

Relazione di Progetto Web Server e Sito Web

Giovanni Maria Rava

14 luglio 2025

Indice

1	Introduzione	2
1.1	Obbiettivo	2
1.2	Requisiti minimi	2
1.3	Estensioni opzionali	3
2	Fondamenti Teorici	4
2.1	Protocollo HTTP	4
2.2	Socket	5
3	Architettura del Server	6
3.1	Fase di inizializzazione	6
3.2	Ciclo principale del server	7
3.3	Ricezione e parsing della richiesta	7
3.4	Recupero del file richiesto	7
3.5	Costruzione della risposta HTTP	8
3.6	Logging delle richieste	8
3.7	Chiusura della connessione	8
4	Analisi tramite Wireshark e Console	9
4.1	Richiesta HTTP GET	9
4.2	Risposta HTTP con codice 200 OK	10
4.3	Richiesta di file inesistente: errore 404	11
4.4	Log della console del server	11
5	Considerazioni finali	13

Capitolo 1

Introduzione

Questa relazione documenta lo sviluppo di un semplice Web Server scritto in linguaggio Python (versione 3.12), in grado di gestire richieste HTTP e servire contenuti statici in formato HTML e CSS. Il progetto ha finalità didattiche ed è stato realizzato nell'ambito del corso di Programmazione di Reti (codice: 70226), durante l'Anno Accademico 2024/2025.

Durante lo sviluppo del progetto sono stati utilizzati diversi ambienti di lavoro e strumenti di sviluppo, elencati di seguito:

- **Spyder 6**, per il debugging interattivo del codice Python;
- **Visual Studio Code**, per l'editing del codice e la gestione dei file HTML/CSS;
- **GitHub**, per il versionamento e l'archiviazione del progetto.

1.1 Obiettivo

Progettare un semplice server HTTP in Python (usando socket) e servire un sito web statico con HTML/CSS.

1.2 Requisiti minimi

- Il server deve rispondere su localhost:8080
- Deve servire almeno tre pagine HTML statiche
- Gestione di richieste GET e risposta con codice 200
- Implementare risposta 404 per file inesistenti

1.3 Estensioni opzionali

- Gestione dei MIME types (.html, .css, .jpg)
- Logging delle richieste
- Aggiunta di animazioni o layout responsive

Capitolo 2

Fondamenti Teorici

2.1 Protocollo HTTP

Il Hypertext Transfer Protocol (HTTP) è il protocollo standard per la comunicazione tra client e server sul Web . HTTP funziona secondo un modello stateless, ovvero ogni transazione è indipendente e non si tiene memoria di essa. Utilizza comandi testuali standard come GET per richiedere risorse. All'interno di HTTP sono presenti messaggi di due tipi:

- richiesta
- risposta

Ogni messaggio è formato da una intestazione (*header*) seguita dal corpo (*body*). L'intestazione è composta da una serie di righe di testo terminate da caratteri di fine linea. Una richiesta inizia con una riga di richiesta, seguita da una o più righe di intestazione. Una risposta inizia con una riga di stato, seguita da una o più righe di intestazione. All'interno del body vengono contenuti i dati da trasferire.

In questo progetto viene implementata in particolare la richiesta **GET**, che consiste nella richiesta di una pagina al server.

Esempio di richiesta:

```
GET /index.html HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0
```

Vengono anche gestiti con particolare attenzione i codici di stato **200** che corrisponde ad **OK** e **404 Not found**

2.2 Socket

Il **socket** è un'interfaccia che le applicazioni usano per interagire con i protocolli dello strato di trasporto. È fornita dal sistema operativo in esecuzione sull'host ed è accessibile tramite primitive di comunicazione. Il concetto di socket si basa sul modello di Input/Output su file di Unix, quindi sulle operazioni di **open**, **read**, **write** e **close**; l'utilizzo avviene secondo le stesse modalità aggiungendo i parametri utili alla comunicazioni quali indirizzi ip, numeri di porta e protocolli. Il socket HTTP, in particolare, viene usato per trasmettere messaggi HTTP tra client e server web.

Capitolo 3

Architettura del Server

L'architettura del server HTTP realizzato è di tipo **sequenziale e monoth-read**, ed è progettata per rispondere a richieste HTTP in ingresso attraverso socket TCP. L'obiettivo è servire contenuti statici (HTML, CSS, immagini) situati nella directory `www/`, restituendo risposte conformi al protocollo HTTP.

Il server è costituito da un unico file `server.py`, suddivisibile logicamente in tre macro-componenti:

- **Fase di inizializzazione** (apertura del socket e ascolto)
- **Gestione della richiesta** (ricezione, parsing, recupero file)
- **Generazione della risposta** (header HTTP, corpo e codice di stato)

3.1 Fase di inizializzazione

Il server crea e configura il socket TCP che lo mette in ascolto su `localhost:8080`:

```
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('localhost', serverPort))
serverSocket.listen(1)
```

Questa fase prepara il server ad accettare una connessione per volta tramite il metodo `accept()`.

3.2 Ciclo principale del server

Il server entra in un ciclo infinito dove attende, elabora e risponde alle richieste dei client:

```
while True:
    connectionSocket, addr = serverSocket.accept()
    ...
    connectionSocket.close()
```

Ogni iterazione rappresenta l'elaborazione completa di una singola connessione HTTP.

3.3 Ricezione e parsing della richiesta

Il server riceve il messaggio dal client, che rappresenta una richiesta HTTP:

```
message = connectionSocket.recv(1024).decode()
parts = message.split()
method = parts[0]
path = parts[1][1:]
```

- `recv(1024)` riceve al massimo 1024 byte dalla richiesta.
- `split()` separa la richiesta in blocchi (es. `GET /index.html HTTP/1.1`).
- Viene estratto il metodo HTTP (`GET`) e il percorso richiesto.
- Se il path è vuoto, viene servito per default `index.html`.

3.4 Recupero del file richiesto

Il server costruisce il path assoluto al file richiesto nella directory `www/`:

```
filepath = os.path.join('www', path)
if not os.path.isfile(filepath):
    ...
```

Se il file non esiste, viene generata una risposta `404 Not Found`. Altrimenti, il contenuto viene letto in modalità binaria.

3.5 Costruzione della risposta HTTP

A seconda dell'esito del controllo sul file, vengono costruite le intestazioni e il corpo della risposta. Ad esempio, in caso positivo:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1234
```

Nel codice:

```
content_type = get_content_type(filepath)
header = (
    "HTTP/1.1 200 OK\\r\\n"
    f"Content-Type: {content_type}\\r\\n"
    f"Content-Length: {len(body)}\\r\\n"
    "Connection: close\\r\\n"
    "\\r\\n"
).encode()
connectionSocket.sendall(header + body)
```

3.6 Logging delle richieste

Ad ogni richiesta viene stampato un log nella console, utile per il monitoraggio e il debugging:

```
log_request(addr, method, path, 200)
```

Esempio di output su console:

```
[2025-07-10 11:45:01] ('127.0.0.1', 54322) - GET index.html -> 200
```

3.7 Chiusura della connessione

Infine, il server chiude il socket dedicato alla comunicazione col client:

```
connectionSocket.close()
```

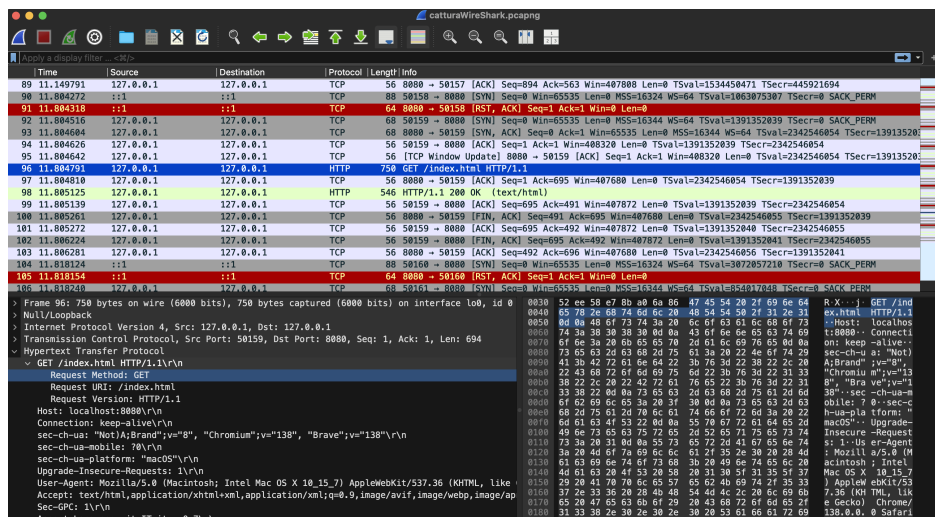
Capitolo 4

Analisi tramite Wireshark e Console

Per validare il corretto funzionamento del Web Server e osservare il traffico di rete generato, è stato utilizzato il tool **Wireshark**. Tramite il monitoraggio dell'interfaccia Loopback, è stato possibile catturare i pacchetti scambiati tra il browser (client) e il server Python in ascolto su localhost:8080.

4.1 Richiesta HTTP GET

Nell'immagine seguente è mostrato il pacchetto contenente la richiesta GET da parte del client per il file /index.html:



Time	Source	Destination	Protocol	Length	Info
89	11.149791	127.0.0.1	TCP	56	8880 → 50157 [ACK] Seq=894 Ack=563 Win=407880 Len=0 TSval=1534458471 TSecr=445921694
90	11.884272	:::1	TCP	88	50157 → 8880 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=1862087387 TSecr=0 SACK_PERM
91	11.884318	:::1	TCP	64	8880 → 50157 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
92	11.884516	127.0.0.1	TCP	68	50159 → 8880 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=1391352039 TSecr=0 SACK_PERM
93	11.884604	127.0.0.1	TCP	68	8880 → 50159 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=2342546054 TSecr=1391352039
94	11.884626	127.0.0.1	TCP	56	50159 → 8880 [ACK] Seq=1 Ack=1 Win=408320 Len=0 TSval=1391352039 TSecr=2342546054
95	11.884642	127.0.0.1	TCP	56	[TCP Window Update] 8880 → 50159 [ACK] Seq=1 Ack=1 Win=408320 Len=0 TSval=2342546054 TSecr=1391352039
96	11.884791	127.0.0.1	HTTP	750	GET /index.html HTTP/1.1
97	11.884810	127.0.0.1	TCP	56	8880 → 50159 [ACK] Seq=1 Ack=695 Win=407680 Len=0 TSval=2342546054 TSecr=1391352039
98	11.885125	127.0.0.1	HTTP	546	HTTP/1.1 200 OK (text/html)
99	11.885139	127.0.0.1	TCP	56	50159 → 8880 [ACK] Seq=695 Ack=491 Win=407872 Len=0 TSval=1391352039 TSecr=2342546054
100	11.885261	127.0.0.1	TCP	56	8880 → 50159 [FIN, ACK] Seq=491 Ack=695 Win=407680 Len=0 TSval=2342546055 TSecr=1391352039
101	11.885272	127.0.0.1	TCP	56	50159 → 8880 [ACK] Seq=695 Ack=492 Win=407872 Len=0 TSval=1391352040 TSecr=2342546055
102	11.886224	127.0.0.1	TCP	56	50159 → 8880 [FIN, ACK] Seq=695 Ack=492 Win=407872 Len=0 TSval=1391352041 TSecr=2342546055
103	11.886281	127.0.0.1	TCP	56	8880 → 50159 [ACK] Seq=492 Ack=696 Win=407680 Len=0 TSval=2342546056 TSecr=1391352041
104	11.818124	:::1	TCP	88	50160 → 8880 [SYN] Seq=0 Win=65535 Len=0 MSS=16324 WS=64 TSval=3072057210 TSecr=0 SACK_PERM
105	11.818154	:::1	TCP	64	8880 → 50160 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
106	11.818199	127.0.0.1	TCP	68	50161 → 8880 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=854917245 TSecr=0 SACK_PERM

Frame 96: 750 bytes on wire (6000 bits), 750 bytes captured (6000 bits) on interface lo0, id 0

Null/Loopback

Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

Transmission Control Protocol, Src Port: 50159, Dst Port: 8080, Seq: 1, Ack: 1, Len: 694

Hypertext Transfer Protocol

GET /index.html HTTP/1.1\r\n

Request Method: GET

Request URI: /index.html

Request Version: HTTP/1.1

Host: localhost:8080\r\n

Connection: keep-alive\r\n

sec-ch-ua: "NotABrand";v="8", "Chromium";v="138", "Brave";v="138"\r\n

sec-ch-ua-mobile: ?0\r\n

sec-ch-ua-platform: "macOS"\r\n

Upgrade-Insecure-Requests: 1\r\n

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/139.0.0.0 Safari/139.0.0.0

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng;q=0.8,application/signed-exchange;v=b3;q=0.7\r\n

Accept-Language: it-IT,it;q=0.7\r\n

Figura 4.1: Pacchetto HTTP contenente richiesta GET

Come si nota dalla sezione “Hypertext Transfer Protocol” di Wireshark, la richiesta include:

- Il metodo GET
- La versione HTTP HTTP/1.1
- Header come Host, User-Agent, ecc.

4.2 Risposta HTTP con codice 200 OK

A seguito della richiesta, il server restituisce il file richiesto con codice di stato 200 OK:

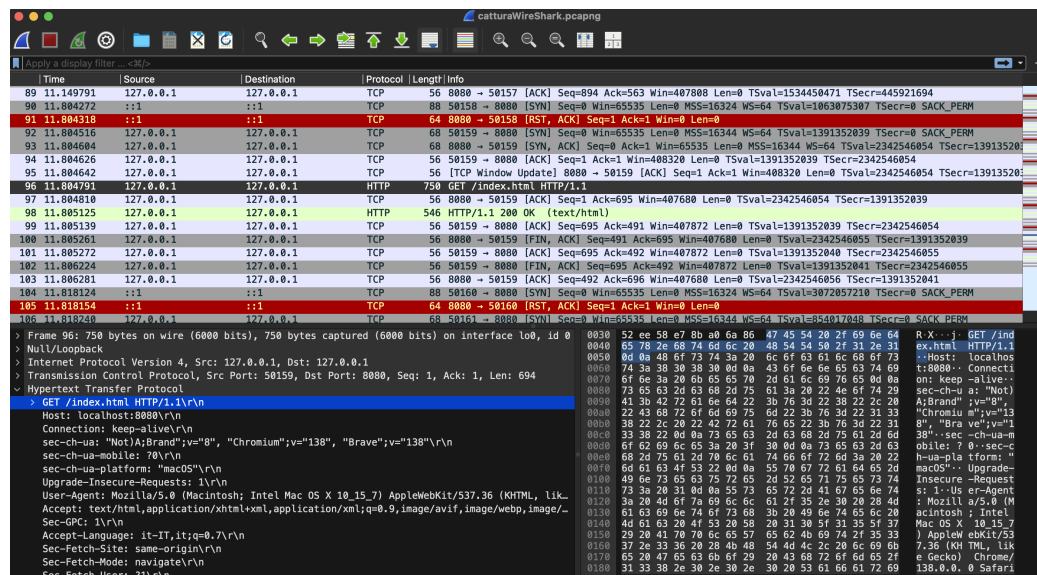


Figura 4.2: Risposta HTTP con codice 200 OK e intestazioni corrette

L’header della risposta include:

- Codice di stato: HTTP/1.1 200 OK
- Content-Type: text/html
- Content-Length
- Connection: close

4.3 Richiesta di file inesistente: errore 404

Se il client tenta di accedere a una risorsa non esistente (es. /contct.html), il server risponde con il messaggio di errore 404 Not Found, come mostrato nella figura seguente:

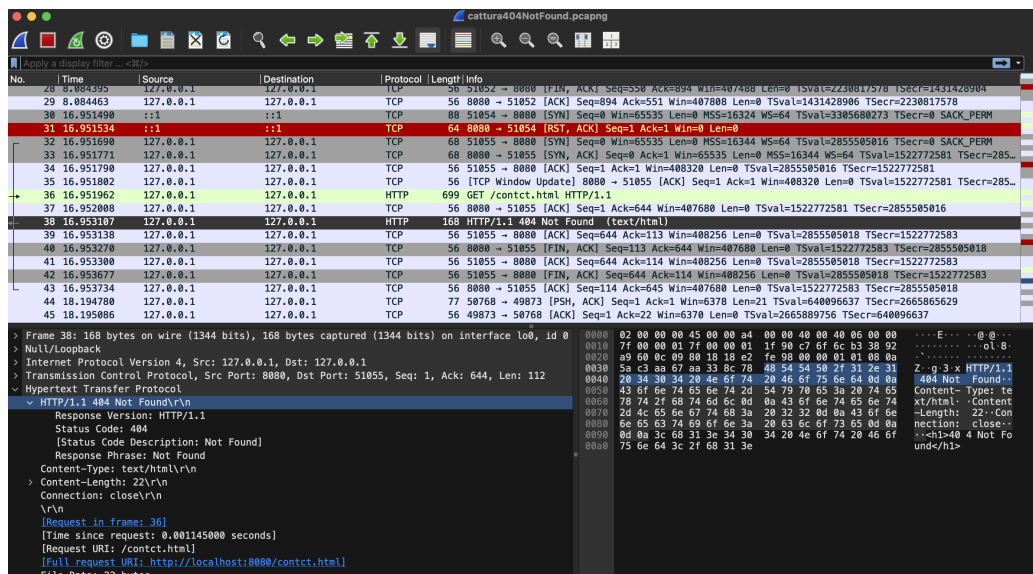


Figura 4.3: Risposta HTTP con errore 404 Not Found

4.4 Log della console del server

Per ogni richiesta ricevuta, il server produce un log in console con informazioni utili per il debugging. Un esempio di output è:

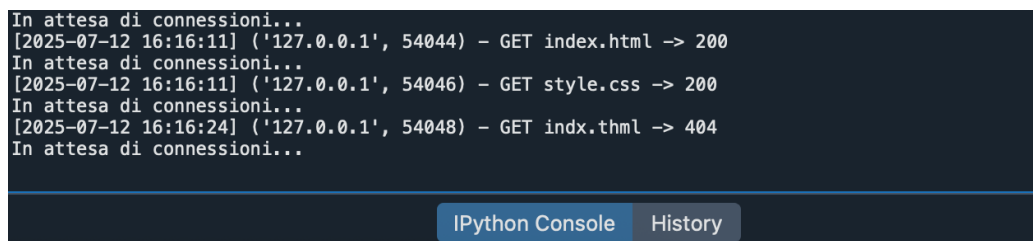


Figura 4.4: Risposta HTTP con codice 200 OK e intestazioni corrette

Questo log mostra:

- Timestamp della richiesta

- Indirizzo IP e porta del client
- Metodo e risorsa richiesta
- Codice di stato HTTP restituito

Capitolo 5

Considerazioni finali

L'analisi con Wireshark ha confermato che il server genera messaggi HTTP validi secondo specifica e risponde coerentemente a tutte le richieste. L'output in console permette di tracciare l'attività del server, facilitando il debugging e l'eventuale estensione futura del progetto. Il server implementa le funzionalità minime di un server HTTP per contenuti statici ma ha molte potenzialità di sviluppo con altre funzionalità, ad esempio, l'aggiunta di altre pagine, il miglioramento della grafica del sito e del contenuto.