DOCUMENTAZIONE WEB SERVER SEMPLICE IN PYTHON

Nicolò Morini 0001071241 - Elaborato Programmazione di Reti

Link al repository GitHub:

https://github.com/N1c0zz/WebServerSemplice

Descrizione del contenuto del repository:

All'interno del repository è presente il codice del web server e un folder "test-client". Quest'ultimo contiene il codice di un client che testa richieste multithread su un server in host locale, oltre ad un subfolder con alcuni file di esempio da poter richiedere al server.

Come eseguire il codice del server:

Da riga di comando posizionandosi nell'apposito folder contente il file python, lanciare una stringa passando il nome del file, l'indirizzo IP sul quale vogliamo eseguirlo e il numero di porta sulla quale vogliamo porlo in ascolto (se non specificata, verrà posta la porta 8080 come default).

```
>python Py_webserver.py localhost 8080
```

Per utilizzare il codice del client di test, è necessario posizionare il file del server all'interno dello stesso folder del client, in quanto il server è implementato in modo da servire file presenti nella stessa directory o sottostanti ad essa.

Descrizione dettagliata riguardo all'implementazione del codice:

Il file "Py_webserver.py" contiene il codice che implementa il web server in grado di gestire e servire richieste HTTP per file statici in maniera concorrente (multithread).

```
import sys, signal
import http.server
import socketserver
```

Oltre ai moduli sys e signal, i quali servono rispettivamente per leggere e gestire input inviati tramite la riga di comando e per gestire particolari input (come l'arresto del server da tastiera), troviamo il modulo "http.server".
Quest'ultimo mette a disposizione una sottoclasse per la creazione e la messa in ascolto di un socket HTTP, oltre a inoltrare le richieste ad un handler, che ne identifica la tipologia e risponde in modo appropriato.

```
def check_parameters ():
    if (len(sys.argv) < 2) or (len(sys.argv) > 3):
        print (len(sys.argv))
        print("Il comando non è corretto. Usa il seguente formato: Py_webserver.py indirizzo_ip_del_server_web porta_del_server(opzionale)")
        sys.exit(0)
```

La funzione "check_parameters" impone un semplice controllo sul numero di argomenti passati a riga di comando e, nel caso di errore, stampa a video il corretto formato delle informazioni da inserire per inizializzare il server.

```
if sys.argv[2:]:
  server_port = int(sys.argv[2])
else:
  server_port = 8080
```

In questo blocco di controllo "If-else", nel caso in cui non fosse stato specificato il numero di porta da utilizzare per l'inizializzazione del server, verrà imposta la porta 8080 come valore di default.

```
server = socketserver.ThreadingTCPServer((server_ip_address, server_port), http.server.SimpleHTTPRequestHandler)
```

Qui viene inizializzato il server, utilizzando la funzione "ThreadingTCPServer".
Quest'ultima si occupa, nei sistemi Windows, di gestire la creazione dei socket relativi alle connessioni dei client, gestendo inoltre la possibilità di richieste simultanee da parte di più client (multithreading).
In input, oltre all'indirizzo IP e al numero di porta passati a riga di comando, troviamo la classe "http.server.SimpleHTTPRequestHandler".
Questa classe si occupa di trovare e fornire i file dalla directory in cui è presente il web server e dalle rispettive sotto-directory.
Inoltre la rispettiva classe base definisce i metodi "do_HEAD" e "do_GET" che si occupano rispettivamente di gestire e rispondere alla richieste HTTP di tipo HEAD e di tipo GET, fornendo i file e inviando i corretti codici di riuscita o meno dell'operazione.

```
server.daemon_threads = True
server.allow_reuse_address = True
```

L'attributo "server.daemon_threads" viene definito dalla classe ThreadingTCPServer e indica il comportamento che il server deve mantenere rispetto alla gestione dei thread.
In particolare, settando il flag come "True", specifico i thread siano gestiti in modo autonomo l'uno dall'altro, comportando che il server possa arrestarsi anche se non tutti i thread siano stati compleatati (ad esempio in caso di un malfunzionamento durante l'esecuzione di un particolare thread).
L'attributo "server.allow_reuse_address" settato come "True" invece, serve a permettere al server di riutilizzare la stessa porta che gli è stata passata a runtime, senza dover attendere che il kernel rilasci la porta sottostante.

```
def signal_handler(signal, frame):
    print( 'Exiting http server (Ctrl+C pressed)')
    try:
      if( server ):
        server.server_close()
    finally:
      sys.exit(0)

signal.signal(signal.SIGINT, signal_handler)
```

In questo blocco di codice si definisce un modo per permettere di terminare l'attività del server tramite una combinazione di tasti da tastiera. In particolare "signal.SIGINT" chiama il "signal_handler()" all'ascolto della sequenza di tasti (CTRL + C).

```
  try:
    while True:
      print("Server HTTP in ascolto su:")
      print("IP:", server_ip_address)
      print("PORT:", server_port)
      print("Per terminare il server premere il comando (Ctrl + C)")
      server.serve_forever()
  except KeyboardInterrupt:
      pass
```

Infine, oltre ad informazioni inviate in output riguardo al server che si sta eseguendo, la funzione "server.serve_forever()" gestisce tutte le richieste inviate al server e ne permette le risposte finchè quest'ultimo non termina o non viene esplicitamente interrotto (tramite ad esempio una combinazione di tasti da tastiera).