Chat Cliente-Servidor con Python

Este proyecto formativo te guiará en la implementación de un sistema de chat básico utilizando sockets TCP/IP y base de datos SQLite. Aprenderás a crear tanto el servidor que gestiona las conexiones como el cliente que envía mensajes, aplicando buenas prácticas de programación y manejo de errores.

El objetivo principal es dominar la comunicación en red mediante sockets, la persistencia de datos con SQLite y la modularización del código para crear aplicaciones robustas y escalables.

Contexto Académico

- **Detalle del Trabajo Práctico Materia: Programación sobre Redes**
- **Tema: Programación Concurrente Año: 2025 - 2° cuatrimestre**
- **consigna y entrega: Práctica Formativa Obligatoria (PFO) 1**
- **Alumno: Coria Daniel**
- **Docente: Germán Ríos**

El sistema implementa una arquitectura cliente-servidor tradicional donde múltiples clientes pueden conectarse simultáneamente al servidor. El servidor escucha en el puerto 5000 y procesa cada mensaje recibido, almacenándolo en una base de datos SQLite antes de enviar una confirmación al cliente.

Esta arquitectura permite la escalabilidad y el mantenimiento independiente de cada componente del sistema.

Configuración del Entorno

Requisitos del Sistema

- Python 3.7 o superior
- Módulo socket (incluido en Python)
- Módulo sqlite3 (incluido en Python)
- Terminal o IDE para desarrollo

Estructura del Proyecto

- `servidor.py` - Lógica del servidor
- `cliente.py` - Aplicación cliente
- `chat.db` - Base de datos SQLite

Asegúrate de tener Python instalado correctamente y un entorno de desarrollo configurado. La estructura modular del proyecto facilita el mantenimiento y la comprensión del código.

Estructura de la Base de Datos

Campo	Tipo	Descripción	Ejemplo
id	INTEGER PRIMARY KEY	Identificador único	1
contenido	TEXT NOT NULL	Texto del mensaje	"Hola servidor"
fecha_envio	TIMESTAMP	Momento del envío	2024-04-25 14:30:15
ip_cliente	TEXT	Dirección IP origen	127.0.0.1

La tabla mensajes almacena toda la información necesaria para el seguimiento completo de las comunicaciones. El campo `id` se incrementa automáticamente, mientras que `fecha_envio` registra el timestamp exacto del mensaje.

Funciones del Servidor- Parte 1

```python
import socket
import sqlite3
import datetime

# -------------------- FUNCIONES --------------------

def inicializar_db():
    """Crea la base de datos y la tabla si no existen."""
    conn = sqlite3.connect("chat.db")
    cursor = conn.cursor()
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS mensajes (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            contenido TEXT NOT NULL,
            fecha_envio TEXT NOT NULL,
            ip_cliente TEXT NOT NULL
        )
    """)
    conn.commit()
    conn.close()

def guardar_mensaje(contenido, ip_cliente):
    """Guarda un mensaje en la base de datos."""
    conn = sqlite3.connect("chat.db")
    cursor = conn.cursor()
    fecha = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    cursor.execute("INSERT INTO mensajes (contenido, fecha_envio, ip_cliente) VALUES (?, ?, ?)",
                   (contenido, fecha, ip_cliente))
    conn.commit()
    conn.close()
    return fecha
```

Resumen Explicativo – Funciones del Servidor (Parte 1)

- **Inicialización de la BD:** crea el archivo `chat.db` y la tabla `mensajes` si no existen.
- **Columnas de la tabla:** `id` (número único), `contenido` (texto del mensaje), `fecha_envio` (fecha y hora exactas), `ip_cliente` (IP del emisor).
- **Guardado de mensajes:** cada mensaje que llega se inserta en la base de datos junto con la fecha y la IP.
- **Confirmación:** `commit()` asegura que la información quede grabada.

Funciones del Servidor- Parte 2

```python
def inicializar_socket():
    """Configura el socket del servidor."""
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(("localhost", 5000))
    server_socket.listen(5)
    print("Servidor escuchando en localhost:5000...")
    return server_socket

def manejar_cliente(client_socket, client_address):
    """Recibe mensajes del cliente y los guarda en la DB."""
    while True:
        data = client_socket.recv(1024)
        if not data:
            break

        mensaje = data.decode().strip()
        if mensaje.lower() == "éxito":
            print(f"Cliente {client_address} finalizó la conexión.")
            break

        fecha = guardar_mensaje(mensaje, client_address[0])
        respuesta = f"Mensaje recibido: {fecha}"
        client_socket.send(respuesta.encode())
        print(f"[{fecha}] {client_address[0]} dijo: {mensaje}")

    client_socket.close()
```

Resumen Explicativo – Funciones del Servidor (Parte 2)

- **Inicialización del socket:**
  - Crea el servidor en `localhost` y puerto `5000`.
  - Lo deja en modo “escucha” para aceptar clientes.
- **Manejo de clientes:**
  - Recibe datos enviados por el cliente.
  - Si el mensaje es `"éxito"`, cierra la conexión.
  - Guarda el mensaje en la base de datos con su fecha e IP.
  - Responde al cliente confirmando la recepción.
  - Muestra en consola qué cliente envió el mensaje y cuándo.

Funciones del Servidor- Parte 3

```python
# -------------------- MAIN --------------------
if __name__ == "__main__":
    try:
        inicializar_db()
        server_socket = inicializar_socket()
        while True:
            client_socket, client_address = server_socket.accept()
            print(f"Conexión establecida con {client_address}")
            manejar_cliente(client_socket, client_address)
    except OSError as e:
        print(f"Error de socket: {e}")
    except sqlite3.Error as e:
        print(f"Error en la base de datos: {e}")
```

Resumen Explicativo – Funciones del Servidor (Parte 3)

Este código inicia la base de datos y un servidor que espera conexiones de clientes. Por cada cliente conectado, imprime su dirección y ejecuta la función que maneja la comunicación. Además, captura errores de socket y de la base de datos para evitar que el programa falle.

Implementación del Cliente

Características del Cliente

El cliente debe establecer una conexión persistente con el servidor y permitir el envío de múltiples mensajes hasta que el usuario escriba "éxito" para terminar la sesión.

- Conexión automática a localhost:5000
- Interfaz de línea de comandos intuitiva
- Visualización de respuestas del servidor
- Manejo de errores de conexión
- Cierre elegante de la conexión

La aplicación cliente mantiene un bucle continuo que solicita mensajes al usuario, los envía al servidor y muestra las confirmaciones recibidas.

```
import socket

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

client_socket.connect(("localhost", 5000))

print("Conectado al servidor. Escribe 'éxito' para salir.\n")

while True:

    mensaje = input("Escribe un mensaje: ")

   client_socket.send(mensaje.encode())

if mensaje.lower() == "exito":

        print("Cerrando conexión...")

        break

 respuesta = client_socket.recv(1024)

    print(f"Servidor -> {respuesta.decode()}")

client_socket.close()
```

Pruebas y Ejecución

Paso 1: Ejecutar el Servidor

1

Abrir una terminal y ejecutar `python servidor.py`. Verificar que el servidor esté escuchando en el puerto 5000.

Paso 2: Iniciar el Cliente

2

En una segunda terminal, ejecutar `python cliente.py` para establecer la conexión con el servidor.

Paso 3: Enviar Mensajes

3

Escribir mensajes de prueba y verificar que el servidor responda con "Mensaje recibido: [contenido]".

Paso 4: Verificar Base de Datos

4

Comprobar que los mensajes se almacenan correctamente en la tabla SQLite con todos los campos.

Paso 5: Finalizar Sesión

5

Escribir "éxito" en el cliente para terminar la conexión de forma elegante.

Buenas Prácticas y Comentarios

Comentarios Descriptivos

Incluye comentarios explicativos en cada sección clave del código. Por ejemplo: `# Configuración del socket TCP/IP` antes de la inicialización del socket.

Manejo de Excepciones

Implementa bloques try-except para capturar errores comunes como puertos ocupados, fallos de base de datos y desconexiones inesperadas.

Modularización del Código

Separa la funcionalidad en funciones específicas para mejorar la legibilidad, facilitar el testing y permitir la reutilización del código.

Logging y Depuración

Utiliza mensajes informativos para monitorear el estado del servidor y facilitar la identificación de problemas durante la ejecución.

Entrega del Proyecto

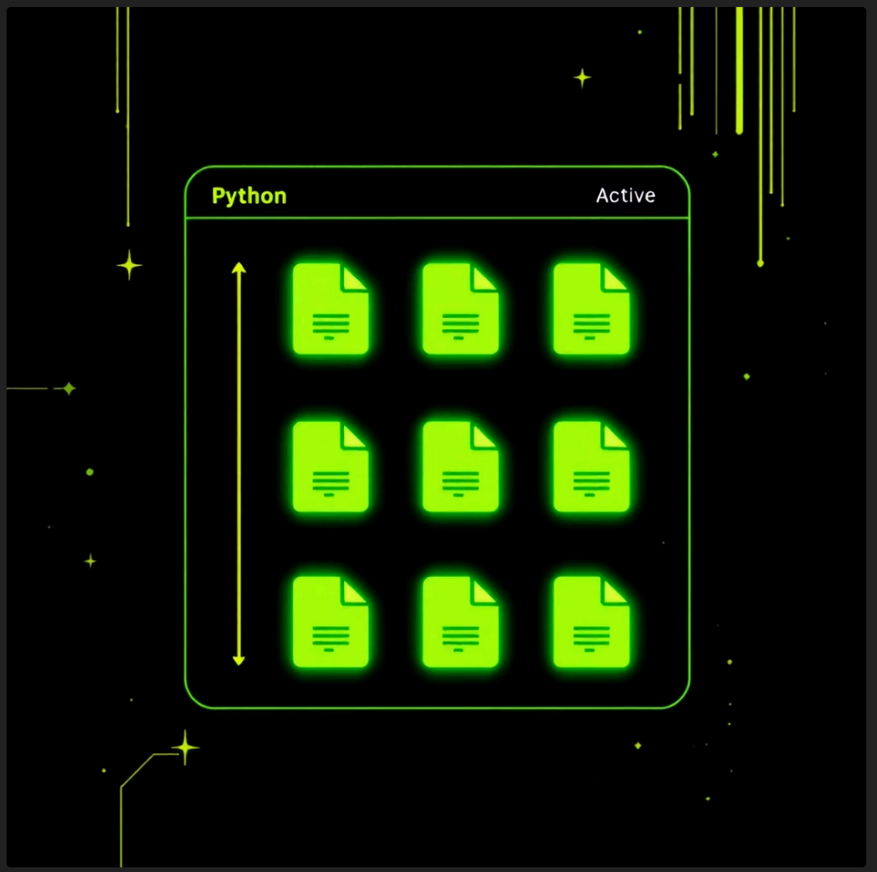
Fechas Importantes

Entrega: 11 de septiembre 2025

Modalidades de Entrega

1. **Repositorio en línea:** Enlace a GitHub: https://github.com/santysanty/chat_basico_Python_PFO1.git
2. **Archivo comprimido:** Subir un .zip o .rar con todos los archivos del proyecto

Asegúrate de incluir un archivo `README.md` con instrucciones de instalación y ejecución, así como documentación adicional sobre el funcionamiento del sistema.



Checklist de Entrega

- ✓ Código del servidor comentado
- ✓ Código del cliente funcional
- ✓ Base de datos SQLite configurada
- ✓ Pruebas de funcionamiento
- ✓ Documentación completa