Relazione Progetto

Traccia 2 - Architettura client-server UDP per trasferimento file

Barbaro Davide
00000892880
davide.barbaro2@studio.unibo.it

Luglio 2022

1 – Introduzione

Il progetto consiste nella realizzazione in linguaggio python di un'applicazione client-server per il trasferimento di file senza bisogno di una connessione (quindi con un protocollo UDP per lo strato di trasporto).

2 – Descrizione

Per realizzare questo progetto ho creato due diversi programmi, uno per la gestione del server, l'altro per il client; bisogna avviare entrambi per il corretto funzionamento e avere installate tutte le librerie necessarie, ma non c'è un ordine preciso di avvio, ma devono essere entrambi avviati prima di poter inviare i comandi dal client:

- **Server.py**

  Eseguibile tramite terminale o tramite l'IDE di Spyder, non sono necessari elementi inseriti come argomento all'avvio.

- **Client.py**

  Eseguibile tramite terminale o tramite l'IDE di Spyder, non sono necessari elementi inseriti come argomento all'avvio.

È necessario avere *Python3* installato sul dispositivo.

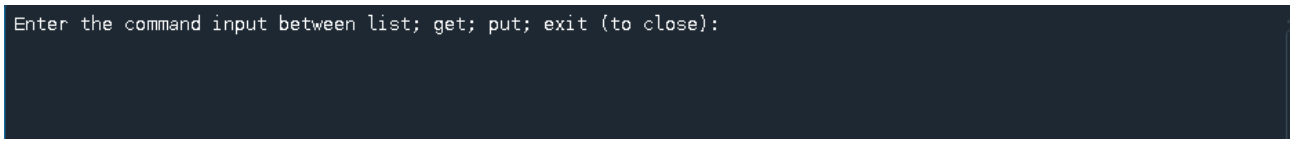
È consigliato eseguire i programmi dalle rispettive directory Server/ e Client/ dentro alla directory del progetto per agevolare la distinzione tra i file relativi al server e quelli relativi al client.

All'interno della directory di progetto ci saranno alcuni file di prova per il trasferimento: "*Test_file_client.txt*" nella directory del client per testare il caricamento sul server; mentre ci saranno due sottodirectory in quella del server: "*empty*" che è lasciata vuota per osservare cosa succede se il client richiede di vedere il contenuto di una directory vuota, "*upload*" dove il server ha già alcuni file per testare la richiesta da parte del client di essi e dove il server farà di default il caricamento dei file a lui inviati.

All'interno della directory di *upload* si troveranno di base due file: "*Test_file_small.txt*" e "*Test_file_large.txt*" che contengono il lorem ipsum (quello grande lo contiene più volte per aumentare le sue dimensioni) e che possono essere usati per testare il caricamento dal server sul client.

Una volta avviato il server rimarrà in ascolto sull'indirizzo locale 127.0.0.1 e sulla porta 10000 e si presenterà così:

Nel momento che il client viene avviato, invece, aprirà il socket e rimarrà in attesa di ulteriori comandi da parte dell'utente:



Per proseguire si deve scegliere uno dei comandi che il client può eseguire:

- **list**: richiederà di inserire un percorso che poi richiederà al server di cercare a partire dalla directory del server (se lasciato vuoto ritornerà come percorso la directory root del server).

  Quest'operazione può avere diversi esiti e il server tornerà al client i vari file che ha trovato nella directory richiesta oppure opportuni messaggi di errore.

- **get**: richiederà di inserire anche il percorso di un file (questo è obbligatorio, non accetterà il comando se il nome e percorso del file viene lasciato vuoto) che poi andrà a cercare a partire dalla directory del server.

  Anche quest'operazione può avere diversi esiti e il server comincerà a inviare i dati del file al client in datagrammi oppure un opportuno messaggio di errore se il file non è stato trovato o se sono stati riscontrati altri problemi che hanno dovuto interrompere l'esecuzione del comando.

- **put**: anche questo comando richiede il percorso di un file (obbligatorio come per get) ma che ricercherà la sua esistenza all'interno della directory del client, dando un messaggio di errore se non è stato trovato.

  Se la ricerca è andata a buon fine allora comincerà l'upload (che di default avviene nella directory "upload" del server, se non riesce a trovarla caricherà il file nella sua directory root) finché non riceve più messaggi da parte del client.

  Dopo di che invierà al client un flag che conferma l'avvenuto caricamento del file nel server. Se viene riscontrato un problema il flag verrà posto a false e il client capirà che è avvenuto un errore e lo farà presente all'utente tramite un *print*.

- **exit**: se viene digitato questo comando il client rilascerà il socket e il programma verrà terminato (la terminazione del client non implica la terminazione del server).

Client in azione con diversi comandi:

```
Enter the command input between list; get; put; exit (to close): list

Insert path here: upload
[+] Files found:
Test_file_large.txt
Test_file_small.txt

Enter the command input between list; get; put; exit (to close): get

Enter the name of the file (path divided with /): upload/Test_file_small.txt

Do you want to rename the file? [y/n]: y

Input new name: FileTestato.txt
[?] File loaded at 445[+] File downloaded successfully

Enter the command input between list; get; put; exit (to close): put

Enter the name of the file (path divided with /): FileTestato.txt
[?] File uploaded at 0[?] Upload complete
[+] File uploaded successfully

Enter the command input between list; get; put; exit (to close):
```

Mentre per gli stessi comandi da lato server:

```
Waiting to receive...
[?] Received command list from 127.0.0.1
[+] Data sent to client

Waiting to receive...
[?] Received command get from 127.0.0.1
[+] File uploaded to client successfully

Waiting to receive...
[?] Received command put from 127.0.0.1
[+] File uploaded to server successfully

Waiting to receive...
```

Queste richieste sono avvenute tutte con successo, successivamente il server ritorna in attesa di ricevere e client chiede all'utente nuovamente un comando.

Se si fosse verificato un errore sarebbe stato stampato un messaggio sia sul client che sul server.

```
Enter the command input between list; get; put; exit (to close): list

Insert path here: Notexist
[!] Error: Directory Path not Found

Enter the command input between list; get; put; exit (to close): get

Enter the name of the file (path divided with /): Filenotexist

Do you want to rename the file? [y/n]: n
[!] Error: File not Found

Enter the command input between list; get; put; exit (to close): put

Enter the name of the file (path divided with /): Filenotexist
[!] Error: File not Found

Enter the command input between list; get; put; exit (to close):
```

Lato Client

```
Waiting to receive...
[?] Received command list from 127.0.0.1
[!] Error: Directory Path not Found

Waiting to receive...
[?] Received command get from 127.0.0.1
[!] Error: File not Found

Waiting to receive...
```

Lato Server

Durante il primo comando *list* abbiamo cercato una directory che non esiste sul server e abbiamo ricevuto dal server un messaggio di errore.

Durante il secondo comando *get* abbiamo cercato un file che non esiste sul server, perciò il server ci ha detto che non l'ha trovato inviandoci un messaggio di errore.

Nell'ultimo comando *put* abbiamo sempre cercato un file che non esiste e, non avendo superato il test dal client, non è neanche passato al server, perciò ci sono solo due risultati di operazioni dal server.

Anche dopo un errore il programma non si interrompe e richiede altri comandi.

3 – Dettagli Implementativi

Sia il Client che il Server sono stati implementati in modo da collegarsi ad un IP e una Porta registrati in variabili all'inizio del programma, se si vuole modificare uno di essi basta modificare il valore di quelle variabili.

I messaggi vengono codificati e decodificati ad ogni trasferimento in formato *latin-1* per funzionare anche se i file trasferiti contengono caratteri speciali (si può provare cercando di caricare i file sorgenti) e viene usato durante il trasferimento un buffer di 4096 byte.

Ogni volta che si ha a che fare con la scrittura o lettura di un file o per la ricerca della lista di file con il comando *list*, viene usato un *try-except* in modo che il programma non si blocchi in caso di eccezioni, ma prosegua inviando in caso un messaggio di errore.

Vengono comunque effettuati vari controlli per l'esistenza di file durante il caricamento da e sul server e inviati messaggi di errore in caso i controlli falliscono.

4 – Librerie utilizzate

- **socket**: per la gestione dei socket.
- **select**: per ricevere dati durante il trasferimento sfruttando un tempo di timeout per vedere se la connessione decade.
- **os**: per la ricerca di file.
- **time**: per fornire sia al client che al server durante il trasferimento di file un tempo di attesa prima di inviare il prossimo datagramma in modo da conferire al ricevente il tempo di elaborarlo.
- **colorama**: per inserire simboli colorati ai *print* che definiscono se è un messaggio di successo, fallimento o altro.