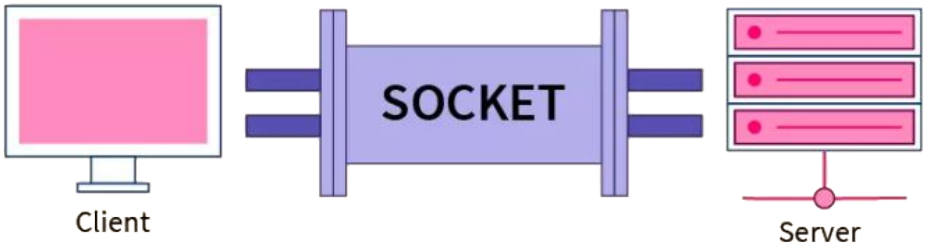
Programación de Procesos y Servicios

UT3.2: Sockets TCP
2º DAM

Sockets

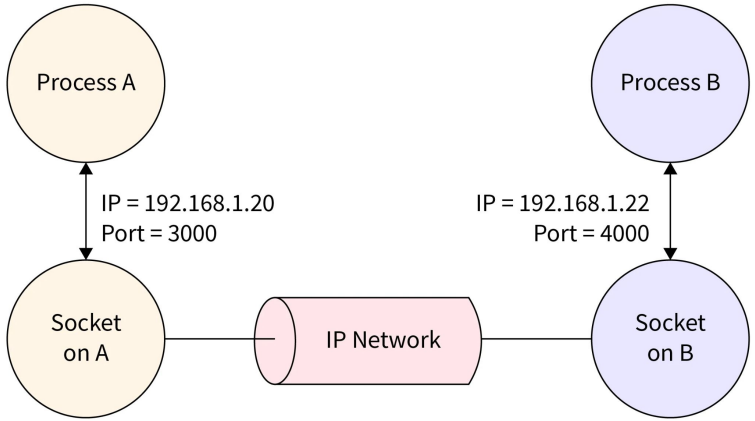
Un **socket** es la puerta de entrada a la transmisión y recepción de datos entre dispositivos, ya sea en una máquina local o a través de internet. Es una interfaz de programación de aplicaciones (API) que proporciona un mecanismo para que los procesos de software se comuniquen entre sí a través de una red



Sockets

Un **socket** representa el extremo de un canal de comunicación establecido entre un emisor y un receptor.

- Para establecer una comunicación entre dos aplicaciones, ambas deben crear sus respectivos sockets, y conectarlos entre sí.
- Una vez conectados, entre ambos sockets se crea una "tubería privada" a través de la red, que permite que las aplicaciones en los extremos envíen y reciban mensajes por ella.



Sockets

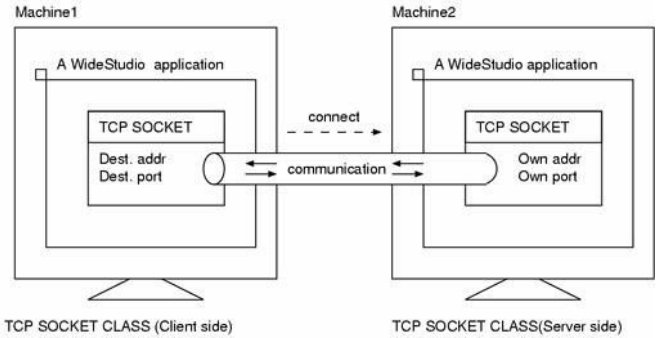
Un **socket** representa el extremo de un canal de comunicación establecido entre un emisor y un receptor.

- Para establecer una comunicación entre dos aplicaciones, ambas deben crear sus respectivos sockets, y conectarlos entre sí.
- Una vez conectados, entre ambos sockets se crea una "tubería privada" a través de la red, que permite que las aplicaciones en los extremos envíen y reciban mensajes por ella.
- Existen dos tipos de Sockets: Sockets Stream y Sockets Datagram

Sockets Stream

Los **sockets stream** son orientados a conexión y, cuando se utilizan sobre la pila IP, hacen uso del protocolo de transporte TCP.

- Son fiables
- Aseguran el orden de entrega correcto.
- Mantienen el canal de comunicación abierto entre ambas partes hasta que se termina la conexión.
- Uno ejerce el papel de proceso servidor y otro el de proceso cliente



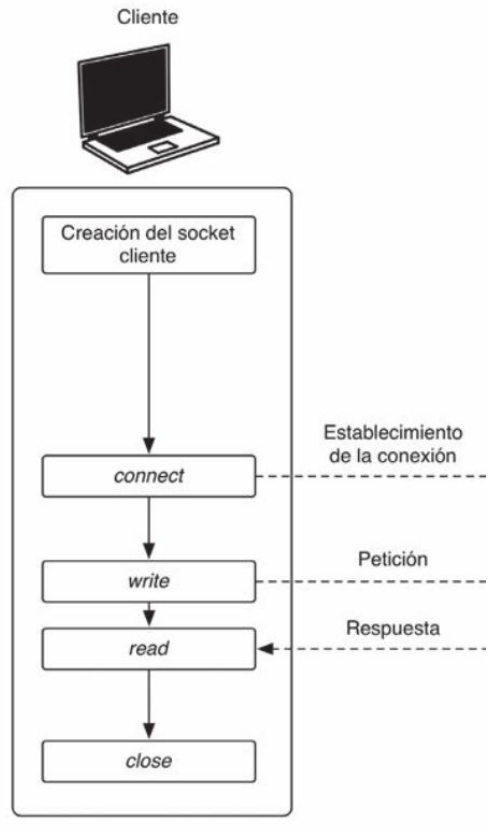
Sockets Stream: Proceso Cliente

1. Creación del socket cliente.

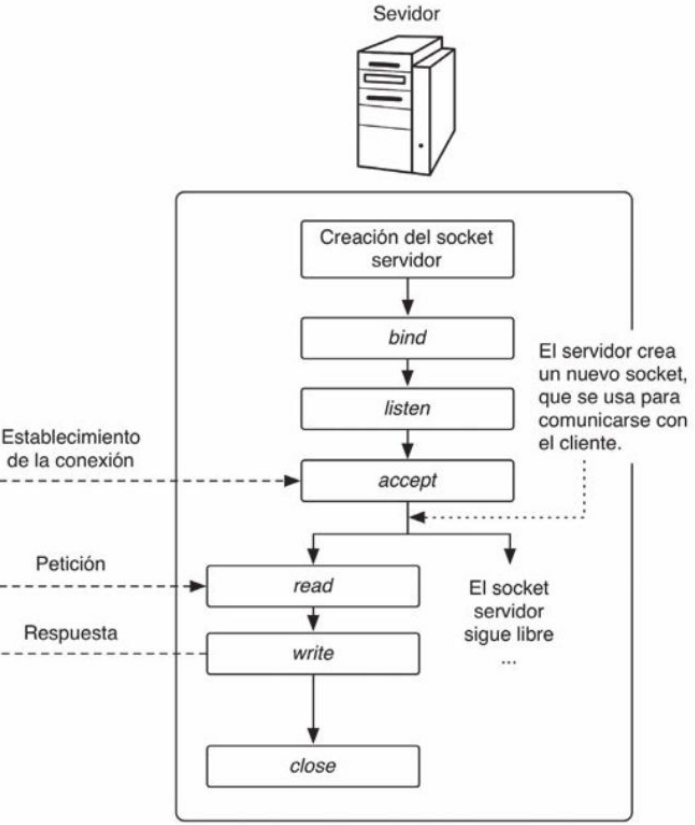
2. Se localiza el socket servidor, utilizando la dirección IP y el puerto de este y se crea un canal de comunicación entre ambos.

3. Una vez establecida la conexión, cliente y servidor pueden enviar y recibir mensajes mediante operaciones de lectura (read) y escritura (write).

4. Si se desea finalizar la comunicación, el proceso cliente puede cerrar el socket cliente (close).



Sockets Stream: Proceso Servidor

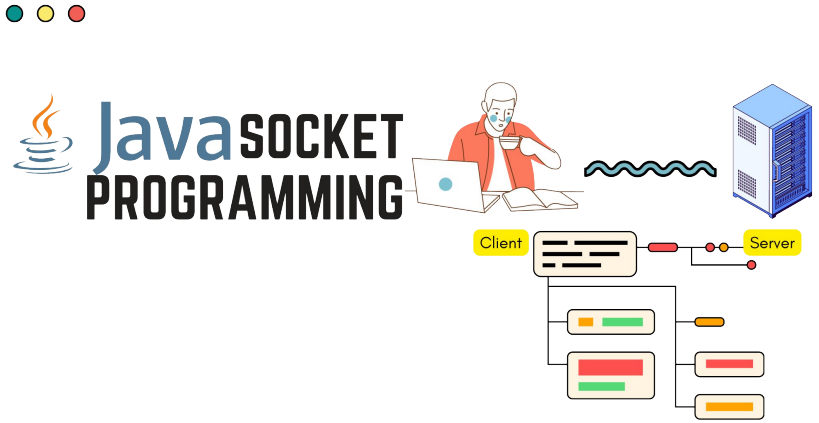


1. Creacion de socket servidor.
2. Asignación de dirección IP y puerto al socket servidor (bind).
3. Escucha por el puerto asignado preparado para aceptar conexiones por parte del cliente (listen).
4. La operación accept bloquea al proceso servidor esperando por una conexión por parte del proceso cliente.
5. Cuando llega una petición de conexión, se crea un nuevo socket que queda conectado con el socket cliente, estableciendo un canal de comunicación estable entre ambos.
6. El socket servidor queda libre, por lo que puede seguir escuchando, a la espera de nuevas conexiones.

Programación de Sockets TCP

En la mayoría de lenguajes de programación de alto nivel existen bibliotecas para crear, destruir y operar con sockets sobre la pila de protocolos IP. En Java existen dos clases principales que permiten la comunicación por sockets TCP:

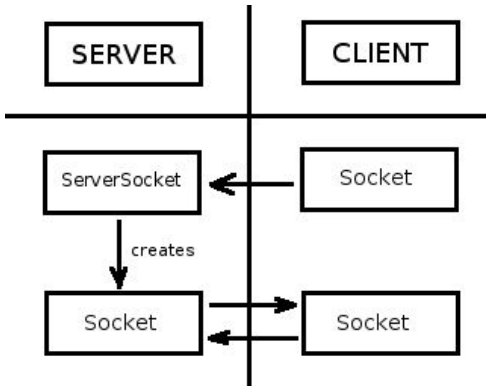
- java.net.Socket, para la creación de sockets stream cliente.
- java.net.ServerSocket, para la creación de sockets stream servidor.



Server Socket

La clase ServerSocket es utilizada por un servidor para crear un socket en el puerto en el que escucha las peticiones de conexión de los clientes.

- Su método *accept* toma una petición de conexión de la cola, o si la cola está vacía, se bloquea hasta que llega una petición.
- El resultado de ejecutar *accept* es una instancia de Socket, a través del cual el servidor tiene acceso a los datos enviados por el cliente.



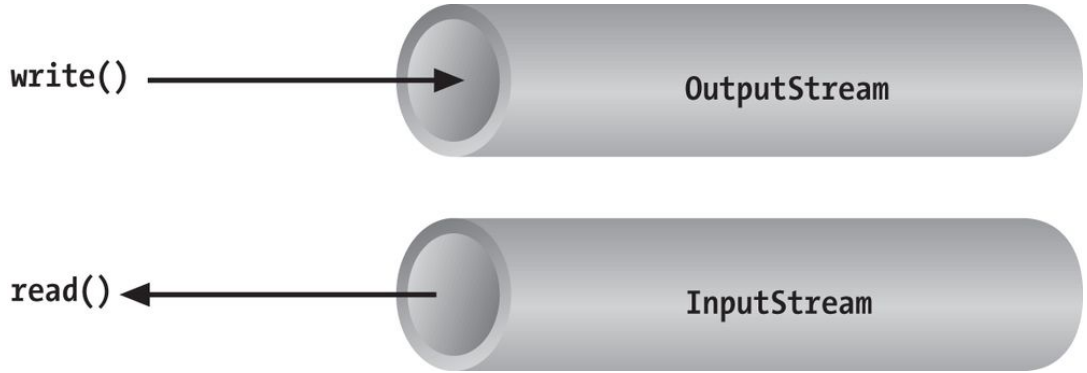
Server Socket

Método	Tipo de Retorno	Descripción
`ServerSocket(int port)`	*Constructor*	Crea un servidor que escucha conexiones en el puerto especificado.
`accept()`	*Socket*	Acepta una conexión entrante, devolviendo un objeto `Socket` que representa la conexión con el cliente.
`close()`	*void*	Cierra el servidor, liberando el puerto y recursos asociados.
`getInetAddress()`	*InetAddress*	Devuelve la dirección IP en la que el servidor está escuchando.
`getLocalPort()`	*int*	Devuelve el número del puerto en el que el servidor está escuchando.
`isClosed()`	*boolean*	Indica si el servidor está cerrado.
`setSoTimeout(int timeout)`	*void*	Establece un tiempo máximo (en milisegundos) para que el método `accept()` espere una conexión. Si se supera, lanza una `SocketTimeoutException`.
`bind(SocketAddress endpoint)`	*void*	Vincula el servidor a una dirección específica y un puerto.

Socket

La clase **Socket** se utiliza para crear y operar con sockets stream clientes. Es utilizada tanto por el cliente como por el servidor.

- El cliente crea un socket especificando el nombre DNS del host y el puerto del servidor, así se crea el socket local y además se conecta con el servicio.
- Esta clase proporciona los métodos *getInputStream* y *getOutputStream* para acceder a los dos streams asociados a un socket



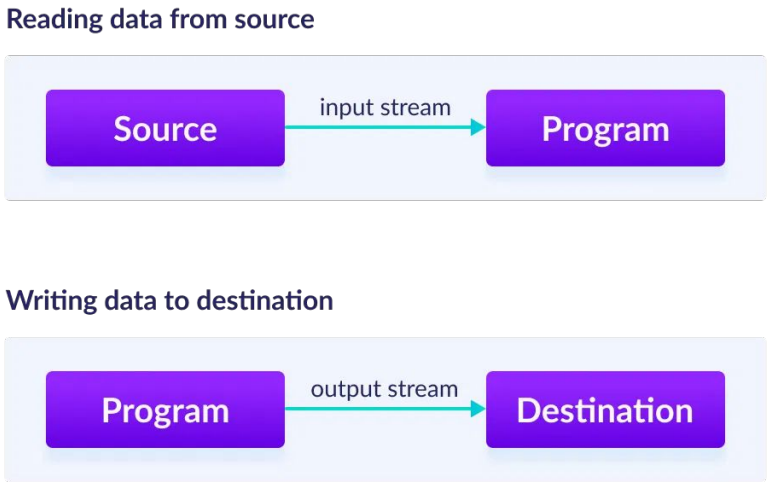
Socket

Método	Tipo de retorno	Descripción
`Socket(String host, int port)`	*N/A (constructor)*	Crea un socket y lo conecta al host y puerto especificados.
`getInputStream()`	`InputStream`	Devuelve un flujo de entrada asociado al socket, usado para recibir datos.
`getOutputStream()`	`OutputStream`	Devuelve un flujo de salida asociado al socket, usado para enviar datos.
`close()`	`void`	Cierra el socket y libera los recursos asociados.
`connect(SocketAddress endpoint)`	`void`	Conecta el socket al endpoint especificado.
`isConnected()`	`boolean`	Indica si el socket está actualmente conectado a un destino.
`isClosed()`	`boolean`	Indica si el socket ha sido cerrado.
`setSoTimeout(int timeout)`	`void`	Establece un tiempo máximo (en milisegundos) para operaciones de lectura.

Gestión de la E/S de Sockets

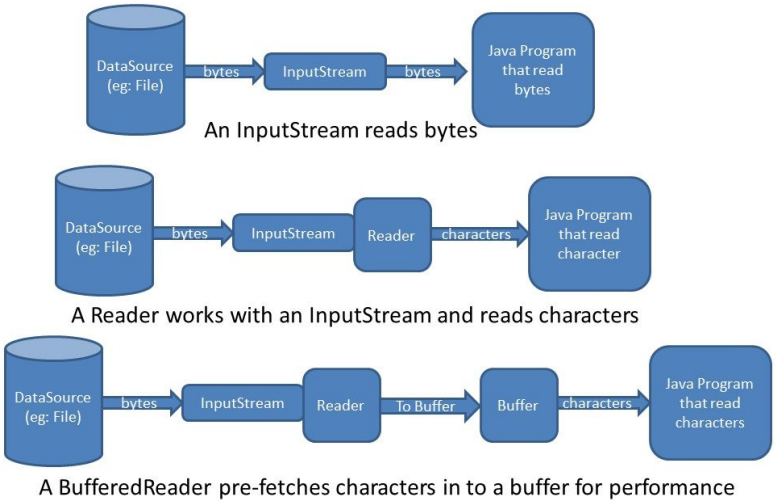
A través de los streams enviamos bytes, que es la forma más básica de generar información, bien sea a través de la red o entre procesos.

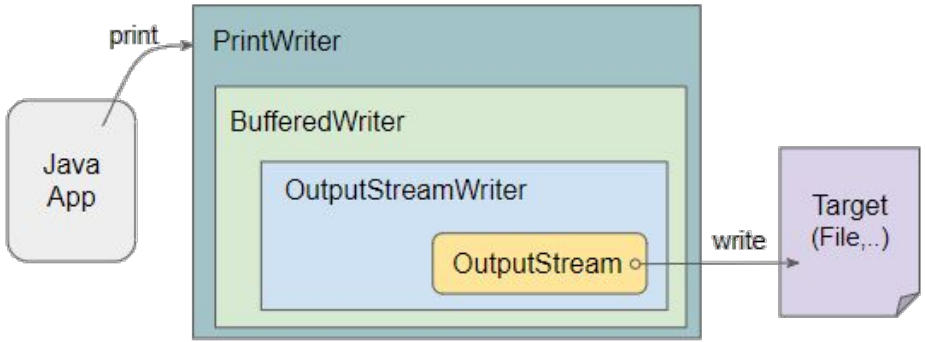
Como es complicado gestionar a nivel de bytes toda la información que queremos enviar o recibir, usamos *Decorators* o *Wrappers* para enviar tipos de datos de un nivel de abstracción mayor.



Gestión de la E/S de Sockets

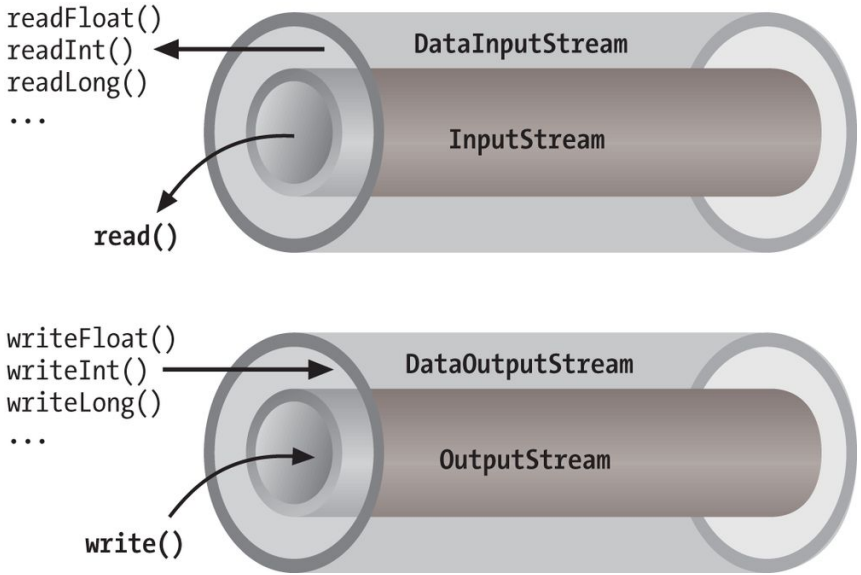
En los temas anteriores, cuando hemos tenido que intercambiar información entre procesos, hemos estado usando *BufferedReader* y *PrintWriter*. Estas clases trabajan a nivel de *Strings*, y son muy útiles cuando lo que queremos intercambiar a través de los streams son cadenas de texto.





Gestión de la E/S de Sockets

Sin embargo, puede haber ocasiones en las que nos interese trabajar con tipos de datos. *DataInputStream* y *DataOutputStream* proporcionan métodos para leer y escribir Strings y todos los tipos de datos primitivos de Java, incluyendo números y valores booleanos.



Gestión de la E/S de Sockets

DataOutputStream codifica esos valores de forma independiente de la máquina y los envía al stream de más bajo nivel para que los gestione como bytes. *DataInputStream* hace lo contrario.

Así, podemos trabajar con *DataInputStream* y *DataOutputStream* a partir de los streams que nos proporcionan los sockets

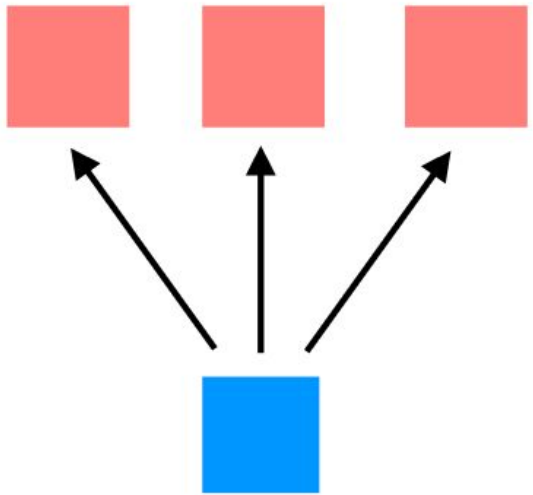
```
// Código en el cliente
DataInputStream dis = new DataInputStream(socket.getInputStream());
dis.readDouble();

// Código en el servidor
DataOutputStream dos = new DataOutputStream(socket.getOutputStream());
dis.writeDouble(number);
```

Gestión de la E/S de Sockets

Es muy importante no mezclar diferentes *wrappers* en el mismo sistema. Aunque todos acaban utilizando el *InputStream* y el *OutputStream*, las codificaciones y la forma de enviar la información no es la misma.

Por lo que, si usas *DataInputStream* en el cliente para leer, debes usar *DataOutputStream* en el servidor para enviar.

client-server



Cliente-Servidor Monohilo TCP

Cliente TCP

```java
public class Client {
	public static void main(String[] args) {
		try (Socket socket = new Socket("localhost", 5000);
			BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
			PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
			BufferedReader console = new BufferedReader(new InputStreamReader(System.in))) {

			System.out.println("Conectado al servidor. Iniciando intercambio de mensajes...");
			String clientMessage, serverMessage;

			while (true) {
				System.out.print("Cliente: "); // Enviar mensaje al servidor
				clientMessage = console.readLine();
				out.println(clientMessage);
				if (clientMessage.equalsIgnoreCase("Bye")) {
					System.out.println("Cliente ha terminado la conexión.");
					break;
				}

				serverMessage = in.readLine(); // Leer mensaje del servidor
				if (serverMessage == null || serverMessage.equalsIgnoreCase("Bye")) {
					System.out.println("El servidor ha terminado la conexión.");
					break;
				}
				System.out.println("Servidor: " + serverMessage);
			}
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

Cliente TCP

Si nos centramos en la parte de comunicaciones, la forma general de implementar un cliente será:

- Crear un objeto de la clase *Socket*, indicando host y puerto donde corre el servicio.
- Obtener las referencias al *stream* de entrada y al de salida al *socket*.
- Leer desde y escribir en el *stream* de acuerdo al protocolo del servicio. Para ello emplear alguna de las facilidades del paquete *java.io.*
- Cerrar los *streams*.
- Cerrar el *socket*.

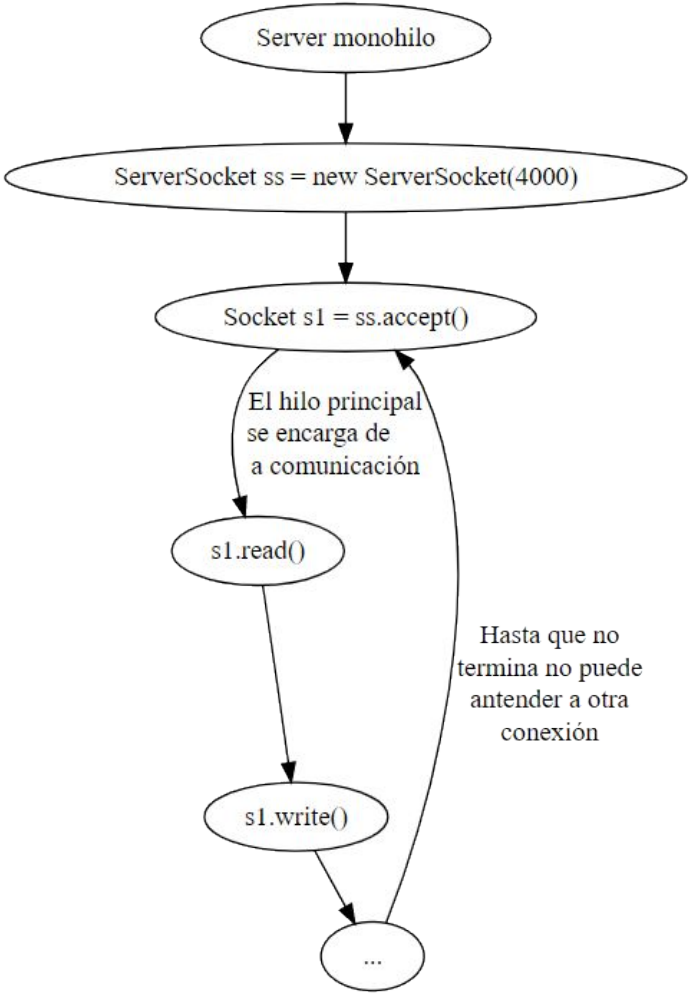
Servidor TCP

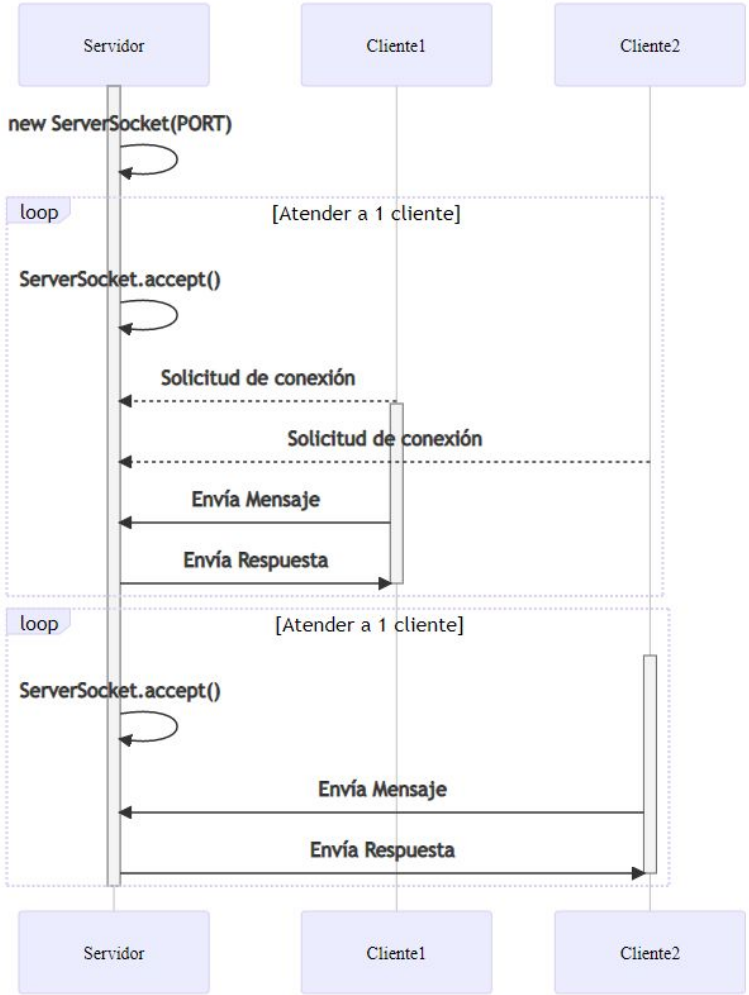
```java
public class SimpleServer {
	public static void main(String[] args) {
		try (ServerSocket serverSocket = new ServerSocket(5000)) {
			System.out.println("Servidor esperando conexiones...");
				try (Socket clientSocket = serverSocket.accept();
				BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
				PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
				BufferedReader console = new BufferedReader(new InputStreamReader(System.in))) {

					System.out.println("Cliente conectado. Iniciando intercambio de mensajes...");

					String clientMessage, serverMessage;
					while (true) {
						clientMessage = in.readLine(); // Leer mensaje del cliente
						if (clientMessage == null || clientMessage.equalsIgnoreCase("Bye")) {
							System.out.println("El cliente ha terminado la conexión.");
							break;
						}
						System.out.println("Cliente: " + clientMessage);

						System.out.print("Servidor: "); // Enviar mensaje al cliente
						serverMessage = console.readLine();
						out.println(serverMessage);
						if (serverMessage.equalsIgnoreCase("Bye")) {
							System.out.println("Servidor ha terminado la conexión.");
							break;
						}
					}
				}
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

Servidor TCP

La forma de implementar un servidor será:

- Crear un objeto de la clase *ServerSocket* para escuchar peticiones en el puerto asignado al servicio.
- Esperar solicitudes de clientes
- Cuando se produce una solicitud:
  - Aceptar la conexión obteniendo un objeto de la clase *Socket*
  - Obtener las referencias al *stream* de entrada y al de salida al *socket* anterior.
  - Leer datos del *socket*, procesarlos y enviar respuestas al cliente, escribiendo en el stream del *socket*.Para ello emplear alguna de las facilidades del paquete java.io.
- Cerrar los *streams*.
- Cerrar los *sockets*.



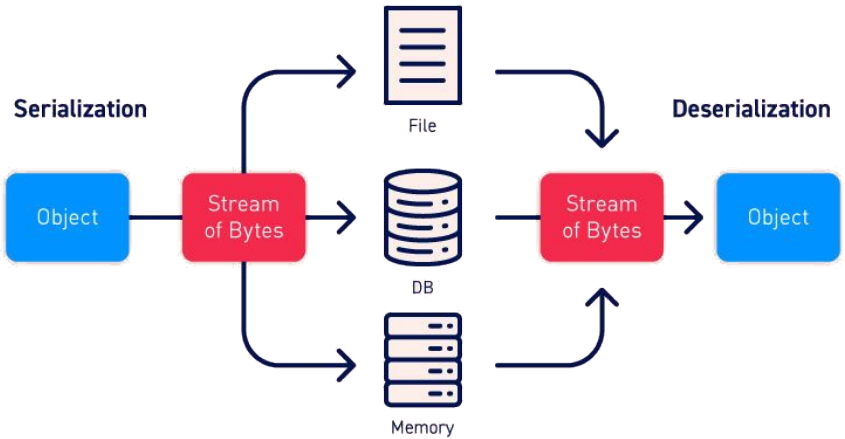


Cliente-Servidor Monohilo TCP

Serialización y envío de objetos

La **serialización** es el proceso de convertir un objeto en una secuencia de bytes para que pueda ser transmitido a través de la red o almacenado en un archivo.

- Cuando un objeto se serializa, Java guarda no solo los datos de los atributos del objeto, sino también la estructura de la clase (nombre de la clase, versión de la clase, etc.).
- La deserialización es el proceso inverso: convertir los bytes de vuelta a un objeto.



Serialización y envío de objetos

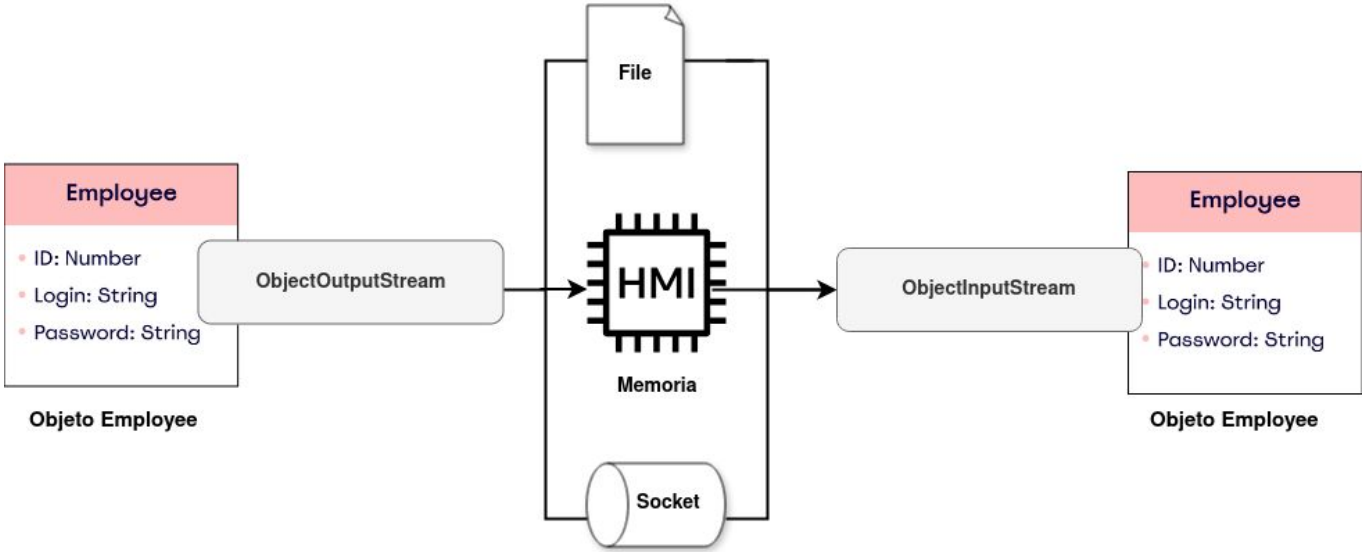
Para que un objeto sea serializable, debe implementar la interfaz Serializable de Java. Esta interfaz no tiene métodos, pero es necesaria para indicar que los objetos de una clase pueden ser convertidos a una secuencia de bytes.

Supongamos que tenemos una clase Persona que queremos enviar a través de la red. La clase debe ser serializable para que sus instancias puedan ser convertidas en una secuencia de bytes.

```
import java.io.Serializable;
public class Persona implements Serializable {
     private String nombre;
     private int edad;

     public Persona(String nombre, int edad) {
     this.nombre = nombre;
     this.edad = edad;
     }
}
```

Serialización y envío de objetos

ObjectOutputStream y **ObjectInputStream** se utilizan para escribir y leer objetos serializables en Java, respectivamente. Su uso está asociado principalmente con operaciones de serialización y deserialización en programas que requieren persistencia o comunicación en red.



Serialización y envío de objetos

ObjectOutputStream

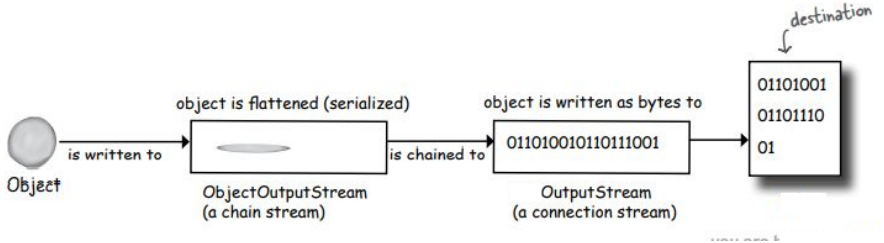
El flujo ObjectOutputStream se utiliza para escribir objetos en una secuencia de salida (como un archivo, un socket o cualquier otra fuente que soporte flujos de salida). Convierte los objetos serializables en una representación binaria que puede ser almacenada o transmitida.

- Solo los objetos que implementan *Serializable* pueden ser escritos.
- Además de objetos, también puedes escribir tipos de datos primitivos.
- *writeObject(Object obj)*: Escribe un objeto en el flujo.

Serialización y envío de objetos

ObjectOutputStream

```
try (ServerSocket serverSocket = new ServerSocket(5000)) {
    try (Socket cliente = serverSocket.accept();
           ObjectOutputStream oos = new ObjectOutputStream(cliente.getOutputStream())) {

        System.out.println("Cliente conectado. Enviando datos...");
        oos.writeObject(new Persona("Laura", 35)); // Enviar un objeto
        oos.writeInt(12345); // Enviar un número adicional

        oos.flush(); // Asegurarse de que los datos se envían
    }
} catch (IOException e) {
    e.printStackTrace();
}
```



Serialización y envío de objetos

ObjectInputStream

El flujo *ObjectInputStream* se utiliza para leer objetos desde una secuencia de entrada. Convierte la representación binaria de un objeto en su forma original en Java.

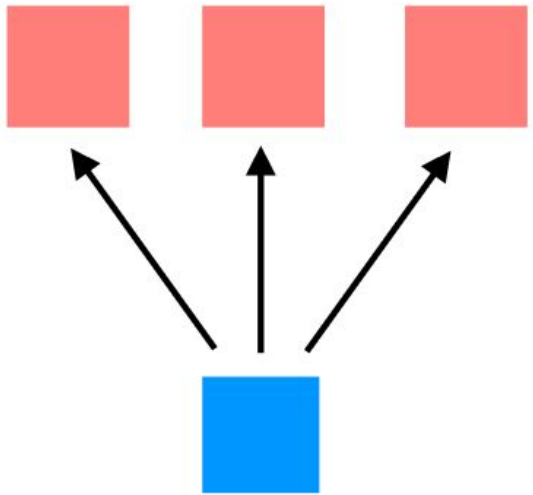
- Solo se pueden leer objetos que fueron escritos con *ObjectOutputStream* y son serializables.
- El flujo utiliza la clase original del objeto para reconstruirlo, por lo que debes asegurarte de que la clase esté disponible en el *classpath*.
- *readObject()*: Lee y deserializa un objeto del flujo. Es necesario realizar un cast al tipo esperado.

Serialización y envío de objetos

ObjectInputStream

```
try (Socket socket = new Socket("localhost", 5000);
     ObjectInputStream ois = new ObjectInputStream(socket.getInputStream())) {

     // Leer el objeto enviado por el servidor
     Persona persona = (Persona) ois.readObject();
     int numero = ois.readInt();

     System.out.println("Recibido: " + persona + ", número: " + numero);
} catch (IOException | ClassNotFoundException e) {
     e.printStackTrace();
}
```

client-server

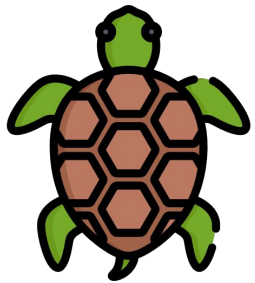


Cliente-Servidor Multihilo TCP

Servidor Multihilo

Un **Servidor Monohilo** solo es capaz de atender a un cliente a la vez.

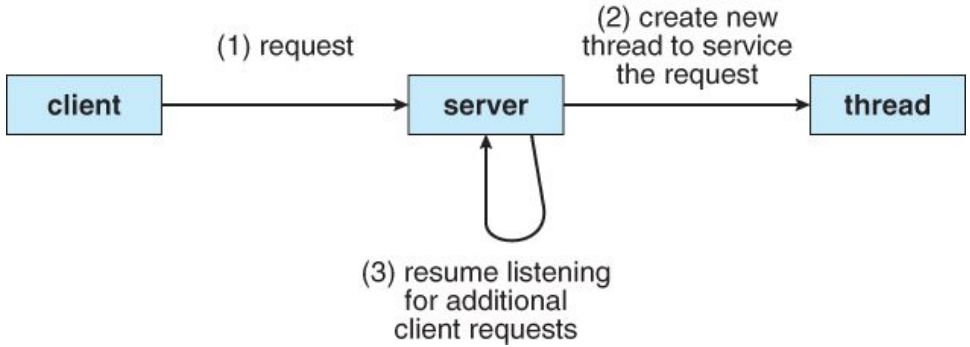
- Cuando un cliente se conecta, el servidor acepta la conexión y queda ocupado procesando las solicitudes de ese cliente hasta que la comunicación termina.
- Si un cliente ocupa mucho tiempo del servidor, otros tendrán que esperar hasta que el servidor termine con la conexión actual.
- El servidor no puede aprovechar eficientemente los recursos del sistema (como CPUs con varios núcleos), ya que todo ocurre en un único hilo.



Servidor Multihilo

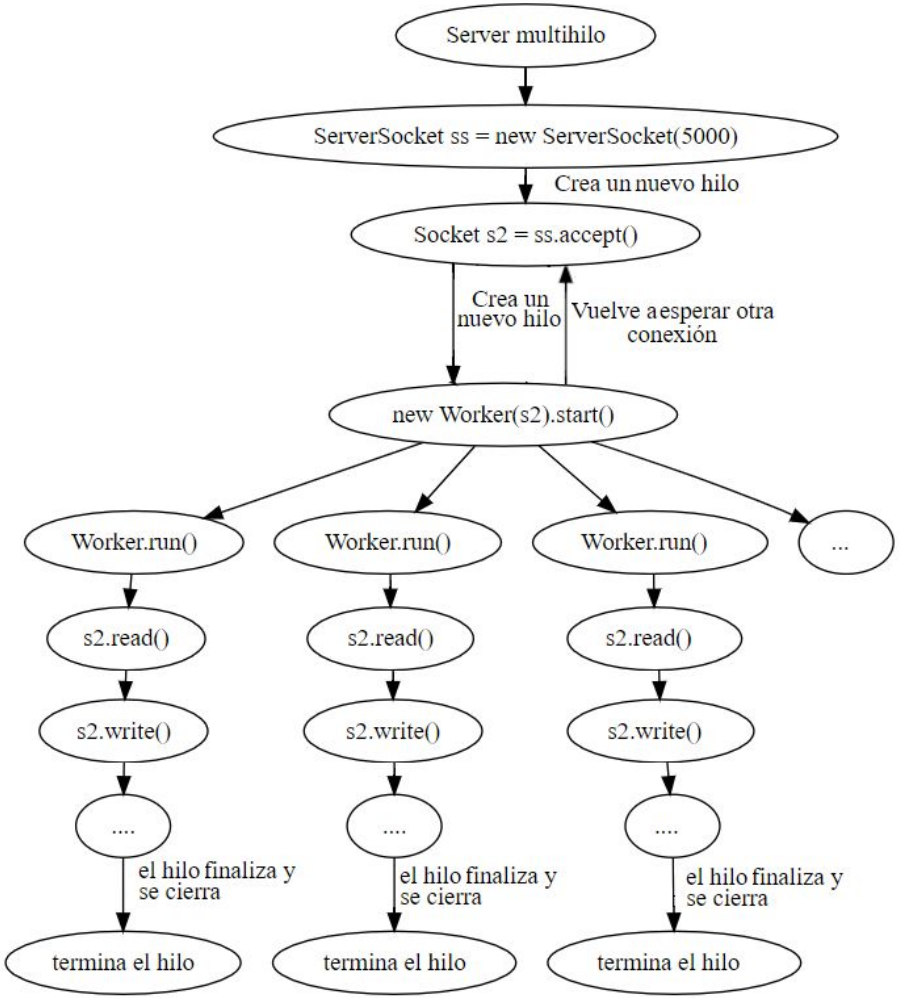
Si queremos que un servidor pueda atender varias peticiones de forma simultánea, debemos usar hilos para dotarlo de esa capacidad. Un **servidor multihilo:**

- Crea un nuevo hilo para cada cliente conectado.
- Permite manejar múltiples clientes simultáneamente.
- Usa *Thread* para gestionar los hilos.



Servidor Multihilo

- El servidor multihilo crea un nuevo hilo que se encarga de las operaciones de E/S con el cliente. Mientras tanto puede esperar la conexión de nuevos clientes con los que volverá a hacer lo mismo.
- El servidor multihilo se ayuda de una clase *`Worker`* que hereda de *Thread*, pudiendo así ejecutarse concurrentemente con el hilo principal.
- Esta clase *`Worker`* es la encargada de realizar toda la comunicación con el cliente y el servidor. Para poder hacerlo, en su constructor recibe el *Socket* que se crea cuando se recibe la conexión de un cliente *`ServerSocket.accept()`*.



Servidor Multihilo

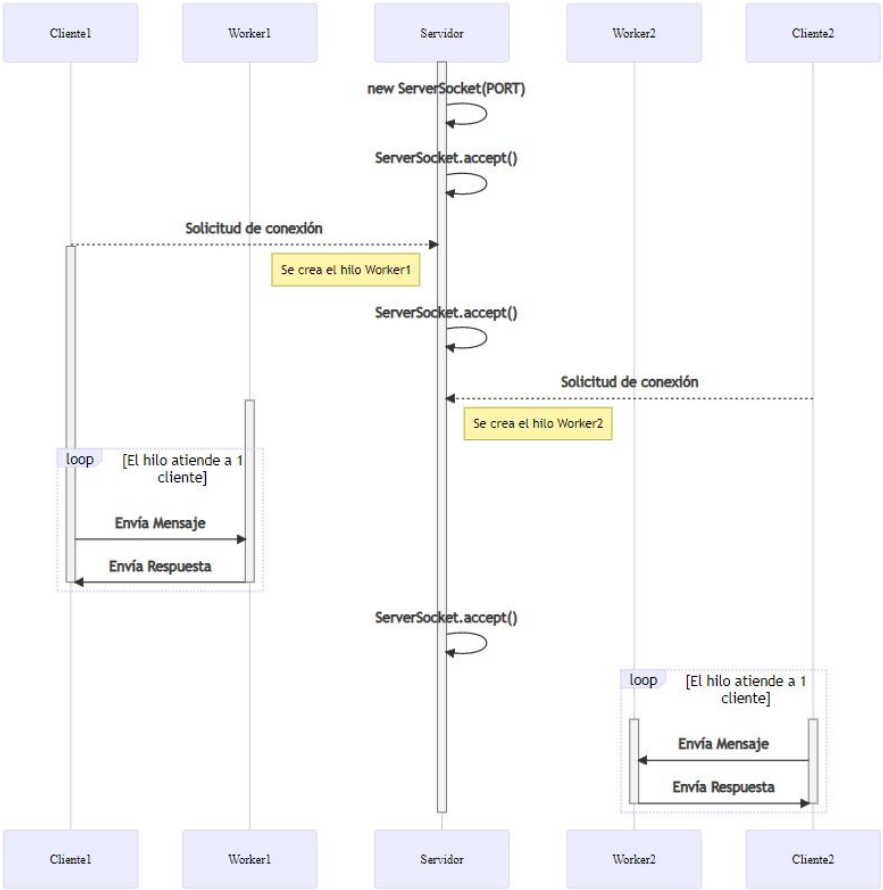
Clase Servidor Multihilo

```
public static final int PORT = 4444;
public static void main(String[] args)  {
      ServerSocket socketServidor = null;
      try {
            socketServidor = new ServerSocket(PORT);
      } catch (IOException e) {
            System.out.println("No puede escuchar en el puerto: " + PORT);
      }
      Socket socketCliente = null;
      System.out.println("Escuchando: " + socketServidor);
      try {
            while (true) {
                  // Se bloquea hasta que recibe alguna petición de un cliente
                  socketCliente = socketServidor.accept();
                  // Se crea un nuevo hilo que se encargará de la comunicación con el cliente
                  new Worker(socketCliente).start();
            }
      ...
}
```

Servidor Multihilo

Clase Worker

```
public class Worker extends Thread {
	private Socket socketCli;
	private BufferedReader entrada = null;
	private PrintWriter salida = null;
	....
	@Override
	public void run() {
	try {
		entrada = new BufferedReader(new InputStreamReader(socketCli.getInputStream())); // Canal de entrada
		// Establece canal de salida
		salida = new PrintWriter(new BufferedWriter(new OutputStreamWriter(socketCli.getOutputStream())), true);

		String mensajeRecibido = entrada.readLine(); // Realizamos la comunicación entre servidor y cliente
		System.out.println("<-- Cliente: " + mensajeRecibido);

		String mensajeEnviado = "Mensaje enviado desde el servidor al cliente"; // Hacemos un envío al cliente
		salida.println(mensajeEnviado);
		System.out.println("--> Cliente: " + mensajeEnviado);
	}
	....
}
```



Cliente-Servidor Multihilo TCP