Informe del Código TPC

El código del cliente TCP está diseñado para conectarse a un servidor, enviar y recibir mensajes. A continuación se detalla la funcionalidad de cada parte del código:

1. Configuración del Cliente

```
server_ip = input("Ingrese la IP del servidor: ")
server_port = 60000

# Crear socket TCP
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((server_ip, server_port))
```

Funcionalidad:

- Entrada de la IP del servidor: Solicita al usuario la dirección IP del servidor al que desea conectarse.
- Puerto del servidor: Define el puerto en el que el servidor está escuchando.
- Creación del socket TCP: Se crea un socket TCP.
- Conexión al servidor: El cliente se conecta al servidor utilizando la IP y el puerto especificados.

2. Función para Recibir Mensajes del Servidor

```
def receive_messages(client_socket):
    while True:
        try:
            message = client_socket.recv(1024).decode('utf-8')
            if message == 'exit':
                print("Conexión cerrada por el servidor.")
                client_socket.close()
                break
            print(message)
        except ConnectionResetError:
            print("Conexión cerrada por el servidor.")
            client_socket.close()
            break
        except:
            print("Error de conexión con el servidor.")
            client_socket.close()
            break
```

Funcionalidad:

- Bucle infinito: Permanece en un bucle continuo para recibir mensajes del servidor.
- Recepción y decodificación del mensaje: Recibe mensajes del servidor con un tamaño máximo de 1024 bytes y los decodifica en formato UTF-8.
- Manejo del mensaje 'exit': Si el mensaje recibido es `'exit'`, se cierra el socket y se sale del bucle.
- ConnectionResetError: Se imprime un mensaje indicando que la conexión fue cerrada por el servidor y se cierra el socket.
- Excepciones generales: Se imprime un mensaje indicando un error de conexión y se cierra el socket.

3. Creación de un Hilo para Recibir Mensajes

```
receive_thread = threading.Thread(target=receive_messages, args=(client_socket,))
receive_thread.start()
```

Funcionalidad:

- Creación de un hilo: Se crea un hilo que ejecuta la función `receive_messages`, permitiendo que el cliente reciba mensajes de forma asíncrona.
- Inicio del hilo: El hilo se inicia, lo que permite la ejecución concurrente de `receive_messages` junto con el envío de mensajes.
-

4. Enviar Mensajes al Servidor

```
while True:
    message = input()
    client_socket.send(message.encode('utf-8'))
    if message == 'exit':
        print("Conexión cerrada.")
        client_socket.close()
        break
    elif "ChatBot: Contraseña incorrecta. Desconectando..." in message:
        print(message)
        client_socket.close()
        break
```

Funcionalidad:

- Bucle infinito: Permanece en un bucle continuo para enviar mensajes al servidor.
- Entrada del mensaje: Solicita al usuario que ingrese un mensaje.
- Envío del mensaje: Envía el mensaje al servidor, codificándolo en formato UTF-8.
- Manejo del mensaje 'exit': Si el mensaje ingresado es `'exit'`, se cierra el socket y se rompe el bucle.
- Manejo de la desconexión por contraseña incorrecta: Si el mensaje contiene `"ChatBot: Contraseña incorrecta. Desconectando..."`, se imprime el mensaje, se cierra el socket y se rompe el bucle.

Conclusión
El código del cliente TCP permite al usuario conectarse a un servidor, recibir y enviar mensajes. Maneja la desconexión tanto desde el servidor como del lado del cliente. Utiliza hilos para manejar la recepción de mensajes de manera asíncrona, permitiendo así que los mensajes se reciban mientras el usuario sigue interactuando con el cliente.

Informe del Código del Servidor TCP

El código del servidor TCP implementa un servidor de chat con múltiples funcionalidades, como el manejo de mensajes públicos y privados, la autenticación de usuarios, y la persistencia de mensajes y usuarios en archivos. A continuación, se detalla la funcionalidad de cada parte del código y de cada función:

1. Configuración del Servidor

```
server_ip = ''
server_port = 60000
```

```
# Crear socket TCP
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((server_ip, server_port))
server_socket.listen(5)

print(f"Servidor escuchando en {server_ip}:{server_port}")

clients = []
usernames = []
client_addresses = []
passwords = {}

# Ruta del archivo de la base de datos
db_folder = 'database'
db_file = os.path.join(db_folder, 'chat.pkl')
users_file = os.path.join(db_folder, 'users.pkl')

# Crear la carpeta de la base de datos si no existe
os.makedirs(db_folder, exist_ok=True)

# Cargar mensajes de la base de datos
if os.path.exists(db_file):
    with open(db_file, 'rb') as f:
        messages = pickle.load(f)
else:
    messages = []

# Cargar usuarios y contraseñas de la base de datos
if os.path.exists(users_file):
    with open(users_file, 'rb') as f:
        stored_users = pickle.load(f)
        usernames = list(stored_users.keys())
        passwords = stored_users
else:
    stored_users = {}
```

Funcionalidad:

- Creación y configuración del socket TCP: El servidor crea un socket TCP, lo asocia a una IP y puerto, y lo pone a escuchar conexiones entrantes.
- Inicialización de variables: Se inicializan listas para almacenar los clientes conectados, nombres de usuarios, direcciones IP de los clientes y contraseñas.
- Configuración de la base de datos: Se define la ruta de los archivos de la base de datos y se crea la carpeta si no existe.
- Carga de datos de la base de datos: Se cargan los mensajes y usuarios guardados en archivos pickle.

2. Función `store_message`

```
def store_message(ip, username, message, recipient=None):
    messages.append((ip, username, message, recipient))
    with open(db_file, 'wb') as f:
        pickle.dump(messages, f)
```

Funcionalidad:

- Almacenamiento de mensajes: Guarda un mensaje en la lista de mensajes junto con la IP del usuario, el nombre de usuario y el destinatario (si es un mensaje privado).
- Persistencia de mensajes: Guarda los mensajes en un archivo pickle para persistencia.

3. Función `get_message_history`

```
def get_message_history(requester_username):
    public_messages = [msg for msg in messages if msg[3] is None]
    private_messages = [msg for msg in messages if msg[3] == requester_username or msg[1] == requester_username]
    return public_messages + private_messages
```

Funcionalidad:

- Obtención del historial de mensajes: Devuelve una lista de mensajes públicos y mensajes privados que involucran al usuario solicitante.

4. Función `disconnected`

```
def disconnected(client_socket):
    if client_socket in clients:
        index = clients.index(client_socket)
        username = usernames[index]
        ip = client_addresses[index]
        message = f"ChatBot: {username} ({ip}) se ha desconectado".encode('utf-8')
        broadcast(message, client_socket)
        clients.remove(client_socket)
        usernames.remove(username)
        client_addresses.remove(ip)
```

Funcionalidad:

- Desconexión de un cliente: Elimina el cliente desconectado de las listas de clientes, nombres de usuarios y direcciones IP. Envía un mensaje de desconexión a todos los demás clientes.

5. Función `broadcast`

```
def broadcast(message, sender_socket):
    for client_socket in clients:
        if client_socket != sender_socket:
            client_socket.send(message)
```

Funcionalidad:

- Difusión de mensajes: Envía un mensaje a todos los clientes conectados, excepto al que envió el mensaje originalmente.

6. Función `send_private_message`

```
def send_private_message(sender_socket, recipient_username, message):
    if recipient_username in usernames:
        recipient_index = usernames.index(recipient_username)
        recipient_socket = clients[recipient_index]
```

```
        sender_index = clients.index(sender_socket)
        sender_username = usernames[sender_index]
        private_message = f"Privado de {sender_username}: {message}".encode('utf-8')
        recipient_socket.send(private_message)
        store_message(client_addresses[sender_index], sender_username, f"Privado para
{recipient_username}: {message}", recipient_username)
    else:
        sender_socket.send(f"Usuario {recipient_username} no encontrado.".encode('utf-8'))
```

Funcionalidad:

- Envío de mensajes privados: Envía un mensaje privado a un usuario específico. Si el destinatario no existe, informa al remitente.

7. Función `hash_password`

```
def hash_password(password):
    return hashlib.sha256(password.encode()).hexdigest()
```

Funcionalidad:

- Hash de contraseñas: Devuelve el hash SHA-256 de una contraseña, utilizado para almacenar y comparar contraseñas de manera segura.

8. Función `handle_client`

```
def handle_client(client_socket, client_address):
    print(f"{client_address} Cliente conectado.")
    try:
        # Solicitar nombre de usuario
        client_socket.send("ChatBot: Username".encode('utf-8'))
        username = client_socket.recv(1024).decode('utf-8')
        # Solicitar contraseña
        client_socket.send("ChatBot: Introduce contraseña:".encode('utf-8'))
        password = client_socket.recv(1024).decode('utf-8')

        # Guardar el nombre de usuario y la contraseña para usuarios nuevos
        if username not in usernames:
            usernames.append(username)
            clients.append(client_socket)
            client_addresses.append(client_address[0])
            passwords[username] = hash_password(password)
            # Almacenar en un archivo pickle los usuarios
            stored_users[username] = passwords[username]
            with open(users_file, 'wb') as f:
                pickle.dump(stored_users, f)
            message = f"ChatBot: {username} ({client_address[0]}) se ha unido al chat".encode('utf-8')
            broadcast(message, client_socket)
        # Verificar la contraseña si el usuario ya entró antes
        else:
            if passwords[username] != hash_password(password):
                client_socket.send("ChatBot: Contraseña incorrecta. Desconectando...".encode('utf-8'))
                client_socket.close()
                return

        while True:
            message = client_socket.recv(1024).decode('utf-8')
```

```
            if message == 'exit':
                print(f"{client_address} Cliente se ha desconectado.")
                client_socket.send('exit'.encode('utf-8'))
                disconnected(client_socket)
                client_socket.close()
                break
            # Sección original que maneja la solicitud de historial
            elif message == 'history':
                attempts = 0
                max_attempts = 3
                # Este bucle verifica que haya intentado 3 veces introducir la contraseña para ver el historial
                while attempts < max_attempts:
                    client_socket.send("ChatBot: Introduce contraseña para ver el historial:".encode('utf-8'))
                    password = client_socket.recv(1024).decode('utf-8')
                    index = clients.index(client_socket)
                    username = usernames[index]
                    # Si la contraseña fue verificada correctamente devuelve el historial
                    if passwords[username] == hash_password(password):
                        history = get_message_history(username)
                        history_message = '\n'.join([f"{ip} ({username}): {msg}" for ip, username, msg, _ in history])
                        client_socket.sendall(history_message.encode('utf-8'))
                        break
                    else:
                        attempts += 1
                        client_socket.send(f"ChatBot: Contraseña incorrecta. Intentos restantes: {max_attempts -
attempts}".encode('utf-8'))
                # Si se terminaron los 3 intentos sale del servidor
                if attempts == max_attempts:
                    client_socket.send("ChatBot: Demasiados intentos fallidos. Desconectando...".encode('utf-8'))
                    client_socket.send('exit'.encode('utf-8'))
                    disconnected(client_socket)
                    client_socket.close()
            # Este código detecta mensajes privados si comienza con @
            elif message.startswith('@'):
                recipient_username, private_message = message.split(' ', 1)
                recipient_username = recipient_username[1:]  # Eliminar el prefijo '@'
                send_private_message(client_socket, recipient_username, private_message)
            # Este código es para los mensajes públicos
            else:
                print(f"{client_address}: {message}")
                index = clients.index(client_socket)
                username = usernames[index]
                ip = client_addresses[index]
                store_message(ip, username, message)
                broadcast(f"{username}: {message}".encode('utf-8'), client_socket)

        except Exception as e:
            print(f"Error: {e}")
            disconnected(client_socket)
            client_socket.close()
```

Funcionalidad:

- Manejo de nuevos clientes: Solicita y autentica el nombre de usuario y la contraseña del cliente.
- Manejo de nuevos clientes: Solicita el nombre de usuario y la contraseña del cliente.

- Si el usuario es nuevo, guarda la información del usuario y la contraseña hasheada, y almacena estos datos en un archivo.
- Si el usuario ya existe, verifica la contraseña.

Comunicación continua:

- En un bucle continuo, recibe y maneja mensajes de los clientes.
- Si el mensaje es `'exit'`, desconecta al cliente.
- Si el mensaje es `'history'`, solicita la contraseña para mostrar el historial y permite hasta 3 intentos.
- Si el mensaje comienza con `'@'`, lo trata como un mensaje privado.
- Si no, lo trata como un mensaje público y lo envía a todos los clientes conectados.

9. Función `receive_connections`

```
def receive_connections():
    # Aceptar conexiones de los clientes
    while True:
        client_socket, client_address = server_socket.accept()
        client_handler = threading.Thread(target=handle_client, args=(client_socket, client_address))
        client_handler.start()
```

Funcionalidad:

- Aceptar conexiones entrantes:
- Espera conexiones de clientes.
- Crea un nuevo hilo para manejar cada cliente conectado, lo que permite la concurrencia y el manejo de múltiples clientes simultáneamente.

10. Función `signal_handler`

```
def signal_handler(sig, frame):
    print('Deteniendo el servidor...')
    for client in clients:
        client.send('exit'.encode('utf-8'))
        client.close()
    sys.exit(0)
```

Funcionalidad:

- Manejo de señales:
- Captura la señal de interrupción (por ejemplo, Ctrl+C) para realizar una desconexión limpia.
- Envía un mensaje de salida a todos los clientes y cierra sus conexiones antes de detener el servidor.

11. Registro de la función de manejo de señales y inicio de la recepción de conexiones

```
signal.signal(signal.SIGINT, signal_handler)
receive_connections()
```

Funcionalidad:

- Registro de manejo de señales:
- Registra `signal_handler` para manejar la señal `SIGINT` (interrupción de teclado).

- Inicio de recepción de conexiones:
- Llama a `receive_connections` para empezar a aceptar conexiones de clientes.

Resumen
Este código implementa un servidor de chat multiusuario que ofrece varias funcionalidades:

- Autenticación de usuarios: con manejo de nombres de usuario y contraseñas.
- Manejo de mensajes públicos: que son difundidos a todos los usuarios conectados.
- Manejo de mensajes privados: entre usuarios específicos.
- Persistencia de mensajes: y datos de usuarios mediante el uso de archivos pickle.
- Desconexión segura: de clientes y manejo de intentos fallidos para ver el historial de mensajes.
- Capacidad de manejo de múltiples clientes: simultáneamente mediante hilos (threads).
- Desconexión y cierre seguros del servidor: al recibir una señal de interrupción.

Estas funcionalidades aseguran una experiencia de chat robusta y segura para los usuarios.