Relazione sul Progetto Client-Server in Python

Lucio Baiocchi

June 13, 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. All'interno del server viene utilizzata una lista per mantenere traccia dei client e dei loro nickname. Il codice del server è mostrato di seguito:

```
import socket
import threading

# Lista dei client connessi
clients = []
aliases = []

# Funzione per gestire i messaggi ricevuti da un client
def handle_client(client):
    while True:
        try:
            message = client.recv(1024)
            broadcast(message, client)
        except:
            index = clients.index(client)
            clients.remove(client)
```

```
            client.close()
            alias = aliases[index]
            broadcast(f'{alias} has left the chat!'.encode('utf-8'), client)
            aliases.remove(alias)
            break

# Funzione per inviare un messaggio a tutti i client
def broadcast(message, client):
    for c in clients:
        if c != client:
            c.send(message)

# Funzione principale per iniziare il server
def main():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(('127.0.0.1', 8080))
    server.listen()

    print('Server is listening...')

    while True:
        client, address = server.accept()
        print(f'Connection from {address}')

        client.send('NICK'.encode('utf-8'))
        alias = client.recv(1024).decode('utf-8')
        aliases.append(alias)
        clients.append(client)

        print(f'Nickname of the client is {alias}')
        broadcast(f'{alias} has joined the chat!'.encode('utf-8'), client)
        client.send('You are now connected!'.encode('utf-8'))

        thread = threading.Thread(target=handle_client, args=(client,))
        thread.start()

if __name__ == "__main__":
    main()
```

2.2 Client

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

```
import socket
import threading
import tkinter
import tkinter.scrolledtext
from tkinter import simpledialog

class ChatClient:
```

```
    def __init__(self, host, port):
        self.client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.client.connect((host, port))

        self.root = tkinter.Tk()
        self.root.title("Chat Room")

        self.chat_box = tkinter.scrolledtext.ScrolledText(self.root)
        self.chat_box.pack(padx=20, pady=5)
        self.chat_box.config(state=tkinter.DISABLED)

        self.input_area = tkinter.Text(self.root, height=3)
        self.input_area.pack(padx=20, pady=5)

        self.send_button = tkinter.Button(self.root, text="Send", command=self.send_message)
        self.send_button.pack(padx=20, pady=5)

        self.alias = simpledialog.askstring("Nickname", "Please choose a nickname", parent=self.root)
        if self.alias:
            self.client.send(self.alias.encode('utf-8'))

        self.receive_thread = threading.Thread(target=self.receive_messages)
        self.receive_thread.start()

        self.root.protocol("WM_DELETE_WINDOW", self.on_closing)
        self.root.mainloop()

    def receive_messages(self):
        while True:
            try:
                message = self.client.recv(1024).decode('utf-8')
                if message == 'NICK':
                    self.client.send(self.alias.encode('utf-8'))
                else:
                    self.chat_box.config(state=tkinter.NORMAL)
                    self.chat_box.insert(tkinter.END, message + '\n')
                    self.chat_box.config(state=tkinter.DISABLED)
                    self.chat_box.yview(tkinter.END)
            except:
                print('An error occurred!')
                self.client.close()
                break

    def send_message(self):
        message = f'{self.alias}: {self.input_area.get("1.0", tkinter.END)}'
        self.client.send(message.encode('utf-8'))
        self.chat_box.config(state=tkinter.NORMAL)
        self.chat_box.insert(tkinter.END, message + '\n')
        self.chat_box.config(state=tkinter.DISABLED)
        self.chat_box.yview(tkinter.END)
```

```
58          self.input_area.delete('1.0', tkinter.END)
59
60      def on_closing(self):
61          self.client.close()
62          self.root.destroy()
63
64  if __name__ == "__main__":
65      host = '127.0.0.1'
66      port = 8080
67      ChatClient(host, port)
```

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.