Relazione sul Progetto Client-Server in Python

Lucio Baiocchi

June 12, 2024

1 Introduzione

Il progetto si propone di creare un sistema di chat room client-server utilizzando la programmazione con i socket in Python. Il server è progettato per gestire più client contemporaneamente, consentendo agli utenti di inviare e ricevere messaggi in una chatroom condivisa. Il client permette agli utenti di connettersi al server, inviare messaggi e ricevere messaggi dagli altri utenti in tempo reale.

2 Implementazione

L'implementazione è suddivisa in due componenti principali: il server e il client. Entrambi sono implementati utilizzando la libreria `socket` di Python per la comunicazione di rete e la libreria `threading` per gestire più connessioni simultaneamente.

2.1 Server

Il server è responsabile di accettare le connessioni dai client, ricevere i messaggi e trasmetterli a tutti i client connessi. Il server utilizza una lista per mantenere traccia dei client e dei loro nickname. Il codice del server è mostrato di seguito:

```
import socket
import threading

clients = []
nicknames = []

def broadcast(message, _client):
    for client in clients:
```

```
9           if client != _client:
10              try:
11                  client.send(message)
12              except:
13                  clients.remove(client)
14
15  def handle(client):
16      while True:
17          try:
18              message = client.recv(1024)
19              broadcast(message, client)
20          except:
21              index = clients.index(client)
22              clients.remove(client)
23              client.close()
24              nickname = nicknames[index]
25              broadcast(f'{nickname} has left the chat!'.encode('
        utf-8'), client)
26              nicknames.remove(nickname)
27              break
28
29  def receive():
30      while True:
31          client, address = server.accept()
32          print(f"Connected with {str(address)}")
33
34          client.send('NICKNAME'.encode('utf-8'))
35          nickname = client.recv(1024).decode('utf-8')
36          nicknames.append(nickname)
37          clients.append(client)
38
39          print(f"Nickname of the client is {nickname}!")
40          broadcast(f"{nickname} joined the chat!".encode('utf-8')
        , client)
41          client.send('Connected to the server!'.encode('utf-8'))
42
43          thread = threading.Thread(target=handle, args=(client,))
44          thread.start()
45
46  server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
47  server.bind(('127.0.0.1', 5555))
48  server.listen()
49
50  print("Server is listening...")
51  receive()
```

Listing 1: Codice del Server

2.2 Client

Il client consente agli utenti di connettersi al server, inviare messaggi e ricevere messaggi dagli altri utenti. Utilizza una GUI (Graphical User Interface) per una migliore interazione con l'utente. Il client richiede un nickname all'utente e lo utilizza per identificarsi nella chat. Il codice del client è mostrato di seguito:

```
1  import socket
2  import threading
3  import tkinter as tk
4  from tkinter import simpledialog, scrolledtext
5
6  class ChatClient:
7      def __init__(self, master):
8          self.master = master
9          self.master.title("Chat Room")
10
11         self.chat_area = scrolledtext.ScrolledText(master)
12         self.chat_area.pack(padx=20, pady=5)
13         self.chat_area.config(state='disabled')
14
15         self.msg_entry = tk.Entry(master, width=50)
16         self.msg_entry.pack(padx=20, pady=5)
17         self.msg_entry.bind("<Return>", self.write)
18
19         self.send_button = tk.Button(master, text="Send",
     command=self.write)
20         self.send_button.pack(padx=20, pady=5)
21
22         self.nickname = simpledialog.askstring("Nickname", "
     Please choose a nickname", parent=master)
23
24         self.client = socket.socket(socket.AF_INET, socket.
     SOCK_STREAM)
25         self.client.connect(('127.0.0.1', 5555))
26
27         self.receive_thread = threading.Thread(target=self.
     receive)
28         self.receive_thread.start()
29
30     def receive(self):
31         while True:
32             try:
33                 message = self.client.recv(1024).decode('utf-8')
34                 if message == 'NICKNAME':
35                     self.client.send(self.nickname.encode('utf-8
     '))
36                 else:
```

```
37                      self.chat_area.config(state='normal')
38                      self.chat_area.insert('end', message + '\n')
39                      self.chat_area.yview('end')
40                      self.chat_area.config(state='disabled')
41              except:
42                  print("An error occurred!")
43                  self.client.close()
44                  break
45
46      def write(self, event=None):
47          message = f'{self.nickname}: {self.msg_entry.get()}'
48          self.client.send(message.encode('utf-8'))
49          self.msg_entry.delete(0, 'end')
50
51  if __name__ == "__main__":
52      root = tk.Tk()
53      client = ChatClient(root)
54      root.mainloop()
```

Listing 2: Codice del Client

3 Ottimizzazioni

Per migliorare la velocità di invio e ricezione dei messaggi, sono state apportate diverse ottimizzazioni:

- **Threading**: Entrambi, client e server, utilizzano il threading per gestire l'invio e la ricezione dei messaggi, assicurando che l'interfaccia utente non si blocchi e che il server possa gestire più client contemporaneamente.
- **Buffering**: Utilizziamo il metodo `recv` con un buffer di 1024 byte per ridurre le chiamate di rete e migliorare l'efficienza della trasmissione dei dati.
- **Gestione delle eccezioni**: La gestione delle eccezioni è stata migliorata per gestire meglio le disconnessioni e altri errori, assicurando che il sistema possa continuare a funzionare anche in caso di problemi di connessione.
- **Architettura scalabile**: Il server è progettato per gestire più client contemporaneamente, trasmettendo i messaggi a tutti i client connessi, migliorando la scalabilità del sistema.

4 Conclusioni

Il progetto ha dimostrato come creare un sistema di chat room client-server utilizzando Python e socket. Ho esplorato l'uso di threading per gestire più connessioni contemporaneamente e implementato diverse ottimizzazioni per migliorare la velocità di invio e ricezione dei messaggi. Questo progetto può essere ulteriormente esteso con funzionalità aggiuntive come autenticazione degli utenti, crittografia dei messaggi e un'interfaccia utente migliorata.