Requisitos Funcionais:

1. O sistema deve aceitar conexões simultâneas de múltiplos clientes.
2. O sistema deve transmitir mensagens enviadas por um cliente a todos os outros clientes conectados.
3. O sistema deve permitir que um cliente envie mensagens privadas diretamente a outro cliente especificado pelo nome.
4. O sistema deve detectar quando um cliente se desconecta e remover este cliente da lista de conexões.
5. O sistema deve lidar com mensagens mal formatadas para mensagens privadas e notificar o remetente sobre o erro.
6. O sistema deve registrar a entrada e saída de clientes, bem como mensagens enviadas, junto com um carimbo de data/hora.
7. O sistema deve exibir mensagens públicas e privadas recebidas em tempo real.
8. O sistema deve permitir que o usuário encerre sua participação enviando o comando sair.
9. O sistema deve exibir informações de conexão, como mensagens de boas-vindas e notificações sobre a entrada ou saída de outros usuários.
10. O sistema deve lidar com falhas de comunicação e notificar o usuário em caso de problemas.
11. No sistema o cliente deve se conectar ao servidor informando seu nome.
12. No sistema o cliente deve ser capaz de enviar mensagens públicas para todos os outros usuários conectados.

13. No sistema o cliente deve ser capaz de enviar mensagens privadas usando o formato @nome mensagem.

Explicação dos Sockets, Threads e Broadcast:

socket_cliente.connect((HOST, PORTA)) é utilizado para conectar o cliente ao servidor do HOST (127.0.0.1), pela porta (9999), essa linha estabelece uma conexão TCP com o servidor na porta fornecida.

socket_cliente.sendall(nome.encode()) é utilizado, recolhendo o nome inserido via input do cliente, e envia para o servidor, usando o metodo (sendall(). Já o encode() converte o nome para bytes, que é o formato necessário para envio via socket.

Em:

```
def receber_mensagens():
  while True:
    try:
      mensagem = socket_cliente.recv(1024).decode()
      if mensagem:
        print(mensagem)
      else:
        break
    except:
      print("Erro ao receber mensagem.")
```

```
        break
```

Utilizamos o metódo recv(1024). Cujo é utilizado para receber dados do servidor, caso a mensagem enviada não seja vazia, ela é decodificada em decode(), e exibida. Caso o servidor feche a conexão, ou não houver dados sendo enviados, o loop termina.

Em:

```
def enviar_mensagens():
    while True:
        try:
            mensagem = input('')
            if mensagem.lower() == 'sair':
                socket_cliente.sendall(f"{nome} saiu do chat.".encode())
                break
            socket_cliente.sendall(mensagem.encode())
        except KeyboardInterrupt:
            print("\nSaindo do chat...")
            break
    socket_cliente.close()
```

A função enviar_mensagens captura as mensagens enviadas pelo usuário e as envia para o servidor. Caso o usuário digite "sair", uma mensagem especial é enviada com a função

socket_cliente.sendall(), informando que o mesmo saiu do chat. Após isso Conexão é encerrada com socket_cliente.close().

Em:

```
def recebe_dados(socket_cliente, endereco):
    nome = socket_cliente.recv(50).decode()
    lista_clientes.append((socket_cliente, nome))
```

é responsável por lidar com mensagens enviadas pelo cliente. Recebe o nome do cliente usando recv(50) que recebe até 50 bytes, e adiciona o nome do cliente a lista de cliente conectados.

Em:

```
if mensagem.startswith('@'):
    nome_destino, msg = tratar_unicast(mensagem)
    enviar_unicast(nome, nome_destino, msg, socket_cliente)
```

Se uma mensagem começar @, o servidor considera uma mensagem privada/unicast, a função unicast separa tanto o nome do usuário, quanto o conteúdo da mensagem.

Já em enviar_unicast, ele localiza o cliente destinado, e apenas envia a mensagem ao mesmo.

Em:

```
broadcast(f"[{horario}] {nome}: {mensagem}", socket_cliente)
```

Caso a mensagem não comece com @, o servidor trata como uma mensagem pública, assim enviando a todos os clientes conectados, exceto a pessoa que a enviou.

Em:

HOST = '127.0.0.1'

PORTA = 9999

sock_server = sock.socket(sock.AF_INET, sock.SOCK_STREAM)

sock_server.bind((HOST, PORTA))

sock_server.listen()

AF_INET define o uso de IPv4

SOCK_STREAM define que o servidor usará TCP.

sock_server.bind((HOST, PORTA)) liga o socket ao endereço de IP 127.0.0.1, e a porta 9999.

sock_server.listen() coloca o servidor em estado de espera, esperando por conexões de clientes.

THREADS:

Threads no cliente:

thread_receber = threading.Thread(target=receber_mensagens)

thread_receber.start()

Thread que executa a função receber_mensagens, que fica em loop recebendo mensagens do servidor. Também permite que o cliente receba mensagens a qualquer momento.

Threads no servidor:

```
thread_cliente = threading.Thread(target=recebe_dados, args=(sock_conn, ender))
thread_cliente.start()
```

O servidor cria uma thread que executa a função "recebe_dados", para cada cliente novo, que é conectado.

Broadcast:

```
def broadcast(mensagem, cliente_excluido=None):
    for cliente, nome in lista_clientes:
        if cliente != cliente_excluido:
            try:
                cliente.send(mensagem.encode())
            except:
                remover(cliente, nome)
```

A função broadcast recebe 2 coisas:

Mensagem: É a mensagem que será enviada aos clientes.

cliente_exlcluido: Que é o client que será removido do recebimento da mensagem, geralmente o próprio remetente.

O broadcast é chamado quando:

broadcast(f"{nome} entrou no chat!", socket_cliente) O cliente entra no chat.

broadcast(f"[{horario}] {nome}: {mensagem}", socket_cliente) Quando o servidor recebe uma mensagem normal de um cliente. Além de formatar com data e hora, nome do remetente e o conteúdo da mensagem.

broadcast(f"{nome} saiu do chat.") Quando o cliente sai do chat