

Instituto Politécnico Nacional

Escuela Superior de Cómputo

Materia: Sistemas Distribuidos

Profesor: Carreto Arellano Chadwick

Grupo: 7CM1

Práctica 7 – Microservicios

Valdés Castillo Diego - 2021630756

Fecha de Entrega: 09/04/2025

Índice

1. Antecedente (Marco teórico).....	2
2. Planteamiento del Problema	3
3. Propuesta de Solución.....	4
4. Materiales y Métodos empleados.....	6
5. Desarrollo de la Solución.....	9
6. Resultados	16
7. Conclusiones.....	26
8. Referencias Bibliográficas	27

1. Antecedente (Marco teórico)

En el desarrollo de los sistemas distribuidos, el enfoque basado en microservicios se ha consolidado como una arquitectura eficiente para el desarrollo de aplicaciones modulares, escalables y fácilmente mantenibles. Esta arquitectura propone dividir un sistema complejo en pequeños servicios independientes, cada uno ejecutándose en su propio proceso y comunicándose con otros servicios mediante protocolos ligeros como HTTP y utilizando formatos estándar como JSON o XML.

El modelo de microservicios permite que cada componente del sistema sea desarrollado, desplegado y escalado de forma autónoma. Esto facilita la implementación de cambios, mejora la tolerancia a fallos y permite la reutilización de componentes en distintas aplicaciones. A diferencia de los enfoques tradicionales centrados en arquitecturas monolíticas o en esquemas de comunicación como sockets, los microservicios promueven una mayor flexibilidad y adaptabilidad frente a cambios en los requerimientos del sistema.

En este contexto, el uso de frameworks como Flask, un microframework escrito en Python, es una herramienta adecuada para construir aplicaciones distribuidas basadas en microservicios. Flask proporciona una interfaz sencilla para definir rutas

(endpoints), manejar peticiones HTTP (GET, POST, etc.) y estructurar respuestas en formatos compatibles con clientes diferentes.

Los microservicios también fomentan una organización clara del sistema, ya que cada servicio se asocia a una única responsabilidad. Esta separación de preocupaciones mejora la cohesión interna y reduce el acoplamiento entre los módulos, elementos fundamentales en el diseño de software distribuido. Asimismo, al implementar servicios independientes que se comunican por red, se simula de forma realista el comportamiento de sistemas distribuidos que deben lidiar con latencias, errores de red y sincronización de datos.

La arquitectura de microservicios representa un enfoque moderno y eficiente para el diseño de sistemas distribuidos. Su implementación con herramientas como Flask permite explorar conceptos clave como la independencia de servicios, la comunicación mediante interfaces REST, el aislamiento de procesos y la tolerancia a fallos, todos ellos fundamentales para la comprensión y construcción de soluciones distribuidas robustas.

2. Planteamiento del Problema

Esta práctica tiene como objetivo fortalecer el conocimiento en el desarrollo de sistemas distribuidos mediante el uso de una arquitectura basada en microservicios, utilizando el protocolo HTTP y el intercambio de datos en formato JSON. En este caso, se propone la implementación de un conjunto de servicios independientes que colaboran para gestionar una simulación de carrera de autos. Cada uno de estos servicios se encarga de tareas específicas como el registro de autos, la simulación del avance y la generación del podio final.

El desafío principal radica en garantizar una comunicación eficiente y coherente entre los diferentes servicios, especialmente considerando que la carrera avanza de manera dinámica y debe mantenerse sincronizada en tiempo real. Cada microservicio debe operar de forma autónoma pero coordinada, respondiendo

correctamente a las peticiones de los clientes que desean registrar autos, iniciar la carrera, consultar el estado de los autos en la pista o conocer el podio.

Para lograrlo, se deben definir claramente las rutas REST de cada microservicio, permitiendo a los clientes realizar peticiones POST para registrar y avanzar autos, y GET para obtener información del estado de la carrera o los resultados. La arquitectura debe garantizar que los servicios puedan procesar múltiples solicitudes concurrentes sin generar bloqueos, condiciones de carrera ni pérdida de información.

Además, es fundamental que el sistema detecte y maneje correctamente los estados del flujo de la carrera, como impedir el avance de un auto que no ha sido registrado o que ya ha finalizado la simulación. Este tipo de validaciones asegura la integridad del sistema y permite mantener una lógica coherente a lo largo de toda la ejecución.

Finalmente, al estar distribuido en microservicios, el sistema debe ser lo suficientemente robusto, escalable y tolerante a fallos, de modo que pueda responder con precisión a cada solicitud, permitiendo una experiencia fluida y en tiempo real para los clientes.

3. Propuesta de Solución

Para abordar este problema, se propone desarrollar un sistema distribuido basado en microservicios, donde cada servicio cumple una función específica dentro de la simulación de una carrera de autos. Esta arquitectura permite escalar y mantener los componentes de manera independiente, al mismo tiempo que garantiza una comunicación eficiente mediante el uso de peticiones HTTP.

La solución está conformada por los siguientes componentes principales:

- **Microservicio de Registro de Autos:**
 - Este servicio se encarga de gestionar el alta de nuevos autos en la carrera.

- Expone una ruta POST (por ejemplo, /register) donde los clientes envían los datos del auto a registrar.
- Almacena la información en una base de datos o repositorio temporal compartido entre los servicios.
- **Microservicio de Simulación de Carrera:**
 - Administra el avance de los autos registrados.
 - Expone una ruta POST (/move) para recibir las solicitudes de movimiento y actualiza la posición de cada auto según una lógica de avance aleatoria.
 - Se asegura de que un auto no avance si no ha sido registrado o si ya ha llegado a la meta.
- **Microservicio de Estado de Carrera:**
 - Permite a los clientes consultar el progreso actual de la carrera.
 - A través de una ruta GET (/race_status), proporciona la información en tiempo real sobre las posiciones de los autos en la pista.
 - Este servicio consulta los datos centralizados para garantizar consistencia.
- **Microservicio de Podio:**
 - Gestiona y entrega los resultados finales de la carrera.
 - Mediante una ruta GET (/podium), permite a los clientes conocer qué autos han llegado en los primeros lugares.
 - Este servicio se activa automáticamente cuando todos los autos han cruzado la meta.
- **Interfaz web:**
 - Los usuarios interactúan con el sistema a través de una página HTML con soporte de JavaScript.
 - Usando funciones como setInterval o fetch, la interfaz envía peticiones periódicas a los microservicios para consultar el estado de la carrera y visualizar el avance de los autos.
 - La pista de carrera se representa visualmente y se actualiza dinámicamente cada segundo.

- **Comunicación entre servicios mediante HTTP (GET y POST):**

- Cada microservicio opera de forma independiente, pero se comunica a través de llamadas HTTP utilizando JSON como formato de intercambio de datos.
- Las peticiones POST se emplean para acciones que modifican el estado (registro y movimiento), mientras que las GET se utilizan para obtener información sin alterar los datos (estado de carrera y podio).

Esta solución garantiza un sistema más modular, escalable y mantenible, permitiendo el despliegue individual de servicios y una mayor tolerancia a fallos. Además, la separación de responsabilidades entre microservicios mejora la eficiencia en la gestión de datos, evita cuellos de botella en el procesamiento, y asegura una experiencia de usuario fluida y actualizada en tiempo real.

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:5001/register" -H "Content-Type: application/json" -d "{\"car_id\": \"car_name\"}"`
 - `curl -X POST "http://localhost:5002/move" -H "Content-Type: application/json" -d "{\"car_id\": \"car_name\", \"distance\": metros}"`
 - `curl -X GET "http://localhost:5003/race_status"`
 - `curl -X GET "http://localhost:5004/podium"`
 - `python nombre_codigo.py`

- **Bibliotecas y tecnologías utilizadas:**
 - **Python + Flask:** Cada microservicio se implementa utilizando Flask para exponer endpoints HTTP que permiten la interacción con el sistema.
 - **Requests:** Biblioteca utilizada para realizar peticiones HTTP entre microservicios (cuando es necesario) y desde el cliente.
 - **JSON y jsonify:** Se emplea para intercambiar datos estructurados entre servicios y clientes en un formato ligero y estándar.
 - **Render_template:** Utilizado en el microservicio de interfaz web para presentar páginas HTML al usuario.
 - **JavaScript:** Para manejar la interacción dinámica en el navegador, haciendo peticiones periódicas a los microservicios para obtener información en tiempo real.
- **Métodos empleados:**
 - **Microservicio de Registro de Autos:**
 - Expone la ruta /register para permitir el registro de autos.
 - Verifica si el auto ya está registrado antes de agregarlo al repositorio central.
 - Almacena temporalmente la información en memoria compartida (por ejemplo, Redis) o en una base de datos sencilla.
 - **Microservicio de Movimiento:**
 - Administra la lógica de avance de los autos mediante la ruta /move.
 - Solo permite avanzar a los autos registrados y no permite que sigan avanzando si ya han alcanzado la meta.
 - Actualiza la posición de cada auto simulando el progreso con valores aleatorios.
 - **Microservicio de Estado de Carrera:**
 - Expuesto mediante /race_status, devuelve información detallada sobre las posiciones de los autos en tiempo real.

- Este microservicio consulta el repositorio común y consolida los datos para el cliente.
- **Microservicio de Podio:**
 - Proporciona el resultado final de la carrera a través de la ruta /podium.
 - Solo se activa cuando todos los autos han finalizado la carrera, clasificándolos según su orden de llegada.
- **Interfaz Web del Cliente:**
 - Se desarrolla una página HTML que se comunica con los microservicios mediante JavaScript.
 - Utiliza setInterval para realizar peticiones GET cada segundo a los servicios /race_status y /podium.
 - La pista de carrera se representa visualmente y se actualiza dinámicamente, mostrando a los autos en movimiento en función de sus posiciones.
- **Estrategia de programación:**
 - El sistema está diseñado siguiendo el modelo de microservicios distribuidos, donde cada servicio es independiente pero se comunica eficientemente mediante peticiones HTTP.
 - Esto permite que cada servicio pueda escalar de forma individual y mantenerse sin afectar a los demás.
 - La lógica de carrera, validaciones y sincronización se maneja dentro de cada servicio responsable, asegurando que no existan conflictos en el flujo de datos.
 - Gracias a la división funcional, se mejora la robustez, la eficiencia y la mantenibilidad del sistema, garantizando actualizaciones en tiempo real y un comportamiento confiable durante toda la simulación.

5. Desarrollo de la Solución

Para implementar la solución de la simulación de una carrera de autos utilizando un enfoque distribuido, se desarrolló una aplicación basada en microservicios, donde cada componente de la lógica de la carrera está encapsulado en un servicio independiente que se comunica a través de peticiones HTTP. La tecnología base incluye Python con Flask para los servicios backend y HTML + JavaScript para la interfaz web.

El proceso comienza cuando un cliente accede a la interfaz web alojada por el microservicio de presentación, el cual permite la interacción con los diferentes servicios a través de peticiones HTTP (GET y POST). Desde la interfaz, los usuarios pueden ver las acciones de los microservicios que son registrar autos, iniciar la carrera, consultar el estado de la competencia y visualizar el podio, todo en tiempo real.

Cada microservicio tiene una función específica:

- **Microservicio de Registro:** permite a los usuarios registrar autos nuevos en la carrera mediante peticiones POST. Este servicio almacena los autos en una estructura compartida o base de datos ligera.
- **Microservicio de Movimiento:** administra el avance de los autos registrados. Cada vez que recibe una solicitud POST de movimiento, calcula el desplazamiento aleatorio del auto y actualiza su posición, siempre que este aún no haya llegado a la meta.
- **Microservicio de Estado de Carrera:** ofrece la información consolidada del estado actual de la competencia, respondiendo a las consultas GET del cliente.
- **Microservicio de Podio:** una vez que los autos han cruzado la meta, este servicio gestiona el orden de llegada y expone los resultados finales de la carrera.
- **Microservicio de Interfaz Web:** administra la interfaz web en el puerto 5000.

Para que la simulación comience, el microservicio de control verifica que al menos 4 autos hayan sido registrados. Una vez cumplido este requisito, los autos pueden empezar a moverse, y cada movimiento se gestiona de forma independiente por el microservicio correspondiente. La posición de cada auto es actualizada y mantenida de forma consistente en el sistema.

Durante la carrera, la interfaz cliente realiza consultas periódicas (cada segundo) al microservicio de estado mediante peticiones GET. Esta información es utilizada para actualizar visualmente la pista de carrera en la página web, reflejando el progreso de cada auto de forma dinámica. A medida que los autos cruzan la línea de meta, el microservicio de podio los registra y limita sus movimientos posteriores.

Finalmente, cuando todos los autos han terminado la carrera, se generan los resultados finales y estos son enviados al cliente a través del microservicio de podio. La interfaz web muestra estos resultados y da por concluida la simulación. Gracias a la separación de responsabilidades en microservicios, la aplicación logra una mayor escalabilidad, mantenibilidad y robustez en su funcionamiento, garantizando 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, 6, 7, 8, 9, 10 y 11:

```
D: > Escuela8voSem > Distribuidos > Practica7 > frontend_service.py > ...
1  #Materia: Sistemas Distribuidos
2  #Autor: Diego Valdes Castillo
3  #Fecha de creacion: 05/04/2025
4  #Version: 1.0
5  #Practica 7 - Microservicios
6  # frontend_service.py - Flask
7  from flask import Flask, render_template, jsonify
8  import requests
9
10 app = Flask(__name__)
11
12 @app.route('/')
13 def index():
14     return render_template('index.html')
15
16 @app.route('/race_status')
17 def race_status():
18     r = requests.get("http://localhost:5003/race_status")
19     return jsonify(r.json())
20
21 if __name__ == '__main__':
22     app.run(port=5000, debug=True)
```

Figura 1. Código en Python – Microservicio Frontend.

```

D: > Escuela8voSem > Distribuidos > Practica7 > register_service.py > register_car
1  #Materia: Sistemas Distribuidos
2  #Autor: Diego Valdes Castillo
3  #Fecha de creacion: 05/04/2025
4  #Version: 1.0
5  #Practica 7 - Microservicios
6  # register_service.py - Flask
7  from flask import Flask, request, jsonify
8  import requests
9
10 app = Flask(__name__)
11 STATE_SERVICE_URL = "http://localhost:5003/data"
12
13 @app.route('/register', methods=['POST'])
14 def register_car():
15     data = request.get_json()
16     car_id = data.get('car_id')
17     if not car_id:
18         return jsonify({"error": "Debes proporcionar un ID de auto"}), 400
19
20     payload = {
21         "action": "register",
22         "car_id": car_id
23     }
24
25     try:
26         r = requests.post(STATE_SERVICE_URL, json=payload)
27         return r.json(), r.status_code
28     except Exception as e:
29         return jsonify({"error": str(e)}), 500
30
31 if __name__ == '__main__':
32     app.run(port=5001, debug=True)

```

Figura 2. Código en Python – Microservicio Register.

```

D: > Escuela8voSem > Distribuidos > Practica7 > move_service.py > ...
1  #Materia: Sistemas Distribuidos
2  #Autor: Diego Valdes Castillo
3  #Fecha de creacion: 05/04/2025
4  #Version: 1.0
5  #Practica 7 - Microservicios
6  # move_service.py - Flask
7  from flask import Flask, request, jsonify
8  import requests
9
10 app = Flask(__name__)
11 STATE_SERVICE_URL = "http://localhost:5003/data"
12
13 @app.route('/move', methods=['POST'])
14 def move_car():
15     data = request.get_json()
16     car_id = data.get('car_id')
17     distance = data.get('distance', 5)
18
19     payload = {
20         "action": "move",
21         "car_id": car_id,
22         "distance": distance
23     }
24
25     try:
26         r = requests.post(STATE_SERVICE_URL, json=payload)
27         return r.json(), r.status_code
28     except Exception as e:
29         return jsonify({"error": str(e)}), 500
30
31 if __name__ == '__main__':
32     app.run(port=5002, debug=True)

```

Figura 3. Código en Python – Microservicio Move.

```

D: > Escuela8voSem > Distribuidos > Practica7 > state_service.py > ...
1  #Materia: Sistemas Distribuidos
2  #Autor: Diego Valdes Castillo
3  #Fecha de creacion: 05/04/2025
4  #Version: 1.0
5  #Practica 7 - Microservicios
6  # state_service.py -Flask
7  from flask import Flask, request, jsonify
8
9  app = Flask(__name__)
10
11  race_status = {
12      "cars": {},
13      "max_distance": 100,
14      "podium": []
15  }
16
17  @app.route('/data', methods=['POST'])
18  def update_race_data():
19      data = request.get_json()
20      action = data.get('action')
21      car_id = data.get('car_id')
22
23      if action == "register":
24          if car_id in race_status["cars"]:
25              return jsonify({"error": "El auto ya está registrado"}), 400
26          race_status["cars"][car_id] = {"position": 0}
27          return jsonify({"message": f"Auto {car_id} registrado exitosamente"})
28
29      elif action == "move":
30          if car_id not in race_status["cars"]:
31              return jsonify({"error": "El auto no está registrado"}), 400
32          if car_id in race_status["podium"]:
33              return jsonify({"error": "Este auto ya terminó la carrera"}), 400
34          if len(race_status["cars"]) < 4:
35              return jsonify({"error": "La carrera inicia cuando haya al menos 4 autos"}), 400

```

Figura 4. Código en Python – Microservicio State - 1.

```

36
37      distance = data.get("distance", 5)
38      race_status["cars"][car_id]["position"] += distance
39
40      if race_status["cars"][car_id]["position"] >= race_status["max_distance"]:
41          race_status["cars"][car_id]["position"] = race_status["max_distance"]
42      if car_id not in race_status["podium"]:
43          race_status["podium"].append(car_id)
44
45      return jsonify({"message": f"{car_id} avanzó {distance} metros", "position": race_status["cars"][car_id]["position"]})
46
47      return jsonify({"error": "Acción inválida"}), 400
48
49  @app.route('/race_status', methods=['GET'])
50  def get_race_status():
51      return jsonify(race_status)
52
53  if __name__ == '__main__':
54      app.run(port=5003, debug=True)
55

```

Figura 5. Código en Python – Microservicio State - 2.

```

D: > Escuela8voSem > Distribuidos > Practica7 > podium_service.py > ...
1  #Materia: Sistemas Distribuidos
2  #Autor: Diego Valdes Castillo
3  #Fecha de creacion: 05/04/2025
4  #Version: 1.0
5  #Practica 7 - Microservicios
6  # podium_service.py - Flask
7  from flask import Flask, jsonify
8  import requests
9
10 app = Flask(__name__)
11 STATE_SERVICE_URL = "http://localhost:5003/race_status"
12
13 @app.route('/podium', methods=['GET'])
14 def get_podium():
15     try:
16         r = requests.get(STATE_SERVICE_URL)
17         data = r.json()
18         podium = data.get("podium", [])
19         if len(podium) >= 4:
20             return jsonify({"podium": podium})
21         return jsonify({"message": "La carrera aún no termina"})
22     except Exception as e:
23         return jsonify({"error": str(e)}), 500
24
25 if __name__ == '__main__':
26     app.run(port=5004, debug=True)

```

Figura 6. Código en Python – Microservicio Podium.

```

D: > Escuela8voSem > Distribuidos > Practica7 > templates > index.html > ...
1  <!--Materia: Sistemas Distribuidos
2  #Autor: Diego Valdes Castillo
3  #Fecha de creacion: 05/04/2025
4  #Version: 1.0
5  #Practica 7 - Microservicios
6  #Codigo para la carrera- Interfaz Web-->
7  <!-- templates/index.html -->
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: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
17             background: linear-gradient(to right, #1e3c72, #2a5298);
18             color: white;
19             text-align: center;
20             padding: 20px;
21         }
22
23         h1 {
24             font-size: 3em;
25             margin-bottom: 20px;
26         }
27
28         .track-container {
29             width: 90%;
30             margin: 20px auto;
31             text-align: left;
32         }
33
34         .car-name-label {
35             margin-left: 10px;
36             margin-bottom: 5px;

```

Figura 7. Código en Python – Interfaz Web - 1.

```

37     font-weight: bold;
38     color: #ffeb3b;
39     text-shadow: 1px 1px 3px black;
40     font-size: 1.1em;
41 }
42
43 .track {
44     background: rgba(255,255,255,0.1);
45     border: 2px solid rgba(255,255,255,0.2);
46     border-radius: 25px;
47     height: 60px;
48     position: relative;
49     overflow: hidden;
50 }
51
52 .car {
53     position: absolute;
54     width: 50px;
55     height: 50px;
56     transition: left 0.5s;
57     top: 5px;
58 }
59
60 .car img {
61     width: 100%;
62     transform: scaleX(-1);
63 }
64
65 .podium {
66     margin-top: 40px;
67 }

```

Figura 8. Código en Python – Interfaz Web - 2.

```

69 .podium h2 {
70     font-size: 2em;
71     margin-bottom: 15px;
72     color: #ffd700;
73 }
74
75 .podium p {
76     background: rgba(0, 0, 0, 0.3);
77     display: inline-block;
78     margin: 5px 10px;
79     padding: 10px 20px;
80     border-radius: 20px;
81     font-weight: bold;
82     font-size: 1.2em;
83     color: #ffffff;
84     border: 2px solid #ffffff22;
85     box-shadow: 0 0 10px rgba(0,0,0,0.3);
86 }
87
88 .podium p:nth-child(2) {
89     background-color: #ffd700;
90     color: #000;
91 }
92
93 .podium p:nth-child(3) {
94     background-color: #c0c0c0;
95     color: #000;
96 }
97
98 .podium p:nth-child(4) {
99     background-color: #cd7f32;
100    color: #000;

```

Figura 9. Código en Python – Interfaz Web - 3.

```

101     }
102   </style>
103 </head>
104 <body>
105   <h1>🏁 Carrera de Autos 🏁</h1>
106   <div id="race"></div>
107   <div class="podium" id="podium"></div>
108
109   <script>
110     async function fetchRaceStatus() {
111       const res = await fetch("/race_status");
112       const data = await res.json();
113       const raceDiv = document.getElementById('race');
114       raceDiv.innerHTML = "";
115
116       for (const [car, info] of Object.entries(data.cars)) {
117         const trackContainer = document.createElement('div');
118         trackContainer.className = "track-container";
119
120         const nameLabel = document.createElement('div');
121         nameLabel.className = "car-name-label";
122         nameLabel.innerText = car;
123
124         const track = document.createElement('div');
125         track.className = "track";
126
127         const carDiv = document.createElement('div');
128         carDiv.className = "car";
129         carDiv.style.left = `${(info.position / data.max_distance) * 100}%`;
130         carDiv.innerHTML = ``;
131

```

Figura 10. Código en Python – Interfaz Web - 4.

```

132       track.appendChild(carDiv);
133       trackContainer.appendChild(nameLabel);
134       trackContainer.appendChild(track);
135       raceDiv.appendChild(trackContainer);
136     }
137
138     const podium = data.podium;
139     const podiumDiv = document.getElementById('podium');
140     if (podium.length > 0) {
141       podiumDiv.innerHTML = `<h2>🏆 Podio 🏆</h2>` +
142         podium.map((c, i) => `<p>${i + 1}. ${c}</p>`).join('');
143     }
144   }
145
146   setInterval(fetchRaceStatus, 1000);
147   fetchRaceStatus();
148 </script>
149 </body>
150 </html>
151

```

Figura 11. Código en Python – Interfaz Web - 5.

6. Resultados

Los resultados obtenidos evidencian que la arquitectura basada en microservicios permite una comunicación eficiente y modular entre los distintos componentes del sistema y los clientes. Cada microservicio responde adecuadamente a las peticiones HTTP (GET y POST), lo que garantiza que la simulación de la carrera se mantenga consistente, sin bloqueos ni pérdida de información, incluso con múltiples solicitudes simultáneas.

Durante las ejecuciones de prueba, se observó que el avance aleatorio de los autos genera distintos resultados en cada carrera, ofreciendo una experiencia dinámica y realista. Este comportamiento se logró gracias al microservicio de movimiento, que procesa de forma independiente las solicitudes para cada auto. La independencia de los servicios asegura que los autos se desplacen sin interferencias entre sí, reflejando la naturaleza impredecible de una competencia real.

El microservicio de control demostró funcionar correctamente al validar que la carrera no iniciara hasta tener al menos cuatro autos registrados. Además, el microservicio de estado respondió eficazmente a las consultas periódicas desde los clientes, proporcionando datos actualizados en tiempo real sobre la posición de los autos, lo que permitió una visualización fluida y sincronizada desde la interfaz web.

Una vez que todos los autos cruzaron la meta, el microservicio de podio generó el ranking final y lo distribuyó correctamente a los clientes conectados. La interfaz mostró estos resultados de manera inmediata, asegurando una experiencia de usuario sin interrupciones.

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 12 podemos ver la ejecución del archivo “frontend_service.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 frontend_servivce.py”.


```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Instale la versión más reciente de PowerShell para obtener nuevas características y mejoras. https://aka.ms/PSWindows

PS D:\Escuela8voSem\Distribuidos\Practica7> python frontend_service.py
* Serving Flask app 'frontend_service'
* 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 http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 375-568-033
```

Figura 12. Frontend con flask en el puerto 5000.

Ahora en la figura 13 podemos ver el ejecución del archivo “register_service.py” para que aloje el servidor de flask en <http://127.0.0.1:5001>, esto lo hacemos desde la terminal con el comando “python register service.py”.

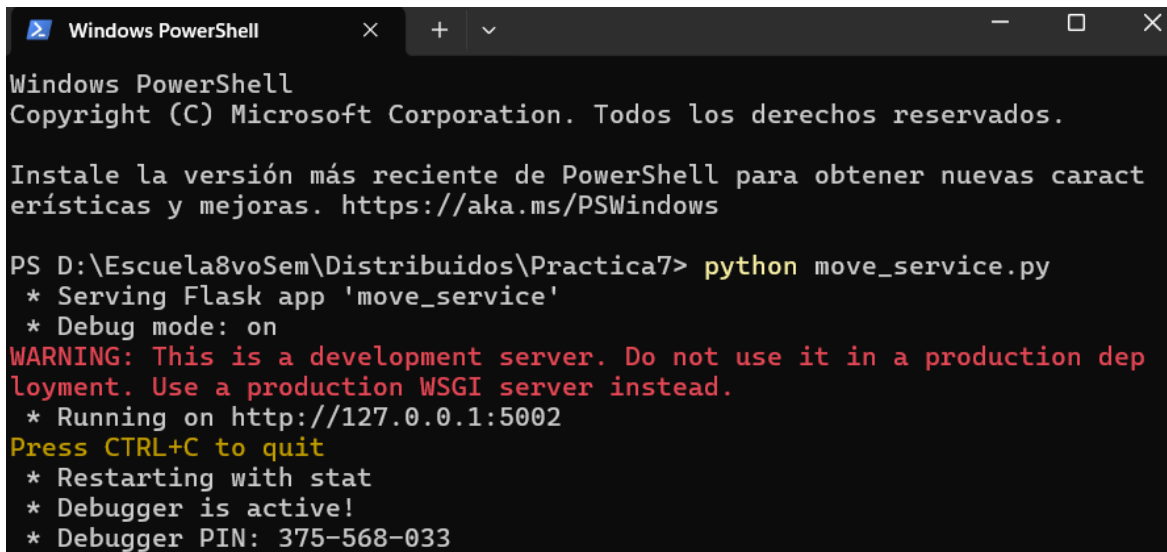
```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Instale la versión más reciente de PowerShell para obtener nuevas características y mejoras. https://aka.ms/PSWindows

PS D:\Escuela8voSem\Distribuidos\Practica7> python register_service.py
* Serving Flask app 'register_service'
* 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 http://127.0.0.1:5001
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 375-568-033
```

Figura 13. Register con flask en el puerto 5001.

Después en la figura 14 podemos ver el ejecución del archivo “move_service.py” para que aloje el servidor de flask en <http://127.0.0.1:5002>, esto lo hacemos desde la terminal con el comando “python move_serivce.py”.

A screenshot of a Windows PowerShell terminal window. The title bar says 'Windows PowerShell'. The text inside shows the standard PowerShell startup messages, including the copyright notice for Microsoft Corporation and a link to update PowerShell. The user has entered the command 'python move_service.py' at the prompt. The output shows that a Flask app named 'move_service' is being served in debug mode on http://127.0.0.1:5002. It includes a warning about using a development server and instructions to press CTRL+C to quit. The debugger is active with a PIN of 375-568-033.

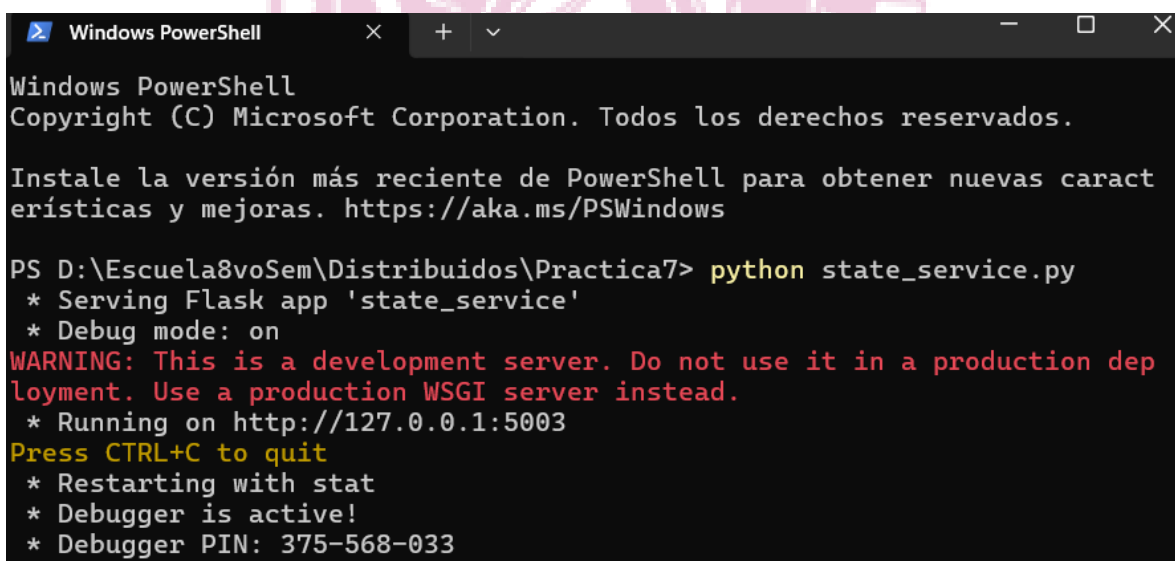
```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Instale la versión más reciente de PowerShell para obtener nuevas caract
erísticas y mejoras. https://aka.ms/PSWindows

PS D:\Escuela8voSem\Distribuidos\Practica7> python move_service.py
* Serving Flask app 'move_service'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production dep
loyment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5002
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 375-568-033
```

Figura 14. Move con flask en el puerto 5002.

Luego en la figura 15 podemos ver el ejecución del archivo “state_service.py” para que aloje el servidor de flask en <http://127.0.0.1:5003>, esto lo hacemos desde la terminal con el comando “python state_serivce.py”.

A screenshot of a Windows PowerShell terminal window, similar to the previous one. The user has entered the command 'python state_service.py'. The output shows that a Flask app named 'state_service' is being served in debug mode on http://127.0.0.1:5003. It includes the same warning about using a development server and instructions to press CTRL+C to quit. The debugger is active with a PIN of 375-568-033.

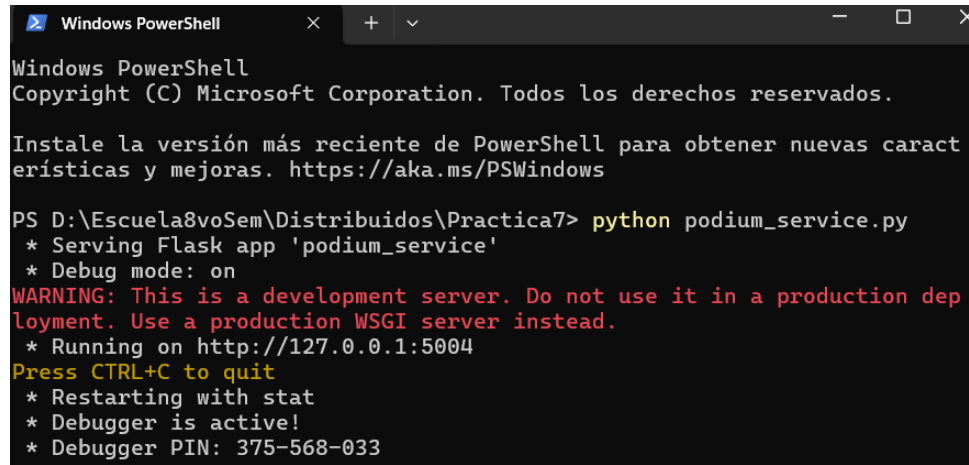
```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Instale la versión más reciente de PowerShell para obtener nuevas caract
erísticas y mejoras. https://aka.ms/PSWindows

PS D:\Escuela8voSem\Distribuidos\Practica7> python state_service.py
* Serving Flask app 'state_service'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production dep
loyment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5003
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 375-568-033
```

Figura 15. State con flask en el puerto 5003.

Finalmente en la figura 16 podemos ver el ejecución del archivo “podium_service.py” para que aloje el servidor de flask en <http://127.0.0.1:5004>, esto lo hacemos desde la terminal con el comando “python podium_servive.py”.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Instale la versión más reciente de PowerShell para obtener nuevas caract
erísticas y mejoras. https://aka.ms/PSWindows

PS D:\Escuela8voSem\Distribuidos\Practica7> python podium_service.py
* Serving Flask app 'podium_service'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production dep
loyment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5004
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 375-568-033
```

Figura 16. Podium con flask en el puerto 5004.

Una vez que ya están listos todos nuestros microservicios en sus puertos correspondientes, 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 encuentren todos los archivos “.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 17 a continuación:

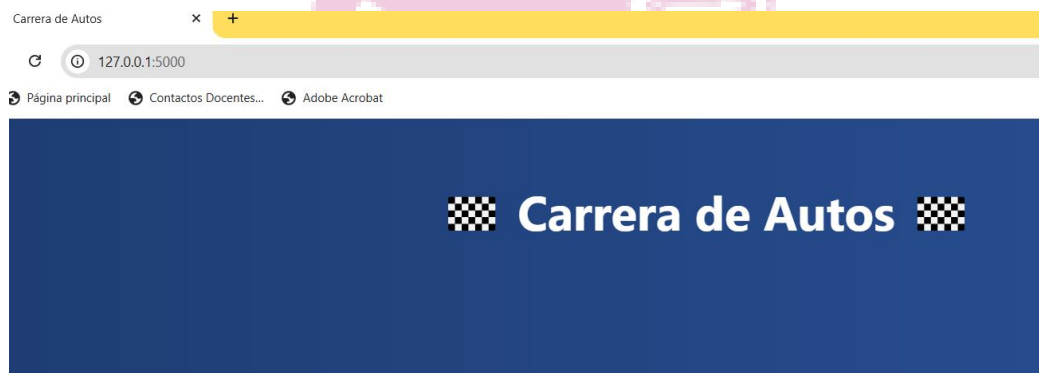
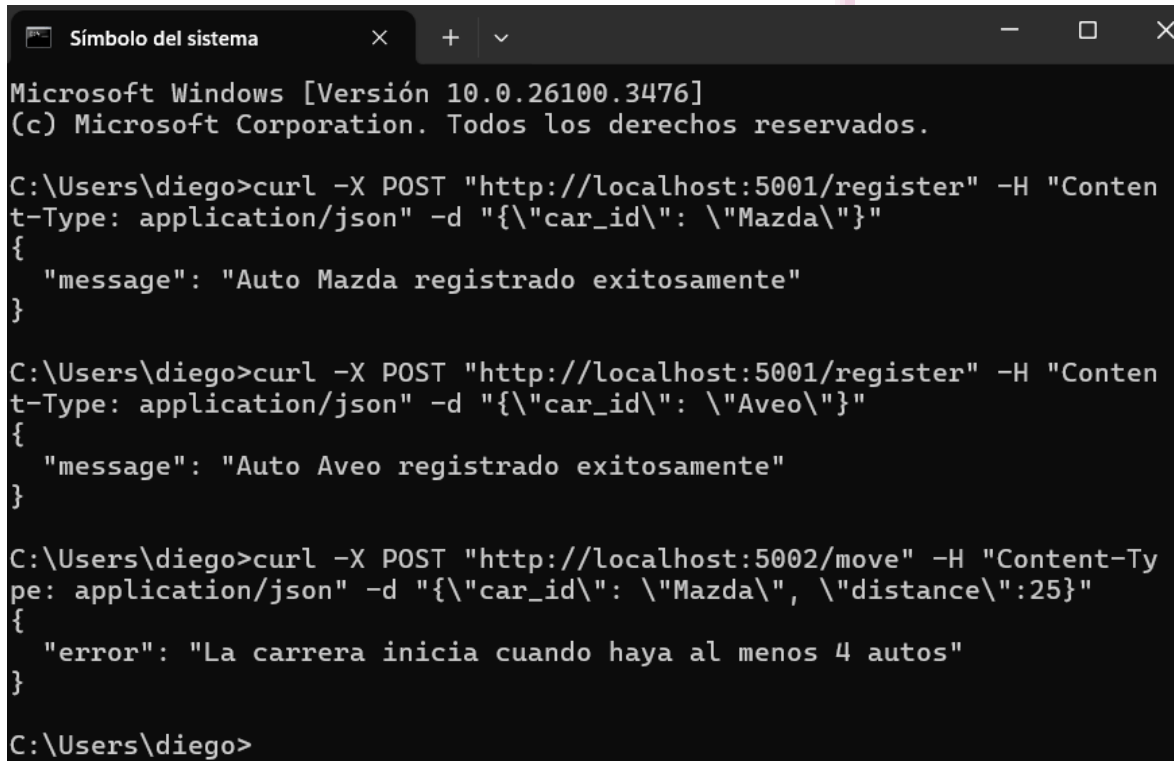


Figura 17. Interfaz Web para la carrera.

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 18 a continuación:



```
Símbolo del sistema
Microsoft Windows [Versión 10.0.26100.3476]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\diego>curl -X POST "http://localhost:5001/register" -H "Content-Type: application/json" -d "{\"car_id\": \"Mazda\"}"
{
  "message": "Auto Mazda registrado exitosamente"
}

C:\Users\diego>curl -X POST "http://localhost:5001/register" -H "Content-Type: application/json" -d "{\"car_id\": \"Aveo\"}"
{
  "message": "Auto Aveo registrado exitosamente"
}

C:\Users\diego>curl -X POST "http://localhost:5002/move" -H "Content-Type: application/json" -d "{\"car_id\": \"Mazda\", \"distance\":25}"
{
  "error": "La carrera inicia cuando haya al menos 4 autos"
}

C:\Users\diego>
```

Figura 18. Error para iniciar la carrera.

Sabiendo esto, registramos 4 autos para la carrera desde la terminal con el comando `curl -X POST "http://localhost:5001/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 20 a continuación:

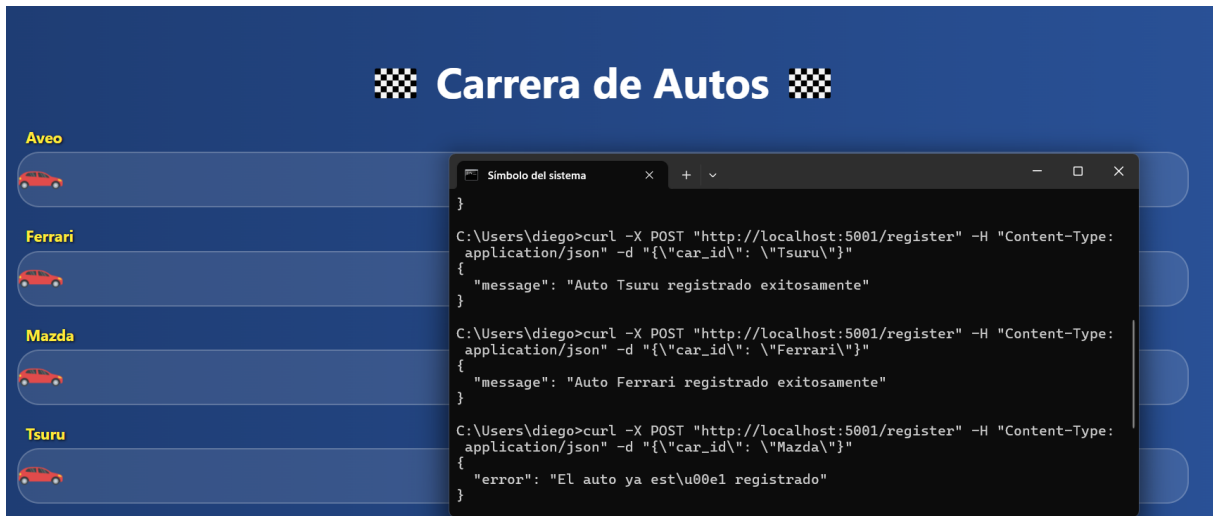


Figura 20. Registro de autos para la carrera.

Ahora si tenemos todo listo para la carrera, para poder avanzar un auto es con el comando `curl -X POST "http://localhost:5002/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 21 a continuación:



Figura 21. Primer avance para la carrera.

Después avanzamos 25m 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 22 y 23 a continuación:



Figura 22. Segundo avance para la carrera.

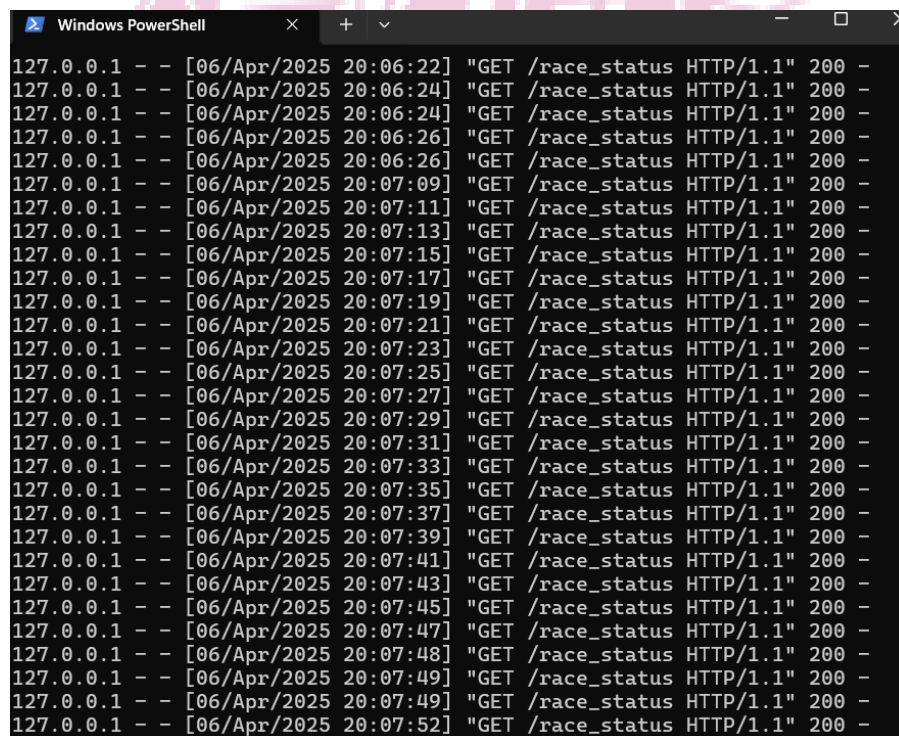


Figura 23. Frontend_service con los GET de la interfaz web.

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 24 a continuación:

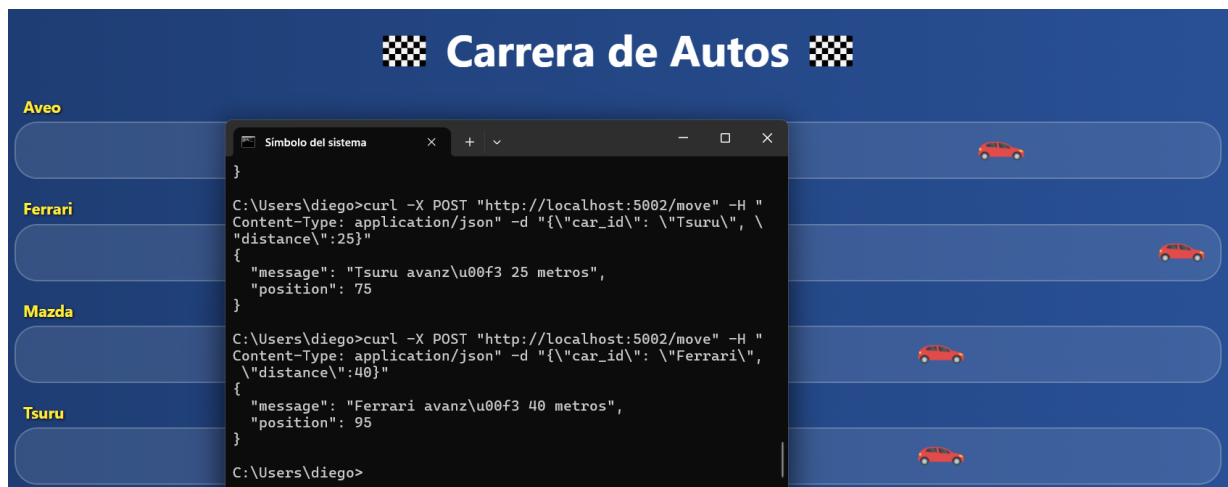


Figura 24. Tercer avance para la carrera.

Después avanzamos 15m a Mazda, 10m a Aveo, 15m a Tsuru y 5m a Ferrari, en este caso Ferrari ya llegó a los 100 m por lo que ya no es necesario avanzar más el auto y ya se encuentra en el podio como se puede ver en la figura 25 a continuación:

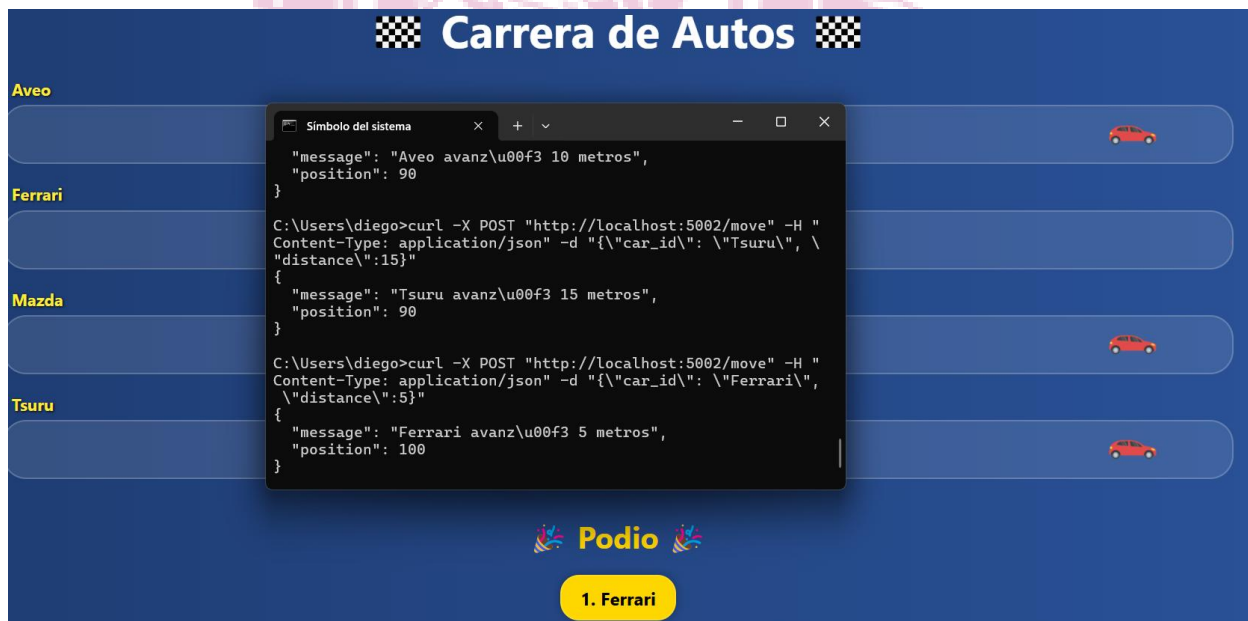


Figura 25. Cuarto avance para la carrera.

Finalmente avanzamos 10m a Mazda, 10m a Aveo y 10m a Tsuru como se puede ver en la figura 26 a continuación:



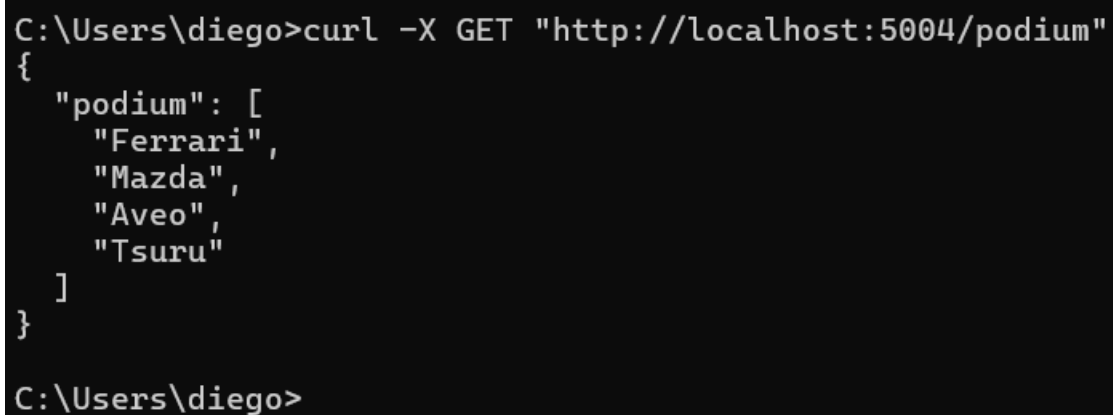
Figura 26. Último avance para la carrera.

Ahora haremos una petición GET con el comando `curl -X GET http://localhost:5003/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:5004/podium` para observar el podio y verificar que sea el mismo que muestra la interfaz web como se puede ver en las figuras 27 y 28 a continuación:



```
C:\Users\diego>curl -X GET "http://localhost:5003/race_status"
{
  "cars": {
    "Aveo": {
      "position": 100
    },
    "Ferrari": {
      "position": 100
    },
    "Mazda": {
      "position": 100
    },
    "Tsuru": {
      "position": 100
    }
  },
  "max_distance": 100,
  "podium": [
    "Ferrari",
    "Mazda",
    "Aveo",
    "Tsuru"
  ]
}
```

Figura 27. Status Final de la carrera.



```
C:\Users\diego>curl -X GET "http://localhost:5004/podium"
{
  "podium": [
    "Ferrari",
    "Mazda",
    "Aveo",
    "Tsuru"
  ]
}

C:\Users\diego>
```

Figura 28. Podio Final de la carrera en el CMD.

7. Conclusiones

La implementación de una solución basada en microservicios ha demostrado ser altamente efectiva para gestionar la comunicación en tiempo real dentro de un entorno distribuido y dinámico, como lo es la simulación de una carrera de autos. Al dividir las responsabilidades en microservicios independientes, se logró una arquitectura modular, escalable y mantenible, donde cada servicio cumple una función específica (registro, movimiento, control de carrera, estado y podio).

A través de peticiones HTTP (GET y POST), los distintos servicios pudieron interactuar de manera eficiente con los clientes, sin la necesidad de mantener conexiones persistentes ni estructuras complejas de comunicación. Esta estrategia permitió registrar autos, procesar su avance y entregar actualizaciones en tiempo real sobre el estado de la carrera, todo de forma organizada y concurrente.

La separación de funcionalidades entre servicios permitió un procesamiento más limpio y controlado de cada parte del sistema. Las peticiones GET fueron fundamentales para consultar el estado de la competencia y los resultados finales, mientras que las POST se utilizaron para realizar acciones como registrar y mover los autos. Esta división mejora tanto el rendimiento como la claridad del sistema, permitiendo escalar o modificar partes individuales sin afectar el funcionamiento global.

Los resultados obtenidos confirmaron que esta arquitectura facilita una comunicación fluida entre los microservicios y los clientes, garantizando que cada interacción ocurra sin bloqueos, inconsistencias o pérdida de datos. Además, esta solución sienta las bases para futuras mejoras, como el despliegue distribuido de servicios o la incorporación de tecnologías adicionales (como colas de mensajes o contenedores), que amplificarían aún más su eficiencia y robustez.

8. Referencias Bibliográficas

- Flask. (n.d.). Documentación de Flask. Recuperado el 1 de abril de 2025, de <https://flask.palletsprojects.com/en/stable/>
- IBM. (s.f.). ¿Qué son los microservicios? Recuperado de <https://www.ibm.com/mx-es/topics/microservices>
- Linux-Console.net. (s.f.). Creación de microservicios con Python y Flask. Recuperado de <https://es.linux-console.net/?p=26587>

