ACTIVIDAD 3

Mejora 1: Protocolo de Bienvenida Proactivo

```
→  java git:(juego) javac com/socket/AppServerSocket.java
→  java git:(juego) java com.socket.AppServerSocket
ServerSocket en puerto: 7777
IP:172.21.86.61, HostName: 172.21.86.61
```

```
C:\Users\silvi\Desktop\RandomSocket\ClientSocket\src\main\java>javac com\socket\App
Cliente.java

C:\Users\silvi\Desktop\RandomSocket\ClientSocket\src\main\java>java com.socket.AppC
liente
<Cliente> Conectando a 172.21.76.59...
<Server> ¡Bienvenido al Juego del Número Secreto! Intenta adivinar del 1 al 10.
<Cliente> Introduce un numero: 5
<Server> El número buscado es MAYOR
<Cliente> Introduce un numero: 4
<Server> El número buscado es MAYOR
<Cliente> Introduce un numero: 8
<Server> El número buscado es MAYOR
<Cliente> Introduce un numero: 1
<Server> El número buscado es MAYOR
<Cliente> Introduce un numero: 9
<Server> El número buscado es MAYOR
<Cliente> Introduce un numero: 0
<Server> El número buscado es MAYOR
<Cliente> Introduce un numero: 10
<Server> Ha adivinado el número
```

1. Qué se cambió

- **En el Servidor (AppServerSocket.java)** → En el código anterior, después de `salida = new PrintWriter(...)`, el servidor entraba directamente en el bucle `while` a esperar datos. En el nuevo, se añadió `salida.println("¡Bienvenido al Juego...!")` antes del bucle.

- **En el Cliente (AppCliente.java)** → Originalmente, el cliente empezaba imprimiendo por su cuenta `<Cliente>Inserte un número:` y enviando datos sin recibir nada primero. Ahora, el código comienza con un bucle `while` que primero lee lo que el servidor tiene que decir (`entradaSocket.readLine()`).

2. Por qué se cambió

- **Sincronización Inversa** → En el código antiguo, si el cliente se conectaba pero el usuario tardaba en escribir, el servidor se quedaba bloqueado sin dar señales de vida.

Silvia Mª Moreno García
2º DAM

- **Claridad del Juego** → Al ser un juego de adivinanza, es vital que el servidor informe al usuario sobre el rango del número (del 1 al 10) nada más conectar.

- **Confirmación de Conexión** → Envía una señal inmediata de que el socket está abierto y funcionando. Si el cliente recibe el saludo, el usuario sabe con total certeza que está conectado a la IP correcta de la VM.

3. Cómo comprobar que funciona

1. **Lanzamiento** → Ejecuta el servidor en la VM Linux.

2. **Conexión** → Ejecuta el cliente en la terminal de Windows usando la IP 172.21.76.59.

3. **Resultado esperado** → En la segunda imagen se observa que lo primero que aparece es el mensaje del servidor:

   <Server> ¡Bienvenido al Juego del Número Secreto! Intenta adivinar del 1 al 10.

4. **Verificación** → Esto confirma que el cliente no tuvo que "adivinar" cuándo empezar a hablar, sino que reaccionó a la iniciativa del servidor.

Mejora 2: Persistencia de Estado (Contador de Intentos)

```
→  java git:(juego) ✗ javac com/socket/AppServerSocket.java
→ java git:(juego) ✗ java com.socket.AppServerSocket
ServerSocket en puerto: 7777
IP:172.21.86.61, HostName: 172.21.86.61
```

```
PS C:\Users\silvi\Desktop\RandomSocket\ClientSocket\src\main\java> javac com\socket\AppCliente.java
PS C:\Users\silvi\Desktop\RandomSocket\ClientSocket\src\main\java> java com.socket.AppCliente
<Cliente> Conectando a 172.21.76.59...
<Server> ¡Bienvenido al Juego del Número Secreto! Intenta adivinar del 1 al 10.
<Cliente> Introduce un numero: 3
<Server> El número es MAYOR. (Intentos: 1)
<Cliente> Introduce un numero:
<Server> Por favor, introduzca un número
<Cliente> Introduce un numero: 8
<Server> El número es MENOR. (Intentos: 3)
<Cliente> Introduce un numero: 6
<Server> El número es MENOR. (Intentos: 4)
<Cliente> Introduce un numero: 5
<Server> El número es MENOR. (Intentos: 5)
<Cliente> Introduce un numero: 4
<Server> ¡GANASTE! Lo lograste en 6 intentos.
```

1. Qué se cambió

- **Variable de Estado** → Se añadió una variable de clase estática `private static int intentos = 0;` en el servidor (AppServerSocket.java).

- **Lógica de Incremento** → Dentro del método `checkNumero`, se añadió la instrucción `intentos++;` para que se ejecute cada vez que llega un nuevo mensaje del cliente.

- **Respuesta Dinámica** → Se modificaron todos los Strings de respuesta (MENOR, MAYOR y GANASTE) para concatenar el valor actual de la variable `intentos`.

2. Por qué se cambió

- **Gamificación** → Un juego de adivinanza pierde interés si el usuario no sabe cuánto le ha costado llegar a la solución.

- **Memoria del Servidor** → Originalmente, el servidor era "olvidadizo" (stateless); trataba cada número de forma aislada. Con este cambio, el servidor mantiene un contexto de la sesión actual, lo que permite ofrecer una respuesta mucho más informativa y personalizada.

3. Cómo comprobar que funciona

1. **Interacción** → Con el servidor y cliente corriendo, introduce un número erróneo en la terminal de Windows.

2. **Primera Validación** → El servidor debe responder con el mensaje de pista seguido de (Intentos: 1).

3. **Segunda Validación** → Introduce otro número erróneo. El servidor debe responder ahora con (Intentos: 2).

4. **Confirmación Final** → Al acertar el número, el mensaje de "¡GANASTE!" debe mostrar el total exacto de intentos realizados durante toda la conexión.

Mejora 3: Cierre Automático y Gestión de Recursos

```
→  java git:(juego) ✗ javac com/socket/AppServerSocket.java
→  java git:(juego) ✗ java com.socket.AppServerSocket
ServerSocket en puerto: 7777
IP:192.168.0.17, HostName: 192.168.0.17
<Server> Partida finalizada. Cerrando socket...
→  java git:(juego) ✗
```

```
Windows PowerShell
Instale la versión más reciente de PowerShell para obtener nuevas c
aracterísticas y mejoras. https://aka.ms/PSWindows

javac com\socket\AppCliente.java\RandomSocket\ClientSocket\src\main
\java>
PS C:\Users\silvi\Desktop\RandomSocket\ClientSocket\src\main\java>
java com.socket.AppCliente
<Cliente> Conectando a 192.168.0.21...
<Server> ¡Bienvenido al Juego del Número Secreto! Intenta adivinar
del 1 al 10.
<Cliente> Introduce un numero: 0
<Server> El número es MAYOR. (Intentos: 1)
<Cliente> Introduce un numero: 3
<Server> El número es MAYOR. (Intentos: 2)
<Cliente> Introduce un numero: 8
<Server> El número es MENOR. (Intentos: 3)
<Cliente> Introduce un numero: 6
<Server> El número es MENOR. (Intentos: 4)
<Cliente> Introduce un numero: 5
<Server> ¡GANASTE! Lo lograste en 5 intentos.
PS C:\Users\silvi\Desktop\RandomSocket\ClientSocket\src\main\java>
```

1. Qué se cambió

- **Control de Bucle** → Se integró una estructura de control condicional (if) dentro del bucle de escucha del servidor y del cliente que busca la palabra clave "GANASTE".

- **Ruptura de Flujo (break)** → Al detectarse la victoria, se invoca la sentencia break, lo que detiene inmediatamente la lectura de datos y permite que el programa avance hacia las líneas finales de código.

- **Cierre de Sockets** → Se añadieron de forma explícita las instrucciones `client.close()` y `srvSock.close()` (en el servidor) y `socket.close()` (en el cliente) tras la salida del bucle.

2. Por qué se cambió

- **Graceful Shutdown (Cierre ordenado)** → Un software profesional no debe depender de que el usuario lo cierre a la fuerza (como con `Ctrl + C`). Esta mejora permite que la aplicación termine su ciclo de vida de forma natural cuando se cumple el objetivo del juego.

- **Liberación de Puertos** → Si no se cierran los sockets correctamente, el sistema operativo puede mantener el puerto `7777` en estado "TIME_WAIT". Esto causaría el error "Address already in use" si intentaras reiniciar el servidor inmediatamente.

- **Seguridad y Limpieza** → Asegura que todos los flujos de datos (streams) se vacíen y cierren, evitando fugas de memoria (memory leaks) en la VM Linux.

3. Cómo comprobar que funciona

1. **Ejecución** → Arranca el servidor en Linux y el cliente en Windows.

2. **Victoria** → Introduce el número correcto (adivina el número mágico).

3. **Verificación en Cliente** → Observa cómo la terminal de Windows muestra el mensaje "¡GANASTE!", finaliza el programa y te devuelve al cursor de comandos de forma automática.

4. **Verificación en Servidor** → En la VM Linux, verás que el servidor imprime `<Server> Partida finalizada. Cerrando socket...` y el proceso de Java termina por sí solo, dejando la terminal lista para un nuevo comando.

❓ Preguntas de comprensión del código

Responde de forma clara y razonada. No se evaluará solo que “funcione”, sino que demuestres que entiendes el código que has modificado o añadido.

1. Arquitectura y comunicación.

1. **¿Qué papel cumple el servidor y cuál el cliente en esta aplicación?**

   - **Servidor (VM)** → Actúa como la entidad pasiva que "escucha" peticiones en un puerto específico (7777). Su función es generar el número secreto, validar los intentos del usuario, gestionar el estado de la partida (contador) y decidir cuándo termina la sesión.

   - **Cliente (Host)** → Es la entidad activa que inicia la conexión. Su papel es servir de interfaz para el usuario, capturando la entrada por teclado y mostrando las respuestas procesadas que recibe desde la red.

2. **¿Qué tipo de comunicación se utiliza (bloqueante / no bloqueante)? ¿Por qué?**

   - Se utiliza comunicación bloqueante. Porque funciones como `srvSock.accept()`, `entrada.readLine()` o `entradaConsola.readLine()` detienen la ejecución del hilo principal hasta que ocurre una conexión o la llegada de datos. Pregunta -> Respuesta.

3. **¿Por qué es importante separar el servidor (VM) y el cliente (host) en este ejercicio?**

   - Para simular un **entorno de red real**. Al estar en máquinas distintas con IPs diferentes (`172.21.76.59` / `192.168.0.21` para la VM), se demuestra que la comunicación por sockets funciona a través de la pila TCP/IP, superando las barreras de sistemas operativos distintos (Linux y Windows) y no limitándose a pruebas locales en `localhost`.

2. Flujo de ejecución

1. **Describe paso a paso qué ocurre desde que el cliente envía un mensaje hasta que recibe la respuesta.**

   1. El **cliente** lee una línea desde la consola de Windows (`entradaConsola.readLine()`).

2. El cliente envía esa cadena al **servidor** a través del socket usando salida.println().

3. El **servidor** recibe la cadena en su flujo de entrada (entrada.readLine()).

4. El **servidor** invoca a checkNumero(datoRec), donde compara el valor con el número secreto e incrementa el contador de intentos.

5. El **servidor** envía el resultado (Mayor/Menor/Ganaste) de vuelta al cliente.

6. El cliente recibe la respuesta y la imprime en pantalla.

2. ¿En qué punto del código se realiza la lectura del mensaje?

- En el servidor, la lectura del mensaje del cliente ocurre dentro del bucle while ((datoRec = entrada.readLine()) != null).

3. ¿Dónde se genera y se envía la respuesta del servidor?

- La respuesta se genera en el método checkNumero() y se envía inmediatamente después mediante salida.println(respuesta).

4. ¿Qué cambios serían necesarios para que el servidor pudiera atender a varios clientes simultáneamente?

- Sería necesario implementar **Multihilo (Threads)**. El servidor debería crear un nuevo hilo de ejecución para cada socket devuelto por accept(), permitiendo que el hilo principal vuelva a escuchar nuevas conexiones mientras otros hilos gestionan las partidas individuales.

3. Cambios realizados

1. Enumera los 3 cambios mínimos que has realizado sobre la aplicación.
Para cada cambio, explica:

- **Qué hacía el código antes.**
- **Qué hace ahora.**
- **Qué problema mejora o qué funcionalidad añade.**

MEJORA 1

- **Antes** → El servidor esperaba en silencio a que el cliente enviara el primer dato.

- **Ahora** → El servidor envía instrucciones nada más establecerse la conexión.

- **Mejora** → Mejora la experiencia de usuario al confirmar que la conexión es exitosa y explicar las reglas del juego de inmediato.

MEJORA 2

- **Antes** → El servidor no guardaba memoria de cuántos números había enviado el cliente.

- **Ahora** → Una variable estática mantiene el conteo de intentos durante toda la sesión.

- **Mejora** → Añade lógica de juego y permite al usuario conocer su desempeño en tiempo real.

MEJORA 3

- **Antes** → Los programas quedaban abiertos tras ganar, requiriendo un cierre forzado (Ctrl+C).

- **Ahora** → El sistema detecta la palabra "GANASTE" y ejecuta .close() en todos los sockets.

- **Mejora** → Garantiza la eficiencia del sistema, liberando el puerto 7777 y evitando fugas de memoria.

2. ¿Qué cambio consideras más importante y por qué?

- Considero que el cambio más importante es el **Cierre Automático y Gestión de Recursos (Mejora 3)**, ya que es el factor que transforma un prototipo funcional en una aplicación de red más profesional. Mientras que las otras mejoras afectan a la experiencia de usuario, esta garantiza la estabilidad del sistema al asegurar que, una vez finalizada la partida, tanto el cliente como el servidor liberen explícitamente los flujos de datos y los sockets. Esto evita que el puerto 7777 quede bloqueado en un estado de espera por el sistema operativo, permitiendo reiniciar el servicio de inmediato sin errores de "puerto ya en uso" y asegurando que no existan fugas de memoria en la máquina virtual.