ChatRoom
Elaborato per
Programmazione di reti
Traccia 1

Giovanni Tentelli

0000921144

giovanni.tentelli@studio.unibo.it

Introduzione

ChatRoom è un elaborato consistente di due script python denominati chatroom_server.py e chatroom_client.py che implementano al meglio i requisiti imposti dalla traccia.
Sia l'implementazione del server che del client non utilizzano nessuna interfaccia grafica e necessitano solo l'uso di una console.

Requisiti

Tutte le funzionalità sono già implementate in Python 3, le applicazioni richiedono solamente il lancio da una console.
Eventuali istruzioni verranno presentati durante l'esecuzione delle applicazioni

Requisiti traccia e funzionamento

Tratto dalla traccia, si chiede di implementare un servizio di chat online, mettendo a disposizione un luogo dove due o più utenti possano scambiare pubblicamente informazioni in contemporanea.
Tale servizio va implementato interamente in linguaggio Python e richiede il modulo socket.

Il Server

I requisiti per l'implementazione del server sono:

- deve accettare e mantenere più connessioni utenti in contemporanea (critico)
- deve poter ricevere messaggi dagli utenti e inviarli al resto dei utenti connessi (critico)
- deve poter rilevare se un utente non è più connesso e agire di conseguenza
- deve poter chiudere la connessione quando un utente esce dalla chat
- deve poter chiudere il servizio correttamente

Avvio e accettazione dei clients

Il server largo uso di thread per implementazione del servizio.
All'inizio lo script chiama main() che si occupa di preparate tutte le variabili, specialmente la seguente parte di script:

```
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server.settimeout(None)
try:
   server.bind((HOST, PORT))
   server.listen()
except socket.error as e:
   print("during server initialization a error has occurred :")
   print("{}".format(e))
   server.close()
   exit(1)

restarter_thread = Thread(target=connection_restarter, args=(server, lock))
restarter_thread.daemon = True
restarter_thread.start()
```

```
def connection_restarter(server, lock):
   while not EXITING:
       acceptor_thread = Thread(target=connection_acceptor, args=(server,
lock))
       acceptor_thread.daemon = True
       acceptor_thread.start()
       acceptor_thread.join()
```

L'ultima riga è la più importante poiché lancia il thread che si assicurerà che il server sarà sempre disponibile ad accettare connessioni fino alla chiusura dello script.
Il resto del main() tratta la chiusura corretta dello script.

Il connection_acceptor() is occupa esclusivamente di accettare nuove connessioni da parte di un client e di legare a un identificatore utente al suddetto client (il quale è temporaneo e scade in contemporanea con la connessione del client, incluso qualunque username il client scelga dopo aver stabilito la connessione).
Successivamente si crea un thread a se stante:

```
Thread(target=client_manager, args=(client, lock)).start()
```

che si occuperà di servire quel specifico client.

```
def connection_acceptor(server, lock):
   print("Waiting for clients...\n")
   client = None
   while not EXITING:
       try:
           client, cl_address = server.accept()
           client.settimeout(30)
           if client is not None:
               print("%s:%s is now connected.\n" % cl_address)
               lock.acquire()
               add_client(client, cl_address)
               lock.release()
               client_thread = Thread(target=client_manager, args=(client,
lock))
       client_thread.daemon = True
       client_thread.start()

       except socket.timeout:
           return None
       except socket.error as e:
           print("Accepting socket error:{}".format(e))
           if client is not None:
               lock.acquire()
               remove_client(client)
               lock.release()
           return None
```

Gestione del client

Il client manager è una funzione molto complessa che gestisce le azioni del cliente, quali la disconnessione, uscita dalla chat, mantenimento della connessione e l'invio di messaggi a tutti gli utenti; ciò che non può fare è ristabilire la connessione e il riconoscimento di utenti che si sono precedentemente disconnessi.
il client manager può essere diviso in 2 blocchi:

- accettazione del nome utente e annuncio dell'entrata in chat dell'utente:

```
username = None
client.send(
   bytes("[SERVER]: Hi! Digit here your name (must be at least 2 characters
long), then press enter!",
         "utf8"))
username = client.recv(BUFSIZ).decode("utf8").strip()
while len(username) < 2 or username in USERNAMES.values() or username ==
QUIT:
   if not len(username) or username == QUIT:
       raise Exception("Client has closed the connection\n")
   elif username == PING:
       client.send(bytes(PING, "utf8"))
   else:
```

```
        client.send(
            bytes("[SERVER]: Name is already taken or smaller than 2
caracthers, please digit another name.",
                  "utf8"))
    username = client.recv(BUFSIZ).decode("utf8").strip()

lock.acquire()
add_username(client, username)
print("user %s:%s " % CLIENTS.get(client), "has chosen username %s\n" %
username)
lock.release()

msg = "[SERVER]: Welcome %s! To leave chat write !quit." % username
client.send(bytes(msg, "utf8"))
data = "%s joined the chat!" % username

lock.acquire()
send_all_clients(bytes(data, "utf8"), "[SERVER]")
lock.release()
```

- loop per il ricevimento dei messaggi utente e distribuzione al resto dei utenti in chat:

```
while True:
    data = client.recv(BUFSIZ).decode("utf8")
    if data == QUIT:
        lock.acquire()
        print("Client %s:%s," % CLIENTS.get(client),
              "with username %s is being removed from userlist. Reason:
Quit\n" % USERNAMES.get(client))
        send_all_clients(bytes("%s has quit." % username, "utf8"), "[SERVER]")
        remove_client(client)
        lock.release()
        client.close()
        return None
    elif data == PING:
        client.send(bytes(PING, "utf8"))
    elif len(data) and data != PING:
        lock.acquire()
        send_all_clients(bytes(data, "utf8"), username)
        lock.release()
    else:
        raise socket.error
```

- gestione delle eccezioni, tali disconnessione e uscita dell'utente dalla chat:

```
except socket.error as e:
    # if any case of socket error always remove client from userlist and
terminate thread
    lock.acquire()
```

```
   print("User %s:%s has recurred a error: {}\n".format(e) %
CLIENTS.get(client))
   if client in USERNAMES.keys():
       print("Client %s:%s," % CLIENTS.get(client),
             "with username %s is being removed from user list. Reason:
Disconnected\n" % USERNAMES.get(client))
       send_all_clients(bytes("%s has disconnected." % USERNAMES.get(client),
"utf8"), "[SERVER]")
   else:
       print("Client %s:%s," % CLIENTS.get(client), "is being removed from
user list. Reason: Disconnected\n")
   remove_client(client)
   lock.release()
   client.close()
   return None
except Exception as e:
   print("A Client error as occurred: {}\n".format(e))
   print("Client %s:%s" % CLIENTS.get(client), "is being removed from user
list. Reason: Disconnected\n")
   lock.acquire()
   remove_client(client)
   lock.release()
   client.close()
   return None
```

Note sul server

- l'applicazione del server non ha implementazioni servizi per la moderazione della chat.
- L'indirizzo sul quale il server fa bind è esplicitato all'interno dello script:

```
HOST = ''
PORT = 53000
```

Il Client

I requisiti del client sono:

- deve poter connettersi e mantenere la connessione con il server (critico)
- deve poter ricevere e inviare messaggi al server (critico)
- deve poter rilevare se non è più connesso e agire di conseguenza
- deve poter comunicare al server quando esce dalla chat e uscire correttamente dal processo

Avvio del Client

All'inizio il main() inizializza le variabili per il controllo dell'applicazione; una volta fatto, si entra in un loop indefinito e la ragione per questo loop è che, per qualsiasi ragione la connessione cade, si ritenti sempre di riaprire la connessione ogni volta che scattano i check di controllo del riavvio.

```
def main():

   #codice per l'inizializzazione

   while True:
       client = connect(ADDR)
       print("Connection successful!\n")
       if client is not None:
           maintainer_thread = Thread(target=connection_maintainer,
args=(client, lock, EXIT, RESTART_CONNECTION))
           maintainer_thread.start()
           receiver_thread = Thread(target=receiver, args=(client, lock,
EXIT, RESTART_CONNECTION))
           receiver_thread.start()
           sender_thread = Thread(target=sender, args=(client, lock, EXIT,
RESTART_CONNECTION))
           sender_thread.daemon = True
           sender_thread.start()

           while True:
               time.sleep(5)

       #codice per il controllo di applicazione
```

Una delle parti più interessanti del codice client è la funzione connect(), lla quale prende la tupla (ip, port) e tenta per un numero limitato di tentativi di connettersi al server per poi ritornare il socket se l'evento è andato a buon fine; altrimenti chiama la chiusura dell'applicazione.

```
def connect(address):
   attempts = 0
   while attempts < RECONNECT_ATTEMPTS:
       try:
           attempts = attempts + 1
```

```
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.connect(address)
            s.settimeout(15)
            return s
        except socket.error as e:
            print("socket error: {}, reconnecting".format(e))
            print('attempt n.', attempts)
            time.sleep(5)
    print("connection attempts failed, shutting down")
    sys.exit(1)
```

Il loop e il comportamento normale

Durante la normale esecuzione dello script (ossia che la connessione è andata a buon fine) il main() chiama tre threads, ognuno con un specifico scopo:

- maintainer_thread: per mantenimento della connessione lato server.
- receiver_thread: per il ricevimento dati dal server e il mantenimento della connessione lato client.
- sender_thread: per mandare i messaggi utente al server.

```
            maintainer_thread = Thread(target=connection_maintainer,
args=(client, lock, EXIT, RESTART_CONNECTION))
            receiver_thread = Thread(target=receiver, args=(client, lock,
EXIT, RESTART_CONNECTION))
            sender_thread = Thread(target=sender, args=(client, lock, EXIT,
RESTART_CONNECTION))
```

Il connection_mantainer() è una semplice funzione che si limita a mandare a intervalli costanti una speciale stringa per mantenere la connessione dal lato server, e questo é quanto.

```
def connection_maintainer(client, lock, EXIT, RESTART_CONNECTION):
    while True:
        if EXIT.is_set() or RESTART_CONNECTION.is_set():
            lock.acquire()
            client.close()
            lock.release()
            return None
        time.sleep(5)
        try:
            client.send(bytes(PING, "utf8"))

        except socket.error as e:
            print("connection maintainer error: {}".format(e))
            lock.acquire()
            RESTART_CONNECTION.set()
            client.close()
            lock.release()
            return None
```

La funzione receiver() si limita a ricevere qualsiasi messaggio che arriva dal server, incluso il ricevimento e gestione di una speciale stringa per il mantenimento della connessione lato client, e ha una struttura di controllo per le eccezioni.

```
def receiver(client, lock, EXIT, RESTART_CONNECTION):
   while True:
       if EXIT.is_set() or RESTART_CONNECTION.is_set():
           lock.acquire()
           client.close()
           lock.release()
           return None
       try:
           msg = client.recv(BUFSIZ).decode("utf8")
           if msg == PING:
               pass
           elif len(msg):
               print(msg)
           else:
               raise socket.error
       except socket.error as e:
           print("{}".format(e))
           lock.acquire()
           RESTART_CONNECTION.set()
           client.close()
           lock.release()
           return None
       except Exception as e:
           print("{}".format(e))
           lock.acquire()
           EXIT.set()
           client.close()
           lock.release()
           return None
```

La funzione sender() si incarica di mandare al server qualsiasi cosa il cliente scriva e ha anch'esso una struttura di controllo per le eccezioni.

```
def sender(client, lock, EXIT, RESTART_CONNECTION):
   while True:
       try:
           if EXIT.is_set() or RESTART_CONNECTION.is_set():
               lock.acquire()
               client.close()
               lock.release()
               return None
           msg = input().strip()
           if len(msg) and msg != PING:
               client.send(bytes(msg, "utf8"))
```

```
            if msg == QUIT:
                print("Quitting application!")
                lock.acquire()
                EXIT.set()
                client.close()
                lock.release()
                return None
            msg = ''

        except socket.error as e:
            print("Could not send message!")
            print("Could not send message! Error {} occurred".format(e))
            lock.acquire()
            RESTART_CONNECTION.set()
            client.close()
            lock.release()
            return None
        except Exception as e:
            print("{}".format(e))
            lock.acquire()
            EXIT.set()
            client.close()
            lock.release()
            return None
```

Note sul client

- L'applicazione può essere solo terminata tramite la funzione connect() dopo x tentativi falliti o quando si stabilita la connessione (in qualunque momento) si utilizzi il comando '!quit'