Actividad 01 Desarrollo de un servicio en red: Documentación

Integrantes: Bianca Amariutei - Kevin Ramirez

a) Mecanismo y protocolos de comunicación

Para conectar el cliente y el servidor hemos elegido Java RMI (Remote Method Invocation). La razón principal es que estamos trabajando en un entorno 100% Java. RMI nos permite invocar métodos de un objeto que está en otra máquina virtual (el servidor) exactamente igual que si lo tuviéramos en local.

Protocolo: Por debajo, RMI se encarga de todo el transporte utilizando TCP/IP, lo que nos asegura que los datos lleguen correctamente. Específicamente usa el protocolo JRMP (Java Remote Method Protocol), que es el estándar de Java para mover objetos por la red.

b) Ventajas e inconvenientes detectados

Al desarrollar la práctica, hemos visto estos puntos clave sobre RMI:

Ventajas:

- **Facilidad de uso:** Es lo más parecido a programar en local. Nos abstrae totalmente de abrir Sockets manuales o tener que crear tramas de texto para enviarnos mensajes.

- **Seguridad:** Al ser todo Java, el compilador nos avisa si estamos pasando mal los tipos de datos entre cliente y servidor. No hay errores de "parseo".

- **Serialización automática:** Podemos pasar objetos y variables complejas sin tener que convertirlas a texto manualmente; Java lo hace solo.

Inconvenientes:

- **Exclusivo de Java:** Si quisiéramos conectar este servidor con una app en Python o C#, no podríamos usar RMI.

- **Rendimiento:** Es un poco más pesado que usar Sockets puros, porque el proceso de convertir los objetos para enviarlos (serialización) consume recursos.

- **Lio con los puertos:** Aunque fijamos el puerto 5555 para el registro, los objetos remotos usan puertos aleatorios, lo que en una red real complicaría configurar el firewall.

c) Paquetes y librerías utilizadas

No hemos necesitado importar ningún .jar externo, ya que Java trae todo lo necesario en el JDK estándar. Hemos usado principalmente:

1. **java.rmi:** Para definir la interfaz Remote y controlar las excepciones de red (RemoteException, que es obligatoria en cada método).

2. **java.rmi.registry:** Para poder registrar nuestro servicio en el servidor y buscarlo desde el cliente (el "listín telefónico" de la aplicación).

3. **java.rmi.server:** Concretamente UnicastRemoteObject, que es la clase "mágica" que permite que nuestro objeto servidor viaje por la red.

d) Estructura y funcionamiento del proyecto

La arquitectura es un modelo clásico **Cliente-Servidor** unido por una Interfaz:

La Interfaz (RMIInterface): Es el nexo de unión. Aquí definimos qué acciones se pueden hacer (contar y reemplazar). Tanto el cliente como el servidor deben tener esta copia exacta para entenderse.

El Servidor (RMIServidor):

- Arranca el registro RMI en el puerto 5555.

- Implementa el código real: aquí es donde realmente se cuentan las palabras o se hace el replaceAll.

- Publica el objeto con el nombre "ManipularTexto" para que esté visible en la red.

El Cliente (RMICliente):

- Pide la IP al usuario y comprueba que tiene un formato válido (x.x.x.x).

- Se conecta al registro de esa IP y busca el servicio "ManipularTexto".

- Cuando el usuario elige una opción, el cliente llama al método. Aunque parece que se ejecuta en el ordenador del cliente, la ejecución real ocurre en el servidor, que devuelve el resultado final para imprimirlo por pantalla.

Apuntes adicionales:

En nuestro código hemos decidido implementar una validación de la IP que introduce el usuario para conectarse al servicio en red (Utilizamos el método modificado que nos dio el profesor en una ocasión anterior), en el cual para conectarse a localhost se debe introducir las IPs 127.0.0.1.

En el rebind del registro del Servidor el puerto especificado es el 0 ya que este indica a la JVM que asigne dinámicamente cualquier puerto libre disponible para la comunicación del objeto remoto, evitando conflictos con otros servicios.