Progetti Laboratorio Programmazione di Reti

Roberto Pisu
0001080143
roberto.pisu@studio.unibo.it

A.A. 2023/2024

Realizzazione di una chatroom condivisa tipo client-server.

Contents

1 Come avviare il programma

Semplicemente, dopo aver scaricato i file, si fa eseguire da riga di comando prima il server: 'py server.py' e successivamente, su altre schede della riga di comando i vari client: 'py client.py'.

2 Introduzione

2.1 Socket

Per poter realizzare una connessione tra client e server, dobbiamo occuparci di come conneterli a livello transport, infatti il livello transport è un livello end-to-end, ovvero non vi è alcun apparato che legge gli header dei protocolli del livello transport ma sono gli host che si occupano di gestire la connessione. Per permettere all'applicazione in python di interagire con i protocolli del livello transport, utilizziamo il socket, importando il corrispettivo modulo in pyhton. Un socket non è altro che un canale logico di comunicazione tra 2 processi, identificato con la coppia: IP-PORTA.

2.2 Scelta del protocollo di livello Transport

Per la realizzazione di una chat con 1 server e tanti client che comunicano tutti tra di loro, ho scelto come protocollo di livello Transport, il protocollo TCP(Trasmission Controll Protocol). Dato che essendo connection-oriented, predilige l'affidabilità della comunicazione piuttosto che la velocità che invece garantisce l'UDP. Infatti è meglio attendere relativamente di più e ricevere tutti i messaggi piuttosto che riceverli in meno tempo ma con la possibilità che qualcuno venga perso. Il protocollo TCP viene definito durante la creazione del socket:

```
socket_del_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

AF_INET = indica che il socket lavora con indirizzo Ipv4.
SOCK_STREAM = indica che si appoggia la protocollo TCP di livello transport.

2.3 Scelta dell'indirizzo ip e della porta

Come indirizzo ip per il server, utilizziamo quello di localhost: "127.0.0.1" che rappresenta l'indirizzo ip della macchina in cui viene eseguito il codice. Per quanto riguarda la porta per il quale il server si metterà in ascolto, utilizziamo una dynamic port, ovvero una porta compresa tra la 49152 e 65535, dato che queste ultime non necessitano di alcuna registrazione all'ente IANA.

2.4 Formato dei messaggi

Come formato dei messaggi ho scelto di usare il JSON, motlo utillizzato per lo scambio di dati client-server. Per poterlo utilizzare in pyton si importa l'apposito modulo json.

2.5 Gestione degli errrori e delle eccezioni

Gli eventuali errori ed eccezzioni che possono occorrere durante l'esecuzione sia del client che del server, sono gestite mediante vari blocchi try-except-finally.

3 Requisiti per eseguire il codice

- La versione di python utilizzata dagli script è la 3.9.1, assicurarsi di avere questa versione o superiore.
- Assicurarsi di avere installate le seguenti librerie python: socket, threading, json, sys
- Gli script funzionano e sono testati sia su sistema operativo Windows che Linux (testati in particolare su windows 11 e Ubuntu 20.04)
- Assicurarsi che la porta 60467 del proprio sistema è libera. Se non lo è occorre cambiare la porta manualmente su entrambi i file.

4 Server

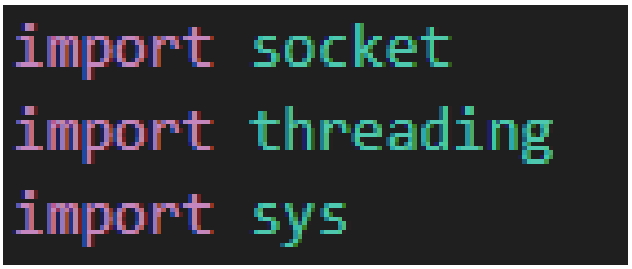
4.1 Idea per il server

Voglio realizzare un server che abbia un socket fisso in ascolto, in attesa che i client si connettino. Una volta che si connette un client, ne stampo le informazioni (nome utente + indirizzo ip) sulla command line del server. Voglio poter gestire più client contemporaneamente, perciò devo realizzare un **server concorrente**, che generi un thread separato per ogni client che si connette. Una volta che il client si è connesso, memorizzo il suo socket in una lista che memorizza tutti i client connessi. Nel thread che gestisce il client ho un ciclo infinito per controllare se mi manda dei messaggi, se mi manda un messaggio, lo inoltro a tutti i socket dei client connessi. Se un client si disconnette dal server, lo stampo a video nella command line, chiudo il suo socket e lo tolgo dalla lista dei client connessi. Inoltre devo gestire la chiusura del server con relativa chiusura del socket associato.

4.2 Implementazione del server

L'implementazione del server è stata fatta nei seguenti passi:

- **Importazione dei moduli necessari**



Gli import che ci servono lato server sono: socket per poter lavorare con i socket, threading per poter lavorare con i thread e sys per poter chiudere il programma quando serve.

- **Creazione del socket**

```
indirizzo_ip = '127.0.0.1'
porta = 60467

socket_del_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
socket_del_server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
socket_del_server.bind((indirizzo_ip, porta))
socket_del_server.listen()

connected_clients = {}

print(f'Server in ascolto sul socket {indirizzo_ip}:{porta}')
```

Definiamo l'indirizzo ip e la porta, dopodiché definisco il socket con la funzione 'socket' dell'ononimo modulo. La funzione setsockopt invece mi permette di settare il flag SO_REUSEADDR a 1, questo mi permette di poter riutilizzare lo stesso socket più volte lo stesso socket, senza quest'opzione si potrebbe incappare un un'errore di 'already used' se si riavvia rapidamente il server.
Collego il socket all'indirizzo ip e alla porta scelti tramite la funzione bind del modulo socket, e metto il socket in ascolto, in attesa di client. 'connected_list' è un dictionary che conterrà socket-nome dei client connessi.

- **Funzione per il thread demone che controlla la chiusura del server**

```
def controllo_ctrl_c():
    while 1:
        try:
            input()
        except EOFError:
            print("Chiusura del server.")
            for client_socket in connected_clients.keys():
                client_socket.close()
            socket_del_server.close()
            sys.exit(0)
```

Per poter chiudere il server dalla command line una volta avviato, dato che il server mi rimane perennemente (dato che c'è un loop infinito) in ascolto in attesa di nuovi client, ho pensato di gestire la sua eventuale chiusura da parte dell'usufruitore dello script tramite un thread che esegue in background. Questo thread controlla all'infinito se l'utente preme ctrl-c, e quando ciò accade, chiude il socket di tutti i client connessi, chiude il socket del server, e chiude il programma.

- **Funzione per il thread che gestisce il client**

```python
def gestisci_client(client_socket, nome_utente):
    try:
        while 1:
            messaggio = client_socket.recv(1024)

            for destinatario_socket, _ in connected_clients.items():
                if destinatario_socket != client_socket:
                    destinatario_socket.send(messaggio)

    except Exception as _:
        print(f'Il client {nome_utente} si è disconnesso')

    finally:
        client_socket.close()
        del connected_clients[client_socket]
```

Per gestire il client lato server, il thread che si occupa di ciò, attraverso un loop, va a leggere gli eventuali messaggi ricevuti da quel client, e per ogni socket dei client connessi, viene inoltrato il messaggio ricevuto dal server. Quindi il server fungerà da tramite, riceve un messaggio da un client, e lo inoltra agli altri client destinatari. Quando un client si disconnette, viene chiuso il suo socket e viene tolto tra la lista dei client connessi al server.

- **Avvio del thread demone**

```python
ctrl_c_thread = threading.Thread(target=controllo_ctrl_c)
ctrl_c_thread.daemon = 1
ctrl_c_thread.start()
```

Faccio partire il thread deamon, ovvero il thread che opererà in background, per impostare il fatto che deve eseguire in background, vado a impostare a 1 il flag deamon.

- **Loop per gestire i client**

```python
while 1:
    client_socket, client_address = socket_del_server.accept()
    nome_utente = client_socket.recv(1024).decode()
    connected_clients[client_socket] = nome_utente
    print(f'Nuovo utente connesso {client_address}, nome: {nome_utente}')
    client_thread = threading.Thread(target=gestisci_client, args=(client_socket, nome_utente))
    client_thread.start()
```

Ciclo infinito che accetta le richieste di connessione dei vari client, quando si connette un client, inserisco il suo socket e il suo nome utente nel dizionario 'connected_clients' e stampo una stringa esplicativa per far vedere chi si è connesso. Dichiaro poi un thread che ha come corpo la funzione 'gestisci_client' e lo avvio con '.start()'.

5 Client

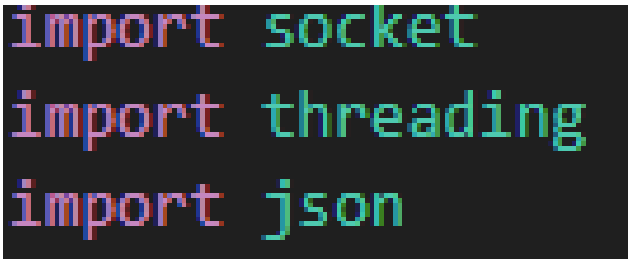
5.1 Idea per il client

L'idea di base per il client è molto semplice, all'avvio, richiedo il nome con il quale vuole essere riconosciuto all'interno della chat. Dopodiché tento di connettermi al server, e una volta connesso, mando il mio nome al server che lo stamperà nella sua shell. Una volta mandato il nome, faccio partire 2 thread separati per gestire in maniera autonoma l'arrivo e l'invio di messaggi. Nel thread che gestisce la ricezione, con un ciclo infinito leggo i dati che sono stati mandati sul socket, e li stampo. Mentre il thread che gestisce l'invio di messaggi, aspetta in input il messaggio, e lo manda al al server che poi si occuperà di inoltrarlo a tutti i client connessi.

5.2 Implementazione del client

L'implementazione del client è stata fatta nei seguenti passi:

- **Importazione dei moduli necessari**



Gli import che ci servono lato server sono: socket per poter lavorare con i socket, threading per poter lavorare con i thread e json per poter mandare i messaggi con il formato json che risulta più leggero rispetto ad altri formati.

- **Inizializziazio ip e porta**

```
indirizzo_ip = '127.0.0.1'
porta = 60467
```

Imposto l'indirizzo ip e la porta del server.

- **Funzione per il thread che gestisce il ricevimento di messaggi**

```
def ricevi_messaggi(socket_del_client):
    try:
        while 1:
            messaggio = socket_del_client.recv(1024)

            messaggio_decodificato = json.loads(messaggio.decode())
            print(f'{messaggio_decodificato["mittente"]} > {messaggio_decodificato["contenuto"]}')

    except Exception as e:
        print('Errore nella ricezione dei messaggi:', str(e))

    finally:
        socket_del_client.close()
```

All'interno di questa funzione, con un loop infinito vado a leggere i messaggi che arrivano, mediante la funzione 'recv'. Poi dopo aver decodificato il messaggio dal json, tramite la funzione 'json.loads', lo stampo, con annesso anche il mittente di tale messaggio.

- **Funzione per il thread che gestisce l'invio di messaggi**

```python
def invia_messaggi(socket_del_client):
    try:
        while 1:
            messaggio = input("-> ")
            if messaggio:
                messaggio_da_inviare = json.dumps({"mittente": mio_nome, "contenuto": messaggio})
                socket_del_client.send(messaggio_da_inviare.encode())

    except Exception as e:
        print('Errore nell\'invio dei messaggi:', str(e))

    finally:
        socket_del_client.close()
```

In questa funzione invece, sempre con un loop, attendo l'input del client, e, se l'input c'è ('if messaggio:' controlla se il messaggio non è vuoto), converto il messaggio in json, componendolo di mittente e di contenuto, tramite la funzione 'json.dumps'. Infine, mando tutto al server tramite la funzione 'send' del modulo socket.

- **Creazione socket del client e connessione al server**

```python
mio_nome = input("Nome: ")

socket_del_client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
socket_del_client.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
socket_del_client.connect((indirizzo_ip, porta))
```

Nel 'main' dello script, creo il socket del client, metto gli stessi settaggi utili per riutilizzare il socket fatti in precedenza per il server, e uso la funzione 'connect' che tenta di connetterlo al server. La funzione 'connect' è bloccante, per cui il client rimane in attesa fino a che il server non accetta la sua connessione.

- **Avvio dei thread per l'inivio e la ricezione di messaggi**

```python
# Invio del nome utente al server
socket_del_client.send(mio_nome.encode())

# Avvio dei thread per la ricezione e l'invio dei messaggi
thread_ricevi_messaggi = threading.Thread(target=ricevi_messaggi, args=(socket_del_client,))
thread_invia_messaggi = threading.Thread(target=invia_messaggi, args=(socket_del_client,))
thread_ricevi_messaggi.start()
thread_invia_messaggi.start()
```

Instanzio i thread che gestiscono la ricezione e l'invio dei messaggi, gli associo le relative funzioni con i relativi argomenti e dopodiché li faccio partire con la funzione '.start' del modulo threading.

6 Codice in esecuzione

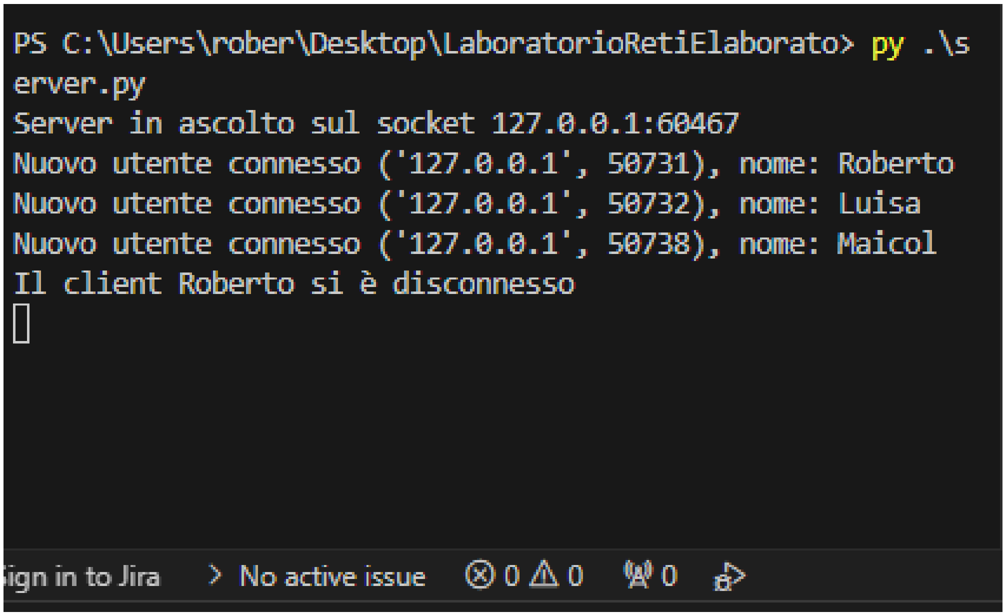


Figure 1: Qui vediamo il server in uso con 3 client inizialmente connessi e 1 di questi si è poi disconnesso

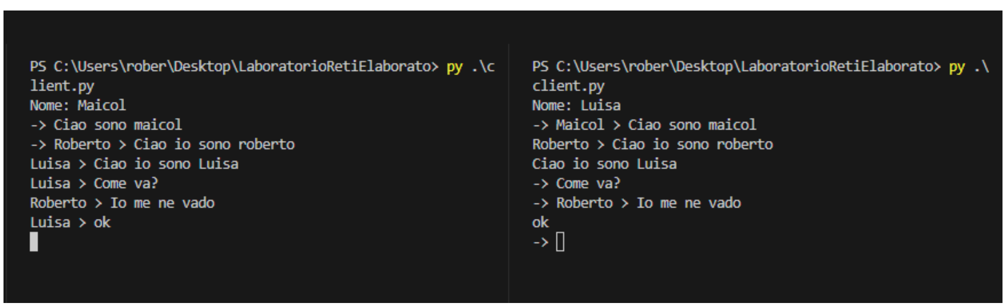


Figure 2: Qui vediamo i client comunicare tra loro, da notare che ci sono anche i messaggi di 'Roberto' che ha scritto prima che si disconnettesse

Nota[1]

[1] Per realizzare i loop infiniti ho utilizzato 'while 1' invece che 'while True' perchè leggermente più veloce.