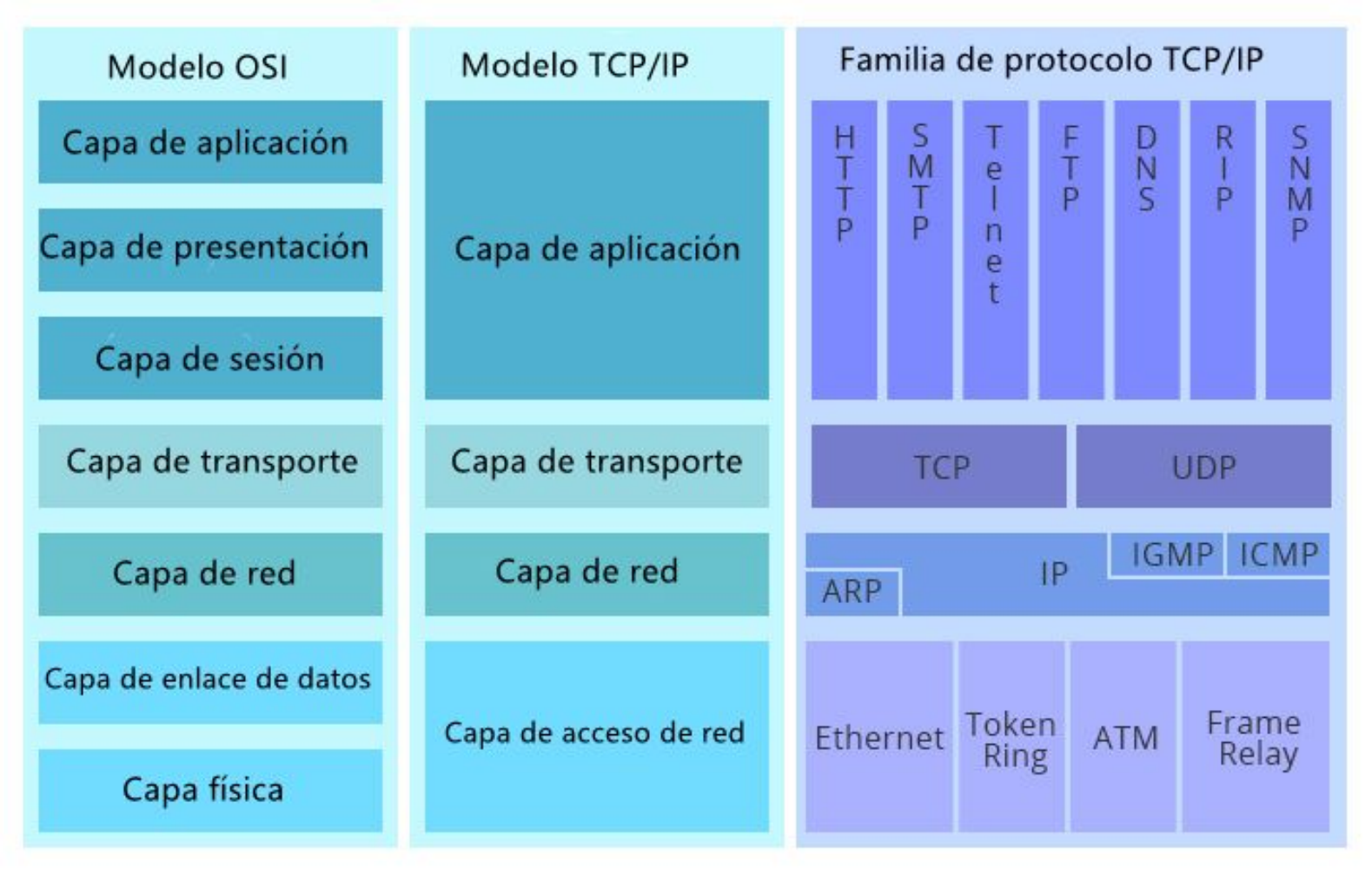
PROGRAMACIÓN DE COMUNICACIONES EN RED

PSP

Repaso de Conceptos: Modelo OSI y sus protocolos



Repaso de conceptos: Nivel de enlace

Cada capa tiene una responsabilidad específica, y los datos se "encapsulan" a medida que descienden por las capas.

Capa de Aplicación

Donde residen los protocolos que usan las aplicaciones (HTTP, FTP, SMTP). Proporciona los servicios al usuario.

Capa de Transporte

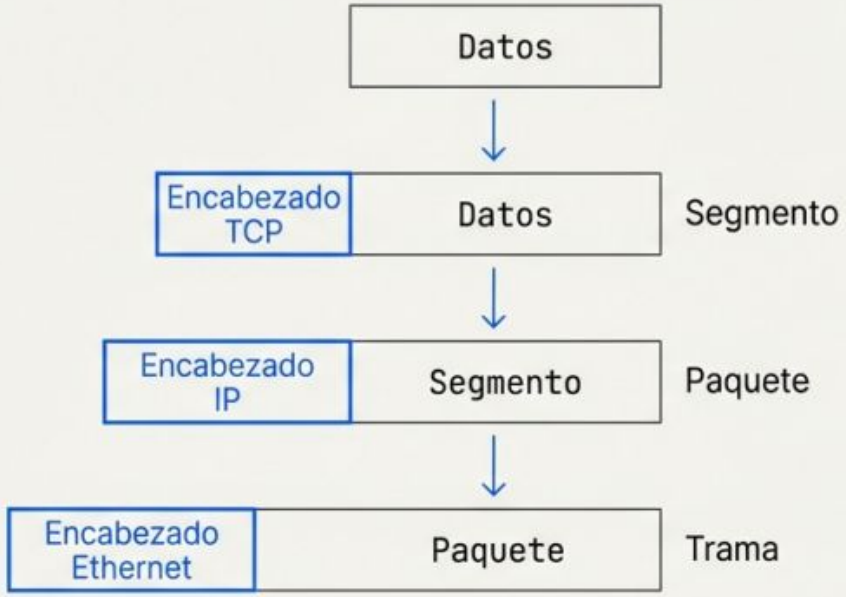
Gestiona la comunicación entre procesos. Aquí es donde operan TCP y UDP, definiendo '*cómo*' se envían los datos.

Capa de Red

Se encarga del direccionamiento y enrutamiento de paquetes a través de la red usando direcciones IP.

Capa de Acceso a Red

Maneja la transmisión física de los datos sobre el medio (Ethernet. Wi-Fi).



Repaso de conceptos: Nivel de Red

El **nivel de red** comunica directamente interfaces de red directamente conectadas a través de un switch.

Cada tarjeta de red es identificada por su dirección MAC (6 bytes separados por :) Ejemplo: *38:2c:b5:79:6b*

Las tarjetas de red se comunican mediante el protocolo IP. Cada tarjeta tendrá una dirección IP y una máscara de red.

Existen dos versiones del protocolo IP:

- IPv4. Las direcciones IP están formadas por 4 bytes.
- IPv6 Las direcciones IP están formadas 16 bytes.

Repaso de conceptos: Nivel de transporte

En las comunicaciones a nivel de transporte se realizan entre un puerto en un host y otro puerto en otro host.

- Un puerto viene identificado por 16 bits
- El host viene identificado por una Ip
- **Socket** = combinación IP+ puerto

En el nivel de transporte tenemos dos protocolos: UDP y TCP

- **UDP**. Envían paquetes de datos independientes (datagramas). No hay control de errores ni existe un orden de llegada con respecto al de emisión. Envío rápido
- **TCP** En este protocolo antes de enviar ningún paquete se establece una conexión lógica. Los datos se reciben en el orden de emisión y existe un mecanismo de retransmisión en caso de errores. Envío fiable

Repaso de conceptos: Nivel de transporte

En la capa de transporte, dos protocolos dominan la comunicación. La elección entre ellos define la naturaleza de la conexión.

TCP (Protocolo de Control de Transmisión)

Analogía: Una llamada telefónica.

Fiable y orientado a la conexión. Antes de enviar datos, se establece una conexión lógica. Garantiza que los datos lleguen en orden y sin errores, retransmitiendo lo que sea necesario.

UDP (Protocolo de Datagramas de Usuario)

Analogía: Enviar una postal.

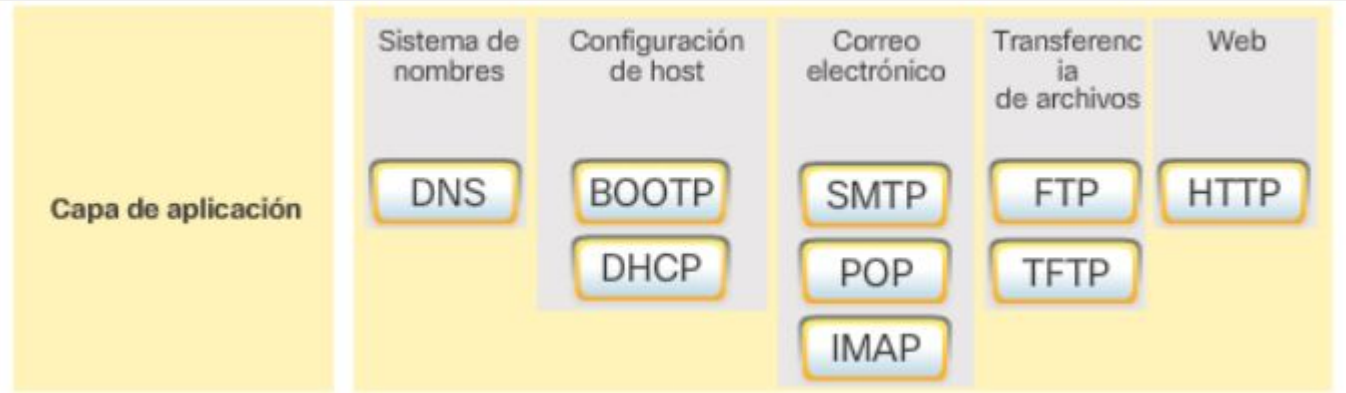
Rápido y no orientado a la conexión. Envía paquetes de datos (datagramas) de forma independiente. No hay garantía de entrega, orden ni control de errores. Su prioridad es la velocidad.

Repaso de conceptos: Nivel de aplicación

Se implementan sobre TCP y/o UDP.

En este nivel se encuentran las aplicaciones disponibles para los usuarios: FTP, Telnet, HTTP,...

Nos centraremos en el protocolo TCP/IP



Repaso de conceptos

Es tu turno:

Explica brevemente qué es y para qué sirve:

- DNS
- DHCP
- SMTP
- POP
- IMAP
- FTP
- HTTP
- HTTPS

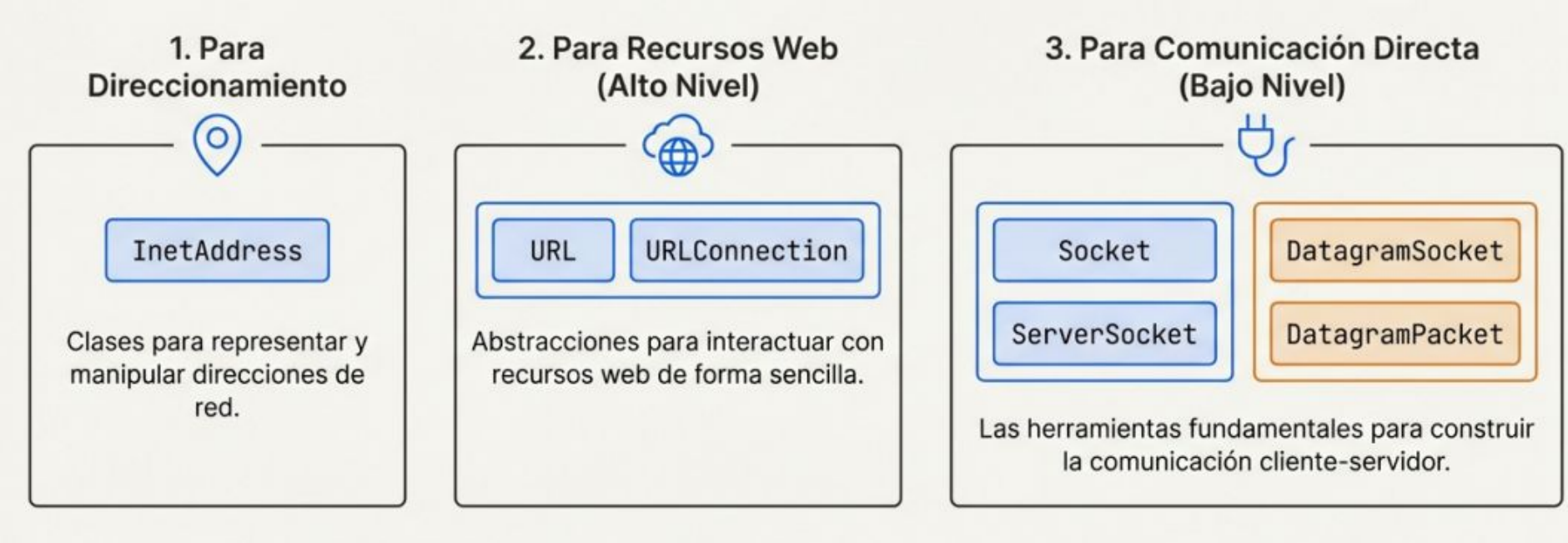
Clases Java utilizadas para la comunicación en red

El paquete java.net contiene clases e interfaces para la implementación de aplicaciones de red. Entre otras::

- Clase **URL**: representa un puntero a un recurso en la red.
- Clase **URLConnection**:: clase abstracta que contiene métodos que permiten la comunicación entre la aplicación y una URL.
- Clases **ServerSocket** y **Socket** para dar soporte a sockets **TCP**.
  - ***ServerSocket***: utilizada por el programa servidor para crear un socket en el puerto en el que **escucha** las peticiones de conexión de los clientes.
  - ***Socket***: utilizada tanto por el cliente como por el servidor para comunicarse entre sí leyendo y escribiendo datos usando streams.

Clases Java utilizadas para la comunicación en red

El paquete java.net contiene clases e interfaces para la implementación de aplicaciones de red. Entre otras::

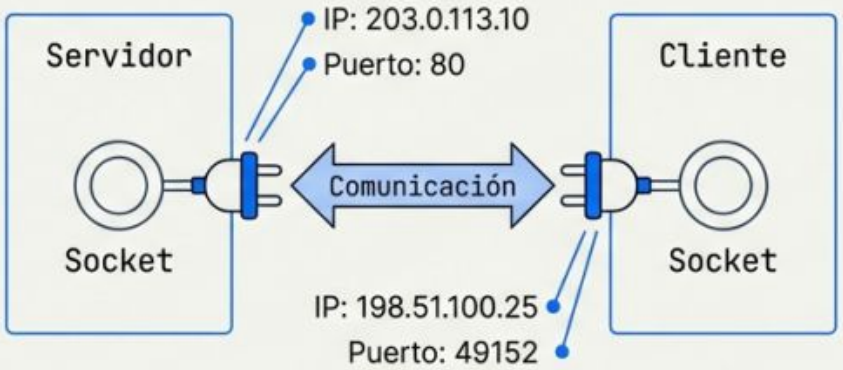


Sockets

Un **socket** es el conector que representa un extremo en una comunicación de red entre dos procesos. Para que la comunicación sea única, un socket combina dos elementos clave:

- **Dirección IP**: Identifica la máquina (host) en la red.
- **Número de Puerto**: Identifica la aplicación o proceso específico dentro de esa máquina.

`Socket = Dirección IP + Puerto`



Sockets

- *El proceso cliente debe conocer la IP y el puerto del proceso servidor.*
- *El cliente podrá enviar el mensaje por el puerto que quiera.*
- *Los mensajes al servidor le deben llegar al puerto acordado.*

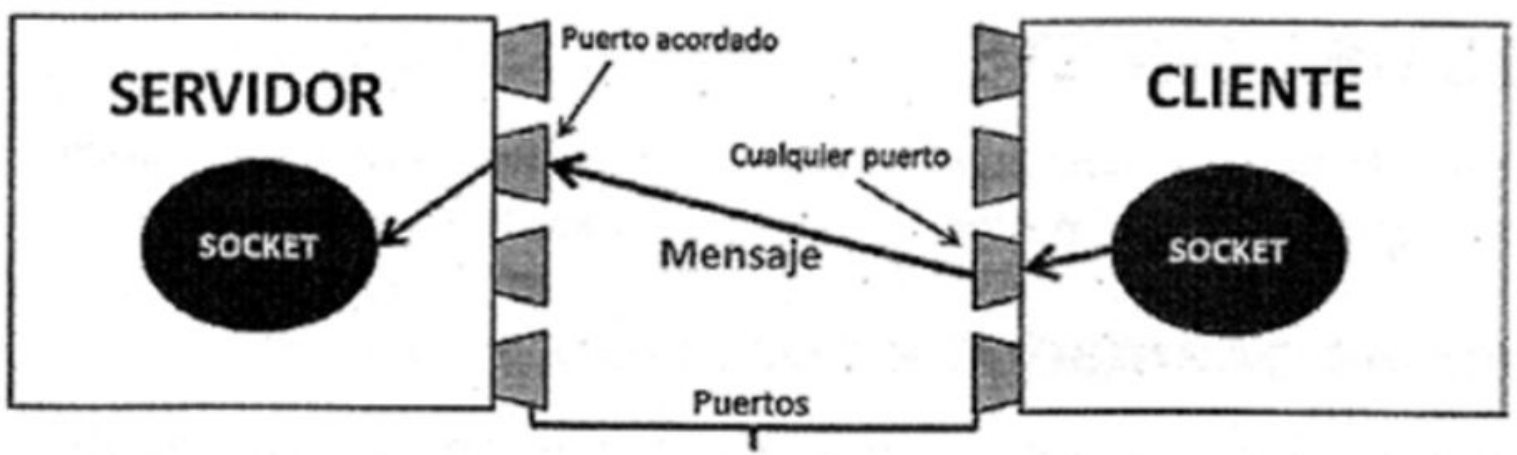


Figura 3.2. Socket y puertos.

Sockets: Servidor

El `ServerSocket` es el primer paso en el lado del servidor. Su misión es esperar y aceptar peticiones de conexión de los clientes en un puerto específico.

El Proceso en 2 Pasos

1. Crear y Vincular

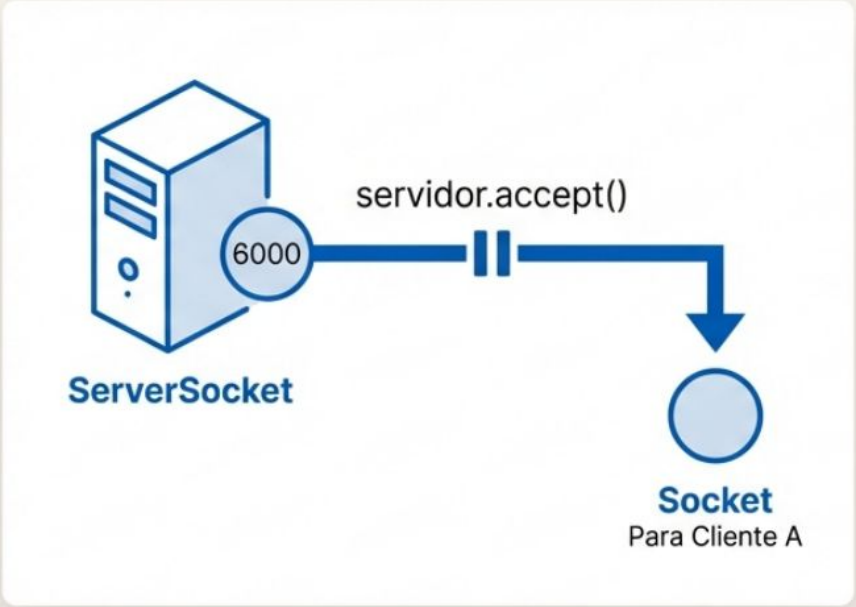
"Estoy escuchando peticiones en el puerto 6000."

```
int puerto = 6000;
ServerSocket servidor = new ServerSocket(puerto);
```

2. Esperar y Aceptar

```
Socket cliente = servidor.accept();
```

"¡Conexión recibida! El método devuelve un nuevo Socket para comunicarme exclusivamente con este cliente."



Sockets: Clientes

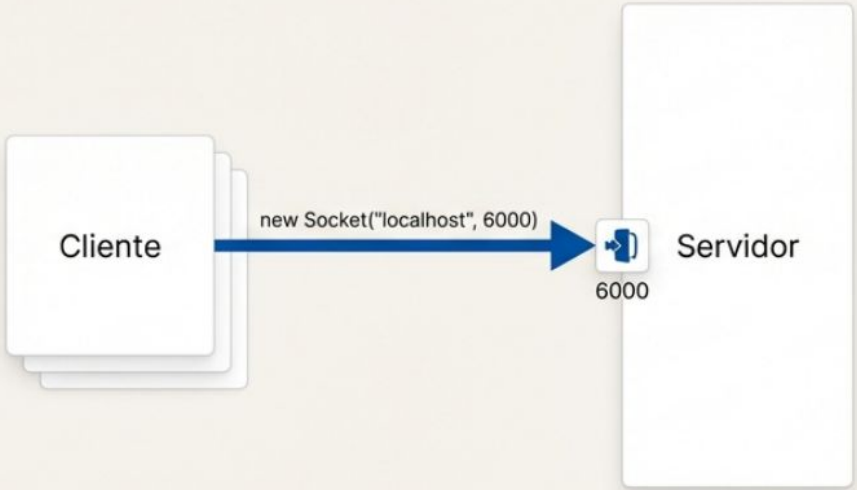
El `Socket` del cliente es el que inicia la conversación. Debe conocer la dirección IP y el puerto del servidor.

El Proceso en 1 Paso

1. **Crear y Conectar**

```
String host = "localhost";
int puerto = 6000;
Socket cliente = new Socket(host, puerto);
```

Intento conectarme a "localhost" en el puerto 6000.



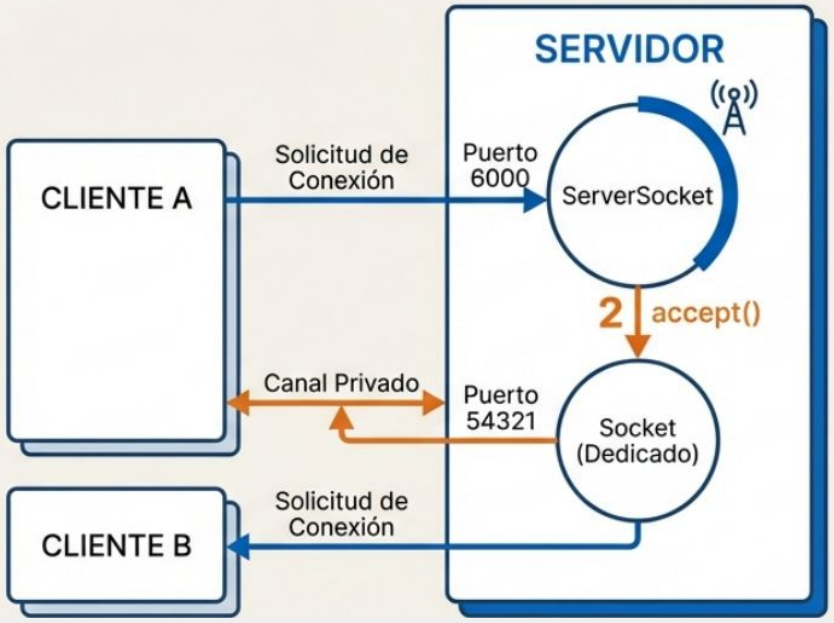
Resultado

Si el servidor tiene un `ServerSocket` escuchando y llama a `accept()`, la conexión se establece. Ambos extremos ahora tienen un `Socket` para comunicarse.

Sockets: Funcionamiento

1. 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, por lo que el cliente realiza la petición a la máquina a través del puerto.
2. Servidor: acepta la conexión. Una vez aceptada, el servidor obtiene **un nuevo socket sobre un PUERTO DIFERENTE** para seguir atendiendo *atendiendo las peticiones de conexión mediante el socket original.*

1. **Liberar el Puerto Principal**
   El `ServerSocket` debe permanecer libre en el puerto original para seguir atendiendo las peticiones de conexión de nuevos clientes.
2. **Crear un Canal Dedicado**
   La nueva conexión utiliza un puerto diferente para establecer un canal de comunicación privado entre el servidor y ese cliente específico, sin interferir con futuras peticiones.



Sockets

Tipos de sockets:

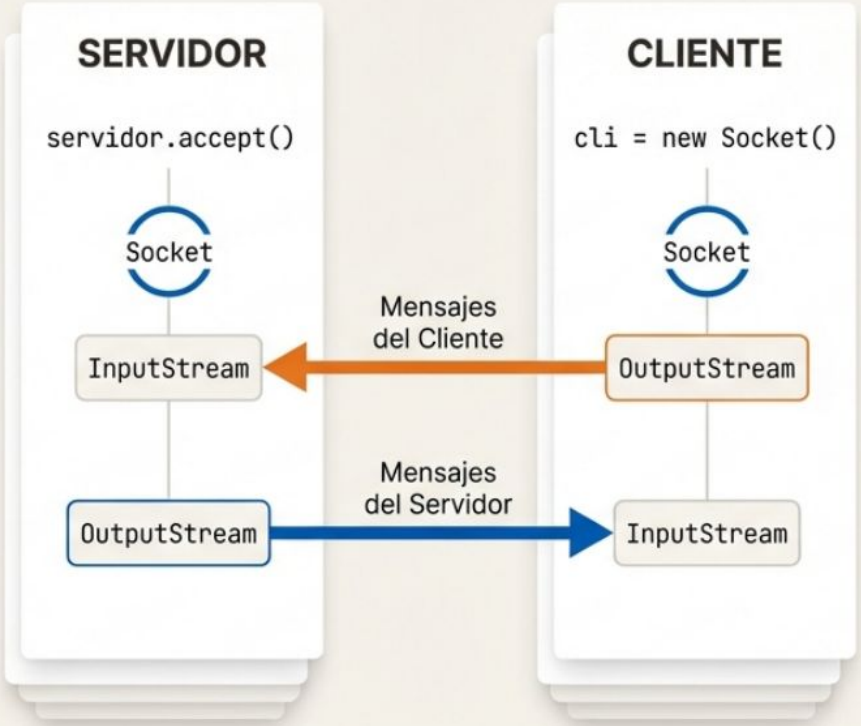
- Orientados a la conexión:: protocolo TCP
- No orientados la conexión:: protocolo UDP

Sockets

Una vez que el `Socket` está conectado, la comunicación se realiza a través de flujos de datos. Cada `Socket` proporciona dos flujos:

- `OutputStream`: Para escribir (enviar) datos al otro extremo. Se obtiene con `socket.getOutputStream()`.
- `InputStream`: Para leer (recibir) datos del otro extremo. Se obtiene con `socket.getInputStream()`.

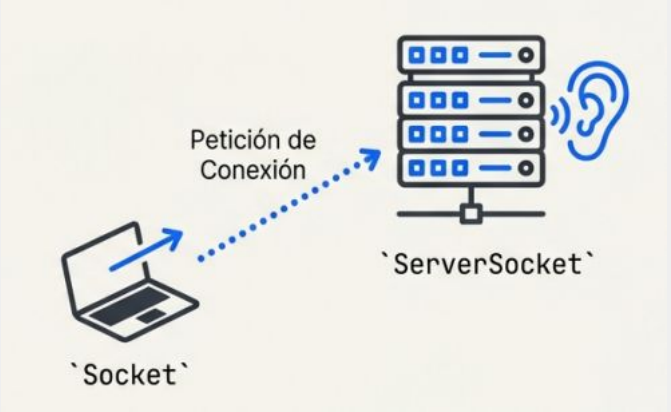
El `OutputStream` del cliente se conecta al `InputStream` del servidor, y viceversa, creando un canal de comunicación bidireccional.



TCP: Estableciendo una Conexión Fiable

El Modelo Cliente-Servidor

- La comunicación **TCP** se basa en dos roles distintos con sus propias herramientas: •
  - **El Servidor (ServerSocket):** Su misión es escuchar en un puerto específico por peticiones de conexión de clientes. Es el punto de espera pasivo.
  - **El Cliente (Socket):** Su misión es iniciar activamente una conexión a la dirección IP y puerto del servidor.



Secuencia:

1. El **ServerSocket** se crea y espera (**accept()**).
2. El **Socket del cliente** se crea, conectándose al servidor.
3. El **accept()** del servidor devuelve un **nuevo Socket** para comunicarse exclusivamente con ese cliente, d**ejando el ServerSocket original libre para escuchar a otros.**

TCP: Estableciendo una Conexión Fiable

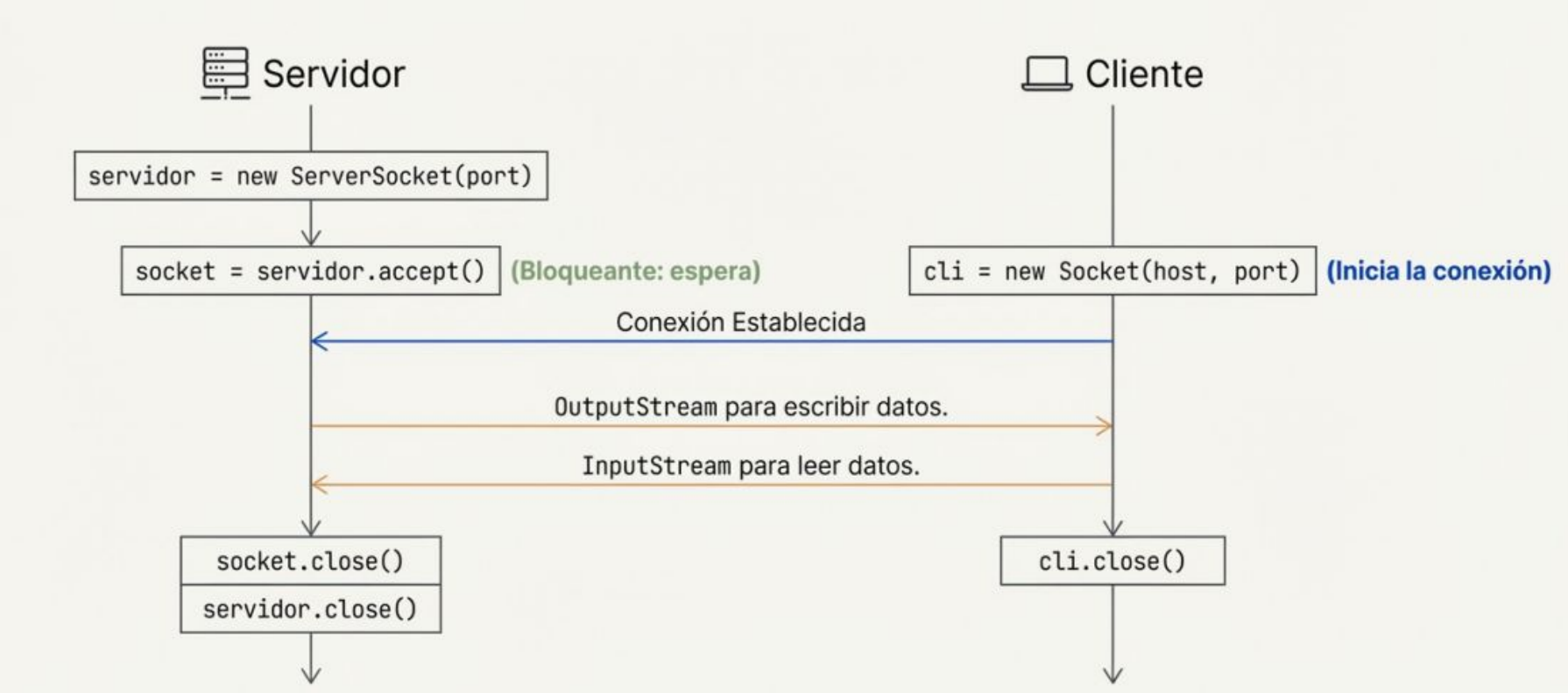
¿Cómo verificar que el puerto es otro?

En el Cliente:

```
String Host = "localhost"; //host servidor con el que el cliente quiere conectarse
int Puerto = 6000;//puerto remoto en el servidor que el cliente conoce
Socket cliente = new Socket(Host, Puerto);//conecta

InetAddress i= cliente.getInetAddress ();
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());
```

TCP: Estableciendo una Conexión Fiable



TCP: Servidor TCP

```
// 1. Definir el puerto (identifica la aplicación en la máquina)
int puerto = 6000;
ServerSocket servidor = null;
Socket clienteConectado = null;

try {
    // 2. Crear el ServerSocket para escuchar peticiones
    servidor = new ServerSocket(puerto);
    System.out.println("Servidor escuchando en el puerto " + puerto);

    // 3. Esperar y aceptar una conexión
    clienteConectado = servidor.accept();
    System.out.println("Cliente conectado desde: " + clienteConectado.getInetAddress());
    // 4. Configurar flujos de lectura y escritura
    BufferedReader entrada = new BufferedReader(new InputStreamReader(clienteConectado.getInputStream()));
    PrintWriter salida = new PrintWriter(clienteConectado.getOutputStream(), true);
    System.out.println("Conectado. Escribe tu mensaje:");
    Scanner sc = new Scanner(System.in);
    String mensaje = entrada.readLine();

    System.out.print("> ");
    System.out.println("Cliente dice: " + mensaje);
    salida.println("Servidor recibió: " + mensaje.toUpperCase());

} catch (IOException e) {
    // Manejo de excepciones de entrada/salida o errores de red [cite: 95]
    System.err.println("Error en el servidor: " + e.getMessage());
    e.printStackTrace();
} finally {
    // 5. Cerrar recursos siempre para liberar el puerto [cite: 116, 133, 134]
    try {
        if (clienteConectado != null)
            clienteConectado.close();
        if (servidor != null)
            servidor.close();
        System.out.println("Recursos cerrados y servidor finalizado.");
    } catch (IOException e) {
        System.err.println("Error al cerrar sockets: " + e.getMessage());
    }       }   }}
```

TCP: Cliente TCP

```
// 1. Definir el host y el puerto del servidor
String host = "localhost"; // IP de la propia máquina
int puerto = 6000;

Socket cliente = null;

try {
    // 2. Se crea el socket y se inicia la conexión (Punto 2 de tu secuencia)
    System.out.println("Conectando al servidor en " + host + " por el puerto " + puerto + "...");
    cliente = new Socket(host, puerto);
    System.out.println("¡Conexión establecida!");

    // 3. Configurar flujos para enviar y recibir datos
    PrintWriter salida = new PrintWriter(cliente.getOutputStream(), true);
    BufferedReader entrada = new BufferedReader(new InputStreamReader(cliente.getInputStream()));

    // Usamos Scanner para que puedas escribir tú el mensaje por consola
    Scanner sc = new Scanner(System.in);

    System.out.print("Escribe un mensaje para el servidor: ");
    String mensajeParaServidor = sc.nextLine();

    // 4. Enviar mensaje al servidor
    salida.println(mensajeParaServidor);
    // 5. Leer la respuesta del servidor
    String respuesta = entrada.readLine();
    System.out.println("Respuesta del servidor: " + respuesta);

} catch (UnknownHostException e) {
    System.err.println("No se puede encontrar el host: " + host);
} catch (IOException e) {
    System.err.println("Error de entrada/salida: " + e.getMessage());
} finally {
    // 6. Cerrar el socket del cliente para liberar recursos
    try {
        if (cliente != null) {
            cliente.close();
            System.out.println("Conexión cerrada.");
        }
    } catch (IOException e) {
        System.err.println("Error al cerrar el cliente: " + e.getMessage());
    }       }   }}
```

Hilos: Conexión de múltiples clientes.

Hasta ahora, los programas servidores que hemos creado solo son capaces de atender a 1 cliente en cada momento.

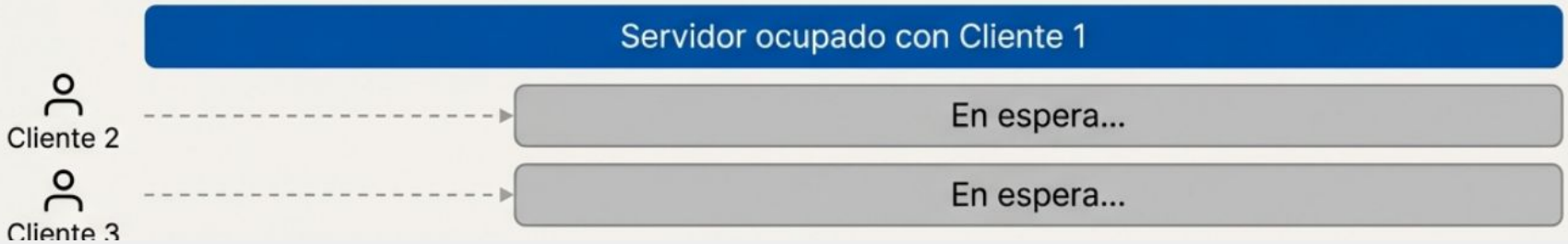
El método accept() y la gestión de flujos (streams) son bloqueantes. Si un cliente está conectado, el servidor no puede atender a otro hasta que el primero termine.

Análisis del Código Secuencial

```
Socket cliente1 = servidor.accept();
// ...toda la comunicación con cliente1...
Socket cliente2 = servidor.accept();
```

El programa se detiene aquí hasta que el cliente 1 se conecta.

Esta línea no se ejecutará hasta que la comunicación con cliente1 haya terminado.



Hilos: Conexión de múltiples clientes.

Sin embargo, lo más típico es que un programa servidor pueda atender a muchos clientes simultáneamente.

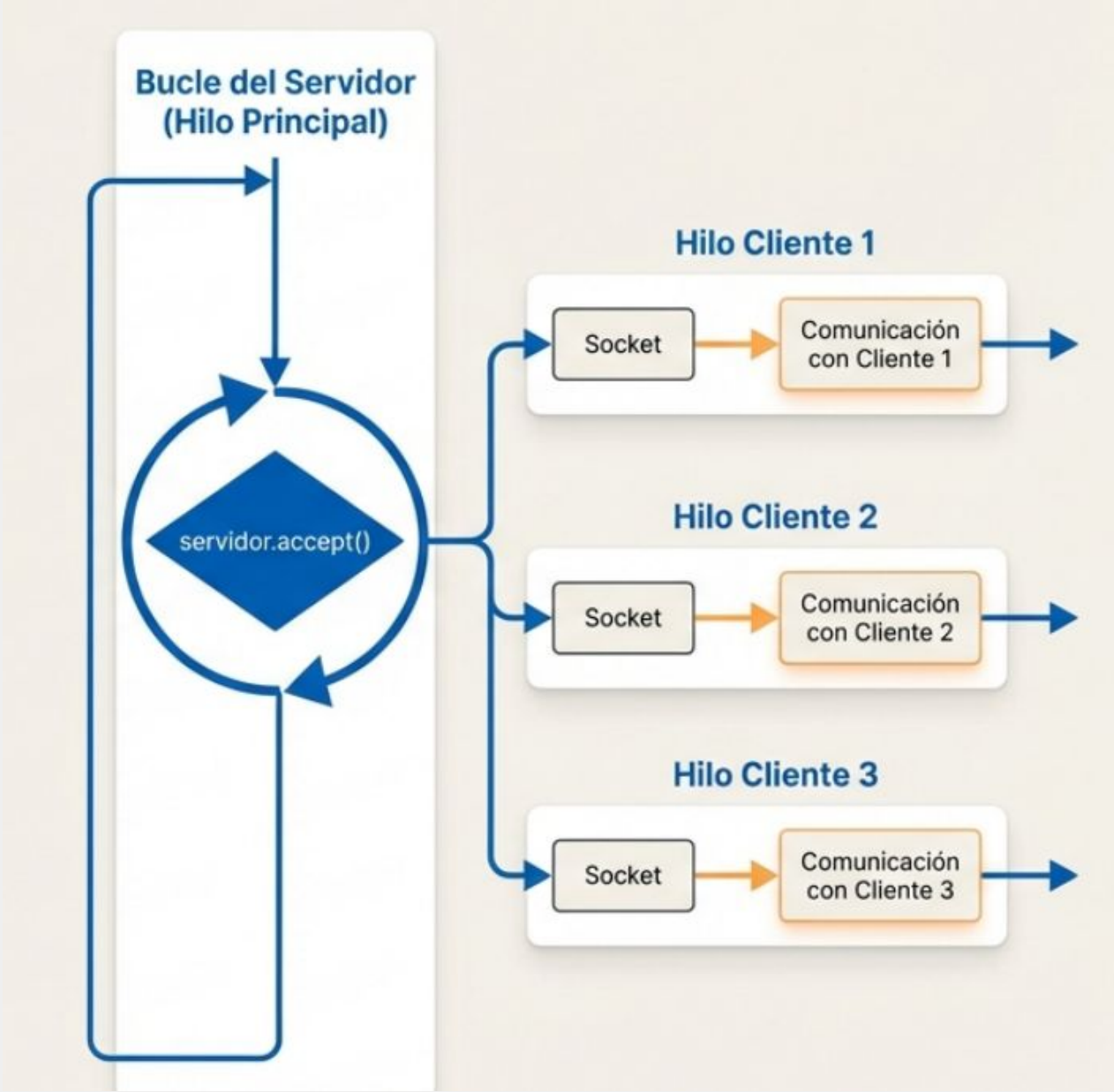
La solución para poder atender a múltiples clientes está en el multihilo: cada cliente será atendido en 1 hilo.

El servidor delega la conversación con cada cliente a un hilo de ejecución separado.

De forma que:

- El hilo principal del servidor se dedica exclusivamente a escuchar peticiones con ServerSocket.accept().
- Por cada conexión aceptada, se lanza un nuevo hilo encargado de gestionar la comunicación con ese cliente específico.

Hilos: Conexión de múltiples clientes.



Hilos: Conexión de múltiples clientes.

Servidores Multihilos: Secuencia

El servidor estará siempre aceptando peticiones. Ante una petición:

1. Se recibe comunicación de cliente: Socket cliente = servidor.accept();
2. Se crea una instancia de una clase que herede de Thread o implemente Runnable. Crearemos una instancia de clase llamada ManejadorCliente que extiende de hilo
3. Y le pasaremos a esta clase en su constructor objeto Socket al nuevo hilo.
4. Por último, llamaremos a start() para iniciar la comunicación en paralelo.

Hilos: Conexión de múltiples clientes.

Servidores Multihilos: Secuencia

El servidor estará siempre aceptando peticiones. Ante una petición:

1. Se recibe comunicación de cliente: Socket cliente = servidor.accept();
2. Se crea una instancia de una clase que herede de Thread o implemente Runnable. Crearemos una instancia de clase llamada ManejadorCliente que extiende de hilo
3. Y le pasaremos a esta clase en su constructor objeto Socket al nuevo hilo.
4. Por último, llamaremos a start() para iniciar la comunicación en paralelo.

Hilos: Conexión de múltiples clientes.

Servidor Multihilos TCP:

```
        int puerto = 6000;
        try (ServerSocket servidor = new ServerSocket(puerto)) {
            System.out.println("Servidor multihilo iniciado en el puerto " + puerto);
            while (true) {
                // 1. Espera a un cliente
                Socket socketCliente = servidor.accept();
                System.out.println("Nuevo cliente conectado: " +
socketCliente.getInetAddress());
                // 2. Lanza un hilo nuevo para este cliente específico
                // Esto permite que el bucle vuelva al accept() inmediatamente
                new ManejadorHilosCliente(socketCliente).start();
            }
        } catch (IOException e) {
            System.err.println("Error en el servidor: " + e.getMessage());
        }
    }
```

Hilos: Conexión de múltiples clientes.

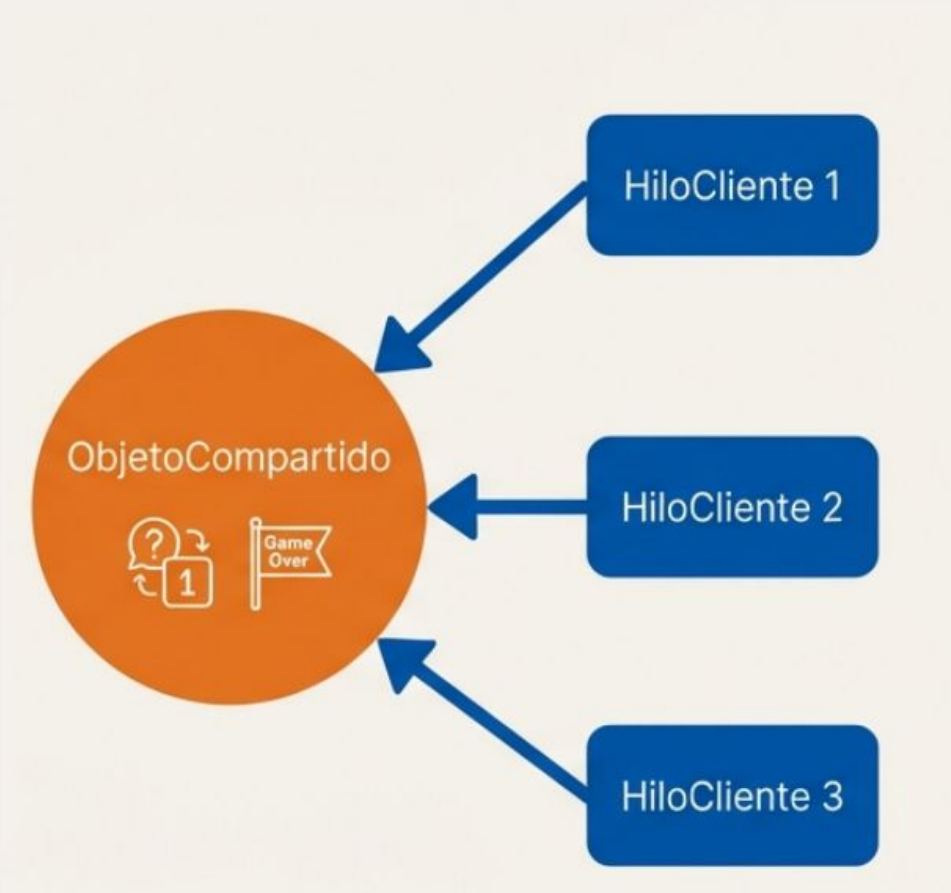
Manejador Hilos Cliente extiende de Thread:

Tendrá:

- Un atributo Socket
- Un constructor que recibe un socket
- En su método run:

```
entrada = new BufferedReader(new InputStreamReader(socket.getInputStream()));
salida = new PrintWriter(socket.getOutputStream(), true);
salida.println("Bienvenido. Escribe algo (o 'fin' para salir):");
String mensaje;
while ((mensaje = entrada.readLine()) != null && !mensaje.equalsIgnoreCase("fin")) {
            System.out.println("Cliente dice: " + mensaje);
            salida.println("Servidor responde: " + mensaje.toUpperCase());
}
if (mensaje != null && mensaje.equalsIgnoreCase("fin")) {
      salida.println("Cerrando sesión. ¡Hasta pronto!");
}
```

Hilos: Compartiendo Objetos



Hilos: Compartiendo Objetos

En este caso, los clientes no generan hilos independientes. Si no que hay un objeto común.

- Tendremos a nuestro ServerSocket escuchando en un puerto
- Cada vez que un cliente se conecte, el servidor aceptará la conexión y lanza un hilo que atiende a ese cliente.
- El hilo recibe el socket del cliente y una referencia al objeto compartido.Hará la tarea que deba con el objeto y devolverá la respuesta al cliente por el OutputStream.
- El objeto que se intercambia debe ser **serializable**. (implements Serializable) para que pueda ser transmitido por la red.

Hilos: Compartiendo Objetos

Intenta resolver este problema:

Se trata de un juego en el que varios clientes (número indeterminado) se conectan a un servidor y tienen que adivinar un número.

En el momento que alguno de los clientes lo adivine, el juego finaliza para todos ellos.

Hilos: Compartiendo Objetos

Servidor:

- Genera un número secreto al inicio.
- Genera un objeto sincronizado que contiene:
  - El número secreto.
  - Una bandera juegoTerminado
  - Un método sincronizado que recibe un número y comprueba si es el buscado, marcando en ese caso juegoTerminado a cierto.
  - Cada cliente que se conecta se atiende en un **hilo**.

Clientes:

- Se conectan al servidor.
- Envían números propuestos.
- Reciben la respuesta del los hilos servidores

Hilos: Compartiendo Objetos

Hilo:

- El hilo estará recibiendo propuestas del cliente e invocando al método sincronizado del objeto:
  - Si el número es correcto → marca juegoTerminado = true y avisa a todos los clientes que el juego ha acabado.
  - Si no es correcto → envía una pista (“mayor” o “menor”).

Hilos: Compartiendo Objetos

Piensa qué modificación habrá que hacer si tras comprobar que no es el número, el servidor “da pistas” a los clientes sobre si el número que tienen que adivinar es mayor o mejor que el número que acaban de proponer.

Y el cliente recibe la pista y vuelve a intentarlo.

Sockets UDP

11. Gestión de Sockets UDP (1)

- Los sockets UDP son más simples y eficientes que los TCP pero no está garantizada la entrega de paquetes.
- **No es necesario establecer una conexión** entre cliente y servidor como en el caso de TCP, por ello cada vez que se envíen datagramas, el emisor debe indicar explícitamente la dirección IP y el puerto del destino para cada paquete, y el receptor debe extraer esta información del paquete que recibe.
- El paquete java.net proporciona las siguientes clases para implementar sockets UDP:
  - DatagramPacket: crea instancias de los paquetes Datagrama que se van a recibir y de los que van a ser enviados.
  - DatagramSocket: da soporte a sockets para el envío y recepción de datagramas UDP.

12. Clase DatagramPacket (1) CONSTRUCTORES

- El paquete del datagrama está formado por los siguientes

CADENA DE BYTES CONTENIENDO EL MENSAJE	LONGITUD DEL MENSAJE	DIRECCIÓN IP DESTINO	Nº DE PUERTO DESTINO

CONSTRUCTOR	MISIÓN
DatagramPacket(byte[] buf, int length)	Constructor para datagramas recibidos. Se especifica la cadena de bytes en la que alojar el mensaje (*buf*) y la longitud (*length*) de la misma
DatagramPacket(byte[] buf, int offset, int length)	Constructor para datagramas recibidos. Se especifica la cadena de bytes en la que alojar el mensaje, la longitud de la misma y el offset (*offset*) dentro de la cadena
DatagramPacket(byte[] buf, int length, InetAddress addrss, int port)	Constructor para el envío de datagramas. Se especifica la cadena de bytes a enviar (*buf*), la longitud (*length*), el número de puerto de destino (*port*) y el el host especificado en la dirección *addrss*
DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)	Igual que el anterior pero se especifica un offset dentro de la cadena de bytes

12. Clase DatagramPacket (2) MÉTODOS

MÉTODOS	MISIÓN
InetAddress getAddress ()	Devuelve la dirección IP del host al cual se le envía el datagrama o del que el datagrama se recibió
byte[] getData()	Devuelve el mensaje contenido en el datagrama tanto recibido como enviado
int getLength()	Devuelve la longitud de los datos a enviar o a recibir
int getPort()	Devuelve el número de puerto de la máquina remota a la que se le va a enviar el datagrama o del que se recibió el datagrama
setAddress (InetAddress addr)	Establece la dirección IP de la máquina a la que se envia el datagrama
setData (byte [buf])	Establece el búfer de datos para este paquete
setLength (int length)	Ajusta la longitud de este paquete
setPort (int Port)	Establece el número de puerto del host remoto al que este datagrama se envía

13. Clase DatagramSocket (1) CONSTRUCTORES

CONSTRUCTOR	MISIÓN
DatagramSocket ()	Construye un socket para datagramas, el sistema elige un puerto de los que están libres
DatagramSocket (int port)	Construye un socket para datagramas y lo conecta al puerto local especificado
DatagramSocket (int port, InetAddress ip)	Permite especificar, además del puerto, la dirección local a la que se va a asociar el socket

14. Clase DatagramSocket (2) MÉTODOS

MÉTODOS	MISIÓN
receive (DatagramPacket paquete)	Recibe un **DatagramPacket** del socket, y llena *paquete* con los datos que recibe (mensaje, longitud y origen). Puede lanzar *IOException*
send (DatagramPacket paquete)	Envía un **DatagramPacket** a través del socket. El argumento *paquete* contiene el mensaje y su destino. Puede lanzar *IOException*
close ()	Se encarga de cerrar el socket
int getLocalPort ()	Devuelve el número de puerto en el host local al que está enlazado el socket, -1 si el socket está cerrado y 0 si no está enlazado a ningún puerto
int getPort()	Devuelve el número de puerto al que está conectado el socket, -1 si no está conectado
connect(InetAddress address, int port)	Conecta el socket a un puerto remoto y una dirección IP concretos, el socket solo podrá enviar y recibir mensajes desde esa dirección
setSoTimeout(int timeout)	Permite establecer un tiempo de espera límite. Entonces el método *receive*() se bloquea durante el tiempo fijado. Si no se reciben datos en el tiempo fijado se lanza la excepción *InterruptedIOException*

15. Gestión de sockets UDP(1)

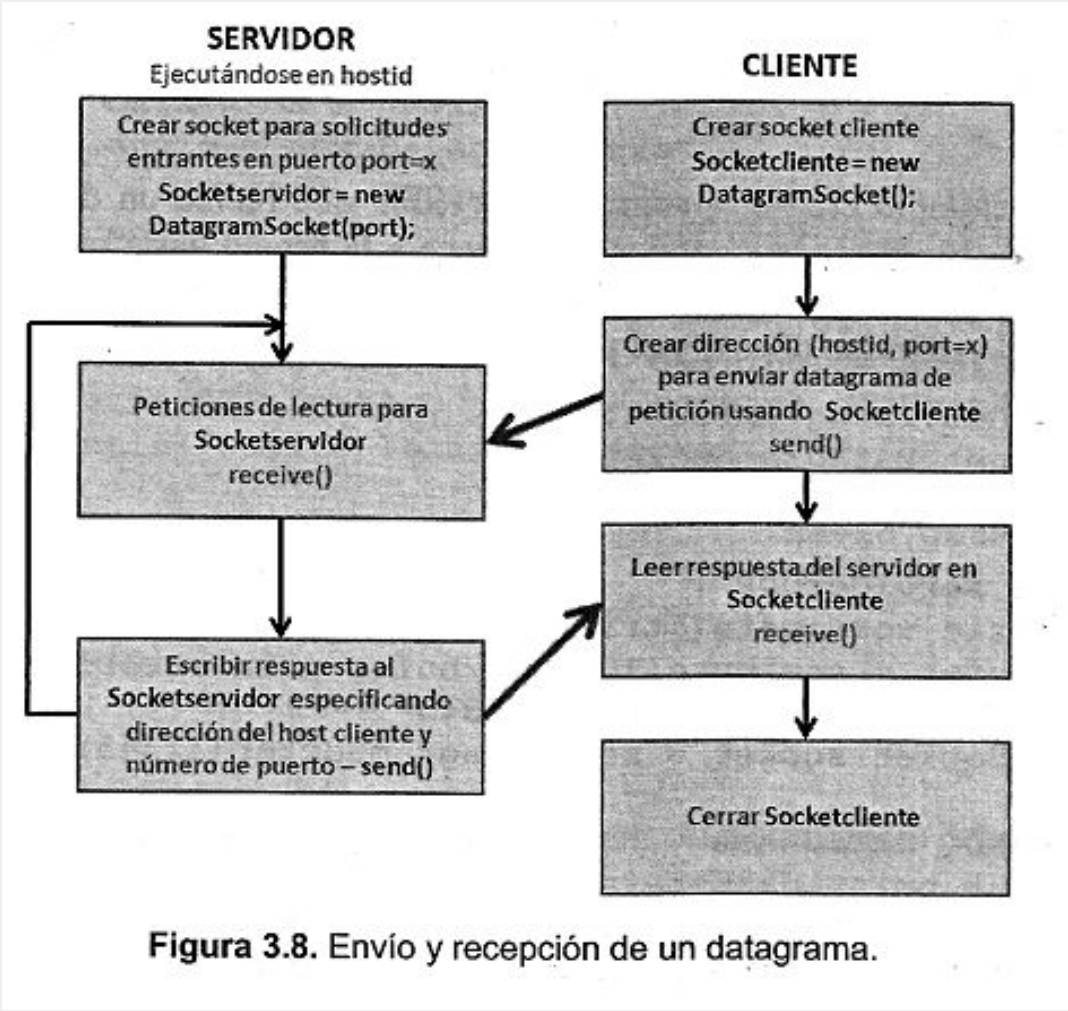


Figura 3.8. Envío y recepción de un datagrama.

15. Gestión de sockets UDP(2)

- El **servidor crea un socket** asociado a un puerto local para escuchar peticiones de clientes. Permanece a la espera de recibir peticiones.
- El **cliente creará un socket** para comunicarse con el servidor. Para enviar datagramas necesita conocer su IP y el puerto por el que escucha. Utilizará el método **send()** del socket para enviar la petición en forma de datagrama.
- El servidor recibe las peticiones mediante el método **receive()** del socket. En el datagrama va incluido además del mensaje, el puerto y la IP del cliente emisor de la petición; lo que le permite al servidor conocer la dirección del emisor del datagrama. Utilizando el método **send()** del socket puede enviar la respuesta al cliente emisor.
- El cliente recibe la respuesta del servidor mediante el método **receive()** del socket.
- El servidor permanece a la espera de recibir más peticiones.

Ver ejemplo de uso:: Ejemplo5

Realizar el boletín de Ejercicios 2

FIN