Server HTTP

Davide Frodà

Giugno 2025

Indice

Introduzione

Per questa presentazione è stato realizzato un progetto che consiste nella realizzazione di un Server **multiclient** e **concorrente**, in linguaggio C, in grado di:

1. Comprendere
2. Portare a termine
3. Rispondere correttamente

alle **richieste HTTP** inviategli da un client generico.

HTTP

HTTP (*HyperText Transfer Protocol*) è il protocollo che gestisce la comunicazione sul web a livello applicazione. Possiede regole molto rigide, descritte negli RFC.
In particolare questa implementazione di un **Server HTTP** prende come riferimento l'**RFC 2616** che decrive il protocollo HTTP alla versione 1.1.

Richieste HTTP

Ogni richiesta HTTP ha una struttura ben definita.

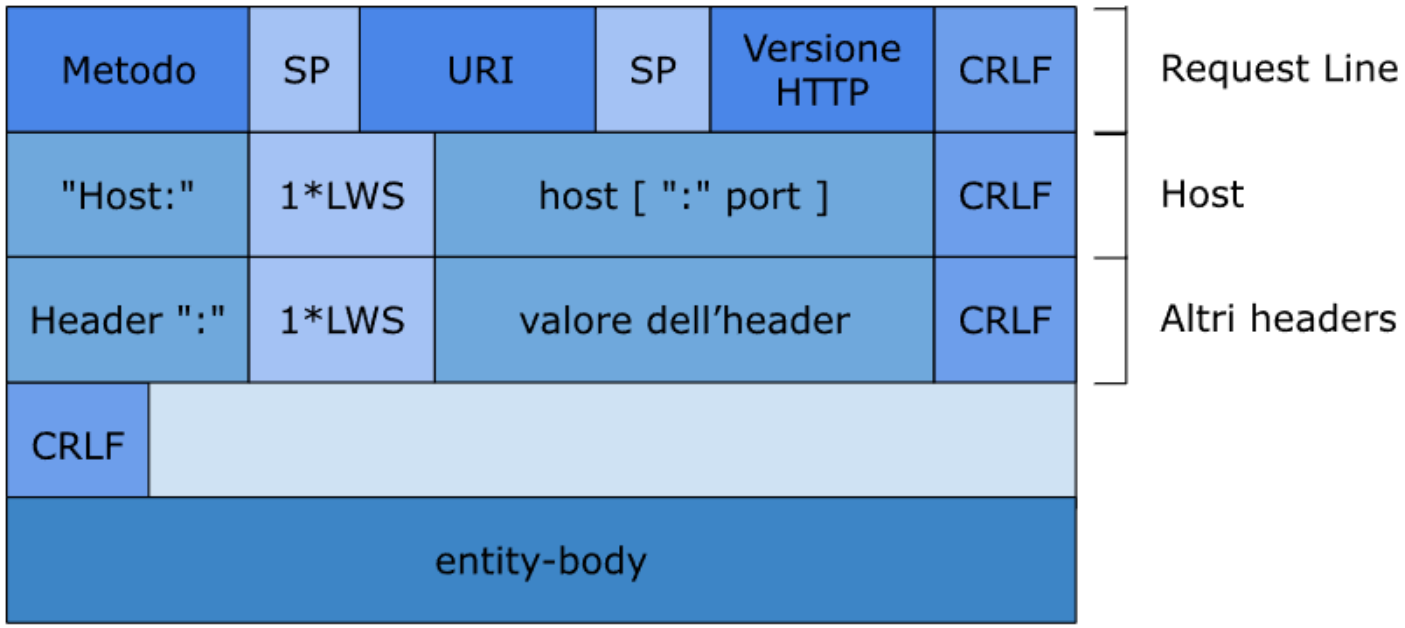


Figure: Struttura di una richiesta HTTP

Risposte HTTP

Le risposte HTTP hanno una struttura simile alle richieste.

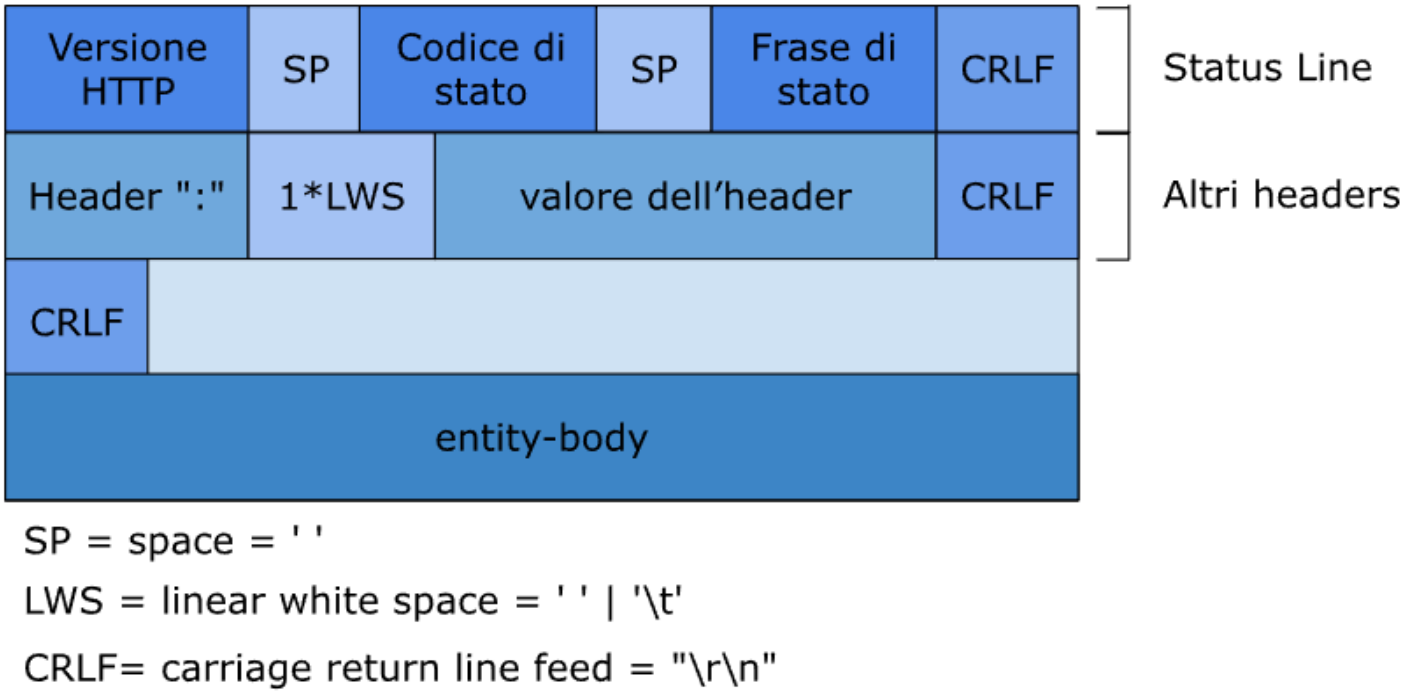


Figure: Struttura di una risposta HTTP

Definizione del Problema

Per la realizzazione di un **Server HTTP concorrente** è necessario:

- Realizzare un **server TCP** capace di gestire molti client contemporaneamente;
- Implementare un sistema di parsing che estragga le informazioni importanti dalla **richiesta HTTP**;
- Implementare un sistema di **sincronizzazione** per gestire gli accessi simultanei ai file.
- Realizzare un sistema capillare di **error detection** che notifichi ai client gli errori riscontrati.
- Realizzare un sistema per la generazione dei **messaggi di risposta**.

Possibili Implementazioni

Per la gestione concorrente di più client sullo stesso **Server** vi sono principalmente 3 possibili implementazioni:

- `select`: chiamata di sistema utilizzata per monitorare più file descriptor;
- `epoll`: API Linux con un meccanismo basato su eventi;
- Gestione `client-thread`: assegnare un thread indipendente ad ogni client.

Metodologia Adottata

Per la gestione concorrente di più client sullo stesso **Server** è stato utilizzato il sistema di **I/O multiplexing** `epoll`.

I client accettati vengono posti nella lista di client monitorati da epoll su eventi di input (`EPOLLIN`) in modalità edge-triggered (`EPOLLET`).

Quando si verifica un evento, ovvero la ricezione di uno o più messaggi dallo stesso **socket**, l'evento viene gestito in un thread separato creato grazie ad una funzione `thread_manager()`.

Struttura del programma

Il programma è stato realizzato seguendo il paradigma della modularità, è composto dai seguenti moduli e dotato di un makefile per la compilazione:

- server
- socket_functions
- epoll_functions
- thread_functions
- parsing_functions
- http_methods
- file_functions
- sending_functions

Operazioni Implementate

Il server HTTP realizzato riesce a gestire 4 diversi metodi HTTP:

- GET
- POST
- PUT
- DELETE

GET

Il metodo GET è utilizzato per richiedere al server una informazione identificata dall'URI della richiesta.

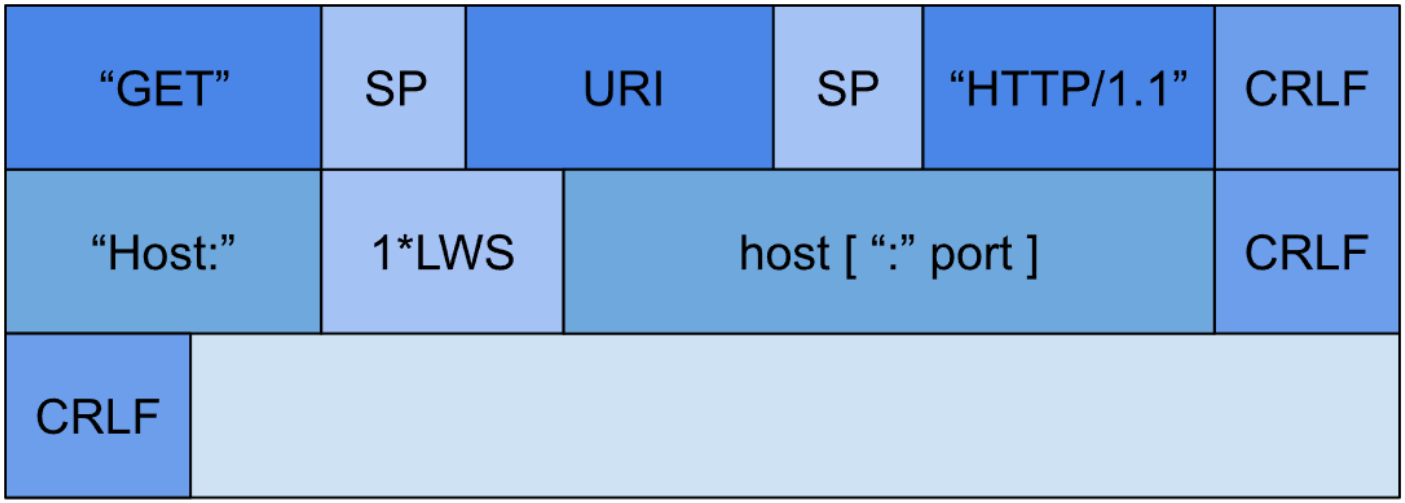


Figure: Richiesta con metodo GET

GET

Quando una richiesta GET viene portata a termine con successo, il client potrà ricevere un solo tipo di risposta:

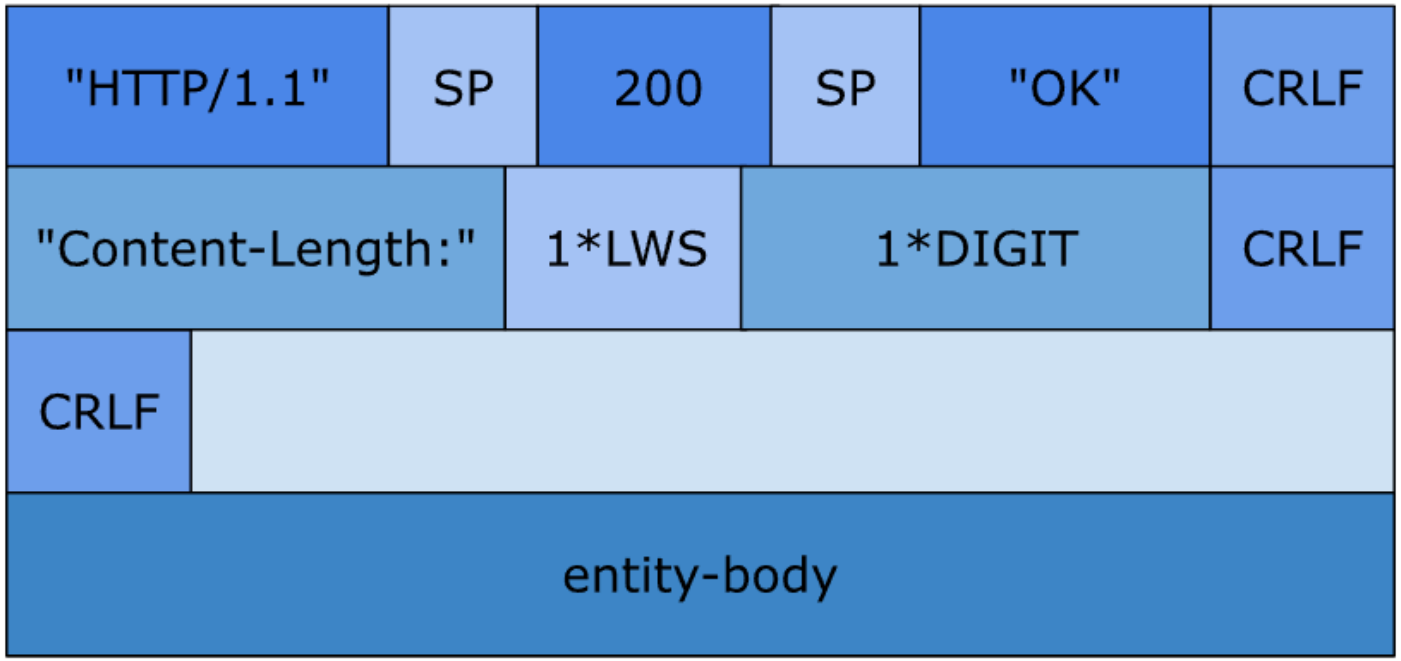


Figure: Risposta con codice di stato 200

GET

Pseudocodice dell'implementazione di GET:

```
function get(path, request, response):
    response.status_code ← 200
    error ← read_with_lock(path, request, response)
    return error
end function
```

Read with Lock

Pseudocodice della funzione read_with_lock():

```
function read_with_lock(path, request, response):
    file ← open_file(path, request.method)
    if not exists(file) then
        response.status_code ← 404
        return true
    end if
    lock_on_read(file)
    response.content ← read_file(file)
    response.content_length ← len(response.content)
    unlock(file)
    return false
end function
```

POST

Il metodo POST viene utilizzato per richiedere al server di accettare l'entità inclusa nella richiesta come nuova entità, subordinata alla risorsa identificata dall'URI della richiesta.

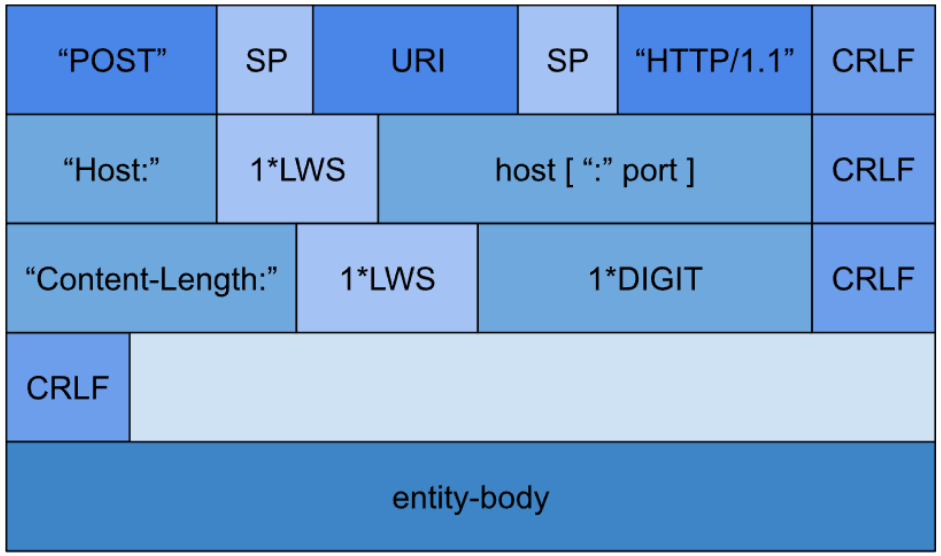


Figure: Richiesta con metodo POST

POST

Quando una richiesta POST viene portata a termine con successo, il client potrà ricevere due differenti risposte:

- `200 OK`: Quando l'URI fa riferimento a una risorsa esistente.
- `201 Created`: Quando l'URI fa riferimento a una risorsa non esistente.

201

La risposta con codice di stato 201 è utilizzata esclusivamente quando viene creata una nuova risorsa all'interno del server.

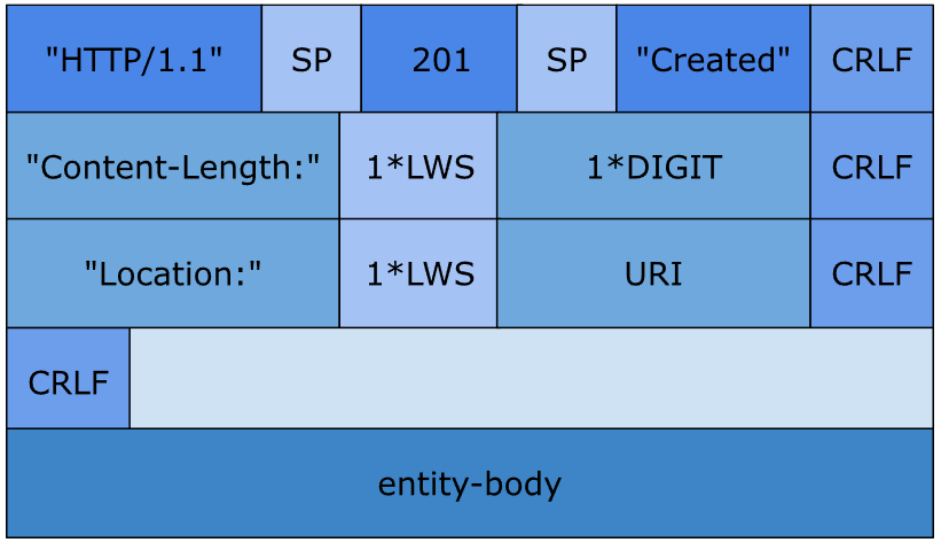


Figure: Risposta con codice di stato 201

POST

Pseudocodice dell'implementazione di POST:

```
function post(path, request, response):
    if is_index(path) then
        response.status_code ← 405
        response.allow ← "GET"
        return true
    end if
    if exists(path) then
        response.status_code ← 200
    else
        response.location ← path
        response.status_code ← 201
    end if
    error ← write_read_with_lock(path, request, response)
    return error
end function
```

PUT

Il metodo PUT richiede che l'entità racchiusa nella richiesta venga memorizzata nell'URI fornito dalla richiesta stessa.

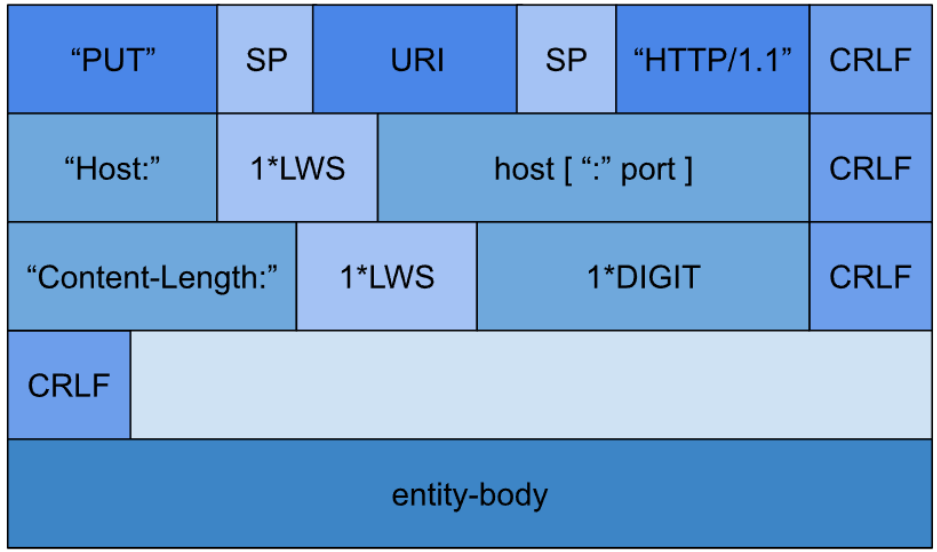


Figure: Richiesta con metodo PUT

PUT

Quando una richiesta PUT viene portata a termine con successo, il client potrà ricevere due differenti risposte:

- `204 No Content`: Quando l'URI fa riferimento a una risorsa esistente.
- `201 Created`: Quando l'URI fa riferimento a una risorsa non esistente.

204

La risposta con codice di stato 204 è utilizzata esclusivamente quando un'operazione viene portata a termine con successo ma non viene restituito del contenuto al client.

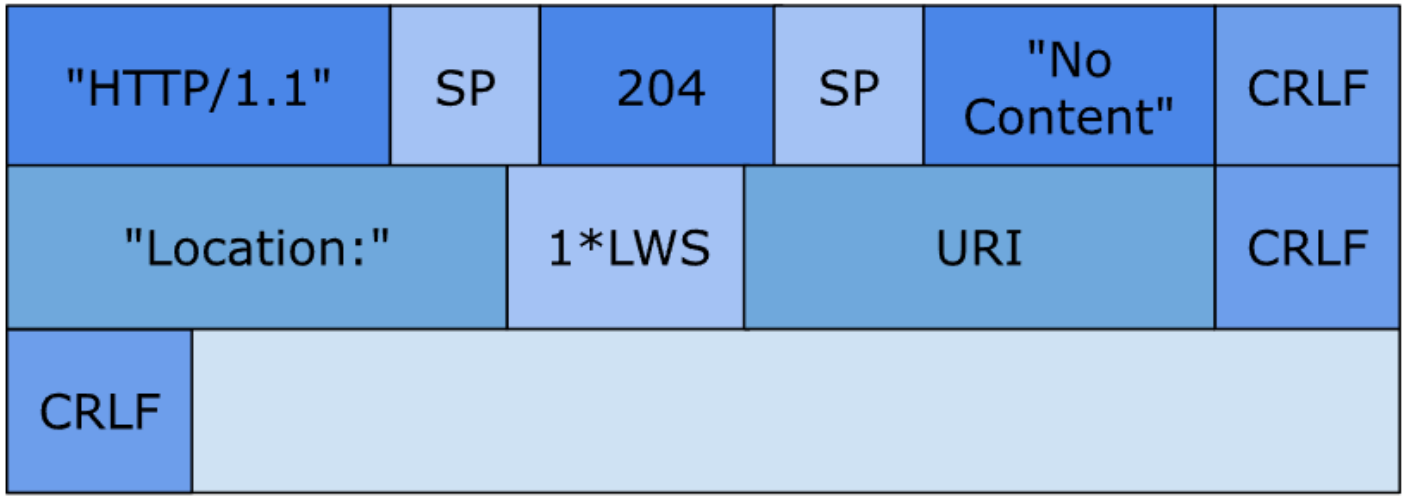


Figure: Risposta con codice di stato 204

PUT

Pseudocodice dell'implementazione di PUT:

```
function put(path, request, response):
    if is_index(path) then
        response.status_code ← 405
        response.allow ← "GET"
        return true
    end if
        response.location ← path
    if exists(path) then
        response.status_code ← 204
    else
        response.status_code ← 201
    end if
    error ← write_with_lock(path, request, response)
    return error
end function
```

Write with Lock

Pseudocodice della funzione write_with_lock():

```
function write_with_lock(path, request, response):
    file ← open_file(path, request.method)
    if not exists(file) then
        response.status_code ← 404
        return true
    end if
    lock_on_write(file)
    write_in_file(file, request.content)
    unlock(file)
    return false
end function
```

DELETE

Il metodo DELETE richiede che il server elimini la risorsa indicata nell'URI fornito dalla richiesta.

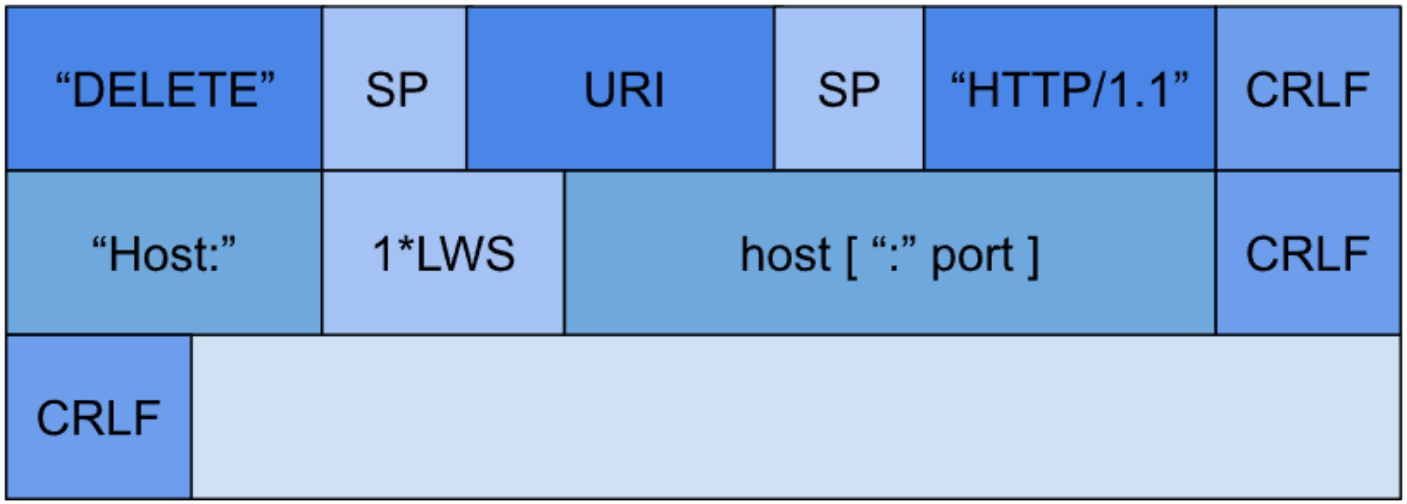


Figure: Richiesta con metodo DELETE

DELETE

Quando una richiesta DELETE viene accettata, il client riceve una risposta con codice di stato 202 (Accepted):

Figure: Risposta con codice di stato 202

DELETE

Pseudocodice dell'implementazione di DELETE:

```
function delete(path, request, response):
    if is_index(path) then
        response.status_code ← 405
        response.allow ← "GET"
        return true
    end if
    response.status_code ← 202
    error ← unlink_with_lock(path, request, response)
    return error
end function
```

Unlink with Lock

Pseudocodice della funzione unlink_with_lock():

```
function unlink_with_lock(path, request, response):
    file ← open_file(path, request.method)
    if not exists(file) then
        response.status_code ← 404
        return true
    end if
    lock_on_write(file)
    if exists(path) then
        delete(path)
    end if
    unlock(file)
    return false
end function
```

Risposte HTTP

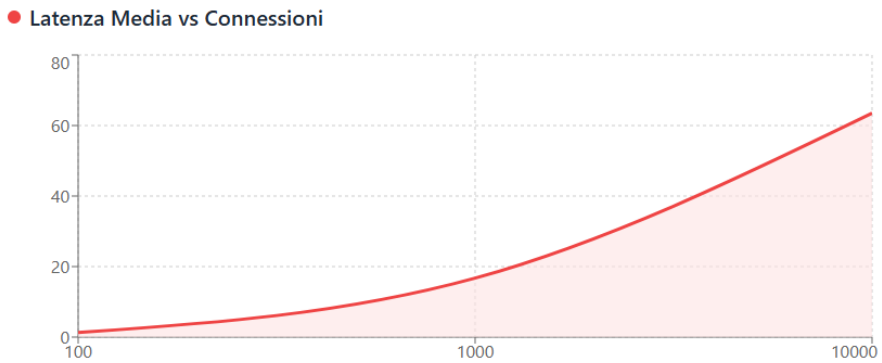
Oltre alle risposte già illustrate nelle precedenti slide all'interno del programma sono state implementate anche le risposte:

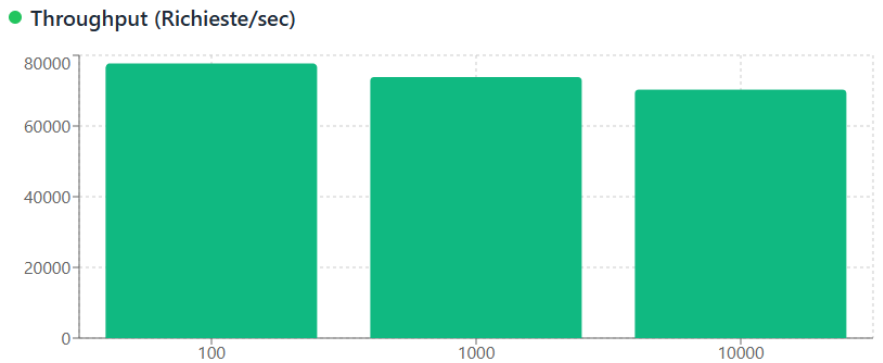
- `400 Bad Request`: La richiesta non è correttamente formulata secondo l'RFC 2616.
- `401 Unauthorize`: La richiesta sta cercando di accedere a delle risorse protette senza la dovuta autorizzazione.
- `404 Not Found`: Il server non ha trovato alcun elemento corrispondente all'URI della richiesta.
- `405 Method Not Allowed`: La richiesta richiede di eseguire un metodo su un elemento che non consente quel metodo.
- `414 Request-URI Too Large`: L'URI presente nella richiesta è troppo lungo per essere elaborato.
- `500 Internal Server Error`: Si è verificato un errore all'interno del server e l'esecuzione della richiesta non è stata completata.
- `501 Not Implemented`: La richiesta contiene degli header che non sono stati implementati e non possono essere ignorati.

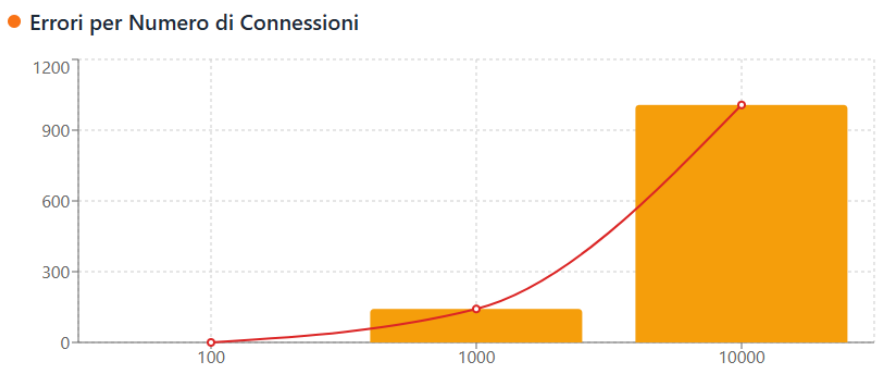
Presentazione dei Risultati

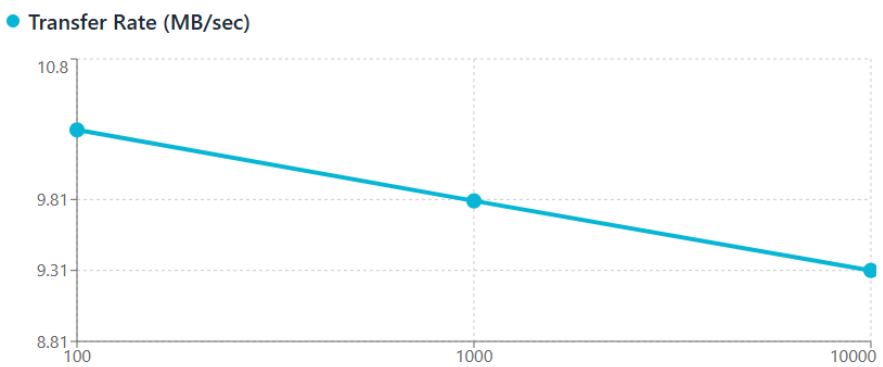
- Il server precedentemente illustrato si dimostra perfettamente funzionante quando contattato tramite il browser e riesce inoltre a rispondere ordinatamente a richieste multiple inviate come singolo pacchetto.
- Sono stati inoltre effettuati dei test di carico sul server simulando, 20 volte l'uno, un carico di 100 connessioni, un carico di 1000 e un carico di 10000.

Risultati dei test di carico









Conclusione

Dai risultati sperimentali ottenuti coi test di carico si può affermare che:

- La **latenza** del server cresce molto velocemente con l'aumentare delle connessioni.
- Il **throughput** rimane abbastanza stabile per tutte e tre i test carico.
- Gli errori aumentano progressivamente con l'aumentare delle connessioni.

Personalmente ritengo questi risultati più che soddisfacenti in quanto il dato positivo del throughput mi porta a credere che l'instabilità raggiunta infine dall'implementazione sia probabilmente dovuta al limite massimo di 10000 file descriptor che il processo è consentito a gestire.