

UT3 – Programación de comunicaciones en red – EJERCICIOS 1 – 2 – 3

Programación de comunicaciones en red

EJERCICIOS 1 – 2 – 3



UT3 – Programación de comunicaciones en red – EJERCICIOS 1 – 2 – 3

EJERCICIO 1: Servidor Echo "Saludo Exprés"

Objetivo General

Diseñar un **servidor** que reciba un mensaje de un **cliente** y se lo reenvíe al mismo cliente, anteponiendo la frase:

“Hola, recibí tu mensaje: <mensaje>”.

Puntos Clave

1. Componente Servidor:

- Deberá utilizar la clase `ServerSocket` (de la librería `java.net`) para abrir un **puerto** y “ponerse a la escucha” (`listen`).
- Cuando llega un cliente, se invoca el método `accept()`, que devuelve un objeto `Socket`.
- **Flujos de entrada y salida** (`InputStream` y `OutputStream`) se obtienen directamente desde el objeto `Socket`. Esto permite la comunicación con el cliente.
- Posteriormente, se pueden manipular estos flujos con `InputStreamReader`, `BufferedReader`, `PrintWriter` o similar, para facilitar el trabajo con **cadenas de texto**.

2. Componente Cliente:

- Se conectará al servidor empleando la clase `Socket` de `java.net`, con la IP del servidor (por ejemplo, `127.0.0.1` si es local) y el puerto en que el servidor está escuchando.
- Tras establecerse la conexión (creación del `Socket`), se podrá escribir y leer a través de los flujos de E/S con los métodos `getOutputStream()` y `getInputStream()`.

3. Intercambio de Mensajes:

- El **cliente** enviará una cadena (p. ej. “Hola servidor”).
- El **servidor** recibirá esa cadena, le añadirá “Hola, recibí tu mensaje: ” y la devolverá.
- El **cliente** mostrará en pantalla la respuesta devuelta por el servidor.

4. Consideraciones:

- **Cerrar** adecuadamente los **sockets** y los **flujos**.
- Manejar excepciones de tipo `IOException`, que son comunes en operaciones de red.
- Utilizar un **bucle** en el servidor para permitir que escuche conexiones sucesivas (o continuar indefinidamente).

Pistas Teóricas

- `ServerSocket serverSocket = new ServerSocket(5000);` // Abre el puerto 5000 para escuchar.
- `Socket clientSocket = serverSocket.accept();` // Acepta una conexión y devuelve el socket del cliente.
- Para manejar cadenas de texto, se puede emplear:

UT3 – Programación de comunicaciones en red – EJERCICIOS 1 – 2 – 3

```
BufferedReader in = new BufferedReader(new  
InputStreamReader(clientSocket.getInputStream()));  
PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
```

Donde true en el PrintWriter habilita el **auto-flush**.

Esqueleto de Código (mínimo)

Servidor (SaludoExpresServidor.java)

```
public class SaludoExpresServidor {  
    public static void main(String[] args) {  
        // 1. Crear ServerSocket en un puerto (p.ej. 5000).  
        // 2. Bucle infinito que llama a accept() para recibir conexiones.  
        // 3. Abrir flujos de E/S para leer del cliente y enviar respuesta.  
        // 4. Imprimir mensaje en consola para ver qué llega.  
        // 5. Enviar la respuesta con "Hola, recibí tu mensaje: <mensaje>".  
        // 6. Cerrar recursos.  
    }  
}
```

Cliente (SaludoExpresCliente.java)

```
public class SaludoExpresCliente {  
    public static void main(String[] args) {  
        // 1. Crear Socket apuntando a localhost y al puerto del servidor (p.ej.  
5000).  
        // 2. Obtener flujos de E/S para enviar un mensaje de prueba.  
        // 3. Leer la respuesta del servidor.  
        // 4. Imprimir por consola la respuesta.  
        // 5. Cerrar recursos.  
    }  
}
```

UT3 – Programación de comunicaciones en red – EJERCICIOS 1 – 2 – 3

EJERCICIO 2: Calculadora Remota

Objetivo General

Construir un sistema cliente-servidor en el que el **servidor** actúe como una **calculadora** que resuelve operaciones sencillas (suma, resta, multiplicación, división) indicadas por el **cliente**.

Puntos Clave

1. Comunicación de Comandos:

- El cliente enviará una **cadena** con el formato:

OPERACION OPERANDO1 OPERANDO2

Ejemplos:

- SUMA 4 5
- RESTA 10 3
- MULTIPLICA 7 2
- DIVIDE 9 3

2. Servidor:

- Recibe la cadena y la **parsea** (divide) en varias partes usando, por ejemplo, `String.split(" ")`.
- Realiza la operación correspondiente:
 - SUMA -> `op1 + op2`
 - RESTA -> `op1 - op2`
 - etc.
- Devuelve el resultado como cadena.
- Debe pensar en casos especiales, por ejemplo, **división entre cero**.

3. Cliente:

- Pide la operación por consola o la tiene codificada.
- Envía el texto al servidor, lee la respuesta y la muestra.

4. Consideraciones:

- Validar que se reciben 3 partes de la cadena (operación y dos números).
- Manejar la conversión de `String` a `double` o `int`.
- Controlar posibles excepciones como `NumberFormatException`.
- Manejar la respuesta del servidor y errores (ej. "Error: división entre cero").
- Cerrar los recursos.

Pistas Teóricas

- El servidor seguirá usando un `ServerSocket`.
- Para cada conexión (vía `accept()`), se tendrá un `Socket`.
- Lectura: `BufferedReader + readLine()` para recibir el comando.
- Escritura: `PrintWriter + println()` para enviar el resultado.

UT3 – Programación de comunicaciones en red – EJERCICIOS 1 – 2 – 3

Esqueleto de Código (mínimo)

Servidor (CalculadoraServidor.java)

```
public class CalculadoraServidor {  
    public static void main(String[] args) {  
        // 1. Abrir un ServerSocket en un puerto (p.ej. 5001).  
        // 2. Aceptar conexiones en un bucle.  
        // 3. Leer el comando (p.ej. "SUMA 4 5").  
        // 4. Separar la cadena y extraer la operación y operandos.  
        // 5. Realizar la operación (suma, resta, etc.).  
        // 6. Enviar el resultado al cliente.  
        // 7. Manejar excepciones y cerrar flujos/sockets.  
    }  
}
```

Cliente (CalculadoraCliente.java)

```
public class CalculadoraCliente {  
    public static void main(String[] args) {  
        // 1. Conectarse al servidor con Socket (localhost, 5001).  
        // 2. Leer (por consola) la operación a realizar, p.ej. "SUMA 4 5".  
        // 3. Enviarla al servidor.  
        // 4. Leer la respuesta (resultado) y mostrarlo.  
        // 5. Cerrar los recursos.  
    }  
}
```

UT3 – Programación de comunicaciones en red – EJERCICIOS 1 – 2 – 3

EJERCICIO 3: Juego de “Adivina el Número”

Objetivo General

El **servidor** genera un **número aleatorio** (por ejemplo, entre 1 y 100) y el **cliente** envía sucesivos intentos para descubrirlo. El servidor responde con “Mayor”, “Menor” o “¡Acertaste!”.

Puntos Clave

1. Servidor:

- Crea un número aleatorio (por ejemplo, usando `java.util.Random`).
- Espera conexiones de un cliente.
- Cuando llega un intento (un número que manda el cliente), compara con el número secreto:
 - Si el intento es menor, responde “Mayor”.
 - Si el intento es mayor, responde “Menor”.
 - Si acierta, responde “¡Acertaste!” y podría finalizar la partida o reiniciarla.
- A continuación, el servidor podría (opcionalmente) permitir un nuevo juego o cerrar la conexión.

2. Cliente:

- Se conecta, muestra un **mensaje** de bienvenida o instrucciones al usuario (por consola).
- Solicita al usuario un número.
- Envía ese número al servidor y lee la respuesta.
- Continúa hasta que el servidor responda “¡Acertaste!” o hasta que decida dejar de jugar.

3. Bucles de Comunicación:

- Tanto servidor como cliente se mantendrán en un bucle leyendo y escribiendo (request-response) hasta que se alcance la condición de **fin del juego**.
- Cerrar adecuadamente los recursos al terminar.

4. Manejo de Errores:

- El cliente podría enviar algo que no sea un número. Se podría controlar con `Integer.parseInt(...)` rodeado de un try/catch.
- El servidor o el cliente pueden desconectarse abruptamente, generando `IOException`. Manejar esas excepciones con cuidado.

5. Puntos Didácticos:

- Uso de un **while** para controlar la persistencia de la partida.
- **Separación de lógica:** el servidor contiene la “lógica del juego” (comparaciones) y el cliente la “interfaz de usuario” (pedir datos por consola y mostrar respuestas).
- Introducir la idea de “código bloqueante”: `accept()`, `readLine()`, etc. pueden bloquear la ejecución si no hay conexiones o si no llega más texto.

UT3 – Programación de comunicaciones en red – EJERCICIOS 1 – 2 – 3

Pistas Teóricas

- `Random rand = new Random();`
`int numeroSecreto = rand.nextInt(100) + 1; // Genera entero entre 1 y 100.`
- **Lectura de cadenas:** `String entrada = in.readLine();`
- **Conversión a número:** `int intento = Integer.parseInt(entrada);`
- **Comparación y respuesta:** `out.println("Mayor");` o `out.println("¡Acertaste!");`

Esqueleto de Código (mínimo)

Servidor (AdivinaNumeroServidor.java)

```
public class AdivinaNumeroServidor {
    public static void main(String[] args) {
        // 1. Crear ServerSocket (p.ej. puerto 5002).
        // 2. Esperar a que un cliente se conecte (accept()).
        // 3. Generar un número aleatorio 1-100.
        // 4. Leer el intento del cliente en un bucle.
        // 5. Comparar el intento con el número secreto:
        //     - "Mayor" si intento < secreto
        //     - "Menor" si intento > secreto
        //     - "¡Acertaste!" si intento == secreto
        // 6. Cuando el cliente acierte o se acaben intentos (opcional), terminar la
        // comunicación.
        // 7. Cerrar flujos y socket.
    }
}
```

Cliente (AdivinaNumeroCliente.java)

```
public class AdivinaNumeroCliente {
    public static void main(String[] args) {
        // 1. Conectarse al servidor (puerto 5002, localhost).
        // 2. Leer mensaje de bienvenida o instrucciones del servidor (opcional).
        // 3. En un bucle:
        //     a) Pedir al usuario un número.
        //     b) Enviar al servidor.
        //     c) Leer la respuesta ("Mayor", "Menor", "¡Acertaste!").
        //     d) Mostrar en pantalla.
        //     e) Si es "¡Acertaste!", salir del bucle.
        // 4. Cerrar recursos.
    }
}
```

UT3 – Programación de comunicaciones en red – EJERCICIOS 1 – 2 – 3

Consejos

- **Familiarizarse con la API de Java para red** (`java.net`), así como con las clases de **entrada/salida** (`java.io`).
- Reforzar el **paradigma cliente-servidor**, la **gestión de puertos**, la **comunicación síncrona** y la **manipulación de cadenas**.
- Se recomienda **compilar y ejecutar** el servidor en una terminal (o IDE) y luego el cliente en otra, para **observar el intercambio de datos** en tiempo real.
- Es útil experimentar con **diferentes escenarios** de ejecución, enviar **mensajes equivocados** (por ejemplo, vacíos o no numéricos) o parar el servidor abruptamente, para ver **cómo reacciona** la aplicación y dónde se pueden mejorar los **mecanismos de control de errores**.