programación en red

introducción

Sin duda la red es el contexto de trabajo fundamental de java. Lo mejor de Java está creado para la red. El paquete **java.net** es el encargado de almacenar clases que permitan generar aplicaciones para redes. En él podremos encontrar clases orientadas a la programación de sockets y herramientas de trabajo con URLs.

También se utiliza mucho el paquete **java.io (**visto en el tema dedicado a la entrada y salida, página 93). Esto se debe a que la comunicación entre clientes y servidores se realiza intercambiando flujos de datos, por lo que las clases para controlar estos flujos son las mismas que las vistas en el tema citado.

sockets

Son la base de la programación en red. Se trata de el conjunto de una dirección de servidor y un número de puerto. Esto posibilita la comunicación entre un cliente y un servidor a través del puerto del socket. Para ello el servidor tiene que estar *escuchando* por ese puerto.

Para ello habrá al menos dos aplicaciones en ejecución: una en el servidor que es la que abre el socket a la escucha, y otra en el cliente que hace las peticiones en el socket. Normalmente la aplicación de servidor ejecuta varias instancias de sí misma para permitir la comunicación con varios clientes a la vez.

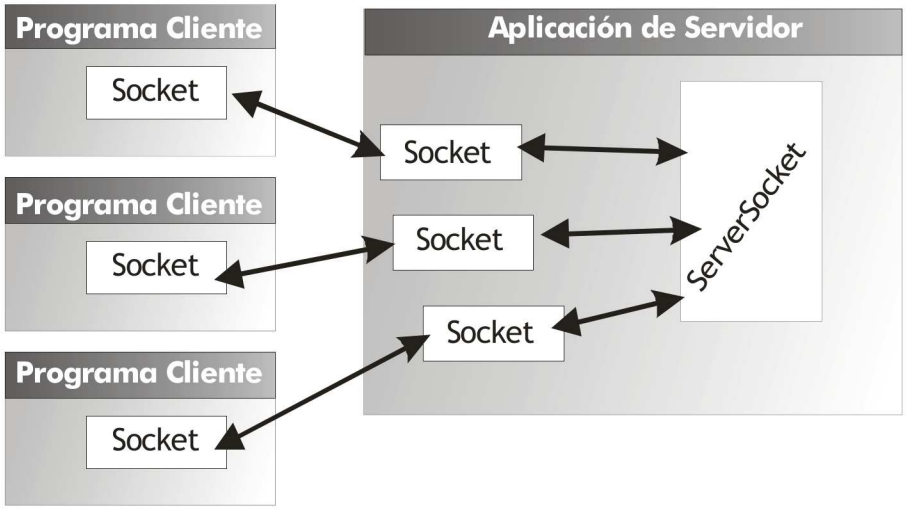

Ilustración 29, Esquema de la comunicación con sockets

clientes

Las aplicaciones clientes son las que se comunican con servidores mediante un socket. Se abre un puerto de comunicación en ordenador del cliente hacia un servidor cuya dirección ha de ser conocida.

La clase que permite esta comunicación es la clase java.net.**Socket**.

construcción de sockets

constructor	uso
Socket(String servidor, int puerto**)** throws IOException, UnknownHostException	Crea un nuevo socket hacia el servidor utilizando el puerto indicado
Socket(InetAdress servidor, int puerto**)** throws IOException	Como el anterior, sólo que el servidor se establece con un objeto **InetAddress**
Socket(InetAdress servidor, int puerto, InetAdress dirLocal, int puertoLocal**)** throws IOException	Crea un socket hacia el servidor y puerto indicados, pero la lectura la realiza la dirección local y puerto local establecidos.
Socket(String servidor, int puerto, InetAdress dirLocal, int puertoLocal**)** throws IOException, UnknownHostException	Crea un socket hacia el servidor y puerto indicados, pero la lectura la realiza la dirección local y puerto local establecidos.

Ejemplo:

```
try{
   Socket s=new Socket("time-a.mist.gov",13);
}
catch(UnknownHostException une){
   System.out.println("No se encuentra el servidor");
}
catch(IOException une){
   System.out.println("Error en la comunicación");
}
```

Es necesario capturar esas excepciones. La excepción **UnknownHostException** es una subclase de **IOException** (a veces se captura sólo esta última para simplificar el código ya que si sólo se incluye ésta, se captura también a la anterior; aunque suele ser más conveniente separar los tipos de error),

lectura y escritura por el socket

El hecho de establecer comunicación mediante un socket con un servidor, posibilita el envío y la recepción de datos. Esto se realiza con las clases de entrada y salida. La clase Socket proporciona estos métodos:

método	uso
InputStream **getInputStream()** throws IOException	Obtiene la corriente de entrada de datos para el socket
OutputStream **getOutputStream()** throws IOException	Obtiene la corriente de salida de datos para el socket

Se puede observar como se obtienen objetos de entrada y salida orientados al byte. Si la comunicación entre servidor y cliente se realiza mediante cadenas de texto (algo muy

habitual) entonces se suele convertir en objetos de tipo **BufferedReader** para la entrada (como se hacía con el teclado) y objetos de tipo **PrintWriter** (salida de datos en forma de texto):

```
try{
   Socket socket=new Socket("servidor.dementiras.com",7633);
   BufferedReader in=new BufferedReader(
         new InputStreamReader(socket.getInputStream()));
   PrintWriter out=new PrintWriter(
         socket.getOutputStream(),true)); // el parámetro
                  //true sirve para volcar la salida al
                  //dispositivo de salida (autoflush)
   boolean salir=false;
   do {
         s=in.readLine();
         if(s!=null) System.out.println(s);
         else salir=true;
   }while(!salir);
}
catch(UnknownHostException une){
   System.out.println("No se encuentra el servidor");
}
catch(IOException une){
   System.out.println("Error en la comunicación");
}
```

servidores

En el caso de un programa de servidor, éste se ha de ocupar de recibir el fuljo de datos que procede del socket del cliente (además tiene que procurar servir a varios clientes a la vez).

Para que un programa abra un socket de servidor. Se usa la clase **ServerSocket** cuyo constructor permite indicar el puerto que se abre:

```
ServerSocket socketServidor=new ServerSocket(8341);
```

Después se tiene que crear un socket para atender a los clientes. Para ello hay un método llamado **accept** que espera que el servidor atienda a los clientes. El funcionamiento es el siguiente, cuando el socket de servidorEste método obtiene un objeto Socket para comunicarse con el cliente. Ejemplo:

```
try{
   ServerSocket s=new ServerSocket(8189);
   Socket recepcion=s.accept();
   //El servidor espera hasta que llegue un cliente
```

```
  BufferedReader in=new BufferedReader(new
                        InputStreamReader(
                              recepcion.getInputStream()));
  PrintWriter out=new PrintWriter(
                        recepcion.getOutputStream(),true);
  out.println("Hola! Introduzca ADIOS para salir");
   boolean done=false;
  while(!done){
       String linea=in.readLine();
       if(linea==null) done=true;
       else{
             out.println("Echo: "+linea);
             if (linea.trim().equals("ADIOS")) done=true;
        }
   }
   recepcion.close();
}catch(IOException e){}
```

Este es un servidor que acepta texto de entrada y lo repite hasta que el usuario escribe ADIOS. Al final la conexión del cliente se cierra con el método **close** de la clase Socket.

servidor de múltiples clientes

Para poder escuchar a más de un cliente a la vez se utilizan threads lanzado un thread cada vez que llega un cliente y hay que manipularle. En el thread el método **run** contendrá las instrucciones de comunicación con el cliente.

En este tipo de servidores, el servidor se queda a la espera de clientes. cada vez que llega un cliente, le asigna un Thread para él. Por lo que se crea una clase derivada de la clase **Thread** que es la encargada de atender a los clientes.

Es decir, el servidor sería por ejemplo:

```
public static void main(String args[]){
  try{
        //Se crea el servidor de sockets
       ServerSocket servidor=new ServerSocket(8347);
       while(true){//bucle infinito
         //El servidor espera al cliente siguiente y le
        //asigna un nuevo socket
        Socket socket=servidor.accept();
        //se crea el Thread cliente y se le pasa el socket
        Cliente c=new Cliente(socket);
        c.start();//se lanza el Thread
       }
   }catch(IOException e){}
```

Por su parte el cliente tendría este código:

```
public class Cliente extends Thread{
    private Socket socket;
    public Cliente(Socket s) {
      socket=s;
    }

    public void run(){
        try{
             //Obtención de las corrientes de datos
            BufferedReader in=new BufferedReader(
                  new InputStreamReader(
                        socket.getInputStream()));
            PrintWriter out=new PrintWriter(
                        socket.getOutputStream(),true);

            out.println("Bienvenido");
            boolean salir=false;//controla la salida
            while (!salir){
               resp=in.readLine();//lectura
               ...//proceso de datos de lectura
               out.println("....");//datos de salida
               if(...) salir=true;//condición de salida
              }
              out.println("ADIOOOOOOS");
              socket.close();
        }catch(Exception e){}

    }
```

métodos de Socket

método	uso
void **setSoTimeout(**int tiempo**)**	Establece el tiempo máximo de bloqueo cuando se está esperando entrada de datos por parte del socket. Si se cumple el tiempo, se genera una interrupción del tipo: **InterruptedIOException.**
void **close()**	Cierra el socket
boolean **isClosed()**	**true** si el socket está cerrado

método	uso
void **shutdownOutput()**	Cierra el flujo de salida de datos para que el servidor (en aquellos que funcionan de esta forma) sepa que se terminó el envío de datos. Disponible desde la versión 1.3
void **shutdownInput()**	Cierra el flujo de entrada de datos. Si se intenta leer desde el socket, se leerá el fin de archivo. Desde la versión 1.3

clase InetAddress

Obtiene un objeto que representa una dirección de Internet. Para ello se puede emplear este código:

```
InetAddress dirección=InetAddress.getByName(
                    "time-a.nist.gov");
System.out.println(dirección.getHostAddress());//129.6.15.28
```

Otra forma de obtener la dirección es usar el método **getAddress** que devuelve un array de bytes. Cada byte representa un número de la dirección.

A veces un servidor tiene varios nombres. Para ello se usa:

```
InetAddress[] nombres =
        InetAddress.getAllByName("www.elpais.es");
System.out.println(nombres.length);
for(int i=0;i<nombres.length;i++) {
  System.out.println(nombres[i].getHostAddress());
}
//Escribe:
//195.176.255.171
//195.176.255.172
```

lista de métodos

método	uso
static InetAddress **getByName(**String servidor**)**	Obtiene el objeto *InetAddress* que corresponde al servidor indicado
static InetAddress **getAllByName(**String servidor**)**	Obtiene todos los objetos *InetAddress* asociados al servidor indicado
static InetAddress **getByAddress(**byte[] direcciónIP**)**	Obtiene el objeto *InetAddress* asociado a esa dirección IP
static InetAddress **getLocalHostName()**	Obtiene el objeto *InetAddress* que corresponde al servidor actual
String **getHostAddress()**	Obtiene la dirección IP en forma de cadena
byte[] **getAddress()**	Obtiene la dirección IP en forma de array de bytes
String **getHostName()**	Obtiene el nombre del servidor