LPS: Red y conexión remota

Federico Peinado
www.federicopeinado.es

Depto. de Ingeniería del Software e Inteligencia Artificial
disia.fdi.ucm.es

Facultad de Informática
www.fdi.ucm.es

Universidad Complutense de Madrid
www.ucm.es

Red

- Conjunto de máquinas conectadas a través de una topología física y uno o más protocolos lógicos
- Los protocolos más relevantes (*Transport Control Protocol* e *Internet Protocol*) son los que dan nombre a las conocidas redes TCP/IP (como es el caso de Internet)
  - Cada máquina está identificada por una *dirección IP,* 4 bytes expresados como 4 números entre 0 y 255
    - Por ejemplo *147.96.1.9*
  - Las máquinas sólo entienden direcciones IP, pero no *nombres de dominio* (DNS). Son los servidores de nombres los que traducen nombres de dominio a IPs.
    - Por ejemplo *www.ucm.es*
  - Una conexión TCP/IP sólo conecta dos ordenadores distintos (existiendo, en principio, por cada máquina *una sola conexión a la vez*)

Conexión remota a puertos

- Para ser precisos cada conexión IP se asocia a *una dirección IP y un puerto determinado* (como un casillero interno) dentro de la máquina de destino
  - Los puertos se identifican mediante 2 bytes, un número entre 0 y 65535
- Gracias a los puertos, cada aplicación puede recibir *distintas conexiones* (cada una en un puerto distinto, claro)
  - Cada aplicación define en qué puertos escucha
  - Los clientes se conectan al puerto especificado
- Los 1024 primeros puertos de todas las máquinas están "reservados" para aplicaciones concretas (aunque algunos no se usan)
  - Algunos de los más conocidos: 21 (*ftp*), 23 (*telnet*), 25 (*smtp*), 80 (*http*), etc. porque los programas de un servidor que implementa estos protocolos siempre escuchará en estos puertos

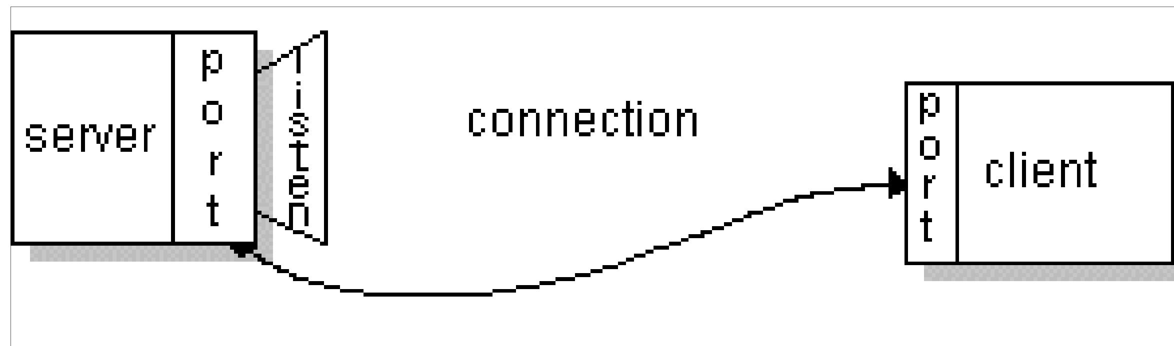
Protocolos de transporte

- Los *protocolos de transporte* sirven para enviar información sofisticada de un puerto de una máquina al puerto de otra
  - *Los protocolos de aplicación* son aún de más alto nivel y dependerán de la aplicación concreta
- Protocolos de transporte que se utilizan en redes IP:
  - TCP
    - Protocolo basado en la conexión punto a punto que provee un flujo fiable de datos entre dos máquinas
      - Ejemplos: HTTP, FTP, SMTP…
  - UDP (*User Datagram Protocol*)
    - Protocolo para envío de paquetes de datos de manera independiente, llamados datagramas, de una computadora a otra *sin garantizar su llegada*
      - Ejemplos: MMS (streaming de video), intercambio de fuentes en redes P2P…

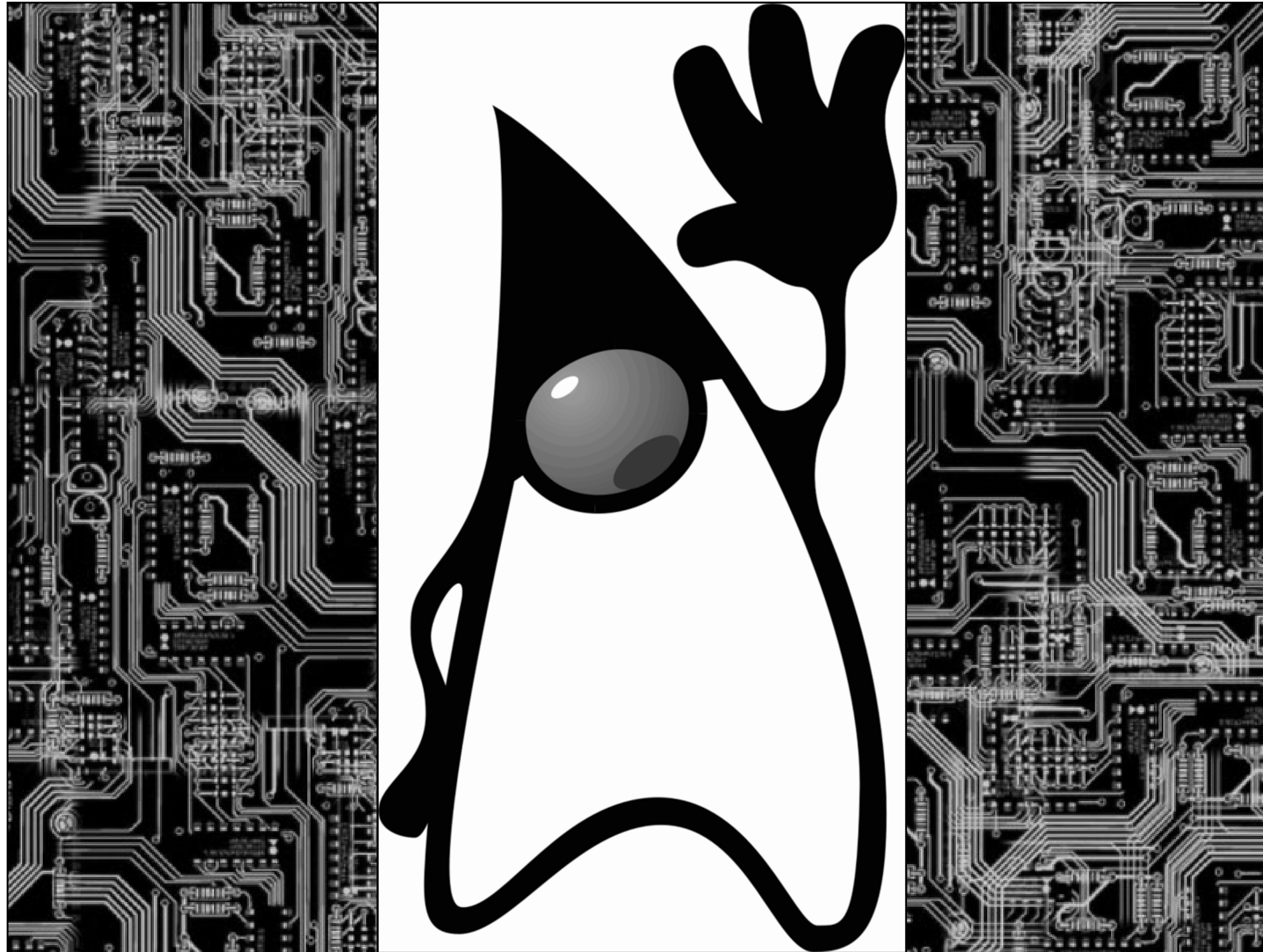
Arquitectura Cliente/Servidor

- Según este modelo, hay dos tipos de máquinas: los clientes y los servidores. Los clientes se conectan a servidores, les hacen solicitudes de servicios y estos responden proporcionándolos
  - Ejemplos: Navegadores/servidores web, clientes/servidores de correo…
- Clientes y servidores establecen conexiones TCP y se intercambian remotamente datos a través de sockets
  - Vínculos de comunicación que se crean entre dos aplicaciones según el protocolo TCP (siempre asociados a una dirección IP y un puerto)
  - Puntos finales de la comunicación en Internet
- Un cliente se comunica con un servidor estableciendo una conexión con el socket que tiene el servidor
  - Conociendo la dirección IP del servidor y el puerto de conexión

Cliente/Servidor con Sockets



Red y conexión remota con Java



Red y conexión remota con Java

- Java tiene diversos mecanismos para la red
  - Sockets: Acceso avanzado para comunicación fiable (TCP)
  - Datagramas: Acceso avanzado para comunicación no fiable (UDP)
  - JDBC: Permite acceso remoto a bases de datos
  - URLs: Acceso simple orientado a la web
  - RMI: Invocación Remota de Procedimientos
  - Servicios web
  - …

TCP/IP y Sockets con Java

- Estudiaremos conexiones TCP de bajo nivel
  - Es decir, el protocolo de aplicación lo definiremos nosotros
  - Para cuando ambas aplicaciones son Java existen mecanismos *de más alto nivel* (como RMI)
- En Java las clases relacionadas con la red están dentro del paquete **`java.net`**
  - La clase **`InetAddress`** sirve para trabajar con direcciones de red
  - La clase **`Socket`** representa un socket
  - La clase **`ServerSocket`** facilita la creación de sockets para esperar conexiones de clientes

InetAddress

- Sirve para manejar direcciones de Internet
  - Gestión de nombres de dominio
  - Validación de direcciones IPs

```
InetAddress host = InetAddress.getByName("www.ucm.es");

System.out.println("Host name: "  host.getHostName());
System.out.println("IP address: " +
                    host.getHostAddress());
```

Cliente

Socket

- Implementa una conexión tipo “socket de cliente” basado en una conexión punto a punto
- Al crearlo, se le pasa una dirección IP y un puerto
- Una vez establecida la conexión, ofrece flujos para leer y escribir en el socket
  - Tiene asociado un flujo de entrada
    - *InputStream getInputStream()*
  - Tiene asociado un flujo de salida
    - *OutputStream getOutputStream()*

Esquema de un cliente simple

1. Abrir la *conexión* (Socket)
2. Crear los flujos de entrada y salida asociados a dicha conexión
3. Leer y escribir de los flujos de entrada y salida de acuerdo al protocolo de aplicación que hayamos definido
4. Cerrar los flujos de entrada y salida
5. Cerrar la conexión

Ejemplo de cliente

- Requisitos: que sea capaz de solicitar a www.ucm.es su página web inicial
- Utilizamos el protocolo HTTP (puerto 80)
- El protocolo de aplicación será:
  1. El cliente se conecta al puerto 80 del servidor
  2. El cliente escribe en el socket la cadena "GET" seguido del nombre del fichero (página)
  3. El servidor envía por el socket el fichero (página)
  4. El servidor cierra la conexión
     - Luego el cliente ya no leerá más información

Ejemplo: Client.java

```
import java.net.*;
import java.io.*;

public class Client {

   public static void main(String[] args) {
      Socket s = null;
      BufferedReader in = null;
      PrintWriter out = null;

      // 1. Abrimos la conexión
      // 2. Creamos los flujos de entrada y salida
      // 3. Escribimos la solicitud (GET)
      // 4. Leemos hasta que no haya nada más
      // 5. Cerramos la conexión
     ...
   }
}
```

Ejemplo: Client.java

```
// 1. Abrimos la conexión
      try {
         s = new Socket("www.google.es", 80);
      } catch (IOException ex) {
         System.err.println("Error conectándonos al servidor.");
         System.err.println(ex);
         return;
      }

 // 2. Creamos los flujos de entrada y salida
      try {
         in = new BufferedReader(new InputStreamReader(
                                             s.getInputStream()));
         out = new PrintWriter(s.getOutputStream());
      } catch (IOException ex) {
         System.err.println("Error al crear los flujos");
         System.err.println(ex);
         return;
      }
```

Ejemplo: Client.java

```
// 3. Escribimos la solicitud (GET)
      out.println("GET index.html");
      out.flush();  // Nos aseguramos que se envía

// 4. Leemos hasta que no haya nada más
      String entrada;
      try {
         while ((entrada = in.readLine()) != null)
            System.out.println(entrada);
      } catch (IOException ex) {
         System.err.println("Error de lectura.");
         System.err.println(ex);
      }
// 5. Cerramos la conexión
      try {
         in.close();
         out.close();
         s.close();
      } catch (IOException ex) {
         System.err.println("Error cerrando la entrada.");
         System.err.println(ex);
      }
```

Ejemplo: Micro-navegador

```
try {
    Socket s = new Socket ("www.ucm.es", 80);

    BufferedReader in = new BufferedReader (new
                        InputStreamReader(s.getInputStream()));

    PrintWriter out = new PrintWriter(s.getOutputStream(), true);
    out.println("GET / HTTP/1.0");
    out.println();

    boolean more = true;
    while (more) {
        String line = in.readLine();
        if (line == null)
            more = false;
        else
            System.out.println(line);
    }

} catch (IOException e) {
     e.printStackTrace();
}
```

Aparecerá el código HTML tomado de aquí...

Servidor

ServerSocket

- Implementa un conexión tipo "socket de servidor" asociada a una dirección IP y un puerto
- Espera hasta que se conecta algún cliente y le asigna un socket cliente
  - **`Socket accept();`**
  - ¡Ojo! La comunicación se realiza entre Sockets *iguales*, ServerSocket sólo ayuda a establecer dicha conexión
  - El Socket asociado al ServerSocket tiene los correspondientes flujos de entrada y salida
    - **`InputStream getInputStream()`**
    - **`OutputStream getOutputStream()`**

Esquema de un servidor simple

1. Se crea el *servidor de escucha* (SocketServer)
2. Se bloquea esperando una *conexión* (Socket)
   - El bloqueo puede tener un cierto *timeout*
3. Un vez conectado un cliente, se crean los flujos de entrada y salida asociados a la conexión
4. Se lee y escribe en ellos en base al protocolo de aplicación que hayamos definido
5. Se cierran los flujos de entrada y salida, y también la conexión
6. Se cierra el servidor de escucha

Ejemplo de servidor

- Requisitos: que sepa esperar y atender a al cliente que se conecte por el puerto 4444
  1. Cuando alguien se conecta se le envía la cadena "Dime algo"
  2. El servidor se queda leyendo del flujo de entrada de la conexión
  3. Todo lo que llega lo va escribiendo por pantalla

Ejemplo: Server.java

```
import java.net.*;
import java.io.*;

class Server {

   public static final int PUERTO = 4444;

   public static void main(String []args) {

      ServerSocket ss;
      Socket conexion;
      BufferedReader conIn;
      BufferedWriter conOut;

      // 1. Creamos el servidor de escucha
      // 2. Esperamos una conexión de un cliente
      // 3. Creamos los flujos de entrada y salida
      // 4. Leemos y escribimos según el protocolo
      // 5. Cerramos flujos de entrada, salida y conexión
      // 6. Cerramos el servidor de escucha
      ...
   }
}
```

Ejemplo: Server.java

```
// 1. Creamos el servidor de escucha
try {
    ss = new ServerSocket(PUERTO);
} catch (IOException e) {
    // Suele fallar porque hay otro servidor escuchando en el
  mismo puerto (¿otra instancia del nuestro?)
    System.err.println("Error al abrir el puerto para escucha.");
    System.err.println(e);
    return;
}
// 2. Esperamos una conexión de un cliente (Versión 1)
try {
   conexion = ss.accept();
} catch (IOException e) {
    System.err.println("Error esperando la conexión.");
    System.err.println(e);
    return;
}
```

Ejemplo: Server.java

```
// 2. Esperamos una conexión de un cliente (Versión 2)

// Hay tiempo de espera máximo; si en ese tiempo no llegan
  clientes, se genera una excepción que podemos procesar.

try {
   ss.setSoTimeout(1000); // Tiempo máximo de espera
   conexion = ss.accept();
} catch (java.net.SocketTimeoutException ex) {
   // Excepción generada si se excede el timeout de espera
} catch (IOException e) {
   System.err.println("Error esperando la conexión.");
   System.err.println(e);
   return;
}
```

Ejemplo: Server.java

```
// 3. Creamos los flujos de entrada y salida
try {
  conIn = new BufferedReader(
        new InputStreamReader(conexion.getInputStream()));
  conOut = new BufferedWriter(
        new OutputStreamWriter(conexion.getOutputStream()));
} catch (IOException e) {
  System.err.println("Error al crear flujos de comunicación.");
  System.err.println(e);
  return;
}
// 4. Leemos y escribimos según el protocolo
try {
  conOut.write("Dime cosas.\n");
  conOut.flush();
  String cad;
  while ((cad = conIn.readLine()) != null)
      System.out.println("-> " + cad);
} catch (IOException ex) {
  System.err.println("Error durante la comunicación.");
  System.err.println(ex);
}
```

Ejemplo: Server.java

```
// 5. Cerramos flujos de entrada, salida y conexión
try {
   conOut.close();
   conIn.close();
   conexion.close();
} catch (IOException ex) {
   System.err.println("Error cerrando conexión");
   System.err.println(ex);
}
// 6. Cerramos el servidor de escucha
try {
   ss.close();
} catch (IOException ex) {
   System.err.println("Error cerrando el servidor");
   System.err.println(ex);
}
```

Probar el servidor

- Para probar el servidor se puede usar *telnet* (requiere activación previa en Windows 7)
  1. Se lanza el servidor
  2. Desde el símbolo del sistema (consola del S.O.):
     `telnet localhost 4444`
  3. Aparecerá la cadena "Dime cosas"
  4. Se puede escribir en el cliente de Telnet, y este lo enviará al servidor a través del socket
  5. Lo escrito aparecerá en el servidor
  6. Para terminar, se puede parar el servidor (Algunos clientes de Telnet permiten cerrar la conexión desde el lado del cliente)

Ejemplo de servidor (Versión 2)

- Requisitos: análogos al anterior, pero cambiando la implementación del protocolo de la aplicación (ahora se permite *enviar y recibir a la vez*)
  1. El servidor comprueba si hay algo que leer en el socket
     - Si es así, lo lee y lo escribe en la consola
  2. El servidor comprueba si hay algo que leer de teclado
     1. Si la cadena es igual a “fin”, termina
     2. Si no, lo manda a través del socket

Ejemplo: Server2.java

```
// 4. Leemos y escribimos según el nuevo protocolo
BufferedReader entradaUsuario = new BufferedReader(
                        new InputStreamReader(System.in));
try {
   while (true) {
      if (conIn.ready()) {
         System.out.println(conIn.readLine());
      } else if (entradaUsuario.ready()) {
         String entrada = entradaUsuario.readLine();
         if (entrada.equals("fin"))
            break;
         conOut.write(entrada);
         conOut.flush();
      }
   }
} catch (IOException ex) {
   System.err.println("Error durante la comunicación.");
   System.err.println(ex);
}
```

Múltiples clientes

- El esquema anterior *no* permite varios clientes
  - Cuando se conecta un cliente, se sigue el protocolo, y cuando se desconecta, la ejecución del servidor *termina*
- Para permitir varios clientes existen dos alternativas:
  1. Permitir clientes de forma *no* simultánea
     - Para el primer caso, basta con volver a hacer un **`accept`** cuando se termina la conexión con un cliente
  2. Permitir varios clientes simultáneos
     - Necesitaremos usar varios **hilos de ejecución**
     - Cada vez que se conecte un cliente lanzaremos un nuevo hilo, encargado de mantener la conexión entre ese cliente y el servidor

Ejemplo: Micro-chat

- Requisitos: permitirá que varios usuarios se conecten e intercambien mensajes cortos en tiempo real y de forma simultánea
  - Creamos un SocketServer escuchando en el puerto 8000
  - Cada vez que se conecta un cliente crea un hilo nuevo para gestionar esa conexión
  - Lee lo que le manda cada cliente y lo envía a todos con un método ***sincronizado***
  - Cuando recibe "ADIOS" termina y elimina al cliente de la lista

Críticas, dudas, sugerencias…

Federico Peinado
www.federicopeinado.es