Elaborato per il corso
Programmazione di Reti

Grazia Bochdanovits de Kavna

31 agosto 2025

Indice

Capitolo 1

Introduzione

Il presente progetto ha l'obiettivo di realizzare un **web server HTTP** in **Python**, realizzato utilizzando socket per gestire le connessioni di rete. Il server è progettato per rispondere sulla **porta 8080** di `localhost` e servire contenuti statici dalla directory `www/`.

Il lavoro implementa tutti i requisiti minimi richiesti dalla traccia: gestione delle richieste `GET` con risposta 200, servizio di almeno tre pagine HTML statiche, e corretta gestione degli errori 404 per file inesistenti.

Sono state inoltre implementate le estensioni opzionali previste, tra cui il supporto per *MIME types* (HTML, CSS), il logging delle richieste con *timestamp*, e un layout responsive con animazioni CSS.

Il server utilizza **multithreading** per gestire connessioni concorrenti e include misure di sicurezza per prevenire attacchi di directory traversal. Il progetto comprende anche pagine di errore personalizzate (`404.html`, `501.html`) e un design moderno.

Capitolo 2

Architettura del progetto

L'architettura del progetto si basa su una chiara separazione tra la logica applicativa del server e i contenuti statici destinati al client. Il sistema è organizzato nei seguenti componenti principali:

- **server.py**: implementa il web server HTTP multithread, in ascolto sulla porta 8080, responsabile della gestione delle connessioni e del servizio dei file statici.
- **www/**: directory contenente il sito web statico, composto da cinque pagine HTML e un file CSS condiviso, con layout responsive e design moderno.

Capitolo 3

Implementazione del web server

3.1 Struttura generale

Il web server è stato implementato in Python utilizzando esclusivamente moduli della standard library. L'architettura segue il pattern classico dei server HTTP multi-threading, dove ogni connessione client viene gestita da un thread separato. La classe principale `HTTPServer` incapsula tutta la logica del server.

La struttura del codice è organizzata nei seguenti componenti principali:

- **Classe HTTPServer**: Classe principale che gestisce l'intero ciclo di vita del server
- **Gestione delle connessioni**: Accettazione delle connessioni e creazione di thread dedicati
- **Parsing delle richieste**: Analisi degli header HTTP e estrazione dei parametri
- **Resolving dei path**: Conversione sicura degli URL in percorsi del filesystem
- **Servizio dei file**: Gestione dei file statici con controllo dei permessi
- **Gestione degli errori**: Produzione di pagine di errore appropriate

3.2 Gestione delle richieste

Il processo di gestione delle richieste segue un flusso ben definito:

3.2.1 Accettazione delle connessioni

Il server rimane in attesa su una socket configurata con opzioni di riutilizzo dell'indirizzo. Quando una connessione viene accettata, viene creato un nuovo thread che gestirà il client fino al completamento della richiesta.

```
def _run_server_loop(self):
    self.running = True
    while self.running:
        try:
            client_socket, client_address = self.socket.
               accept()
            thread = threading.Thread(
                target=self._handle_client,
                args=(client_socket, client_address),
                daemon=True
            )
            thread.start()
        except socket.timeout:
            continue
        except KeyboardInterrupt:
            self.logger.info("Shutting down...")
            break
        except Exception as e:
            if self.running:
                self.logger.error(f"Accept error: {e}")
            break
```

3.2.2 Parsing della richiesta

La richiesta HTTP viene analizzata estraendo il metodo, il path e la versione del protocollo. Sono supportati esclusivamente richieste GET secondo lo standard HTTP/1.x.

```
def _process_request(self, client_socket, request_data):
    lines = request_data.split('\n')
    if not lines or not lines[0].strip():
        return self._send_error(client_socket, 400)

    parts = lines[0].strip().split()
    if len(parts) != 3:
        return self._send_error(client_socket, 400)

    method, path, version = parts
```

```
    if method != 'GET':
        return self._send_error(client_socket, 405)
    if not version.startswith('HTTP/1.'):
        return self._send_error(client_socket, 400)

    self._handle_get(client_socket, path)
```

3.2.3 Resolving del percorso

Il path richiesto viene normalizzato e convertito in un percorso del filesystem locale, con controlli di sicurezza per prevenire directory traversal attacks.

```
def _resolve_path(self, url_path):
    try:
        path = unquote(urlparse(url_path).path)
        path = os.path.normpath(path)

        if path in ('/', '.'):
            path = '/home.html'
        if path.startswith('/'):
            path = path[1:]

        full_path = os.path.join(self.www_dir, path)

        # Security check
        if os.path.abspath(full_path).startswith(self.
           www_dir):
            return full_path

    except Exception as e:
        self.logger.warning(f"Path resolution error: {e}
           ")
    return None
```

3.2.4 Servizio dei file

I file vengono serviti con il corretto Content-Type determinato automaticamente. Sono implementati controlli per:

- Dimensioni massime dei file (10MB)
- Tipi di file supportati (esclusi .php, .jsp, .asp)

- Gestione automatica di index.html per le directory
- Pagine di errore personalizzabili

```
def _serve_file(self, client_socket, file_path):
    try:
        # Size limit check
        if os.path.getsize(file_path) > 10 * 1024 *
           1024:
            return self._send_error(client_socket, 413)

        with open(file_path, 'rb') as f:
            content = f.read()

        mime_type = mimetypes.guess_type(file_path)[0]
           or 'application/octet-stream'
        self._send_response(client_socket, 200, content,
            mime_type)

    except FileNotFoundError:
        self._send_error(client_socket, 404)
    except PermissionError:
        self._send_error(client_socket, 403)
    except Exception as e:
        self.logger.error(f"File serve error: {e}")
        self._send_error(client_socket, 500)
```

3.2.5 Gestione degli errori

Il server produce pagine di errore appropriate per i vari codici di stato HTTP. Sono supportate sia pagine personalizzate (es. 404.html) che pagine di default generate dinamicamente.

```
def _get_error_content(self, status_code):
    # Try custom error page first
    error_file = os.path.join(self.www_dir, f"{
       status_code}.html")
    if os.path.isfile(error_file):
        try:
            with open(error_file, 'r', encoding='utf-8')
                as f:
                return f.read()
        except Exception:
```

```
            pass

    # Default error page
    status_text = self.STATUS_CODES.get(status_code, "Error")
    return f"""<!DOCTYPE html>
        <html><head><title>{status_code} {status_text}</title></head>
        <body style="font-family:Arial;text-align:center;margin-top:100px;">
        <h1 style="color:#e74c3c;">{status_code}</h1>
        <h2>{status_text}</h2>
        <p>La risorsa richiesta non    disponibile.</p>
        <a href="/" style="color:#3498db;">Torna alla homepage</a>
        </body></html>"""
```

3.2.6 Logging

Tutte le attività del server vengono registrate sia su file che su console, includendo timestamp, indirizzi IP client e richieste effettuate.

```
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[logging.FileHandler('server.log'), logging.StreamHandler()]
)
```

Questa implementazione garantisce un server stabile, sicuro e conforme agli standard HTTP di base, adatto per servire contenuti statici in ambienti didattici e di testing.

Capitolo 4

Sito web statico

Il sito web è composto da cinque pagine HTML statiche, tutte stilizzate mediante un unico file CSS condiviso (`common.css`).

4.1 Pagine HTML

- **home.html**: Homepage del sito con navigazione principale
- **login.html**: Pagina di autenticazione con form per le credenziali
- **contact.html**: Pagina di contatto con informazioni e form
- **404.html**: Pagina di errore personalizzata per risorse non trovate
- **501.html**: Pagina per funzionalità non ancora implementate

Capitolo 5

Conclusioni

Il server sviluppato in Python soddisfa pienamente i requisiti della traccia, rispondendo su `localhost:8080`, gestendo correttamente richieste GET e fornendo pagine di errore personalizzate. Sono state realizzate tutte le estensioni opzionali, includendo la gestione dei MIME type, il logging delle richieste e un sito statico con layout responsive.

5.1 Sviluppi Futuri

Il progetto, pur essendo completo e funzionale, può essere esteso con diverse funzionalità avanzate come il supporto HTTPS per comunicazioni cifrate e l'implementazione di metodi HTTP aggiuntivi come POST per gestire l'invio di dati dai form. Queste estensioni renderebbero il server più versatile e adatto ad applicazioni reali.

Appendice A

Guida utente

A.1 Installazione e Avvio

1. **Clonazione del repository**:
   `git clone https://github.com/this-Grace/ProgrammazioneReti-project.git`
2. **Accesso alla directory**:
   `cd ProgrammazioneReti-project`
3. **Avvio del server**:
   `python server.py`
4. **Accesso al sito**:
   Aprire un browser e navigare su `http://localhost:8080`

A.2 Accesso alle pagine

- **Homepage**: `http://localhost:8080/`
- **Pagina di login**: `http://localhost:8080/login.html`
- **Pagina di contatti**: `http://localhost:8080/contact.html`
- **Pagina 404**: `http://localhost:8080/404.html`
- **Pagina 501**: `http://localhost:8080/501.html`

A.2.1 Note sull'accesso

- La root `/` restituisce automaticamente `home.html`.

- File non esistenti mostrano la pagina `404.html`.
- Il form di login reindirizza a `501.html`.

A.3 Arresto del Server

- Premere `Ctrl+C` nel terminale.
- Attendere la chiusura delle connessioni.
- Verificare il messaggio: `Server stopped.`