

Analisi del problema

Si vuole realizzare un'applicazione in grado di fornire le funzionalità di un semplice web server http, che sia in grado di gestire più richieste contemporaneamente, rispondendo correttamente a richieste di tipo GET su file statici.

Nella gestione delle risposte, si richiede di generare pacchetti correttamente strutturati, che abbiano un header corretto e forniscano un codice di stato coerente.

Analisi dell'implementazione

Per la realizzazione dell'applicazione, si usa il linguaggio di programmazione Python, sfruttando alcune delle librerie e delle classi che mette a disposizione.

Per prima cosa, vengono importati alcuni moduli:

- **sys:** permette di gestire alcuni aspetti di basso livello, come l'accettazione di input da riga di comando oppure l'interazione con aspetti legati principalmente all'interprete;
- **signal:** permette di definire degli handler personalizzati da eseguire quando si riceve un segnale da tastiera;
- **http.server:** modulo che definisce alcune classi per l'implementazione di server http. È un modulo di rilevanza più accademica che commerciale, per questioni di sicurezza legate a controlli troppo blandi;
- **socketserver:** fornisce alcune classi per realizzare dei server, come per esempio la classe TCPServer che crea un server basandosi sul protocollo TCP.

Si passa poi alla definizione di un handler personalizzato per il segnale con cui si vuole gestire la chiusura del server:

```
def closing_signal_handler(signal, frame):
    print('Closing server: Ctrl + C was pressed')
    try:
        if (server):
            server.server_close()
    finally:
        sys.exit(0)
```

Questa funzione servirà da handler per i segnali da tastiera, e in particolare gestirà la chiusura del server (come descritto nella stampa). Quando invocata, con un blocco try-except-else-finally – di cui però si definiscono solo i blocchi di codice relativi al try e al finally – controllerà se il server istanziato è ancora attivo, e in caso richiamerà la sua funzione di chiusura, per interrompere poi il processo in ogni caso (clausola finally). Dopodiché, con il comando `“signal.signal(signal.SIGINT, closing_signal_handler)”` si potrà associare l'handler appena definito alla gestione del segnale SIGINT (ovvero Ctrl + C).

Dopo aver definito l'handler per la chiusura, si comincia a istanziare il vero e proprio server. Per prima cosa, si vuole stabilire su che porta lavorerà il server.

```
if sys.argv[1:]:
    port = sys.argv[1]
else:
    port = 8080
```

In questo blocco di codice, si verifica se il vettore degli argomenti passati da tastiera ha più di un elemento, e in caso ne abbia almeno due si usa il secondo (quindi il primo parametro passato in input dopo il nome dell'eseguibile quando lo si lancia da terminale) come valore della porta. Altrimenti, si setta come predefinito il numero di porta 8080, registered port number solitamente usato per i web server.

Si procede poi ad istanziare il vero server e lanciarne il funzionamento. Con il comando "`server = socketserver.ThreadingTCPServer(('', port), http.server.SimpleHTTPRequestHandler)`" si istanzia nella variabile `server` il vero e proprio server http che risponderà alle richieste ricevute dal programma. Il server viene creato della classe `ThreadingTCPServer`, contenuta nel modulo `socketserver`, che consiste nella creazione di un server multithreading – ovvero in grado di gestire più richieste simultanee – associato alla porta indicata dal valore della variabile `port`.

Oltre al valore di porta, si fornisce come parametro per l'istanza la classe `SimpleHTTPRequestHandler` del modulo `http.server`. Questa classe fornisce in risposta tutti i file dalla directory da cui viene invocata e da quelle inferiori. Il lavoro è fondamentalmente svolto da due funzioni implementate dalla classe `BaseHTTPRequestHandler`:

- **do_HEAD:** gestisce le richieste di tipo HEAD, che invia le intestazioni che invierebbe per la richiesta GET equivalente;
- **do_GET:** mappa la richiesta su un file locale interpretandolo con un percorso relativo alla directory di lavoro corrente. Se altrimenti la richiesta viene mappata su una directory specifica, si svolge cercando un file di nome `index.html` oppure `index.htm`. È la stessa funzione responsabile della restituzione del codice di errore 404 quando un file non viene trovato.

Si procede poi settando alcuni parametri del server:

- `server.daemon_threads = True`: flag che indica se il programma centrale procede autonomamente rispetto ai propri thread. Il suo valore predefinito è `False`, e prescriverebbe che il processo centrale possa terminare solo quando tutti i suoi thread sono terminati. Per superare questo problema, e svincolare i vari thread, lo si setta a `True`;
- `server.allow_reuse_address = True`: flag che indica se il server riutilizzerà un indirizzo. Di default è settato a `False`, e il suo valore viene sovrascritto per permettere il riutilizzo di una porta prima che la sottostante venga rilasciata dal kernel.

Infine, si entra nel loop centrale dell'applicazione:

```

try:
    while True:
        server.serve_forever()
except KeyboardInterrupt:
    pass

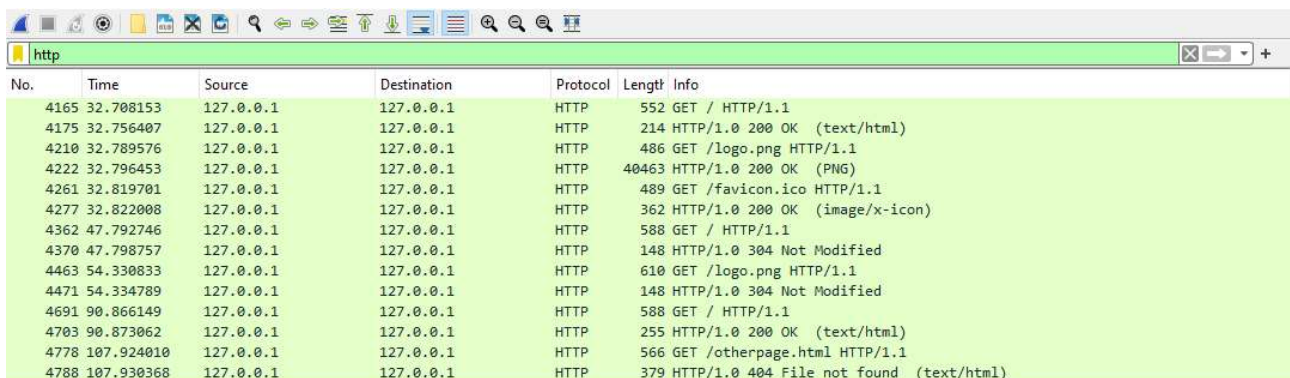
```

Anche qui si vede l'uso del costrutto try: si entra in un loop infinito in cui il server si mette in ascolto delle richieste che gli possono arrivare (controllando periodicamente se il processo viene interrotto da tastiera, grazie al valore predefinito del parametro poll_interval). La corretta esecuzione del try viene interrotta quando emerge un'eccezione di tipo KeyboardInterrupt, ovvero un'eccezione built-in di Python in grado di intercettare l'interruzione di un processo da tastiera (in questo caso causata dalla pressione di Ctrl + C). Il costrutto except serve per intercettare questa eccezione, anche se effettivamente l'istruzione pass significa che non viene svolta nessuna azione.

Al termine del programma, viene posta l'istruzione "server.server_close()", che serve a chiudere correttamente il server prima della terminazione del programma nel caso in cui il flusso di istruzioni raggiunga questo punto del codice (non avviene per esempio premendo Ctrl + C, dal momento che si entra nel signal handler che interrompe l'esecuzione e termina il programma).

Analisi dei pacchetti

Dopo aver realizzato il programma, lo si è testato e si è analizzata la struttura dei pacchetti che vengono scambiati durante la comunicazione.



No.	Time	Source	Destination	Protocol	Length	Info
4165	32.708153	127.0.0.1	127.0.0.1	HTTP	552	GET / HTTP/1.1
4175	32.756407	127.0.0.1	127.0.0.1	HTTP	214	HTTP/1.0 200 OK (text/html)
4210	32.789576	127.0.0.1	127.0.0.1	HTTP	486	GET /logo.png HTTP/1.1
4222	32.796453	127.0.0.1	127.0.0.1	HTTP	40463	HTTP/1.0 200 OK (PNG)
4261	32.819701	127.0.0.1	127.0.0.1	HTTP	489	GET /favicon.ico HTTP/1.1
4277	32.822008	127.0.0.1	127.0.0.1	HTTP	362	HTTP/1.0 200 OK (image/x-icon)
4362	47.792746	127.0.0.1	127.0.0.1	HTTP	588	GET / HTTP/1.1
4370	47.798757	127.0.0.1	127.0.0.1	HTTP	148	HTTP/1.0 304 Not Modified
4463	54.330833	127.0.0.1	127.0.0.1	HTTP	610	GET /logo.png HTTP/1.1
4471	54.334789	127.0.0.1	127.0.0.1	HTTP	148	HTTP/1.0 304 Not Modified
4691	90.866149	127.0.0.1	127.0.0.1	HTTP	588	GET / HTTP/1.1
4703	90.873062	127.0.0.1	127.0.0.1	HTTP	255	HTTP/1.0 200 OK (text/html)
4778	107.924010	127.0.0.1	127.0.0.1	HTTP	566	GET /otherpage.html HTTP/1.1
4788	107.930368	127.0.0.1	127.0.0.1	HTTP	379	HTTP/1.0 404 File not found (text/html)

Lavorando con Wireshark, si possono leggere i pacchetti che vengono scambiati sull'indirizzo di loopback, filtrandoli per tipologia. Sfruttando questa funzionalità, si analizza il comportamento del programma mentre si scambia messaggi con un browser, in questo caso Firefox. Si svolgono le seguenti operazioni:

- **richiesta della home page:** senza indicare un indirizzo, si richiede la pagina index.html (prima richiesta, GET /). La risposta che si ottiene è 200, codice di stato che denota la corretta risposta del server. A questa richiesta sono collegate la GET /logo.png, che

viene visualizzata all'interno della pagina index.html e pertanto deve essere richiesta dal browser, e la richiesta GET /favicon.ico, che viene formulata dal browser ogni volta che si connette per visualizzare un'eventuale icona del sito;

- **refresh e richieste ripetute:** quando viene svolto un refresh della pagina, o quando vengono chiesti file che sono già stati richiesti, si nota che si ottengono in risposta alcuni pacchetti con codice di stato 304. Questo codice significa che il file richiesto non è stato modificato, e pertanto è ancora valida la versione che si era già visualizzata in precedenza. Si è provato anche a refreshare la home page dopo aver modificato il file index.html – penultima richiesta – e si è ottenuto come risultato il codice di stato 200, restituito perché è stata correttamente trasmessa la nuova versione del file modificato;
- **richiesta di file inesistenti:** l'ultima richiesta, formulata per questioni di testing, prevede di richiedere al server di fornire un file inesistente, un'ipotetica *"otherpage.html"* non presente all'interno della directory, per cui si riceve codice di stato 404, errore relativo all'impossibilità del server di reperire un determinato file.