LabReSiD25: HTTP Server

Marco Iaria, Matricola 557295

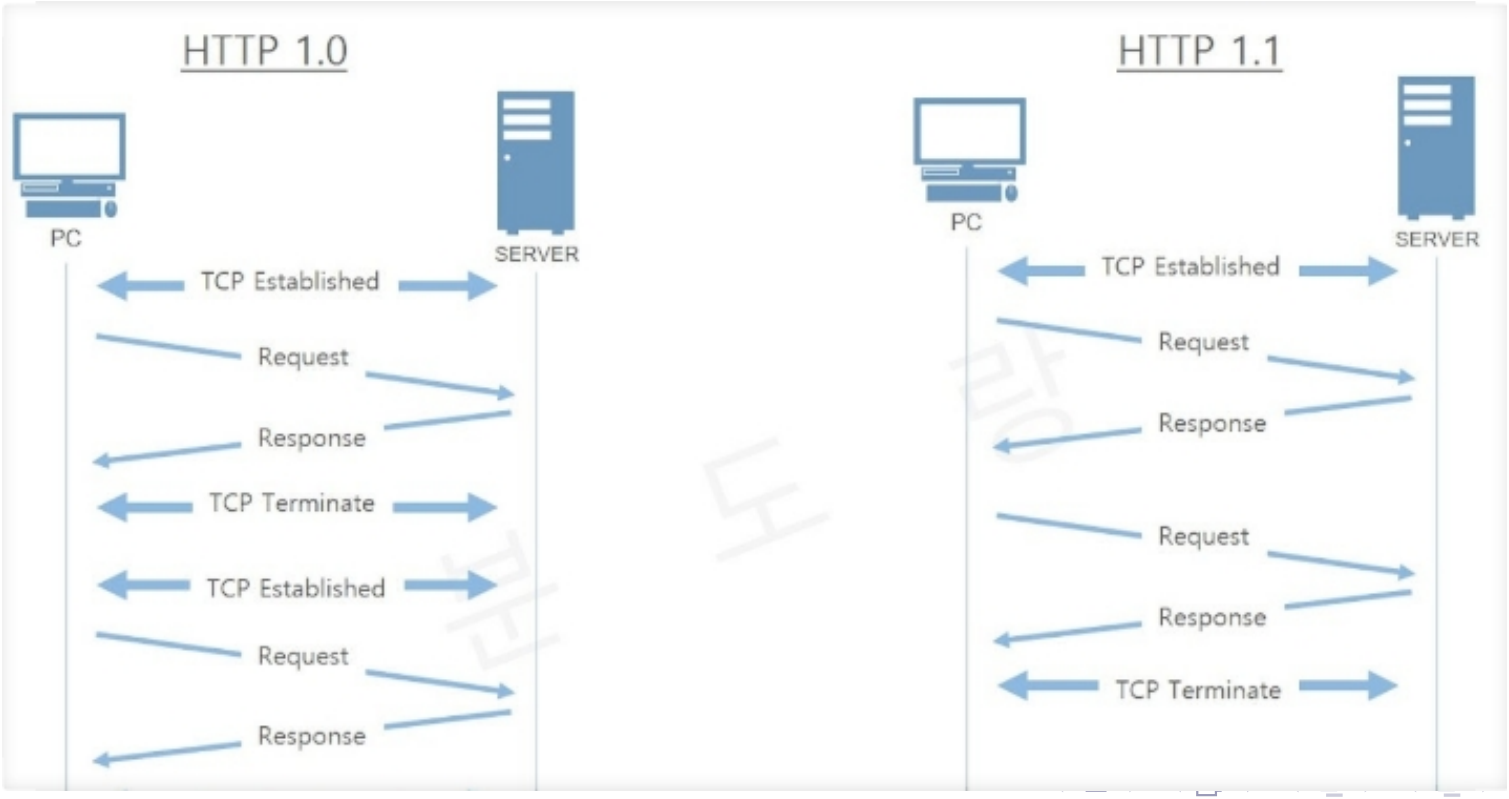
November 2025

Introduzione

E' possibile unire la tecnologia dei **socket** con quella dei **thread** per programmare un server HTTP concorrente conforme ad RFC 2616. La presentazione dimostra come sia possibile crearlo con il linguaggio C usando la **programmazione modulare**.
E' stato realizzato un **Server HTTP concorrente** in grado di gestire le richieste `GET`, `POST`, `PUT`, `DELETE` e le risposte
`200`, `201`, `204`, `400`, `401`, `404`, `500` e `501`.

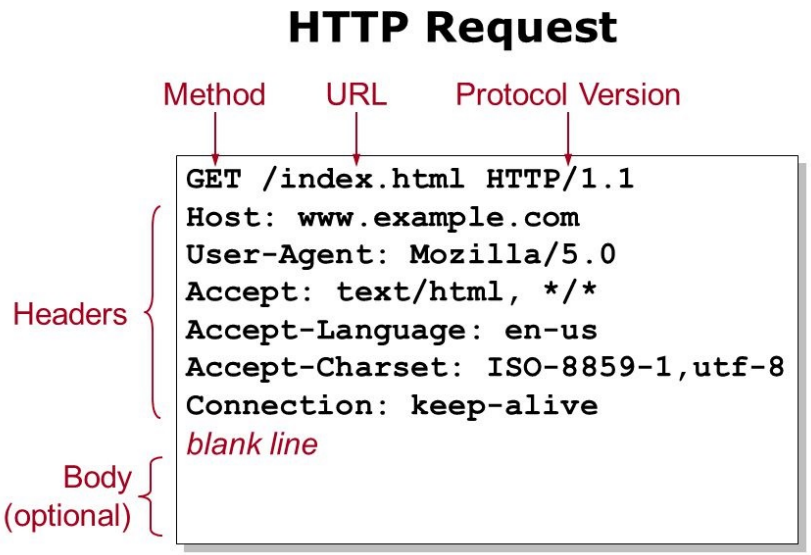
HTTP 1.0 vs HTTP 1.1

La caratteristica più importante che contraddistingue un server HTTP 1.1 ad un server HTTP 1.0 è la persistenza delle connessioni. Un server HTTP 1.0 accetta una connessione **TCP**, risponde ad una richiesta e chiude la connessione. Questo non accade nel caso di un server HTTP 1.1, che lascia la connessione aperta in attesa di ulteriori richieste.



Request HTTP

Una richiesta HTTP 1.1 è composta da una **Request-Line**, un **Message Header** e un **Message Body** opzionale.



Il server deve essere in grado di leggere e comprendere la richiesta testuale inviata dal client.

Gestione autenticazione

Per ognuna delle quattro richieste implementate e' stata sviluppata una funzione, con particolare riguardo verso le richieste `DELETE`, `PUT` e `POST` che sono protette da un **layer di autenticazione**.
Il sistema di autenticazione e' basato su `Base64`, algoritmo di encoding che utilizza la tabella ASCII per convertire 3 caratteri da 8 bit in 4 caratteri da 6 bit.

Base64	a	G	V	s	b	G	8	h
decimal	26	6	21	44	27	6	60	33
binary value	011010	000110	010101	101100	011011	000110	111100	100001

binary value	01101000	01100101	01101100	01101100	01101111	00100001
ASCII nr	104	101	108	108	111	33
characters	h	e	l	l	o	!

La sfida - Gestione Connessioni

Per gestire migliaia di connessioni persistenti contemporaneamente e in modo efficiente, linux ci permette di usare `epoll`, un meccanismo del kernel per monitorare migliaia di file descriptor.
Tuttavia questo non è sufficiente, in quanto le operazioni di creazione/distruzione di thread per ogni richiesta è costoso. Per risolvere questo problema è necessario sviluppare un **Thread Pool**, una collezione di thread già creati pronti ad eseguire tasks.
I componenti di un thread pool sono diversi:

- Array di thread;
- Coda dei lavori;
- Mutex per proteggere la coda;
- Variabile di condizione per svegliare i thread;
- Flag per terminare il pool.

Inizializzazione Server

1. **Init Thread Pool**:
   - Inizializzazione mutex e condition variables;
   - Creazione worker threads;
2. **Init Socket**: Creazione server socket;
3. **Init epoll**: Inizializzazione istanza epoll.

Pseudocodice: epoll

Il server inizializza epoll in modalità Level-Triggered, accetta le connessioni in arrivo e, prima di aggiungere una richiesta alla coda, rimuove il client socket richiedente dall'istanza epoll, passando la gestione della connessione al worker thread. Questo viene fatto per non ricevere la stessa notifica dal ciclo while una volta terminata l'elaborazione degli eventi.

```
1   pool = thread_pool_create();
2   while true:
3       eventi = epoll_wait()
4
5       for each evento in lista_eventi:
6           if evento.fd == server_socket:
7               // Nuova connessione
8               client_fd = accept()
9               epoll_ctl_add(client_fd)
10
11          else:
12              // Dati da client esistente
13              client_fd = evento.fd
14              epoll_ctl_del(client_fd)
15              thread_pool_add_task(pool, client_fd)
```

Thread Pool Create

Per inizializzare correttamente il **Thread Pool** il server chiama la funzione `thread_pool_create`, che inizializza una struttura di tipo `thread_pool_t`, contenente:

- **Variabili di accesso alla coda**
- **Array di threads**
- **mutex per accesso alla coda**
- **condition variable**

Pseudocodice: Thread Pool Create

```
1   func thread_pool_create():
2       if MAX_THREADS && MAX_QUEUE_SIZE <= 0:
3           return NULL
4
5       pool.head = pool.tail = pool.task_count = pool.shutdown = 0
6
7       pool.threads = memory_allocation(sizeof(thread) * MAX_THREADS)
8
9       mutex_init(pool.mutex)
10      cond_init(pool.cond)
11
12      for i in range 0 to MAX_THREADS:
13          create(pool.threads[i], worker_thread, pool)
14          detach(pool.threads[i])
```

Thread Pool Worker

Ogni thread creato esegue la funzione `woker_thread`, che mette il thread in attesa di un task o di un eventuale shutdown. Se la coda non è vuota, questo recupera il file descriptor del client e chiama `parse_request` per processarlo.
Viene impiegato un mutex per proteggere l'accesso alla coda, e una condition variable per gestire il risveglio dei thread.

Pseudocodice: Thread Pool Worker

```
1   func worker_thread():
2       while true:
3           lock(pool.mutex)
4
5           while coda_vuota and not shutdwon:
6               wait(pool.cond)
7
8           if shutdown:
9               unlock(pool.mutex)
10              thread_exit
11
12
13          client_fd = estrai_dalla_coda
14          rimuovi_fd_coda()
15
16          unlock(pool.mutex)
17          parse_request(client_fd)
```

Thread Pool Add Task

Il server aggiunge il file descriptor del client effettuante una richiesta all'interno della coda del thread pool. Il server controlla la disponibilità di spazio e successivamente aggiunge il fd.

```
1   func thread_pool_add_task(pool, client_fd):
2       lock(pool.mutex)
3
4       if pool.task_count == MAX_QUEUE_SIZE:
5           unlock(pool.mutex)
6           return -1
7
8       aggiungi_fd_coda()
9
10      unlock(pool.mutex)
11      signal(pool.cond)
```

Il server utilizza il mutex presente nella struttura pool per avere accesso esclusivo alla coda e una condition variable per svegliare un thread.

Parsing Richieste

Ogni thread prima di ricevere una richiesta imposta un timeout di 5 secondi sulla connessione, solo dopo riceve i messaggi dal socket. Le richieste vengono elaborate con una `Regular Expression`, che permette di recuperare il `method` e la `request-URI` da una request line HTTP/1.1
Il server saprà quale metodo eseguire, lanciando anche eventuali risposte di errore in caso di richiesta malformata (`400 Bad Request`) o metodo non implementato (`501 Not Implemented`).
I metodi implementati sono:

- **GET**: Recupera un file e lo restituisce al client;
- **POST**: Effettua operazioni di scrittura/append su file;
- **PUT**: Effettua operazioni di scrittura su file;
- **DELETE**: Rimuove un file.

Pseudocodice: Parsing Richieste

```
1   func parse_request(client_fd):
2       set_timeout(interval=5s)
3
4       keep_alive = true //HTTP 1.1 Default
5
6       while (keep_alive):
7           received = receive_data(client_fd, buffer)
8
9           // Il client ha chiuso la connessione
10          if (buffer == "Connection:close"):
11              keep_alive = false
12
13          regex = "([A-Z]+)␣/([^␣]*)"
14          if (regexec(regex, buffer, match) == 0):
15              method = match[1]
16              file = match[2]
17
18              if (method == GET): get()
19              elif (method == POST): post()
20              elif (method == PUT): put()
21              elif (method == DELETE): delete()
22              else: not_implemented_501()
23          else: bad_request_400()
```

Decoding Credenziali

Per utilizzare i metodi **POST, PUT e DELETE**, il server richiede delle credenziali di accesso, propriamente codificate in base64 e inserite nel `header field` **Authorization**.
Queste credenziali vengono decodificate con un algoritmo implementato all'interno della funzione `base64_decode`. La funzione raccoglie gruppi di **4 caratteri da 6 bit**, che vengono mappati usando la **Base64 Character table**, e convertiti in gruppi di **3 caratteri da 8 bit** attraverso operazioni di shifting e addizione binaria.
Se la stringa non è un multiplo perfetto di 4, viene aggiunto un padding `'='` e processato come il resto della stringa.

Pseudocodice: Decoding Credenziali

```
1   func base64_decode(src, out_buffer):
2       char_array_4[4] = char_array_3[3] = new_array()
3       while (input_index < input_len AND src[input_index] != '=' AND
            is_valid_base64):
4           char_array_4[chunck_index++] = src[input_index++] //Accumula 4
                caratteri
5
6           if (chunck_index == 4):
7               for each character in char_array_4: base64_decode_table(character)
8
9               char_array_3[0] = (val_0 << 2) + ((val_1 & 0x30) >> 4)
10              char_array_3[1] = ((val_1 & 0x0F) << 4) + ((val_2 & 0x3C) >> 2)
11              char_array_3[2] = ((val_2 & 0x03) << 6) + val_3
12
13              for i in range 0 to 3: out_buffer[output_index++] = char_array_3[i
                    ]
14
15              i = 0
16
17      if (chunk_index > 0):
18          < decode_base64_with_padding()
```

Scenario di test

Lo scenario di test analizzato comprende una istanza di client che effettua richieste HTTP sulla porta 8080. Lo strumento utilizzato per effettuare i test e' cURL. Le richieste effettuate sono:

- **GET** - Multiple richieste, accesso alla cartella root;
- **POST** - Aggiunta di dati su file già esistente;
- **PUT** - Creazione di dati;
- **DELETE** - Rimozione file.

I risultati mostrano come il server rispetti le regole di autenticazione imposte e sia in grado di elaborare richieste multiple sulla stessa connessione.

Testing

Multiple richieste GET elaborate su connessione persistente:

```
curl http://localhost:8080/file.txt http://localhost:8080/file2.txt

#Output
{output file.txt}
{output file2.txt}
```

Richiesta POST senza credenziali:

```
curl -i -X POST http://localhost:8080/file.txt

#Output
[Header HTTP 401 Not Authorized]
```

Richiesta POST con credenziali:

```
curl -i --user "root:toor" -X POST http://localhost:8080/file.txt

#Output
[Header HTTP 200 OK]
{output file.txt}
```

Sviluppi Futuri

- Permettere l'elaborazione di file di medie/grandi dimensioni;
- Aggiungere un layer **SSL/TSL** per cifrare le comunicazioni (`HTTPS`);
- Includere un'**interfaccia web dinamica** per effettuare operazioni con il browser.

Conclusioni

Il server e' quindi in grado di gestire le richieste HTTP più comuni in modo funzionale e conforme a RFC 2616, garantendo connessioni persistenti per l'elaborazione di più richieste su unica connessione.