Università di Bologna - Sede Cesena

Corso di Laurea in Ingegneria Informatica

Realizzazione di un Web Server minimale in Python e pubblicazione di un sito statico

Relazione Tecnica

Cattolico Giuseppe
Corso - Programmazione di Reti

12 giugno 2025

Indice

1 Introduzione **2**

2 Obiettivi del progetto **2**

3 Analisi dei Requisiti **2**
3.1 Requisiti funzionali 2
3.2 Requisiti non funzionali 2
3.3 Estensioni opzionali implementate 2

4 Progettazione del Sistema **3**

5 Implementazione delle funzioni **3**
5.1 Funzione `handle_client()` 3
5.2 Funzione `get_mime_type()` 6
5.3 Funzione `log_request()` 6
5.4 Funzione `main()` 7

6 Test e Validazione **8**

7 Estensioni Opzionali **8**

8 Estetica Sito web **8**

9 Considerazioni Finali **9**

1 Introduzione

Questa relazione descrive lo sviluppo di un semplice server HTTP in Python, realizzato nell'ambito del laboratorio di Programmazione di Reti. Il server è in grado di gestire richieste GET e servire contenuti statici HTML e CSS, con funzionalità base ed estensioni opzionali.

2 Obiettivi del progetto

L'obiettivo è implementare un web server minimale che:

- Ascolta su `localhost:8080`
- Gestisce richieste HTTP di tipo GET
- Serve almeno 3 pagine HTML statiche
- Risponde con codici HTTP appropriati (200 OK, 404 Not Found)
- Estensioni opzionali: MIME types, logging, layout responsive

3 Analisi dei Requisiti

3.1 Requisiti funzionali

- Supporto a richieste HTTP GET
- Risposta con codice 200 e contenuto corretto se il file esiste
- Risposta con codice 404 se il file non esiste

3.2 Requisiti non funzionali

- Utilizzo esclusivo di Python e socket TCP
- Nessuna dipendenza esterna o framework web

3.3 Estensioni opzionali implementate

- Rilevamento del tipo MIME
- Logging delle richieste
- Aggiunta di animazioni e layout responsive

4 Progettazione del Sistema

Il server si compone dei seguenti moduli:

- `main()`: inizializza il socket, ascolta le connessioni
- `handle_client()`: gestisce la singola connesione del client
- `get_mime_type()`: rileva il tipo di contenuto da restituire
- `log_request()`: stampa a console le richieste ricevute

Il server utilizza file contenuti nella cartella `www/`, e protegge da percorsi non autorizzati tramite validazione del path.

5 Implementazione delle funzioni

In questa sezione viene presentata un'analisi dettagliata di tutte le funzioni che compongono il codice principale del server. Ogni funzione è descritta evidenziandone il ruolo, le scelte implementative e le motivazioni progettuali.

5.1 Funzione `handle_client()`

La funzione `handle_client(connectionSocket, addr)` è responsabile della gestione completa di una singola connessione client. È progettata per essere eseguita in un thread separato, così da consentire al server di gestire più richieste simultaneamente.

```
def handle_client(connectionSocket, addr):

    try:
        message = connectionSocket.recv(BUFFER_SIZE)

        if not message:
            return

        request_data = message.decode('utf-8')
        request_lines = request_data.split('\n')
        request_line = request_lines[0]
        words = request_line.split()

        if len(words) < 3:
            return

        method, path, version = words

        print(message, '::', words[0], ':', words[1])

        if method != 'GET':
            response = "HTTP/1.1 405 Method Not Allowed\r\n\r\n"
            connectionSocket.send(response.encode('utf-8'))
            log_request(addr, request_line, "405 Method Not Allowed")
            return

        if path == '/':
```

```
            path = '/index.html'

        print(path, '||', path[1:])

        file_path = os.path.join(DOCUMENT_ROOT, path.lstrip('/'))

        if not os.path.exists(file_path) or not os.path.isfile(file_path
            ) or not os.path.abspath(file_path).startswith(os.path.
            abspath(DOCUMENT_ROOT)):

            connectionSocket.send(bytes("HTTP/1.1 404 Not Found\r\n\r\n"
                , "UTF-8"))
            connectionSocket.send(bytes("<html><head><title>404 Not
                Found</title></head><body><h1>404 Not Found</h1><p>La
                pagina richiesta non esiste.</p></body></html>\r\n", "UTF
                -8"))

            log_request(addr, request_line, "404 Not Found")
            connectionSocket.close()
            return

        try:
            with open(file_path, 'r+') as f:
                outputdata = f.read()

            print(outputdata)

            mime_type = get_mime_type(file_path)


            connectionSocket.send("HTTP/1.1 200 OK\r\n".encode())
            connectionSocket.send(f"Content-Type: {mime_type}\r\n".
                encode())
            connectionSocket.send(f"Content-Length: {len(outputdata)}\r\
                n\r\n".encode())
            connectionSocket.send(outputdata.encode())
            connectionSocket.send("\r\n".encode())

            log_request(addr, request_line, "200 OK")

        except IOError:
            connectionSocket.send(bytes("HTTP/1.1 500 Internal Server
                Error\r\n\r\n", "UTF-8"))
            connectionSocket.send(bytes("<html><head><title>500 Internal
                Server Error</title></head><body><h1>500 Internal Server
                Error</h1></body></html>\r\n", "UTF-8"))

            log_request(addr, request_line, "500 Internal Server Error")

    except Exception as e:
        print(f"Errore nella gestione del client: {str(e)}")
    finally:
        connectionSocket.close()
```

Listing 1: Funzione di gestione del client

Le principali fasi di questa funzione sono descritte di seguito:

- **Ricezione della richiesta:** si utilizza `recv()` per leggere i dati inviati dal client. Si

imposta un `BUFFER_SIZE` per limitare la dimensione massima del messaggio ricevuto.

- **Parsing della richiesta:** si decodifica il messaggio in UTF-8 e si estrae la prima riga della richiesta HTTP (il request line), che contiene metodo, percorso e versione HTTP. Questo parsing manuale è sufficiente per la gestione di richieste `GET` su file statici.
- **Gestione dei soli metodi GET:** la funzione verifica che il metodo sia `GET`, come richiesto dai requisiti. In caso contrario, risponde con il codice di stato `405 Method Not Allowed`.
- **Normalizzazione del percorso:** se il client richiede la root (`/`), viene reindirizzato al file `index.html` per coerenza con la struttura tipica di un sito statico.
- **Sicurezza del percorso:** viene costruito il percorso assoluto del file richiesto. Si controlla che il file esista, sia un file (e non una directory), e che il percorso risieda nella cartella `www/`. Questo evita vulnerabilità come path traversal (`../`) che potrebbero permettere di accedere a file esterni.
- **Lettura del file e risposta 200:** se il file esiste, viene letto in modalità testuale (`'r+'`) e inviato al client. L'header HTTP viene costruito manualmente includendo:
  - `HTTP/1.1 200 OK`
  - Il `Content-Type` determinato in base all'estensione tramite la funzione `get_mime_type()`
  - La lunghezza del contenuto (`Content-Length`)

  Infine, il contenuto del file viene inviato al client.
- **Risposta 404 Not Found:** se il file non esiste o non è accessibile, viene restituita una pagina HTML con messaggio di errore e codice di stato `404`.
- **Risposta 500 Internal Server Error:** in caso di errore durante la lettura del file, viene restituito un messaggio di errore generico con codice `500`.
- **Logging:** tutte le richieste vengono loggate nella console tramite la funzione `log_request()`, che mostra indirizzo del client, la richiesta e il codice restituito. Questo è utile per il debug e per il tracciamento delle attività.
- **Chiusura della connessione:** la connessione socket viene chiusa nel blocco `finally`, garantendo che venga sempre liberata la risorsa, anche in caso di errore.

Scelte progettuali:

- La gestione in thread consente una maggiore scalabilità rispetto ad una gestione seriale delle connessioni.
- L'elaborazione della richiesta HTTP è stata volutamente semplificata per scopi didattici, concentrandosi sul solo metodo `GET`.
- È stato preferito l'invio manuale dell'header HTTP per mostrare il formato effettivo del protocollo e permettere maggiore flessibilità nella costruzione delle risposte.
- L'utilizzo di controlli di sicurezza sui percorsi (come `os.path.abspath().startswith()`) evita potenziali exploit comuni nei web server.

5.2 Funzione `get_mime_type()`

La funzione `get_mime_type(file_path)` ha il compito di determinare il tipo MIME (Multipurpose Internet Mail Extensions) associato all'estensione del file richiesto. Questo tipo viene inserito nell'intestazione HTTP della risposta, permettendo al browser di interpretare correttamente il contenuto ricevuto.

```
def get_mime_type(file_path):
    extension = os.path.splitext(file_path)[1].lower()
    mime_types = {
        '.html': 'text/html',
        '.css': 'text/css',
        '.js': 'application/javascript',
        '.jpg': 'image/jpeg',
        '.jpeg': 'image/jpeg',
        '.png': 'image/png',
        '.gif': 'image/gif',
        '.txt': 'text/plain'
    }
    return mime_types.get(extension, 'application/octet-stream')
```

Spiegazione delle scelte:

- Si utilizza `os.path.splitext` per estrarre l'estensione del file.
- Le estensioni vengono convertite in minuscolo per evitare problemi dovuti a maiuscole (`.HTML`, `.JPG`).
- Si usa un dizionario per mappare le estensioni più comuni ai loro MIME type.
- In caso di estensione non riconosciuta, si restituisce `application/octet-stream`, ovvero il tipo generico per file binari.

Questa funzione è utile per migliorare la compatibilità del server con diversi tipi di file statici (HTML, CSS, immagini, JavaScript, ecc.).

5.3 Funzione `log_request()`

La funzione `log_request(client_address, request_line, status_code)` ha lo scopo di registrare nella console del server ogni richiesta ricevuta, con relativi dettagli.

```
def log_request(client_address, request_line, status_code):
    print(f"{client_address[0]}:{client_address[1]}-{request_line}-{
        status_code}")
```

Funzionalità:

- Mostra l'indirizzo IP e la porta del client;
- Mostra la riga della richiesta HTTP ;
- Mostra il codice di stato restituito dal server (es. `200 OK`, `404 Not Found`, ecc.).

Motivazioni:

- È utile durante la fase di test e debugging;
- Fornisce un semplice meccanismo di logging.

5.4 Funzione `main()`

La funzione `main()` rappresenta il punto di ingresso del server. Si occupa della configurazione e dell'avvio del socket, e accetta più connessioni client in modo sequenziale, in quanto non è stata implementata la logica del multithreading.

```
def main():
    serverSocket = socket(AF_INET, SOCK_STREAM)
    serverSocket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)

    try:
        if not os.path.exists(DOCUMENT_ROOT):
            os.makedirs(DOCUMENT_ROOT)
            print(f"Creata directory {DOCUMENT_ROOT}")

        server_address = ('localhost', serverPort)
        serverSocket.bind(server_address)

        serverSocket.listen(1)
        print('the web server is up on port:', serverPort)

        while True:
            print('Ready to serve...')

            connectionSocket, addr = serverSocket.accept()
            print(connectionSocket, addr)


            handle_client(connectionSocket, addr)

    except KeyboardInterrupt:
        print("\nServer arrestato.")
    except Exception as e:
        print(f"Errore del server: {str(e)}")
    finally:
        serverSocket.close()
```

Spiegazione delle scelte:

- **Creazione del socket:** si utilizza `AF_INET` per IPv4 e `SOCK_STREAM` per TCP.
- **SO_REUSEADDR:** evita errori se il socket è già in uso da esecuzioni precedenti.
- **Verifica della directory `www/`:** se non esiste, viene creata per assicurare che il server abbia contenuti da servire.
- **Bind e listen:** il socket viene associato all'indirizzo `localhost:8080` e messo in ascolto.
- **Gestione sicura:** nel blocco `try/except` si intercetta `KeyboardInterrupt` per terminare il server in modo ordinato.

6 Test e Validazione

Sono stati effettuati diversi test per verificare:

- Accesso a `/index.html`, `/pagina2.html`, `/pagina3.html`
- Risposta 404 a file inesistenti
- Compatibilità con browser

7 Estensioni Opzionali

- **MIME Types:** implementati per HTML, CSS, immagini.
- **Logging:** IP client, richiesta e codice di stato stampati a console.
- **Responsive Layout:** nella cartella `www/`, i file CSS includono regole responsive.

8 Estetica Sito web

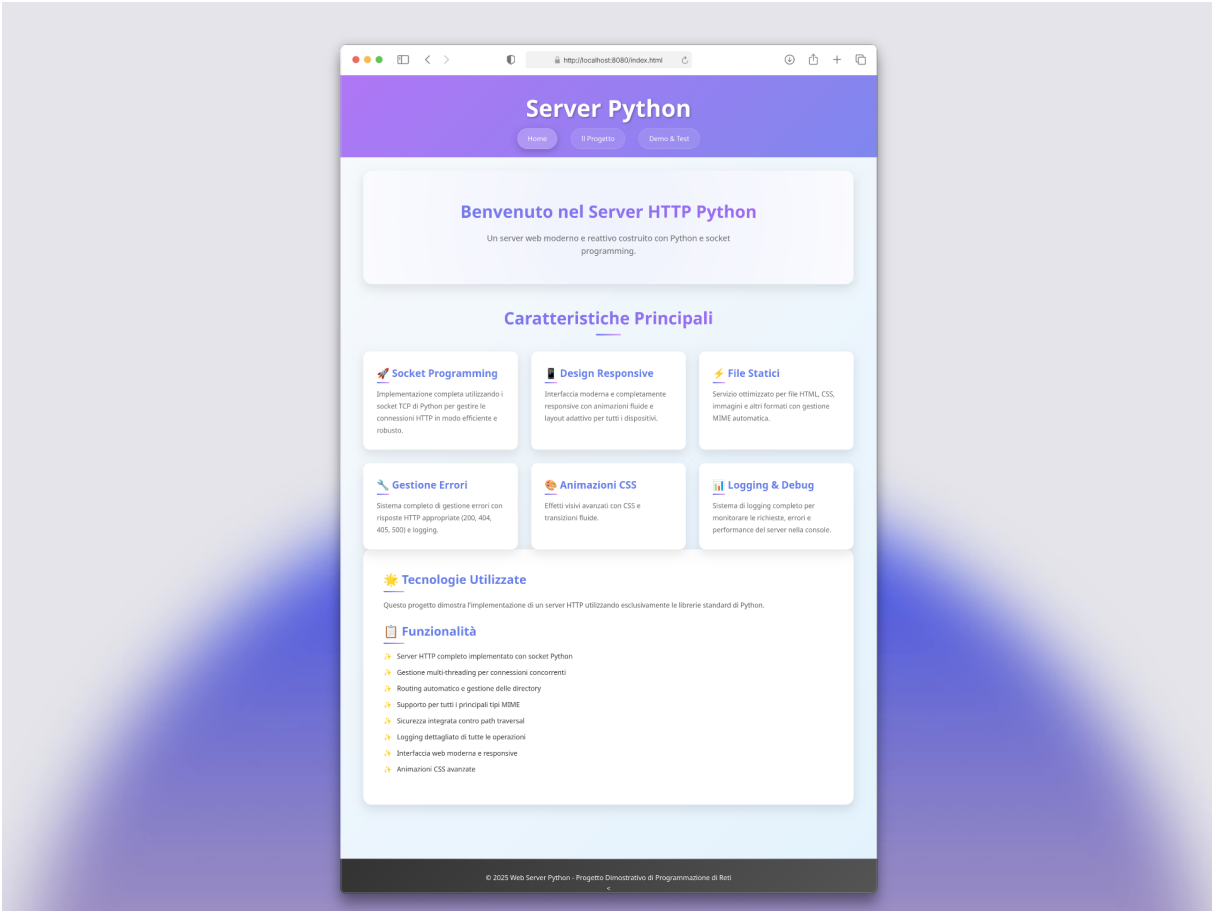

Figura 1: Home page

9 Considerazioni Finali

Il progetto ha rappresentato un'importante occasione di apprendimento pratico, permettendo di comprendere più a fondo il funzionamento delle richieste HTTP e la programmazione con i socket.

Durante lo sviluppo, sono emerse diverse sfide, tra cui:

- La corretta gestione del parsing delle richieste HTTP, che richiede attenzione particolare per evitare errori in presenza di richieste malformate o incomplete.
- L'implementazione della sicurezza nell'accesso ai file, per evitare vulnerabilità di tipo path traversal e assicurare che il server serva solo contenuti all'interno della directory `www`.
- La gestione degli errori e dei codici di risposta HTTP, con particolare attenzione alle situazioni di file non trovato (404) o errori interni (500).