Alunos: Matheus Augusto, Tiago Gorri, Germano Lagana

Requisitos Funcionais

RF1: O sistema deve permitir que o usuário coloque o seu nome para entrar no chat. O nome não pode ter espaço e nem “@”.

RF2: O sistema deve bloquear a entrada de alguém com o mesmo nome de um usuário já cadastrado, e pedir para que seja utilizado outro nome.

RF3: Todos os usuários conectados recebem a mensagem quando um novo usuário entra no chat, incluindo IP e porta.

RF4: O sistema deve permitir que o chat seja em tempo real, utilizando threads.

RF5: O sistema deve permitir que os usuários possam enviar mensagens no chat para que todos possam ver

RF6: O sistema permite mensagens privadas usando "@" para direcionar a mensagem a um destinatário específico.

RF7: O sistema não pode permitir que o usuário envie uma mensagem privada para si mesmo

RF8: O sistema deve enviar uma mensagem para todos avisando quando um usuário sair do chat.

RF9: O cliente é removido do chat ao clicar em "Sair".

RF10: O sistema deve permitir que, mesmo que um usuário tenha saído do chat, o sistema siga funcionando, fazendo com que outros usuários possam entrar.

RF11: Caso o cliente tente conectar sem o servidor estar ativo, aparece um pop-up com uma mensagem de erro.

RF12: O IP e a porta do cliente são exibidos para o próprio usuário e para os demais.

Sockets

Servidor:

- Aqui, é criado um socket de servidor usando a família de endereços AF_ INET (IPv4) e o tipo de socket SOCK_STREAM (TCP).

```
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # Cria o socket
```

- Permite reutilizar o endereço e porta rapidamente após o servidor ser encerrado. Isso evita erros caso a porta ainda esteja "presa" por uma conexão anterior.

```
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)  # Permite reutilizar o endereço
```

- O servidor é associado ao endereço 0.0.0.0 (significa "todos os IPs disponíveis") e à porta 12345.

```
server.bind(('0.0.0.0', 12345))  # Associa o socket a um endereço e porta
```

- O servidor escuta novas conexões usando server.listen(). Quando um cliente tenta se conectar, o servidor aceita essa conexão e retorna:

    - sock: A nova socket usada para se comunicar com o cliente.

    - addr: O endereço do cliente (IP e porta).

```
server.listen()  # Habilita o socket para aceitar conexões
sock, addr = server.accept()  # Aceita uma nova conexão
```

- O servidor usa send() para transmitir mensagens codificadas.

```
sock.send((msg + '\n').encode('utf-8'))
```

- O servidor lê dados da socket criada para o cliente.

```
msg = sock.recv(1024).decode('utf-8').strip()  # Recebe mensagem
```

- sock.close(): Fecha a conexão individual com um cliente.

```
sock.close()  # Fecha o socket do cliente
```

- server.close(): Fecha a socket do servidor, encerrando o serviço para todos.

```
server.close()  # Fecha o socket do servidor
```

Cliente:

- socket.AF_INET: Indica que o cliente usará o protocolo IPv4.

- socket.SOCK_STREAM: Configura a comunicação para usar o protocolo TCP (orientado a conexão).

- Estabelece uma conexão com o servidor no IP 192.168.68.110 e na porta 12345.

- Após essa linha, o cliente fica conectado e pronto para enviar e receber dados.

```
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # Cria um socket para comunicação via TCP
client.connect(('192.168.68.110', 12345))  # Conecta ao servidor no IP e porta especificados
```

- Usa o método send() para enviar mensagens ao servidor.

- As mensagens são convertidas para o formato de bytes usando encode('utf-8'), necessário para transmissão.

```
client.send(msg.encode('utf-8'))  # Envia a mensagem ao servidor
```

- O método recv(1024) lê até 1024 bytes de dados enviados pelo servidor.

- Após a recepção, os dados são decodificados (decode('utf-8')) para transformar de bytes em string e removem-se espaços extras com strip().

```
msg = client.recv(1024).decode('utf-8').strip()  # Recebe mensagens do servidor, decodifica e remove espaços extras
```

- Fecha a conexão da socket no cliente.

- Isso libera os recursos usados pela conexão e sinaliza ao servidor que o cliente foi desconectado.

```
client.close()  # Fecha a conexão com o servidor
```

Broadcast

- O tratamento broadcast nos dois códigos é implementado no servidor, e sua funcionalidade está ligada à distribuição de mensagens para todos os clientes conectados, com algumas condições específicas. Ele é utilizado para:

- Informar mensagens públicas enviadas por clientes.

- Enviar notificações gerais, como entradas e saídas de usuários.

- Atualizar a lista de usuários ativos.

```python
# Função para enviar mensagens a todos os clientes, exceto o remetente (opcional)
def broadcast(msg, exclude=None):
    for c in clients.values():  # Itera sobre todos os clientes conectados
        if c != exclude:  # Exclui o remetente se necessário
            try:
                c.send((msg + '\n').encode('utf-8'))  # Envia a mensagem
            except:
                pass
```

Unicast

O tratamento unicast é responsável por enviar mensagens de um cliente específico para outro cliente, ou seja, uma comunicação direta entre dois participantes no chat. Esse processo é implementado de forma a permitir mensagens privadas entre usuários, e ele está presente no servidor, que é responsável por gerenciar e direcionar essas mensagens. É feito utilizando o símbolo @ seguido do nome do destinatário (@nome_destinatario mensagem).

```python
if msg.startswith("@"):  # Mensagem privada
    try:
        target_name, private_msg = msg[1:].split(' ', 1)  # Divide o destinatário e a mensagem
        if target_name == name:  # Bloqueia mensagens para si mesmo
            sock.send("Você não pode enviar mensagens privadas para si mesmo.\n".encode('utf-8'))
        elif target_name in names:  # Verifica se o destinatário existe
            target_sock = clients[names[target_name]]  # Obtém o socket do destinatário
            target_sock.send(f"{name}: {private_msg} (privado)\n".encode('utf-8'))  # Envia a mensagem privada
        else:
            sock.send(f"Usuário {target_name} não encontrado.\n".encode('utf-8'))  # Erro de destinatário
    except ValueError:
        sock.send("Formato inválido. Use @nome mensagem.\n".encode('utf-8'))  # Erro de formato
else:  # Mensagem pública
    broadcast(f"{name}: {msg}", exclude=sock)  # Envia para todos, exceto o remetente
```

Threads

Servidor

No servidor, as threads são usadas para lidar com múltiplos clientes ao mesmo tempo. Quando um novo cliente se conecta, uma nova thread é criada para gerenciar essa conexão de forma independente da thread principal do servidor. Isso permite que o servidor atenda vários clientes simultaneamente. conexões e distribuindo mensagens em tempo real para todos os participantes do chat.

- A primeira thread que o servidor cria é para gerenciar os comandos administrativos (como o comando para sair do servidor), sem bloquear o restante da execução do servidor.

```
# Função para gerenciar comandos do servidor (como encerramento)
def server_control(server):
    global server_running
    while server_running:
        command = input("Digite 'sair' para encerrar o servidor: ").strip().lower()  # Comando de entrada
        if command == "sair":  # Caso o comando seja para encerrar
            print("Encerrando o servidor...")
            broadcast("SERVIDOR_ENCERRADO")  # Notifica todos os clientes
            for c in clients.values():  # Fecha os sockets de todos os clientes
                c.close()
            server.close()  # Fecha o socket do servidor
            server_running = False  # Define que o servidor não está mais rodando
            print("Servidor encerrado com sucesso.")
```

- Thread responsável:

```
# Thread para gerenciar os comandos do servidor
threading.Thread(target=server_control, args=(server,), daemon=True).start()
```

- A principal utilização de threads no servidor é para gerenciar cada cliente individualmente. Quando um novo cliente se conecta, o servidor cria uma nova thread para lidar com ele sem bloquear a aceitação de novas conexões. Isso é feito na seguinte parte do código:

```
# Função principal para iniciar o servidor
def start_server():
    global server_running
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # Cria o socket
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)  # Permite reutilizar o endereço
    server.bind(('0.0.0.0', 12345))  # Associa o socket a um endereço e porta
    server.listen()  # Habilita o socket para aceitar conexões
    print("Servidor está funcionando...")

    # Thread para gerenciar os comandos do servidor
    threading.Thread(target=server_control, args=(server,), daemon=True).start()

    # Loop principal para aceitar conexões de clientes
    while server_running:
        try:
            sock, addr = server.accept()  # Aceita uma nova conexão
            threading.Thread(target=handle_client, args=(sock, addr), daemon=True).start()  # Inicia uma thread para o cliente
        except:
            break
```

Cliente

No cliente, as threads são usadas para separar a recepção de mensagens (que acontece de forma contínua enquanto o cliente está no chat) e a entrada do usuário (onde ele digita as mensagens).

- O cliente utiliza uma thread para ficar recebendo mensagens do servidor de forma contínua, enquanto a thread principal lida com o envio de mensagens. O objetivo dessa thread é não bloquear a interface do cliente para que ele possa sempre receber e exibir mensagens, sem precisar esperar o envio de uma mensagem.

```python
# Função para receber mensagens do servidor
def receive_messages():
    try:
        while True:
            msg = client.recv(1024).decode('utf-8').strip()  # Recebe mensagens do servidor, decodifica e remove espaços extras
            if msg == "SERVIDOR_ENCERRADO":  # Verifica se o servidor enviou um comando de encerramento
                messagebox.showinfo("Desconectado", "O servidor foi encerrado. O chat será fechado.")  # Exibe aviso de desconexão
                root.destroy()  # Fecha a janela principal do chat
                break  # Encerra o loop de recebimento
            chat.config(state=tk.NORMAL)  # Permite edição no campo de chat para inserir a nova mensagem
            chat.insert(tk.END, msg + '\n')  # Adiciona a mensagem recebida ao final do chat
            chat.config(state=tk.DISABLED)  # Desabilita a edição novamente para evitar alterações manuais
    except:  # Caso ocorra algum erro
        client.close()  # Fecha a conexão com o servidor
```

Como é executada:

```python
threading.Thread(target=receive_messages, daemon=True).start()  # Inicia uma thread para receber mensagens do servidor
```