Protocolo TCP - Sockets Streams

Programación de servicios y procesos

Contenidos:

1) Clases y librerías usadas para la comunicación en red
2) Sockets Stream

Clases y librerías usadas para la comunicación en red

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

- La clase **URL**, Uniform Resource Locator (Localizador Uniforme de Recursos). Representa un puntero a un recurso en la Web.

- La clase **URLConnection**, que admite operaciones más complejas en las URL.

- Las 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.

- Las clases **DatagramSocket**, **MulticastSocket** y **DatagramPacket** para dar soporte a la comunicación vía datagramas UDP.

- La clase **InetAddress**, es una abstracción que representa las direcciones de Internet (IP).

¿Qué son los Sockets?

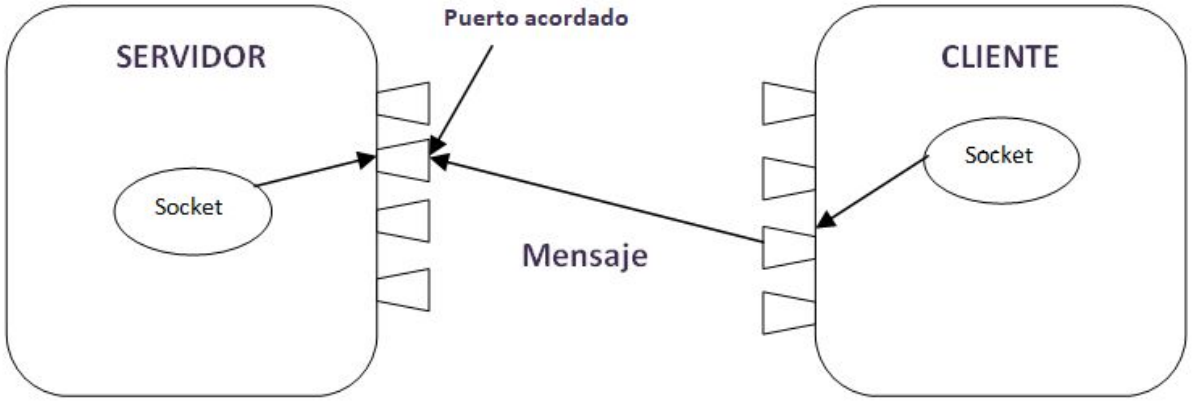
Los protocolos TCP y UDP utilizan la abstracción de sockets para proporcionar los puntos extremos de la comunicación entre aplicaciones o procesos.

La comunicación entre procesos consiste en la transmisión de un mensaje entre un conector de un proceso y un conector de otro proceso, a este conector es a lo que llamamos socket.

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.

El proceso cliente debe conocer el puerto y la IP del proceso servidor.



Funcionamiento general de un Socket

Normalmente en una aplicación cliente-servidor, el programa servidor se ejecuta en una máquina específica y tiene un socket que está unido a un número de puerto específico.

El programa cliente conoce la máquina en la que se ejecuta el servidor y el número de puerto por el que escucha las peticiones. El cliente realiza una petición de conexión ante dicho puerto y él mismo utilizará un puerto local asignado por su sistema.

Si todo va bien, el servidor acepta la conexión y una vez aceptada, el servidor obtiene un nuevo socket sobre un puerto diferente. Esto se debe a que por un lado debe seguir atendiendo las peticiones de conexión mediante el socket original y por otro debe atender las necesidades del cliente que se conectó.

En el cliente, si se acepta la conexión, se crea un socket y el cliente puede utilizarlo para comunicarse con el servidor. El cliente y el servidor pueden ahora comunicarse escribiendo y leyendo por sus respectivos sockets.

Sockets Stream

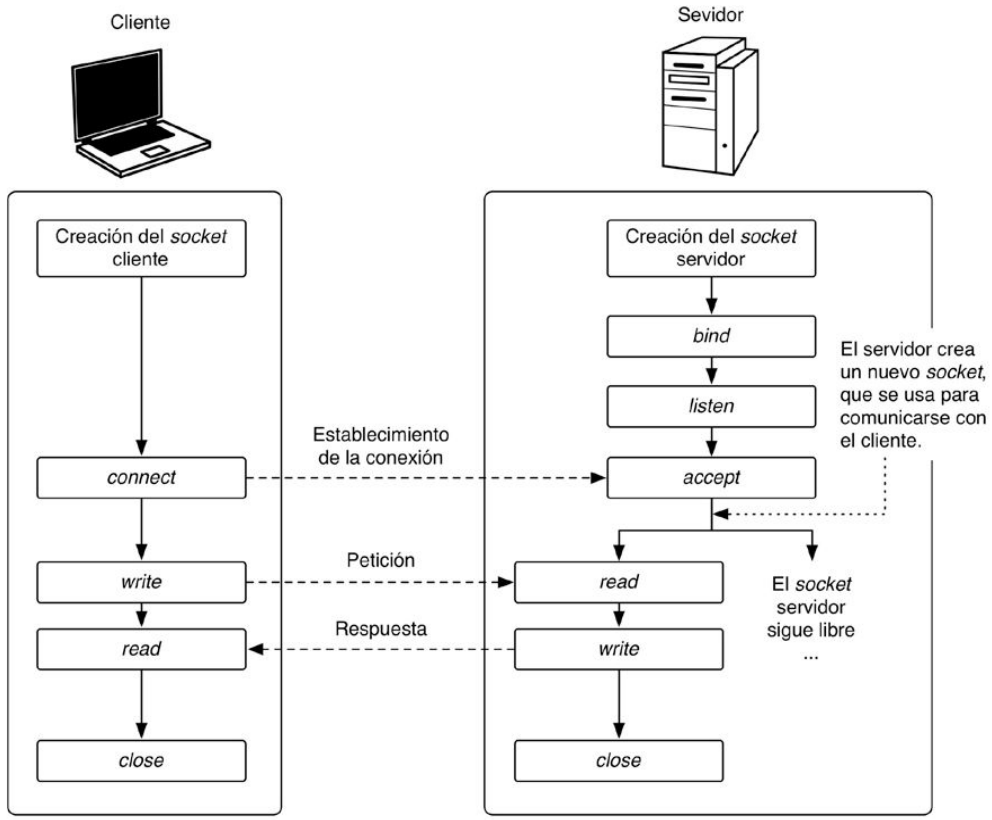
- Son **orientados a conexión**.
- Cuando operan sobre **IP, emplean TCP**.
- Un socket stream se utiliza para comunicarse siempre con el mismo receptor, manteniendo el canal de comunicación abierto entre ambas partes hasta que se termina la conexión.
- Una parte ejerce la función de **proceso cliente** y otra de **proceso servidor**.

Proceso cliente:

1) Creación del socket.
2) Conexión del socket (connect)
3) Envío y recepción de mensajes.
4) Cierre de la conexión (close)

Proceso servidor:

1) Creación del socket.
2) Asignación de dirección y puerto (bind).
3) Escucha (listen).
4) Aceptación de conexiones (accept). Esta operación implica la creación de un nuevo socket, que se usa para comunicarse con el cliente que se ha conectado.
5) Envío y recepción de mensajes.
6) Cierre de la conexión (close).



Sockets Stream

La comunicación entre las aplicaciones se realiza por medio del protocolo TCP. Por tanto es una conexión fiable en la que se garantiza la entrega de los paquetes de datos y el orden en que fueron enviados. TCP utiliza un sistema de acuse de recibo de mensajes de tal forma que si el emisor no recibe dicho acuse dentro de un tiempo determinado, vuelve a transmitir el mensaje.

Los procesos que se van a comunicar deben establecer antes una conexión mediante un stream (secuencia ordenada de bytes).

Los sockets TCP se utilizan en la gran mayoría de las aplicaciones IP. Algunos servicios con sus número de puerto reservado son: FTP (puerto 21), Telnet (23), HTTP (80), SMTP (25).

Este tipo de sockets utilizan stream sockets que tienen asociadas las clases Socket para el cliente y ServerSocket para el servidor.

La clase ServerSocket

Constructors
ServerSocket() Creates an unbound server socket.
ServerSocket(int port) Creates a server socket, bound to the specified port.
ServerSocket(int port, int backlog) Creates a server socket and binds it to the specified local port number, with the specified backlog. (Número máximo de peticiones que se pueden mantener en cola)
ServerSocket(int port, int backlog, **InetAddress** bindAddr) Create a server with the specified port, listen backlog, and local IP address to bind to.

La clase ServerSocket

Methods	
Socket (A través del socket devuelto, se establecerá la conexión con el cliente)	**accept**() Listens for a connection to be made to this socket and accepts it.
void	**close**() Closes this socket.
int	**getLocalPort**() Returns the port number on which this socket is listening.

La clase Socket

Constructors
Socket() Creates an unconnected socket, with the system-default type of SocketImpl.
Socket(InetAddress address, int port) Creates a stream socket and connects it to the specified port number at the specified IP address.
Socket(InetAddress address, int port, InetAddress localAddr, int localPort) Creates a socket and connects it to the specified remote address on the specified remote port.
Socket(String host, int port) Creates a stream socket and connects it to the specified port number on the named host.

La clase Socket

Methods	
void	close() Closes this socket.
InetAddress	getInetAddress() Returns the address to which the socket is connected.
InputStream	getInputStream() Returns an input stream for this socket.
int	getLocalPort() Returns the local port number to which this socket is bound.
OutputStream	getOutputStream() Returns an output stream for this socket.
int	getPort() Returns the remote port number to which this socket is connected.

Socket Stream

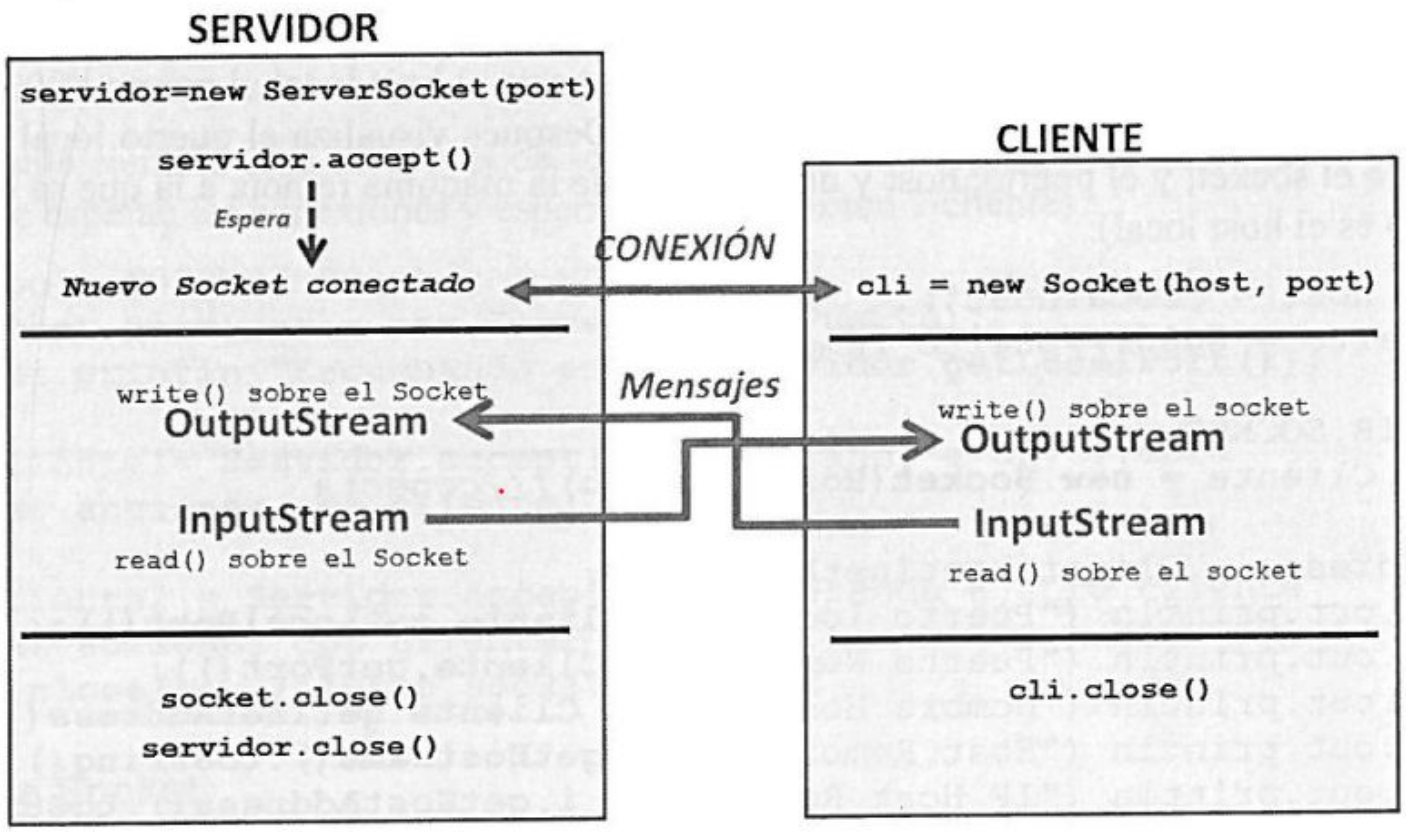


Figura 3.5. Modelo de Socket TCP.

Ejemplo comunicación basado en sockets TCP - Servidor

```
6  public class TCPejemplo1Servidor {
7    public static void main(String[] arg) throws IOException {
8      // ASIGNO UN PUERTO A MI APLICACION E INSTANCIO UN SERVERSOCKET CON EL PUERT0
9      // ASÍ COMO UN SOCKET PARA CUANDO SE CONECTA UN CLIENTE
10     int numeroPuerto = 6000;// Puerto
11     ServerSocket servidor = new ServerSocket(numeroPuerto);
12     Socket clienteConectado = null;
13     System.out.println("Esperando al cliente.....");
14     clienteConectado = servidor.accept();
15
16     // CREO FLUJO DE ENTRADA DEL CLIENTE
17     InputStream entrada = null;
18     entrada = clienteConectado.getInputStream();
19     DataInputStream flujoEntrada = new DataInputStream(entrada);
20
21     // EL CLIENTE ME ENVIA UN MENSAJE
22     System.out.println("Recibiendo del CLIENTE: \n\t" +
23                        flujoEntrada.readUTF());
```

Crear el servidor asociado a un puerto

Esperamos la conexión del cliente y la aceptamos. Esto crea un objeto de tipo socket

Obtenemos el flujo de entrada

Usamos DataInputStream para recuperar los mensajes que el cliente escriba en el socket

Pintamos por consola lo que envía el cliente

La clase **DataInputStream** permite la lectura de líneas de texto y tipos primitivos Java. Algunos de sus métodos son: readInt(), readDouble(), readLine(), readUTF(), etc.

Ejemplo comunicación basado en sockets TCP - Servidor

```
25      // CREO FLUJO DE SALIDA AL CLIENTE
26      OutputStream salida = null;
27      salida = clienteConectado.getOutputStream();
28      DataOutputStream flujoSalida = new DataOutputStream(salida);
29
30      // ENVIO UN SALUDO AL CLIENTE
31      flujoSalida.writeUTF("Saludos al cliente del servidor");
32
33      // CERRAR STREAMS Y SOCKETS
34      entrada.close();
35      flujoEntrada.close();
36      salida.close();
37      flujoSalida.close();
38      clienteConectado.close();
39      servidor.close();
40    }// main
41 }// fin
```

Establecemos el flujo de salida

Usamos DataOutputStream para escribir los mensajes envíamos al cliente

Escribimos el mensaje

El orden de cierre es relevante. Primero se deben cerrar los streams relacionados con un socket antes que el propio socket

La clase **DataOutputStream** dispone de métodos para escribir tipos primitivos Java. Algunos de sus métodos son: writeInt(), writeDouble(), writeBytesLine(), writeUTF(), etc.

Ejemplo comunicación basado en sockets TCP - Cliente

```
6  public class TCPejemplo1Cliente  {
7    public static void main(String[] args) throws Exception {
8      String Host = "192.168.56.1"; //"localhost";
9      int Puerto = 6000;//puerto remoto
10
11     System.out.println("PROGRAMA CLIENTE INICIADO....");
12     Socket Cliente = new Socket(Host, Puerto);
13
14     // CREO FLUJO DE SALIDA AL SERVIDOR
15     DataOutputStream flujoSalida = new
16             DataOutputStream(Cliente.getOutputStream());
17
18     // ENVIO UN SALUDO AL SERVIDOR
19     flujoSalida.writeUTF("Saludos al SERVIDOR DESDE EL CLIENTE");
20
```

Asignamos la IP (por IP o por nombre de host) y el Hosts al que nos queremos conectar

Abrimos la conexión

Escribimos el mensaje

Ejemplo comunicación basado en sockets TCP - Cliente

```
21      // CREO FLUJO DE ENTRADA AL SERVIDOR
22      DataInputStream flujoEntrada = new
23              DataInputStream(Cliente.getInputStream());
24
25      // EL SERVIDOR ME ENVIA UN MENSAJE
26      System.out.println("Recibiendo del SERVIDOR: \n\t" +
27                  flujoEntrada.readUTF());
28
29      // CERRAR STREAMS Y SOCKETS
30      flujoEntrada.close();
31      flujoSalida.close();
32      Cliente.close();
33    }// main
34  }//
```

Recibimos el mensaje del servidor

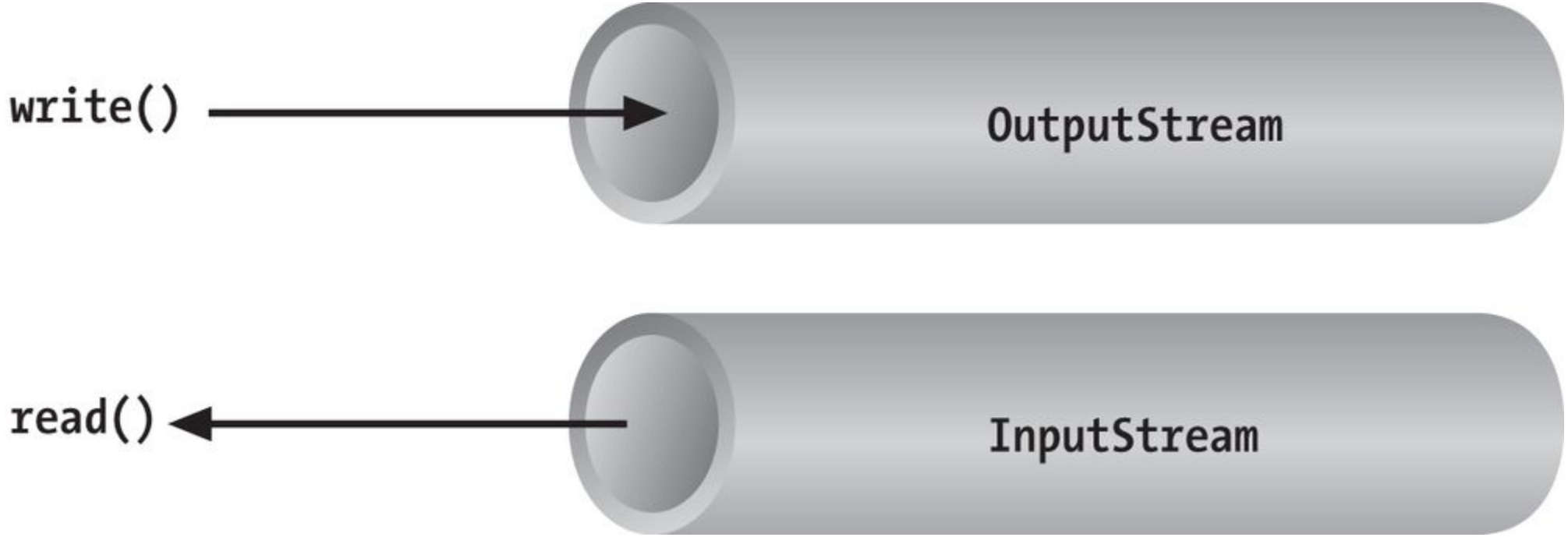
Lo pintamos por consola

Cerramos el socket

Estos proyectos están en los recursos del tema, proyectos: UT4_Ejem1_SocketTCPServer y UT4_Ejem1_SocketTCPClient.

Streams para E/S en los sockets

Si vemos ejemplos en Internet o en tutoriales, podemos observar que hay dos formas mayoritarias de enviar y recibir la información a través de los streams que proporciona un socket.



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.

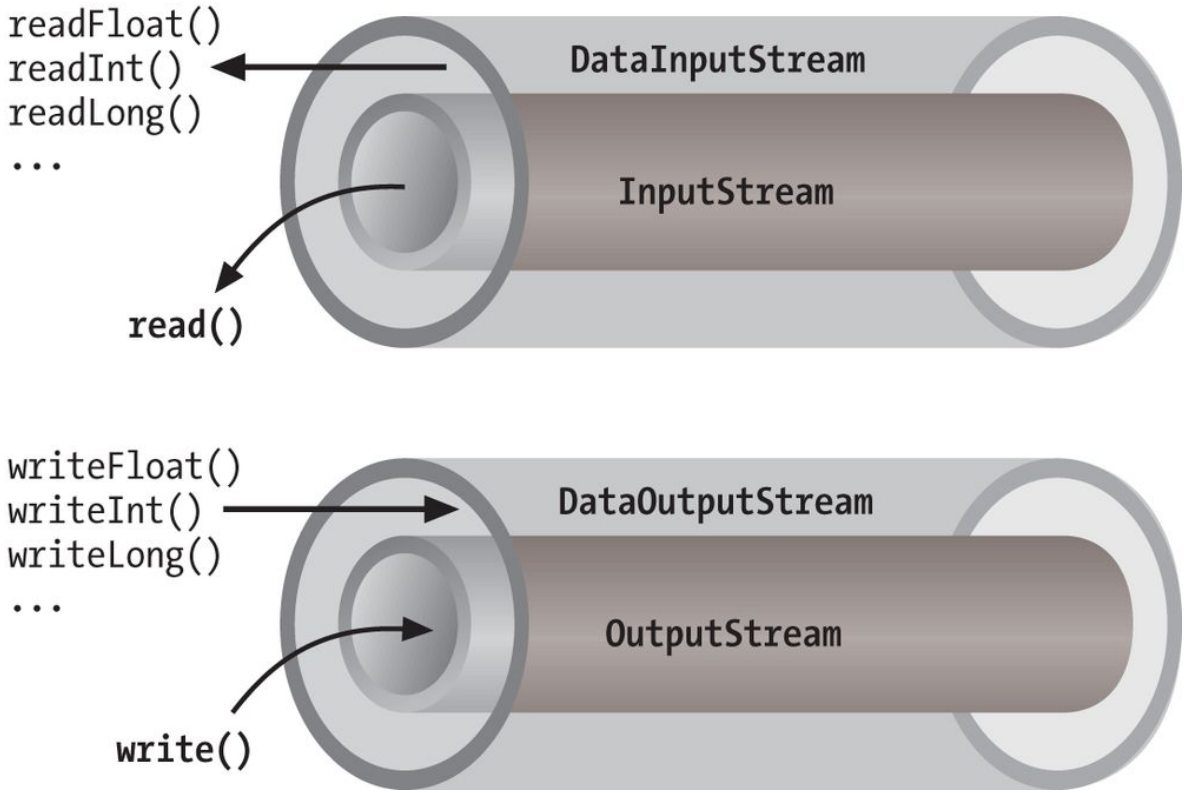
Streams para E/S en los sockets

En los protocolos de comunicaciones, más del 90% de la información que se intercambia, a nivel de protocolo, es en formato texto. 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.

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.

Los métodos **readUTF()** y **writeUTF()** de DataInputStream y DataOutputStream leen y escriben un String de caracteres Unicode usando la codificación UTF-8.



Streams para E/S en los sockets

Elige un método y usa siempre el mismo

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. Además de usar los métodos complementarios para la lectura y escritura, por ejemplo readInt / writeInt.