

Instituto Politécnico Nacional

Escuela Superior de Cómputo

Materia: Sistemas Distribuidos

Profesor: Carreto Arellano Chadwick

Grupo: 7CM1

Práctica 6 – Servicios Web

Valdés Castillo Diego - 2021630756

Fecha de Entrega: 02/04/2025

Índice

1. Antecedente (Marco teórico).....	2
2. Planteamiento del Problema	4
3. Propuesta de Solución.....	5
4. Materiales y Métodos empleados.....	6
5. Desarrollo de la Solución.....	9
6. Resultados	13
7. Conclusiones.....	21
8. Referencias Bibliográficas	22

1. Antecedente (Marco teórico)

En el desarrollo de sistemas distribuidos, los Servicios Web juegan un papel fundamental al permitir la comunicación entre diferentes aplicaciones a través de la web, independientemente del lenguaje de programación o plataforma utilizada. A diferencia de las arquitecturas tradicionales de comunicación basada en sockets, los Servicios Web utilizan protocolos estándar como HTTP para la transferencia de datos, facilitando la interoperabilidad entre sistemas.

El modelo Cliente-Servidor en Servicios Web se basa en la interacción entre un cliente, que realiza peticiones, y un servidor, que procesa dichas solicitudes y devuelve respuestas en formato estructurado, generalmente JSON o XML. Este enfoque permite el acceso remoto a recursos y funcionalidades expuestas por el servidor sin necesidad de compartir código fuente o realizar configuraciones complejas de red.

En la implementación de servicios web, se distinguen tres componentes esenciales:

- **Servidor Web:** Es el encargado de gestionar las solicitudes de los clientes y responder con los datos requeridos. En este caso, se ha implementado con Flask en Python, proporcionando endpoints para el registro y movimiento de autos en una simulación de carrera.

- **Cliente:** Interactúa con el servidor enviando peticiones HTTP a través de métodos como GET y POST. En esta práctica, la interfaz web desarrollada en HTML permite a los usuarios visualizar y controlar la carrera.
- **Protocolo de Comunicación:** HTTP es el protocolo utilizado para el intercambio de información entre cliente y servidor, asegurando que las solicitudes y respuestas sean interpretadas correctamente en ambas partes.

Flask es un microframework de Python diseñado para desarrollar aplicaciones web de manera rápida y sencilla. A diferencia de otros frameworks más pesados, Flask proporciona una estructura flexible que permite a los desarrolladores definir rutas, gestionar solicitudes HTTP y renderizar plantillas HTML sin necesidad de configuraciones complejas. Su ligereza y modularidad lo hacen ideal para proyectos que requieren servicios web eficientes y escalables.

Una de las principales ventajas de Flask es su facilidad de integración con diversas tecnologías y bases de datos, lo que lo convierte en una opción popular para la creación de API RESTful. Además, su diseño minimalista permite a los desarrolladores añadir solo las herramientas necesarias, evitando sobrecargas innecesarias en la aplicación. Esto facilita la construcción de aplicaciones distribuidas que pueden comunicarse a través de la web sin complicaciones.

La implementación de Servicios Web con Flask presenta varias ventajas:

- **Facilidad de integración:** Al utilizar HTTP y JSON, las aplicaciones pueden comunicarse con otros sistemas sin necesidad de adaptaciones específicas.
- **Escalabilidad:** Es posible extender la funcionalidad del servidor sin modificar la estructura base, permitiendo la adición de nuevos endpoints y características.
- **Transparencia en la comunicación:** Los clientes pueden consumir los servicios sin conocer la lógica interna del servidor, lo que facilita su uso y mantenimiento.

2. Planteamiento del Problema

Esta práctica tiene como objetivo fortalecer el conocimiento en el desarrollo de aplicaciones web utilizando el framework Flask y la comunicación a través de peticiones HTTP (GET y POST). En este caso, se implementará un servicio web para gestionar una simulación de carrera de autos en la que los participantes (autos) se registran, avanzan y se clasifican en un podio según su rendimiento, todo ello en tiempo real.

El desafío principal es asegurar una gestión eficiente de las interacciones entre el servidor y los clientes, especialmente considerando que la carrera avanza dinámicamente y debe actualizarse constantemente. El servidor debe manejar las peticiones de registro de autos, movimiento de los mismos, y consultas sobre el estado de la carrera y el podio. Es crucial garantizar que cada auto avance correctamente en la pista y que los estados de la carrera se actualicen sin generar inconsistencias o bloqueos.

El servidor debe poder manejar múltiples peticiones simultáneas, asegurando que no haya conflictos o pérdida de datos. La implementación de las rutas en Flask permite a los clientes realizar peticiones POST para registrar y mover los autos, y peticiones GET para consultar el estado de la carrera y obtener el podio. Además, se debe asegurar que el flujo de datos se mantenga correcto, evitando que un auto se mueva si no ha sido registrado o si ya ha terminado la carrera.

Por último, la arquitectura debe ser lo suficientemente robusta como para manejar las solicitudes de manera eficiente y precisa, garantizando que la simulación se ejecute sin interrupciones y que el cliente reciba las actualizaciones en tiempo real, lo que mejora la simulación de la carrera.

3. Propuesta de Solución

Para abordar este problema, se propone desarrollar una aplicación web utilizando Flask en Python que permita gestionar una carrera de autos de manera interactiva a través de peticiones HTTP. La solución consta de los siguientes componentes principales:

- Servicio Web con Flask:
 - Se implementa un servidor web en Flask, el cual gestiona todas las interacciones con los clientes mediante peticiones GET y POST.
 - El servidor estará encargado de manejar el registro de autos, el avance de los mismos en la carrera y la consulta del estado de la carrera.
 - Las rutas en Flask (/register, /move, /race_status, y /podium) permiten que los clientes se conecten al servidor, registren autos, muevan los autos y obtengan información sobre la carrera y los resultados.
- Gestión de la carrera:
 - El servidor mantiene el estado de la carrera en un diccionario, que incluye los autos registrados, su posición en la pista y el podio de los ganadores.
 - Los autos avanzan en intervalos de tiempo simulados, y su posición se actualiza en tiempo real según las peticiones recibidas.
 - El servidor asegura que la carrera no comience hasta que haya al menos 4 autos registrados y que los autos no puedan moverse si ya han terminado la carrera o si no están registrados.
- Interfaz web para los clientes:
 - Los clientes interactúan con el servicio web a través de una página HTML que muestra el estado de la carrera en tiempo real.
 - Utilizando JavaScript y peticiones periódicas (setInterval), el cliente solicita actualizaciones sobre el estado de la carrera (como la posición de los autos y el podio) y visualiza estos cambios de forma dinámica.

- La interfaz gráfica muestra una pista de carrera donde los autos avanzan visualmente, y se actualiza cada segundo para reflejar los movimientos.
- Comunicación a través de peticiones GET y POST:
 - Las peticiones POST permiten registrar nuevos autos y moverlos en la carrera, enviando la información al servidor para su procesamiento.
 - Las peticiones GET permiten a los clientes obtener el estado actualizado de la carrera y consultar el podio final, sin necesidad de realizar peticiones manuales o sincronización entre clientes.

Esta solución basada en Flask y el modelo de comunicación HTTP garantiza que los datos se gestionen de manera eficiente y accesible a través de la web. Además, la implementación de peticiones periódicas permite que el cliente reciba actualizaciones en tiempo real sobre el estado de la carrera. La estructura centralizada del servidor web facilita la comunicación sin la necesidad de manejar conexiones complejas entre los clientes y el servidor, y la lógica de la carrera se gestiona de manera coherente, evitando inconsistencias y bloqueos en la simulación.

4. Materiales y Métodos empleados

- **Hardware:** Laptop ASUS y Teléfono iPhone 16.
- **Software:** Google Chrome, Word y Acrobat Reader.
- **Lenguaje de programación:** Python (Flask) y HTML
- **Entorno de desarrollo:** IDE compatible con Python y HTML, como Visual Studio Code con las debidas extensiones de depuración y ejecución, además de la terminal con los siguientes comandos:
 - `curl -X POST "http://localhost:5000/register" -H "Content-Type: application/json" -d '{"car_id": "car_name"}'`
 - `curl -X POST "http://localhost:5000/move" -H "Content-Type: application/json" -d '{"car_id": "car_name", "distance": metros}'`

- curl -X GET "http://localhost:5000/race_status"
- curl -X GET "http://localhost:5000/podium"
- python nombre_codigo.py
- **Bibliotecas utilizadas:**
 - **Flask:** Framework utilizado para desarrollar el servicio web en Python.
 - **request:** Se usa para manejar las peticiones HTTP (GET y POST) de los clientes.
 - **jsonify:** Permite crear respuestas en formato JSON para enviar datos estructurados a los clientes.
 - **render_template:** Se utiliza para renderizar las páginas HTML y mostrarlas en el navegador.
- **Métodos empleados:**
 - **Servidor Flask:**
 - Se crea un servidor Flask que escucha en el puerto 5000 y gestiona las peticiones GET y POST para interactuar con los clientes.
 - El servidor maneja las rutas /register (para registrar autos en la carrera), /move (para mover autos en la carrera), /race_status (para obtener el estado de la carrera), y /podium (para consultar los resultados finales).
 - El servidor mantiene un diccionario (race_status) que almacena los autos registrados, sus posiciones en la carrera y los autos que han llegado al podio.
 - Para cada auto, el servidor actualiza su posición al recibir las peticiones POST correspondientes, simulando un avance en la carrera.
 - Se gestiona el flujo de la carrera, asegurando que no se inicie hasta que haya al menos 4 autos registrados, y que los autos no puedan avanzar si ya han llegado a la meta.
 - **Interfaz web:**

- Se desarrolla una página HTML que permite a los usuarios interactuar con el servicio web. En ella, se visualiza una representación gráfica de la pista de carrera y los autos moviéndose.
- JavaScript maneja las peticiones GET periódicas para obtener actualizaciones del estado de la carrera (posición de los autos y podio).
- Los autos se mueven en la pantalla de manera animada, con posiciones que se actualizan cada segundo según los datos enviados por el servidor.
- Se utiliza un ciclo setInterval para realizar las consultas cada segundo y mantener la interfaz actualizada.
- **Petición y manejo de datos:**
 - Los clientes envían peticiones POST al servidor para registrar un nuevo auto y moverlo en la carrera. Estos datos son enviados en formato JSON, lo que permite al servidor procesar los movimientos de manera eficiente.
 - Las respuestas del servidor, también en formato JSON, contienen el estado actualizado de la carrera, incluyendo la posición de cada auto y el podio si la carrera ha finalizado.
- **Estrategia de programación:**
 - La implementación del sistema sigue un enfoque basado en la arquitectura cliente-servidor, utilizando Flask para la comunicación entre el servidor y los clientes a través de peticiones HTTP.
 - El servidor maneja las solicitudes de manera eficiente, asegurando que la información se mantenga consistente entre los diferentes clientes conectados.
 - Se utiliza una lógica sencilla para simular una carrera de autos, donde los autos avanzan en función de las peticiones POST y la actualización de la interfaz web ocurre en tiempo real mediante peticiones GET periódicas.

- La sincronización entre las peticiones y la actualización del estado de la carrera garantiza que no haya conflictos ni bloqueos, permitiendo que los clientes reciban información precisa y oportuna sobre el avance de la carrera.

5. Desarrollo de la Solución

Para implementar la solución de la carrera de autos gestionada a través de un servicio web, se desarrolló una aplicación basada en Flask, Python y HTML, donde el servidor se encarga de gestionar la carrera y proporcionar actualizaciones en tiempo real a los clientes conectados.

El proceso comienza cuando un cliente accede a la página web del servidor, el cual está escuchando en el puerto 5000. El cliente interactúa con el servidor a través de peticiones HTTP, específicamente mediante métodos GET y POST, que permiten registrar autos, moverlos en la carrera y consultar el estado en tiempo real.

El servidor de carrera es responsable de gestionar la competencia, y su estado se mantiene en un diccionario `race_status`. Los autos registrados se agregan a este diccionario, con sus posiciones inicializadas en cero. El servidor espera a que haya al menos 4 autos registrados antes de iniciar la simulación. A partir de ese momento, cada auto avanza de forma independiente según las solicitudes POST recibidas, simulando un avance en la carrera de manera aleatoria mediante distancias definidas por los usuarios.

Durante la ejecución de la carrera, los clientes consultan periódicamente el estado de la competencia mediante peticiones GET a la ruta `/race_status`. Esta consulta devuelve información sobre las posiciones de los autos y el estado actual del podio. Los resultados de la carrera se muestran dinámicamente en la página web, donde los autos se desplazan visualmente sobre una pista.

Para garantizar la consistencia y evitar errores de sincronización, el servidor asegura que solo se permitan movimientos válidos para los autos que aún no hayan alcanzado la meta y que la carrera no comience hasta que los requisitos mínimos

de registro se cumplan. Además, cuando un auto alcanza la meta, se agrega al podio y no podrá avanzar más.

Una vez que todos los autos han completado la carrera, el servidor determina las posiciones finales y envía una respuesta a los clientes con los resultados. Los resultados finales se muestran en la página web del cliente, y la comunicación entre el servidor y los clientes se cierra adecuadamente para asegurar una experiencia fluida y sin interrupciones.

A continuación se puede ver la solución propuesta en código documentado en la figura 1, 2, 3, 4, 5 y 6:

```
D: > Escuela8voSem > Distribuidos > Practica6 > carrera.py > ...
1  #Materia: Sistemas Distribuidos
2  #Autor: Diego Valdes Castillo
3  #Fecha de creacion: 01/04/2025
4  #Version: 1.0
5  #Practica 6 - Servicios Web
6  #Codigo de la carrera en pyhton - flask
7
8  from flask import Flask, request, jsonify, render_template
9
10 app = Flask(__name__)
11
12 # Estado de la carrera
13 race_status = {
14     "cars": {}, # Se inicializa vacio para registrar autos dinamicamente
15     "max_distance": 100,
16     "podium": []
17 }
18
19 @app.route('/')
20 def index():
21     return render_template('index.html')
22
23 @app.route('/register', methods=['POST'])
24 def register_car():
25     """Registra un nuevo auto en la carrera."""
26     car_id = request.json.get('car_id')
27     if not car_id:
28         return jsonify({"error": "Debes proporcionar un ID de auto"}), 400
29     if car_id in race_status["cars"]:
30         return jsonify({"error": "El auto ya esta registrado"}), 400
31
32     race_status["cars"][car_id] = {"position": 0}
33     return jsonify({"message": f"Auto {car_id} registrado exitosamente"})
34
35 @app.route('/move', methods=['POST'])
```

Figura 1. Código implementado en Python - 1.

```

36 def move_car():
37     """Mueve un auto basado en una solicitud POST."""
38     if len(race_status["cars"]) < 4:
39         return jsonify({"error": "La carrera iniciara cuando haya 4 autos registrados"}), 400
40
41     car_id = request.json.get('car_id')
42     distance = request.json.get('distance', 5)
43
44     if car_id not in race_status["cars"]:
45         return jsonify({"error": "Carro no registrado"}), 400
46
47     if car_id in race_status["podium"]:
48         return jsonify({"error": "Este auto ya termino la carrera"}), 400
49
50     race_status["cars"][car_id]["position"] += distance
51     if race_status["cars"][car_id]["position"] >= race_status["max_distance"]:
52         race_status["cars"][car_id]["position"] = race_status["max_distance"]
53         race_status["podium"].append(car_id)
54
55     return jsonify({"message": f"{car_id} avanza {distance} metros", "position": race_status["cars"][car_id]["position"]})
56
57 @app.route('/race_status', methods=['GET'])
58 def get_race_status():
59     """Devuelve el estado actual de la carrera."""
60     return jsonify(race_status)
61
62 @app.route('/podium', methods=['GET'])
63 def get_podium():
64     """Devuelve el podio final."""
65     if len(race_status["podium"]) == 4:
66         return jsonify({"podium": race_status["podium"]})
67     return jsonify({"message": "La carrera aun no ha terminado"})
68
69 if __name__ == '__main__':
70     app.run(host='0.0.0.0', port=5000, debug=True)

```

Figura 2. Código implementado en Python - 2.

```

D:\> Escuela8voSem > Distribuidos > Practica6 > templates > index.html > ...
1  <!--Materia: Sistemas Distribuidos
2  #Autor: Diego Valdes Castillo
3  #Fecha de creacion: 01/04/2025
4  #Version: 1.0
5  #Practica 6 - Servicios Web
6  #Codigo de la carrera en pyhton - flask-->
7
8  <!DOCTYPE html>
9  <html lang="es">
10 <head>
11     <meta charset="UTF-8">
12     <meta name="viewport" content="width=device-width, initial-scale=1.0">
13     <title>Carrera de Autos</title>
14 <style>
15     body {
16         font-family: Arial, sans-serif;
17         text-align: center;
18         background: linear-gradient(to right, #1e3c72, #2a5298);
19         color: white;
20         margin: 0;
21         padding: 20px;
22     }
23     h1 {
24         margin-bottom: 20px;
25     }
26     .track {
27         width: 80%;
28         height: 40px;
29         background: rgba(255, 255, 255, 0.2);
30         margin: 10px auto;
31         position: relative;
32         border-radius: 20px;
33         overflow: hidden;
34         display: flex;
35         align-items: center;

```

Figura 3. Código implementado en HTML para Interfaz - 1.

```

36     }
37     .car {
38         width: 40px;
39         height: 40px;
40         position: absolute;
41         transition: left 0.5s ease-in-out;
42         display: flex;
43         align-items: center;
44         justify-content: center;
45         font-size: 24px;
46     }
47     .car img {
48         width: 100%;
49         height: auto;
50         transform: scaleX(-1); /* Invierte la dirección del auto */
51     }
52     .car-name {
53         position: absolute;
54         left: 10px;
55         color: white;
56         font-weight: bold;
57         font-size: 18px;
58     }
59     .podium {
60         margin-top: 30px;
61         font-size: 18px;
62         font-weight: bold;
63         background: rgba(0, 0, 0, 0.5);
64         padding: 10px;
65         border-radius: 10px;
66         display: inline-block;
67     }

```

Figura 4. Código implementado en HTML para Interfaz - 2.

```

68     </style>
69 </head>
70 <body>
71     <h1>Carrera de Autos</h1>
72     <div id="race"></div>
73     <div class="podium" id="podium"></div>
74     <script>
75         async function fetchRaceStatus() {
76             const response = await fetch('/race_status');
77             const data = await response.json();
78             const raceDiv = document.getElementById('race');
79             raceDiv.innerHTML = '';
80
81             for (const [car, info] of Object.entries(data.cars)) {
82                 let trackDiv = document.createElement('div');
83                 trackDiv.classList.add('track');
84
85                 let nameDiv = document.createElement('div');
86                 nameDiv.classList.add('car-name');
87                 nameDiv.innerText = car;
88
89                 let carDiv = document.createElement('div');
90                 carDiv.classList.add('car');
91                 carDiv.style.left = `${(info.position / data.max_distance) * 100}%`;
92                 carDiv.innerHTML = ``;
93
94                 trackDiv.appendChild(nameDiv);
95                 trackDiv.appendChild(carDiv);
96                 raceDiv.appendChild(trackDiv);
97             }
98
99             if (data.podium.length > 0) {
100                 const podiumDiv = document.getElementById('podium');

```

Figura 5. Código implementado en HTML para Interfaz - 3.

```

101         podiumDiv.innerHTML = "<h2>Podio</h2>" + data.podium.map((car, index) => `<p>${index + 1}. ${car}</p>`).join('');
102     }
103 }
104 setInterval(fetchRaceStatus, 1000);
105 fetchRaceStatus();
106 </script>
107 </body>
108 </html>
109

```

Figura 6. Código implementado en HTML para Interfaz - 4.

6. Resultados

Los resultados obtenidos muestran que la comunicación entre el servidor de carrera y los múltiples clientes se lleva a cabo de manera eficiente, permitiendo que cada cliente reciba actualizaciones en tiempo real sobre el progreso de su auto en la competencia. Las peticiones HTTP, tanto GET como POST, se procesan correctamente y las respuestas se devuelven a los clientes sin pérdidas de información ni bloqueos, lo que demuestra una adecuada gestión de la comunicación entre el servidor y los clientes.

Al ejecutar la simulación en diferentes ocasiones, se observa que el orden de llegada de los autos varía debido a los avances aleatorios en las distancias que cubren, logrando así una simulación dinámica y realista de la carrera. Cada auto avanza de forma independiente, lo que hace que el resultado de la carrera cambie en cada ejecución, lo que refleja la aleatoriedad inherente a las competencias reales.

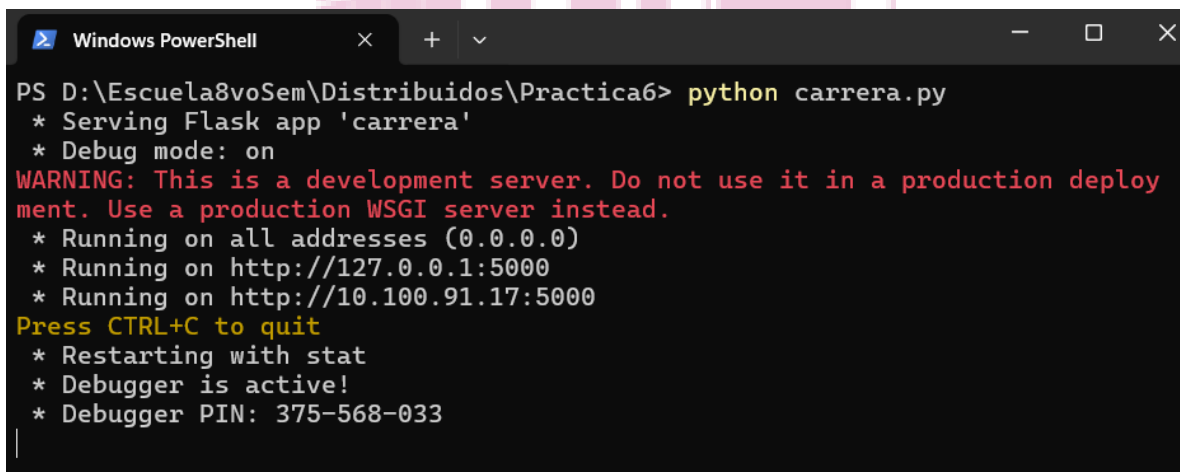
Además, se confirma que el servidor gestiona correctamente las solicitudes de los clientes. La carrera no comienza hasta que se han registrado al menos cuatro autos, y el servidor asegura que las posiciones de los autos se actualicen de manera consistente y en tiempo real. Cuando todos los autos alcanzan la meta, el servidor genera y envía correctamente el podio final a los clientes, mostrando los resultados de manera sincronizada en la página web.

La solución también ha demostrado ser estable, manteniendo las conexiones activas y proporcionando actualizaciones continuas sobre el estado de la carrera sin

interrupciones, garantizando una experiencia de usuario fluida dentro de la arquitectura basada en Flask y HTTP.

A continuación se puede ver una prueba de ejecución de los códigos desde la terminal y en una página web para ver de manera más ilustrativa el desarrollo de la carrera.

Primero en la figura 7 podemos ver el ejecución del archivo “carrera.py” para que aloje el servidor de flask en <http://127.0.0.1:5000>, esto lo hacemos desde la terminal con el comando “python carrera.py”.



```
Windows PowerShell
PS D:\Escuela8voSem\Distribuidos\Practica6> python carrera.py
* Serving Flask app 'carrera'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://10.100.91.17:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 375-568-033
```

Figura 7. Servidor con flask en el puerto 5000.

Una vez que ya esta listo nuestro servidor tenemos que abrir nuestro localhost <http://127.0.0.1:5000> para poder ver la simulación de la carrera en la página web, para que se pueda acceder a este localhost es importante que en la misma ruta donde se encuentra el archivo “carrera.py”, también este una carpeta llamada templates y dentro de esa carpeta tener nuestro archivo “index.html”, para que de esta manera no se tenga problema alguno en acceder a la interfaz web de la carrera como se puede ver en la figura 8 a continuación:

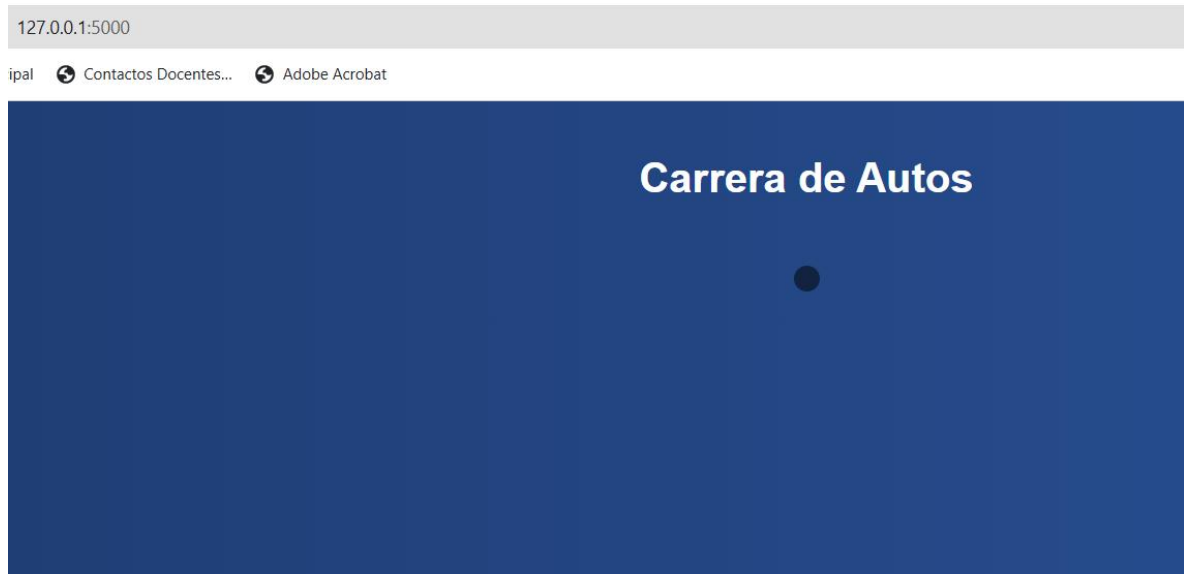


Figura 8. Interfaz Web para la carrera.

Ahora si es momento de simular la carrera, para esto tenemos que abrir otra terminal en cmd para que funcione como cliente y realizar las peticiones GET y POST necesarias, primero con un POST registraremos 4 autos para la carrera, ya que si no se detectan por lo menos 4 autos en la carrera, el servicio no deja avanzar a los autos e indica que todavía hay autos por registrarse en la carrera como se puede ver en la figura 9 a continuación:

```
Símbolo del sistema
"message": "Auto Mazda registrado exitosamente"
}
C:\Users\diego>curl -X POST "http://localhost:5000/register" -H "Content-Type: application/json" -d "{\"car_id\": \"Tsuru\"}"
{
  "message": "Auto Tsuru registrado exitosamente"
}
C:\Users\diego>curl -X POST "http://localhost:5000/move" -H "Content-Type: application/json" -d "{\"car_id\": \"Mazda\", \"distance\": 10}"
{
  "error": "La carrera iniciara cuando haya 4 autos registrados"
}
C:\Users\diego>
```

Figura 9. Error al iniciar para la carrera.

Sabiendo esto, registramos 4 autos para la carrera desde la terminal con el comando `curl -X POST "http://localhost:5000/register" -H "Content-Type: application/json" -d '{"car_id": "nombre_carro"}'`, cambiando el nombre del carro para que sean diferentes.

En esta prueba registraremos 4 autos con los nombres Mazda, Aveo, Tsuru y Ferrari como se puede ver en la figura 10 a continuación:

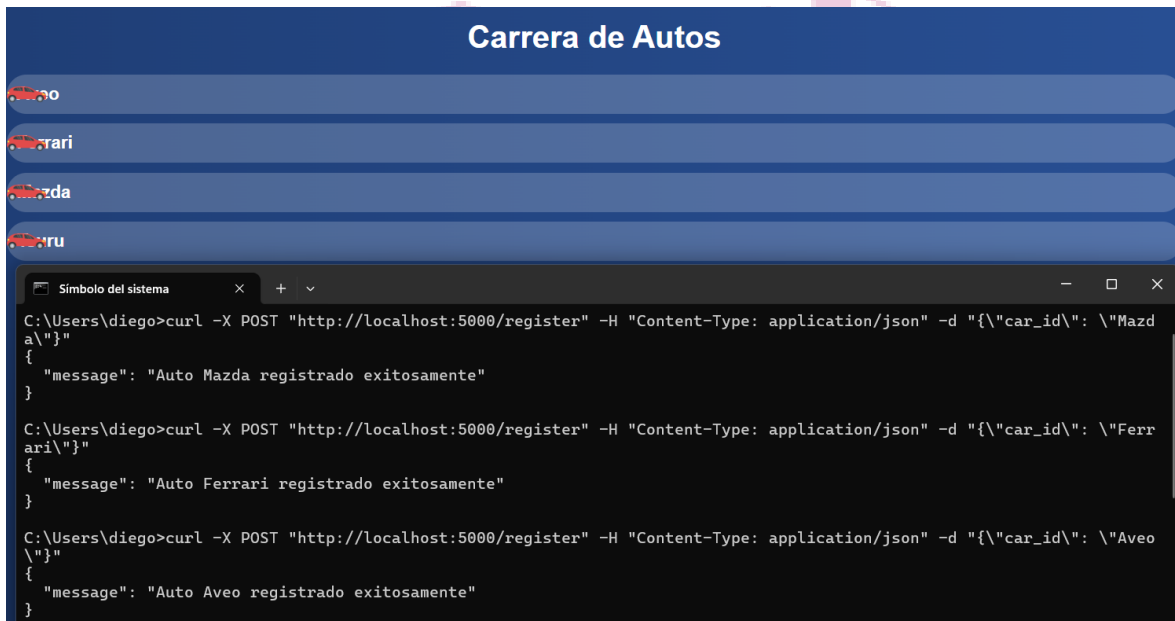


Figura 10. Registro de auto para la carrera.

Ahora si tenemos todo listo para la carrera, para poder avanzar un auto es con el comando `curl -X POST "http://localhost:5000/move" -H "Content-Type: application/json" -d '{"car_id": "carro_avanzar", "distance": avance_metros}'`, en car_id ponemos el auto que queremos avanzar y en distance ponemos los metros que queremos que avance, la carrera termina cuando todos llegan a los 100 metros y muestra el podio final conforme el orden de llegada.

Primero avanzamos 25m a Mazda, 15m a Aveo, 30m a Tsuru y 45m a Ferrari como se puede ver en la figura 11 a continuación:

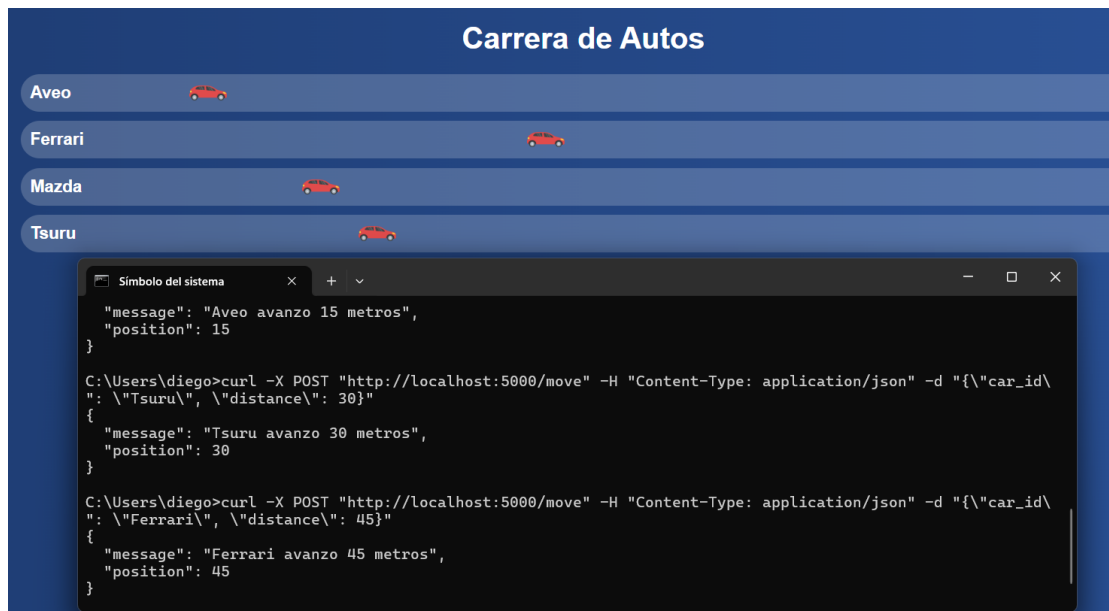


Figura 11. Primer avance para la carrera.

Después avanzamos 25 m a Mazda, 25m a Aveo, 20m a Tsuru y 10m a Ferrari, en la interfaz web podemos ver como se simula el avance de los carros conforme a los POST que ya hicimos, esta interfaz cada segundo realiza una petición GET al servidor para el status de la carrera y saber si tiene que hacer alguna actualización en la interfaz como se puede ver en la figura 12 y 13 a continuación:

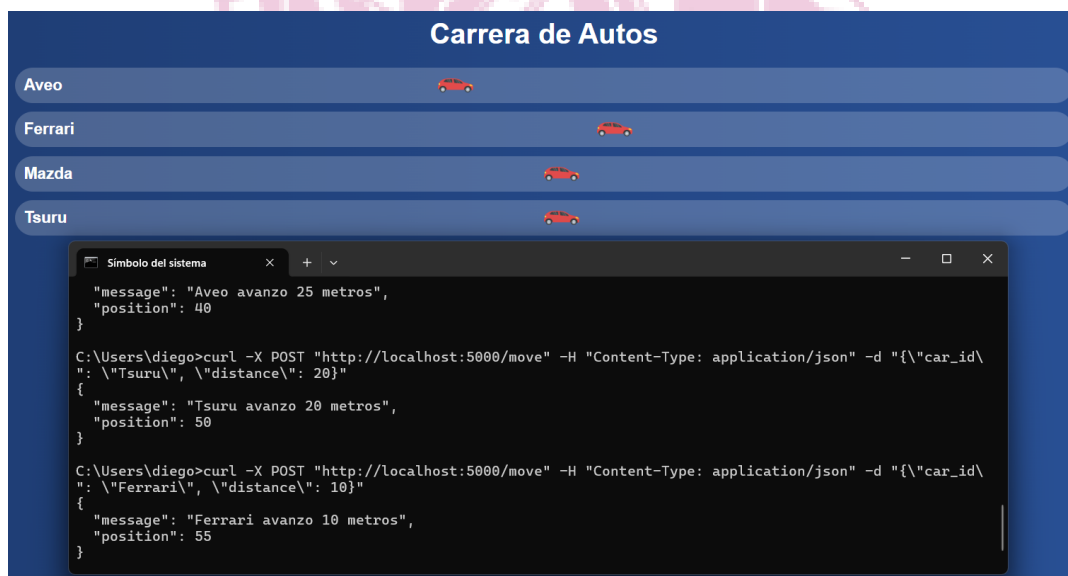


Figura 12. Segundo avance para la carrera.

```
Windows PowerShell
127.0.0.1 - - [02/Apr/2025 10:03:43] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:03:44] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:03:45] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:03:46] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:03:47] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:03:48] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:03:49] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:03:50] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:03:51] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:03:52] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:03:53] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:03:54] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:03:55] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:03:56] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:03:57] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:03:58] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:03:59] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:04:00] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:04:01] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:04:02] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:04:03] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:04:04] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:04:05] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:04:06] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:04:07] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:04:08] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:04:09] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:04:10] "GET /race_status HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 10:04:11] "GET /race_status HTTP/1.1" 200 -
```

Figura 13. Servidor con los GET de la interfaz web.

Seguimos con el avance de 25m a Mazda, 40m a Aveo, 25m a Tsuru y 40m a Ferrari como se puede ver en la figura 14 a continuación:

Carrera de Autos

Aveo

Ferrari

Mazda

Tsuru

```
Símbolo del sistema
{
  "message": "Aveo avanzo 40 metros",
  "position": 80
}

C:\Users\diego>curl -X POST "http://localhost:5000/move" -H "Content-Type: application/json" -d '{"car_id": "\Tsuru", "distance": 25}'
{
  "message": "Tsuru avanzo 25 metros",
  "position": 75
}

C:\Users\diego>curl -X POST "http://localhost:5000/move" -H "Content-Type: application/json" -d '{"car_id": "\Ferrari", "distance": 40}'
{
  "message": "Ferrari avanzo 40 metros",
  "position": 95
}
```

Figura 14. Tercer avance para la carrera.

Después avanzamos 15m a Mazda, 10m a Aveo, 15m a Tsuru y 5m a Ferrari, en este caso Ferrari ya llego a los 100 m por lo que ya no es necesario avanzar más el auto como se puede ver en la figura 15 a continuación:

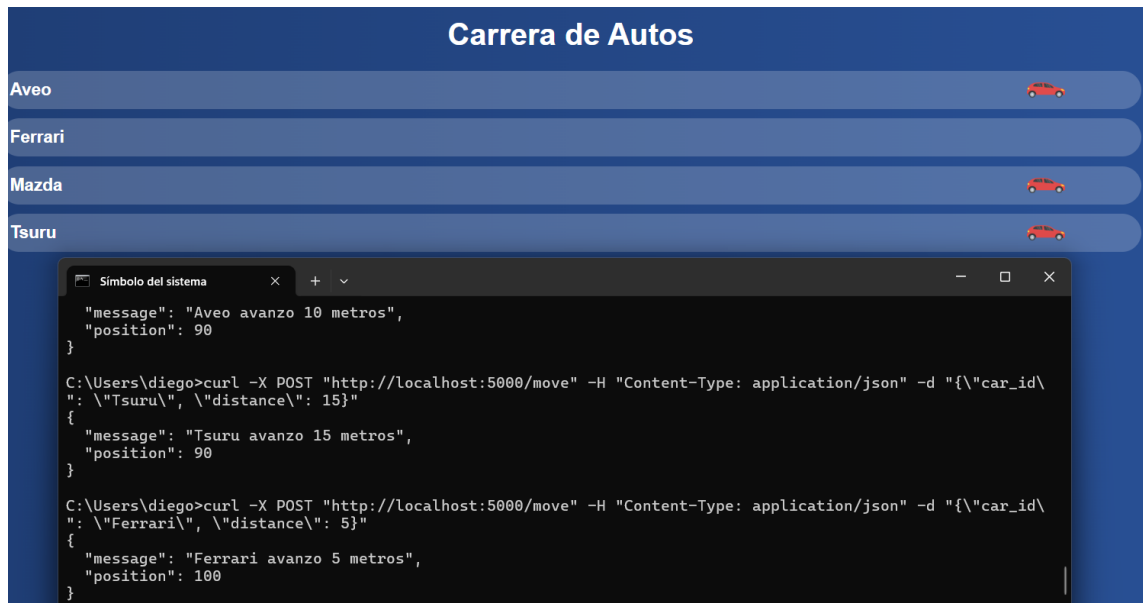


Figura 15. Cuarto avance para la carrera.

Finalmente avanzamos 10m a Mazda, 10m a Aveo y 10m a Tsuru como se puede ver en la figura 16 a continuación:

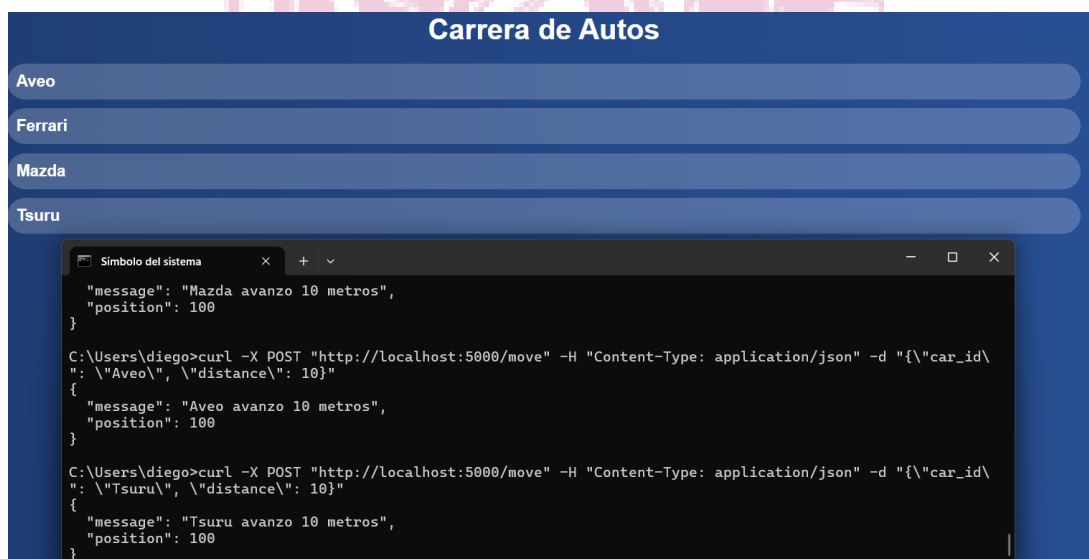
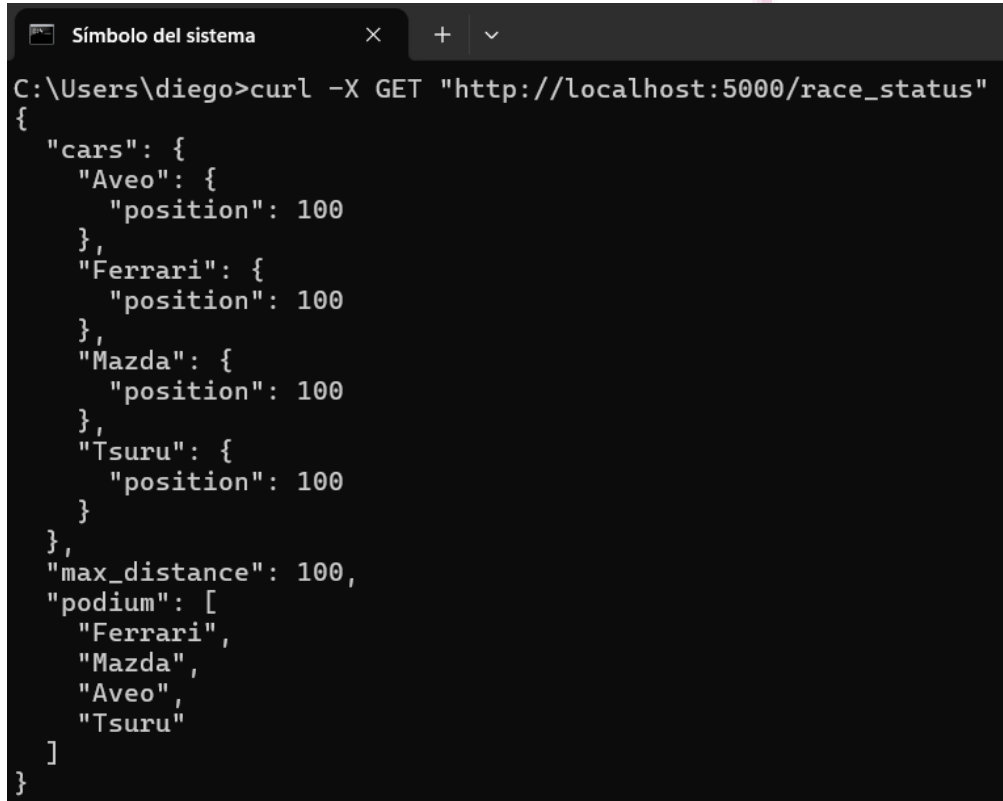


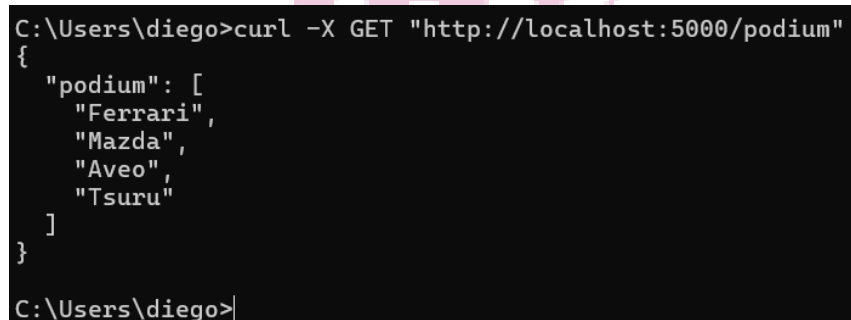
Figura 16. Último avance para la carrera.

Ahora haremos una petición GET con el comando `curl -X GET http://localhost:5000/race_status` para ver el status de la carrera y verificar que todos los autos están en 100 m, es decir, la máxima distancia y haremos otro GET con el comando `curl -X GET http://localhost:5000/podium` para observar el podio y verificar que sea el mismo que muestra la interfaz web como se puede ver en las figuras 17, 18 y 19 a continuación:



```
Símbolo del sistema
C:\Users\diego>curl -X GET "http://localhost:5000/race_status"
{
  "cars": {
    "Aveo": {
      "position": 100
    },
    "Ferrari": {
      "position": 100
    },
    "Mazda": {
      "position": 100
    },
    "Tsuru": {
      "position": 100
    }
  },
  "max_distance": 100,
  "podium": [
    "Ferrari",
    "Mazda",
    "Aveo",
    "Tsuru"
  ]
}
```

Figura 17. Status Final de la carrera.



```
C:\Users\diego>curl -X GET "http://localhost:5000/podium"
{
  "podium": [
    "Ferrari",
    "Mazda",
    "Aveo",
    "Tsuru"
  ]
}
C:\Users\diego>
```

Figura 18. Podio Final de la carrera en el CMD.



Figura 18. Podio Final de la carrera en la interfaz.

7. Conclusiones

La implementación de un sistema basado en Flask, Python y servicios web ha demostrado ser una solución efectiva para gestionar la comunicación en tiempo real dentro de un entorno distribuido. A través del uso de peticiones HTTP, específicamente las peticiones GET y POST, se ha logrado diseñar una plataforma en la que múltiples clientes interactúan con un servidor de carrera. Estas peticiones permiten registrar autos, moverlos en la carrera y obtener actualizaciones sobre el estado de la competencia sin la necesidad de gestionar conexiones directas o complejas.

Los servicios web implementados proporcionan una arquitectura simplificada para la comunicación entre los clientes y el servidor, haciendo que el proceso de interacción sea eficiente y escalable. Las peticiones GET se utilizan para obtener el estado actual de la carrera y el podio final, mientras que las peticiones POST permiten registrar un nuevo auto y actualizar su posición en la carrera. Esta separación de las peticiones para obtener información y modificar el estado de la competencia facilita la gestión de la comunicación y optimiza el rendimiento del sistema.

Los resultados obtenidos muestran que la arquitectura basada en Flask y servicios web facilita una comunicación fluida entre los clientes y el servidor, permitiendo que cada cliente reciba actualizaciones inmediatas sobre el progreso de su auto en la carrera sin bloqueos ni pérdida de datos.

8. Referencias Bibliográficas

- DigitalOcean. (n.d.). Cómo hacer una aplicación web usando Flask en Python 3. DigitalOcean. Recuperado el 1 de abril de 2025, de <https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3-es>
- Flask. (n.d.). Documentación de Flask. Recuperado el 1 de abril de 2025, de <https://flask.palletsprojects.com/en/stable/>
- Web Corporativa. (2021, 3 de mayo). ¿Qué es un servicio web y ejemplos?. Web Corporativa. Recuperado el 1 de abril de 2025, de <https://webcorporativa.es/que-es-un-servicio-web-y-ejemplos/>