Unidad 6 Comunicaciones en Java

Indice

1. Conceptos básicos

1.1 Comunicación entre aplicaciones.

Roles cliente y servidor.

Cuando se hacen programas Java 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.

Su arquitectura típica es la siguiente:

```
while (true){
        peticion=esperarPeticion();
        hiloAsociado=new Hilo();
        hiloAsociado.atender(peticion);
}
```

1.2 Recordatorio de los flujos en Java

InputStreams y OutputStreams

Manejan bytes a secas. Por ejemplo, si queremos leer un fichero byte a byte usaremos FileInputStream y si queremos escribir usaremos FileOutputStream.

Son operaciones a muy bajo nivel.

Readers y Writers

En lugar de manejar bytes manejan caracteres (recordemos que hoy en día y con Unicode una letra como la ñ en realidad podría ocupar más de un byte).

Así, cuando queramos leer letras de un archivo usaremos clases como FileReader y FileWriter.

Las clases Readers y Writers en realidad se apoyan sobre las InputStreams y OutputStreams.

A veces nos interesará mezclar conceptos y por ejemplo poder tener una clase que use caracteres cuando a lo mejor Java nos ha dado una clase que usa bytes. Así, por ejemplo InputStreamReader puede coger un objeto que lea bytes y nos devolverá caracteres. De la misma forma OutputStreamWriter coge letras y devuelve los bytes que la componen.

BufferedReaders y PrintWriters

Cuando trabajamos con caracteres (que recordemos pueden tener varios bytes) normalmente no trabajamos de uno en uno. Es más frecuente usar líneas que se leen y escriben de una sola vez. Así por ejemplo, la clase PrintWriter tiene un método print(ln) que puede imprimir elementos complejos como floats o cadenas largas.

Además, Java ofrece clases que gestionan automáticamente los buffers por nosotros lo que nos da más comodidad y eficiencia. Por ello es muy habitual hacer cosas como esta:

```
lectorEficiente = new
        BufferedReader(new FileReader("fich1.txt"));
escritorEficiente = new
        BufferedWriter(new FileWriter("fich2.txt"));
```

En el primer caso creamos un objeto FileReader que es capaz de leer caracteres de fich1.txt. Como esto nos parece poco práctico creamos otro objeto a partir del primero de tipo BufferedReader que nos permitirá leer bloques enteros de texto.

De hecho, si se comprueba la ayuda de la clase FileReader se verá que solo hay un método read que devuelve un int, es decir el siguiente carácter disponible, lo que hace que el método sea muy incómodo. Sin embargo BufferedReader nos resuelve esta incomodidad permitiéndonos trabajar con líneas.

1.3 Elementos de programación de aplicaciones en red. Librerías.

En Java toda la infraestructura de clases para trabajar con redes está en el paquete java.net.

En muchos casos nuestros programas empezarán con la sentencia import java.net.* pero muchos entornos (como Eclipse) son capaces de importar automáticamente las clases necesarias.

La clase URL¶

La clase URL permite gestionar accesos a URLs del tipo

http://marca.com/fichero.html y descargar cosas con bastante sencillez.

Al crear un objeto URL se debe capturar la excepción MalformedURLException que sucede cuando hay algún error en la URL, como por ejemplo escribir htp://marca.com en lugar de http://marca.com (obsérvese que el primero tiene un sola t en http en lugar de dos).

La clase URL nos ofrece un método openStream que nos devuelve un flujo básico de bytes. Podemos crear objetos más sofisticados para leer bloques como muestra el programa siguiente:

```
public class GestorDescargas {
	public void descargarArchivo(
			String url_descargar,
			String nombreArchivo){
		System.out.println("Descargando "
				+url_descargar);
		try {
			URL laUrl=new URL(url_descargar);
			InputStream is=laUrl.openStream();
			InputStreamReader reader=
					new InputStreamReader(is);
			BufferedReader bReader=
					new BufferedReader(reader);
			FileWriter escritorFichero=
				new FileWriter(nombreArchivo);
			String linea;
			while ((linea=bReader.readLine())!=null){
				escritorFichero.write(linea);
			}
			escritorFichero.close();
			bReader.close();
			reader.close();
			is.close();
		} catch (MalformedURLException e) {
			System.out.println("URL mal escrita!");
			return ;
		} catch (IOException e) {
			System.out.println(
				"Fallo en la lectura del fichero");
			return ;
		}
```

```
	}
	public static void main (String[] argumentos){
		GestorDescargas gd=new GestorDescargas();
		String base=
			"http://10.13.0.20:8000"+
					"/ServiciosProcesos/textos/";
		for (int i=1; i<=5; i++){
			String url=base+"tema"+i+".rst";
			gd.descargarArchivo(url);
		}
	}
}
```

1.4 Repaso de redes

En redes el protocolo IP es el responsable de dos cuestiones fundamentales:

- Establecer un sistema de direcciones universal (direcciones IP)
- Establecer los mecanismos de enrutado.

Como programadores el segundo no nos interesa, pero el primero será absolutamente fundamental para contactar con programas que estén en una ubicación remota.

Una ubicación remota siempre tendrá una dirección IP pero solo a veces tendrá un nombre DNS. Para nosotros no habrá diferencia ya que si es necesario el sistema operativo traducirá de nombre DNS a IP.

Otro elemento necesario en la comunicación en redes es el uso de un puerto de un cierto protocolo:

- TCP: ofrece fiabilidad a los programas.
- UDP: ofrece velocidad sacrificando la fiabilidad.

A partir de ahora cuando usemos un número de puerto habrá que comprobar si ese número ya está usado.

Por ejemplo, es mala idea que nuestros servidores usen el puerto 80 TCP para aceptar peticiones, probablemente ya esté en uso. Antes de usar un puerto en una aplicación comercial deberíamos consultar la lista de «IANA assigned ports».

En líneas generales se pueden usar los puertos desde 1024 TCP a 49151 TCP, pero deberíamos comprobar que el número que elegimos no sea un número usado por un puerto de alguna aplicación que haya en la empresa.

En las prácticas de clase usaremos el 9876 TCP. Si se desea conectar desde el instituto con algún programa ejecutado en casa se deberá «abrir el puerto 9876 TCP». Abrir un puerto consiste en configurar el router para que SÍ ACEPTE TRÁFICO INICIADO DESDE EL EXTERIOR cosa que no hace nunca por motivos de protección.

1.5 Funciones y objetos de las librerías.

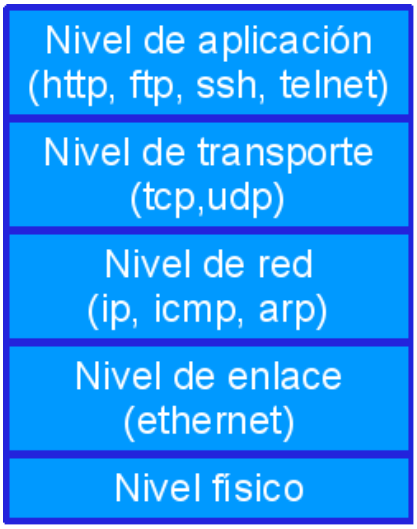
La clase URL proporciona un mecanismo muy sencillo pero por desgracia completamente atado al protocolo de las URL.

Java ofrece otros objetos que permiten tener un mayor control sobre lo que se envía o recibe a través de la red. Por desgracia esto implica que en muchos casos tendremos solo flujos de bajo nivel (streams).

En concreto Java ofrece dos elementos fundamentales para crear programas que usen redes

- Sockets
- ServerSockets

La pila de protocolos TCP/IP permite la transmisión de datos entre redes de computadores. Dicha pila consta de una serie de capas tal y como se puede apreciar en el diagrama siguiente:



Normalmente, cuando se escriben aplicaciones Java en red trabajaremos con el nivel de aplicación, y utilizaremos además protocolos del nivel de transporte. Por este motivo es preciso recordar las principales diferencias entre los dos protocolos básicos del nivel de transporte: TCP (Transmission Control Protocol) y UDP (User Datagram Protocol).

2. Diferencias entre protocolos de transporte

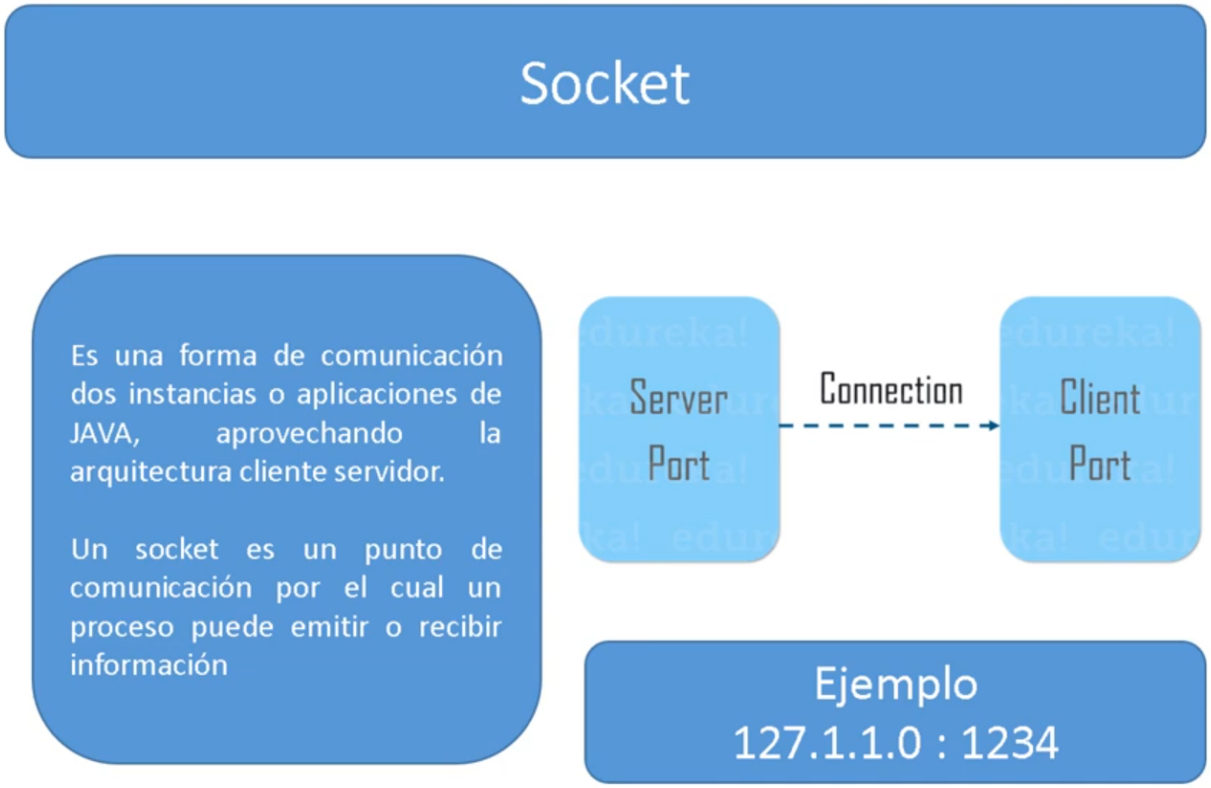
2.1. TCP

- Es un protocolo orientado a conexión
- Provee un flujo de bytes fiable entre dos ordenadores (llegada en orden, correcta, sin pérdidas → control de flujo, control de congestión...)
- Protocolos de nivel de aplicación que usan TCP: telnet, HTTP, FTP, SMTP

2.2. UDP

- Es un protocolo no orientado a conexión
- Envía paquetes de datos (datagramas) independientes sin garantías
- Permite broadcast y multicast
- Protocolos de nivel de aplicación que usan UDP: DNS, TFTP

3. Sockets



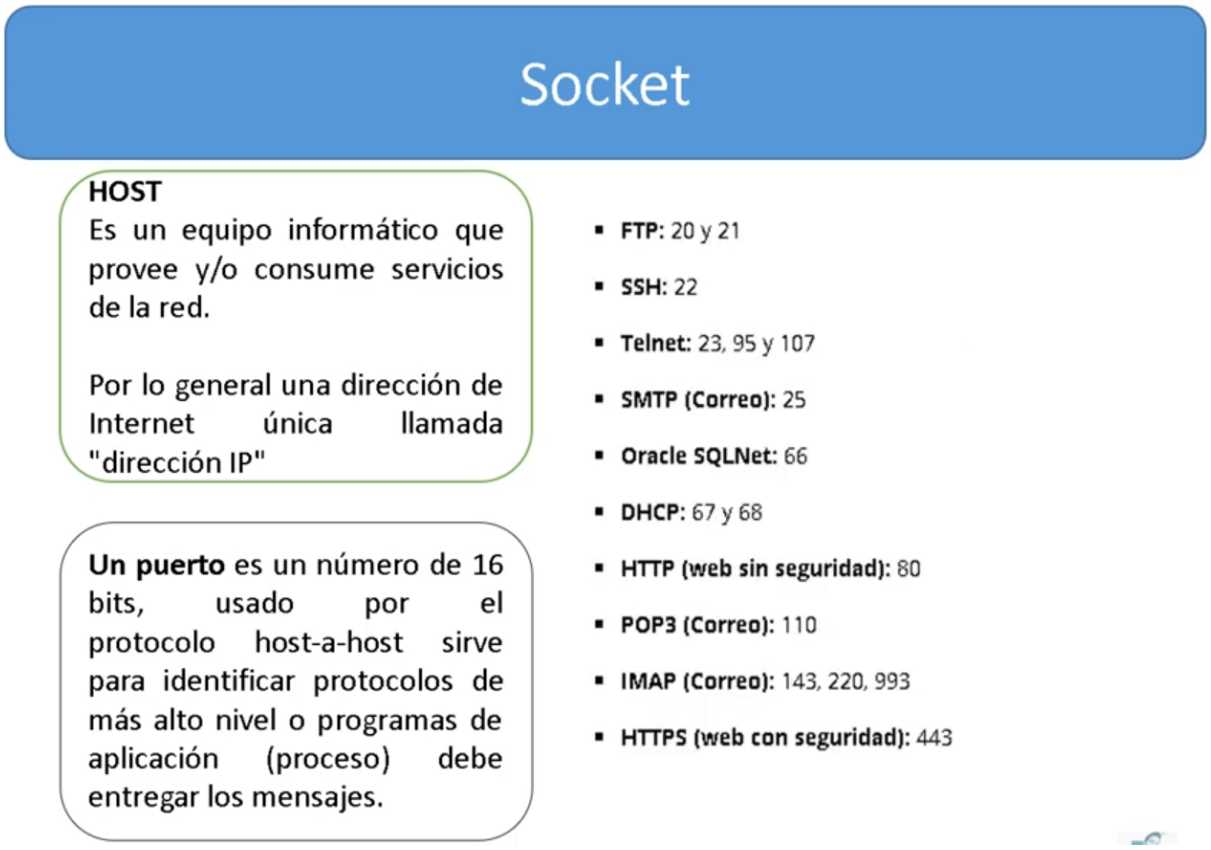
Los sockets son un mecanismo que nos permite establecer un enlace entre dos programas que se ejecutan independientes el uno del otro (generalmente un programa cliente y un programa servidor).

Cuando estamos trabajando en una red de ordenadores y queremos establecer una comunicación (recibir o enviar datos) entre dos procesos que se están ejecutando en dos máquinas diferentes de dicha red, ¿qué necesitamos para que dichos procesos se puedan comunicar entre sí?

Supongamos que una de las aplicaciones solicita un servicio (cliente), y la otra lo ofrece (servidor).

Una misma máquina puede tener una o varias conexiones físicas a la red y múltiples servidores pueden estar escuchando en dicha máquina. Si a través de una de dichas conexiones físicas se recibe una petición por parte de un cliente ¿cómo se identifica qué proceso debe atender dicha petición? Es aquí donde surge el concepto de puerto, que

permite tanto a TCP como a UDP dirigir los datos a la aplicación correcta de entre todas las que se están ejecutando en la máquina. Todo servidor, por tanto, ha de estar registrado en un puerto para recibir los datos que a él se dirigen.



Los datos transmitidos a través de la red tendrán, por tanto, información para identificar la máquina mediante su dirección IP (si IPv4 32 bits, y si IPv6 128 bits) y el puerto (16 bits) a los que van dirigidos.

Los puertos:

• son independientes para TCP y UDP

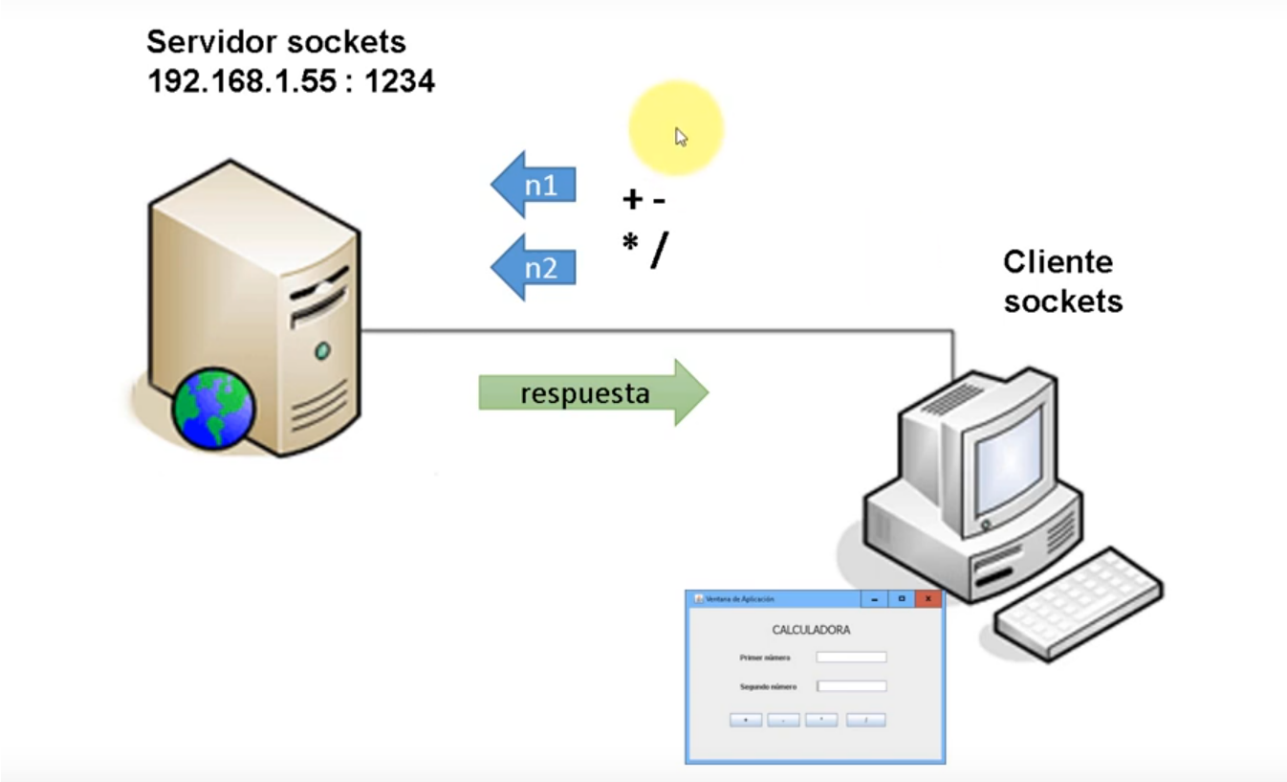
• se identifican por un número de 16 bits (de 0 a 65535)

• algunos de ellos están reservados (de 0 a 1023), puesto que se emplean para servicios conocidos como HTTP, FTP, etc. y no deberían ser utilizados por aplicaciones de usuario.

Por tanto, un socket se puede definir como un extremo de un enlace de comunicación bidireccional entre dos programas que se comunican por la red (se asocia a un número de puerto).

• Se identifica por una dirección IP de la máquina y un número de puerto.

- Existe tanto en TCP como un UDP.



3.1 Sockets TCP

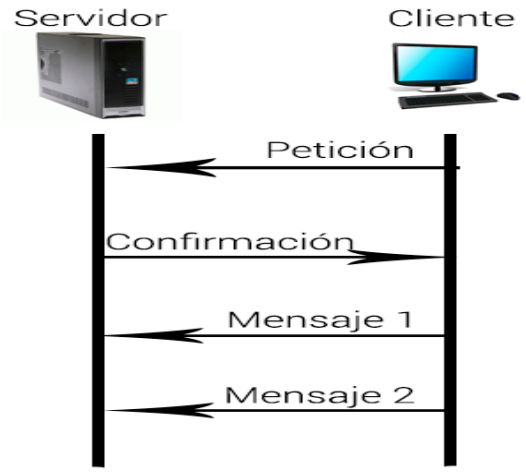
Los sockets TCP son orientados a conexión y fiables. Esto implica que antes de poder enviar y recibir datos es necesario establecer una conexión entre el cliente y el servidor.

- mantiene un canal de comunicación con el otro extremo de forma constante durante la conexión
- La entrega de paquetes es ordenada

Una vez que la conexión está establecida, el protocolo TCP garantiza que los datos enviados son recibidos correctamente y debidamente ordenados en el otro extremo.

Java por medio de la librería *java.net* nos provee dos clases: *Socket* para implementar la conexión desde el lado del cliente y *ServerSocket* que nos permitirá manipular la conexión desde el lado del servidor.

Tanto el cliente como el servidor no necesariamente deben estar implementados en Java, solo deben conocer sus direcciones IP y el puerto por el cual se comunicarán(el rango de puertos es de 1024 hasta 65535).



El servidor estará a la espera de una conexión, en cuanto el cliente inicie enviará un mensaje de petición al servidor, éste le responderá afirmativamente y una vez recibida la confirmación, el cliente enviará un par de mensajes y la conexión finalizará.

- Los datos se transmiten en Internet usando paquetes.
- Los datos se maneja en capas,por ejemplo:

Los sockets son una abstracción para facilitar la programación de aplicaciones con conectividad.

- Los sockets permiten:
  - A los clientes y servidores:
    - Conectar con la máquina remota (prot. orientados a conexión)
    - Enviar datos
    - Recibir datos
    - Cerrar una conexión(prot. orientados a a conexión)
  - A los servidores:
    - Asociar un socket a un puerto
    - Escuchar para recibir datos
    - Aceptar conexiones en el puerto asociado

Java separa en dos clases la funcionalidad de sockets orientados a conexión.
- La clase Socket usada por clientes y servidores.
- La clase ServerSocket usada únicamente por servidores.

3.1.1 Identificadores de socket

Todo socket se identifica por tres parámetros básicos:

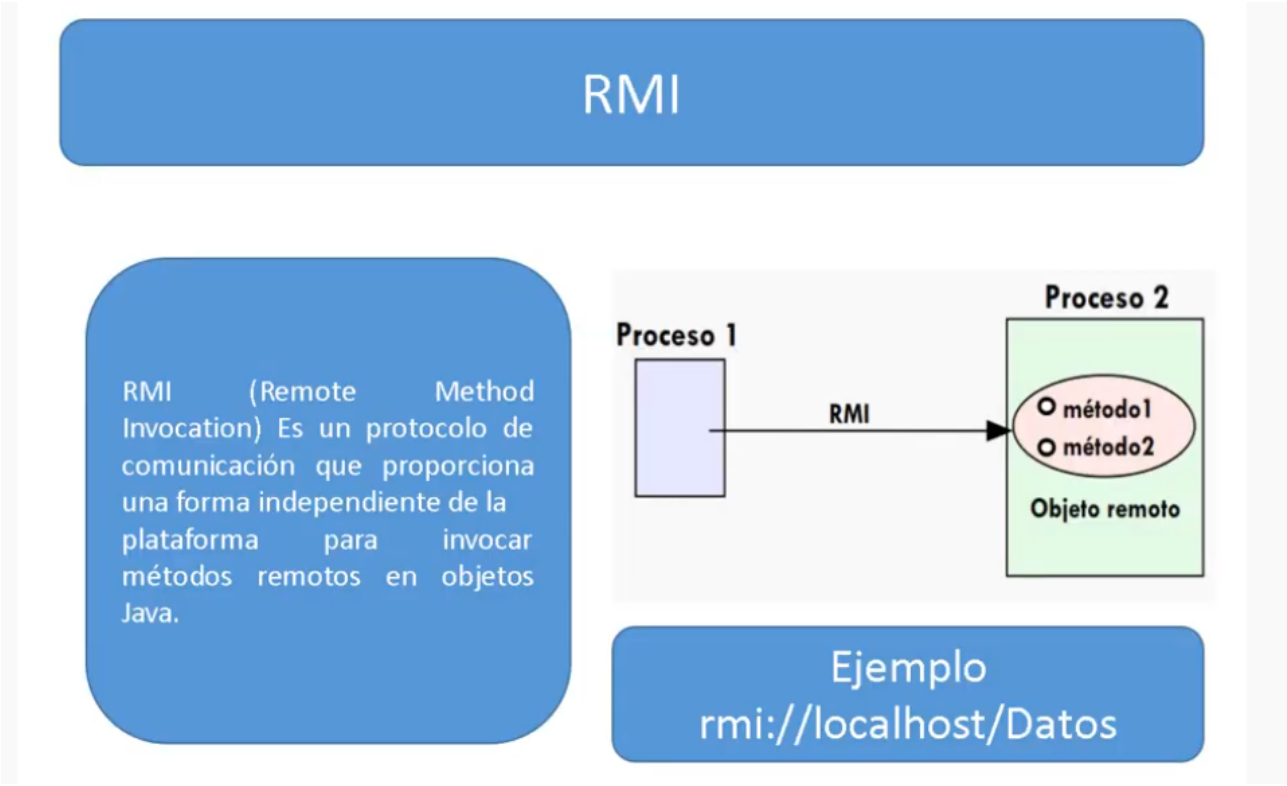
- HOST: nombre o dirección de la máquina.
    - Ej: www.uc3m.es
    - Ej. 163.117.141.114
- PUERTO: identifica a un proceso dentro de la máquina, es decir, una máquina con una dirección puede tener varios procesos intercambiando tráfico. Con el puerto distinguimos unos de otros.
- PROTOCOLO: mecanismo de intercambio de datos que se usará.

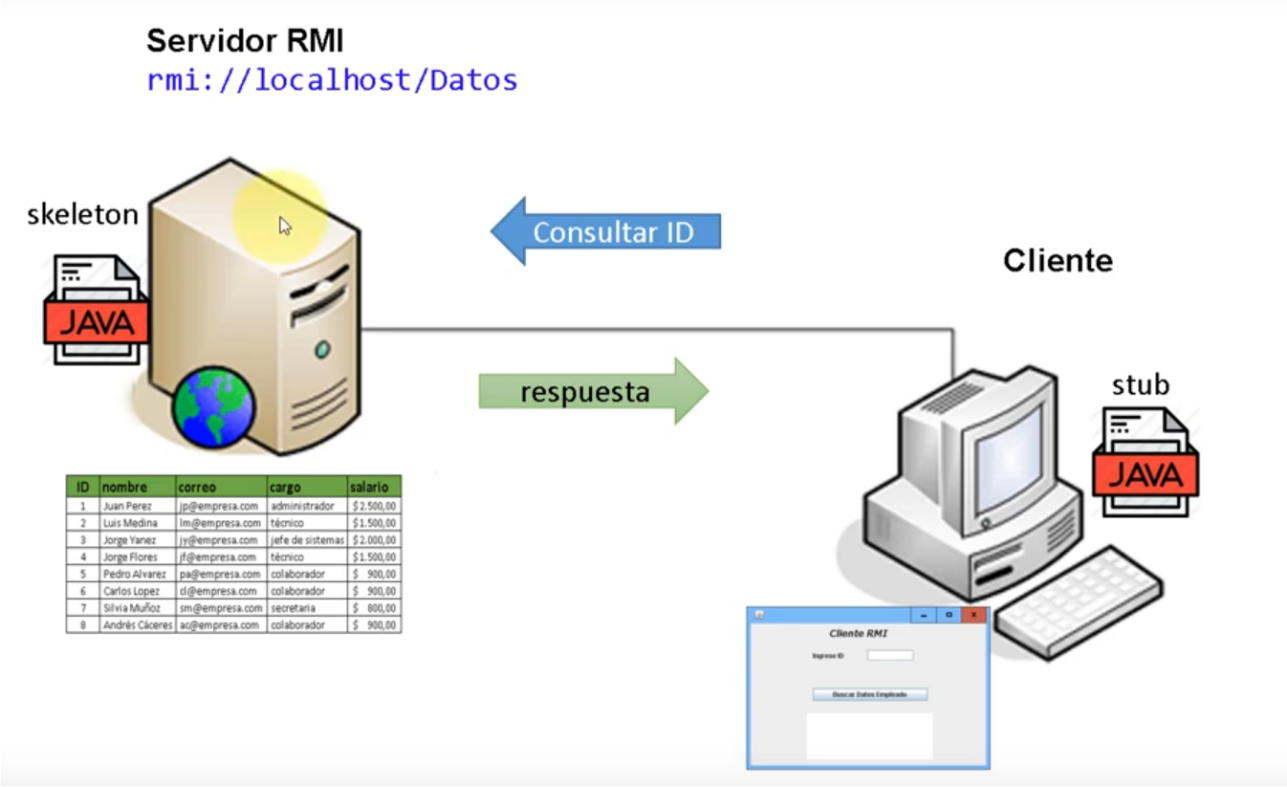
3.2 Sockets UDP

Los sockets UDP son no orientados a conexión. Los clientes no se conectarán con el servidor sino que cada comunicación será independiente, sin poderse garantizar la recepción de los paquetes ni el orden de los mismos. Es en este momento en donde podemos definir el concepto de datagrama, que no es más que un mensaje independiente, enviado a través de una red cuya llegada, tiempo de llegada y contenido no están garantizados.

- No hay canal de comunicación constante
- La entrega es desordenada
- Hay varias formas de comunicarse
    - De uno a uno: Cliente-Servidor, Peer to Peer (P2P)
    - De uno a todos: broadcast
    - De uno a varios: multicast

4. RMI


RMI
RMI (Remote Method Invocation) Es un protocolo de comunicación que proporciona una forma independiente de la plataforma para invocar métodos remotos en objetos Java.
Proceso 1
RMI
Proceso 2
método1
método2
Objeto remoto
Ejemplo
rmi://localhost/Datos


Servidor RMI
rmi://localhost/Datos
skeleton
JAVA
Consultar ID
respuesta
Cliente
stub
JAVA
Cliente RMI
ID nombre correo cargo salario
1 Juan Perez jp@empresa.com administrador $ 2.500,00
2 Luis Medina lm@empresa.com técnico $ 1.500,00
3 Jorge Yanez jy@empresa.com jefe de sistemas $ 2.000,00
4 Jorge Flores jf@empresa.com técnico $ 1.500,00
5 Pedro Alvarez pa@empresa.com colaborador $ 900,00
6 Carlos Lopez cl@empresa.com colaborador $ 900,00
7 Silvia Muñoz sm@empresa.com secretaria $ 800,00
8 Andrés Cáceres ac@empresa.com colaborador $ 900,00