Cliente:

Esse código implementa um cliente de chat simples em Python, onde o usuário pode se conectar a um servidor de chat, enviar e receber mensagens.

1. Shebang Line e Imports

```
#!/usr/bin/env python3
import socket
import threading
import time
import sys
import os
```

- A linha `#!/usr/bin/env python3` indica que o script deve ser executado com o Python 3, usando o caminho padrão.
- Os módulos importados (`socket`, `threading`, `time`, `sys`, `os`) fornecem funcionalidades para criar a conexão com o servidor, manipular múltiplas threads, lidar com o tempo, interagir com o sistema e o terminal.

2. Função `clear_terminal()`

```
def clear_terminal():
    os.system('cls' if os.name == 'nt' else 'clear')
```

- Esta função limpa a tela do terminal. Se o sistema operacional for Windows (`'nt'`), ele usa o comando `cls`, caso contrário (Linux ou macOS), usa `clear`.

3. Função `get_nickname()`

```
def get_nickname():
    while True:
        nickname = input("nickname: ")
        if (3 <= len(nickname) <= 10):
            return nickname
        else:
            clear_terminal()
            print("The nickname is invalid. try again.\n")
            time.sleep(2)
            clear_terminal()
```

- Esta função entra em um loop até o usuário fornecer um apelido (nickname) que tenha entre 3 e 10 caracteres.
- Se o apelido for inválido, a tela é limpa e uma mensagem de erro aparece, pedindo ao usuário para tentar novamente.

4. Função `nickname_setter()`

```
def nickname_setter():
    global stopping_thread
    global client_socket

    while not stopping_thread:
        print("Enter a nickname between 3 and 10 characters:\n")
        try:
            nickname = get_nickname()
            client_socket.send(f"{nickname}".encode("utf-8"))

            response = client_socket.recv(1024).decode("utf-8")

            if (response == "FALSE"):
                clear_terminal()
                print("This nickname already exists. Try another
one!")
                time.sleep(2)
                clear_terminal()
            else:
                clear_terminal()
                break
        except:
            print("An unexpected error has occurred. Connection
closed!")
            client_socket.close()
            stopping_thread = True
            break
```

- Esta função envia o apelido escolhido pelo usuário ao servidor.
- Se o servidor responder que o apelido já está em uso (`response == "FALSE"`), o usuário é solicitado a tentar novamente.
- Se houver um erro de conexão, a função fecha o socket e define a variável `stopping_thread` como `True` para encerrar o programa.

5. Função `message_send()`

```
def message_send():
    global stopping_thread
    while not (stopping_thread):
        message = input()
        try:
            client_socket.send(message.encode("utf-8"))
            if (message.lower() == r"/close" or
message.lower().startswith(r"/close")):
                stopping_thread = True
                break
        except:
            print("An unexpect error has occured. Closing
connection!\n")
            client_socket.close()
            stopping_thread = True
```

- Esta função coleta as mensagens digitadas pelo usuário e as envia para o servidor.
- Se o usuário digitar `/close`, o cliente encerra a conexão e para de enviar mensagens.
- Se houver um erro, o socket é fechado e a variável `stopping_thread` é ativada para encerrar a conexão.

6. Função `message_recv()`

```
def message_recv():
    global stopping_thread
    while not (stopping_thread):
        try:
            message = client_socket.recv(2048).decode("utf-8")
            print(f"{message}\n")
        except:
            print("An unexpect error has occured. Closing
connection!\n")
            client_socket.close()
            stopping_thread = True
```

- Essa função recebe as mensagens do servidor e as imprime no terminal.
- Se houver um erro ao receber uma mensagem, o socket é fechado e a conexão é encerrada.

7. Configuração de Conexão

```
server_address = "192.168.112.1"
server_port = 12345
stopping_thread = False
```

- O cliente tenta se conectar ao servidor no endereço `192.168.112.1` e porta `12345`.
- A variável `stopping_thread` controla quando as threads de envio e recebimento de mensagens devem ser interrompidas.

8. Criação e Conexão do Socket

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

try:
    client_socket.connect((server_address, server_port))
    print(f"Connected to {server_address} : {server_port}\n")
except socket.error as msg:
    print(f"Connection error. {msg}\n")
    sys.exit()
```

- Cria um socket do tipo TCP (`socket.SOCK_STREAM`) e tenta conectar ao servidor.

9. Execução de Threads para Envio e Recebimento de Mensagens

```
rcv_thread = threading.Thread(target=message_recv)
send_thread = threading.Thread(target=message_send)

rcv_thread.start()
send_thread.start()
```

- O programa cria duas threads: uma para receber mensagens e outra para enviá-las, permitindo a comunicação simultânea.

10. Loop Principal

```
while True:
    if not (stopping_thread):
        pass
```

```
else:
    client_socket.close()
    sys.exit("Connection closed!\n")
```

- O loop principal verifica constantemente se a variável `stopping_thread` foi ativada para fechar a conexão e encerrar o programa.

Resumo:
Este código implementa um cliente de chat baseado em TCP, onde o usuário escolhe um apelido, se conecta a um servidor, e pode enviar e receber mensagens simultaneamente através de threads. O programa também trata possíveis erros de conexão e encerra a comunicação corretamente ao receber um comando `/close`.

Servidor:

Este código implementa um servidor de chat simples em Python que permite a vários clientes se conectarem, enviar mensagens e receberem mensagens uns dos outros em tempo real. O servidor também usa threads para lidar com múltiplas conexões simultâneas. Vamos detalhar o funcionamento de cada parte do código:

1. Shebang Line e Imports

```
#!/usr/bin/env python3

import socket

import threading

import os
```

- A linha `#!/usr/bin/env python3` indica que o script deve ser executado com o Python 3.
- Os módulos importados são:
  - `socket`: para criar e gerenciar as conexões de rede.
  - `threading`: para permitir a execução simultânea de múltiplos clientes.
  - `os`: para executar comandos do sistema operacional, como limpar o terminal.

2. Função `clear_terminal()`

```
def clear_terminal():

    os.system('cls' if os.name == 'nt' else 'clear')
```

- Esta função limpa a tela do terminal. Se o sistema for Windows, usa o comando `cls`; para outros sistemas operacionais (Linux/macOS), usa `clear`.

3. Função `broadcast_msg()`

python

Copiar código

```
def broadcast_msg(message, clientSender):
```

```
    for client in clients_sockets:
        if (client != clientSender):
            client.send(message.encode("utf-8"))
```

- Esta função envia uma mensagem para todos os clientes conectados, exceto o cliente que enviou a mensagem original (`clientSender`).
- O loop percorre a lista de `clients_sockets` (que armazena os sockets de todos os clientes conectados) e envia a mensagem para cada um.

4. Função `recv_message()`

```
def recv_message(client):
    global stop_threading
    while (not stop_threading):
        try:
            message = client.recv(2048).decode("utf-8")
            client_indexing = clients_sockets.index(client)
            nickname = clients_nicknames[client_indexing]
            if message == "/close":
                print(f"{clients_addresses[client_indexing]} has
been disconnected\n")

clients_addresses.remove(clients_addresses[client_indexing])
                clients_sockets.remove(client)
                clients_nicknames.remove(nickname)
                msg = f"{nickname} disconnected!"
                broadcast_msg(msg, client)
                client.close()
```

```
                stop_threading = True
                break
            else:
                msg = f"{nickname}: {message}\n"
                broadcast_msg(msg, client)
        except:
            client_indexing = clients_sockets.index(client)
            nickname = clients_nicknames[client_indexing]
            print(f"An unexpect error has occured at the rcv msg!
Closing connection at {clients_addresses[client_indexing]}")
            clients_sockets.remove(client)
            clients_nicknames.remove(nickname)

clients_addresses.remove(clients_addresses[client_indexing])
            broadcast_msg(f"{nickname} has been disconnected")
            client.close()
            stop_threading = True
```

- Esta função recebe mensagens de um cliente e retransmite para os outros clientes:
  - Usa `client.recv()` para receber uma mensagem do cliente.
  - Se a mensagem for `"/close"`, o cliente será desconectado: o servidor remove o cliente das listas `clients_sockets`, `clients_nicknames` e `clients_addresses`, envia uma mensagem de desconexão para os outros clientes e fecha a conexão do cliente.
  - Se for qualquer outra mensagem, o servidor retransmite para todos os clientes conectados (usando `broadcast_msg()`).
  - Se ocorrer um erro ao receber a mensagem, o cliente será desconectado, e o erro será tratado imprimindo uma mensagem no console.

5. Configuração de Conexão (Porta e Endereço)

```
port = 12345

host = "0.0.0.0"
```

- O servidor é configurado para escutar conexões na porta 12345 e no endereço IP `"0.0.0.0"`. Este endereço faz com que o servidor aceite conexões de qualquer interface de rede (ou seja, ele ficará acessível em toda a rede local).

6. Listas e Flag Global

```
clients_sockets = []

clients_nicknames = []

clients_addresses = []

nicknames = []

stop_threading = False
```

- Essas listas armazenam:
  - `clients_sockets`: os sockets de todos os clientes conectados.
  - `clients_nicknames`: os apelidos dos clientes.
  - `clients_addresses`: os endereços dos clientes.
- A flag `stop_threading` é usada para controlar o encerramento das threads.

7. Criação e Configuração do Socket do Servidor

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

server_socket.bind((host, port))

server_socket.listen()

print(f"server listening at port: {port}...")
```

- O servidor cria um socket TCP (`socket.SOCK_STREAM`) e o associa ao endereço e porta definidos.
- O socket começa a escutar por novas conexões com `server_socket.listen()`.

8. Aceitação de Conexões

```
while True:

    client, address = server_socket.accept()

    print(f"A connection has been established with
{str(address)}\n\n")
```

- O servidor entra em um loop infinito onde ele aceita novas conexões de clientes usando `server_socket.accept()`.
- Quando um cliente se conecta, o servidor imprime o endereço do cliente.

9. Processo de Apelido (Nickname)

```
while True:

    try:

        nickname = client.recv(1024).decode("utf-8")

        if (nickname in clients_nicknames):

            client.send("FALSE".encode("utf-8"))

        else:

            client.send("TRUE".encode("utf-8"))

            break

    except:

        print("An unexpect error has occured! Closing connection")

        client.close()
```

- O servidor pede ao cliente que envie um apelido (nickname).
- Se o apelido já estiver em uso (ou seja, está na lista `clients_nicknames`), o servidor envia `"FALSE"` para o cliente, e ele terá que escolher outro.
- Se o apelido for único, o servidor envia `"TRUE"` e continua.

10. Adição de Clientes à Lista e Criação de Threads

```
clients_nicknames.append(nickname)

clients_addresses.append(address)

clients_sockets.append(client)


client_thread = threading.Thread(target=recv_message, args=[client])

client_thread.start()
```

- Depois de aceitar a conexão e verificar o apelido, o servidor adiciona o cliente às listas de sockets, apelidos e endereços.
- Ele então cria uma nova thread para lidar com as mensagens desse cliente, permitindo que o servidor processe várias conexões simultaneamente.

Resumo

Este código implementa um servidor de chat usando sockets TCP e threads. Ele permite que vários clientes se conectem e conversem em tempo real. O servidor gerencia as conexões, controla os apelidos dos clientes, retransmite mensagens e trata erros, como desconexões inesperadas.

Funções principais:

- `broadcast_msg()`: envia mensagens para todos os clientes, exceto o remetente.
- `recv_message()`: recebe mensagens de um cliente, trata desconexões e retransmite mensagens para outros.
- Threads são usadas para garantir que cada cliente tenha seu próprio canal de comunicação com o servidor.