# Relazione Tecnica – Progetto Reti

Autore: Vera Murgia
Matricola: 0001114041
Università di Bologna - Corso di Laurea triennale in Ingegneria e Scienze Informatiche
Anno 2025

## Introduzione

Il progetto consiste nella realizzazione di un **web server semplice in Python** capace di gestire richieste HTTP `GET` per servire pagine web statiche da una directory locale. È stato implementato utilizzando il modulo `socket` della libreria standard Python, senza l'uso di framework esterni, per fornire una comprensione di base dei meccanismi di comunicazione client-server su rete TCP/IP.

---

## Obiettivi

- Creare un server HTTP minimale in grado di ricevere e processare richieste GET da un browser.
- Servire file HTML statici da una directory `www`.
- Gestire le risposte HTTP con i codici di stato 200 (OK) e 404 (Not Found).
- Introdurre un sistema di logging basilare per le richieste ricevute.
- Implementare un layout responsive e animazioni CSS per migliorare l'esperienza utente delle pagine web servite.

---

## Architettura e Struttura del Progetto

- **Server Python (`server.py`):**
  Implementa il socket server TCP in ascolto sulla porta 8080. Per ogni connessione accettata, legge la richiesta HTTP, estrae il metodo e il percorso, quindi cerca il file richiesto nella cartella `www`. Se il file esiste, risponde con header 200 e il contenuto del file, altrimenti risponde con un messaggio 404.
- **Directory `www`:**
  Contiene i file HTML statici (`index.html`, `about.html`, `contact.html`, `404.html`) e il file CSS `style.css` per lo styling responsive e animazioni.
- **File CSS (`style.css`):**
  Gestisce il layout responsive tramite media queries, l'allineamento e il max-width per

rendere la visualizzazione ottimale su dispositivi desktop e mobili, oltre a definire animazioni di fade-in per gli elementi della pagina.

---

# Organizzazione del progetto

```
├── server.py
├── www/
│   ├── index.html
│   ├── about.html
│   ├── contact.html
│   ├── 404.html
│   └── style.css

```

La struttura del progetto segue il principio di separazione tra logica server e contenuti web.

---

# Descrizione tecnica

Il file principale `server.py` implementa le seguenti funzionalità:

- Creazione di un socket TCP con `AF_INET` e `SOCK_STREAM`.
- Ascolto su `localhost` porta `8080`.
- Gestione delle richieste HTTP tramite parsing della prima riga.
- Risposta con:
  - File richiesto (es. `index.html`) se presente.
  - Codice `404` e pagina di errore se il file non esiste.
  - Codice `405` se viene usato un metodo diverso da `GET`.

Il server serve file dalla directory `www`, in cui si trovano:

- `index.html`, `about.html`, `contact.html`, `404.html`
- un foglio di stile `style.css` condiviso tra tutte le pagine

---

# Funzionalità Implementate

## Gestione richieste HTTP

Il server supporta esclusivamente il metodo `GET`. Le richieste con metodi diversi ricevono un errore 405 Method Not Allowed.

## Servizio di pagine statiche

Viene restituito il contenuto dei file HTML richiesti, con gestione del percorso `/` che corrisponde a `index.html`. Se il file non è presente, viene inviata una risposta 404 con un messaggio di errore.

## Logging

Ogni richiesta ricevuta viene stampata in console per monitorare l'attività del server.

## Layout Responsive e Animazioni

Le pagine HTML sono state aggiornate con un file CSS esterno che implementa:

- un layout centrato con larghezza massima per facilitare la lettura;
- font chiari e leggibili;
- animazioni di fade-in per rendere l'esperienza visiva più gradevole;
- adattamento della dimensione del testo su dispositivi con larghezza inferiore a 600px.

---

# Funzionamento del server

Nel file `server.py` è dov'è contenuto il server. Vediamo come è strutturato e cosa fa ogni sua parte.

## Import e configurazione iniziale

```
1   import socket
2   import os
```

Importa i moduli necessari:

- `socket`: per creare e gestire la comunicazione TCP in python.
- `os`: per accedere ai file nel filesystem.

```
1   # Directory in cui si trova lo script server.py
2   BASE_DIR = os.path.dirname(os.path.abspath(__file__))
3
4   # Directory che contiene i file HTML, CSS, ecc.
5   WWW_DIR = os.path.join(BASE_DIR, 'www')
```

Imposto i percorsi assoluti per accedere ai file statici (HTML, immagini ecc.) nella cartella `www`.
Questo permette al server di funzionare anche se viene spostato in un'altra directory.

## Gestione della richiesta del client

```
def handle_request(client_socket):
```

La funzione `handle_request` gestisce una singola richiesta HTTP da parte del client.
Riceve, interpreta e risponde al client con il file richiesto o un errore.
È la funzione principale per gestire ogni richiesta HTTP ricevuta da un client.

```
    try:
        request = client_socket.recv(1024).decode('utf-8',
errors='ignore')
        if not request:
            client_socket.close()
            return
```

Ricevo e decodifico la richiesta.
Se è vuota (es. connessione aperta ma senza dati), chiudo il socket.

```
        request_line = request.splitlines()[0]
        print(f"[DEBUG] Richiesta: {request_line}")
```

Estraggo la **prima riga** della richiesta HTTP (es: `GET /index.html HTTP/1.1`) e la stampo a schermo per debug.

```
        try:
            method, path, _ = request_line.split()
        except ValueError:
            client_socket.close()
            return
```

Divido la riga in **metodo**, **percorso**, e **versione HTTP**.
Se fallisce (richiesta malformata), chiudo la connessione.

## Gestione metodo GET

```
        if method != 'GET':
            response = "HTTP/1.1 405 Method Not Allowed\r\n\r\nSolo il
metodo GET è supportato.".encode('utf-8')
```

Se il metodo non è `GET`, invio una risposta `405 Method Not Allowed`.

```
        else:
            if path == '/':
                path = 'index.html'
            else:
                path = path.lstrip('/')
```

```
6
```

Se il client chiede `/`, lo reindirizzo a `index.html`.
Rimuovo lo slash iniziale per costruire il percorso corretto del file richiesto.

```
            file_path = os.path.join(WWW_DIR, path)
```

Costruisco il **percorso assoluto** al file nella cartella `www`.

## File trovato

```
            if os.path.isfile(file_path):
                with open(file_path, 'rb') as f:
                    content = f.read()
                header = "HTTP/1.1 200 OK\r\n\r\n".encode('utf-8')
                response = header + content
```

Se il file esiste, lo leggo e lo invio con intestazione `200 OK`.

## File non trovato

```
            else:
                error_404_path = os.path.join(WWW_DIR, '404.html')
                if os.path.isfile(error_404_path):
                    with open(error_404_path, 'rb') as f:
                        content = f.read()
                    header = "HTTP/1.1 404 Not Found\r\n\r\n".encode('utf-8')
                    response = header + content
                else:
                    response = "HTTP/1.1 404 Not Found\r\n\r\nFile non trovato".encode('utf-8')
```

Se il file non esiste:

- Cerco di restituire una **pagina 404 personalizzata** (`404.html`).
- Se non esiste neanche quella, invio un messaggio di errore testuale.

```
        client_socket.sendall(response)
```

In ogni caso, invio la risposta completa al client.

```
    except Exception as e:
        print("[ERRORE]", e)
    finally:
        client_socket.close()
```

Se c'è un errore, lo stampo. Chiudo sempre la connessione alla fine.

## Funzione principale per avviare il server

```
def start_server():
    host = 'localhost'
    port = 8080
```

Il server ascolta sulla porta `8080` di `localhost`.

```
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
        server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        server_socket.bind((host, port))
        server_socket.listen(5)
        print(f"Server in ascolto su http://{host}:{port}")
```

Creo il socket TCP, lo associo alla porta, e lo metto in ascolto.
`SO_REUSEADDR` serve per evitare errori se si riavvia il server rapidamente.

```
        while True:
            client_socket, addr = server_socket.accept()
            print(f"[CONNESSIONE] Da {addr}")
            handle_request(client_socket)
```

Accetto connessioni in un ciclo infinito.
Ogni nuova connessione viene gestita dalla funzione `handle_request`.

## Avvio del programma

```
if __name__ == '__main__':
    start_server()
```

Avvia il server solo se il file viene eseguito direttamente (non importato da altri).

---

# Interfaccia utente

Le pagine HTML sono responsive e animate:

- Layout centrato, massimo 800px, compatibile con mobile.
- Animazioni `fadeIn` e `fadeInUp` per un effetto più moderno.
- Navigazione tra le pagine tramite link `nav`.
- Pagina di errore 404 con design coerente e messaggio utente.

## Funzionamento e test

- Il server è stato testato su `localhost:8080`.
- Sono state verificate:
  - la visualizzazione delle pagine (`index.html`, `about.html`, `contact.html`)
  - il corretto caricamento del CSS
  - la visualizzazione della pagina `404.html` in caso di errore

---

## Conclusioni

Il progetto costituisce un ottimo punto di partenza per comprendere i fondamenti della comunicazione HTTP e la gestione di server TCP in Python. Nonostante la semplicità, è possibile estendere questo prototipo per costruire un web server più robusto e completo.

---