**Eje 4: Árboles binarios con socket en Python.**

Melqui A. Romero, Esteban Villada Henao y Karolaine Z. Villero

Fundación Universitaria del Área Andina

Facultad De Ingeniería y Ciencias Básicas, Ingeniería en Sistemas, Modelos de Programación II

Ing. Deivys Morales

30 de marzo del 2025

**Eje 4: Árboles binarios con socket en Python.**

En la actualidad, la comunicación en red y las estructuras de datos juegan un papel fundamental en el desarrollo de aplicaciones distribuidas. Un árbol binario es una estructura de datos jerárquica que permite organizar la información de manera eficiente. Por otro lado, los sockets permiten la comunicación entre dispositivos a través de redes. Este trabajo explora la implementación de árboles binarios mediante sockets para optimizar el manejo de datos en aplicaciones distribuidas.

En este documento, por medio de un ejemplo realizado en Python, observaremos como un Árbol binario con stocket trabaja.

# Objetivos

## Objetivo General

Desarrollar una aplicación que implemente la estructura de árboles binarios utilizando sockets para la transmisión y manipulación de datos en una red.

## 

## Objetivos Específicos

1. Investigar la teoría de árboles binarios y su aplicación en sistemas distribuidos.
2. Analizar el funcionamiento de los sockets como medio de comunicación entre procesos en red.
3. Diseñar un sistema de comunicación basado en sockets que permita la inserción, búsqueda y eliminación de nodos en un árbol binario.
4. Implementar una aplicación en un lenguaje de programación adecuado para la gestión de árboles binarios mediante sockets.
5. Evaluar el rendimiento de la aplicación en términos de eficiencia y seguridad de transmisión de datos.

**Tabla de contenido**

[Tablas de Figuras 5](#_Toc194259614)

[1. Objetivo General 6](#_Toc194259615)

[2. Objetivos Específicos 6](#_Toc194259616)

[3. Árboles y Stocket. 7](#_Toc194259617)

[3.1 Árboles Binarios 7](#_Toc194259618)

[3.2 Sockets 9](#_Toc194259619)

[2. Estructura general del código 11](#_Toc194259620)

[2. 1 Descripción 11](#_Toc194259621)

[Desplegable. 12](#_Toc194259622)

[1. Arquitectura del Sistema 12](#_Toc194259623)

[a) Interfaz Web y Comunicación HTTP 12](#_Toc194259624)

[b) Módulo de Traducción y Comunicación por REST 13](#_Toc194259625)

[2. Flujo de Datos y Procesos 14](#_Toc194259626)

[1. Ingreso del Mensaje: 15](#_Toc194259627)

[2. Recepción en el Servidor HTTP: 15](#_Toc194259628)

[3. Invocación del Traductor: 16](#_Toc194259629)

[4. Respuesta y Actualización del Chat: 17](#_Toc194259630)

[5. Despliegue en Railway: 18](#_Toc194259631)

[Local: 18](#_Toc194259632)

[1. Arquitectura del Sistema 19](#_Toc194259633)

[2. Componentes del sistema 19](#_Toc194259634)

[Cliente (cliente\_main.py y cliente.py) 19](#_Toc194259635)

[Servidor (server.py) 21](#_Toc194259636)

[Manejo de Clientes (manejador.py) 22](#_Toc194259637)

[Árbol Binario (arbol\_binario.py) 23](#_Toc194259638)

[Pruebas y resultados 25](#_Toc194259639)

[Ejemplo de Entrada y Salida 25](#_Toc194259640)

[3.3. Video explicativo 25](#_Toc194259641)

[Conclusiones 26](#_Toc194259642)

[Referencias 27](#_Toc194259643)

Tablas de Figuras

[Figura 1 Código Arbol Binario Part. 1 8](#_Toc194259644)

[Figura 2 Código Árbol Binario Par. 2 9](#_Toc194259645)

[Figura 3 Código Árbol Binario Part. 3 10](#_Toc194259646)

[Figura 4 Código Estructura Socket 11](#_Toc194259647)

[Figura 5 Menú Principal del Proyecto 12](#_Toc194259648)

[Figura 6 Estructura del código fuente desplegado en railway 14](#_Toc194259649)

[Figura 7 Visualización de las peticiones hechas sobre magic loop sobre la traducción 15](#_Toc194259650)

[Figura 8 Visualización del frontend del chat que invocara las peticiones 16](#_Toc194259651)

[Figura 9 Definición del servidor por el puerto 5000 17](#_Toc194259652)

[Figura 10 Definición del path /chat sobre metodo POST para realizar la petición de traducir 17](#_Toc194259653)

[Figura 11 Definición de la función traducir\_texto para realizar la petición en magicloops 18](#_Toc194259654)

[El código JavaScript en el cliente recibe esta respuesta y añade el mensaje traducido al área de chat. *Figura 12 Petición traducida y visualizada dentro del frontend* 18](#_Toc194259655)

[Figura 13 Estructura de Árbol Binario en Cliente.py 21](#_Toc194259656)

[Figura 14 Estructura Socket Server.py 22](#_Toc194259657)

[Figura 15 Conexión e inicio del Árbol Binario Part. 1 23](#_Toc194259658)

[Figura 16 Árbol binario manejador.py 24](#_Toc194259659)

[Figura 17 reglas del árbol binario 25](#_Toc194259660)

1. Árboles y Stocket.

3.1 Árboles Binarios

Un árbol binario es una estructura de datos en la que cada nodo puede tener como máximo dos hijos. Se utiliza en diversas aplicaciones como búsqueda, ordenamiento y almacenamiento jerárquico de datos.

Figura 1 Código Arbol Binario Part. 1



Figura 2 Código Árbol Binario Par. 2

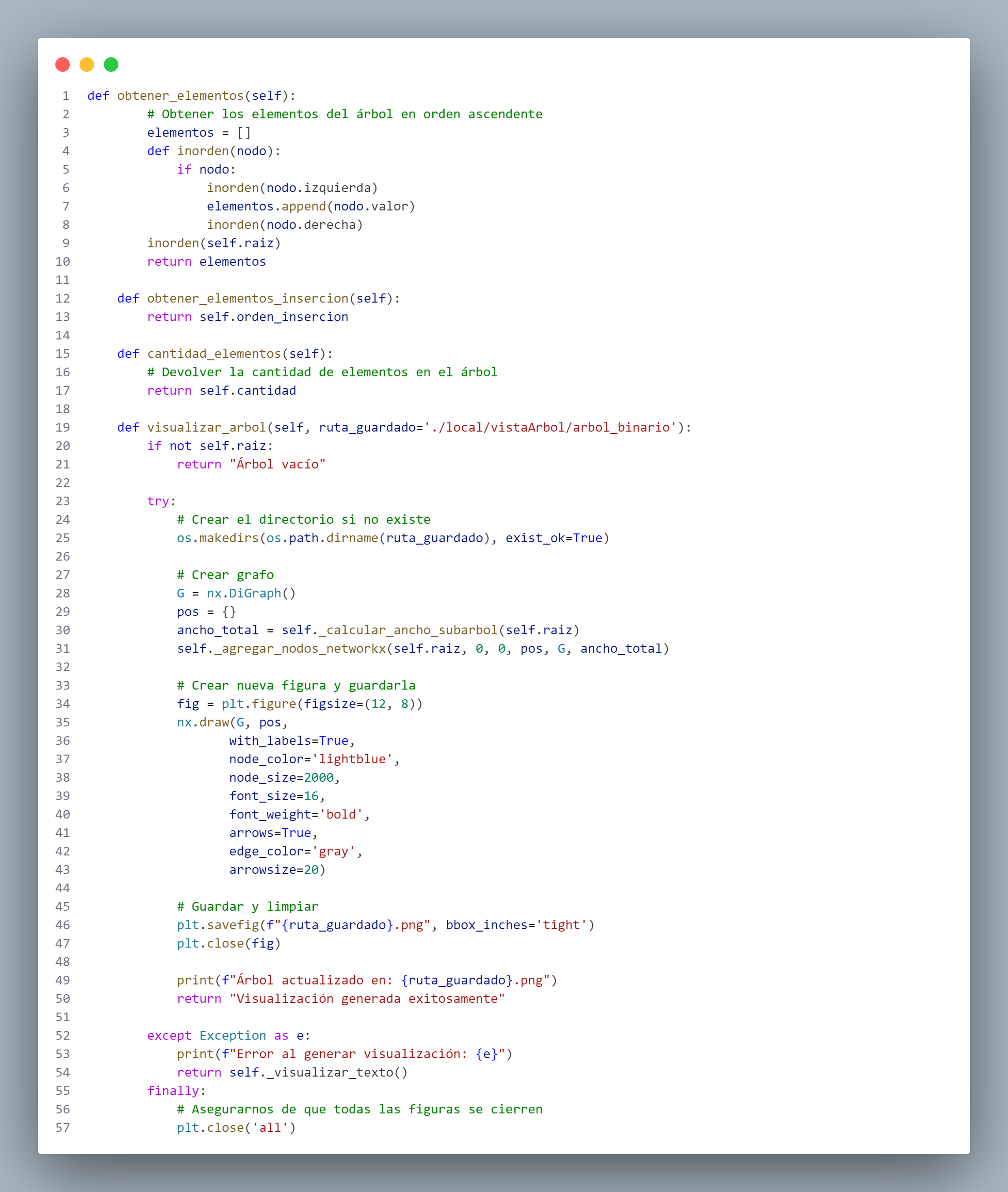


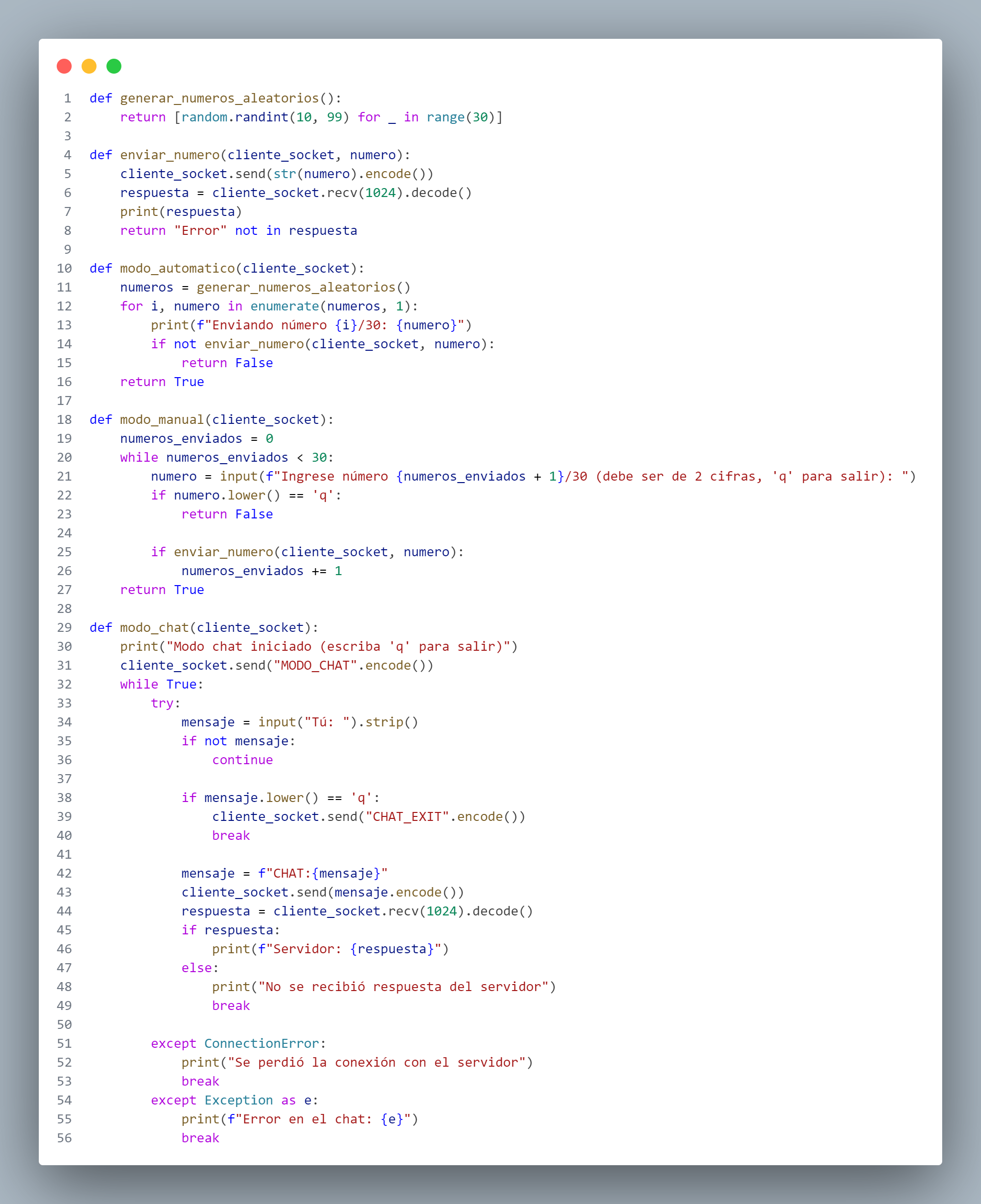
Figura 3 Código Árbol Binario Part. 3



3.2 Sockets

Los sockets son puntos de comunicación que permiten la transmisión de datos entre procesos a través de redes. Se dividen en sockets de flujo (TCP) y datagrama (UDP), cada uno con características específicas de confiabilidad y velocidad.

Figura 4 Código Estructura Socket

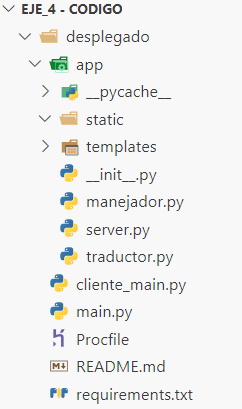
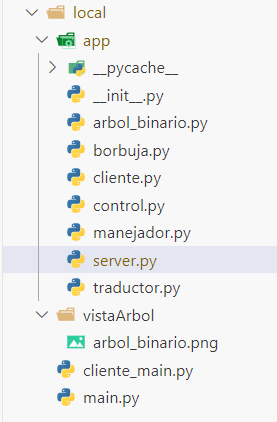


2. Estructura general del código

2. 1 Descripción

Para lograr establecer una comunicación entre el cliente y el servidor que permita por parte del cliente enviar 30 datos numéricos enteros de 2 cifras, uno a uno al servidor y este vaya construyendo un árbol binario con los datos que le envía el cliente a través de la comunicación por medio de socket, creamos un programa que nos permita realizarlo tanto de manera automática, como manual.  
Está conformado por dos partes (Desplegables y Local) las cuales mostramos a continuación:

Figura 5 Menú Principal del Proyecto

Desplegable.

El proyecto desplegado es un sistema de chat, que traduce los mensajes el cual se despliega en Railway. La idea principal es que el usuario ingresa un mensaje a través de una interfaz web (definida en el archivo .**/templates/index.html**) y el servidor responde con una traducción del mensaje. La comunicación entre el cliente y el servidor se realiza por medio de peticiones HTTP, mientras que internamente el proceso de traducción se apoya en la comunicación por *REST*

1. Arquitectura del Sistema

El sistema se puede dividir en dos grandes bloques:

a) Interfaz Web y Comunicación HTTP

• **Cliente (Front-End):**

La interfaz web (archivo **./templates/index.html**) cuenta con un diseño de chat donde el usuario escribe su mensaje. Al presionar el botón de “Enviar”, se ejecuta un código JavaScript que:

• Agrega el mensaje del usuario al área de chat.

• Realiza una petición fetch con método **POST** a la ruta /chat enviando el mensaje en formato JSON.

• Recibe la respuesta del servidor (el mensaje traducido) y lo muestra en la ventana de chat.

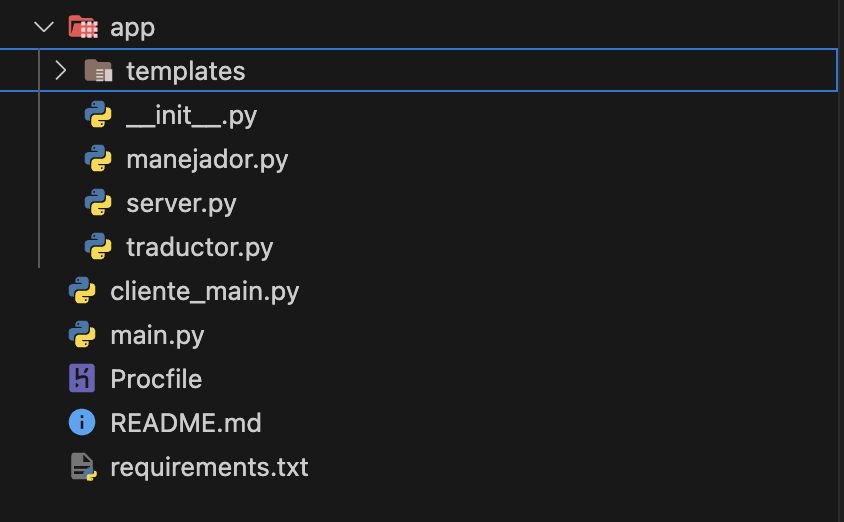
• **Servidor HTTP (Back-End):**

El archivo **main.py** (junto a otros archivos como **server.py** o **manejador.py**) conforma el núcleo del servidor. Utilizando el framework **Flask**:

• Se define la ruta /chat que recibe las peticiones del cliente.

• Una vez recibido el mensaje, se invoca el módulo encargado de la traducción.

Figura 6 Estructura del código fuente desplegado en railway



Nota: Elaboración propia

b) Módulo de Traducción y Comunicación por REST

**• Módulo Traductor:**

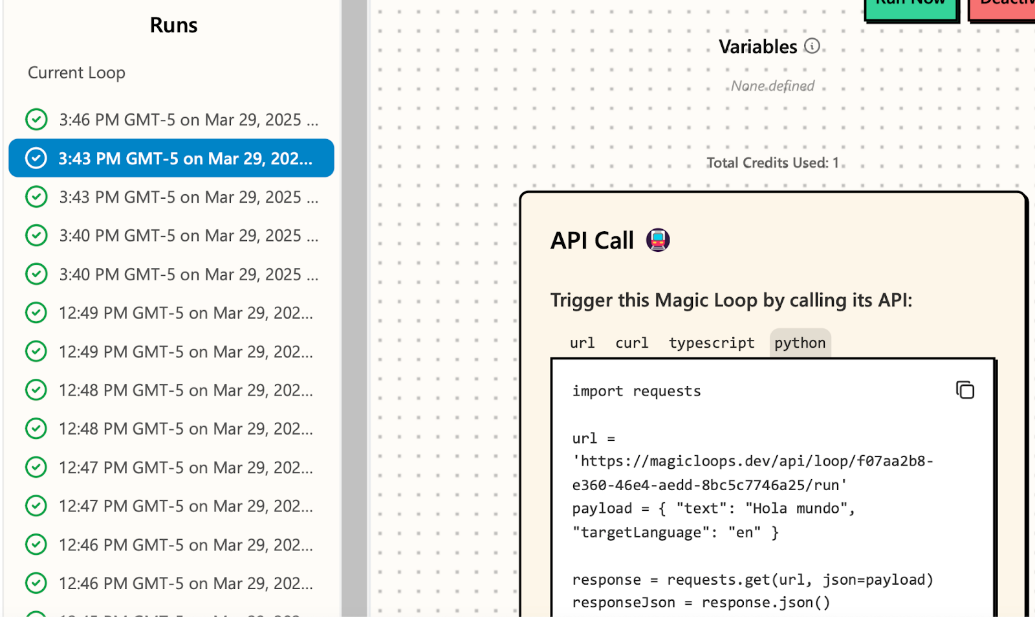
El archivo **traductor.py** contiene la lógica para traducir el mensaje. Aquí es donde se implementa la función que, mediante el uso del API, se comunica con un servicio de traducción.

De esta forma, la aplicación Flask actúa como intermediaria: recibe la solicitud web, invoca al traductor que se comunica vía API con el servicio de traducción, y devuelve la respuesta traducida al cliente web.

**Consumo de API Pública de Magic Loops:**

Además de gestionar la comunicación mediante API, el módulo encargado de la traducción (implementado en **traductor.py**) realiza una petición HTTP a la API pública de Magic Loops. Esta API se invoca utilizando la librería requests y se le envía el texto original junto con el código del idioma destino en formato JSON. La respuesta de la API incluye el texto traducido, que se extrae y se envía de vuelta al cliente.

Figura 7 Visualización de las peticiones hechas sobre (magic loop, s.f.) sobre la traducción



Nota: Elaboración propia

2. Flujo de Datos y Procesos

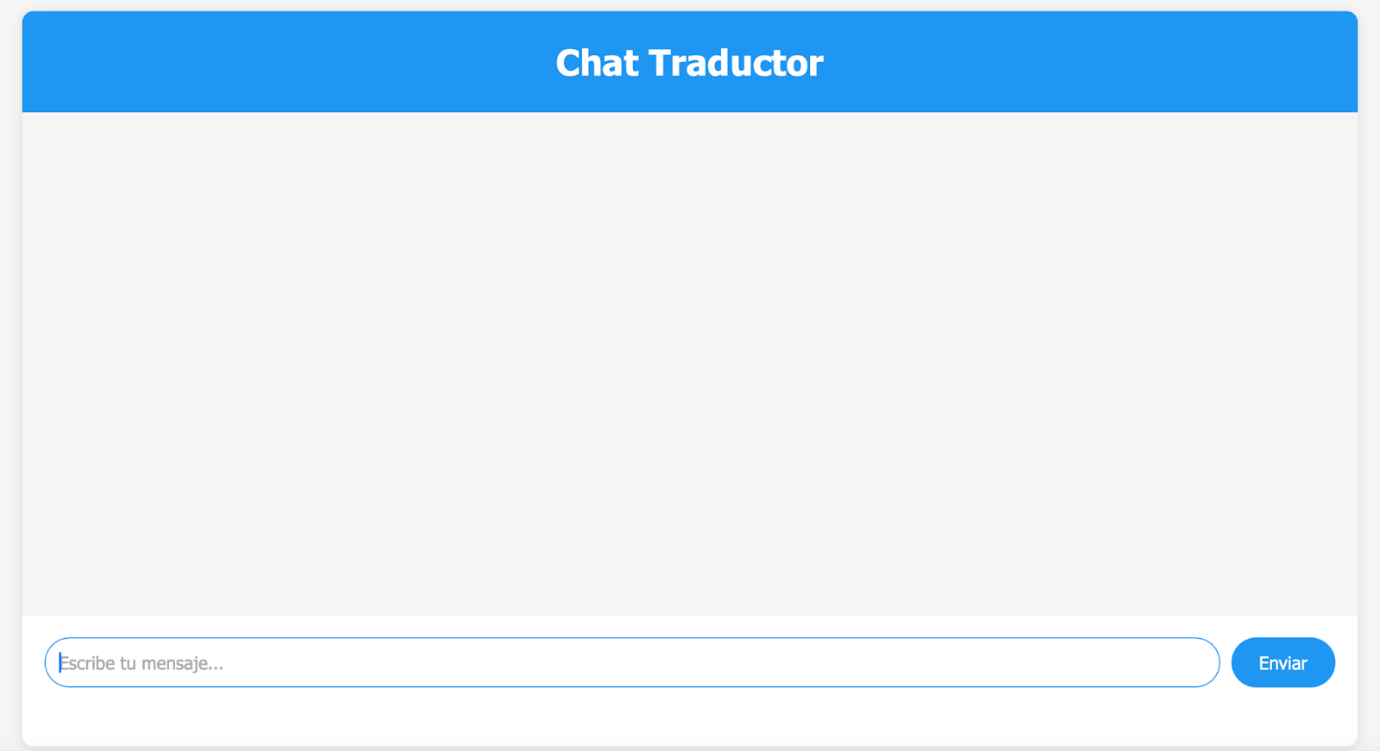
El proceso completo se puede describir en los siguientes pasos:

1. Ingreso del Mensaje:

El usuario escribe un mensaje en la interfaz web y hace clic en “Enviar”.

*(Código en* ***index.html*** *con JavaScript que captura el evento click y usa fetch para enviar la petición a /chat.)*

Figura 8 Visualización del frontend del chat que invocara las peticiones

**

Nota: Elaboración propia

2. Recepción en el Servidor HTTP:

La aplicación Flask (configurada en **main.py**) recibe la petición POST en el endpoint /chat.

Se extrae el mensaje enviado por el cliente.

Figura 9 Definición del servidor por el puerto 5000

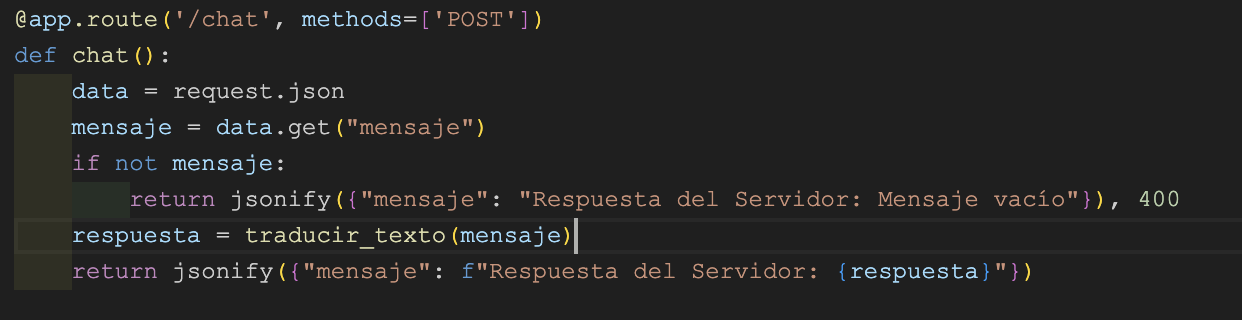


Nota: Elaboración propia

3. Invocación del Traductor:

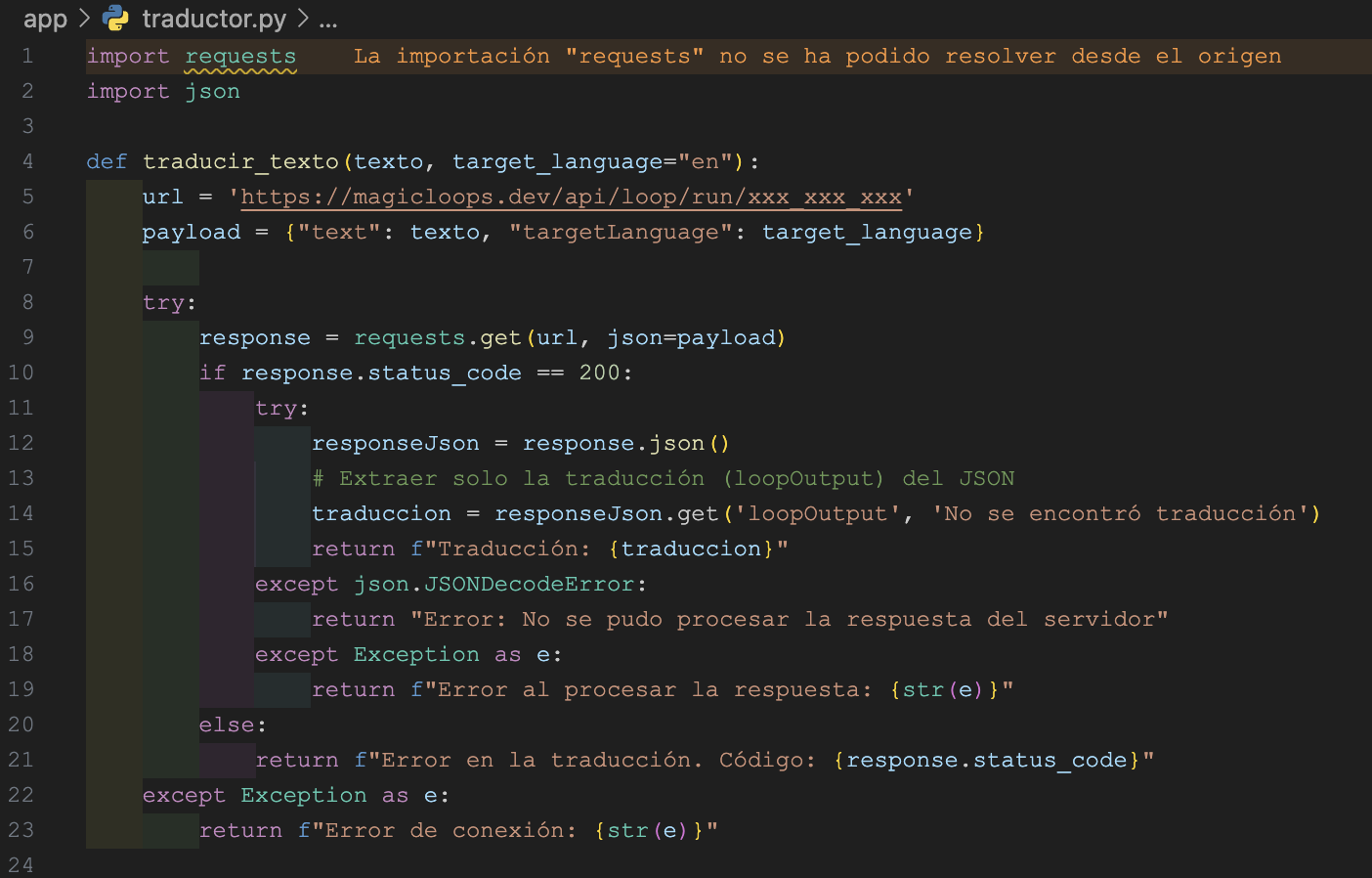
El servidor llama a la función del módulo traductor.py que se encarga de la traducción.

Figura 10 Definición del path /chat sobre metodo POST para realizar la petición de traducir



Nota: Elaboración propia

Figura 11 Definición de la función traducir\_texto para realizar la petición en magicloops

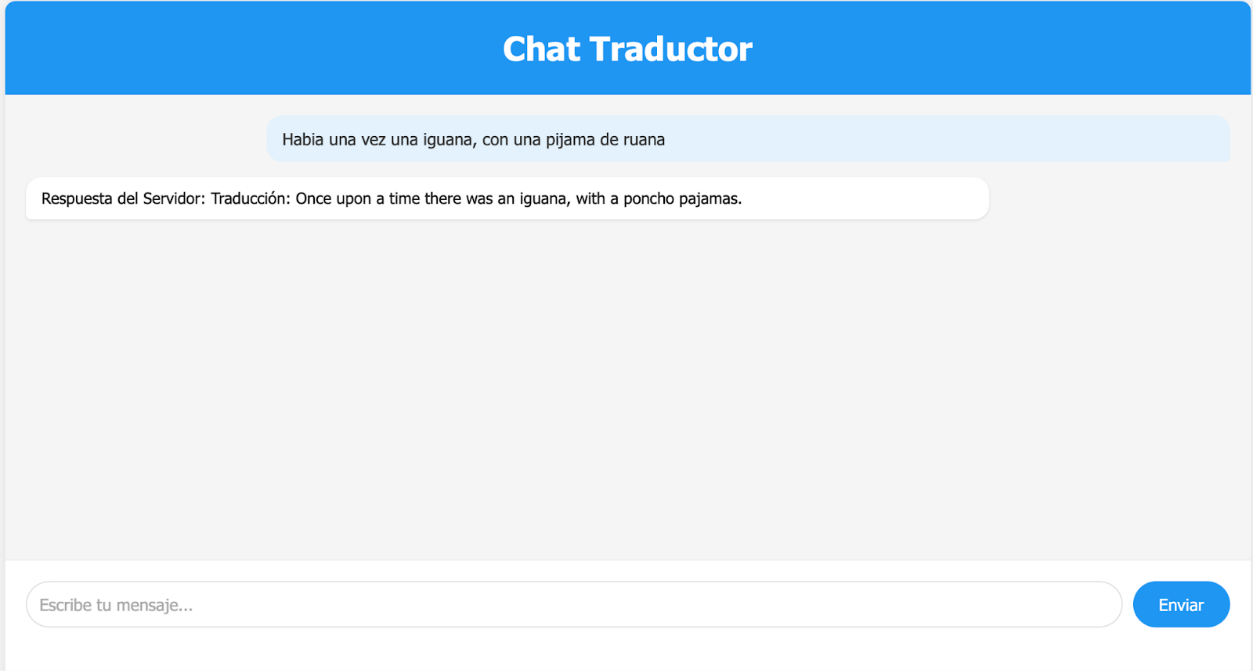


Nota: Elaboración propia

4. Respuesta y Actualización del Chat:

Una vez recibido el mensaje traducido, el servidor HTTP lo envía en formato JSON como respuesta a la petición.

El código JavaScript en el cliente recibe esta respuesta y añade el mensaje traducido al área de chat.  
*Figura 12 Petición traducida y visualizada dentro del frontend*



Nota: Elaboración propia

5. Despliegue en Railway:

El archivo **Procfile** junto con **requirements.txt** y la estructura del proyecto permiten desplegar el sistema en Railway.

Railway utiliza el comando especificado en el **Procfile** (por ejemplo, usando **gunicorn**) para ejecutar la aplicación Flask en producción.

Resultados:

Podemos ver que toda la aplicación del chat “Desplegable” que integra un traductor se puede encontrar públicamente desplegada en <https://web-production-b5a88.up.railway.app/>. La funcionalidad se divide en:

• **Front-End:** Interfaz web en index.html con JavaScript para gestionar el chat.

• **Back-End HTTP:** Aplicación Flask (en main.py) que recibe mensajes, invoca la traducción y envía respuestas.

• **Servicio de Traducción:** Módulo traductor.py que implementa la lógica de traducción y se comunica mediante API con un servicio dedicado, cuya gestión se encuentra en server.py y manejador.py.

• **Despliegue**: Configurado para ejecutarse en Railway mediante Procfile y *requirements.txt*.

Local:

El programa "local" implementa una arquitectura cliente-servidor donde el cliente envía datos numéricos al servidor a través de sockets, y el servidor construye un árbol binario con esos datos. Esta comunicación ocurre en una red local y no en un entorno web, a diferencia del sistema de chat basado en Flask.

1. Arquitectura del Sistema

El sistema sigue una arquitectura cliente-servidor, donde:

* **El cliente** envía números enteros de dos cifras uno por uno al servidor.
* **El servidor** recibe los datos y los almacena en un **Árbol Binario de Búsqueda (ABB)**.
* **Ambos se comunican a través de sockets TCP/IP**, asegurando la transmisión fiable de datos.

2. Componentes del sistema

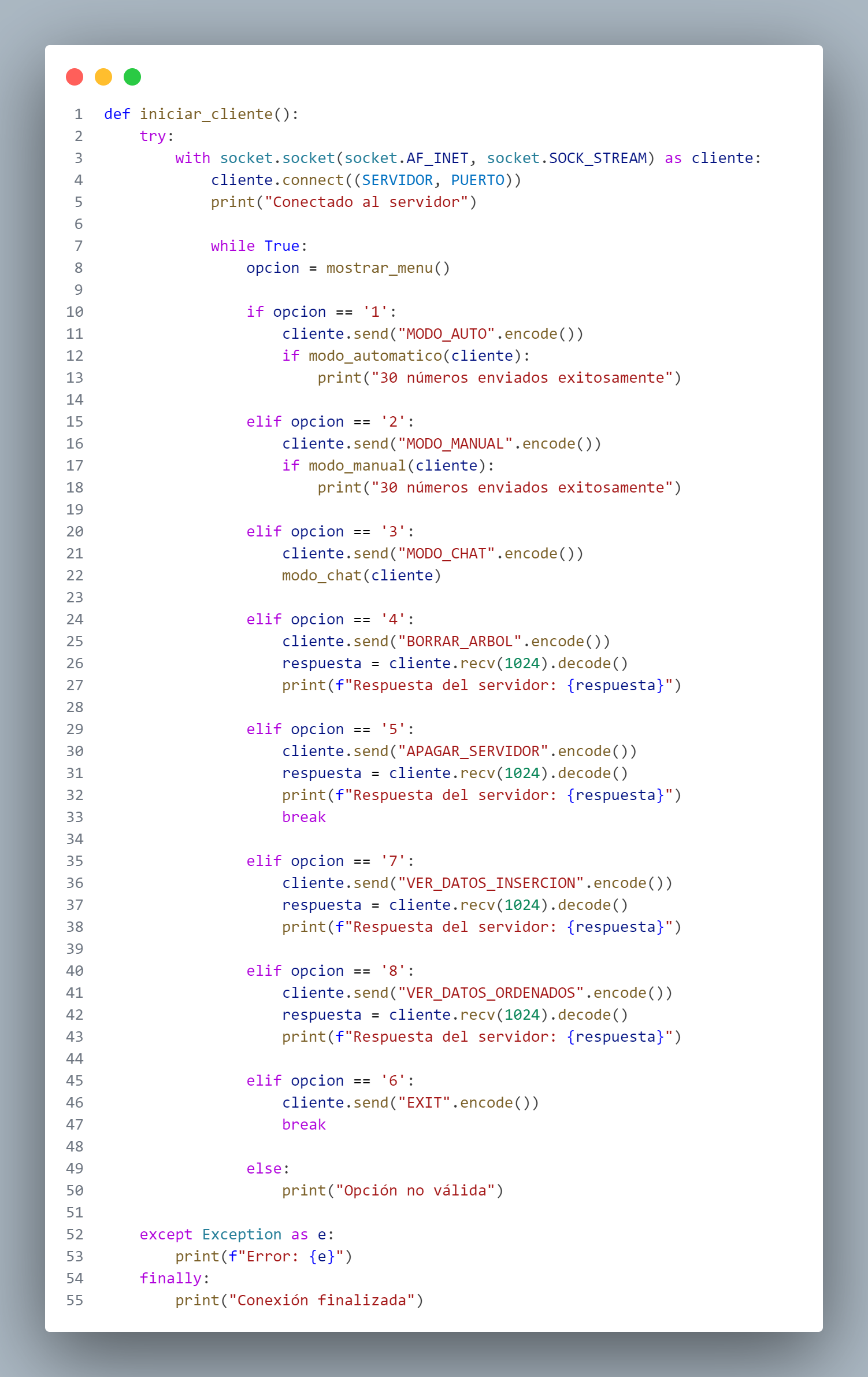
El programa está estructurado en distintos archivos que cumplen funciones específicas:

Cliente (cliente\_main.py y cliente.py)

Este módulo inicia la conexión con el servidor, envía los números uno por uno y espera una respuesta.

* Se establece la conexión con el servidor en la IP 127.0.0.1 y el puerto 65432.
* Se envían **30 números enteros** al servidor.
* Se recibe una **confirmación del servidor** tras cada envío.

Figura 13 Estructura de Árbol Binario en Cliente.py



Servidor (server.py)

Este módulo gestiona las conexiones de los clientes y construye el árbol binario de búsqueda.

* Se **crea y configura un socket** en el servidor.
* Se **escucha en el puerto 65432** y se aceptan conexiones de clientes.
* Se lanza un **hilo independiente por cliente** para manejar múltiples conexiones.

Figura 14 Estructura Socket Server.py



Manejo de Clientes (manejador.py)

Este módulo recibe los números y los inserta en el árbol binario.

* Recibe el número enviado por el cliente.
* Lo **convierte en entero** y lo **inserta en el árbol binario**.
* Responde con un mensaje de confirmación.

Figura 15 Conexión e inicio del Árbol Binario Part. 1



Figura 16 Árbol binario manejador.py

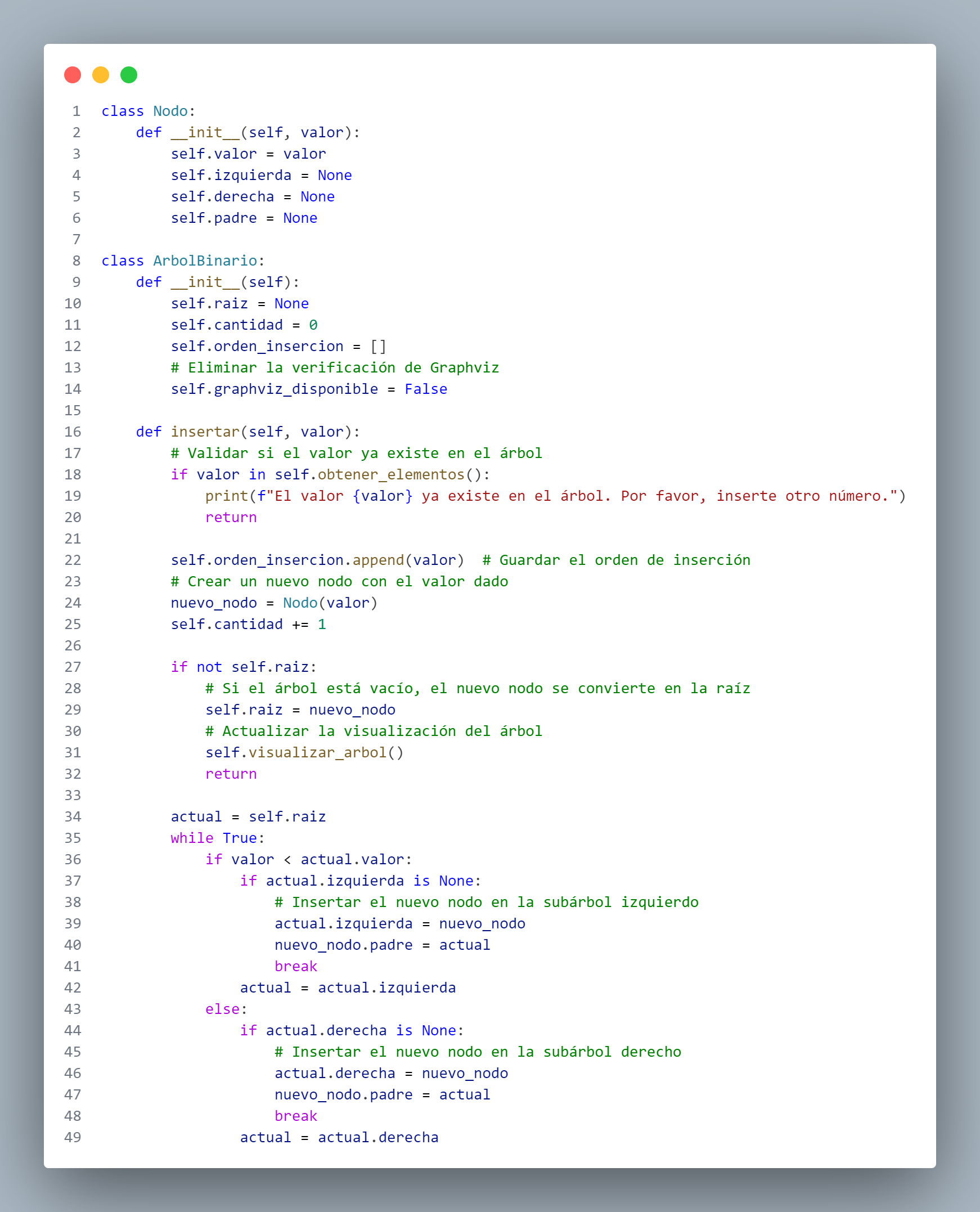


Árbol Binario (arbol\_binario.py)

Este módulo implementa la estructura del árbol binario de búsqueda.

* La clase Nodo representa cada número del árbol.
* La clase ArbolBinario maneja la inserción según las **reglas del árbol binario**:
  + Si el número es **menor**, va a la izquierda.
  + Si el número es **mayor**, va a la derecha.

Figura 17 reglas del árbol binario



Pruebas y resultados

Ejemplo de Entrada y Salida

Si el cliente envía la siguiente secuencia de números:

IMAGEN

El árbol se construye de la siguiente manera:

IMAGEN

El servidor responde con:

IMAGEN

Se puede recorrer el árbol en **orden inorden**, mostrando los números en orden ascendente:

Al implementar el sistema cliente-servidor con sockets y árboles binarios de búsqueda, permitió demostrar la integración de estructuras de datos con técnicas de comunicación en red. A lo largo del proyecto, se logró:

• Establecer una comunicación eficiente entre cliente y servidor, permitiendo la transmisión de datos de manera organizada y confiable.

• Implementar un Árbol Binario de Búsqueda que almacena los valores de forma estructurada, asegurando búsquedas e inserciones óptimas.

• Aplicar principios de concurrencia mediante hilos para manejar múltiples clientes de manera simultánea.

• Optimizar el procesamiento de datos al organizar los números enviados en una estructura eficiente.

3.3. Video explicativo

Para la comprensión del programa se ha realizado un video explicativo, donde se muestra desde la estructura del programa hasta la ejecución del mismo. El cual fue subido a la plataforma YouTube: <https://youtu.be/00pc4j97oQc>

Conclusiones

El desarrollo de esta actividad permitió implementar una comunicación efectiva entre un cliente y un servidor utilizando sockets, lo que nos ayudó a comprender cómo se gestionan los datos en redes y cómo pueden integrarse con estructuras de datos avanzadas como los árboles binarios. A lo largo del proceso, logramos construir un servidor que recibe 30 datos numéricos enteros de dos cifras enviados por el cliente, almacenándolos en un árbol binario de búsqueda en tiempo real.

Esta implementación refuerza la importancia del manejo de concurrencia a través de hilos para gestionar múltiples clientes y garantizar la seguridad de los datos mediante mecanismos como bloqueos de acceso (locks). Además, la validación y procesamiento de los datos en el servidor fueron aspectos clave para mantener la estabilidad del sistema y asegurar su correcto funcionamiento.

Desde el punto de vista académico, este ejercicio nos permitió aplicar conceptos fundamentales de Modelos de Programación, como la manipulación de estructuras de datos dinámicas, la programación orientada a eventos y la optimización de procesos en un entorno distribuido. Además, la integración de herramientas de visualización con matplotlib nos brindó una representación gráfica del árbol binario, facilitando su análisis.

En conclusión, la actividad no solo consolidó nuestros conocimientos sobre comunicación en red y estructuras de datos, sino que también nos brindó una perspectiva más amplia sobre el diseño y desarrollo de sistemas eficientes en entornos distribuidos. Esto nos prepara para enfrentar retos más complejos en el desarrollo de software, optimizando la transmisión y procesamiento de datos en aplicaciones del mundo real.

# Referencias

*https://web-production-b5a88.up.railway.app/*. (s.f.). Obtenido de Magiloop.