PROGRAMACIÓN DE SERVICIOS Y PROCESOS

Programación de comunicaciones en red

Programación de comunicaciones en red

Protocolos de comunicaciones. Puertos.

Un **protocolo de comunicaciones** es un conjunto de normas que usan los ordenadores para gestionar la comunicación en el intercambio de información. Dos ordenadores con distinta configuración pero el mismo protocolo de comunicación se podrán comunicar sin problema.

Hoy en día, el protocolo más popular es **TCP/IP** cuyas siglas vienen de Transfer Control Protocol / Internet Protocol. Este protocolo se basa en un modelo por capas que permite que se pueda utilizar en cualquier equipo independientemente del sistema operativo utilizado. El término capa se refiere al hecho de que la información viaja por la red atravesando diferentes niveles de protocolos ya que cada capa procesa sucesivamente la información y la envía a la capa siguiente.

Programación de comunicaciones en red

Protocolos de comunicaciones. Puertos.

Las capas de este modelo son las siguientes:

- Capa de **acceso a la red**. Se encarga de ofrecer la capacidad de acceder a cualquier red física. Contiene las especificaciones necesarias para la transmisión de datos por una red física.
- Capa de **Internet**. Se encarga de permitir que los nodos incluyan paquetes en cualquier red y viajen de forma independiente a su destino. Estos paquetes pueden llegar incluso en un orden distinto a como se enviaron. Esta capa define un formato de paquetes y un protocolo oficial denominado IP.
- Capa de **transporte**. En esta capa se encuentran dos protocolos:
  - *TCP (protocolo de control de la transmisión)*
  - *UDP (protocolo de datagrama de usuario)*

La principal diferencia entre ambos es que el primero es orientado a la conexión mientras que el segundo es un protocolo sin conexión y no confiable por lo que su uso se limita a aplicaciones que no necesitan control de flujo.

- Capa de **aplicación**. Contiene los protocolos de más alto nivel como son SMTP, FTP, etc. El software de esta capa se comunica mediante los protocolos de la capa de transporte TCP o UDP.

Programación de comunicaciones en red

Protocolos de comunicaciones. Puertos.

Los ordenadores tienen una o varias conexiones físicas a la red a través de las que se reciben los datos dirigidos a la máquina. Estos datos contienen información de direccionamiento para indicar a qué máquina y aplicación van dirigidos. Esta información está formada por 32 bits para indicar la máquina y 16 para indicar el puerto.

Una **dirección IP** nos permite identificar de manera única un equipo en la red, pero se necesita alguna manera de identificar la aplicación a la que se dirigen los datos y para esto se utilizan los **puertos**. De esta manera conseguimos que cuando un equipo recibe información la remita directamente a la aplicación a la que va destinada.

Un **puerto** es, por tanto, un número de 16 bits utilizado para identificar a que protocolo o proceso van dirigidos los datos.

Programación de comunicaciones en red

Protocolos de comunicaciones. Puertos.

Existen multitud de puertos por lo que la Autoridad de Números Asignados de Internet (IANA) desarrollo una aplicación estándar para apoyar con las configuraciones de red. De esta manera podemos distinguir los siguientes tipos de puertos:

- Puertos conocidos o **reservados**: son aquellos que van del 0 al 1023 y están reservados para procesos del sistema o para programas que ejecutan usuarios privilegiados.
- Puertos **registrados**: son aquellos que van del 1024 al 49151.
- Puertos **dinámicos** o **privados**: aquellos que van del 49152 al 65535.

Programación de comunicaciones en red

Protocolos de comunicaciones. Puertos.

Algunos de los puertos más conocidos son:

Puerto	Servicio o aplicación
21	FTP
23	Telnet
25	SMTP
53	Sistema de nombre de dominio
63	Whois
70	Gopher
79	Finger
80	HTTP
110	POP3
119	NNTP

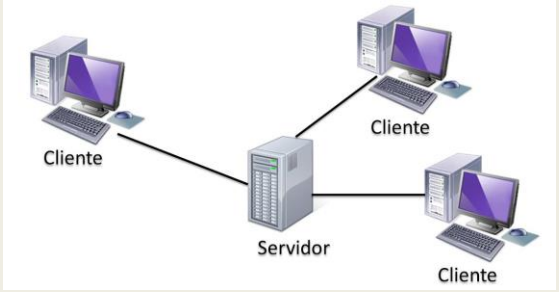
Anexo:Puertos de red - Wikipedia, la enciclopedia libre

Programación de comunicaciones en red

Comunicación entre aplicaciones.

Las aplicaciones utilizan el **modelo cliente servidor** para comunicarse. Se trata de un sistema distribuido donde el software está dividido en dos partes las cuales se pueden ejecutar en el mismo o diferente sistema:

- **Servidor**: aplicación que ofrece servicios a clientes. Cabe destacar que un mismo servidor puede dar servicio a varios clientes y que en un mismo sistema puede haber varios servidores.
- **Cliente**: aplicación que solicita servicios al servidor. Este parte es la que interactúa con el usuario de la aplicación. Cabe destacar que un mismo cliente puede comunicarse simultáneamente con varios servidores.



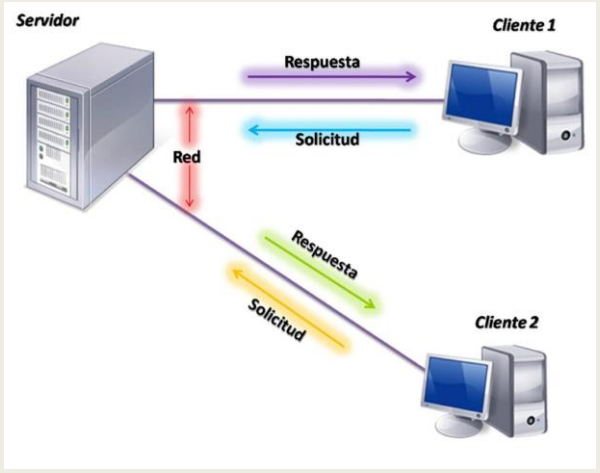
Gracias a este modelo se consigue una separación clara de responsabilidades de manera que en cada una de las partes se realizan determinadas funciones.

Programación de comunicaciones en red

Comunicación entre aplicaciones.

Aunque el flujo de la comunicación puede ir en ambos sentidos, el funcionamiento más habitual para el establecimiento de la comunicación entre cliente y servidor es:

1. Un usuario invoca la parte cliente de una aplicación.
2. El cliente construye la solicitud del servicio demandado por el usuario y la envía al servidor de la aplicación.
3. El servidor recibe la solicitud, realiza el servicio solicitado y devuelve los resultados.



Programación de comunicaciones en red

Comunicación entre aplicaciones.

Al cliente se le suele denominar parte activa y al servidor parte pasiva puesto que está esperando a que los clientes se comuniquen con él.

Entre las **principales utilidades** del uso de aplicaciones cliente servidor cabría destacar:

- **Facilidad de mantenimiento** debido a que todo el mantenimiento se realiza en el servidor.
- Pueden ser desarrolladas utilizando **distintos lenguajes** de programación.
- Ligereza debido a que la **carga de trabajo** y **consumo de recursos** se realiza en el lado del **servidor**.

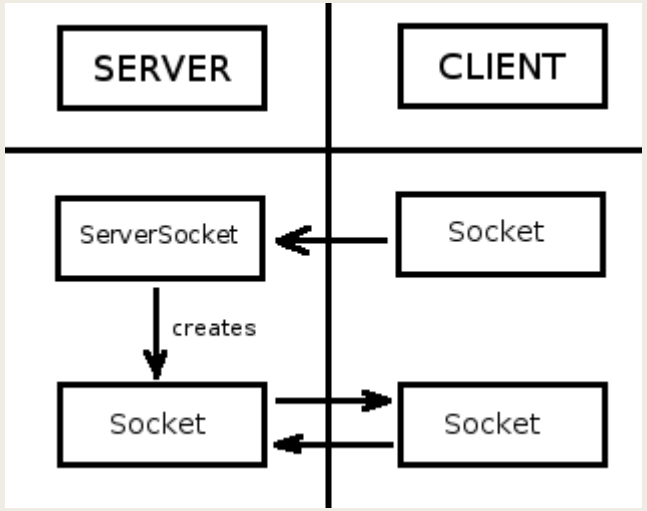
Programación de comunicaciones en red

Sockets. Tipos y características

Un **socket** es un mecanismo o canal de comunicación entre procesos que pertenecen a la misma o diferente máquina. Es, por tanto, una forma de que dos programas se comuniquen entre sí para compartir información.

Fueron popularizados por Berckley Software Distribution y se identifican por una dirección IP, un protocolo y un número de puerto.

Los sockets deben ser capaces de utilizar el protocolo de streams TCP y el protocolo de datagramas UDP.

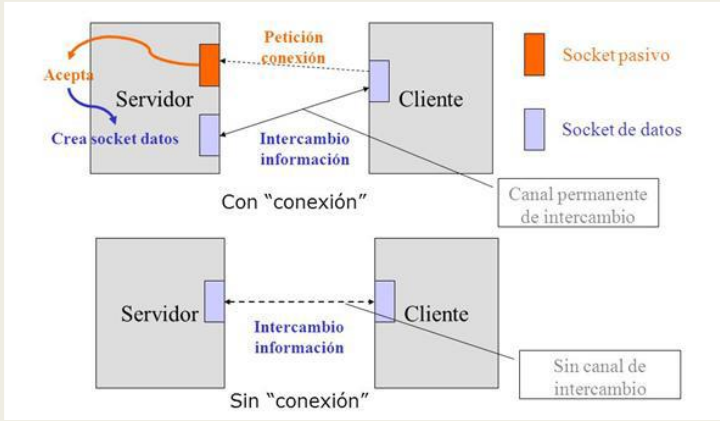


Programación de comunicaciones en red

Sockets. Tipos y características

Podemos distinguir dos tipos de sockets:

- **Stream**. Se trata de un socket basado en el protocolo **TCP** que es **orientado a conexión** por lo que es **fiable** ya que garantiza el orden de entrega de los mensajes.
- **Datagrama**. Se trata de un socket basado en el protocolo **UDP** que es **no orientado a conexión** y por tanto no asegura el orden de entrega de los mensajes (no fiable) ni la llegada de estos. El protocolo UDP es **más eficiente** pero **menos fiable** que el TCP.

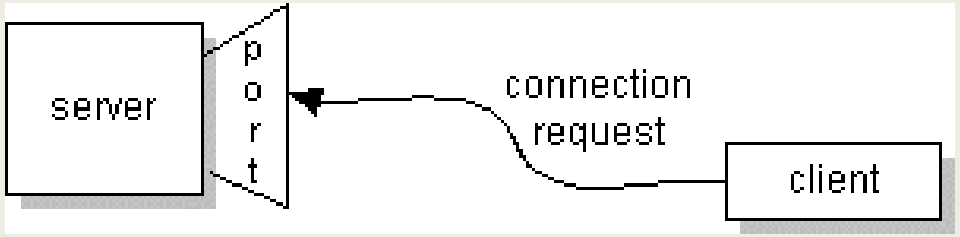


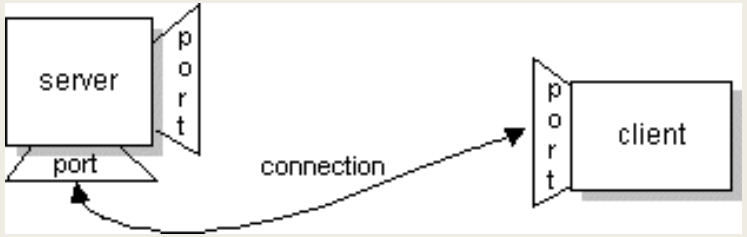
Programación de comunicaciones en red

Sockets. Tipos y características

El funcionamiento básico sería el siguiente:

- El servidor se ejecuta sobre un equipo con un socket que responde en un puerto específico. Este espera hasta que algún cliente haga una petición.
- Por su parte el cliente, que conoce el nombre del equipo con el que se quiere comunicar y el número de puerto en el que está conectado, realiza una petición de conexión.
- El servidor acepta dicha petición y obtiene un nuevo socket sobre un puerto distinto.

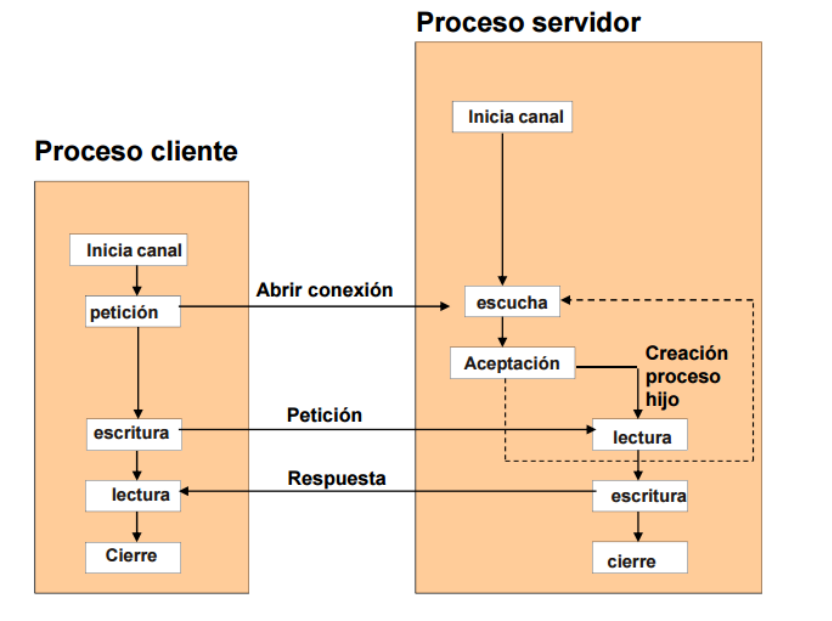




Programación de comunicaciones en red

Programación de aplicaciones cliente y servidor en red

El **paradigma cliente / servidor** es el más utilizado para la programación de aplicaciones de red. Este modelo es usado para describir la interacción entre procesos que se ejecutan de manera simultánea. Se basa en el hecho de que si dos aplicaciones necesitan comunicarse, una comienza la ejecución y espera indefinidamente hasta que la otra responde y a continuación continua con el proceso.

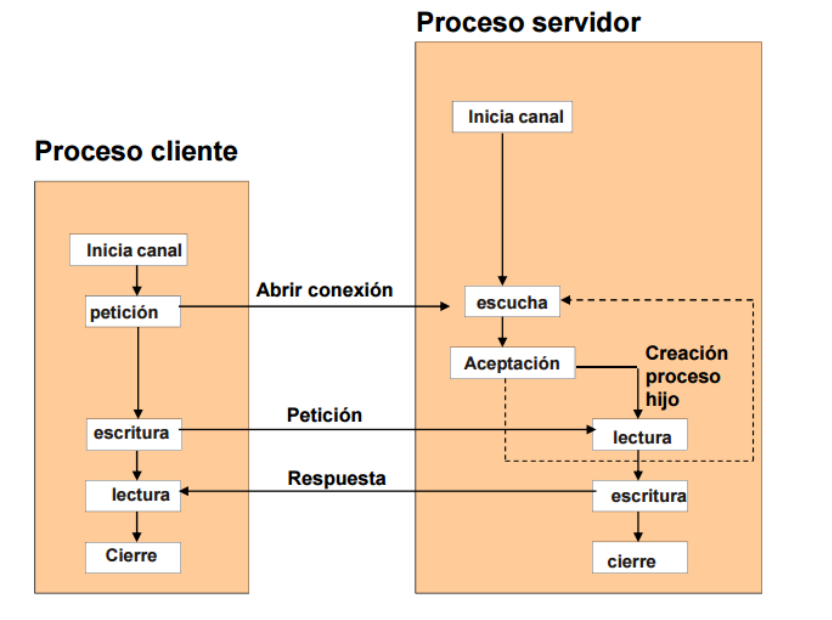


Programación de comunicaciones en red

Programación de aplicaciones cliente y servidor en red

En este paradigma, como vimos en puntos anteriores de la unidad, existen dos componentes:

- **Servidor**: aplicación que ofrece servicios a clientes. Normalmente un mismo servidor da servicio a varios clientes.
- **Cliente**: aplicación que solicita servicios al servidor.



Programación de comunicaciones en red

Programación de aplicaciones cliente y servidor en red

Centrándonos en la programación de aplicaciones, un **servidor** se ejecuta en una máquina específica y tiene un **socket** asociado a un número de puerto específico, donde espera a que un cliente le envíe una **petición**.

Por la parte de los **clientes**, como estos saben el nombre de la máquina sobre la que se está ejecutando el servidor y el número de puerto, solicita **conexión** con el servidor mediante esos parámetros.

Si todo va bien, el servidor **acepta la conexión** y **crea un nuevo socket en un puerto diferente**.

De esta manera el socket inicial se **mantiene a la escucha** de nuevas peticiones de clientes y este nuevo socket permite la conexión entre el servidor y el cliente en cuestión.

Programación de comunicaciones en red

Programación de aplicaciones cliente y servidor en red

En el siguiente punto se estudiará la **programación de sockets** en el lenguaje de programación **Java** para lo que se hace uso del paquete **java.net** que contiene las clases necesarias para generar aplicaciones para redes.

Además hay que tener en cuenta que la comunicación entre cliente y servidor se realiza mediante el **intercambio de flujos de datos** por lo que otro paquete que se utilizará con frecuencia será **java.io** que ya se estudió el curso anterior.

Programación de comunicaciones en red

Programación de sockets en Java

El ciclo de vida de un socket estaría compuesto por las siguientes etapas:

- **Creación** y **apertura** del socket
- **Lectura** y **escritura** de información, es decir, recepción y envío de datos por el socket.
- **Cierre** del socket

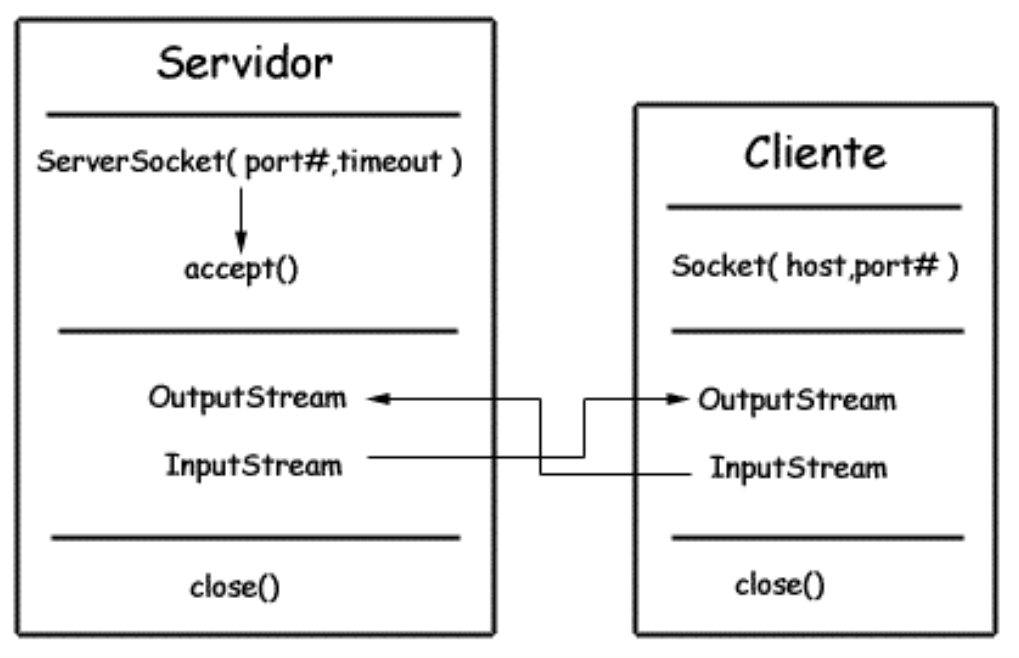
Para la creación de sockets, el lenguaje de programación Java proporciona en el paquete java.net las siguientes clases:

- ***java.net.socket****: proporciona métodos para la entrada y salida a través de streams.*
- ***java.net.serversocket****: se utiliza en las aplicaciones servidoras para escuchar las peticiones que realizan los clientes conectados.*

Programación de comunicaciones en red

Programación de sockets en Java

Para la apertura de sockets tenemos que diferenciar entre si estamos programando en el lado del **cliente** o del **servidor**. Veamos antes de nada una imagen que ilustra el proceso:



Programación de comunicaciones en red

Programación de sockets en Java

En el caso del **cliente**, un socket se abre de la siguiente manera:

```
Socket miCliente = new Socket("maquina",numeroPuerto);
```

Donde maquina es el nombre de la máquina donde queremos abrir la conexión.

En el caso del **servidor**, un socket se abre de la siguiente manera:

```
Socket miServicio = new ServerSocket(numeroPuerto);
```

Donde numeroPuerto es el puerto del servidor.

Programación de comunicaciones en red

Programación de sockets en Java

A continuación tenemos que crear **Streams** de **entrada** y de **salida** en ambas aplicaciones de manera que sea posible el envío y recepción de datos por el socket. Para esto se utilizan las clases:

- **DataInputStream**: permite la lectura de texto y tipos de datos primitivos de Java.
- **DataOutputStream**: permite la escritura de cualquier tipo de datos primitivo de Java.

Y para obtener los streams a partir del socket utilizamos los métodos:

- **getInputStream()**: devuelve un objeto de tipo InputStream.
- **getOutputStream()**: devuelve un objeto de tipo OutputStream.

Programación de comunicaciones en red

Programación de sockets en Java

Para crear un Stream de entrada en la parte cliente de la aplicación:

```
DataInputStream entrada= new DataInputStream(miCliente.getInputStream());
```

Para crear un Stream de entrada en la parte servidor de la aplicación:

```
DataInputStream entrada= new DataInputStream(miServicio.getInputStream());
```

Para crear un Stream de salida en la parte cliente de la aplicación:

```
DataOutputStream salida= new DataOutputStream(miCliente.getOutputStream());
```

Para crear un Streams de salida en la parte servidor de la aplicación:

```
DataOutputStream salida= new DataOutputStream(miServicio.getOutputStream());
```

Programación de comunicaciones en red

Programación de sockets en Java

Como se observa, los flujos anteriores son orientados al byte. Si la comunicación entre el cliente y el servidor se realiza mediante cadenas de texto habría que convertir en objetos de tipo **BufferedReader** para la **entrada** y objetos de tipo **PrintWriter** para la **salida** de texto.

Siempre es necesario cerrar todos los canales de entrada y de salida que se han abierto durante la ejecución de la aplicación, para esto realizamos lo siguiente en el lado del cliente:

```
entrada.close();
salida.close();
miCliente.close();
```

y en el servidor:

```
entrada.close();
salida.close();
miServicio.close();
```

Programación de comunicaciones en red

Ejemplo – Clase Conexion

```
public class Conexion
{
    private final int PUERTO = 1234; //Puerto para la conexi□n
    private final String HOST = "localhost"; //Host para la conexi□n
    protected String mensajeServidor; //Mensajes entrantes (recibidos) en el servidor
    protected ServerSocket ss; //Socket del servidor
    protected Socket cs; //Socket del cliente
    protected DataOutputStream salidaServidor, salidaCliente; //Flujo de datos de salida

    public Conexion(String tipo) throws IOException //Constructor
    {
        if(tipo.equalsIgnoreCase("servidor"))
        {
            ss = new ServerSocket(PUERTO);//Se crea el socket para el servidor en puerto 1234
            cs = new Socket(); //Socket para el cliente
        }
        else
        {
            cs = new Socket(HOST, PUERTO); //Socket para el cliente en localhost en puerto 1234
        }
    }
}
```

Programación de comunicaciones en red

Ejemplo – Clase Servidor

```
public class Servidor extends Conexion //Se hereda de Conexion para hacer uso de los sockets y demás
{
    public Servidor() throws IOException{super("servidor");} //Se usa el constructor para servidor de Conexion
    public void startServer()//M□todo para iniciar el servidor
    {
        try{
            System.out.println("Esperando..."); //Esperando conexión
            cs = ss.accept(); //Accept comienza el socket y espera una conexión desde un cliente
            System.out.println("Cliente en línea");
            //Se obtiene el flujo de salida del cliente para enviarle mensajes
            salidaCliente = new DataOutputStream(cs.getOutputStream());
            //Se le envía un mensaje al cliente usando su flujo de salida
            salidaCliente.writeUTF("Petición recibida y aceptada");
            //Se obtiene el flujo entrante desde el cliente
            BufferedReader entrada = new BufferedReader(new InputStreamReader(cs.getInputStream()));
            while((mensajeServidor = entrada.readLine()) != null){
             //Mientras haya mensajes desde el cliente se muestra por pantalla el mensaje recibido
                System.out.println(mensajeServidor);
            }
            System.out.println("Fin de la conexión");
            ss.close();//Se finaliza la conexión con el cliente
        }
        catch (Exception e){
            System.out.println(e.getMessage());
        }
    }
}
```

Programación de comunicaciones en red

Ejemplo – Clase Cliente

```
public class Cliente extends Conexion
{
    //Se usa el constructor para cliente de Conexion
    public Cliente() throws IOException{super("cliente");}
    public void startClient() //M□todo para iniciar el cliente
    {
        try{
            //Flujo de datos hacia el servidor
            salidaServidor = new DataOutputStream(cs.getOutputStream());
            //Se enviarán dos mensajes
            for (int i = 0; i < 2; i++)
            {
                //Se escribe en el servidor usando su flujo de datos
                salidaServidor.writeUTF("Este es el mensaje " + (i+1) + "\n");
            }
            cs.close();//Fin de la conexi□n
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
        }
        System.out.println("Fin de la conexión");
    }
}
```

Programación de comunicaciones en red

Ejemplo – Clase MainServidor

```
public class MainServidor
{
    public static void main(String[] args) throws IOException
    {
        Servidor serv = new Servidor(); //Se crea el servidor
        System.out.println("Iniciando servidor");
        serv.startServer(); //Se inicia el servidor
    }
}
```

Programación de comunicaciones en red

Ejemplo – Clase MainCliente

```
public class MainCliente
{
    public static void main(String[] args) throws IOException
    {
        Cliente cli = new Cliente(); //Se crea el cliente
        System.out.println("Iniciando cliente");
        cli.startClient(); //Se inicia el cliente
    }
}
```

Programación de comunicaciones en red

Ejercicio

Modificar el programa anterior para que el servidor permanezca esperando conexiones de clientes y no finalice su ejecución.