documento HTML richiesto con l'aggiunta delle coppie. Se non ha aggiunto una coppia, non ritorna un body. Il codice inviato nella risposta sarà 201 (CREATED) se ha creato una coppia, altrimenti 204 (NO CONTENT).
- **DELETE:** La funzione *handle_delete* cancella una coppia chiave-valore presente, e ritorna il documento. Se non cancella nulla, non ritorna un body. Il codice inviato nella risposta sarà 201 (CREATED) se ha cancellato una coppia, oppure 204 (NO CONTENT).
- **Altri:** Per le altre richieste, la funzione *handle_request* scrive un semplice body HTML per indicare il codice 400, e invia il codice 400 (BAD REQUEST) stesso nella risposta.

Sono presenti anche altre 3 funzioni usate nella gestione delle richieste:

- *get_html_file*: legge un file HTML e ritorna i suoi contenuti
- *parse_form_data*: prende in input il body di una richiesta contenente coppie chiave-valore, e in base al parametro *op*, esegue un'operazione sulle coppie che corrisponde alla richiesta che ha chiamato la funzione. Ritorna l'esito dell'operazione.
- *write_http_response*: Scrive una risposta HTTP.

```

1 void write_http_response(char *response_buffer, int
   status_code, const char *content_type, const char *body) {
2
3     //scrivi il testo corrispondente all'operazione
4     const char *status_text;
5
6     switch (status_code) {
7         case 200: status_text = "OK"; break;
8         case 201: status_text = "Created"; break;
9         case 204: status_text = "No_Content"; break;
10        case 400: status_text = "Bad_Request"; break;
11        default: status_text = "Internal_Server_Error
           "; break;
12    }
13
14    //determina la dimensione del body, se presente
15    int content_length = body ? strlen(body) : 0;
16
17    //scrivi la richiesta
18    sprintf(
19        response_buffer,
20        "HTTP/1.1_%d_%s\r\n"
21        "Content-Length:_%d\r\n"
22        "Content-Type:_%s\r\n"
23        "Connection:_%close\r\n"
24        "\r\n"
25        "%s",
26        status_code, status_text,
27        content_length, content_type, body
28        ? body : ""
29    );
30 }

```

4 Risultati

Per evidenziare i risultati ottenuti e mostrare il funzionamento, verrà usato il browser *Firefox* e il comando *Curl* per inviare richieste al server.

4.1 GET

Per provare la richiesta GET usando firefox, basta connetersi al server via URL, il browser invierà una richiesta GET.



```
locke8@locke8-VirtualBox: ~/Desktop/final-542503
--- REQUEST-----
GET
/test3.html
HTTP/1.1
Host: localhost:8090
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br, zstd
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
DNT: 1
Sec-GPC: 1
Priority: u=0, i

--- RESPONSE -----
HTTP/1.1 200 OK
Content-Length: 2539
Content-Type: text/html
Connection: close

<!DOCTYPE html>

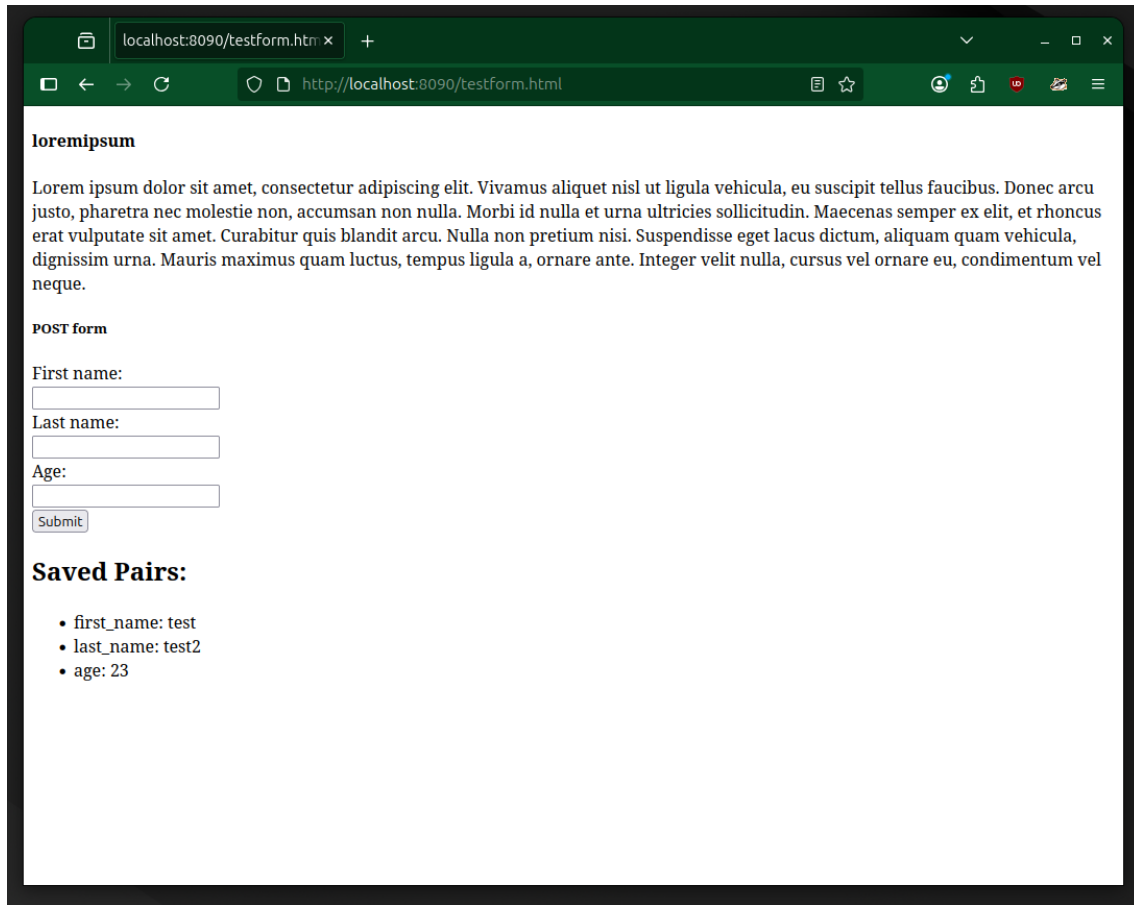
<html>

  <head>
    <style>
```

Il serve invia correttamente il documento HTML, e ha inviato una risposta con codice 200 (OK).

4.2 POST

Per provare il POST, verrà usata una pagina HTML con un form impostato a *method=POST*.



localhost:8090/testform.htm x +

http://localhost:8090/testform.html

loremipsum

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus aliquet nisl ut ligula vehicula, eu suscipit tellus faucibus. Donec arcu justo, pharetra nec molestie non, accumsan non nulla. Morbi id nulla et urna ultricies sollicitudin. Maecenas semper ex elit, et rhoncus erat vulputate sit amet. Curabitur quis blandit arcu. Nulla non pretium nisi. Suspendisse eget lacus dictum, aliquam quam vehicula, dignissim urna. Mauris maximus quam luctus, tempus ligula a, ornare ante. Integer velit nulla, cursus vel ornare eu, condimentum vel neque.

POST form

First name:

Last name:

Age:

Saved Pairs:

- first_name: test
- last_name: test2
- age: 23

```
locke8@locke8-VirtualBox: ~/Desktop/final-542503
Sec-Fetch-User: ?1
DNT: 1
Sec-GPC: 1
Priority: u=0, i

first_name=test&last_name=test2&age=23
--- REQUEST-----
POST
/testform.html
HTTP/1.1
Host: localhost:8090
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br, zstd
Content-Type: application/x-www-form-urlencoded
Content-Length: 38
Origin: http://localhost:8090
Connection: keep-alive
Referer: http://localhost:8090/testform.html
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
DNT: 1
Sec-GPC: 1
Priority: u=0, i
first_name=test&last_name=test2&age=23
--- RESPONSE -----
HTTP/1.1 201 Created
Content-Length: 1172
Content-Type: text/html
Connection: close

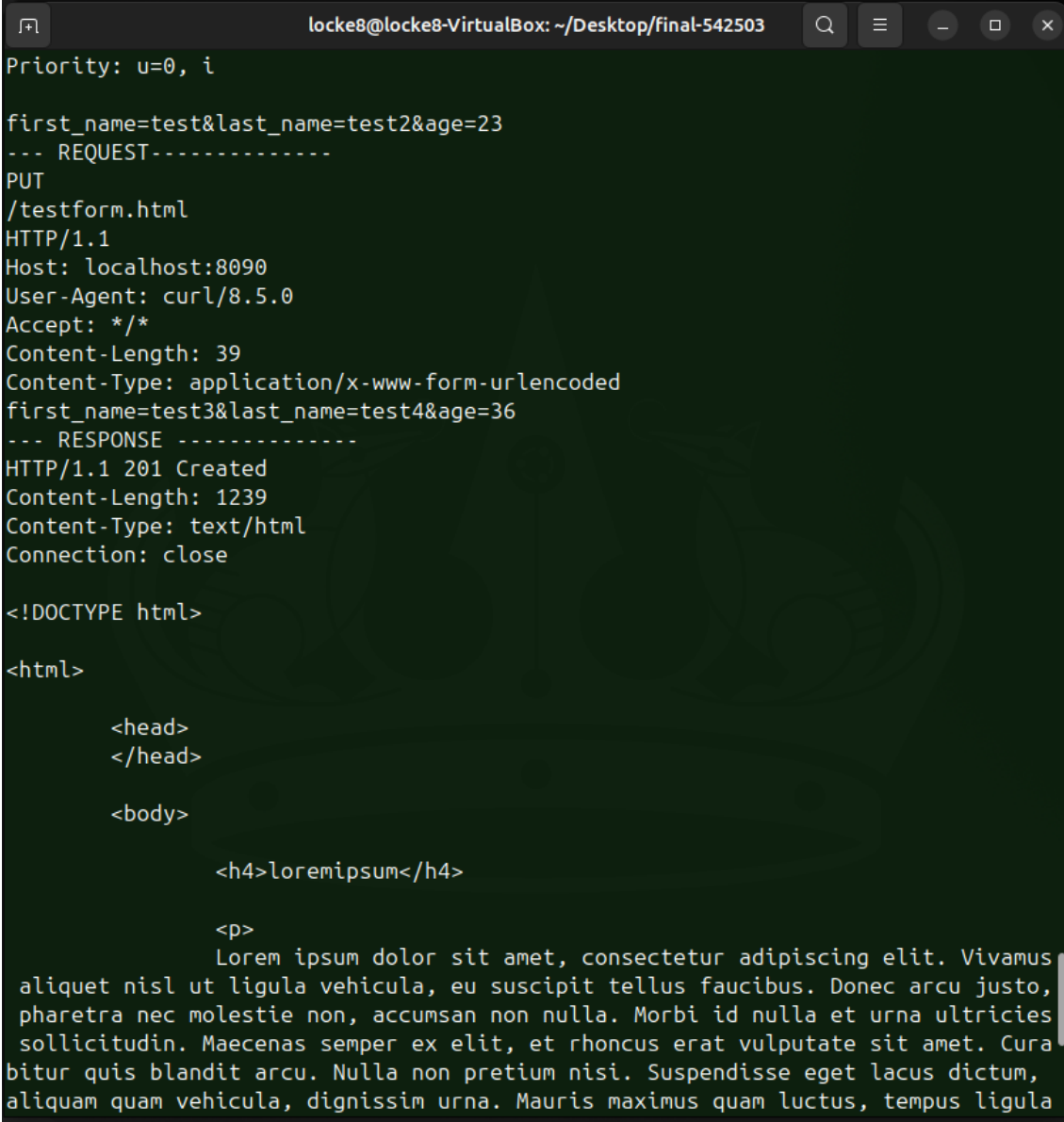
<!DOCTYPE html>
```

il server aggiunge la coppia chiavi valore, come richiesto, e ha inviato una risposta con codice 201 (CREATED).

4.3 PUT

Per provare il PUT, verrà usato il comando curl:

```
1 curl -X PUT http://localhost:8090/testform.html -d "
    first_name=test3&last_name=test4"
```



```
locke8@locke8-VirtualBox: ~/Desktop/final-542503
Priority: u=0, i

first_name=test&last_name=test2&age=23
--- REQUEST-----
PUT
/testform.html
HTTP/1.1
Host: localhost:8090
User-Agent: curl/8.5.0
Accept: */*
Content-Length: 39
Content-Type: application/x-www-form-urlencoded
first_name=test3&last_name=test4&age=36
--- RESPONSE -----
HTTP/1.1 201 Created
Content-Length: 1239
Content-Type: text/html
Connection: close

<!DOCTYPE html>

<html>

    <head>
    </head>

    <body>

        <h4>loremipsum</h4>

        <p>
            Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus
            aliquet nisl ut ligula vehicula, eu suscipit tellus faucibus. Donec arcu justo,
            pharetra nec molestie non, accumsan non nulla. Morbi id nulla et urna ultricies
            sollicitudin. Maecenas semper ex elit, et rhoncus erat vulputate sit amet. Cura
            bitur quis blandit arcu. Nulla non pretium nisi. Suspendisse eget lacus dictum,
            aliquam quam vehicula, dignissim urna. Mauris maximus quam luctus, tempus ligula

<h2>Saved Pairs:</h2><ul><li>first_name: test</li><li>last_name: test2</li><li>a
ge: 23</li><li>first_name: test3</li><li>last_name: test4</li><li>age: 36</li></
ul>
```

Il server aggiunge la coppia chiave-valore, e ha inviato una risposta con codice 201 (CREATED).

Inviando il comando una seconda volta:


```
--- REQUEST-----  
PUT  
/testform.html  
HTTP/1.1  
Host: localhost:8090  
User-Agent: curl/8.5.0  
Accept: */*  
Content-Length: 32  
Content-Type: application/x-www-form-urlencoded  
first_name=test3&last_name=test4  
--- RESPONSE -----  
HTTP/1.1 204 No Content  
Content-Length: 0  
Content-Type: text/html  
Connection: close
```

Il server non ha ri-aggiunto la coppia, e ha inviato una risposta con codice 204 (NO CONTENT).

4.4 DELETE

Per provare il DELETE, verrà prima usato il comando:

```
1 curl -X PUT http://localhost:8090/testform.html -d "  
    first_name=test3&last_name=test4"
```

per poi usare:

```
1 curl -X DELETE http://localhost:8090/testform.html -d "  
    first_name=test3&last_name=test4"
```

```
locke8@locke8-VirtualBox: ~/Desktop/final-542503
first_name=test3&last_name=test4&age=36
--- REQUEST-----
DELETE
/testform.html
HTTP/1.1
Host: localhost:8090
User-Agent: curl/8.5.0
Accept: */*
Content-Length: 39
Content-Type: application/x-www-form-urlencoded
first_name=test3&last_name=test4&age=36
--- RESPONSE -----
HTTP/1.1 201 Created
Content-Length: 1106
Content-Type: text/html
Connection: close

<!DOCTYPE html>

<html>

    <head>
    </head>

    <body>

        <h4>loremipsum</h4>

        <p>
            Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus
            aliquet nisl ut ligula vehicula, eu suscipit tellus faucibus. Donec arcu justo,
            pharetra nec molestie non, accumsan non nulla. Morbi id nulla et urna ultricies
            sollicitudin. Maecenas semper ex elit, et rhoncus erat vulputate sit amet. Cura
            bitur quis blandit arcu. Nulla non pretium nisi. Suspendisse eget lacus dictum,
            aliquam quam vehicula, dignissim urna. Mauris maximus quam luctus, tempus ligula
            a, ornare ante. Integer velit nulla, cursus vel ornare eu, condimentum vel nequ
```

```
</html>

<h2>Saved Pairs:</h2><ul></ul>
```

Il server ha cancellato la coppia chiave-valore, e ha inviato una risposta con codice 201 (CREATED).

Inviando il comando una seconda volta:

```
--- REQUEST-----  
DELETE  
/testform.html  
HTTP/1.1  
Host: localhost:8090  
User-Agent: curl/8.5.0  
Accept: */*  
Content-Length: 39  
Content-Type: application/x-www-form-urlencoded  
first_name=test3&last_name=test4&age=36  
--- RESPONSE -----  
HTTP/1.1 204 No Content  
Content-Length: 0  
Content-Type: text/html  
Connection: close
```

Il server non ha eseguito nessun cambiamento, e ha inviato una risposta con codice 204 (NO CONTENT).

4.5 BAD REQUEST

usando il comando curl:

```
1 curl -X OPTIONS http://localhost:8090/testform.html
```

il server manda una risposta con codice 400, dato che non sa' gestire questo tipo di richiesta

```
--- REQUEST-----
OPTIONS
/testform.html
HTTP/1.1
Host: localhost:8090
User-Agent: curl/8.5.0
Accept: */*

--- RESPONSE -----
HTTP/1.1 400 Bad Request
Content-Length: 50
Content-Type: text/html
Connection: close

<html><body><h1>400 Bad Request</h1></body></html>
```

4.6 Concorrenza

Per verificare la capacità del server di gestire i client in maniera concorrente, verranno usate 3 istanze del comando curl inviate contemporaneamente:

```
1 curl -X POST http://localhost:8090/testform.html -d "
    first_name=test1&last_name=test2&age=22" &
2 curl -X PUT http://localhost:8090/testform.html -d "
    first_name=test1&last_name=test2&age=22" &
3 curl -X DELETE http://localhost:8090/testform.html -d "
    first_name=test1&last_name=test2&age=22" &
4 wait
```

La prima richiesta che viene gestita è il DELETE, essendo che la coppia non è ancora stata creata, il server invia una risposta 204.

```
--- REQUEST-----  
DELETE  
/testform.html  
HTTP/1.1  
Host: localhost:8090  
User-Agent: curl/8.5.0  
Accept: */*  
Content-Length: 39  
Content-Type: application/x-www-form-urlencoded  
first_name=test1&last_name=test2&age=22  
--- RESPONSE -----  
HTTP/1.1 204 No Content  
Content-Length: 0  
Content-Type: text/html  
Connection: close
```

La seconda richiesta ad essere gestita è il POST, il server crea due copie, e invia una risposta 201.

```
--- REQUEST-----  
POST  
/testform.html  
HTTP/1.1  
Host: localhost:8090  
User-Agent: curl/8.5.0  
Accept: */*  
Content-Length: 39  
Content-Type: application/x-www-form-urlencoded  
first_name=test1&last_name=test2&age=22  
--- RESPONSE -----  
HTTP/1.1 201 Created  
Content-Length: 1173  
Content-Type: text/html  
Connection: close  
  
<!DOCTYPE html>
```

```
<h2>Saved Pairs:</h2><ul><li>first_name: test1</li><li>last_name: test2</li><li>age: 22</li></ul>
```

La terza richiesta ad essere gestita è il PUT, essendo che le coppie sono già state create, il server invia una risposta 204.

```
--- REQUEST-----  
PUT  
/testform.html  
HTTP/1.1  
Host: localhost:8090  
User-Agent: curl/8.5.0  
Accept: */*  
Content-Length: 39  
Content-Type: application/x-www-form-urlencoded  
first_name=test1&last_name=test2&age=22  
--- RESPONSE -----  
HTTP/1.1 204 No Content  
Content-Length: 0  
Content-Type: text/html  
Connection: close
```

5 Conclusione e Sviluppi Futuri

Il server sviluppato è capace di gestire le richieste principali del protocollo HTTP, e di fornire risposte soddisfacenti. Grazie alle capacità di concorrenza, il server riesce a gestire molteplici clienti in maniera efficiente.

Il server è in oltre una solida base di sviluppo, che potrebbe essere esteso con relativa facilità, grazie alla struttura modulare del codice.

Alcune possibili estensioni del server includono:

- Supportare ulteriori metodi HTTP (HEAD, OPTIONS)
- Supportare connessioni con **keep-alive** (meccanismo per mantenere la connessione col client aperta anche dopo aver inviato la risposta)
- Adottare la versione sicura di HTTP, ovvero HTTPS
- Mandare le coppie chiave-valore a un CGI, per supportare l'esecuzione di programmi esterni, e quindi permettere la creazione di contenuti dinamici