

## **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 el marco del desarrollo de habilidades de programación orientadas a la solución de problemas con estructuras de datos y comunicación cliente-servidor, este trabajo presenta una aplicación desarrollada en Python que implementa la construcción de un árbol binario a través del uso de sockets. Esta propuesta responde a la necesidad de integrar conceptos clave como la programación orientada a objetos, la implementación de estructuras jerárquicas (en este caso, árboles binarios) y la utilización de mecanismos de comunicación en red, permitiendo el envío de datos entre un cliente y un servidor en tiempo real.

El servidor es el encargado de construir de manera dinámica el árbol binario con los datos numéricos proporcionados por el cliente. La información enviada corresponde a 30 números enteros de dos cifras, transmitidos de manera secuencial, lo que permite observar cómo se conforma la estructura del árbol en función del orden de inserción. La construcción del árbol se visualiza gráficamente, facilitando su análisis estructural.

Esta actividad fomenta el trabajo colaborativo, la toma de decisiones, la organización lógica del código y el análisis del flujo de datos, lo que refuerza competencias tanto técnicas como cognitivas. A su vez, permite profundizar en conceptos fundamentales como el recorrido inorden, el uso de hilos (threads), el control del flujo de comunicación, y la aplicación de principios de diseño modular y reutilizable en la programación.

## **Objetivos**

### **Objetivo general**

Desarrollar una aplicación en Python que permita construir un árbol binario en el servidor a partir de la recepción de datos enviados por el cliente, utilizando la comunicación por sockets, con el fin de aplicar los conceptos de estructuras de datos y programación cliente-servidor.

### **Objetivos específicos**

- Implementar una arquitectura cliente-servidor que permita la transmisión de datos mediante sockets.
- Aplicar los principios de la programación orientada a objetos para la construcción y gestión de un árbol binario.
- Diseñar un flujo lógico que permita al cliente enviar 30 números enteros de dos cifras al servidor.
- Visualizar gráficamente el árbol binario generado a partir de los datos recibidos.
- Fomentar el trabajo colaborativo, la organización del código y el análisis estructurado de datos.

## Tabla de contenido

Objetivos .....	3
Objetivo general .....	3
Objetivos específicos.....	3
Desarrollo de la actividad.....	6
Arquitectura de la aplicación cliente-servidor .....	6
Lógica de funcionamiento del sistema .....	7
Implementación del árbol binario .....	8
Visualización del árbol binario .....	10
Fragmentos de código utilizado .....	11
Modo automático.....	11
Modo manual.....	11
Modo chat.....	12
Resultados y evidencias .....	13
Imagen del árbol binario generado.....	13
Respuesta del servidor ante comandos del cliente .....	14
Parte Desplegada .....	15
Arquitectura del Sistema .....	15
a) Interfaz Web y Comunicación HTTP .....	15
Cliente (Front-End): .....	15
Servidor HTTP (Back-End): .....	15
b) Módulo de Traducción y Comunicación por REST .....	16
Módulo Traductor: .....	16

Consumo de API Pública de Magic Loops: .....	16
Flujo de Datos y Procesos .....	18
Ingreso del Mensaje: .....	18
Recepción en el Servidor HTTP: .....	18
Invocación del Traductor: .....	19
Respuesta y Actualización del Chat: .....	20
Despliegue en Railway: .....	21
Resultados: .....	21
Video explicativo .....	22
Conclusiones .....	23
Referencias .....	24

## **Desarrollo de la actividad**

A continuación se describe el proceso de desarrollo de la aplicación solicitada, la cual permite la creación de un árbol binario desde un servidor, utilizando comunicación por sockets en Python. Este desarrollo se llevó a cabo de manera estructurada, aplicando principios de programación orientada a objetos, modularidad y trabajo colaborativo. La aplicación está dividida en dos partes principales: el cliente y el servidor. El servidor es el encargado de recibir los datos, procesarlos e ir construyendo el árbol binario. El cliente se conecta al servidor a través de un socket TCP/IP y tiene diferentes modos de interacción: modo automático, modo manual y modo chat.

### **Arquitectura de la aplicación cliente-servidor**

El servidor puede gestionar múltiples conexiones utilizando hilos (threads), garantizando que varios clientes puedan interactuar de forma concurrente sin afectar el procesamiento del árbol. Además, se implementan mecanismos para apagar el servidor, borrar el árbol y visualizar los datos ordenados.

Código del archivo `server.py`:

```

1  SERVIDOR = '127.0.0.1'
2  PUERTO = 65432
3  arbol_binario = ArbolBinario()
4  candado = threading.Lock()
5
6  def iniciar_servidor():
7      servidor = None
8      try:
9          servidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10         servidor.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
11
12         # Intentar vincular el puerto
13         try:
14             servidor.bind((SERVIDOR, PUERTO))
15         except OSError as e:
16             print(f"Error al vincular puerto {PUERTO}. Asegúrese de que no esté en uso.")
17             return
18
19         servidor.listen(5) # Permitir hasta 5 conexiones pendientes
20         print(f"Servidor escuchando en {SERVIDOR}:{PUERTO}...")
21         print("Presione Ctrl+C para detener el servidor")
22
23         while not obtener_estado_apagado():
24             try:
25                 servidor.settimeout(1)
26                 conexion, direccion = servidor.accept()
27                 print(f"Cliente conectado desde {direccion}")
28                 hilo = threading.Thread(target=manejar_cliente,
29                                         args=(conexion, direccion, arbol_binario, candado))
30                 hilo.daemon = True # Hacer el hilo daemon
31                 hilo.start()
32             except socket.timeout:
33                 continue
34             except KeyboardInterrupt:
35                 print("\nDeteniendo el servidor...")
36                 apagar()
37                 break
38             except Exception as e:
39                 print(f"Error en la conexión: {e}")
40
41     except Exception as e:
42         print(f"Error en el servidor: {e}")
43     finally:
44         if servidor:
45             servidor.close()
46         # Cerrar todas las figuras de matplotlib
47         plt.close('all')
48         print("Servidor apagado correctamente.")

```

Figura 1. Código archivo `server.php`

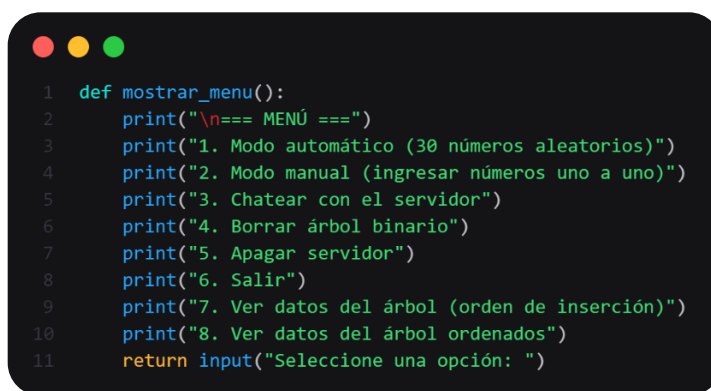
## Lógica de funcionamiento del sistema

La lógica del sistema se basa en el envío de datos por parte del cliente al servidor, quien procesa dichos datos e inserta cada número en el árbol binario. El servidor permite tres modos principales de interacción:

- Modo automático: El cliente genera automáticamente 30 números aleatorios de dos cifras y los envía uno por uno al servidor.
- Modo manual: El usuario ingresa manualmente los números, verificando que cumplan con el criterio (dos cifras, sin duplicados).
- Modo chat: El cliente puede enviar mensajes que serán traducidos por el servidor mediante una API externa.

A cada número recibido, el servidor valida el rango (entre 10 y 99) y, si es válido, lo inserta en el árbol. También permite ver los datos en orden de inserción, ordenados con el algoritmo de burbuja, borrar el árbol y apagar el servidor.

Código del archivo `cliente.py`:



```

1  def mostrar_menu():
2      print("\n=== MENÚ ===")
3      print("1. Modo automático (30 números aleatorios)")
4      print("2. Modo manual (ingresar números uno a uno)")
5      print("3. Chatear con el servidor")
6      print("4. Borrar árbol binario")
7      print("5. Apagar servidor")
8      print("6. Salir")
9      print("7. Ver datos del árbol (orden de inserción)")
10     print("8. Ver datos del árbol ordenados")
11     return input("Seleccione una opción: ")

```

*Figura 2. Código archivo `cliente.php` Menú*

## Implementación del árbol binario

El componente principal de la lógica estructural es la clase `ArbolBinario`, ubicada en el archivo `arbol_binario.py`. Esta clase utiliza nodos que contienen referencias a sus hijos



izquierdo y derecho, y sigue el principio de inserción recursiva para mantener el orden propio de los árboles binarios de búsqueda.

Cada vez que un número nuevo es insertado, se crea un nodo y se coloca en su posición correspondiente dependiendo de su valor. Además, se mantiene un registro del orden en que se insertaron los valores, permitiendo mostrar tanto el recorrido inorden como la secuencia de entrada.

Código del archivo `arbol_binario.py`:

```

1  class Nodo:
2      def __init__(self, valor):
3          self.valor = valor
4          self.izquierda = None
5          self.derecha = None
6          self.padre = None
7
8  class ArbolBinario:
9      def __init__(self):
10         self.raiz = None
11         self.cantidad = 0
12         self.orden_insercion = []
13         self.graphviz_disponible = False
14
15     def insertar(self, valor):
16         if valor in self.obtener_elementos():
17             print(f"El valor {valor} ya existe en el árbol. Por favor, inserte otro número.")
18             return
19
20         self.orden_insercion.append(valor) # Guardar el orden de inserción
21         nuevo_nodo = Nodo(valor)
22         self.cantidad += 1
23
24         if not self.raiz:
25             self.raiz = nuevo_nodo
26             # Actualizar la visualización del árbol
27             self.visualizar_arbol()
28             return
29
30         actual = self.raiz
31         while True:
32             if valor < actual.valor:
33                 if actual.izquierda is None:
34                     # Insertar el nuevo nodo en la subárbol izquierdo
35                     actual.izquierda = nuevo_nodo
36                     nuevo_nodo.padre = actual
37                     break
38                 actual = actual.izquierda
39             else:
40                 if actual.derecha is None:
41                     # Insertar el nuevo nodo en la subárbol derecho
42                     actual.derecha = nuevo_nodo
43                     nuevo_nodo.padre = actual
44                     break
45                 actual = actual.derecha

```

Figura 3. Código archivo `arbol_binario.py` Instancia he inserción

## Visualización del árbol binario

Una de las funcionalidades clave del proyecto es la capacidad de generar una representación gráfica del árbol binario. Esta visualización se realiza utilizando las bibliotecas `matplotlib` y `networkx`, y se guarda automáticamente en un archivo de imagen (`arbol_binario.png`) cada vez que se actualiza el árbol.

La visualización se construye de forma jerárquica, mostrando claramente las relaciones padre-hijo en el árbol. Se calcula el ancho de cada subárbol para posicionar los nodos correctamente y se utiliza un grafo dirigido para representar las conexiones entre nodos.

Código de visualización del árbol:

```

1  def visualizar_arbol(self, ruta_guardado='./local/vistaArbol/arbol_binario'):
2      if not self.raiz:
3          return "Árbol vacío"
4
5      try:
6          os.makedirs(os.path.dirname(ruta_guardado), exist_ok=True)
7
8          G = nx.DiGraph()
9          pos = {}
10         ancho_total = self._calcular_ancho_subarbol(self.raiz)
11         self._agregar_nodos_networkx(self.raiz, 0, 0, pos, G, ancho_total)
12         fig = plt.figure(figsize=(12, 8))
13         nx.draw(G, pos,
14                 with_labels=True,
15                 node_color='lightblue',
16                 node_size=2000,
17                 font_size=16,
18                 font_weight='bold',
19                 arrows=True,
20                 edge_color='gray',
21                 arrowsize=20)
22         plt.savefig(f"{ruta_guardado}.png", bbox_inches='tight')
23         plt.close(fig)
24
25         print(f"Árbol actualizado en: {ruta_guardado}.png")
26         return "Visualización generada exitosamente"
27
28     except Exception as e:
29         print(f"Error al generar visualización: {e}")
30         return self._visualizar_texto()
31     finally:
32         plt.close('all')

```

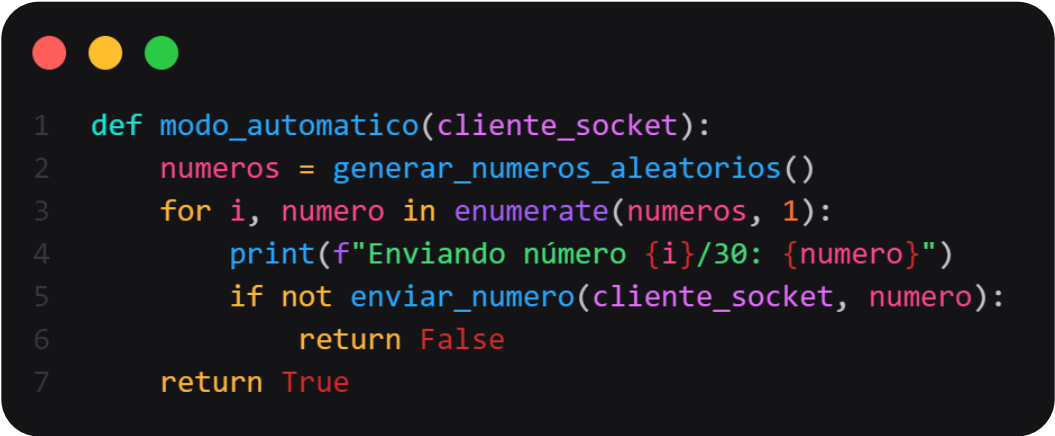
*Figura 4. Código archivo `arbol_binario.php` visualización*

### Fragmentos de código utilizado

La aplicación cuenta con tres modos de interacción principales entre el cliente y el servidor. A continuación, se presentan los fragmentos de código más relevantes que controlan el comportamiento en cada uno de estos modos.

#### Modo automático

En este modo, el cliente genera de manera automática 30 números aleatorios de dos cifras y los envía al servidor sin intervención del usuario. Esta opción es útil para pruebas rápidas y para llenar el árbol de forma directa.



```
1 def modo_automatico(cliente_socket):
2     numeros = generar_numeros_aleatorios()
3     for i, numero in enumerate(numeros, 1):
4         print(f"Enviando número {i}/30: {numero}")
5         if not enviar_numero(cliente_socket, numero):
6             return False
7     return True
```

*Figura 5. Código archivo cliente.php modo automatico*

#### Modo manual

Este modo permite que el usuario introduzca los números uno a uno. El sistema valida cada entrada para garantizar que sea un número válido de dos cifras.

```

1  def modo_manual(cliente_socket):
2      numeros_enviados = 0
3      while numeros_enviados < 30:
4          numero = input(f"Ingrese número {numeros_enviados + 1}/30 (debe ser de 2 cifras, 'q' para salir): ")
5          if numero.lower() == 'q':
6              return False
7
8          if enviar_numero(cliente_socket, numero):
9              numeros_enviados += 1
10     return True

```

*Figura 6. Código archivo cliente.php modo manual*

## Modo chat

Permite al cliente enviar mensajes en lenguaje natural al servidor, el cual se encarga de traducirlos usando una API externa y devuelve la traducción.

```

1  def modo_chat(cliente_socket):
2      print("Modo chat iniciado (escriba 'q' para salir)")
3      cliente_socket.send("MODO_CHAT".encode())
4      while True:
5          try:
6              mensaje = input("Tú: ").strip()
7              if not mensaje:
8                  continue
9
10             if mensaje.lower() == 'q':
11                 cliente_socket.send("CHAT_EXIT".encode())
12                 break
13
14             mensaje = f"CHAT:{mensaje}"
15             cliente_socket.send(mensaje.encode())
16             respuesta = cliente_socket.recv(1024).decode()
17             if respuesta:
18                 print(f"Servidor: {respuesta}")
19             else:
20                 print("No se recibió respuesta del servidor")
21                 break
22
23         except ConnectionError:
24             print("Se perdió la conexión con el servidor")
25             break
26         except Exception as e:
27             print(f"Error en el chat: {e}")
28             break

```

*Figura 7. Código archivo cliente.php modo chat*

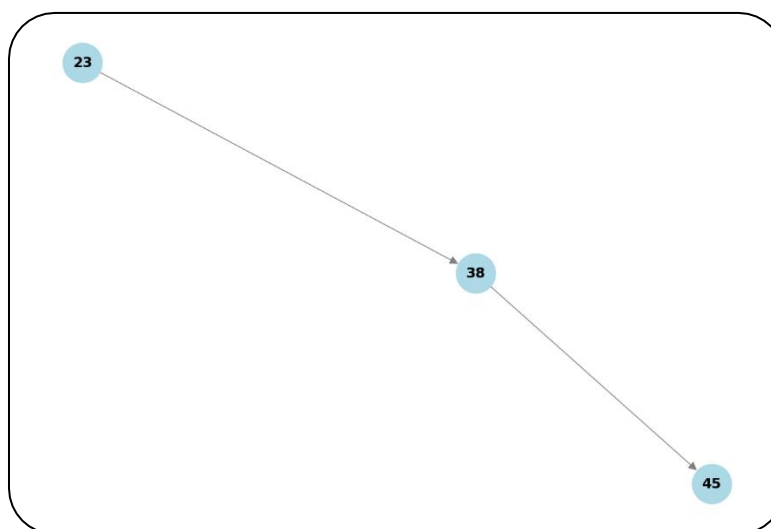
## Resultados y evidencias

En esta sección se presentan las salidas gráficas y textuales obtenidas durante la ejecución de la aplicación. Se evidencia el éxito del envío de los datos, la construcción progresiva del árbol binario y la respuesta del servidor ante los comandos emitidos desde el cliente.

### Imagen del árbol binario generado

Cada vez que el servidor recibe e inserta un nuevo dato, actualiza la visualización del árbol binario. La imagen generada corresponde a una estructura que refleja el orden lógico y jerárquico de los elementos insertados.

Imagen generada:



*Figura 8. Imagen correspondiente a los datos insertados y salida del código*

La visualización permite observar claramente las ramas izquierda y derecha del árbol, donde los valores menores al nodo raíz se sitúan a la izquierda y los mayores a la derecha, cumpliendo con la lógica de un árbol binario de búsqueda.

## Respuesta del servidor ante comandos del cliente

El servidor responde de manera inmediata y adecuada a los distintos comandos enviados por el cliente. A continuación, se listan algunas de las respuestas obtenidas:

Inserción exitosa: “Número 45 insertado. Árbol actual: [23, 38, 45]”

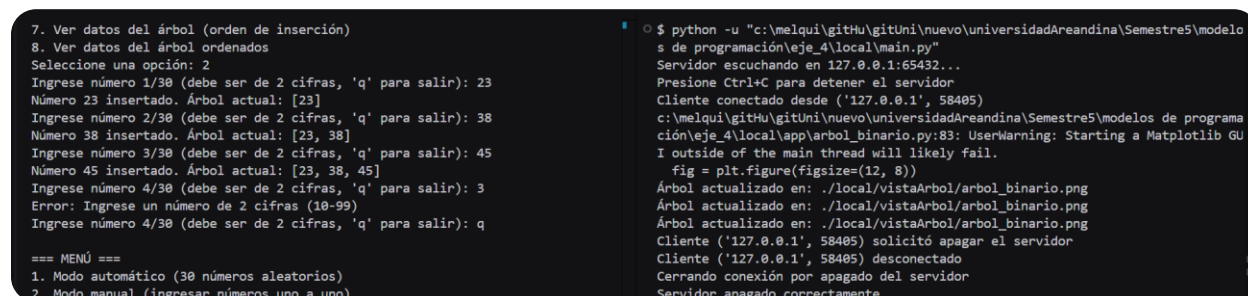
Número inválido: “Error: Ingrese un número de 2 cifras (10-99)”

Borrar árbol: “Árbol binario borrado completamente”

Ver datos ordenados: “Datos del árbol ordenados: [23, 38, 45]”

- Apagar servidor: “Apagando el servidor...”

Estas respuestas permiten al usuario validar las acciones ejecutadas, y ofrecen retroalimentación inmediata sobre el estado actual del sistema, lo que mejora la experiencia de uso y la comprensión del flujo lógico.



```

7. Ver datos del árbol (orden de inserción)
8. Ver datos del árbol ordenados
Seleccione una opción: 2
Ingreso número 1/30 (debe ser de 2 cifras, 'q' para salir): 23
Número 23 insertado. Árbol actual: [23]
Ingreso número 2/30 (debe ser de 2 cifras, 'q' para salir): 38
Número 38 insertado. Árbol actual: [23, 38]
Ingreso número 3/30 (debe ser de 2 cifras, 'q' para salir): 45
Número 45 insertado. Árbol actual: [23, 38, 45]
Ingreso número 4/30 (debe ser de 2 cifras, 'q' para salir): 3
Error: Ingrese un número de 2 cifras (10-99)
Ingreso número 4/30 (debe ser de 2 cifras, 'q' para salir): q

=== MENÚ ===
1. Modo automático (30 números aleatorios)
2. Modo manual (ingresar números uno a uno)

$ python -u "c:\melqui\gitHu\gitUni\nuevo\universidadAreandina\Semestre5\modelo
s de programación\ej_4\local\main.py"
Servidor escuchando en 127.0.0.1:65432...
Presione Ctrl+C para detener el servidor
Cliente conectado desde ('127.0.0.1', 58405)
c:\melqui\gitHu\gitUni\nuevo\universidadAreandina\Semestre5\modelos de programa
ción\ej_4\local\app\arbol_binario.py:83: UserWarning: Starting a Matplotlib GU
I outside of the main thread will likely fail.
  fig = plt.figure(figsize=(12, 8))
Árbol actualizado en: ./local/vistaArbol/arbol_binario.png
Árbol actualizado en: ./local/vistaArbol/arbol_binario.png
Árbol actualizado en: ./local/vistaArbol/arbol_binario.png
Cliente ('127.0.0.1', 58405) solicitó apagar el servidor
Cliente ('127.0.0.1', 58405) desconectado
Cerrando conexión por apagado del servidor
Servidor apagado correctamente.

```

*Figura 9. Inserción y respuesta del servidor, desde terminal*

## Parte Desplegada

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*

### 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:

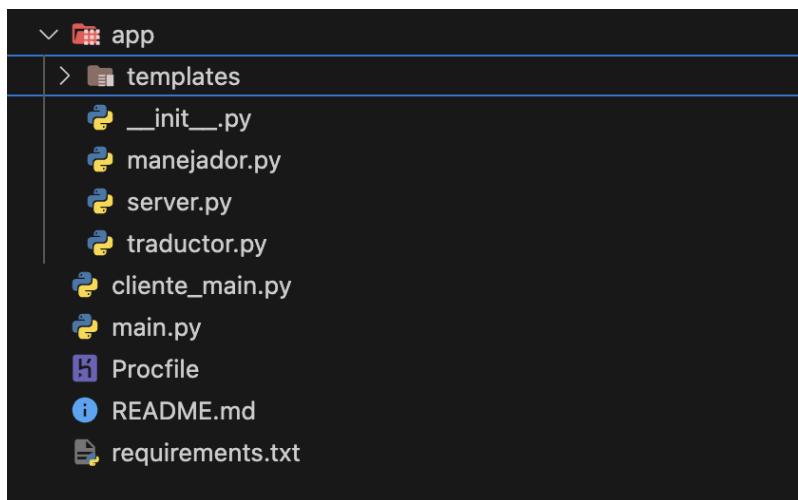
- Agrega el mensaje del usuario al área de chat.
- Realiza una petición fetch con método POST a la ruta `/chat` enviando el mensaje en formato JSON.
- Recibe la respuesta del servidor (el mensaje traducido) y lo muestra en la ventana de chat.

##### 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 10. 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.

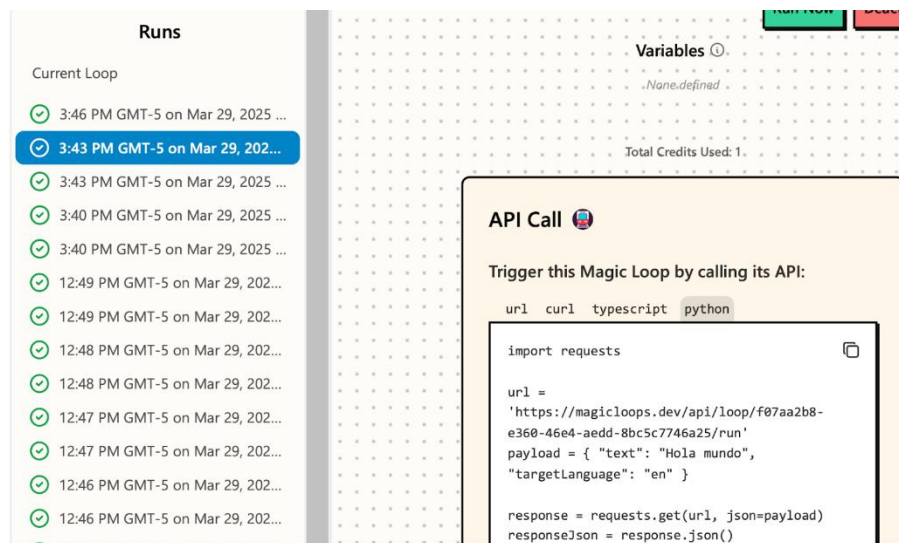
De esta forma, la aplicación Flask actúa como intermediaria: recibe la solicitud web, invoca al traductor que se comunica vía API con el servicio de traducción, y devuelve la respuesta traducida al cliente web.

##### 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 11. Visualización de las peticiones hechas sobre (magic loop, s.f.)*

Nota: Elaboración propia

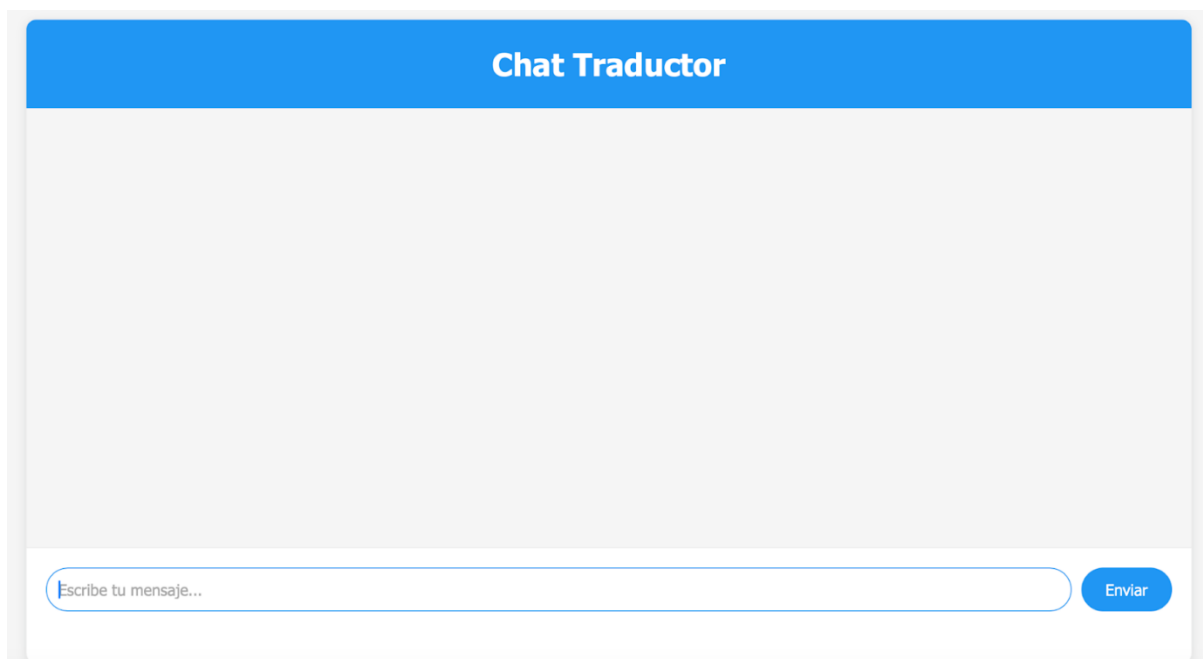
## Flujo de Datos y Procesos

El proceso completo se puede describir en los siguientes pasos:

### 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 12. Visualización del frontend del chat que invocara las peticiones*

Nota: Elaboración propia

### 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.



```

main.py > ...
1  import os
2  from app.server import app
3
4  if __name__ == '__main__':
5      port = int(os.environ.get('PORT', 5000))
6      app.run(host='0.0.0.0', port=port)

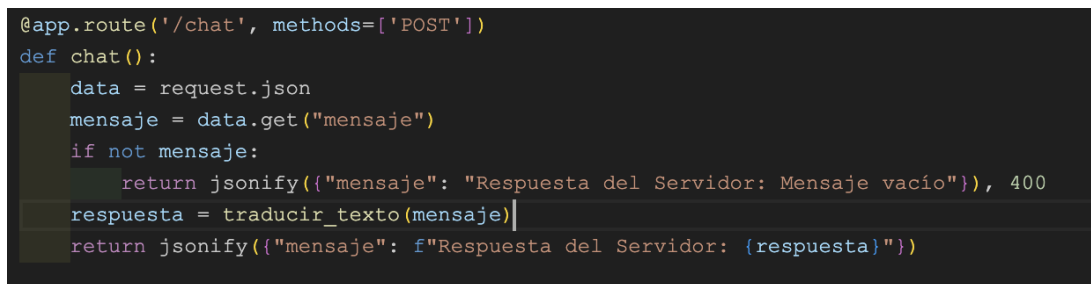
```

*Figura 13. Definición del servidor por el puerto 5000*

Nota: Elaboración propia

### Invocación del Traductor:

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



```

@app.route('/chat', methods=['POST'])
def chat():
    data = request.json
    mensaje = data.get("mensaje")
    if not mensaje:
        return jsonify({"mensaje": "Respuesta del Servidor: Mensaje vacío"}), 400
    respuesta = traducir_texto(mensaje)
    return jsonify({"mensaje": f"Respuesta del Servidor: {respuesta}"})

```

*Figura 14. Definición del path /chat sobre metodo POST para realizar la petición de traducir*

Nota: Elaboración propia

```

app > traductor.py > ...
1  import requests      La importación "requests" no se ha podido resolver desde el origen
2  import json
3
4  def traducir_texto(texto, target_language="en"):
5      url = 'https://magicloops.dev/api/loop/run/xxx xxx xxx'
6      payload = {"text": texto, "targetLanguage": target_language}
7
8      try:
9          response = requests.get(url, json=payload)
10         if response.status_code == 200:
11             try:
12                 responseJson = response.json()
13                 # Extraer solo la traducción (loopOutput) del JSON
14                 traduccion = responseJson.get('loopOutput', 'No se encontró traducción')
15                 return f"Traducción: {traduccion}"
16             except json.JSONDecodeError:
17                 return "Error: No se pudo procesar la respuesta del servidor"
18             except Exception as e:
19                 return f"Error al procesar la respuesta: {str(e)}"
20         else:
21             return f"Error en la traducción. Código: {response.status_code}"
22     except Exception as e:
23         return f"Error de conexión: {str(e)}"
24

```

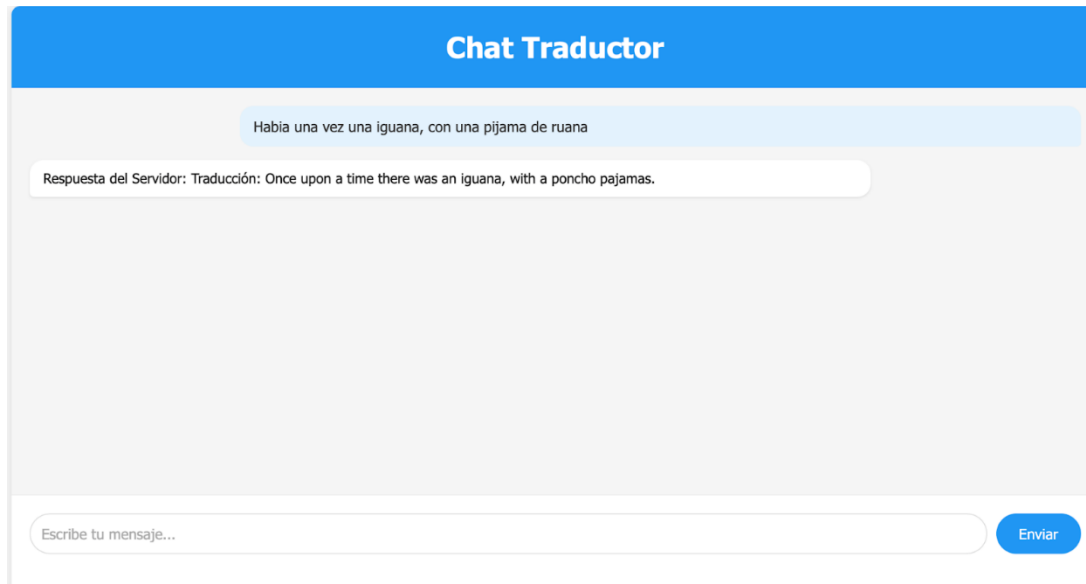
*Figura 15. Definición de la función traducir\_texto para realizar la petición en magicloops*

Nota: Elaboración propia

## 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 16. Petición traducida y visualizada dentro del frontend*

Nota: Elaboración propia

### 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.

### **Video explicativo**

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

## **Conclusiones**

El desarrollo de esta actividad permitió integrar múltiples conceptos fundamentales en el área de la programación, como lo son las estructuras de datos, la comunicación en red mediante sockets y la programación orientada a objetos. La implementación de un árbol binario en un entorno cliente-servidor no solo fortaleció habilidades técnicas, sino también la capacidad de análisis lógico y solución estructurada de problemas.

Durante la construcción de la aplicación, se puso en práctica la lógica necesaria para manejar entradas, organizar datos jerárquicamente y generar representaciones visuales que fortalecen la comprensión del comportamiento de un árbol binario. Además, se exploraron diferentes formas de interacción con el sistema, desde la automatización de tareas hasta la simulación de procesos de traducción mediante un chat integrado.

Asimismo, este proyecto destacó la importancia del trabajo colaborativo, la organización del código y el uso de librerías externas para ampliar las funcionalidades. Se evidencia que, con una buena planificación y distribución de roles, es posible desarrollar soluciones eficientes, funcionales y bien estructuradas que cumplen con los objetivos planteados desde el inicio.

## Referencias

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

Python Software Foundation. (n.d.). *socket* — *Low-level networking interface*. Python 3.12.

<https://docs.python.org/3/library/socket.html>

NetworkX Developers. (n.d.). *NetworkX documentation*.

<https://networkx.org/documentation/stable/>