élogos

INTERCONEXIÓN DE APLICACIONES MEDIANTE SOCKETS

élogos

ÍNDICE

INTERCONEXIÓN DE APLICACIONES MEDIANTE SOCKETS

1. Clases básicas para trabajar en red

Características de las clases URL, URLConnection e InetAddress

Todas las clases Java para el manejo de red se encuentran en el paquete java.net. Una de ellas es la clase URL.

La clase URL, cuyos objetos representan un recurso accesible a través de la Web.

Para crear un objeto URL, podemos utilizar el constructor: URL (String direccion).

Siendo *direccion* la cadena de caracteres que representa la dirección Web del recurso.

Entre los principales métodos están:

- getHost(). Devuelve el nombre del host donde se aleja el recurso.
- openStream(). Devuelve un objeto InputStream para poder acceder al contenido del recurso.
- openConnection(). Devuelve un objeto URLConnection asociado al recurso.

Ejemplo

```
public class Test{
    public static void main(String[] args){
    try{
        //conecta con la página de inicio de Sun
        //lee su contenido y lo muestra en la consola
    URL url =new URL("http://java.sun.com");
    BufferedReader bf=new BufferedReader (new
       InputStreamReader(url.openStream()));
    String cad="";
    while((cad=bf.readLine())!=null){
       System.out.println(cad);
      }
    }
   catch(MalformedURLException e){ //excepción de librería
       e.printStackTrace();
    }
   catch (IOException e){
       e.printStackTrace();
    }
  }
}
```

La Clase URLConnection, es una clase abstracta que representa una conexión a un recurso apuntado por un objeto URL. Para crear un objeto URLConnection, utilizaremos el método *openConnection()* de URL:

```
URL url =new URL("http://www.google.es");
URLConnection con=url.openConnection ();
```

A través de la conexión establecida con el recurso mediante este objeto URLConnection, es posible interactuar con el recurso y realizar tanto operaciones de lectura como de escritura sobre el mismo.

Entre sus principales métodos están:

- getInputStream (). Devuelve un objeto InputStream para poder acceder al contenido del recurso
- getOutputStream (). Devuelve un objeto OutputStream para poder enviar datos al recurso
- connect (). Establece una conexión con el recurso si no se hubiera creado al obtener el URLConnection
- getHeaderField (String name). Devuelve el contenido de la cabecera cuyo nombre se especifica

La Clase InetAddress

Un objeto InetAddress representa una dirección IP asociada a un equipo de la red.

La clase InetAddress no dispone de constructores, por lo que para crear un objeto InetAddress necesitamos recurrir a alguno de los siguientes métodos estáticos de la clase:

- static InetAddress getByName(String host). Devuelve el objeto InetAddress asociado a la máquina cuyo nombre de host se pasa como parámetro.
- static InetAddress getByAddress(byte[] dir). Devuelve el objeto InetAddress a partir de un array de bytes que contiene la dirección ip del equipo.

Una vez que disponemos del objeto InetAddress, recurriremos a los siguiente métodos para obtener información sobre el host al que hace referencia:

- String getHostAddress(). Devuelve la dirección IP como un String
- String getHostName(). Devuelve el nombre del host
- byte[] getAddress(). Devuelve la dirección IP como un array de bytes

2. Interconexión de conexiones mediante sockets

La clase Socket

La Clase Socket representa una conexión con TCP entre dos ordenadores. El Socket tiene tres funcionalidades básicas:

- Permite conectarse a un puerto del servidor.
- Enviar o recibir datos.
- Cerrar la conexión.

SeverSocket, además de lo que permite Socket, también permite lo siguiente:

- Esperar en un puerto.
- Aceptar conexiones.
- Aceptar un cliente que ha llegado al servidor.

Abrir un socket

Para abrir un Socket, se deben realizar 3 pasos:

1. Crear un objeto de tipo Socket utilizando su constructor. El constructor termina o bien cuando se conecta al servidor, o bien cuando se lanza una excepción. Si se lanza una excepción significa que no se ha establecido la conexión.
2. Una vez realizada la conexión se obtiene un stream de entrada y un stream de salida para establecer el diálogo full-duplex.
3. Se cierra la conexión.

Constructores de socket

- Socket (String host, int port) throws UnknownHostException, IoException
- Socket (InetAddress host, int port) throws IOException
- Socket (String host, int port, InetAddress interface, int localPort) throws UnknownHostException
- Socket (InetAddress host, int port, InetAddress interface, int localPort) throws IOException

Ejemplo

Se van a comprobar los 100 primeros puertos del servidor:

```
import java.net.*;
import java.io.*;
public class ScannerPuertos {
public static void main (String[] args) {
InetAddress dir;
//Se comprueba que se pase el argumento.
if (args.length != 1) {
System.out.println ("Debe teclear dirección del Host");
return;
}
```

```
//Resolvemos el nombre de la maquina.
try {
dir = InetAddress.getByName(args[0]);
System.out.println (dir);
}
catch(UnknownHostException ex) {
System.out.println("No se puede resolver la dirección: "
+ex);
return;
}
 for (int i=1;i <=100; i++) {
try {
Socket socket = new Socket(dir,i);
System.out.println ("El Puerto "+i+" está abierto.");
socket.close();
}
catch (IOException ex) {
System.out.print ("\r Puerto: "+i+" está cerrado");
}
}
} // fin de función main.
} // fin de la clase.
```

Métodos del socket

```
InputStream <Socket>.getInputStream() throws IOException
OutputStream <Socket>.getOutputStream() throws IOException
```

Devuelven el InputStream y el OutputStream para establecer la conversación durante la conexión.

A través de los objetos InputStream y OutputStream, la aplicación puede intercambiarse datos con el programa con el que está conectado el socket

Lectura de datos a través del socket

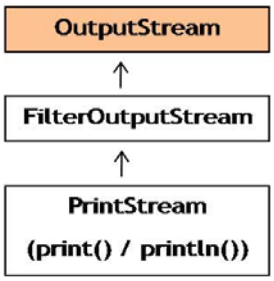
A partir del objeto InputStream devuelto por el método getInputStream() del Socket, se puede crear un objeto BufferedReader que nos permita recuperar cada línea de caracteres de entrada:

```
Socket sc=new Socket("www.ibm.com", 80);

    BufferedReader bf=new BufferedReader( new InputStreamReader(sc.getInputStream()));

String mensaje = bf.readLine();
```

Escritura en el socket

Existen dos formas para escribir con el OutputStream:

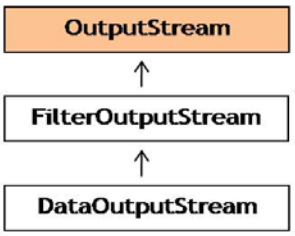
1




```
//Donde os representa un objeto
OutputStream.
PrintStream ps = new PrintStream(os);
                    ps.println ("Hola");
```

PrintStream sólo sirve para escribir texto en un Socket de salida.

2



- void <DataOutputStream>.writeInt (int i) throws IOException
- void <DataOutputStream>.writeDouble (double d) throws IOException
- void <DataOutputStream>.writeUTF (String text) throws IOException

Cierre de la conexión

Para realizar el cierre de la conexión se tiene el siguiente método:

void <Socket>.close throws IOException

- Cuando el interlocutor cierre el Socket, la siguiente operación read() que se realice devolverá un valor -1. En caso de que queden datos en el buffer primero se reciben los datos y posteriormente el valor –1.
- Si en vez de utilizar el método read() se utiliza el método readLine devolverá null cuando se cierra el Socket.

En ambos casos el hilo se desbloquea. cuando se llama al método read() el hilo se queda bloqueado hasta que llegan datos o llega -1.

Cuando el interlocutor cierra lo conexión el programa Java debe cerrar la conexión, sin embargo, en Java existe un error de diseño ya que la conexión se cierra completamente. Es decir en TCP se tiene HALF-CLOSE y FULL-CLOSE, en Java hasta la versión 1.2 sólo se tiene el método close que cierra la lectura y la escritura. En la versión 1.3 se tienen métodos para realizar estos Semicierres (HALF-CLOSE).

La Clase ServerSocket

En TCP el servidor espera hasta que alguien entre. Cuando se crea un ServerSocket, en el servidor, el hilo se queda bloqueado hasta que se conecte un cliente. Cuando el cliente se conecta, el hilo del servidor va a obtener un Socket que se va a utilizar para conectarse con el cliente.

Constructores

1. ServerSocket (int port) throws BindException, IOException

Se le dice en qué puesto se debe dejar esperando, el hilo del ServerSocket, una conexión. Razones por lo que pude fallar la creación de un objeto de tipo ServerSocket:

El puerto está siendo utilizado.
En UNIX se intenta utilizar un puerto menor que el 1024 sin ser el “superusuario”, no deja utilizar el puerto.

2. ServerSocker (int port, int queuelength)throws IOException

Permite dar el tamaño de la cola Listener. Lo que es normal es aumentar el tamaño del Listener aunque el cliente tenga que esperar para conectarse.

Aceptar y cerrar una conexión

Socket <ServerSocket>.accept() throws IOException

El código anterior sirve para que el servidor recoja al cliente que se ha conectado, te devuelve el Socket del cliente que ha entrado en el servidor. Al acabar la llamada al método accept se obtiene el objeto Socket del cliente.

- Para la comunicación se tienen los siguientes métodos:

  InputStream <Socket>.getInputStream() throws IOException

  OutputStream <Socket>.getOutputStream() throws IOException

- Para cerrar la comunicación se tiene el siguiente método:

  void <Socket>.close() throws IOException

Comunicación entre aplicaciones

Utilizando las clases Socket y ServerSocket, podemos comunicar aplicaciones Java a través de una red. En este escenario, la aplicación que hace de servidor utiliza ServerSocket para escuchar peticiones por parte de la aplicación cliente.

Una vez establecida la conexión, ambas aplicaciones se intercambian datos a través de dos objetos Sockets, uno en cliente y otro en servidor.

Ejemplo

Interconexión de aplicaciones mediante sockets

Aplicación Cliente

```
import java.net.*;
import java.io.*;
public class Cliente {
   public static void main(String[] args) {
      int puerto=9000;
      String mensaje=null;
      try{
         Socket conexion=new Socket("localhost",puerto);
         PrintWriter out=new PrintWriter(
            conexion.getOutputStream(),true);
         //solicita un dato al cliente
         BufferedReader bf=new BufferedReader(
            new InputStreamReader(System.in));
         System.out.println("Introduce tu nombre");
         //envía el dato solicitado al servidor
         out.println(bf.readLine());
         //envío inmediato
         out.flush();
         //recupera la respuesta generada por el servidor
         //al enviarle el dato y la muestra en pantalla
         BufferedReader bfsocket=new BufferedReader(
          new InputStreamReader(conexion.getInputStream()));
         mensaje=bfsocket.readLine();

         System.out.println("Mensaje recibido "+mensaje);

         conexion.close();
      }
      catch(UnknownHostException e){
         System.out.println("El host indicado no existe");
      }
      catch(IOException e){
         System.out.println("Error en la entrada/salida de datos");
      }
   }
}
```

Aplicación Servidor

```
import java.net.*;
import java.io.*;
public class Servidor {
   public static void main(String[] args) {
      int puerto=9000;
         try{
        ServerSocket servidor=new ServerSocket(puerto);
        //realiza la tarea indefinidamente
        while(true){
           System.out.println("Esperando petición...");
           //cuando recibe una petición desde un cliente
           // crea un hilo para atenderlo y le pasa el socket
           //que le conecta con el cliente
           Socket con=servidor.accept();
           (new ProcesaConexion(con)).start();

        }
      }
      catch(UnknownHostException e){
         System.out.println("El host indicado no existe");
      }
      catch(IOException e){
         System.out.println("Error en la entrada/salida de datos");
      }
   }
}
//////////////////
import java.net.*;
import java.io.*;
public class ProcesaConexion extends Thread{
   Socket sc;
     public ProcesaConexion(Socket sc){
      this.sc=sc;

   }
   public void run(){
      try{
         PrintWriter out=new
            PrintWriter(sc.getOutputStream());
         BufferedReader bf=new BufferedReader(new
            InputStreamReader(sc.getInputStream()));
         //recupera la cadena enviada por el cliente
         //al establecer la conexión
         String nombre=bf.readLine();
         //envia mensaje de saludo
         out.println("Bienvenido: "+nombre);
         //envío inmediato
         out.flush();
         sc.close();
      }
      catch(IOException e){
         System.out.println("Error en la entrada/salida de datos");
      }
      }
}
```

3. Programación de datagramas

Hay dos clases que representan la programación con los datagramas:

1. La clase DatagramPacket, representa un datagrama a enviar o recibir.

2. La clase DatagramSocket, es el objeto que permite enviar o recibir los DatagramPacket.

   - En UDP no se diferencia entre el cliente y el servidor, se van a poder enviar datagramas a varios receptores y se van a poder recibir de varios sitios.
   - TCP es unicast (uno a uno).
   - UDP es multicast (varios a uno/uno a varios).

La clase DatagramPacket

La clase DatagramPacket tiene dos constructores, un constructor para los datagramas que se quieren enviar y un constructor para los datagramas que se van a recibir.

Constructor para recibir datos

Esta excepción no habrá que controlarla ya que deriva de RuntimeException y como sabemos todas las que deriven de esta última no hay que controlarlas.

DatagramPacket (byte buffer [], int length)

Los datos que llegan serán depositados en el buffer. El parámetro length indica el número de datos que se quieren recibir y por lo tanto indica también que el tamaño del buffer no puede ser mayor que la longitud indicada en length.

En el caso del que el tamaño del buffer sea superior a lo indicado por length, se producirá la excepción siguiente: **IllegalArgumentException.**

Constructor para enviar datos

DatagramPacket (byte buffer [], int length, InetAddress ia, int port)

Los objetos DatagramPacket se van a poder reutilizar para los distintos envíos o recepciones que se realicen. Para ello se debe crear un buffer lo suficientemente grande como para que no haya problemas y en el parámetro length se le indicará lo que mide.

Interesa saber la **dirección IP** y el **puerto**.

Estos métodos en los datagramas recibidos indican la dirección IP y el puerto de origen, y en los datagramas enviados indican la dirección IP y el puerto destino.
Para modificar el tamaño del buffer o el contenido del buffer se tienen **métodos**.

Para obtener la dirección IP se tiene el siguiente método:

InetAddress <DatagramPacket>.getAdress();

Para obtener el puerto se tiene el siguiente método:

int <DatagramPacket>.getPort();

Métodos

void <DatagramPacket>.setData (byte buffer[])
void <DatagramPAcket>.setlength (int length)

La clase DatagramaSocket

La clase DatagramSocket es el objeto que utilizará el cliente y el servidor para enviar y recibir los DatagramPacket.

Tiene tres constructores:

- **DatagramSocket () throws SocketException** es el constructor que utilizará el cliente y se le asignará un puerto anónimo.

- **DatagramSocket (int port) throws SocketException** es el constructor que utilizará el servidor que indica el puerto en el que ha de instalarse.

- **DatagramSocket (int port, InetAddress interface) throws SocketException** , este constructor también lo utiliza el servidor e indica dónde hay que enviar el datagrama.

Métodos para enviar y recibir datagramas:

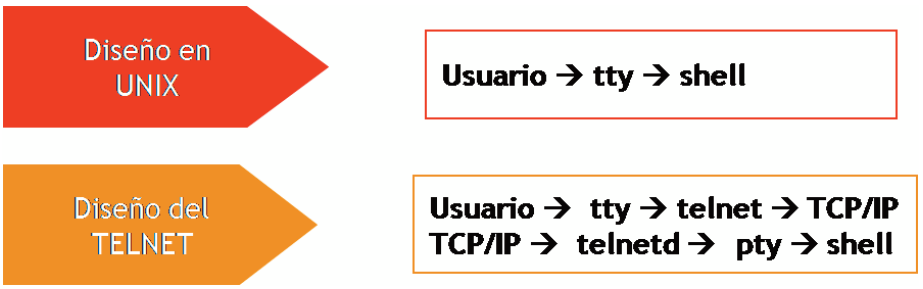
```
void <DatagramSocket>.send (DatagramPacket dp)  throws IOException

void <DatagramSocket>.receive (DatagramPacket dp)  throws IOException
```

4. TELNET y FTP

TELNET

TELNET es un protocolo estándar que permite ejecutar comandos en un shell (intérprete de comando situado en otro ordenador), con lo cual se va a poder administrar otro ordenador en remoto.

Este servicio inicialmente se inventó en UNIX pero actualmente está en cualquier sistema.



El tty es un programa que recibe pulsaciones de teclas las cuales son enviadas al shell. El shell es el programa que se encarga de ejecutar los comandos.

El tty manda las pulsaciones al telnet que lo envía al TCP/IP. TCP/IP manda las peticiones al telnet, el cual los envía al pty y finalmente el pty lo envía la shell.

FTP (File Tranfer Protocol)

FTP (File Tranfer Protocol) es un protocolo creado en 1985 y está documentado en el RFC 959. Permite enviar y recibir archivos entre máquinas.

URL desde browser para conectarse a FTP:

ftp:///[<NombreUsuario>][:<Contraseña>]@]<Host>[:<puerto>][<Directorio>]

El Protocolo FTP
El protocolo FTP tiene dos Socket:

Interconexión de aplicaciones mediante sockets

El Socket de comandos

El Socket de comandos, únicamente le sirve al cliente para enviar comandos al servidor y recibir respuestas de éste. Es un puerto que permanece abierto durante toda la sesión. El servidor utilizará el puerto 21 y el cliente un puerto anónimo.

```
                          USER flopez \r\n
active open  ------------------------------------------->  passive open
puerto 1173   331 Password required for flopez             puerto 21
```

El cliente y el servidor se envían mutuamente mensajes de texto plano y el servidor los identifica con un código de tres dígitos seguido de un texto explicativo del código.

El Socket de datos

El Socket de datos, se abre cada vez que se va a enviar un archivo y se cierra tras haber transferido el archivo. En definitiva sirve para transferir archivos.

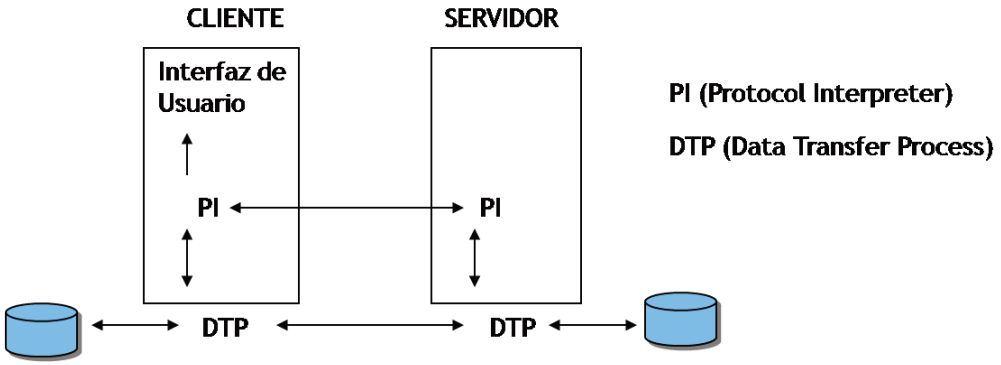
```
                          Dir. IP          Puerto
140.252.13.54             ---------------  --------
                      PORT 140, 252, 13, 54, 4, 150 \r\n
active open  ------------------------------------------->  passive open
puerto 1173                                                puerto 21
                      SYN to 140.252.13.54 port 1174
passive open <-------------------------------------------  active open
puerto 1174                                                puerto 20
```

Según el RFC 959 el servidor tiene que conectarse al puerto 20 para conectarse al cliente. Utilizar el puerto 20 tiene el inconveniente de que después de cerrar el puerto se queda en situación TIME-WAIT durante un tiempo. Tras transferir un fichero hay que esperar que transcurra el TIME_WAIT para transferir el siguiente fichero.

En una RFC 1134 posterior, se hizo una revisión de FTP por el problema descrito anteriormente y se dijo que el servidor en vez de utilizar el puerto 20, lo que tiene que hacer es utilizar un puerto anónimo y además cada vez distinto, con el fin de evitar los retardos producidos por el TIME_WAIT.

Estructura interna del Cliente y el Servidor FTP

Veamos un esquema con la estructura interna del Cliente y el servidor FTP:



PI, es el que se encarga de enviar los comandos (por el Socket de comandos) e interpreta los códigos de repuesta. Los comandos siempre van del cliente al servidor y el servidor siempre genera códigos de respuesta.

DTP, se encarga de enviar y recibir los archivos. Su misión es enviar archivos por el Socket de datos.
El DTP va a tener un hilo por cada fichero que se esté enviando o recibiendo.