Instituto Politécnico Nacional

Escuela Superior de Cómputo

Materia: Sistemas Distribuidos

Profesor: Carreto Arellano Chadwick

Grupo: 7CM1

Práctica 5 – Objetos Distribuidos con RMI

Valdés Castillo Diego - 2021630756

Fecha de Entrega: 26/03/2025

Índice

1. Antecedente (Marco teórico)

El modelo de Objetos Distribuidos con RMI (Remote Method Invocation) es una arquitectura utilizada en sistemas distribuidos que permite que múltiples clientes interactúen con un servidor remoto a través de llamadas a métodos, como si estuvieran en la misma máquina. Este enfoque facilita la comunicación entre procesos sin la necesidad de manejar conexiones directas mediante sockets, proporcionando una abstracción más sencilla y eficiente para la distribución de tareas.

A diferencia del modelo tradicional Cliente-Servidor, donde un único servidor maneja múltiples clientes mediante conexiones directas, en RMI se utiliza un Registro RMI, que actúa como un intermediario entre los clientes y el servidor. Este registro permite que los clientes localicen referencias de objetos remotos y las utilicen sin conocer su ubicación exacta en la red.

En la implementación de sistemas distribuidos basados en RMI, se definen tres componentes esenciales:

- **Registro RMI:** Actúa como un directorio central donde los objetos remotos son registrados y desde donde los clientes pueden obtener referencias a ellos.

- **Servidor:** Expone métodos remotos que los clientes pueden invocar, permitiendo la ejecución de tareas distribuidas.
- **Clientes:** Se conectan al Registro RMI, localizan el objeto remoto y realizan llamadas a los métodos del servidor para recibir información y ejecutar acciones de forma remota.

La arquitectura basada en Objetos Distribuidos con RMI ofrece varias ventajas, entre ellas:

- **Distribución de procesos:** Permite que múltiples clientes interactúen con un único servidor de manera eficiente, delegando tareas de forma remota.
- **Transparencia en la comunicación:** RMI oculta los detalles de la red, haciendo que la invocación de métodos remotos sea similar a la ejecución de métodos locales.
- **Facilidad de escalabilidad:** Es posible ampliar el sistema sin modificar la estructura base, agregando nuevos clientes que interactúen con el servidor de manera distribuida.

En aplicaciones como simulaciones y sistemas colaborativos, RMI permite coordinar múltiples clientes con un servidor centralizado, asegurando una gestión eficiente de la comunicación y el procesamiento de eventos. Su implementación facilita la ejecución de tareas concurrentes sin la necesidad de manejar manualmente la comunicación entre los procesos, lo que lo convierte en una herramienta ideal para la distribución de cargas de trabajo en sistemas distribuidos.

2. Planteamiento del Problema

Esta práctica tiene como finalidad reforzar el conocimiento sobre el modelo de Objetos Distribuidos con RMI (Remote Method Invocation), explorando la gestión de comunicación entre múltiples clientes y un servidor mediante llamadas a métodos remotos en Java. Para ello, se implementará un sistema en el que un Registro RMI permitirá que los clientes localicen y se conecten a un Servidor de Carrera, el cual

administrará la simulación y enviará en tiempo real las actualizaciones de estado a los clientes conectados.

El principal reto radica en garantizar una comunicación eficiente y confiable entre las distintas partes del sistema. El Servidor de Carrera debe gestionar múltiples hilos para simular la competencia, permitiendo que cada auto avance de manera independiente sin generar retrasos o pérdida de información. Además, el servidor debe actualizar periódicamente el estado de la carrera y responder a las consultas de los clientes de manera sincronizada para evitar inconsistencias.

Uno de los aspectos clave en esta implementación es la gestión de concurrencia y la comunicación distribuida, asegurando que múltiples clientes puedan acceder simultáneamente a la información sin generar bloqueos en el sistema. Para lograrlo, el servidor implementa métodos remotos sincronizados, lo que permite que cada cliente obtenga información precisa y en el orden correcto.

Por último, es fundamental que la arquitectura basada en RMI funcione de manera estable, asegurando que las conexiones entre clientes y servidor sean gestionadas de manera adecuada. Además, es necesario garantizar que la recopilación y presentación de los resultados de la carrera se realicen sin errores, permitiendo que la simulación se ejecute de manera fluida y sin interrupciones.

3. Propuesta de Solución

Para abordar este problema, se propone diseñar una aplicación en Java que implemente una arquitectura basada en Objetos Distribuidos con RMI, compuesta por tres componentes principales:

- **Registro RMI:**
    - Se utiliza un Registro RMI que actúa como directorio central donde el servidor registra sus métodos remotos.
    - Los clientes acceden a este registro para localizar y conectarse al servidor de manera transparente.

    - Permite que los clientes realicen llamadas remotas sin conocer la ubicación exacta del servidor.
- **Servidor de Carrera:**
    - Se crea una clase que implementa la interfaz remota CarreraRMI, definiendo los métodos disponibles para los clientes.
    - Se registra en el Registro RMI utilizando LocateRegistry.createRegistry(1099), permitiendo que los clientes lo ubiquen y accedan a sus métodos.
    - Gestiona la carrera de autos mediante la sincronización de múltiples hilos, donde cada hilo representa un auto en competencia.
    - Cada auto avanza en intervalos de tiempo aleatorios para simular condiciones realistas de una carrera.
    - Los clientes pueden invocar métodos remotos para obtener actualizaciones en tiempo real sobre el estado de la competencia.
    - Una vez que todos los autos han alcanzado la meta, el servidor determina las posiciones finales y envía un mensaje de finalización a los clientes.
- **Clientes:**
    - Los clientes se conectan al Registro RMI y buscan el objeto remoto CarreraRMI para interactuar con el servidor.
    - Se registran en la carrera llamando al método remoto registrarAuto(String nombre).
    - Durante la competencia, consultan el estado de la carrera mediante el método remoto obtenerEstado(), recibiendo actualizaciones periódicas sobre el avance de los autos.
    - La información de la carrera se muestra en la consola en tiempo real.
    - Al finalizar la carrera, reciben una notificación con los resultados finales antes de cerrar la conexión.

Esta arquitectura basada en Java RMI mejora la eficiencia del sistema al permitir la ejecución de una simulación distribuida sin la necesidad de gestionar conexiones manuales mediante sockets. La comunicación entre clientes y servidor se mantiene

ordenada y libre de interferencias gracias al esquema centralizado del Registro RMI. Además, la sincronización de los métodos remotos garantiza que los datos se actualicen de manera consistente y sin bloqueos, proporcionando una simulación fluida y en tiempo real de la carrera de autos.

4. Materiales y Métodos empleados

- Hardware: Laptop ASUS y Teléfono iPhone 16.
- Software: Google Chrome, Word y Acrobat Reader.
- Lenguaje de programación: Java
- Entorno de desarrollo: IDE compatible con Java, como Visual Studio Code con las debidas extensiones de depuración y ejecución, además de la terminal con comandos javac y java.
- Bibliotecas utilizadas:
    - java.rmi.* para la implementación de la arquitectura de objetos distribuidos con RMI.
    - java.rmi.registry.* para la gestión del Registro RMI.
    - java.util.* para la manipulación de listas y estructuras de datos en el servidor.
- Métodos empleados:
    - Registro RMI:
        - Se inicia un Registro RMI en el puerto 1099 para gestionar la localización del servidor.
        - Permite que los clientes obtengan una referencia al servidor sin necesidad de conocer su dirección exacta en la red.
    - Servidor de Carrera:
        - Implementa la interfaz remota CarreraRMI, definiendo los métodos que los clientes pueden invocar de manera remota.
        - Se registra en el Registro RMI, permitiendo que los clientes lo localicen y accedan a sus métodos.

        - Gestiona la carrera de autos mediante hilos, donde cada auto avanza en intervalos de tiempo aleatorios para simular condiciones realistas.
        - Sincroniza el acceso a los datos para evitar inconsistencias en la actualización de posiciones.
        - Proporciona el estado de la carrera a los clientes mediante el método remoto obtenerEstado().
        - Determina las posiciones finales y envía los resultados una vez que todos los autos han alcanzado la meta.
    - Clientes:
        - Se conectan al Registro RMI para obtener la referencia al servidor de la carrera.
        - Se registran en la carrera invocando el método remoto registrarAuto(String nombre).
        - Consultan el estado de la carrera periódicamente mediante obtenerEstado(), recibiendo actualizaciones sobre el avance de los autos.
        - Muestran en consola la información recibida del servidor en tiempo real.
        - Manejan posibles excepciones para evitar fallos en caso de desconexión inesperada.
        - Finalizan la conexión una vez que la carrera ha concluido.
- Estrategia de programación: La implementación del sistema sigue un enfoque basado en el uso de Java RMI para la comunicación distribuida entre el servidor y los clientes, evitando la necesidad de gestionar conexiones manualmente mediante sockets. En este modelo, el Registro RMI actúa como punto central para la localización de servicios, simplificando la interacción entre los componentes del sistema.

  Para simular una carrera de autos dinámica y sin patrones predefinidos, se introduce aleatoriedad en los tiempos de avance de cada auto, lo que genera una competencia impredecible. La sincronización de los métodos remotos en

el servidor garantiza la consistencia en la actualización de datos, asegurando que los clientes reciban información precisa y sin bloqueos en la comunicación.

5. Desarrollo de la Solución

Para llevar a cabo la implementación de esta solución, se diseñó un sistema basado en Objetos Distribuidos con RMI en Java, donde un Registro RMI gestiona la localización del servidor y permite que múltiples clientes interactúen con él de manera remota. En este modelo, un único Servidor de Carrera es responsable de administrar la competencia y proporcionar actualizaciones en tiempo real a los clientes conectados.

El proceso comienza cuando un cliente se conecta al Registro RMI, que escucha solicitudes en el puerto 1099. Una vez establecida la conexión, el cliente obtiene una referencia al Servidor de Carrera, lo que le permite invocar métodos remotos sin necesidad de gestionar directamente las conexiones de red.

Cuando tres clientes se han registrado en la carrera, el servidor inicia automáticamente la simulación. Cada auto avanza de forma independiente dentro de un hilo, generando pausas aleatorias para simular un desarrollo impredecible. Durante la ejecución, los clientes consultan periódicamente el estado de la carrera a través del método remoto obtenerEstado(), lo que les permite visualizar el progreso de la competencia en tiempo real.

Para garantizar un flujo de ejecución ordenado, el servidor sincroniza el acceso a los datos, asegurando que las actualizaciones de posición de los autos sean consistentes. Una vez que todos los autos han alcanzado la meta, el servidor determina las posiciones finales y envía un mensaje de finalización a los clientes. Finalmente, cada cliente recibe los resultados antes de que la conexión se cierre correctamente, garantizando una comunicación eficiente 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 > Practica5 > J ClienteCarrera.java > ...

```java
//Materia: Sistemas Distribuidos
//Autor: Diego Valdes Castillo
//Fecha de  creacion: 20/03/2025
//Version: 1.0
//Practica 5 Objetos Distribuidos con RMI
//Codigo del Cliente para la carrera

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.net.InetAddress;

public class ClienteCarrera {
    public static void main(String[] args) {
        if (args.length < 1) {
            System.out.println("Uso: java ClienteCarrera <NombreAuto>");
            return;
        }

        try {
            String servidorIP = InetAddress.getLocalHost().getHostAddress();
            Registry registry = LocateRegistry.getRegistry(servidorIP, 1099);
            CarreraRMI carrera = (CarreraRMI) registry.lookup("Carrera");

            String nombreAuto = args[0]; // Ahora el usuario proporciona el nombre del auto
            carrera.registrarAuto(nombreAuto);
            System.out.println(nombreAuto + " ha entrado en la carrera.");

            while (true) {
                Thread.sleep(2000);
                String estado = carrera.obtenerEstado();
                System.out.println("Estado de la carrera:\n" + estado);

                if (estado.contains("La carrera ha terminado")) {
                    System.out.println("La carrera ha finalizado.");
                    break;
```

Figura 1. Código implementado en Java para el Cliente - 1.

```java
                }
            }

            System.out.println("Cliente cerrado correctamente.");
            System.exit(0);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Figura 2. Código implementado en Java para el Cliente - 2.

D: > Escuela8voSem > Distribuidos > Practica5 > CarreraRMI.java > ...

```java
//Materia: Sistemas Distribuidos
//Autor: Diego Valdes Castillo
//Fecha de  creacion: 20/03/2025
//Version: 1.0
//Practica 5 Objetos Distribuidos con RMI
//Codigo de la interfaz de RMI para la carrera

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface CarreraRMI extends Remote {
    void registrarAuto(String nombre) throws RemoteException;
    void iniciarCarrera() throws RemoteException;
    String obtenerEstado() throws RemoteException;
}

```

Figura 3. Código implementado en Java para la interfaz RMI.

D: > Escuela8voSem > Distribuidos > Practica5 > ServidorCarrera.java > ...

```java
//Materia: Sistemas Distribuidos
//Autor: Diego Valdes Castillo
//Fecha de  creacion: 20/03/2025 - Version: 1.0
//Practica 5 Objetos Distribuidos con RMI - Codigo del Servidor para la carrera
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class ServidorCarrera implements CarreraRMI {
    private List<String> autos = new ArrayList<>();
    private List<Integer> posiciones = new ArrayList<>();
    private List<Long> tiempos = new ArrayList<>();
    private boolean carreraIniciada = false;
    private boolean carreraTerminada = false;

    public ServidorCarrera() throws RemoteException {}

    public synchronized void registrarAuto(String nombre) throws RemoteException {
        if (!carreraIniciada) {
            autos.add(nombre);
            posiciones.add(0);
            tiempos.add(0L);
            System.out.println(nombre + " registrado en la carrera.");

            if (autos.size() == 3) {
                iniciarCarrera();
            }
        }
    }

    public synchronized void iniciarCarrera() throws RemoteException {
        if (!carreraIniciada && autos.size() == 3) {
```

Figura 4. Código implementado en Java para el Servidor - 1.

```
            carreraIniciada = true;
            System.out.println("La carrera ha comenzado!");

            for (int i = 0; i < autos.size(); i++) {
                int autoIndex = i;
                new Thread(() -> {
                    long tiempoInicio = System.currentTimeMillis();
                    while (posiciones.get(autoIndex) < 100) {
                        posiciones.set(autoIndex, posiciones.get(autoIndex) + (int) (Math.random() * 10));
                        try {
                            Thread.sleep(1000);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                        mostrarEstado();
                    }
                    tiempos.set(autoIndex, System.currentTimeMillis() - tiempoInicio);
                    System.out.println(autos.get(autoIndex) + " ha terminado la carrera!");
                    if (posiciones.stream().allMatch(p -> p >= 100)) {
                        mostrarPodio();
                        carreraTerminada = true;
                    }
                }).start();
            }
        }
    }

    public synchronized void mostrarEstado() {
        StringBuilder estado = new StringBuilder("Progreso de la carrera:\n");
        for (int i = 0; i < autos.size(); i++) {
            estado.append(autos.get(i)).append(": ").append(posiciones.get(i)).append(" metros\n");
        }
        System.out.println(estado);
    }
```

Figura 5. Código implementado en Java para el Servidor - 2.

```
    public synchronized void mostrarPodio() {
        List<String> podio = autos.stream()
                .sorted((a1, a2) -> Long.compare(tiempos.get(autos.indexOf(a1)), tiempos.get(autos.indexOf(a2))))
                .collect(Collectors.toList());

        System.out.println("=== Podio Final ===");
        for (int i = 0; i < podio.size(); i++) {
            System.out.println((i + 1) + "º lugar: " + podio.get(i));
        }
    }

    public String obtenerEstado() throws RemoteException {
        if (carreraTerminada) return "La carrera ha terminado.";
        StringBuilder estado = new StringBuilder();
        for (int i = 0; i < autos.size(); i++) {
            estado.append(autos.get(i)).append(": ").append(posiciones.get(i)).append(" metros\n");
        }
        return estado.toString();
    }

    public static void main(String[] args) {
        try {
            System.setProperty("java.rmi.server.hostname", "192.168.56.1");
            ServidorCarrera server = new ServidorCarrera();
            CarreraRMI stub = (CarreraRMI) UnicastRemoteObject.exportObject(server, 0);
            Registry registry = LocateRegistry.createRegistry(1099);
            registry.rebind("Carrera", stub);
            System.out.println("Servidor de carrera listo en 192.168.56.1:1099");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Figura 6. Código implementado en Java para el Servidor - 3.

6. Resultados

Los resultados obtenidos evidencian que la comunicación entre el Servidor de Carrera y los múltiples clientes se desarrolla de manera eficiente, permitiendo que cada cliente reciba actualizaciones en tiempo real sobre el progreso de su auto en la competencia. Las llamadas remotas realizadas mediante RMI son correctamente procesadas y devueltas a los clientes sin pérdidas de información ni bloqueos, lo que confirma una gestión adecuada de la concurrencia y una ejecución estable del sistema distribuido.

Al ejecutar el programa en distintas ocasiones, se observa que el orden de llegada de los autos varía debido a los tiempos de espera aleatorios, logrando así una simulación dinámica y realista de la carrera. Además, se comprueba que el Servidor de Carrera administra correctamente las solicitudes de los clientes, asegurando que la carrera finalice de manera sincronizada y que las conexiones remotas se mantengan activas hasta el final del proceso, garantizando una comunicación fluida y sin interrupciones dentro de la arquitectura basada en Objetos Distribuidos con RMI.

A continuación se puede ver en la figura 7 una prueba de ejecución de los códigos desde la terminal:

En la figura 7 podemos observar que los códigos se ejecutaron desde la terminal, para esto primero se compilan los códigos con el comando *"javac CarreraRMI.java"*, *"javac ServidorCarrera.java"* y *"javac ClienteCarrera.java"* después se ejecuta primero el servidor con el comando *"java ServidorCarrera"* el cual se activa en el puerto 192.168.56.1:1099 y finalmente con el comando "*java ClienteCarrera Nombre_Auto",* se ejecutan los 3 clientes con diferente nombre de auto para cada uno.

Además en esta prueba de ejecución podemos ver que el servidor conectado en el puerto 1099 espera la conexión de 3 clientes para poder iniciar la carrera, en este caso el ganador fue el Mustang como se puede ver en el podio que muestra el servidor.

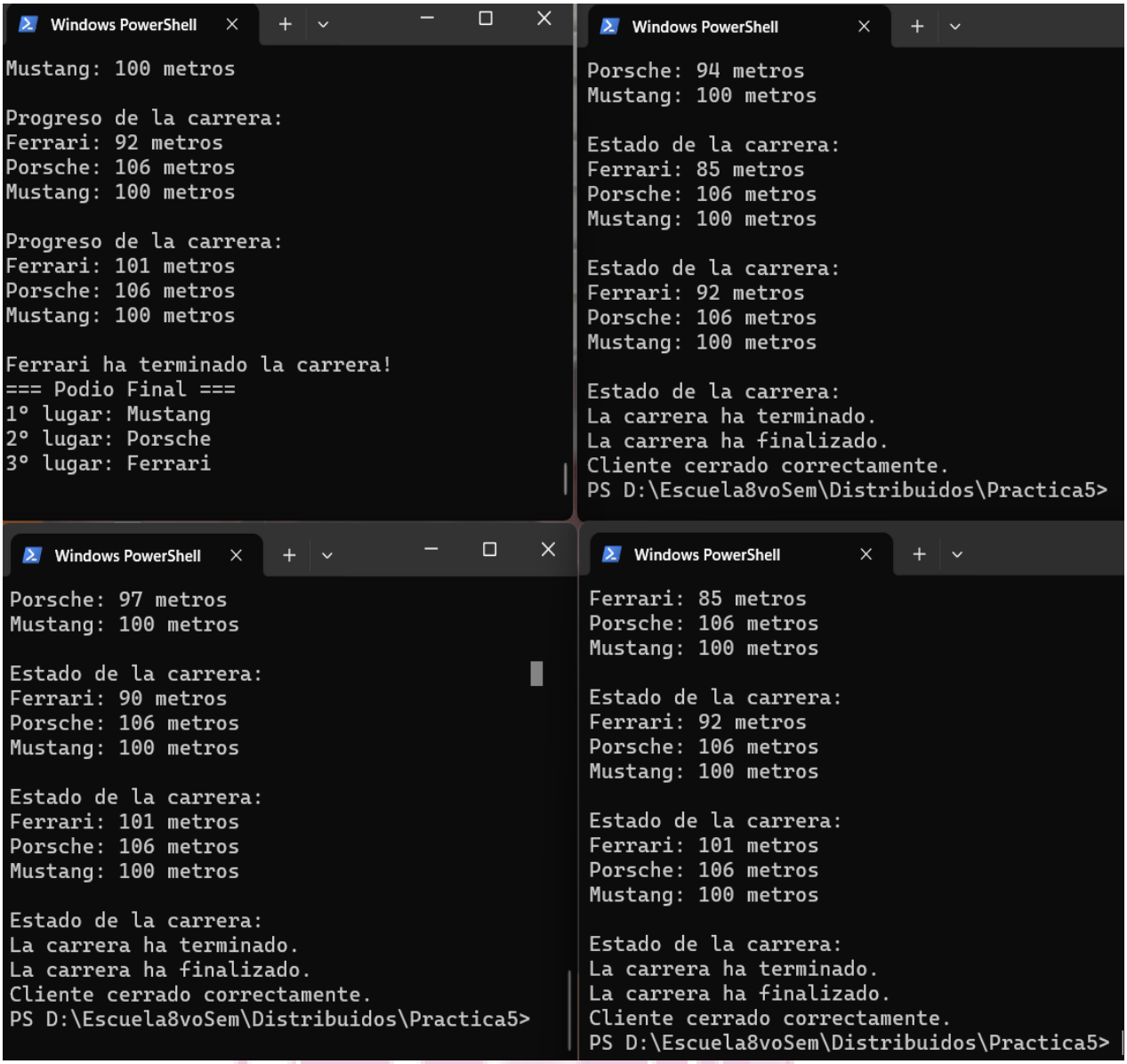

Figura 7. Prueba de Ejecución 1 – Servidor con 3 clientes.

Ahora a continuación se puede ver en la figura 8 una segunda prueba de ejecución de los códigos desde la terminal:

En la figura 8 podemos observar que los códigos se ejecutaron desde la terminal, para esto primero se compilan los códigos con el comando *"javac CarreraRMI.java"*, *"javac ServidorCarrera.java"* y *"javac ClienteCarrera.java"* después se ejecuta primero el servidor con el comando *"java ServidorCarrera"* el cual se activa en el puerto 192.168.56.1:1099 y finalmente con el comando *"java ClienteCarrera*

Nombre_Auto”, se ejecutan los 3 clientes con diferente nombre de auto para cada uno.

Además en esta prueba de ejecución podemos ver que el servidor conectado en el puerto 1099 espera la conexión de 3 clientes para poder iniciar la carrera, en este caso el ganador fue el Porsche como se puede ver en el podio que muestra el servidor.

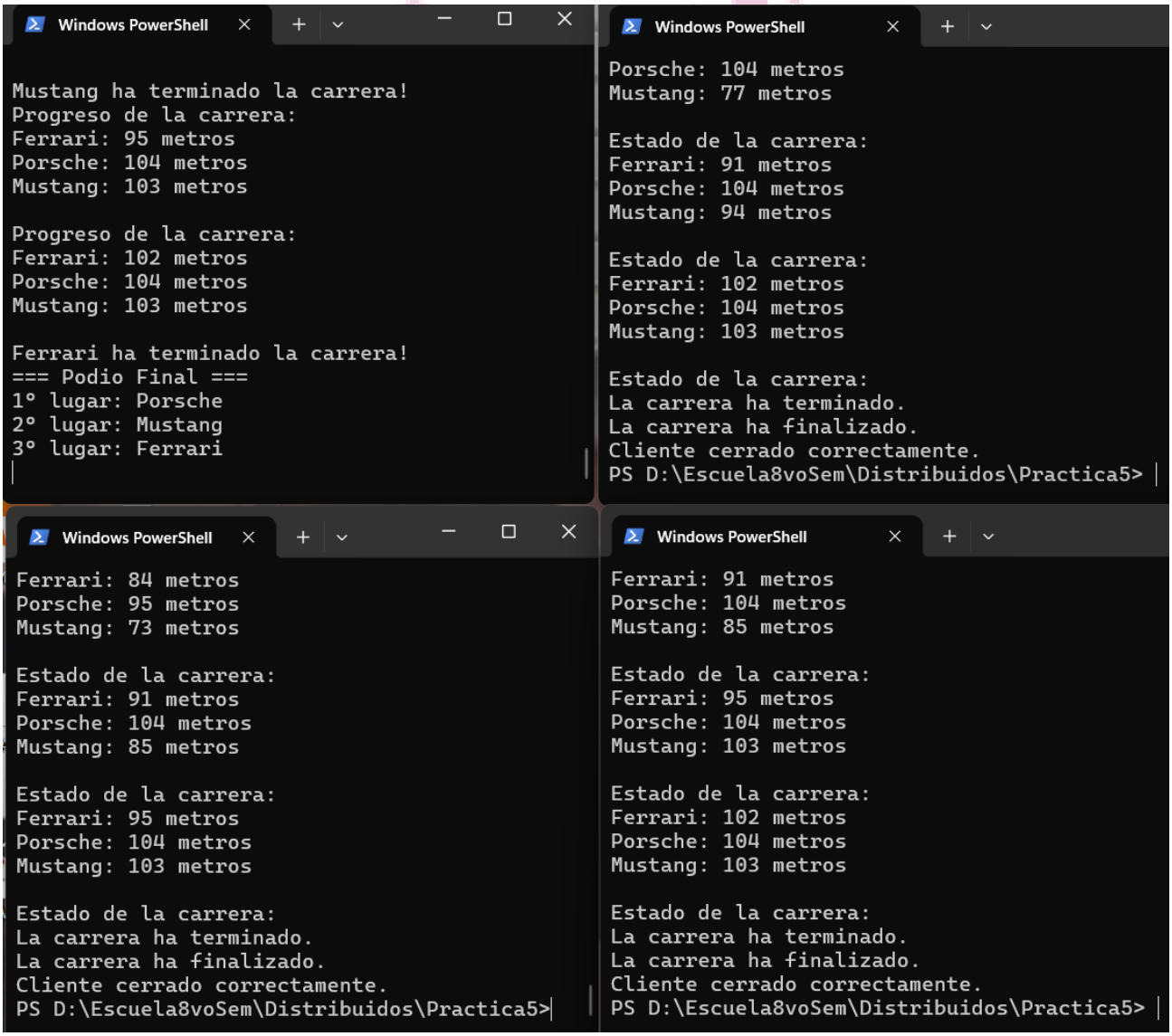

Figura 8. Prueba de Ejecución 2 – Servidor con 3 clientes.

7. Conclusiones

La implementación de Objetos Distribuidos con RMI en Java ha demostrado ser una herramienta efectiva para establecer comunicación en tiempo real dentro de un sistema distribuido. Se logró diseñar un sistema en el que múltiples clientes interactúan con un Servidor de Carrera a través de un Registro RMI, el cual facilita la localización del servidor y la invocación remota de métodos sin necesidad de gestionar conexiones directas. Gracias a esta estructura, se simplifica la comunicación entre los clientes y el servidor, asegurando un acceso eficiente a la información y evitando la sobrecarga en la gestión de conexiones.

Los resultados obtenidos reflejan que la arquitectura distribuida permite que cada cliente reciba actualizaciones inmediatas sobre el progreso de su auto en la carrera sin bloqueos ni pérdida de datos. Además, la ejecución concurrente en el Servidor de Carrera garantiza que los autos avancen de manera independiente, generando una simulación dinámica y realista. Para asegurar el correcto funcionamiento del sistema, se han implementado mecanismos de sincronización en los métodos remotos del servidor, evitando inconsistencias en la actualización de datos y asegurando que todos los clientes reciban información consistente y en el orden adecuado.

8. Referencias Bibliográficas

- Menchaca Méndez, R., & García Carballeira, F. (2001). Java RMI. Revista Digital Universitaria, 2(1). Recuperado de https://www.revista.unam.mx/vol.2/num1/art3/
- Sistemas Distribuidos. (s.f.). Objetos distribuidos. Recuperado de https://sistemasdistribuidosetitc.weebly.com/objetos-distribuidos.html
- Yahoo Video Search. (s.f.). Objetos distribuidos RMI. Recuperado de https://mx.video.search.yahoo.com/search/video?fr=mcafee&p=objetos+distribuidos+rmi&type=E211MX885G0#id=1&vid=fec2206d9ac6d1abf418838d1b094ff7&action=click