RA3. Programa mecanismos de comunicación en red empleando sockets y analizando el escenario de ejecución.

a) Se han identificado escenarios que precisan establecer comunicación en red entre varias aplicaciones.

b) Se han identificado los roles de cliente y de servidor y sus funciones asociadas.

c) Se han reconocido librerías y mecanismos del lenguaje de programación que permiten programar aplicaciones en red.

d) Se ha analizado el concepto de socket, sus tipos y características.

e) Se han utilizado sockets para programar una aplicación cliente que se comunique con un servidor.

f) Se ha desarrollado una aplicación servidor en red y verificado su funcionamiento.

g) Se han desarrollado aplicaciones que utilizan sockets para intercambiar información.

h) Se han utilizado hilos para posibilitar la comunicación simultánea de varios clientes con el servidor.

i) Se han caracterizado los modelos de comunicación más usuales en las arquitecturas de aplicaciones distribuidas.

j) Se han depurado y documentado las aplicaciones desarrolladas

UT3. Comunicación en red

1 Cliente y servidor

Cuando se hacen programas que se comuniquen lo habitual es que uno o varios actúen de cliente y uno o varios actúen de servidores.

- **Servidor**: espera peticiones, recibe datos de entrada y devuelve respuestas.
- **Cliente**: genera peticiones, las envía a un servidor y espera respuestas.

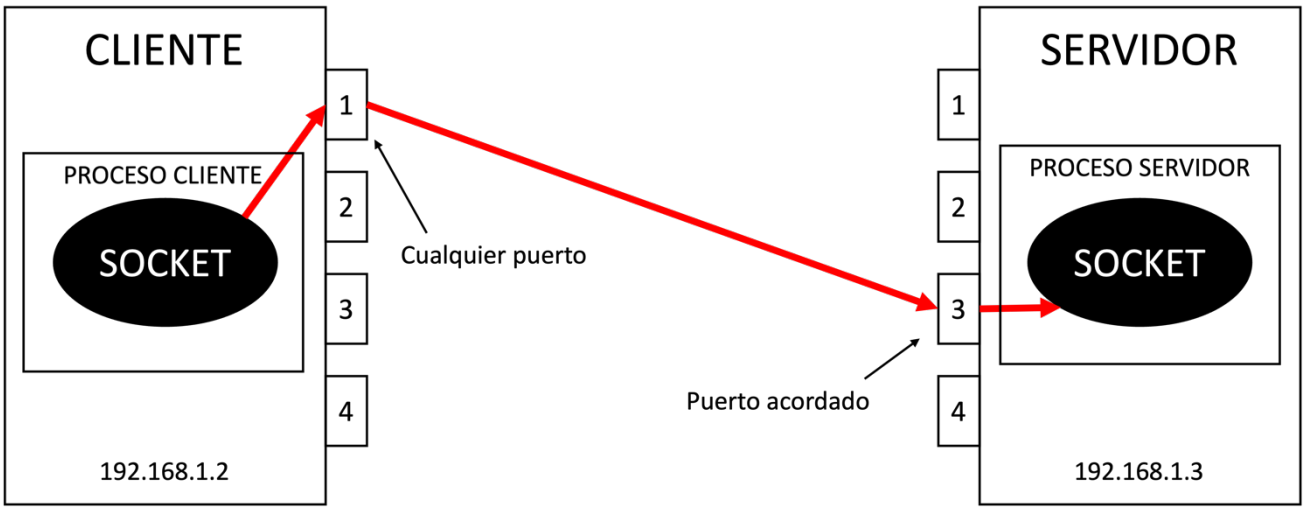
Un factor fundamental en los servidores es que tienen que ser capaces de procesar varias peticiones a la vez: deben ser multihilo por norma general.

2 Sockets

Los protocolos TCP y UDP utilizan la abstracción de sockets para proporcionar los puntos extremos de la comunicación entre aplicaciones o procesos.

Para los procesos receptores de mensajes, su conector debe tener asociado dos campos:

- La dirección **IP del host** en el que la aplicación está corriendo.
- El **puerto local** a través del cual la aplicación se comunica y que identifica el proceso.



Comunicación entre el cliente (ip 192.168.1.2, puerto 1) y el servidor (ip 192.168.1.3, puerto 3)

El programa cliente conoce el nombre de la máquina en la que se ejecuta el servidor y el número de puerto por el que escucha las peticiones. Para realizar una solicitud de conexión, el cliente realiza la petición a la máquina a través del puerto.

En Java, los sockets del servidor se declaran con `ServerSocket`:

```java
public class Servidor {

  public static void main(String[] args) throws IOException {

    int puerto = 6000; // puerto local del servidor
    ServerSocket servidor = new ServerSocket(puerto);   // inicia el server socket del servidor para atender una conexión con un cliente

    System.out.println("Escuchando en " + servidor.getLocalPort());

    Socket cliente1 = servidor.accept();    // esperando a un cliente, cuando un cliente conecta pasamos a la siguiente línea
    // ...realizar acciones con cliente1...

    Socket cliente2 = servidor.accept();    // esperando a otro cliente, cuando un cliente conecta pasamos a la siguiente línea
    // ...realizar acciones con cliente2...

    servidor.close();   // cierre socket servidor

  }
}
```

Los sockets del cliente se declaran con `Socket`:

```java
public class Cliente {

  public static void main(String[] args) throws IOException {

    String host = "localhost";  // dirección del host del servidor (localhost porque está en la máquina local)
    int puerto = 6000;    // puerto del servidor (puerto remoto)

    // Abre el socket (y la conexión con el server socket)
    Socket cliente = new Socket(host, puerto);

    InetAddress i = cliente.getInetAddress();   // para obtener la información sobre la conexión con el servidor

    System.out.println("puerto local: " + cliente.getLocalPort());
    System.out.println("puerto Remoto: " + cliente.getPort());
    System.out.println("host Remoto: " + i.getHostName().toString());
    System.out.println("IP Host Remoto: " + i.getHostAddress().toString());

    cliente.close(); // Cierra el socket

  }

}
```

3 Intercambio de datos

Una vez que un cliente y un servidor se conectan, inician el intercambio ordenado de datos. En Java, el intercambio realiza con métodos read y write de las clases `DataInputStream` y `DataOutputStream`. Hay métodos read y write para todos los tipos básicos:

Tipos	**`DataInputStream`**	**`DataOutputStream`**
String	`readUTF()`	`writeUTF()`
Boolean	`readBoolean()`	`writeBoolean()`
Short	`readShort()`	`writeShort()`
Int	`readInt()`	`writeInt()`
Long	`readLong()`	`writeLong()`
Char	`readChar()`	`writeChar()`
Double	`readDouble()`	`writeDouble()`
Float	`readFloat()`	`writeFloat()`
Byte	`readByte()`	`writeByte()`

Si queremos enviar datos más complejos como objetos necesitaremos utilizar técnicas más avanzadas.

3.1 Parsing de objetos en Java

Es muy posible que los procesos quieran intercambiarse datos complejos como objetos. Un objeto es una instancia de una clase que se almacena en la memoria reservada del proceso. Si queremos enviar un objeto a través de una red debemos **parsearlo a un string en formato JSON**.

Objeto Java	**Objeto en formato JSON**

`new Animal (`
`  "Tobby",`
`  6,`
`  10.2f,`
`  "Perro"`
`);`
	`{`
	`  "nombre": "Tobby",`
	`  "edad": 6,`
	`  "peso": 10.2,`
	`  "especie": "Perro"`
	`}`

Los objetos en formato JSON pueden enviarse a través de la red como una cadena de caracteres:

```
{"nombre":"Tobby","edad":6,"peso":10.2,"especie":"Perro"}
```

Para ello utilizaremos la librería Gson, que cuenta con los métodos:

Método	**Descripción**

`String toJson(Object objeto)`	Transforma un objeto Java a String en formato JSON.
`Object fromJson(String json, Class<Object> clase)`	Transforma un String en formato JSON a un objeto Java.

4 Servidores multihilo

Lo ideal en un servidor es que esté siempre disponible para atender peticiones de clientes. Para eso, el servidor debe gestionar cada petición con un hilo.

```java
public class Servidor {

  public static void main(String[] arg) throws IOException {
    int numeropuerto = 6000; // puerto
    // establece el socket del servidor
    ServerSocket servidor = new ServerSocket(numeropuerto);

    for (int i = 0; true; i++) {
      System.out.println("Esperando al CLIENTE.....");
      // servidor esperando a que se conecte un cliente

      // cuando un cliente se conecta...
      Socket clienteConectado = servidor.accept();

      // el servidor lanza un hilo para atender al cliente
      Servicio s = new Servicio(clienteConectado);
      Thread t = new Thread(s);
      t.setName("S-" + i);
      t.start();
    }

    // servidor.close();
  }
}
```

```java
public class Servicio implements Runnable {
  Socket clienteConectado = null;

  public Servicio(Socket clienteConectado) {
    this.clienteConectado = clienteConectado;
  }

  @Override
  public void run() {
    // Declaración de flujos de entrada y salida
}
```

5 Ejercicios

5.1 Ejercicios base

1. Crear un sistema cliente-servidor en el que:
    - El cliente envía al servidor el siguiente mensaje `“Saludos al SERVIDOR desde el CLIENTE”`.
    - El servidor responde al cliente con el siguiente mensaje `“Saludos al CLIENTE del SERVIDOR”`.
    - Termina la comunicación.

Servidor	Cliente

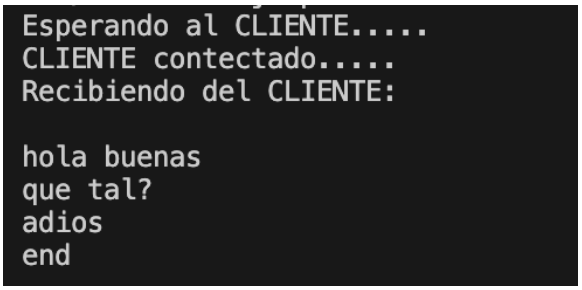
`Esperando al CLIENTE.....`
`Recibiendo del CLIENTE:`
`  Saludos al SERVIDOR desde el CLIENTE`
	`CLIENTE INICIADO....`
	`Recibiendo del SERVIDOR:`
	`  Saludos al CLIENTE del SERVIDOR`

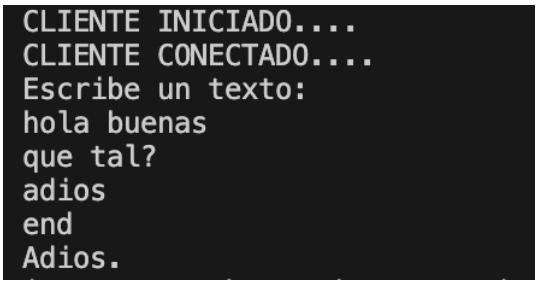
2. Crear un sistema cliente-servidor en el que:
    - El servidor envía el mensaje `“Escribe un número entero positivo:”`.
    - El cliente envía un número entero positivo.
    - El servidor calcula el sumatorio de todos los números comprendidos entre 0 y el número recibido del cliente.
    - El servidor envía el mensaje `“El sumatorio es: X”`, siendo X el resultado del cálculo.
    - El cliente imprime el mensaje recibido por el servidor.

3. Pulir el ejercicio 2 para asegurar que, mientras el cliente no envía un número entero positivo, el servidor debe seguir enviando `“Escribe un número entero positivo:”`.

4. Crear un sistema cliente-servidor en el que:
    - El servidor envía al cliente el mensaje `“Escribe un texto:”`.
    - El cliente pide al usuario que introduzca un texto de varias líneas (hasta que el usuario escriba `“end”`).
    - El cliente envía al servidor el texto de varias líneas.
    - El servidor envía al cliente el mensaje de despedida `“Adios.”`.

Servidor	Cliente

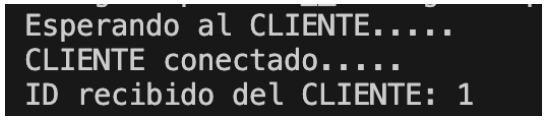
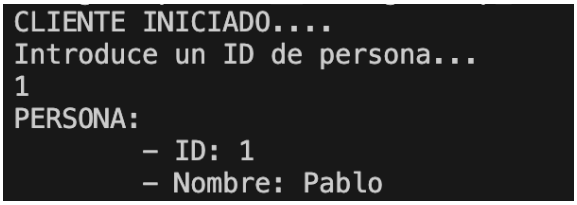
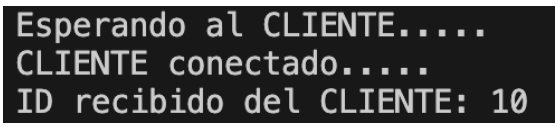
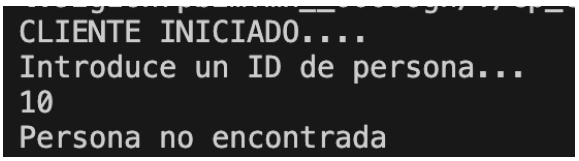




5.2 Ejercicios Intercambio de objetos

5. Crear un sistema cliente-servidor en el que el servidor tiene una lista de personas almacenadas, y el cliente quiere obtener información de una persona concreta.

   La comunicación es la siguiente:
   - El servidor envía el mensaje `"Introduce un ID de persona…"`.
   - El cliente espera a que el usuario escriba por teclado un ID.
   - El cliente envía ese ID al servidor.
   - El servidor busca en su lista de personas, la primera cuyo ID coincida con el número recibido.
     - Si la persona ha sido encontrada, el servidor envía el objeto `Persona` al cliente.
     - Si la persona no ha sido encontrada, el servidor envía el mensaje `"Persona no encontrada"`.
   - El cliente imprime el objeto `Persona` o el mensaje `"Persona no encontrada"`.

`Servidor`	`Cliente`
	
	

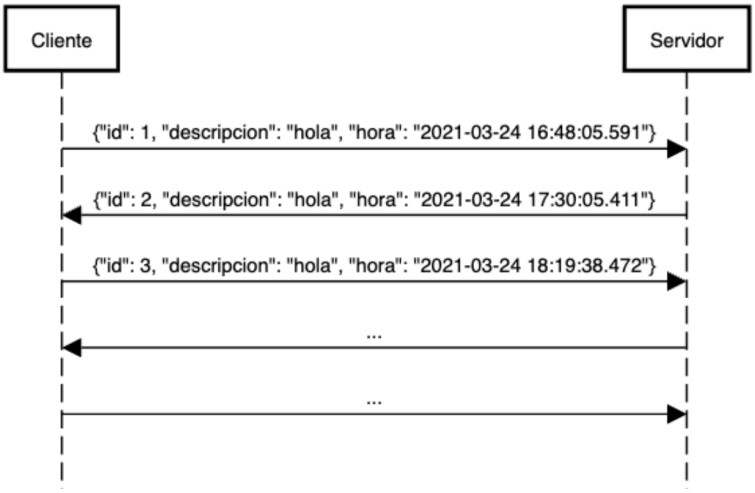
Necesitamos que el `Servidor` contenga una lista de objetos de la clase `Persona`. La clase tiene los atributos: `id`, `nombre`, `edad`, `localidad`. Para enviar un objeto a través de la red, necesitamos parsearlo a un string. Para ello, se recomienda utilizar la librería `Gson`.

Gson permite transformar objetos de Java a formato Json (string), y viceversa.

6. Crear un sistema cliente-servidor en el que se produzca un intercambio de mensajes en bucle infinito. El cliente y el servidor hablan por medio de objetos JSON con el siguiente formato:

```
{
        "id": 1,
        "descripcion": "Mensaje de ejemplo",
        "hora": "2021-03-24 16:48:05.591",
}
```

Cliente y Servidor se limitan a enviar y recibir alternativamente mensajes como se observa en el diagrama de secuencia:



Cuando el servidor recibe un mensaje, espera entre 1 y 3 segundos para responder al cliente.

Los ID se van incrementando de uno en uno. La descripción puede ser siempre la misma. La hora debe ser la fecha y hora actual al enviar el mensaje (como sugerencia usar la clase `Timestamp`).

5.3 Ejercicios Servidores multihilo

7. Crear un sistema cliente-servidor basado en el ejercicio 1. En este caso queremos que el servidor pueda recibir peticiones de muchos clientes simultáneamente. Para ello, necesitamos que `Servidor` lance un hilo de la clase `Servicio` cuando recibe la petición de un cliente.

8. Crear un sistema cliente-servidor basado en el ejercicio 3 (sumador de enteros positivos). En este caso queremos que el servidor pueda recibir peticiones de muchos clientes simultáneamente. Para ello, necesitamos que `Servidor` lance un hilo de la clase `Servicio` cuando recibe la petición de un cliente.

```
CLIENTE INICIADO....
Escribe un número entero positivo:
45
El resultado es: 1035
```

Cliente 1

```
CLIENTE INICIADO....
Escribe un número entero positivo:
hola
Escribe un número entero positivo:
5
El resultado es: 15
```

Cliente 2

```
S-0     CLIENTE conectado.....
S-1     CLIENTE conectado.....
S-0     Recibido: 45
S-1     Recibido: hola
S-1     Recibido: 5
```

Servidor

9. Crear un sistema cliente-servidor basado en el ejercicio 4 (escribir texto de varias líneas). En este caso queremos que el servidor pueda recibir peticiones de muchos clientes simultáneamente. Para ello, necesitamos que `Servidor` lance un hilo de la clase `Servicio` cuando recibe la petición de un cliente.

```
CLIENTE INICIADO....
CLIENTE CONECTADO....
Escribe un texto:
hola soy un cliente
un saludo!
end
Adios.
```

Cliente 1

```
CLIENTE INICIADO....
CLIENTE CONECTADO....
Escribe un texto:
hola soy otro cliente
me despido!
end
Adios.
```

Cliente 2

```
S-0     CLIENTE conectado.....
S-1     CLIENTE conectado.....
S-0     hola soy un cliente
S-1     hola soy otro cliente
S-0     un saludo!
S-0     end
S-1     me despido!
S-1     end
```

Servidor

10. Crear un sistema cliente-servidor basado en el ejercicio 5 (lista de personas). En este caso queremos que el servidor pueda recibir peticiones de muchos clientes simultáneamente. Para ello, necesitamos que `Servidor` lance un hilo de la clase `Servicio` cuando recibe la petición de un cliente.