Chat Básico en Python, Sistemas Distribuidos

GÓMEZ EDGAR, Universidad Central de Venezuela, Venezuela

Se implementa un sistema de chat distribuido en Python basado en la arquitectura cliente-servidor. El servidor, concurrente, gestiona múltiples conexiones mediante sockets TCP y el uso de hilos, retransmitiendo mensajes a todos los clientes conectados. Cada cliente también emplea hilos para mantener una interfaz responsiva, separando el envío y recepción de mensajes. Se describe el protocolo de comunicación, la estructura de clases y el manejo de conexiones. El trabajo ejemplifica conceptos fundamentales de sistemas distribuidos como concurrencia, comunicación en red y el modelo cliente-servidor.

CCS Concepts: • **Organización de sistemas informáticos → Sistemas Distribuidos**; *Cliente Servidor*; *Concurrencia*; Hilos.

Additional Key Words and Phrases: Sistemas Distribuidos, Cliente Servidor, Concurrancia, Hilos

ACM Reference Format:
GÓMEZ EDGAR. 2026. Chat Básico en Python, Sistemas Distribuidos. *Proc. ACM Hum.-Comput. Interact.* 1, 1, Article 1 (January 2026), 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1. Introducción

En este paper abordamos la implementación de un chat básico con un modelo cliente servidor, donde el servidor crea un socket para una conexión bidireccional exponiendo un puerto red de mediante el cual se conecta cada cliente. El servidor usa hilos para evitar que su hilo principal se vea bloqueado mientras maneja otras cargas de trabajo, lo cual le permite siempre estar atento a nuevas conexiones entrantes y el cliente lo usa para que su interfaz sea responsiva al momento de enviar y recibir mensajes.

2. Conceptos Claves

Algunos conceptos claves son:

- Modelo Cliente Servidor: Es una arquitectura de software que organiza la comunicación y la distribución de tareas entre dos tipos de entidades:
  - Cliente: Un programa o proceso que solicita servicios o recursos. Generalmente inicia la comunicación y tiene una interfaz para interactuar con el usuario o con otro sistema.
  - Servidor: Servidor: Un programa o proceso que provee servicios o recursos a uno o múltiples clientes. Está diseñado para esperar (estar "a la escucha") peticiones y responder a ellas.
  - Características clave:
    - Comunicación asimétrica (el cliente inicia, el servidor responde).
    - Centralización de la lógica de negocio y los datos en el servidor.
    - Los clientes suelen ser ligeros y dependen del servidor.
- Socket: Es una abstracción software que proporciona una interfaz para enviar y recibir datos a través de la red, utilizando protocolos como TCP o UDP. Funcionamiento básico:
  - El servidor crea un socket y lo "enlaza" a un puerto específico para "escuchar".
  - El cliente crea un socket y se "conecta" a la dirección IP y puerto del servidor.
  - Una vez establecida la conexión, ambos extremos pueden leer y escribir datos a través de sus sockets.

Author's Contact Information: GÓMEZ EDGAR, Universidad Central de Venezuela, Caracas, Venezuela.

2026. ACM 2573-0142/2026/1-ART1
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- Hilos: Son unidades de ejecución más pequeñas dentro de un proceso. Un proceso puede tener múltiples hilos que se ejecutan de manera concurrente, compartiendo los mismos recursos del proceso (como memoria, archivos abiertos, etc.), pero cada hilo tiene su propia pila de ejecución y contador de programa. Características clave:
  - Concurrencia: Permiten realizar múltiples tareas aparentemente al mismo tiempo dentro de un mismo programa.
  - Comparten memoria: La comunicación entre hilos de un mismo proceso es muy eficiente, pero requiere sincronización para evitar condiciones de carrera.
  - Más ligeros que procesos: Crear y cambiar entre hilos tiene menos sobrecarga que crear procesos.

En la programación de servidores, es común usar hilos para manejar múltiples clientes simultáneamente. Cuando un servidor acepta una nueva conexión de un cliente a través de un socket, puede crear un hilo dedicado para atender las peticiones de ese cliente específico, permitiendo que el servidor principal siga escuchando nuevas conexiones.

En el lado del cliente, los hilos se utilizan principalmente para manejar concurrencia en la interfaz de usuario (UI) y la comunicación en red de manera eficiente.

3. Implementacion del Servidor

3.1. Estructura de la Clase ChatServer

Usando el paradigma orientado a objetos (POO), codificamos el servidor como una clase llamada `ChatServer` con su constructor:

Listing 1. Constructor de la clase ChatServer

```
    class ChatServer:
        def __init__(self, host_ip=socket.gethostbyname(socket.gethostname()), host_port=12345):
            self.clients_sockets_list = []
            self.clients_names_list = []
            self.server_socket = None

            self.HOST_IP = host_ip
            self.HOST_PORT = host_port
            self.ENCODER = "utf-8"
            self.BUFFER_SIZE = 1024

```

El constructor obtiene la IP del servidor (por defecto la IP local) y el puerto. Se definen las siguientes estructuras de datos:

- `clients_sockets_list`: Lista que almacena los sockets de cada cliente conectado.
- `clients_names_list`: Lista que almacena los nombres de los clientes conectados.
- `server_socket`: Socket principal del servidor para aceptar conexiones.
- `HOST_IP` y `HOST_PORT`: Dirección IP y puerto donde el servidor escuchara.
- `ENCODER`: Codificación utilizada para los mensajes (UTF-8).
- `BUFFER_SIZE`: Tamaño del buffer para recibir mensajes (1024 bytes).

3.2. Inicialización del Socket del Servidor

El método `startSocketServer()` crea y configura el socket del servidor:

Listing 2. Inicializacion del socket del servidor

```python
    def startSocketServer(self):
        try:
            # Crear un socket del lado del servidor usando IPV4 (AF_INET) y TCP (SOCK_STREAM)
            self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self.server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

            # Vincular el socket a la IP y puerto definidos
            self.server_socket.bind((self.HOST_IP, self.HOST_PORT))

            # Escuchar conexiones entrantes (hasta 10 en cola)
            self.server_socket.listen(10)
            print(f"Server listening on {self.HOST_IP}:{self.HOST_PORT}")
        except Exception as ex:
            print(f"Socket error: {ex}")
            if self.server_socket:
                self.server_socket.close()
                self.server_socket = None
            return

```

- `socket.AF_INET`: Especifica que se usara IPv4.
- `socket.SOCK_STREAM`: Indica que se usara el protocolo TCP (orientado a conexión).
- `setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)`: Permite reutilizar la direccion y puerto inmediatamente despues de cerrar el servidor.
- `bind()`: Asocia el socket a la direccion IP y puerto especificados.
- `listen(10)`: Pone al socket en modo escucha, aceptando hasta 10 conexiones en cola.

3.3. Manejo de Conexiones de Clientes

El metodo `connect_client()` es el ciclo principal del servidor que acepta nuevas conexiones:

Listing 3. Aceptacion de conexiones de clientes

```python
    def connect_client(self):
        while True:
            # Aceptar una conexion entrante
            client_socket, client_address = self.server_socket.accept()
            print(f"Connection established with {client_address}...")

            # Enviar "NAME" para pedir al cliente su nombre
            name_flag = "NAME"
            client_socket.send(name_flag.encode(self.ENCODER))
            client_name = client_socket.recv(self.BUFFER_SIZE).decode(self.ENCODER)

            # Agregar nuevo socket y nombre del cliente a las listas respectivas
            self.clients_sockets_list.append(client_socket)
            self.clients_names_list.append(client_name)

            # Notificar al servidor, al cliente individual y a todos los clientes
            print(f"Client '{client_name}' connected.")
```

```
                welcome_message = f"Welcome to the chat, {client_name}!\n"
                client_socket.send(welcome_message.encode(self.ENCODER))
                self.broadcast_message(f"{client_name} has joined the chat.\n")

                # Crear un hilo para recibir mensajes de este cliente
                recieve_message_thread = threading.Thread(target=self.recieve_message,
    args=(client_socket,))
                recieve_message_thread.start()

```

El flujo de aceptación de un cliente es:

1. El servidor acepta la conexión mediante `accept()`, que bloquea hasta que un cliente se conecta.
2. Se envía la señal "NAME" al cliente para solicitar su identificador.
3. Se recibe el nombre del cliente y se almacena en las listas.
4. Se envía un mensaje de bienvenida al cliente nuevo.
5. Se notifica a todos los clientes existentes sobre el nuevo participante.
6. Se crea un hilo dedicado para manejar los mensajes entrantes de este cliente.

3.4. Retransmisión de Mensajes (Broadcast)

El metodo `broadcast_message()` envía un mensaje a todos los clientes conectados:

Listing 4. Envio de mensajes a todos los clientes

```
        def broadcast_message(self, message):
            for client_socket in self.clients_sockets_list:
                try:
                    client_socket.send(message.encode(self.ENCODER))
                except Exception as ex:
                    print(f"Socket error: {ex}")
                    # Si hay error, remover al cliente
                    client_index = self.clients_sockets_list.index(client_socket)
                    client_name = self.clients_names_list[client_index]
                    self.clients_sockets_list.remove(client_socket)
                    self.clients_names_list.remove(client_name)
                    client_socket.close()
                    self.broadcast_message(f"{client_name} has left the chat.\n")

```

Este método itera sobre todos los sockets de clientes y envia el mensaje a cada uno. Si falla el envio a un cliente especifico, se asume que se desconecto y se procede a:

1. Remover su socket y nombre de las listas.
2. Cerrar su socket.
3. Notificar a los demas clientes sobre su salida.

3.5. Recepción de Mensajes de un Cliente

Cada cliente tiene un hilo dedicado que ejecuta `recieve_message()`:

Listing 5. Recepcion de mensajes de un cliente especifico

```
        def recieve_message(self, client_socket):
```

```
            while True:
                try:
                    message = client_socket.recv(self.BUFFER_SIZE).decode(self.ENCODER)
                    if message == "QUIT":
                        # Cliente solicita salir
                        client_index = self.clients_sockets_list.index(client_socket)
                        client_name = self.clients_names_list[client_index]
                        self.clients_sockets_list.remove(client_socket)
                        self.clients_names_list.remove(client_name)
                        client_socket.close()
                        self.broadcast_message(f"{client_name} has left the chat.\n")
                        break
                    else:
                        # Retransmitir mensaje a todos los clientes
                        client_index = self.clients_sockets_list.index(client_socket)
                        client_name = self.clients_names_list[client_index]
                        full_message = f"{client_name}: {message}"
                        print(full_message)
                        self.broadcast_message(full_message)
                except Exception as ex:
                    print(f"Socket error: {ex}")
                    # Manejo de desconexion inesperada
                    client_index = self.clients_sockets_list.index(client_socket)
                    client_name = self.clients_names_list[client_index]
                    self.clients_sockets_list.remove(client_socket)
                    self.clients_names_list.remove(client_name)
                    client_socket.close()
                    self.broadcast_message(f"{client_name} has left the chat.\n")
                    break

```

Este metodo se ejecuta en un bucle infinito dentro de un hilo dedicado por cliente:

- Recibe mensajes del cliente mediante `recv()`.
- Si el mensaje es "QUIT", procede a desconectar al cliente de manera ordenada.
- Si es un mensaje normal, lo formatea con el nombre del cliente y lo retransmite a todos.
- Captura excepciones para manejar desconexiones abruptas.

3.6. Hilos en el Servidor

El servidor utiliza hilos de dos maneras principales:

1. **Hilo principal**: Ejecuta `connect_client()` en un bucle infinito, aceptando nuevas conexiones.
2. **Hilos secundarios**: Uno por cada cliente conectado, ejecutando `recieve_message()` para manejar los mensajes entrantes de ese cliente especifico.

Esta arquitectura permite que el servidor:

- Acepte multiples clientes simultaneamente.
- Mantenga comunicación bidireccional con cada cliente.
- Sea escalable hasta el limite de recursos del sistema.
- Mantenga responsividad incluso con muchos clientes conectados.

4. Implementación del Cliente

4.1. Estructura de la Clase ChatClient

El cliente tambien se implementa como una clase orientada a objetos:

Listing 6. Constructor de la clase ChatClient

```
        class ChatClient:
            def __init__(self, host_ip, host_port=12345):
                self.client_socket = None
                self.client_name = None

                self.HOST_IP = host_ip
                self.HOST_PORT = host_port
                self.ENCODER = "utf-8"
                self.BUFFER_SIZE = 1024

```

- `client_socket`: Socket para la comunicación con el servidor.
- `client_name`: Nombre identificador del cliente en el chat.
- Los demas parámetros son consistentes con los del servidor para asegurar compatibilidad.

4.2. Conexión al Servidor

El método `start()` maneja la conexion inicial y la configuración de hilos:

Listing 7. Conexion inicial del cliente al servidor

```
        def start(self):
            try:
                self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                self.client_socket.connect((self.HOST_IP, self.HOST_PORT))

                # Handshake de nombre antes de lanzar hilos
                first_msg = self.client_socket.recv(self.BUFFER_SIZE).decode(self.ENCODER)
                if first_msg == "NAME":
                self.client_name = input("Enter your name: ")
                self.client_socket.send(self.client_name.encode(self.ENCODER))

                # Crear hilos para enviar y recibir mensajes concurrentemente
                recieve_thread = threading.Thread(target=self.recieve_message, daemon=True)
                send_thread = threading.Thread(target=self.send_message, daemon=True)
                recieve_thread.start()
                send_thread.start()

                recieve_thread.join()
                send_thread.join()
            except Exception as ex:
                print(f"Socket error: {ex}")
                if self.client_socket:
                self.client_socket.close()
                return
```

```
25
```

El proceso de conexión incluye:

1. Crear un socket y conectarse al servidor.
2. Realizar el "handshake" de nombre: recibir la senal "NAME", enviar el nombre.
3. Crear dos hilos: uno para recibir mensajes y otro para enviarlos.
4. Marcar los hilos como "daemon"para que terminen cuando el programa principal termine.
5. Esperar a que ambos hilos terminen (aunque en teoria se ejecutan indefinidamente).

4.3. Envio de Mensajes

El método `send_message()` se ejecuta en un hilo separado:

Listing 8. Envio de mensajes del cliente

```
    def send_message(self):
        while True:
            message = input()
            try:
                self.client_socket.send(message.encode(self.ENCODER))
            except Exception as ex:
                print(f"Socket error: {ex}")
                if self.client_socket:
                self.client_socket.close()
                break

```

- Lee entrada del usuario mediante `input()`.
- Envía el mensaje al servidor mediante el socket.
- Maneja excepciones por desconexión del servidor.

4.4. Recepción de Mensajes

El mótodo `recieve_message()` tambien se ejecuta en un hilo separado:

Listing 9. Recepcion de mensajes del cliente

```
    def recieve_message(self):
        while True:
            try:
                data = self.client_socket.recv(self.BUFFER_SIZE)
                if not data:
                    print("Servidor cerro la conexion.")
                    break
                message = data.decode(self.ENCODER)
                if message != "NAME":  # Ya manejado en start()
                    print(message)
            except Exception as ex:
                print(f"Socket error: {ex}")
                if self.client_socket:
                    self.client_socket.close()
                    break
```

16

- Recibe datos del servidor mediante `recv()`.
- Si no hay datos (socket cerrado), termina el bucle.
- Decodifica y muestra el mensaje (excepto la senal "NAME" ya procesada).
- Maneja excepciones por problemas de conexión.

4.5. Hilos en el Cliente

El cliente utiliza dos hilos principales:

1. **Hilo de recepción**: Escucha constantemente mensajes del servidor y los muestra al usuario. Este hilo evita que el programa se bloquee esperando mensajes entrantes.
2. **Hilo de envio**: Permite al usuario escribir mensajes en cualquier momento, independientemente de si esta recibiendo mensajes o no.

Esta separación es crucial para una experiencia de usuario fluida, ya que permite:

- Recibir mensajes en tiempo real mientras se escribe.
- No bloquear la interfaz durante operaciones de red.
- Mantener responsividad incluso con alta actividad en el chat.

5. Protocolo de Comunicación

El sistema implementa un protocolo simple pero efectivo:

1. **Conexión inicial**: El cliente se conecta al servidor.
2. **Handshake de nombre**:
   - Servidor ->Cliente: Envia "NAME"
   - Cliente ->Servidor: Envia nombre del usuario
   - Servidor ->Cliente: Envia mensaje de bienvenida personalizado
   - Servidor ->Todos: Anuncia la nueva conexion
3. **Comunicación normal**:
   - Cliente ->Servidor: Mensaje de texto
   - Servidor ->Todos: Retransmite mensaje con formato "Nombre: mensaje"
4. **Desconexión**:
   - Cliente ->Servidor: Envia "QUIT" para desconexión ordenada
   - O: Se detecta error de socket por desconexión abrupta
   - Servidor ->Todos: Anuncia la salida del usuario

6. Ventajas de la Implementación

- **Concurrencia**: El uso de hilos permite manejar multiples clientes eficientemente.
- **Modularidad**: La separación en clases facilita el mantenimiento y la extensión.
- **Robustez**: Manejo de excepciones para conexiones interrumpidas.
- **Escalabilidad**: La arquitectura permite anadir mas funcionalidades facilmente.

7. Ejecución del Chat Básico

Ejecutamos el servidor, donde podemos ver la IP del servidor y el puerto Fig 1.

```
PS C:\Users\Edgar Gomez\Desktop\Tareas\Tarea_Chat_Basico> python .\chat_server.py
Server listening on 192.168.0.127:12345
```

Fig. 1

Conectamos el primer cliente ingresando la IP del servidor y el puerto conjuntamente con el alias del cliente, es este caso "Luis", Fig 2.

```
PS C:\Users\Edgar Gomez\Desktop\Tareas\Tarea_Chat_Basico> python .\chat_client.py
Enter server IP address: 192.168.0.127
Enter server port: 12345
Enter your name: Luis
Welcome to the chat, Luis°
Luis has joined the chat.
```

Fig. 2

En este momento el servidor muestra todas las interacciones en el chat, muestra que "Luis" se ha conectado, Fig 3.

```
PS C:\Users\Edgar Gomez\Desktop\Tareas\Tarea_Chat_Basico> python .\chat_server.py
Server listening on 192.168.0.127:12345
Connection established with ('192.168.0.127', 64562)...
Client 'Luis' connected
```

Fig. 3

Conectamos el segundo cliente ingresando la IP del servidor y el puerto conjuntamente con el alias del cliente, es este caso "Pedro", Fig 4.

```
PS C:\Users\Edgar Gomez\Desktop\Tareas\Tarea_Chat_Basico> python .\chat_client.py
Enter server IP address: 192.168.0.127
Enter server port: 12345
Enter your name: Pedro
Welcome to the chat, Pedro°
Pedro has joined the chat.
```

Fig. 4

Ahora "Luis" ve la notificacion de que "Pedro" se ha unido al chat, Fig 5.

```
PS C:\Users\Edgar Gomez\Desktop\Tareas\Tarea_Chat_Basico> python .\chat_client.py
Enter server IP address: 192.168.0.127
Enter server port: 12345
Enter your name: Luis
Welcome to the chat, Luis°
Luis has joined the chat.

Pedro has joined the chat.
```

Fig. 5

Conectamos el tercer cliente ingresando la IP del servidor y el puerto conjuntamente con el alias del cliente, es este caso "Ana", Fig 6.

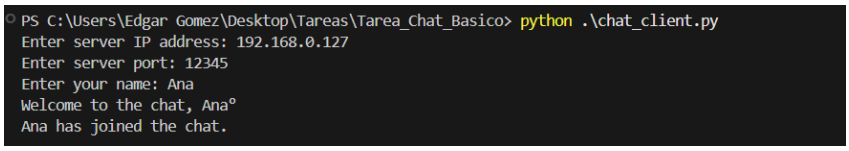

Fig. 6

Ahora "Luis" envia un mensaje, Fig 7.

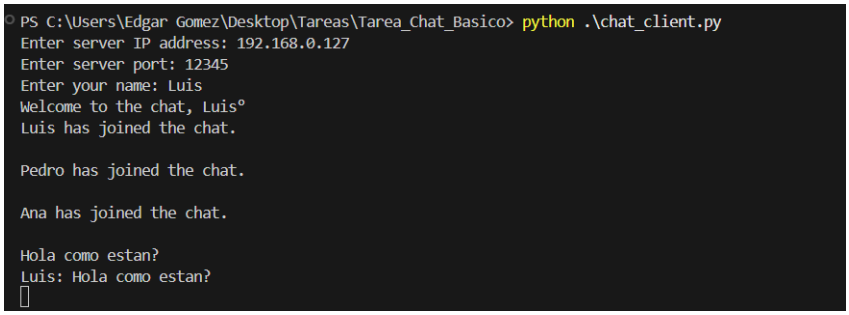

Fig. 7

El mensaje es recibido por "Pedro", Fig 8.

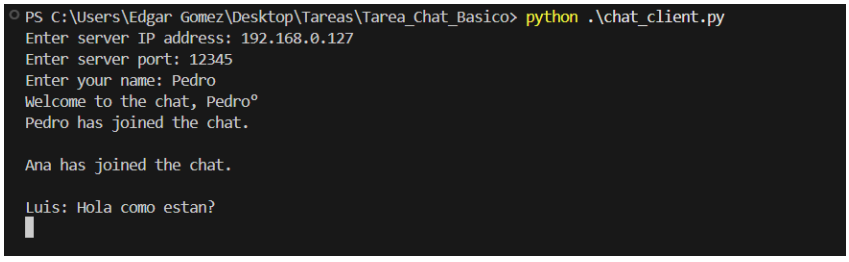

Fig. 8

El mensaje es recibido por "Ana", Fig 9.

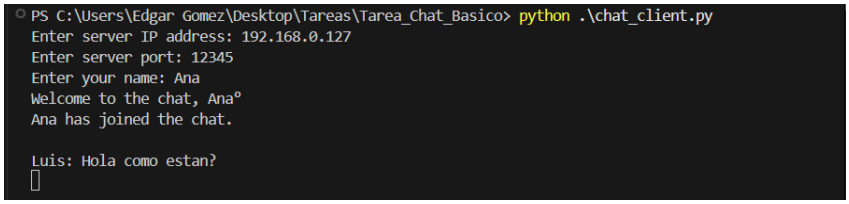

Fig. 9

Ya establecido se desarrolla una conversacion mas larga donde podemos ver desde la perspectiva de "Luis". Fig 10.

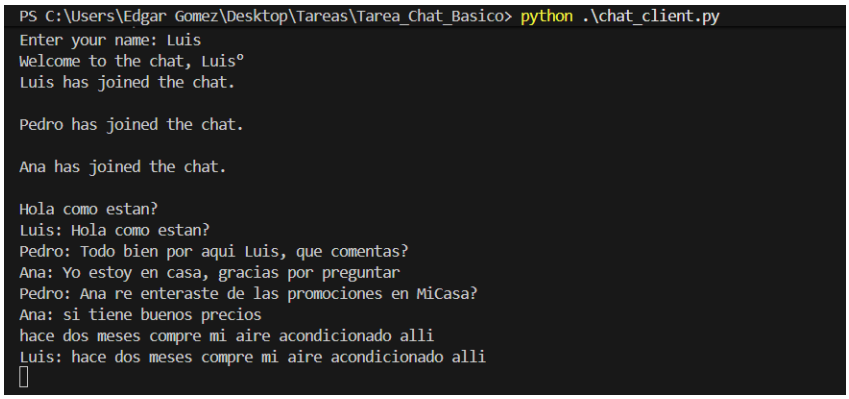

Fig. 10

Chat desde la perspectiva de "Pedro", Fig 11.

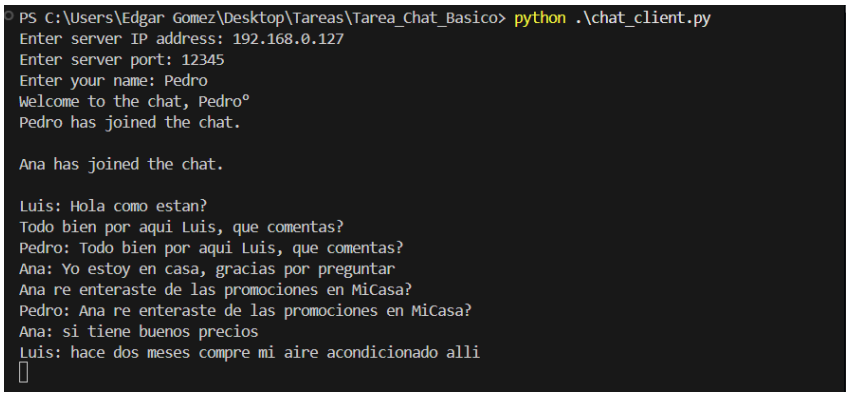

Fig. 11

Chat desde la perspectiva de "Ana", Fig 12.

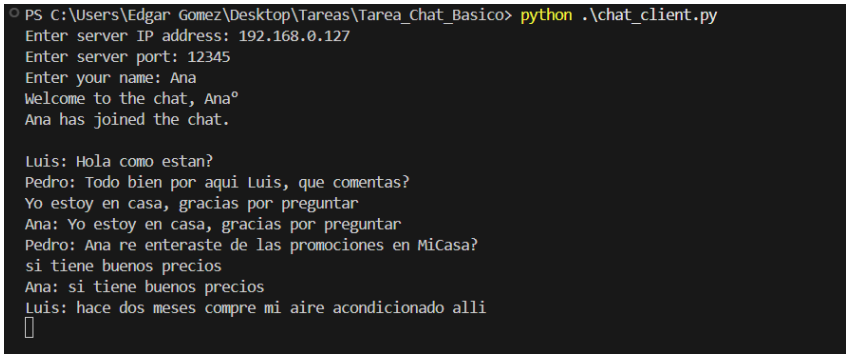

Fig. 12

Si un usuario quiere abandonar el chat, debe escribir la palabra "QUIT", por ejemplo "Ana", Fig 13.

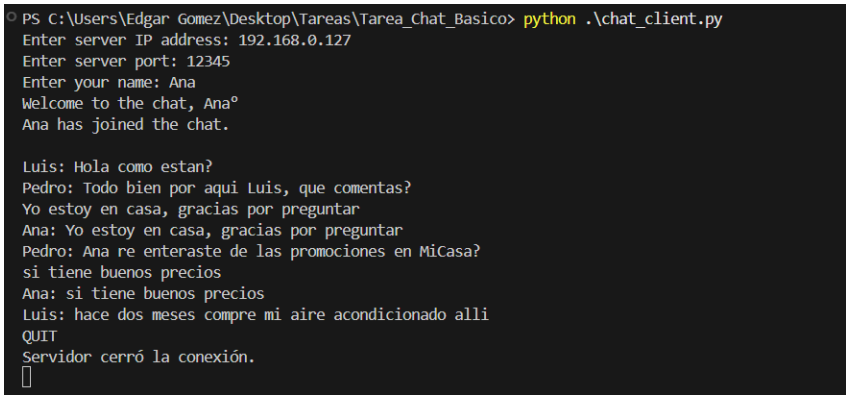

Fig. 13

Los demás usuarios son notificados de que un usuario abandono el chat, Fig 14.

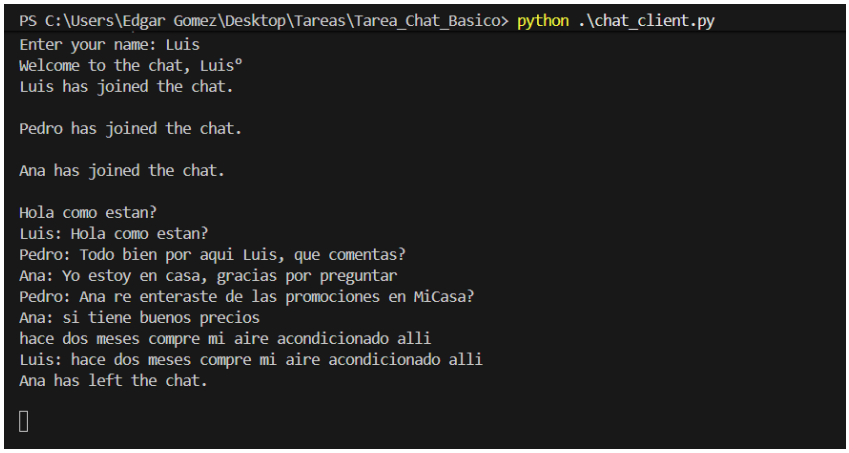

Fig. 14

8. Link Repositorio GitHub

En el siguiente enlace puede encontrar el código completo de la implementación del chat básico en Python.

Repositorio: https://github.com/aurquiel/chatbasicopython

9. Conclusión

Esta implementación demuestra los conceptos fundamentales de los sistemas distribuidos en un contexto práctico. El modelo cliente-servidor con sockets TCP proporciona una base solida para aplicaciones de comunicación en red. El uso de hilos permite concurrencia tanto en el servidor (para manejar multiples clientes) como en el cliente (para tener un UI responsiva). Aunque es una implementación básica, ilustra claramente los principios arquitectonicos que subyacen a sistemas distribuidos y la comunicación que implementan.