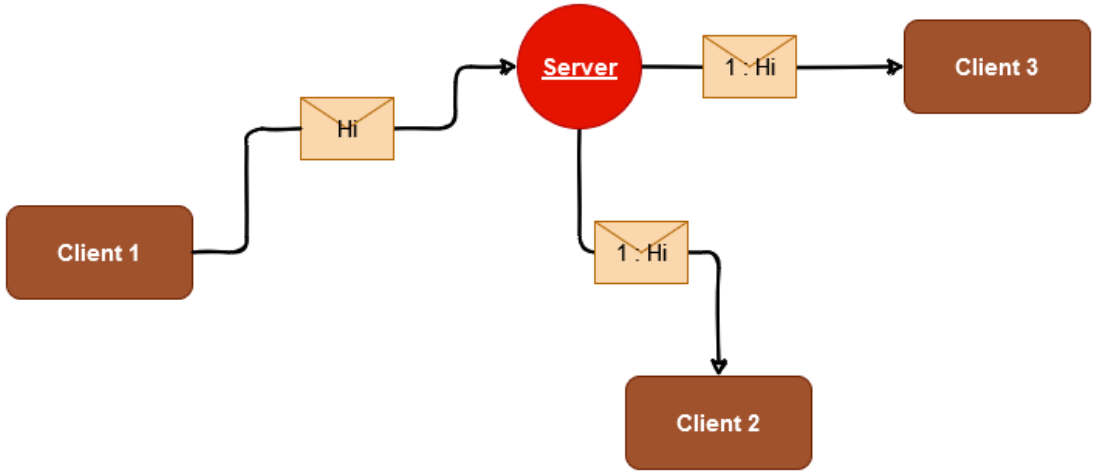
CHATROOM CLIENT-SERVER

Elaborato Programmazione di Reti

ANALISI

- Realizzazione di una chatroom condivisa client-server in Python utilizzando socket programming.
- Il server deve essere in grado di gestire più client contemporaneamente e deve consentire agli utenti di inviare e ricevere messaggi in una chatroom condivisa. Il client deve consentire agli utenti di connettersi al server, inviare messaggi alla chatroom e ricevere messaggi dagli altri utenti.



IMPLEMENTAZIONE – LATO SERVER

Il server una volta eseguito si mette in ascolto su una socket TCP in attesa di connessioni del client.

Per realizzare il server, come anche il client, ho utilizzato socket TCP e non UDP perché più adatte in questo caso

```
clients = {}
addresses = {}

HOST = '127.0.0.1'
PORT = 53000
BUFSIZ = 1024
ADDR = (HOST, PORT)

SERVER = socket(AF_INET, SOCK_STREAM)
SERVER.bind(ADDR)

if __name__ == "__main__":
    try:
        SERVER.listen(5)
        print ("Server in ascolto in attesa di connessioni: ")
        ACCEPT_THREAD = Thread(target=handle_connections)
        ACCEPT_THREAD.start()
        ACCEPT_THREAD.join()
        SERVER.close()
    except:
        print ("Errore nell avvio del server")
        exit()
```

- Quando un client si connette gli viene richiesto il nome con cui verrà identificato nella chat. Succesivamente verrà comunicato tramite broadcast agli altri utenti della chat che un nuovo client si è connesso.
- Quindi il client verrà gestito tramite thread permettendo così al server di gestire più client contemporaneamente.
- Inoltre sono stati inseriti controlli di errore nel caso il client uscisse o si disconnettese prima di utenticarsi

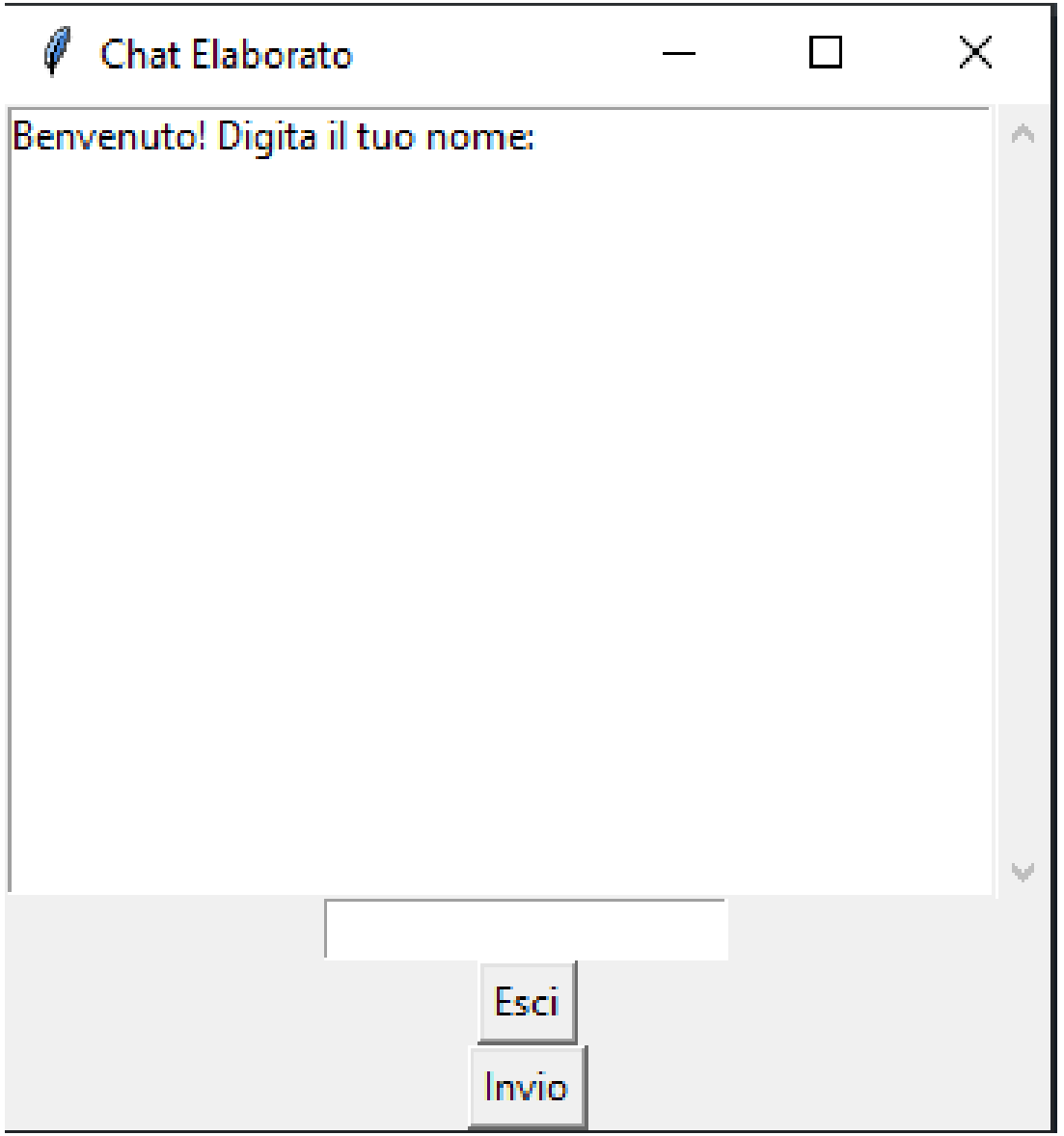
```python
#funzione per accettare la connessione del client
def handle_connections():
    while True:
        try:
            client_socket, client_address = SERVER.accept()
            print("%s:%s si è collegato." % client_address)
            client_socket.send(bytes("Benvenuto! Digita il tuo nome: ", "utf8"))
            try:
                name=client_socket.recv(BUFSIZ).decode("utf-8")
                addresses[client_socket]= client_address
                clients[client_socket]=name
                msg = "%s si è unito all chat!" % name
                broadcast(bytes(msg, "utf8"))
                Thread(target=handle_clients, args=(client_socket,name)).start()
            except:
                print("Errore nella connessione del client")
        except:
            print("Errore durante l'accettazione della connessione del client")
```

- Il server sarà poi in grado di gestire i vari client ed inviare in broadcast i messaggi inviati da un client.
- Inoltre sarà in grado di gestire la perdita di connessione da parte di un client rilevando l'errore ed eliminando il client

```python
#Funzione che gestisce la connessione dei client al server
def handle_clients (client_socket, name):
    while True:
        try:
            msg = client_socket.recv(BUFSIZ)
            broadcast(msg, name+": ")
        except ConnectionResetError:
            print("Client disconnesso")
            del clients[client_socket]
            broadcast(bytes("%s ha abbandonato la Chat." % name, "utf8"))
            client_socket.close()
            break
```

IMPLEMENTAZIONE - LATO CLIENT

- Il client è stato progettato utilizzando una GUI tkinter per rendere più intuitivo e semplice l'utilizzo della chat.
- All'apertura la chat si presenterà così:



- All’avvio del client verranno richiesti il server host e la porta del client che permetteranno al client di connettersi al server.
- Dopo essersi collegati al server si aprirà l’interfaccia grafica che permetterà al client di utentificarsi all’interno della chat come prima cosa e che poi consentirà l’invio di messaggi.

```
HOST = input('Inserire il Server host: ')
if not HOST:
    HOST='127.0.0.1'
if HOST != '127.0.0.1':
    print("Non esiste questo server host!")
PORT = input('Inserire la porta del server host: ')
if not PORT:
    PORT = 53000
else:
    PORT = int(PORT)

BUFSIZ = 1024
ADDR = (HOST, PORT)

client_socket = socket(AF_INET, SOCK_STREAM)
client_socket.connect(ADDR)


receive_thread = Thread(target=receive_msg)
receive_thread.start()

tkt.mainloop()
```

- Una volta identificati il client potrà mandare messaggi e mettersi in attesa di ricevere messaggi.
- Nella funzione receive_msg() che viene passata come argomento del thread del client viene aggiunto un controllo d'errore per la connessione del client. Infatti in caso di errore è probabile che il client abbia abbandonato la chat
- L'utente potrà anche disconnettersi dalla chat tramite il pulsante esci e sia il client che il server gestiranno questa situazione. Inoltre il server notificherà a tutti gli altri client connessi l'abbandono dalla chat del relativo client

```python
#funzione per la ricezione dei messaggi e controllo connessione client
def receive_msg():
    while True:
        try:
            msg=client_socket.recv(BUFSIZ).decode("utf-8")
            msg_list.insert(tkt.END, msg)
        except OSError:
            print("Client disconnesso dal server")
            break

#funzione per inviare messaggi dal client
def send_msg (event=None):
    msg = my_msg.get()
    my_msg.set("")
    client_socket.send(bytes(msg,"utf-8"))
```

Esecuzione

Il codice potrà essere eseguito sia da linea di comando come script python sia tramite un editor come abbiamo visto spyder utilizzato a lezione.

Ovviamente per un corretto funzionamento dovremmo eseguire prima il server che così sarà in grado di accettare connessioni in entrata e poi eseguire il client che quindi potrà correttamente connettersi al server in ascolto

Considerazioni Finali

- Il servizio di chat client-server realizzato offre funzionalità di base per lo scambio di messaggi che potrebbero essere ampliate e modificate introducendo nuove funzionalità più avanzate