15: Computación distribuida

Históricamente, la programación a través de múltiples máquinas ha sido fuente de error, difícil y compleja.

El programador tenía que conocer muchos detalles sobre la red y, en ocasiones, incluso el hardware. Generalmente era necesario comprender las distintas "capas" del protocolo de red, y había muchas funciones diferentes en cada una de las bibliotecas de la red involucradas con el establecimiento de conexiones, el empaquetado y desempaquetado de bloques de información; el reenvío y recepción de esos bloques; y el establecimiento de acuerdos. Era una tarea tremenda.

Sin embargo, la idea básica de la computación distribuida no es tan complicada, y está abstraída de forma muy elegante en las bibliotecas de Java. Se desea:

- Conseguir algo de información de una máquina lejana y moverla a la máquina local, o viceversa. Esto se logra con programación básica de red.
- Conectarse a una base de datos, que puede residir en otra máquina. Esto se logra con la *Java DataBase Connectivity* (JDBC), que es una abstracción alejada de los detalles difíciles y específicos de cada plataforma de SQL (el lenguaje de consulta estructurado o *Structured Query Language* que se usa en la mayoría de transacciones de bases de datos).
- Proporcionar servicios vía un servidor web. Esto se logra con los *servlets* de Java y las *Java Server Pages* (JSP).
- Ejecutar métodos sobre objetos Java que residan en máquinas remotas, de forma transparente, como si estos objetos residieran en máquinas locales. Esto se logra con el *Remote Method Invocation* (RMI) de Java.
- Utilizar código escrito en otros lenguajes, que está en ejecución en otras arquitecturas. Esto se logra usando la *Common Object Request Broker Architecture* (CORBA), soportado directamente por Java.
- Aislar la lógica de negocio de aspectos de conectividad, especialmente en conexiones con bases de datos, incluyendo la gestión de transacciones y la seguridad. Esto se logra usando *Enterprise JavaBeans* (EJB). Los EJB no son una arquitectura distribuida, sino que las aplicaciones resultantes suelen usarse en un sistema cliente-servidor en red.
- Fácilmente, dinámicamente, añadir y quitar dispositivos de una red que representa un sistema local. Esto se logra con Jini de Java.

En este capítulo se dará a cada tema una ligera introducción. Nótese, por favor, que cada tema es bastante voluminoso y de por sí puede ser fuente de libros enteros, por lo que este capítulo sólo pretende familiarizar al lector con estos temas, y no convertirlo en un experto (sin embargo, se puede recorrer un largísimo camino en la programación en red, *servlets* y JSP con la información que se presenta aquí).

Programación en red

Una de las mayores fortalezas de Java es su funcionamiento inocuo en red. Los diseñadores de bibliotecas de red de Java han hecho que trabajar en red sea bastante similar a la escritura y lectura de archivo, excepto en que el "archivo" exista en una máquina remota y la máquina remota pueda decidir exactamente qué desea hacer con la información que se le envía o solicita. Siempre que sea posible, se han abstraído los detalles de la red subyacente, dejándose para la JVM y la instalación de Java que haya en la máquina local. El modelo de programación que se usa es el de un archivo; de hecho, se envuelve la conexión de red (un "socket") con objetos flujo, por lo que se acaban usando las mismas llamadas a métodos que con el resto de flujos. Además, el multihilo inherente a Java es extremadamente útil al tratar con otro aspecto de red: la manipulación simultánea de múltiples conexiones.

Esta sección introduce el soporte de red de Java usando ejemplos fáciles de entender.

Identificar una máquina

Por supuesto, para comunicarse con una máquina desde otra y para asegurarse de estar conectado con una máquina particular, debe haber alguna forma de identificar de forma unívoca las máquinas de una red. Las primeras redes se diseñaron de forma que proporcionaran nombres únicos de máquinas dentro de la red local. Sin embargo, Java trabaja dentro de Internet, lo que requiere una forma de identificar una máquina unívocamente de entre todas las demás máquinas *del mundo*. Esto se logra con la dirección IP (*Internet Protocol*) que puede existir de dos formas:

1. La forma familiar DNS (*Domain Name System*). Nuestro nombre de dominio es **bruceeckel.com,** y de tener un computador denominado **Opus** en nuestro dominio, su nombre de dominio habría sido **Opus.bruceeckel.com**. Éste es exactamente el tipo de nombre que se usa al enviar un correo a la gente, y suele incorporarse en una dirección World Wide Web.
2. Alternativamente, puede usarse la forma del "cuarteto punteado", que consta de cuatro números separados por puntos, como **123.255.28.120**.

En ambos casos, la dirección IP se representa internamente como un número de 32[1] bits (de forma que cada uno de los cuatro números no puede exceder de 255), y se puede lograr un objeto Java especial para que represente este número de ninguna de las formas de arriba, usando el método **static InetAddress.getByName( )** que está en **java.net**. El resultado es un objeto de tipo **InetAddress**, que puede usarse para construir un "socket" como se verá después.

Como un simple ejemplo del uso de **InetAddress.getByName( )**, considérese lo que ocurre si se tiene un proveedor de servicios de Internet (ISP o *Internet Service Provider*) vía telefónica. Cada vez que uno llama, le asignan una dirección IP temporal. Pero mientras se está conectado, la dirección IP de cada uno tiene la misma validez que cualquier otra dirección IP de la red. Si alguien se co-

[1] Esto implica un máximo de algo más de cuatro millones de números, que se está agotando rápidamente. El nuevo estándar para direcciones IP usará un número de 128 bits, que debería producir direcciones IP únicas suficientes para el futuro predecible.

necta a la máquina utilizando su dirección IP puede conectarse a un servidor web o a un servidor FTP en ejecución en tu máquina. Por supuesto, necesitan saber la dirección IP y cómo éste varía cada vez que uno se conecta, ¿cómo puede saberse?

El programa siguiente usa **InetAddress.getByName( )** para producir la dirección IP actual. Para usarla, hay que saber el nombre del computador que se esté usando. En Windows 95/98 hay que ir a "Configuración", "Panel de Control", "Red" y después seleccionar la solapa "Identificación". El campo "Nombre de PC" es el que hay que poner en la línea de comandos:

```
//: c15:QuienSoyYo.java
// Averigua la dirección de red cuando
// se está conectado a Internet.
import java.net.*;

public class QuienSoyYo {
  public static void main(String[] args)
      throws Exception {
    if(args.length != 1) {
      System.err.println(
        "Uso: QuienSoyYo NombrePC");
      System.exit(1);
    }
    InetAddress a =
      InetAddress.getByName(args[0]);
    System.out.println(a);
  }
} ///:~
```

En este caso, la máquina se denomina "pepe". Por tanto, una vez que nos hemos conectado a nuestro ISP ejecutamos el programa:

```
java QuienSoyyo pepe
```

Recibimos un mensaje como éste (por supuesto, la dirección es distinta cada vez):

```
pepe/199.190.87.75
```

Si le decimos a un amigo esta dirección y tenemos un servidor web en ejecución en nuestro PC, éste puede conectarse a la misma acudiendo a la URL *http://199.190.87.75* (sólo mientras continuemos conectados durante esa sesión). Ésta puede ser una forma útil de distribuir información a alguien más, o probar la configuración de un sitio web antes de enviarlo a un servidor "real".

Servidores y clientes

Todo el sentido de una red es permitir a dos máquinas conectarse y comunicarse. Una vez que ambas máquinas se han encontrado mutuamente, pueden tener una conversación bidireccional. Pero,

¿cómo se identifican mutuamente? Es como perderse en un parque de atracciones: una máquina tiene que estar en un lugar y escuchar mientras la otra máquina dice: "Eh, ¿dónde estás?"

A la máquina que "sigue en un sitio" se le denomina el *servidor*, y a la que la busca se le denomina el *cliente*. Esta distinción es importante sólo mientras el cliente está intentando conectarse al servidor. Una vez que se han conectado, se convierte en un proceso de comunicación bidireccional y no vuelve a ser importante el hecho de si alguno desempeñó el papel de servidor y el otro el de cliente.

Por tanto, el trabajo del servidor es esperar una conexión, y ésta se lleva a cabo por parte del objeto servidor especial que se crea. El trabajo del cliente es intentar hacer una conexión al servidor, y esto lo logra el objeto cliente especial que se crea. Una vez que se hace la conexión se verá que en ambos extremos, servidor y cliente, la conexión se convierte mágicamente en un objeto flujo E/S, y a partir de ese momento se puede tratar la conexión como si simplemente se estuviera leyendo y escribiendo a un archivo. Por consiguiente, tras hacer la conexión, simplemente se usarán los comandos de E/S familiares del Capítulo 11. Ésta es una de las mejores facetas del funcionamiento en red de Java.

Probar programas sin una red

Por muchas razones, se podría no tener una máquina cliente, una máquina servidora y una red disponibles para probar los programas. Se podrían estar llevando a cabo ejercicios en una situación parecida a la de una clase, o se podrían estar escribiendo programas que no son aún lo suficientemente estables como para ponerlos en la red. Los creadores del Internet Protocol eran conscientes de este aspecto, y crearon una dirección especial denominada **localhost** para ser una dirección IP "bucle local" para hacer pruebas sin red. La forma genérica de producir esta dirección en Java es:

```
InetAddress addr = InetAddress.getByName(null);
```

Si se pasa un **null** a **getByName( )**, éste pasa por defecto a usar el **localhost**. La **InetAddress** es lo que se usa para referirse a la máquina particular, y hay que producir este nombre antes de seguir avanzando. No se pueden manipular los contenidos de una **InetAddress** (pero se pueden imprimir, como se verá en el ejemplo siguiente). La única forma de crear un **InetAddress** es a través de uno de los siguientes métodos miembro **static** sobrecargados: **getByName( )** (que es el que se usará generalmente), **getAllByName( )** o **getLocalHost( )**.

También se puede producir la dirección del bucle local pasándole el String **localhost**:

```
InetAddress.getByName("localhost").
```

(asumiendo que "localhost" está configurado en la tabla de "hosts" de la máquina), o usando su forma de cuarteto punteado para nombrar el número IP reservado del bucle:

```
InetAddress.getByName("127.0.0.1");
```

Las tres formas producen el mismo resultado.

Puerto: un lugar único dentro de la máquina

Una dirección IP no es suficiente para identificar un único servidor, puesto que pueden existir muchos servidores en una misma máquina. Cada máquina IP también contiene *puertos*, y cuando se establece un cliente o un servidor hay que elegir un puerto en el que tanto el cliente como el servidor acuerden conectarse; si se encuentra alguno, la dirección IP es el barrio, y el puerto es el bar.

El puerto no es una ubicación física en una máquina, sino una abstracción software (fundamentalmente para propósitos de reservas). El programa cliente sabe como conectarse a la máquina vía su dirección IP, pero, ¿cómo se conecta a un servicio deseado (potencialmente uno de muchos en esa máquina)? Es ahí donde aparecen los puertos como un segundo nivel de direccionamiento. La idea es que si se pregunta por un puerto en particular, se está pidiendo el servicio asociado a ese número. La hora del día es un ejemplo simple de un servicio. Depende del cliente el que éste conozca el número de puerto sobre el que se está ejecutando el servicio deseado.

Los servicios del sistema se reservan el uso de los puertos del 1 al 1.024, por lo que no se deberían usar estos números o cualquier otro puerto que se sepa que está en uso. La primera elección para los ejemplos de este libro será el puerto 8080 (en memoria del venerable chip Intel 8080 de 8 bits de nuestro primer computador, una máquina CP/M).

Sockets

El *socket* es la abstracción software que se usa para representar los "terminales" de una conexión entre dos máquinas. Para una conexión dada, hay un socket en cada máquina, y se puede imaginar un "cable" hipotético entre las dos máquinas, estando cada uno de los extremos del "cable" enchufados al socket. Por supuesto, se desconoce completamente el hardware físico y el cableado entre máquinas. Lo fundamental de esta abstracción es que no hay que saber nada más que lo necesario.

En Java, se crea un socket para hacer una conexión a la otra máquina, después se logra un **InputStream** y un **OutputStream** (o, con los convertidores apropiados, un **Reader** y un **Writer**) a partir del socket para poder tratar la conexión como un objeto flujo de E/S. Hay dos clases de socket basadas en flujos: un **ServerSocket** que usa un servidor para "escuchar" eventos entrantes y un **Socket** que usa un cliente para iniciar una conexión. Una vez que un cliente hace una conexión socket, el **ServerSocket** devuelve (vía el método **accept( )**) un **Socket** correspondiente a través del cual tendrán lugar las comunicaciones en el lado servidor. A partir de ese momento se tiene una auténtica conexión **Socket** a **Socket** y se tratan ambos extremos de la misma forma, puesto que *son* iguales. En este momento se usan los métodos **getInputStream( )** y **getOutputStream( )** para producir los objetos **InputStream** y **OutputStream** correspondientes de cada **Socket**. Éstos deben envolverse en espacios de almacenamiento intermedio y clases formateadoras exactamente igual que cualquier otro objeto flujo descrito en el Capítulo 11.

El uso del término **ServerSocket** habría parecido ser otro ejemplo de un esquema confuso de nombres en las bibliotecas de Java. Podría pensarse que habría sido mejor denominar al **ServerSocket**, "ServerConnector" o algo sin la palabra "Socket". También se podría pensar que, tanto **ServerSocket** como **Socket,** deberían ser heredados de alguna clase base común. Sin duda, ambas clases tienen varios métodos en común, pero no son suficientes como para darles a ambos una clase base co-

mún. En vez de ello, el trabajo de **ServerSocket** es esperar hasta que se le conecte otra máquina, y devolver después un **Socket**. Por esto, **ServerSocket** parece no tener un nombre muy adecuado para lo que hace, pues su trabajo no es realmente ser un socket, sino construir un objeto **Socket** cuando alguien se conecta al mismo.

Sin embargo, el **ServerSocket** crea un "servidor" físico o socket oyente en la máquina host. Este socket queda esperando a posibles conexiones entrantes y devuelve un socket de "establecimiento" (con los puntos de finalización local y remoto definidos) vía el método **accept( )**. La parte confusa es que estos dos sockets (el oyente y el establecimiento están asociados con el mismo socket servidor. El socket oyente sólo puede aceptar nuevas peticiones de conexión, y no paquetes de datos. Por tanto, mientras **ServerSocket** no tenga mucho sentido desde el punto de vista programático, sí lo tiene "físicamente".

Cuando se crea un **ServerSocket**, sólo se le da un número de puerto. No hay que dar una dirección IP, porque ya está en la máquina que representa. Sin embargo, cuando se crea un **Socket** hay que dar, tanto una dirección IP como un número de puerto al que se está intentando conectar. (Sin embargo, el **Socket** que vuelve de **ServerSocket.accept( )** ya contiene toda esta información.)

Un servidor y cliente simples

Este ejemplo hace el uso más simple de servidores y clientes basados en sockets. Todo lo que hace el servidor es esperar a una conexión, después usa el **Socket** producido por esa conexión para crear un **InputStream** y un **OutputStream**. Éstos se convierten en un **Reader** y un **Writer**, y después envueltos en un **BufferedReader** y un **PrintWriter**. Después de esto, todo lo que se lee de **BufferedReader** se reproduce en el **PrintWriter** hasta que recibe la línea "FIN", momento en el que se cierra la conexión.

El cliente hace la conexión al servidor, después crea un **OutputStream** y lleva a cabo la misma envoltura que el servidor. Las líneas de texto se envían a través del **PrintWriter** resultante. El cliente también crea un **InputStream** (de nuevo, con conversiones y envolturas apropiadas) para escuchar lo que esté diciendo el servidor (que, en este caso, es simplemente una reproducción de las mismas palabras).

Tanto el servidor como el cliente usan el mismo número de puerto, y el cliente usa la dirección del bucle local para conectar al servidor en la misma máquina, con lo que no hay que probarlo a través de la red. (En algunas configuraciones, podría ser necesario estar *conectado* a una red para que este programa funcione, incluso si no se está *comunicando* a través de una red.)

He aquí el servidor:

```
//: c15:ServidorParlante.java
// Servidor muy simple que simplemente
// reproduce lo que envía el cliente.
import java.io.*;
import java.net.*;

public class ServidorParlante {
```

```
  // Elegir un Puerto del rango 1-1024:
  public static final int PUERTO = 8080;
  public static void main(String[] args)
       throws IOException {
    ServerSocket s = new ServerSocket(PUERTO);
    System.out.println("Empezado: " + s);
    try {
      // Se bloquea hasta que se dé alguna conexión:
      Socket socket = s.accept();
      try {
        System.out.println(
          "Conexion aceptada: "+ socket);
        BufferedReader entrada =
          new BufferedReader(
            new InputStreamReader(
              socket.getInputStream()));
        // La salida suele hacerse vaciando a ráfagas
        // por parte de PrintWriter:
        PrintWriter salida =
          new PrintWriter(
            new BufferedWriter(
              new OutputStreamWriter(
                socket.getOutputStream())),true);
        while (true) {
          String str = entrada.readLine();
          if (str.equals("FIN")) break;
          System.out.println("Reproduciendo: " + str);
          salida.println(str);
        }
      // Elegir siempre los dos sockets...
      } finally {
        System.out.println("cerrando...");
        socket.close();
      }
    } finally {
      s.close();
    }
  }
} ///:~
```

Se puede ver que el **ServerSocket** simplemente necesita un número de puerto, no una dirección IP (¡puesto que se está ejecutando en *esta* máquina!). Cuando se llama a **accept( )** el método *se bloquea* hasta que algún cliente intente conectarse al mismo. Es decir, está ahí esperando a una conexión, pero los otros procesos pueden ejecutarse (véase Capítulo 14). Cuando se hace una conexión, **accept( )** devuelve un objeto **Socket** que representa esa conexión.

La responsabilidad de limpiar los sockets se enfoca aquí cuidadosamente. Si falla el constructor **ServerSocket**, el programa simplemente finaliza (nótese que tenemos que asumir que el constructor de **ServerSocket** no deje ningún socket de red abierto sin control si falla). En este caso, **main( ) lanza IOException,** por lo que no es necesario un bloque **try**. Si el constructor **ServerSocket** tiene éxito, hay que guardar otras llamadas a métodos en un bloque **try-finally** para asegurar que, independientemente de cómo se deje el bloque, se cierre correctamente el **ServerSocket**.

Para el **Socket** devuelto por **accept( )** se usa la misma lógica. Si falla **accept( )**, tenemos que asumir que el **Socket** no existe o guarda recursos, por lo que no es necesario limpiarlo. Sin embargo, si tiene éxito, las siguientes sentencias tendrán que estar en un bloque **try-finally,** de forma que si fallan, se seguirá limpiando el **Socket**. Es necesario tener cuidado aquí porque los sockets usan recursos importantes no relacionados con la memoria, por lo que hay que ser diligente para limpiarlos (puesto que no hay ningún destructor que lo haga en Java).

Tanto el **ServerSocket** como el **Socket** producidos por **accept( )** se imprimen a **System.out**. Esto significa que se invoca automáticamente a sus métodos **toString( )**. Éstos producen:

```
ServerSocket[addr=0.0.0.0,PORT=0,localport=8080]
Socket[addr=127.0.0.1,PORT=1077,localport=8080]
```

La siguiente parte del programa parece simplemente como si sólo se abrieran y escribieran archivos, de no ser porque el **InputStream** y **OutputStream** se crean a partir del objeto **Socket**. Tanto el objeto **InputStream** como el objeto **OutputStream** se convierten a objetos **Reader** y **Writer** usando las clases "conversoras" **InputStreamReader** y **OutputStreamWriter**, respectivamente. También se podrían haber usado directamente las clases **InputStream** y **OutputStream** de Java 1.0, pero con la salida, hay una ventaja añadida al usar el enfoque **Writer**. Éste aparece con **PrintWriter**, que tiene un constructor sobrecargado que toma un segundo argumento, un indicador **boolean** que indica si hay que vaciar automáticamente la salida al final de cada sentencia **println( )** (pero *no* **print( )**). Cada vez que se escribe a **out**, hay que vaciar su espacio de almacenamiento intermedio, de forma que la información fluya por la red. El vaciado es importante en este ejemplo particular, porque tanto el cliente como el servidor, esperan ambos a una línea proveniente de la otra parte antes de proceder. Si no se da el vaciado, la información no saldrá a la red hasta llenar el espacio de almacenamiento intermedio, causando muchos problemas en este ejemplo.

Cuando se escriban programas de red hay que tener cuidado con el uso del vaciado automático. Cada vez que se vacía el espacio de almacenamiento intermedio, hay que crear y enviar un paquete. En este caso, eso es exactamente lo que queremos, pues si no se envía el paquete que contiene la línea, se detendría el intercambio amistoso bidireccional entre el cliente y el servidor. Dicho de otra forma, el final de línea es el fin de cada mensaje. Pero en muchos casos, los mensajes no están delimitados por líneas, por lo que es mucho más eficiente no usar el autovaciado y dejar en su lugar que el espacio de almacenamiento intermedio incluido decida cuándo construir y enviar un paquete. De esta forma se pueden enviar paquetes mayores, y el procesamiento será más rápido.

Nótese que, como ocurre con casi todos los flujos que se abren, éstos tienen sus espacios de almacenamiento intermedio. Hay un ejercicio al final de este capítulo para mostrar al lector lo que ocurre si no se utilizan los espacios de almacenamiento intermedio (todo se ralentiza).

El bucle **while** infinito lee líneas del **BufferedReader entrada** y escribe información a **System.out** y al **PrintWriter salida**. Nótese que **entrada** y **salida** podrían ser cualquier flujo, lo que ocurre simplemente es que están conectados a la red.

Cuando el cliente envía la línea "FIN", el programa sale del bucle y cierra el **Socket**.

He aquí el cliente:

```
//: c15:ClienteParlante.java
// Cliente muy simple que sólo envía
// líneas al servidor y lee líneas
// que el servidor envía.
import java.net.*;
import java.io.*;

public class ClienteParlante {
  public static void main(String[] args)
      throws IOException {
    // Pasar null a getByName() genera la dirección
    // IP especial "Loopback Local", que permite
    // hacer pruebas en una máquina con o sin red:
    InetAddress addr =
      InetAddress.getByName(null);
    // De forma alternativa, puedes usar
    // la dirección o nombre:
    // InetAddress addr =
    //     InetAddress.getByName("127.0.0.1");
    // InetAddress addr =
    //     InetAddress.getByName("localhost");
    System.out.println("addr = " + addr);
    Socket socket =
      new Socket(addr, ServidorParlante.PUERTO);
    // Guardar todo en un try-finally para asegurarse
    // de que se cierra el socket:
    try {
      System.out.println("socket = " + socket);
      BufferedReader entrada =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // La salida es vaciada automáticamente
      // por PrintWriter:
      PrintWriter salida =
        new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
```

```
                socket.getOutputStream())),true);
        for(int i = 0; i < 10; i ++) {
          salida.println("Hola " + i);
          String str = entrada.readLine();
          System.out.println(str);
        }
        salida.println("FIN");
      } finally {
        System.out.println("cerrando...");
        socket.close();
      }
    }
} ///:~
```

En el método **main( )** se pueden ver las tres formas de producir la **InetAddress** de la dirección IP del bucle local: utilizando **null**, **localhost** o la dirección explícitamente reservada **127.0.0.1**. Por supuesto, si se desea establecer una conexión con una máquina que está en la red, deberá sustituirse esa dirección por la dirección IP de la máquina. Al imprimir la **InetAddress addr** (vía la llamada automática a su método **toString( )**) el resultado es:

```
localhost/127.0.0.1
```

Pasando a **getByName( )** un **null**, éste encontraba por defecto el **localhost**, lo que producía la dirección especial **127.0.0.1**.

Nótese que el **Socket** de nombre **socket** se crea tanto con la **InetAddress** como con el número de puerto. Para entender lo que significa imprimir uno de estos objetos **Socket**, recuérdese que una conexión a Internet viene determinada exclusivamente por estos cuatro fragmentos de datos: **clientHost**, **clientPortNumber**, **serverHost** y **serverPortNumber**. Cuando aparece un servidor, toma su puerto asignado (8080) en el bucle local (127.0.0.1). Al aparecer el cliente, se le asigna el siguiente puerto disponible en la máquina, en este caso el 1077, que resulta estar en la misma máquina que el servidor. Ahora, para que los datos se muevan entre el cliente y el servidor, cada uno de los extremos debe saber a dónde enviarlos. Por consiguiente, durante el proceso de conexión al servidor "conocido", el cliente envía una "dirección de retorno" de forma que el servidor pueda saber a dónde enviar sus datos. Esto es lo que se aprecia en la salida ejemplo para el lado servidor:

```
Socket[addr=127.0.0.1,port=1077,localport=8080]
```

Esto significa que el servidor acaba de aceptar una conexión de 127.0.0.1 en el puerto 1077 al escuchar en su puerto local (8080). En el lado cliente:

```
Socket[addr=localhost/127.0.0.1,PORT=8080,localport=1077]
```

lo que significa que el cliente hizo una conexión con la 127.0.0.1 en el puerto 8080 usando el puerto local 1077.

Se verá que cada vez que se arranca de nuevo el cliente, se incrementa el número de puerto local. Empieza en el 1025 (uno más del bloque de puertos reservados) y va creciendo hasta volver a arrancar la máquina, momento en el que vuelve a empezar en el 1025. (En las máquinas UNIX, una vez que se llega al límite superior del rango de sockets, los números vuelven también al número mínimo disponible.)

Una vez que se ha creado el objeto **Socket**, el proceso de convertirlo en un **BufferedReader** y en un **PrintWriter** es el mismo que en el servidor (de nuevo, en ambas ocasiones se empieza con un **Socket**). Aquí, el cliente inicia la conversación enviando la cadena "hola" seguida de un número. Nótese que es necesario volver a vaciar el espacio de almacenamiento intermedio (cosa que ocurre automáticamente vía el segundo argumento al constructor **PrintWriter**). Si no se vacía el espacio de almacenamiento intermedio, se colgaría toda la conversación, pues nunca se llegaría a enviar el primer "hola" (el espacio de almacenamiento intermedio no estaría lo suficientemente lleno como para que se produjera automáticamente un envío). Cada línea que se envíe de vuelta al servidor se escribe en **System.out** para verificar que todo está funcionando correctamente. Para terminar la conversación, se envía el "FIN" preacordado. Si el cliente simplemente cuelga, el servidor enviará una excepción.

Se puede ver que en el ejemplo se tiene el mismo cuidado para asegurar que los recursos de red representados por el **Socket** se limpien adecuadamente, mediante el uso de un bloque **try-finally**.

Los sockets producen una conexión "dedicada" que persiste hasta que sea desconectada explícitamente. (La conexión dedicada puede seguir siendo desconectada de forma no explícita si se cuelga alguno de los lados o intermediarios.) Esto significa que las dos partes están bloqueadas en la comunicación, estando la conexión constantemente abierta. Esto parece un enfoque lógico para las redes, pero añade algo de sobrecarga a la propia red. Más adelante, en este mismo capítulo se verá un enfoque distinto al funcionamiento en red, en el que las conexiones son únicamente temporales.

Servir a múltiples clientes

El **ServidorParlante** funciona, pero solamente puede manejar a un cliente en cada momento. En un servidor típico, se deseará poder tratar con muchos clientes simultáneamente. La respuesta es el multihilo, y en los lenguajes que no lo soportan directamente esto significa todo tipo de complicaciones. En el Capítulo 14 se vio que en Java el multihilo es tan simple como se pudo, considerando que es de por sí un aspecto complejo. Dado que la capacidad de usar hilos en Java es bastante directa, construir un servidor que gestione múltiples clientes es relativamente sencillo.

El esquema básico es construir un **ServerSocket** único en el servidor, e invocar a **accept( )** para que espere a una nueva conexión. Cuando **accept( )** devuelva un **Socket**, se toma éste y se usa para crear un nuevo hilo cuyo trabajo es servir a ese cliente en particular. Después, se llama de nuevo a **accept( )** para esperar a un nuevo cliente.

En el siguiente código servidor, puede verse que tiene una apariencia similar a la de **ServidorParlante.java**, excepto en que todas las operaciones que sirven a un cliente en particular han quedado desplazadas al interior de una clase hilo separada:

```
//: c15:ServidorMultiParlante.java
// Un servidor que usa el multihilo
// para manejar cualquier número de clientes.
import java.io.*;
import java.net.*;

class ServirUnParlante extends Thread {
  private Socket socket;
  private BufferedReader entrada;
  private PrintWriter salida;
  public servirUnParlante(Socket s)
      throws IOException {
    socket = s;
    entrada =
      new BufferedReader(
        new InputStreamReader(
          socket.getInputStream()));
    // Habilitar el autovaciado:
    salida =
      new PrintWriter(
        new BufferedWriter(
          new OutputStreamWriter(
            socket.getOutputStream())), true);
    // Si cualquiera de las llamadas de arriba lanza una
    // excepción, el llamador es responsable de
    // cerrar el socket. De lo contrario lo cerrara
    // el hilo.
    start(); // Llama a run()
  }
  public void run() {
    try {
      while (true) {
        String str = entrada.readLine();
        if (str.equals("FIN")) break;
        System.out.println("Haciendo eco: " + str);
        salida.println(str);
      }
      System.out.println("cerrando...");
    } catch(IOException e) {
      System.err.println("Excepcion E/S");
    } finally {
      try {
        socket.close();
      } catch(IOException e) {
        System.err.println("Socket sin cerrar");
```

```
        }
      }
    }
}

public class ServidorMultiParlante {
  static final int PUERTO = 8080;
  public static void main(String[] args)
      throws IOException {
    ServerSocket s = new ServerSocket(PUERTO);
    System.out.println("Servidor iniciado");
    try {
      while(true) {
        // Se bloquea hasta que se da una conexión:
        Socket socket = s.accept();
        try {
          new ServirUnParlante(socket);
        } catch(IOException e) {
          // Si falla, cerrar el socket,
          // si no, lo cerrará el hilo:
          socket.close();
        }
      }
    } finally {
      s.close();
    }
  }
} ///:~
```

El hilo **ServirUnParlante** toma el objeto **Socket** producido por **accept( )** en el método **main( )** cada vez que un cliente nuevo haga una conexión. Posteriormente, y como antes, crea un **BufferedReader** y un objeto **PrintWriter** con autovaciado utilizando el **Socket**. Finalmente, llama al método **start( )** especial de **Thread**, que lleva a cabo la inicialización de hilos e invoca después a **run( )**. Éste hace el mismo tipo de acción que en el ejemplo anterior: leer algo del socket y después reproducirlo de vuelta hasta leer la señal especial "FIN".

La responsabilidad de cara a limpiar el socket debe ser diseñada nuevamente con cuidado. En este caso, el socket se crea fuera de **ServirUnParlante**, por lo que es posible compartir esta responsabilidad. Si el constructor **ServirUnParlante** falla, simplemente lanzará la excepción al llamador, que a continuación limpiará el hilo. Pero si el constructor tiene éxito, será el objeto **ServirUnParlante** el que tome la responsabilidad de limpiar el hilo, en su **run( )**.

Nótese la simplicidad del **ServidorMultiParlante**. Como antes, se crea un **ServerSocket** y se invoca a **accept( )** para permitir una nueva conexión. Pero en esta ocasión, se pasa el valor de retorno de **accept( )** (un **Socket**) al constructor de **ServirUnParlante**, que crea un nuevo hilo para manejar esa conexión. Cuando acaba la conexión, el hilo simplemente desaparece.

Si falla la creación de **ServerSocket**, se lanza de nuevo la excepción a través del método **main( )**. Pero si la creación tiene éxito, el **try-finally** externo garantiza su limpieza. El **try-catch** interno simplemente previene del fallo del constructor de **ServirUnParlante**; si el constructor tiene éxito, el hilo **ServirUnParlante** cerrará el socket asociado.

Para probar que el servidor verdaderamente gestiona múltiples clientes, he aquí el siguiente programa, que crea muchos clientes (usando hilos) que se conectan al mismo servidor. El máximo número de hilos permitido viene determinado por **final int MAX_HILOS**.

```
//: c15:ClienteMultiParlante.java
// Cliente que prueba el ServidorMultiParlante
// arrancando múltiples clientes.
import java.net.*;
import java.io.*;

class HiloClienteParlante extends Thread {
  private Socket socket;
  private BufferedReader entrada;
  private PrintWriter salida;
  private static int contador = 0;
  private int id = contador++;
  private static int conteoHilos = 0;
  public static int conteoHilos() {
    return conteoHilos;
  }
  public HiloClienteParlante(InetAddress addr) {
    System.out.println("Construyendo el cliente " + id);
    conteoHilos++;
    try {
      socket =
        new Socket(addr, HiloClienteParlante.PUERTO);
    } catch(IOException e) {
      System.err.println("Fallo el Socket ");
      // Si falla la creación del socket,
      // no hay que limpiar nada.
    }
    try {
      entrada =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Habilitar el auto-vaciado:
      salida =
        new PrintWriter(
          new BufferedWriter(
```

```
                new OutputStreamWriter(
                  socket.getOutputStream())), true);
        start();
      } catch(IOException e) {
        // Debería cerrarse el socket al darse
        // cualquier otro fallo distinto del de
        // constructor:
        try {
          socket.close();
        } catch(IOException e2) {
          System.err.println("Socket sin cerrar");
        }
      }
      // De otra forma, el socket se cerrará en el
      // método run() del hilo.
    }
    public void run() {
      try {
        for(int i = 0; i < 25; i++) {
          salida.println("Cliente " + id + ": " + i);
          String str = entrada.readLine();
          System.out.println(str);
        }
        salida.println("FIN");
      } catch(IOException e) {
        System.err.println("Excepcion de E/S");
      } finally {
        // Cerrarlo siempre:
        try {
          socket.close();
        } catch(IOException e) {
          System.err.println("Socket sin cerrar");
        }
        conteoHilos--; // Acabando este hilo
      }
    }
}

public class ClienteMultiParlante {
  static final int MAX_HILOS = 40;
  public static void main(String[] args)
      throws IOException, InterruptedException {
    InetAddress addr =
      InetAddress.getByName(null);
    while(true) {
```

```
        if(HiloClienteParlante.conteoHilos()
            < MAX_HILOS)
          new HiloClienteParlante(addr);
        Thread.currentThread().sleep(100);
      }
    }
} ///:~
```

El constructor **HiloClienteParlante** toma un **InetAddress** y lo usa para abrir un **Socket**. Probablemente se empiece a ver el patrón: siempre se usa el **Socket** para crear algún tipo de objeto **Reader** y/o **Writer** (o **InputStream** y/u **OutputStream**), que es la única forma de usar el **Socket**. (Por supuesto, puede escribir una o dos clases que automaticen este proceso en vez de llevar a cabo todo el tecleado cuando éste se vuelve molesto.)De nuevo, **start( )** lleva a cabo la inicialización de hilos e invoca a **run( )**. Aquí, se envían los mensajes al servidor y se reproduce esa información proveniente del servidor en la pantalla. Sin embargo, el hilo tiene un tiempo de vida limitado y suele acabarse. Nótese que el socket se limpia si falla el constructor después de haber creado el socket, pero antes de que acabe el constructor. De otra forma, la responsabilidad de llamar al **close( )** para el socket se relega al método **run( )**.

El **conteoHilos** mantiene un seguimiento de cuántos objetos **HiloClienteParlante** existen en cada momento. Se incrementa como parte del constructor y se disminuye al acabar **run( )** (lo que significa que el hilo está finalizando). En **ClienteMultiParlante.main( )** puede verse que se prueba el número de hilos, y si hay demasiados no se crean más. Después, el método se duerme. De esta forma, algunos hilos acabarían eventualmente y se podrían crear más. Se puede experimentar con **MAX_HILOS** para ver dónde empieza a tener problemas un sistema en particular por la existencia de demasiadas conexiones.

Datagramas

Los ejemplos vistos hasta el momento usan el *Transmission Control Protocol* (TCP, conocido también como *sockets basados en flujos*), diseñado para garantizar la fiabilidad de manera que la información llegue siempre. Permite la retransmisión de datos perdidos, proporciona múltiples caminos a través de distintos enrutadores en caso de que uno falle, y los bytes se entregan en el mismo orden en que son enviados. Todo este control y fiabilidad tiene un coste: TCP supone una gran sobrecarga.

Hay un segundo protocolo, denominado *User Datagram Protocol* (UDP), que no garantiza que los paquetes se entreguen y que lleguen en el orden en que son enviados. Se llama "protocolo no orientado a la conexión" (TCP es un "protocolo orientado a la conexión"), lo cual no suena bien, pero dado que es muchísimo más rápido, en ocasiones es útil. Hay algunas aplicaciones, como una señal de audio, en las que no es crítico el hecho de que se puedan perder algunos paquetes, pero la velocidad es vital. O considérese un servidor de hora del día, en el que no importa verdaderamente el que se pierda algún mensaje. Además, algunas aplicaciones deberían ser capaces de disparar un mensaje UDP al servidor para asumir con posterioridad, si no hay respuesta en un periodo razonable de tiempo, que el mensaje se ha perdido.

Generalmente, se trabajará directamente programando bajo TCP, y sólo ocasionalmente se hará uso de UDP. Hay un tratamiento de UDP más completo, incluyendo un ejemplo, en la primera edición de este libro (disponible en el CD ROM adjunto a este libro, o como descarga gratuita de *http://www.BruceEckel.com*).

Utilizar URL en un applet

Es posible que un *applet* pueda mostrar cualquier URL a través del navegador web en el que se esté ejecutando. Esto se puede lograr con la línea:

```
getAppletContext().showDocument(u);
```

donde **u** es el objeto **URL**. He aquí un ejemplo simple que le remite a otra página web. Aunque simplemente se logra una redirección a una página HTML, también se podría remitir a la salida de un programa CGI.

```
//: c15:MostrarHTML.java
// <applet code=MostrarHTML width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class MostrarHTML extends JApplet {
  JButton enviar = new JButton("Ir");
  JLabel l = new JLabel();
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    enviar.addActionListener(new Al());
    cp.add(enviar);
    cp.add(l);
  }
  class Al implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
      try {
        // Esto podría ser un programa CGI en vez de
        // una página HTML.
        URL u = new URL(getDocumentBase(),
          "MarcoBuscador.html");
        // Mostrar la salida de la URL usando
        // el navegador web, como una página ordinaria:
```

```
            getAppletContext().showDocument(u);
          } catch(Exception e) {
            l.setText(e.toString());
          }
        }
      }
      public static void main(String[] args) {
        Console.run(new MostrarHTML(), 100, 50);
      }
} ///:~
```

La belleza de la clase **URL** radica en cuánto hace por ti. Es posible conectarse a servidores web sin saber mucho o nada de lo que está ocurriendo verdaderamente.

Leer un archivo de un servidor

Una variación del programa anterior lee un archivo ubicado en el servidor. En este caso, el archivo es especificado por el cliente:

```
//: c15:Buscador.java
// <applet code=Buscador width=500 height=300>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class Buscador extends JApplet {
  JButton buscarlo= new JButton("Coger los Datos");
  JTextField f =
    new JTextField("Buscador.java", 20);
  JTextArea t = new JTextArea(10,40);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    buscarlo.addActionListener(new BuscarL());
    cp.add(new JScrollPane(t));
    cp.add(f); cp.add(buscarlo);
  }
  public class BuscarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      try {
        URL url = new URL(getDocumentBase(),
          f.getText());
```

```
            t.setText(url + "\n");
            InputStream is = url.openStream();
            BufferedReader entrada = new BufferedReader(
              new InputStreamReader(is));
            String linea;
            while ((linea = entrada.readLine()) != null)
              t.append(linea + "\n");
          } catch(Exception ex) {
            t.append(ex.toString());
          }
        }
      }
      public static void main(String[] args) {
        Console.run(new Buscador(), 500, 300);
      }
} ///:~
```

La creación del objeto **URL** es similar al ejemplo anterior —**getDocumentBase( )** es como antes el punto de comienzo, pero en esta ocasión, el nombre del archivo se lee de **JTextField**. Una vez que se crea el objeto **URL**, se ubica su versión **String** en el **JTextArea,** de forma que se pueda ver la apariencia que tiene. Después, se proporciona un **InputStream** desde el **URL**, que en este caso simplemente producirá un flujo de los caracteres del archivo. Después de convertir a un **Reader**, y del tratamiento por espacios de almacenamiento intermedio, se lee cada línea para ser añadida al **JTextArea**. Nótese que el **JTextArea** se ha ubicado dentro de un **JScrollPane,** de forma que todo el desplazamiento de pantallas se gestiona automáticamente.

Más aspectos de redes

Además de lo cubierto en este tratamiento introductorio hay, de hecho, bastantes más aspectos relacionados con las redes. La capacidad de red de Java también proporciona gran soporte para URL, incluyendo gestores de protocolos para distintos tipos de contenidos que pueden descubrirse en las distintos sitios Internet. Se pueden encontrar más facetas de tratamiento de red de Java descritas con gran detalle en *Java Network Programming*, de Elliote Rusty Harold (O'Reilly, 1997).

Conectividad a Bases de Datos de Java (JDBC)

Se ha estimado que la mitad de todos los desarrollos software involucran aplicaciones cliente/servidor. Una gran promesa de Java ha sido la habilidad de construir aplicaciones de base de datos cliente/servidor independientes de la plataforma. En realidad esto se ha conseguido gracias a la Conectividad a Bases de Datos de Java (JDBC, *Java DataBase Connectivity*).

Uno de los mayores problemas de las bases de datos ha sido la guerra de características entre las propias compañías de bases de datos. Hay un lenguaje de bases de datos "estándar", el *Structure Query Languaje* (SQL-92), pero probablemente todo el mundo conoce el proveedor de la base de datos con la que trabaja a pesar del estándar. JDBC está diseñado para ser independiente de la plataforma, por lo que en tiempo de programación no hay que preocuparse por la base de datos que se esté utilizando. Sin embargo, sigue siendo posible construir llamadas específicas de ese fabricante desde JDBC, por lo que no se puede decir que haya restricción alguna a la hora de hacer lo que se tenga hacer.

Un lugar en el que los programadores pueden necesitar usar nombres del tipo SQL es en la sentencia `TABLE CREATE` de SQL al crear una nueva tabla de base de datos y definir el tipo SQL de cada columna. Desgraciadamente hay bastantes variaciones entre los tipos de SQL soportados por los distintos productos de base de datos. Bases de datos distintas que soportan tipos SQL con la misma semántica y estructura puede que den a esos tipos distintos nombres. La mayoría de las bases de datos principales soportan un tipo de datos SQL para valores binarios grandes: en Oracle, a este tipo se le denomina `LONG RAW`, Sybase lo llama `IMAGE`, Informix lo llama `BYTE`, y DB2 le llama `LONG VARCHAR FOR BIT DATA`. Por consiguiente, si la portabilidad de la base de datos es una de las metas a lograr, debería intentarse utilizar exclusívamente identificadores de tipos SQL genéricos.

La portabilidad es importante al escribir para un libro en el que los lectores pueden estar probando los ejemplos con cualquier tipo de almacén de datos desconocido. Hemos intentado escribir estos ejemplos para que sean lo más portables posible. El lector también podría darse cuenta de que se ha aislado el código específico de la base de datos para centralizar cualquier cambio que haya que realizar para hacer que estos ejemplos sean operativos en algún otro entorno.

JDBC, como muchas de las API de Java, está diseñado persiguiendo la simplicidad. Las llamadas a métodos que se hagan se corresponden con las operaciones lógicas que uno pensaría que debe hacer al recopilar datos desde la base de datos: conectarse a la base de datos, crear una sentencia y ejecutar la petición, y tener acceso al conjunto resultado.

Para permitir esta independencia de la plataforma, JDBC proporciona un *gestor de controladores* que mantiene dinámicamente todos los objetos controlador que puedan requerir las consultas a la base de datos. Por tanto, si se tienen tres tipos distintos de bases de datos a las que se desea establecer comunicación, se necesitarán tres objetos controlador. Estos objetos se registran a sí mismos en el gestor de controladores en el momento de su carga, y se puede forzar esta carga utilizando **Class.forName( )**.

Para abrir una base de datos, hay que crear un "URL de base de datos" que especifica:

1. Que se está usando JDBC con "jdbc".
2. El "subprotocolo": el nombre del controlador o el nombre de un mecanismo de conectividad de base de datos. Puesto que el diseño de JDBC se inspiró en ODBC, el primer subprotocolo disponible es el puente "jdbc-odbc", denominado "odbc".
3. El identificador de la base de datos. Éste varía en función del controlador de base de datos que se use, pero generalmente proporciona un nombre lógico al que el software administrador de la base de datos vincula a un directorio físico en el que están almacenadas las tablas de la base

de datos. Para que un identificador de base de datos tenga algún significado hay que registrar el nombre utilizando el software de administración de base de datos. (Este proceso de registro varía de plataforma a plataforma.)

Toda esta información se combina en un String, el "URL de base de datos". Por ejemplo, para conectarse a través del subprotocolo ODBC a una base de datos denominada "gente", el URL de la base de datos podría ser:

```
String dbUrl = "jdbc:odbc:gente";
```

Si se está estableciendo una conexión a través de una red, el URL de la base de datos debe contener la información de conexión que identifique la máquina remota, pudiendo resultar algo intimidatorio. He aquí un ejemplo de una base de datos Fuga a la que se invoca desde un cliente remoto utilizando RMI:

```
jdbc:rmi://192.168.170.27:1099/jdbc:fuga:db
```

La URL de la base de datos es verdaderamente dos llamadas a jdbc en una. La primera parte, `"jdbc:rmi://192.168.170.27:1099/"` usa RMI para hacer la conexión al motor de base de datos remota escuchando en el puerto 1099 de la dirección IP 192.168.170.27. La segunda parte del URL, `"jdbc:fuga:db"` conlleva los ajustes más típicos al usar el subprotocolo y el nombre de la base de datos, pero esto sólo ocurrirá una vez que la primera sección haya hecho la conexión vía RMI a la máquina remota.

Cuando uno está listo para conectarse a la base datos, hay que invocar al método **static DriverManager.getConnection( )** y pasarle el URL de la base de datos, el nombre de usuario y la contraseña para entrar en la base de datos. A cambio, se recibe un objeto **Connection** que puede ser usado posteriormente para preguntar y manipular la base de datos.

El ejemplo siguiente abre una base de datos de información de contacto y busca el apellido de una persona, suministrado en línea de comandos. Sólo selecciona los nombres de la gente que tienen dirección de correo electrónico, y después imprime todos los que casen con el apellido suministrado:

```
//: c15:jdbc:Buscar.java
// Busca direcciones de correo electrónico
// en una base de datos local usando JDBC.
import java.sql.*;

public class Buscar {
  public static void main(String[] args)
  throws SQLException, ClassNotFoundException {
    String dbUrl = "jdbc:odbc:gente";
    String usuario = "";
    String contrasena = "";
    // Cargar el controlador (se registra solo)
    Class.forName(
      "sun.jdbc.odbc.JdbcOdbcDriver");
```

```
    Connection c = DriverManager.getConnection(
      dbUrl, usuario, contrasena);
    Statement s = c.createStatement();
    // Código SQL:
    ResultSet r =
      s.executeQuery(
        "SELECT PRIMERO, SEGUNDO, CORREO " +
        "FROM gente.csv gente " +
        "WHERE " +
        "(LAST='" + args[0] + "') " +
        " AND (correo Is Not Null) " +
        "ORDER BY FIRST");
    while(r.next()) {
      // El uso de mayúsculas no importa:
      System.out.println(
        r.getString("Ultimo") + ", "
        + r.getString("pRIMERO")
        + ": " + r.getString("CORREO") );
    }
    s.close(); // También cierra el Resultset
  }
} ///:~
```

Puede verse que la creación del URL de la base de datos se hace como se describió anteriormente. En este ejemplo, no hay protección por contraseñas en la base de datos, por lo que los campos nombre de usuario y contraseña son cadenas de caracteres vacías.

Una vez que se hace la conexión con **DriverManager.getConnection( )**, se puede usar el objeto **Connection** resultante para crear un objeto **Statement** utilizando el método **createStatement( )**. Con el **Statement** resultante, se puede llamar a **executeQuery( )**, pasándole una cadena de caracteres que contenga cualquier sentencia SQL compatible con el estándar SQL-92. (En breve se verá cómo se puede generar esta sentencia automáticamente, evitando tener que saber mucho SQL.)

El método **executeQuery( )** devuelve un objeto **ResultSet**, que es un iterador: el método **next( )** mueve el iterador al siguiente recurso de la sentencia, o devuelve **false** si se ha alcanzado el fin del conjunto resultado. Una ejecución de **executeQuery( )** siempre genera un objeto **ResultSet** incluso si la solicitud conlleva un conjunto vacío (es decir, no se lanza una excepción). Nótese que hay que llamar a **next( )** una vez, por lo menos antes de intentar leer ningún registro. Si el conjunto resultado está vacío, esta primera llamada a **next( )** devolverá **false**. Por cada registro del conjunto resultado, es posible seleccionar los campos usando (entre otros enfoques) el nombre del campo como cadena de caracteres. Nótese también que se ignora el uso de mayúsculas o minúsculas en el nombre del campos —con una base de datos SQL no importa. El tipo de dato que se devuelve se determina invocando a **getInt( )**, **getString( )**, **getFloat( )**, etc. En este momento, se tienen los datos de la base de datos en formato Java nativo y se puede hacer lo que se desee con ellos usando código Java ordinario.

Hacer que el ejemplo funcione

Con JDBC, entender el código es relativamente fácil. La parte complicada es hacer que funcione en un sistema en particular. La razón por la que es complicada es que requiere que cada lector averigüe cómo cargar adecuadamente su controlador JDBC, y cómo configurar una base de datos utilizando su software de administración de base de datos. Por supuesto, este proceso puede variar radicalmente de máquina a máquina, pero el proceso que sabíamos usar para que funcionara bajo un Windows de 32 bits puede dar algunas pistas para que cada uno se enfrente a su propia situación:

Paso 1: Encontrar el Controlador JDBC

El programa de arriba contiene la sentencia:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Ésta implica una estructura de directorios que es determinante. Con esta instalación particular del JDK 1.1, *no había ningún archivo* llamado **JdbcOdbcDriver.class**, por lo que si se echa un vistazo a este ejemplo y se empieza a buscarlo, uno se frustra. Otros ejemplos publicados usan un pseudonombre, como "miControlador.ClassName", que es aún menos útil. De hecho, la sentencia de carga de arriba para el controlador jdbc-odbc (el único que, de hecho, viene con el JDK) sólo aparece en unos pocos sitios en la documentación en línea (en particular, en una página de nombre "JDBC-ODBC Bridge Driver"). Si la sentencia de carga de arriba no funciona, entonces puede que el nombre haya cambiado como parte de los cambios de versión de Java, por lo que habría que volver a recorrerse toda la documentación.

Si la sentencia de carga está mal, se obtendrá justo en ese momento una excepción. Para probar si la sentencia de carga del controlador está funcionando correctamente o no, puede marcarse como comentario la sentencia; si el programa no lanza excepciones, será señal de que el controlador se está cargando correctamente.

Paso 2: Configurar la base de datos

Esto es, de nuevo, específico al Windows de 32 bits; puede ser que alguien tenga que investigar si está trabajando con otra plataforma.

En primer lugar, abra el panel de control. Se verá que hay dos iconos que dicen "ODBC". Hay que usar el que dice "Fuentes de Datos ODBC 32 bits", puesto que el otro es para lograr retrocompatibilidad con software ODBC de 16 bits y no generará ningún resultado en el caso de JDBC. Cuando se abre el icono "Fuentes de Datos ODBC 32 bits", se verá una caja de diálogo con solapas, entre las que se incluyen "DSN de Usuario", "DSN de Sistema", "DSN de Archivo", etc., donde DSN significa "*Data Source Name*" ("Nombre de la fuente de datos"). Resulta que para el puente JDBC-ODBC, el único lugar en el que es importante configurar la base de datos es "DSN del Sistema", pero también se deseará probar la configuración y hacer consultas, y para ello también será necesario configurar la base de datos en "DSN de Archivo". Esto permitirá a la herramienta Microsoft Query (que viene con Microsoft Office) encontrar la base de datos. Nótese que existen también otras herramientas de otros vendedores.

La base de datos más interesante es una que esté actualmente en uso. El ODBC estándar soporta varios formatos de archivos distintos, incluyendo algunos tan venerables como DBase. Sin embargo, también incluye el formato simple "ASCII separado por comas", que generalmente toda herramienta de datos tiene la capacidad de escribir. Simplemente tomamos la base de datos de "gente", mantenida por nosotros durante años usando herramientas de gestión de contactos, y la exportamos a un archivo ASCII separado por comas (que suelen tener extensión **.csv**). En la sección "DSN de sistema", seleccionamos "Agregar", elegimos el controlador de texto para gestionar nuestro archivo ASCII separado por comas, y después deseleccionamos "usar directorio actual" para que me permitiera especificar el directorio al que exportar el archivo de datos.

Se verá al hacer esto que verdaderamente no se especifica un archivo, sólo un directorio. Eso es porque una base de datos suele representarse como una colección de archivos bajo un único directorio (aunque podría estar representada también de otra forma). Cada archivo suele contener una única tabla, y las sentencias SQL pueden producir resultados provenientes de múltiples tablas de la base de datos (a esto se le llama *join*). A una base de datos que sólo contiene una base de datos (como mi base de datos "gente") se le suele llamar *flat-file database*. La mayoría de problemas que van más allá del simple almacenamiento y recuperación de datos suelen requerir de varias tablas, que deben estar relacionadas mediante *joins* para producir los resultados deseados, y a éstas se las denomina bases de datos *relacionales*.

Paso 3: Probar la configuración

Para probar la configuración, será necesario disponer de alguna forma de descubrir si la base de datos es visible desde un programa que hace consultas a la misma. Por supuesto, se puede ejecutar simplemente el programa ejemplo JDBC de arriba, incluyendo la sentencia:

```
Connection c = DriverManager.getConnection(
  dbUrl, usuario, contrasena);
```

Si se lanza una excepción, la configuración era incorrecta.

Sin embargo, es útil hacer que una herramienta de generación de consultas se vea involucrada en esto. Utilizamos Microsoft Query, que viene con Microsoft Office, pero podría preferirse alguna otra. La herramienta de consultas debería saber dónde está la base de datos y Microsoft Query exigía que fuera al campo Administrador de ODBC de la etiqueta "DSN de Archivo" y añadiera una nueva entrada ahí, especificando de nuevo el controlador de texto y el directorio en el que reside nuestra base de datos. Se puede nombrar a la entrada como se desee, pero es útil usar el mismo nombre usado en "DSN de Sistema".

Una vez hecho esto se verá que la base de datos está disponible al crear una nueva consulta usando la herramienta de consultas.

Paso 4: Generar la consulta SQL

La consulta que creamos usando Microsoft Query no sólo nos mostraba que la base de datos estaba ahí y en orden correcto, sino que también creaba automáticamente el código SQL que necesitaba insertar en nuestro programa Java. Queríamos una consulta que buscara los registros que tuvie-

ran el mismo apellido que se tecleara en la línea de comandos al arrancar el programa Java. Por tanto, y como punto de partida, buscamos un apellido específico, "Eckel". También queríamos que se mostraran sólo aquellos nombres que tuvieran alguna dirección de correo electrónico asociada. Los pasos que seguimos para crear esta consulta fueron:

1. Empezar una nueva consulta y utilizar el Query Wizard. Seleccionar la base de datos "gente". (Esto equivale a abrir la conexión de base de datos usando el URL de base de datos apropiado).
2. Seleccionar la tabla "gente" dentro de la base de datos. Desde dentro de la tabla, seleccionar las columnas PRIMERO, SEGUNDO y CORREO.
3. Bajo "Filtro de Datos", seleccionar SEGUNDO y elegir "equals" con un argumento "Eckel". Hacer clic en el botón de opción "And".
4. Seleccionar CORREO y elegir "Is not Null".
5. Bajo "Sort by", seleccionar PRIMERO.

Los resultados de esta consulta mostrarán si se está obteniendo lo deseado.

Ahora se puede presionar el botón SQL y sin hacer ningún tipo de investigación, aparecerá el código SQL correcto, listo para ser cortado y pegado. Para esta consulta, sería algo así:

```
SELECT gente.PRIMERO, gente.SEGUNDO, gente.CORREO
FROM gente.csv gente
WHERE (gente.SEGUNDO='Eckel') AND
(gente.CORREO Is Not Null)
ORDER BY gente.PRIMERO
```

Es posible que las cosas vayan mal, especialmente si las consultas son más complicadas, pero usando una herramienta de generación de consultas, se pueden probar éstas de forma interactiva y generar automáticamente el código correcto. Es difícil defender la postura de hacer esto a mano.

Paso 5: Modificar y cortar en una consulta

Se verá que el código de arriba tiene distinto aspecto del que se usó en el programa. Eso es porque la herramienta de consultas usa especificación completa de todos los nombres, incluso cuando sólo esté involucrada una tabla. (Cuando hay más de una tabla, la especificación completa evita colisiones entre columnas de distintas tablas que pudieran tener igual nombre.) Puesto que esta consulta simplemente involucra a una tabla, se puede eliminar el especificador "gente" de la mayoría de los nombres, así:

```
SELECT PRIMERO,SEGUNDO,CORREO
FROM gente.csv gente
WHERE (SEGUNDO='Eckel') AND
(EMAIL Is Not Null)
ORDER BY PRIMERO
```

Además, no se deseará codificar todo el programa para que sólo busque un nombre en concreto. En vez de ello, se debería buscar el nombre proporcionado como parámetro de línea de comandos. Hacer estos cambios y convertir la sentencia SQL en un **String** de creación dinámica produce:

```
"SELECT PRIMERO,SEGUNDO CORREO " +
"FROM gente.csv gente " +
"WHERE " +
" (LAST='" + args[0] + "') " +
" AND (CORREO Is Not Null) " +
"ORDER BY PRIMERO");
```

SQL tiene otra forma de insertar nombres en una consulta, denominado *procedimientos almacenados*, que se usan para lograr incrementos de velocidad. Pero como experimentación de base de datos para un primer enfoque, construir las cadenas de consulta en Java es una opción más que correcta.

A partir de este ejemplo se puede ver que usando las herramientas actualmente disponibles —en particular la herramienta de construcción de consultas— es bastante directo programar con SQL y JDBC.

Una versión con IGU del programa de búsqueda

Es más útil dejar que el programa de búsqueda se ejecute continuamente y simplemente cambiar al mismo y teclear un nombre cuando se desee buscar a alguien. El siguiente programa crea el programa de búsqueda como una aplicación/*applet* y también añade finalización automática, de forma que los datos se mostrarán sin forzar al lector a teclear el apellido completo:

```
//: c15:jdbc:BuscarV.java
//versión IGU de Buscar.java.
// <applet code=BuscarV
// width=500 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
import java.sql.*;
import com.bruceeckel.swing.*;

public class BuscarV extends JApplet {
  String dbUrl = "jdbc:odbc:gente";
  String usuario = "";
  String contrasena = "";
  Statement s;
  JTextField buscarPor = new JTextField(20);
  JLabel conclusion =
    new JLabel("                                    ");
```

```
JTextArea resultados = new JTextArea(40, 20);
public void init() {
  buscarPor.getDocument().addDocumentListener(
    new BuscarL());
  JPanel p = new JPanel();
  p.add(new Label("Apellido a buscar:"));
  p.add(buscarPor);
  p.add(conclusion);
  Container cp = getContentPane();
  cp.add(p, BorderLayout.NORTH);
  cp.add(resultados, BorderLayout.CENTER);
  try {
    // Cargar el controlador (se registra automáticamente)
    Class.forName(
      "sun.jdbc.odbc.JdbcOdbcDriver");
    Connection c = DriverManager.getConnection(
      dbUrl, usuario, contrasena);
    s = c.createStatement();
  } catch(Exception e) {
    resultados.setText(e.toString());
  }
}
class BuscarL implements DocumentListener {
  public void changedUpdate(DocumentEvent e){}
  public void insertUpdate(DocumentEvent e){
    valorTextoCambiado();
  }
  public void removeUpdate(DocumentEvent e){
    valorTextoCambiado();
  }
}
public void valorTextoCambiado() {
  ResultSet r;
  if(buscarPor.getText().length() == 0) {
    conclusion.setText("");
    resultados.setText("");
    return;
  }
  try {
    // Finalización de nombres:
    r = s.executeQuery(
      "SELECT SEGUNDO FROM gente.csv gente " +
      "WHERE (SEGUNDO Like '" +
      buscarPor.getText()  +
      "%') ORDER BY SEGUNDO");
```

```
        if(r.next())
          conclusion.setText(
            r.getString("segundo"));
        r = s.executeQuery(
          "SELECT PRIMERO, SEGUNDO, CORREO " +
          "FROM gente.csv gente " +
          "WHERE (SEGUNDO='" +
          conclusion.getText() +
          "') AND (CORREO Is Not Null) " +
          "ORDER BY PRIMERO");
      } catch(Exception e) {
        resultados.setText(
          buscarPor.getText() + "\n");
        resultados.append(e.toString());
        return;
      }
      resultados.setText("");
      try {
        while(r.next()) {
          resultados.append(
            r.getString("Primero") + ", "
            + r.getString("sEGUNDO") +
            ": " + r.getString("CORREO") + "\n");
        }
      } catch(Exception e) {
        resultados.setText(e.toString());
      }
    }
    public static void main(String[] args) {
      Console.run(new BuscarV(), 500, 200);
    }
} ///:~
```

Mucha de la lógica de la base de datos es la misma, pero puede verse que se añade un **DocumentListener** al **JTextField** (véase la entrada **javax.swing.JTextField** de la documentación Java HTML de *http://www.java.sun.com* para encontrar detalles), de forma que siempre que se teclee un nuevo carácter, primero se intenta terminar el apellido buscándolo en la base de datos y usando el primero que se muestre. (Se coloca en la **JLabel conclusion**, y se usa como texto de búsqueda). De esta forma, tan pronto como se hayan tecleado los suficientes caracteres para que el programa encuentre el apellido buscado de manera unívoca, se puede parar el mismo.

Por qué el API JDBC parece tan complejo

Cuando se accede a la documentación en línea de JDBC puede asustar. En concreto, en la interfaz **DatabaseMetaData** —que es simplemente vasta, contrariamente al resto de interfaces de Java— hay mé-

todos como **dataDefinitionCausesTransactionCommit( ), getMaxColumnNameLength( ), getMaxStatementLength( ), storesMixedCaseQuotedIdentifiers( ), supportsANSI92IntermediateSQL( ), supportsLimitedOuterJoins( )** y demás. ¿De qué va todo esto?

Como se mencionó anteriormente, las bases de datos, desde su creación, siempre han parecido una fuente constante de tumultos, especialmente debido a la demanda de aplicaciones de bases de datos, y por consiguiente, las herramientas de bases de datos suelen ser tremendamente grandes. Recientemente —y sólo recientemente— ha habido una convergencia en el lenguaje común SQL (y hay muchos otros lenguajes comunes de bases de datos de uso frecuente). Pero incluso con un SQL "estándar" hay tantas variaciones del tema que JDBC debe proporcionar la gran interfaz **DatabaseMetaData,** de forma que el código pueda descubrir las capacidades del SQL estándar en particular que usa la base de datos a la que se está actualmente conectado. Abreviadamente, puede escribirse SQL simple y transportable, pero si se desea optimizar la velocidad, la codificación se multiplicará tremendamente al investigar las capacidades de la base de datos de un vendedor concreto.

Esto, por supuesto, no es culpa de Java. Las discrepancias entre los productos de base de datos son simplemente algo que JDBC intenta ayudar a compensar. Pero recuerde que le puede facilitar las cosas la escritura de sentencias de base de datos *(queries)* genéricas, sin apenas preocupar al rendimiento, o, si debe afinar en el rendimiento, conozca la plataforma en la que está escribiendo para evitar tener que escribir todo ese código de investigación.

Un ejemplo más sofisticado

Un segundo ejemplo[2] más interesante involucra una base de datos multitabla que reside en un servidor. Aquí, la base de datos está destinada a proporcionar un repositorio de actividades de la comunidad y permitir a la gente apuntarse a esos eventos, de forma que se le llama Base de Datos de Interés de la Comunidad (CID, o *Community Interests Database*). Este ejemplo sólo proporcionará un repaso de la base de datos y su implementación. Hay muchos libros, seminarios y paquetes software que nos ayudarán a diseñar y desarrollar una base de datos.

Además, este ejemplo presupone que anteriormente se ha instalado una base de datos SQL en un servidor (aunque también podría ejecutarse en una maquina local), y que se ha integrado y descubierto un controlador JDBC apropiado para esa base de datos. Existen varias bases de datos SQL gratuitas disponibles, y algunas se instalan incluso automáticamente con varias versiones de Linux. Cada una es responsable de elegir la base de datos y de ubicar el controlador JDBC; este ejemplo está basado en una base de datos SQL llamada "Fuga".

Para facilitar los cambios en la información de conexión, el controlador de la base de datos, el URL de la misma, el nombre de usuario y la contraseña se colocaron en una clase diferente:

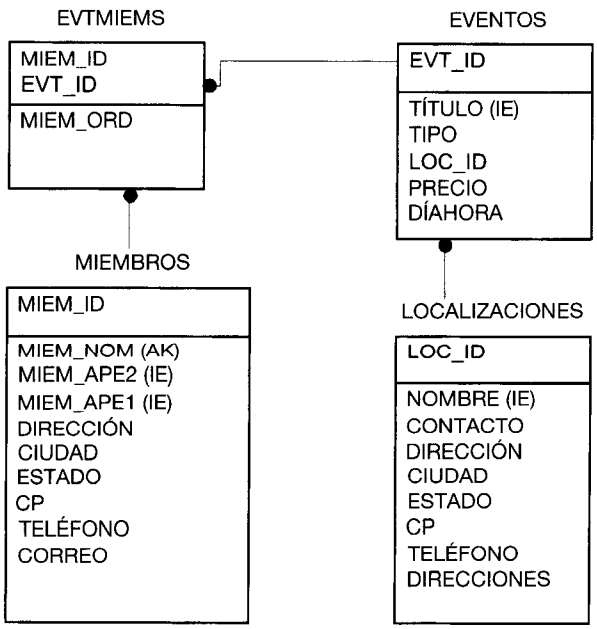
```
//: c15:jdbc:ConectarCID.java
// Información de conexión a base de datos para
// la base de datos de interés de la comunidad (CID).
```

[2] Creado por Dave Bartlett.

```
public class ConectarCID {
  // Toda la información específica de Fuga:
  public static String controladorBD =
    "COM.fuga.core.JDBCDriver";
  public static String URLdb =
    "jdbc:fuga:d:/docs/_work/JSapienDB";
  public static String usuario = "";
  public static String contrasena = "";
} ///:~
```

En este ejemplo, no hay protección por contraseñas en la base de datos, por lo que el nombre de usuario y la contraseña están vacíos.

La base de datos consiste en un conjunto de tablas con la estructura que se muestra:



"Miembros" contiene la información de miembros de la comunidad, "Eventos" y "Localizaciones" contienen información sobre las actividades y cuándo tendrán lugar, y "Evtmiems" conecta los eventos con los miembros a los que les gustaría asistir. Puede verse que un miembro de datos de una tabla produce una clave en la otra.

La clase siguiente contiene las cadenas SQL que crearán estas tablas de bases de datos (véase una guía de SQL si se desea obtener explicación del código SQL):

```
//: c15:jdbc:CIDSQL.java
// Strings SQL para crear las tablas de la CID.
```

```
public class CIDSQL {
  public static String[] sql = {
    // Crear la tabla MIEMBROS:
    "drop table MIEMBROS",
    "create table MIEMBROS " +
    "(MIEM_ID INTEGER primary key, " +
    "MIEM_NOM VARCHAR(12) not null unique, "+
    "MIEM_APE2 VARCHAR(40), " +
    "MIEM_APE1 VARCHAR(20), " +
    "DIRECCION VARCHAR(40), " +
    "CIUDAD VARCHAR(20), " +
    "ESTADO CHAR(4), " +
    "CP CHAR(5), " +
    "TELEFONO CHAR(12), " +
    "CORREO VARCHAR(30))",
    "create unique index " +
    "APE2_IND on MIEMBROS(MIEM_APE2)",
    // Crear la tabla EVENTOS
    "drop table EVENTOS",
    "create table EVENTOS " +
    "(EVT_ID INTEGER primary key, " +
    "EVT_TITULO VARCHAR(30) not null, " +
    "EVT_TIPO VARCHAR(20), " +
    "LOC_ID INTEGER, " +
    "PRECIO DECIMAL, " +
    "DIAHORA TIMESTAMP)",
    "create unique index " +
    "TITULO_IND on EVENTOS(EVT_TITULO)",
    // Crear la tabla EVTMIEMS
    "drop table EVTMIEMS",
    "create table EVTMIEMS " +
    "(MIEM_ID INTEGER not null, " +
    "EVT_ID INTEGER not null, " +
    "MIEM_ORD INTEGER)",
    "create unique index " +
    "EVTMIEM_IND on EVTMIEMS(MIEM_ID, EVT_ID)",
    // Crear la tabla LOCALIZACIONES
    "drop table LOCALIZACIONES",
    "create table LOCALIZACIONES " +
    "(LOC_ID INTEGER primary key, " +
    "LOC_NOMBRE VARCHAR(30) not null, " +
    "CONTACTO VARCHAR(50), " +
    "DIRECCION VARCHAR(40), " +
    "CIUDAD VARCHAR(20), " +
    "ESTADO VARCHAR(4), " +
```

```
      "CP VARCHAR(5), " +
      "TELEFONO CHAR(12), " +
      "DIRECCIONES VARCHAR(4096))",
      "create unique index " +
      "NOMBRE_IND on LOCALIZACIONES(LOC_NOMBRE)",
   };
} ///:~
```

El programa siguiente usa la información de **ConectarCID** y **CIDSQL** para cargar el controlador JDBC y establecer la conexión a la base de datos y crear después la estructura de tablas del diagrama de arriba. Para conectar con la base de datos se invoca al método **static DriverManager.getConnection( )**, pasándole el URL de la base de datos, el nombre de usuario y una contraseña para entrar en la misma. Al retornar, se obtiene un objeto **Connection** que puede usarse para hacer consultas y manipular la base de datos. Una vez establecida la conexión se puede simplemente meter el SQL en la base de datos, en este caso, recorriendo el array **CIDSQL**. Sin embargo, la primera vez que se ejecute el programa, el comando "drop table" fallará, causando una excepción, que es capturada, y de la que se informa, para finalmente ignorarla. La razón del comando "drop table" es permitir una experimentación sencilla: se puede modificar el SQL que define las tablas y después volver a ejecutar el programa, lo que hará que las viejas tablas sean reemplazadas por las nuevas.

En este ejemplo, tiene sentido dejar que se lancen las excepciones a la consola:

```
//: c15:jdbc:CrearTablasCID.java
// Crea las tablas de base de datos
// para la base de datos de interés de la comunidad.
import java.sql.*;

public class CrearTablasCID {
   public static void main(String[] args)
   throws SQLException, ClassNotFoundException,
   IllegalAccessException {
      // Cargar el controlador (se registra a sí mismo)
      Class.forName(ConectarCID.controladorBD);
      Connection c = DriverManager.getConnection(
         ConectarCID.URLdb, ConectarCID.usuario,
         ConectarCID.contrasena);
      Statement s = c.createStatement();
      for(int i = 0; i < CIDSQL.sql.length; i++) {
         System.out.println(CIDSQL.sql[i]);
         try {
            s.executeUpdate(CIDSQL.sql[i]);
         } catch(SQLException sqlEx) {
            System.err.println(
               "Probablemente falló un 'drop table' ");
         }
      }
```

```
    s.close();
    c.close();
  }
} ///:~
```

Nótese que se pueden controlar todos los cambios en la base de datos, cambiando **String**s en la tabla **CIDSQL**, sin modificar **CrearTablasCID**.

El método **executeUpdate( )** devolverá generalmente el número de las filas afectadas por la sentencia SQL. Este método se usa más frecuentemente para ejecutar sentencias `INSERT`, `UPDATE` o `DELETE`, para modificar una o más columnas. Para sentencias como `CREATE TABLE`, `DROP TABLE`, y `CREATE INDEX`, **executeUpdate( )** siempre devuelve cero.

Para probar la base de datos, se carga con algunos datos de ejemplo. Esto requiere una serie de `INSERT`s seguidos de una `SELECT` para producir un resultado conjunto. Para facilitar las adiciones y cambios a los datos de prueba, éste se dispone en un array bidimensional de **Object**s, y el método **executeInsert( )** puede usar después la información de una columna de la tabla para crear el comando SQL apropiado.

```
//: c15:jdbc:CargarBD.java
// Carga y prueba la base de datos.
import java.sql.*;

class PruebaConjunto {
  Object[][] datos = {
    { "MIEMBROS", new Integer(1),
      "dbartlett", "Bartlett", "David",
      "123 Mockingbird Lane",
      "Gettysburg", "PA", "19312",
      "123.456.7890",  "bart@you.net" },
    { "MIEMBROS", new Integer(2),
      "beckel", "Eckel", "Bruce",
      "123 Over Rainbow Lane",
      "Crested Butte", "CO", "81224",
      "123.456.7890", "beckel@you.net" },
    { "MIEMBROS", new Integer(3),
      "rcastaneda", "Castaneda", "Robert",
      "123 Downunder Lane",
      "Sydney", "NSW", "12345",
      "123.456.7890", "rcastaneda@you.net" },
    { "LOCALIZACIONES", new Integer(1),
      "Center for Arts",
      "Betty Wright", "123 Elk Ave.",
      "Crested Butte", "CO", "81224",
      "123.456.7890",
      "Ir de esta manera." },
```

```
      { "LOCALIZACIONES", new Integer(2),
        "Witts End Conference Center",
        "John Wittig", "123 Music Drive",
        "Zoneville", "PA", "19123",
        "123.456.7890",
        "Ir de esta manera." },
      { "EVENTOS", new Integer(1),
        "Project Management Myths",
        "Software Development",
        new Integer(1), new Float(2.50),
        "2000-07-17 19:30:00" },
      { "EVENTOS", new Integer(2),
        "Life of the Crested Dog",
        "Archeology",
        new Integer(2), new Float(0.00),
        "2000-07-19 19:00:00" },
      // Asociar gente a eventos
      {  "EVTMIEMS",
        new Integer(1),   // Dave va al evento
        new Integer(1),   // Software.
        new Integer(0) },
      { "EVTMIEMS",
        new Integer(2),   // Bruce va al evento
        new Integer(2),   // Arqueología.
        new Integer(0) },
      { "EVTMIEMS",
        new Integer(3),   // Robert va al evento
        new Integer(1),   // Software.
        new Integer(1) },
      { "EVTMIEMS",
        new Integer(3), // ... y
        new Integer(2), // al evento Arqueología.
        new Integer(1) },
    };
    // Usar el conjunto de datos por defecto:
    public PruebaConjunto() {}
    // Usar un conjunto de datos distinto:
    public PruebaConjunto(Object[][] dat) { datos = dat; }
}

public class CargarDB {
    Statement sentencia;
    Connection conexion;
    PruebaConjunto pconjunto;
    public CargarBD(PruebaConjunto p) throws SQLException {
```

```
    pconjunto = p;
    try {
      // Cargar el controlador (se registra solo)
      Class.forName(ConectarCID.controladorBD);
    } catch(java.lang.ClassNotFoundException e) {
      e.printStackTrace(System.err);
    }
    conexion = DriverManager.getConnection(
      ConectarCID.URLdb, ConectarCID.usuario,
      ConectarCID.contrasena);
    sentencia = conexion.createStatement();
  }
  public void limpiar() throws SQLException {
    sentencia.close();
    conexion.close();
  }
  public void ejecutarInsertar(Object[] datos) {
    String sql = "insert into "
      + datos[0] + " values(";
    for(int i = 1; i < datos.length; i++) {
      if(datos[i] instanceof String)
        sql += "'" + datos[i] + "'";
      else
        sql += datos[i];
      if(i < datos.length - 1)
        sql += ", ";
    }
    sql += ')';
    System.out.println(sql);
    try {
      sentencia.executeUpdate(sql);
    } catch(SQLException sqlEx) {
      System.err.println("Falló inserción.");
      while (sqlEx != null) {
        System.err.println(sqlEx.toString());
        sqlEx = sqlEx.getNextException();
      }
    }
  }
  public void cargar() {
    for(int i = 0; i< pconjunto.datos.length; i++)
      ejecutarInsertar(pconjunto.datos[i]);
  }
  // Lanzar excepciones a la consola:
  public static void main(String[] args)
```

```
  throws SQLException {
    CargarBD db = new CargarBD(new PruebaConjunto());
    bd.cargar();
    try {
      // Obtener un ResultSet a partir de la base de datos cargada:
      ResultSet rs = db.statement.executeQuery(
        "select " +
        "e.EVT_TITULO, m.MIEM_APE2, m.MIEM_APE1 "+
        "from EVENTOS e, MIEMBROS m, EVTMIEMS em " +
        "where em.EVT_ID = 2 " +
        "and e.EVT_ID = em.EVT_ID " +
        "and m.MIEM_ID = em.MIEM_ID");
      while (rs.next())
        System.out.println(
          rs.getString(1) + "   " +
          rs.getString(2) + ", " +
          rs.getString(3));
    } finally {
      db.limpiar();
    }
  }
} ///:~
```

La clase **PruebaConjunto** contiene un conjunto de datos por defecto producido al usar el constructor por defecto; sin embargo, también se puede crear un objeto **PruebaConjunto** usando un conjunto de datos alternativo con el segundo constructor. El conjunto de datos se guarda en un array bidimensional de **Object** porque puede ser de cualquier tipo, incluidos **String**s o tipos numéricos. El método **ejecutarInsertar( )** utiliza RTTI para distinguir entre datos **String** (que deben ir entre comillas)y los no-**String** a medida que construye el comando SQL a partir de los datos. Tras imprimir este programa en la consola, se usa **executeUpdate( )** para enviarlo a la base de datos.

El constructor de **CargarBD** hace la conexión, y **cargar( )** recorre los datos y llama a **ejecutarInsertar( )** por cada registro. El método **limpiar( )** cierra la sentencia y la conexión; para garantizar que se invoque, éste se ubica dentro de una cláusula finally.

Una vez cargada la base de datos, una sentencia **executeQuery( )** produce el resultado conjunto de ejemplo. Puesto que la consulta combina varias tablas, es un ejemplo de un *join*.

Hay más información de JDBC disponible en los documentos electrónicos que vienen como parte de la distribución de Java de Sun. Además, se puede encontrar más en el libro *JDBC Database Access with Java* (Hamilton, Cattel, y Fisher, Addison-Wesley, 1997). También suelen aparecer otros libros sobre este tema con bastante frecuencia.

Servlets

El acceso de clientes desde Internet o intranets corporativas es una forma segura de permitir a varios usuarios acceder a datos y recursos de forma sencilla[3]. Este tipo de acceso está basado en el uso de los estándares *Hypertext Markup Language* (HTML) e *Hypertext Transfer Protocol* (HTTP) de la World Wide Web por parte de los clientes. El conjunto de API Servlet abstrae un marco de solución común para responder a peticiones HTTP.

Tradicionalmente, la forma de gestionar un problema como el de permitir a un cliente de Internet actualizar una base de datos es crear una página HTML con campos de texto y un botón de "enviar". El usuario teclea la información apropiada en los campos de texto y presiona el botón "enviar". Los datos se envían junto con una URL que dice al servidor qué hacer con los datos especificando la ubicación de un programa *Common Gateway Interface* (CGI) que ejecuta el servidor, proporcionando al programa los datos al ser invocado. El programa CGI suele estar escrito en Perl, Python, C, C++ o cualquier lenguaje que pueda leer de la entrada estándar y escribir en la salida estándar. Esto es todo lo que es proporcionado por el servidor web: se invoca al programa CGI, y se usan flujos estándar (u, opcionalmente en caso de entrada, una variable de entorno) para la entrada y la salida. El programa CGI es responsable de todo lo demás. Primero, mira a los datos y decide si el formato es correcto. Si no, el programa CGI debe producir HTML para describir el problema; esta página se pasa al servidor Web (vía salida estándar del programa CGI), que lo vuelve a enviar al usuario. El usuario debe generalmente salvar la página e intentarlo de nuevo. Si los datos son correctos, el programa CGI los procesa de forma adecuada, añadiéndolos quizás a una base de datos. Después, debe producir una página HTML apropiada para que el servidor web se la devuelva al usuario.

Sería ideal ir a una solución completamente basada en Java para este ejemplo —un *applet* en el lado cliente que valide y envíe los datos, y un servlet en el lado servidor para recibir y procesar los datos. Desgraciadamente, aunque se ha demostrado que los *applets* son una tecnología con gran soporte, su uso en la Web ha sido problemático porque no se puede confiar en que en un navegador cliente web haya una versión particular de Java disponible; de hecho, ¡ni siquiera se puede confiar en que un navegador web soporte Java! En una Intranet, se puede requerir que esté disponible cierto soporte, lo que permite mucha mayor flexibilidad en lo que se puede hacer, pero en la Web el enfoque más seguro es manejar todo el proceso en el lado servidor y entregar HTML plano al cliente. De esa forma, no se negará a ningún cliente el uso del sitio porque no tenga el software apropiado instalado.

Dado que los servlets proporcionan una solución excelente para soporte en el lado servidor, son una de las razones más populares para migrar a Java. No sólo proporcionan un marco que sustituye a la programación CGI (y elimina muchos problemas complicados de los CGIs), sino que todo el código gana portabilidad de plataforma gracias al uso de Java, teniendo acceso a todos los APIs de Java (excepto, por supuesto, a los que producen IGU, como Swing).

---

[3] Dave Bartlett contribuyó desarrollo de este material, y también en la sección JSP.

El servlet básico

La arquitectura del API servlet es la de un proveedor de servicios clásico con un método **service( )** a través del cual se enviarán todas las peticiones del cliente por parte del software contenedor del servlet, y los métodos de ciclo de vida **init( )** y **destroy( )**, que se invocan sólo cuando se carga y descarga el servlet (esto raramente ocurre).

```
public interface Servlet
  public void init(ServletConfig config)
    throws ServletException;
  public ServletConfig getServletConfig();
  public void service(ServletRequest req,
    ServletResponse res)
    throws ServletException, IOException;
  public String getServletInfo();
  public void destroy();
}
```

El único propósito de **getServletConfig( )** es devolver un objeto **ServletConfig** que contenga los parámetros de inicialización y arranque de este servidor. El método **getServletInfo( )** devuelve una cadena de caracteres que contiene información sobre el servlet, como el autor, la versión y los derechos del autor.

La clase **GenericServlet** es una implementación genérica de esta interfaz y no suele usarse. La clase **HttpServlet** es una extensión de **GenericServlet** y está diseñada específicamente para manejar el protocolo HTTP —**HttpServlet** es la que se usará la mayoría de veces.

El atributo más conveniente de la API de servlets lo constituyen los objetos auxiliares que vienen con la clase **HttpServlet** para darle soporte. Si se mira al método **service( )** de la interfaz **Servlet**, se verá que tiene dos parámetros: **ServletRequest** y **ServletResponse**. Con la clase **HttpServlet** se extienden estos dos objetos para HTTP: **HttpServletRequest** y **HttpServletResponse**. He aquí un ejemplo simple que muestra el uso de **HttpServletResponse**:

```
//: c15:servlets:ReglasServlets.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ReglasServlets extends HttpServlet {
  int i = 0; // Servlet "persistencia"
  public void service(HttpServletRequest req,
  HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter salida = res.getWriter();
    salida.print("<HEAD><TITLE>");
    salida.print("Una estrategia de lado servidor");
```

```
      salida.print("</TITLE></HEAD><BODY>");
      salida.print("<h1>¡Reglas Servets! " + i++);
      salida.print("</h1></BODY>");
      salida.close();
    }
} ///:~
```

ReglasServlets es casi tan simple como puede serlo un servlet. El servlet sólo se inicializa una vez llamando a su método **init( )**, al cargar el servidor tras arrancar por primera vez el contenedor de servlets. Cuando un cliente hace una petición a una URL que resulta representar un servlet, el contenedor de servlets intercepta la petición y hace una llamada al método **service( )**, tras configurar los objetos **HttpServletRequest** y **HttpServletResponse**.

La responsabilidad principal del método **service( )** es interactuar con la petición HTTP que ha enviado el cliente, y construir una respuesta HTTP basada en los atributos contenidos dentro de la petición. **ReglasServlets** sólo manipula el objeto respuesta sin mirar a lo que el cliente puede haber enviado.

Tras configurar el tipo de contenido de la respuesta (cosa que debe hacerse siempre antes de procurar el **Writer** u **OutputStream**), el método **getWriter( )** del objeto respuesta produce un objeto **PrintWriter**, que se usa para escribir información de respuesta basada en caracteres (alternativamente, **getOutputStream( )** produce un **OutputStream**, usado para respuesta binaria, que sólo se usa en soluciones más especializadas).

El resto del programa simplemente manda HTML de vuelta al cliente (se asume que el lector entiende HTML, por lo que no se explica esa parte) como una secuencia de **String**s. Sin embargo, nótese la inclusión del "contador de accesos" representado por la variable **i**. Éste se convierte automáticamente en un **String** en la sentencia **print( )**.

Cuando se ejecuta el programa, se verá que se mantiene el valor de **i** entre las peticiones al servidor. Ésta es una propiedad esencial de los servlets: puesto que en el contenedor sólo se carga un servlet de cada clase, y éste nunca se descarga (a menos que finalice el contenedor de servlets, lo cual es algo que normalmente sólo ocurre si se reinicia la máquina servidora), ¡cualquier campo de esa clase servidora se convierte en un objeto persistente! Esto significa que se pueden mantener sin esfuerzo valores entre peticiones servlets, mientras que con CGI había que escribir los valores al disco para preservarlos, lo que requería una cantidad de trabajo bastante elevada para que funcionara con éxito, y solía producir soluciones exclusivas para una plataforma.

Por supuesto, en ocasiones, el servidor web, y por consiguiente, el contenedor de servlets, tienen que ser reiniciados como parte del mantenimiento o por culpa de un fallo de corriente. Para evitar perder cualquier información persistente, se invoca automáticamente a los métodos **init( )** y **destroy( )** del servlet siempre que se carga o descarga el servlet, proporcionando la oportunidad de salvar los datos durante el apagado, y restaurarlos tras el nuevo arranque. El contenedor de servlets llama al método **destroy( )** al terminarse a sí mismo, por lo que siempre se logra una oportunidad de salvar la información valiosa, en la medida en que la máquina servidora esté configurada de forma inteligente.

Hay otro aspecto del uso de **HttpServlet**. Esta clase proporciona métodos **doGet( )** y **doPost( )**, que diferencian entre un envío CGI "GET" del cliente, y un CGI "POST". GET y POST simplemente varían en los detalles de la forma en que envían los datos, que es algo que preferimos ignorar. Sin embargo, la mayoría de información publicada que hemos visto parece ser favorable a la creación de métodos **doGet( )** y **doPost( )** separados en vez de un método **service( )** genérico, que maneje los dos casos. Este favoritismo parece bastante común, pero nunca lo he visto explicado de forma que nos haga creer que se deba a algo más que a la inercia de los programadores de CGI que están habituados a prestar atención a si se está usando GET o POST. Por tanto, y por mantener el espíritu de "hacer todo siempre de la forma más simple que funcione"[4], simplemente usaremos el método **service( )** en estos ejemplos, y que se encargue de los GET *frente a* POST. Sin embargo, hay que mantener presente que podríamos dejarnos algo, por lo que de hecho sí que podría haber una buena razón para usar en su lugar **doGet( )** o **doPost( )**.

Siempre que se envía un formulario a un servidor, el **HttpServletRequest** viene precargado con todos los datos del formulario, almacenados como pares clave-valor. Si se conocen los nombres de los campos, pueden usarse directamente con el método **getParameter( )** para buscar los valores. También se puede lograr un **Enumeration** (la forma antigua del **Iterator**) para los nombres de los campos, como se muestra en el ejemplo siguiente. Este ejemplo también demuestra cómo se puede usar un único servlet para producir la página que contiene el formulario y para responder a la pagina (más adelante se verá una solución mejor, con JSP). Si la **Enumeration** está vacía, no hay campos; esto significa que no se envió ningún formulario. En este caso, se produce el formulario y el botón de enviar reinvocará al mismo servlet. Sin embargo, si los campos existen, se muestran.

```
/: c15:servlets:EcoFormulario.java
// Vuelca los pares nombre-valor de cualquier
// formulario HTML
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class EcoFormulario extends HttpServlet {
  public void service(HttpServletRequest req,
    HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter salida = res.getWriter();
    Enumeration campos = req.getParameterNames();
    if(!campos.hasMoreElements()) {
      // No se envía formulario -crear uno:
      salida.print("<html>");
      salida.print("<form method=\"POST\"" +
        " action=\"EcoFormulario\">");
      for(int i = 0; i < 10; i++)
```

[4] Uno de los eslóganes principales de la Programación Extrema (XP). Ver *http:www.xprogramming.com*.

```
            salida.print("<b>Campo" + i + "</b> " +
              "<input type=\"text\""+
              " size=\"20\" name=\"Campo" + i +
              "\" value=\"Valor" + i + "\"><br>");
          salida.print("<INPUT TYPE=SUBMIT name=someter"+
          " Value=\"Someter\"></form></html>");
        } else {
          salida.print("<h1>Tu formulario contenia:</h1>");
          while(campos.hasMoreElements()) {
            String campo= (String)campos.nextElement();
            String valor= req.getParameter(campo);
            salida.print(campo + " = " + valor+ "<br>");
          }
        }
        salida.close();
    }
} ///:~
```

Una pega que se verá aquí es que Java no parece haber sido diseñado con el procesamiento de cadenas de caracteres en mente —el formateo de la página de retorno no supone más que quebraderos de cabeza debido a los saltos de línea, las marcas de escape y los signos "+" necesarios para construir objetos **String**. Con una página HTML extensa no sería razonable codificarla directamente en Java. Una solución es mantener la página como un archivo de texto separado, y abrirla y pasársela al servidor web. Si se tiene que llevar a cabo cualquier tipo de sustitución de los contenidos de la página, la solución no es mucho mejor debido al procesamiento tan pobre de las cadenas de texto en Java. En estos casos, probablemente se hará mejor usando una solución más apropiada (nuestra elección sería Phyton; hay una versión que se fija en Java llamada JPython) para generar la página de respuesta.

Servlets y multihilo

El contenedor de servlets tiene un conjunto de hilos que irá despachando para gestionar las peticiones de los clientes. Es bastante probable que dos clientes que lleguen al mismo tiempo puedan ser procesados por **service( )** a la vez. Por consiguiente, el método **service( )** debe estar escrito de forma segura para hilos. Cualquier acceso a recursos comunes (archivos, bases de datos) necesitará estar protegido haciendo uso de la palabra clave **synchronized**.

El siguiente ejemplo simple pone una cláusula **synchronized** en torno al método **sleep( )** del hilo. Éste bloqueará al resto de los hilos hasta que se haya agotado el tiempo asignado (5 segundos). Al probar esto, deberíamos arrancar varias instancias del navegador y acceder a este servlet tan rápido como se pueda en cada una —se verá que cada una tiene que esperar hasta que le llega su turno.

```
//: c15:servlets:HiloServlet.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

```
public class HiloServlet extends HttpServlet {
  int i;
  public void service(HttpServletRequest req,
    HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter salida = res.getWriter();
    synchronized(this) {
      try {
        Thread.currentThread().sleep(5000);
      } catch(InterruptedException e) {
        System.err.println("Interrumpido");
      }
    }
    salida.print("<h1>Finalizado " + i++ + "</h1>");
    salida.close();
  }
} ///:~
```

También es posible sincronizar todo el servlet poniendo la palabra clave **synchronized** delante del método **service( )**. De hecho, la única razón de usar la cláusula **synchronized** en su lugar es por si la sección crítica está en un cauce de ejecución que podría no ejecutarse. En ese caso, se podría evitar también la sobrecarga de tener que sincronizar cada vez utilizando una cláusula **synchronized**. De otra forma, todos los hilos tendrían que esperar de todas formas, por lo que también se podría **sincronizar** todo el método.

Gestionar sesiones con servlets

HTTP es un protocolo "sin sesión", por lo que no se puede decir desde un acceso al servidor a otro si se trata de la misma persona que está accediendo repetidamente al sitio, o si se trata de una persona completamente diferente. Se ha invertido mucho esfuerzo en mecanismos que permitirán a los desarrolladores web llevar a cabo un seguimiento de las sesiones. Las compañías no podrían hacer comercio electrónico sin mantener un seguimiento de un cliente y por ejemplo, de los elementos que éste ha introducido en su carro de la compra.

Hay bastantes métodos para llevar a cabo el seguimiento de sesiones, pero el más común es con "*cookies*" persistentes, que son una parte integral de los estándares Internet. El Grupo de Trabajo HTTP de la Internet Engineering Task Force ha descrito las *cookies* en el estándar oficial en RFC 2109 (*ds.internic.net/rfc/rfc 2109.txt* o compruebe *www.cookiecentral.com*).

Una *cookie* no es más que una pequeña pieza de información enviada por un servidor web a un navegador. El navegador almacena la cookie en el disco local y cuando se hace otra llamada al URL con la que está asociada la *cookie*, éste se envía junto con la llamada, proporcionando así que la información deseada vuelva a ese servidor (generalmente, proporcionando alguna manera de que el servidor pueda determinar que es uno el que llama). Los clientes, sin embargo, pueden desactivar la habilidad del navegador para aceptar *cookies*. Si el sitio debe llevar un seguimiento de un cliente que ha desactivado las *cookies*, hay que incorporar a mano otro método de seguimiento de sesiones

(reescritura de URL o campos de formulario ocultos), puesto que las capacidades de seguimiento de sesiones construidas en el API servlet están diseñadas para *cookies*.

La clase **Cookie**

El API servlet (en versiones 2.0 y superior) proporciona la clase **Cookie**. Esta clase incorpora todos los detalles de cabecera HTTP y permite el establecimiento de varios atributos de *cookie*. Utilizar la *cookie* es simplemente un problema de añadirla al objeto respuesta. El constructor toma un nombre de *cookie* como primer parámetro y un valor como segundo. Las cookies se añaden al objeto respuesta antes de enviar ningún contenido.

```
Cookie oreo = new Cookie("TIJava", "2000");
res.addCookie(cookie);
```

Las *cookies* suelen recubrirse invocando al método **getCookies( )** del objeto **HttpServletRequest**, que devuelve un array de objetos *cookie*.

```
Cookie[] cookies = req.getCookies();
```

Después se puede llamar a **getValue( )** para cada cookie, para producir un **String** que contenga los contenidos de la *cookie*. En el ejemplo de arriba, **getValue("TIJava")** producirá un **String** de valor "2000".

La clase **Session**

Una sesión es una o más solicitudes de páginas por parte de un cliente a un sitio web durante un periodo definido de tiempo. Si uno compra, por ejemplo, ultramarinos en línea, se desea que una sesión dure todo el periodo de tiempo desde que se añade al primer elemento a "Mi carrito de la compra" hasta el momento en el que se compruebe todo. Cada elemento que se añada al carrito de la compra vendrá a producir una nueva conexión HTTP, que no tiene conocimiento de conexiones previas o de los elementos del carrito de la compra. Para compensar esta falta de información, los mecanismos suministrados por la especificación de *cookies* permiten al servlet llevar a cabo seguimiento de sesiones.

Un objeto **Session** servlet vive en la parte servidora del canal de comunicación; su meta es capturar datos útiles sobre este cliente a medida que el cliente recorre e interactúa con el sitio web. Esta información puede ser pertinente para la sesión actual, como los elementos del carro de la compra, o pueden ser datos como la información de autentificación introducida cuando el cliente entró por primera vez en el sitio web, y que no debería ser reintroducida durante un conjunto de transacciones particular.

La clase **Session** del API servlet usa la clase **Cookie** para hacer su trabajo. Sin embargo, todo el objeto **Session** necesita algún tipo de identificador único almacenado en el cliente y que se pasa al servidor. Los sitios web también pueden usar otros tipos de seguimiento de sesión, pero estos mecanismos serán más difíciles de implementar al no estar encapsulados en el API servlet (es decir, hay que escribirlos a mano para poder enfrentarse a la situación de deshabilitación de las *cookies* por parte del cliente).

He aquí un ejemplo que implementa seguimiento de sesión con el API servlet:

```
//: c15:servlets:SeguirSesion.java
// Usando la clase HttpSession.
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SeguirSesion extends HttpServlet {
  public void service(HttpServletRequest req,
  HttpServletResponse res)
  throws ServletException, IOException {
    // Retirar el objeto Session antes de enviar
    // ninguna salida al cliente.
    HttpSession sesion = req.getSession();
    res.setContentType("text/html");
    PrintWriter salida = res.getWriter();
    salida.println("<HEAD><TITLE> SeguirSesion ");
    salida.println(" </TITLE></HEAD><BODY>");
    salida.println("<h1> SeguirSesion </h1>");
    // Un contador de accesos simple para esta sesion.
    Integer ival = (Integer)
      sesion.getAttribute("sesspeek.cntr");
    if(ival==null)
      ival = new Integer(1);
    else
      ival = new Integer(ival.intValue() + 1);
    sesion.setAttribute("sesspeek.cntr", ival);
    salida.println("Has accedido a esta página <b>"
      + ival + "</b> veces.<p>");
    salida.println("<h2>");
    salida.println("Grabados datos de la sesion </h2>");
    // Iterar por todos los datos de la sesión:
    Enumeration nombresSes =
      sesion.getAttributeNames();
    while(nombresSes.hasMoreElements()) {
      String nombre =
        nombresSes.nextElement().toString();
      Object valor = sesion.getAttribute(nombre);
      salida.println(name + " = " + value + "<br>");
    }
    salida.println("<h3> Estadisticas de la sesion </h3>");
    salida.println("ID Sesion : "
      + sesion.getId() + "<br>");
```

```
      salida.println("Nueva Sesion: " + sesion.isNew()
        + "<br>");
      salida.println("Hora de Creacion: "
        + sesion.getCreationTime());
      salida.println("<I>(" +
        new Date(sesion.getCreationTime())
        + ")</I><br>");
      salida.println("Hora del ultimo acceso: " +
        sesion.getLastAccessedTime());
      salida.println("<I>(" +
        new Date(sesion.getLastAccessedTime())
        + ")</I><br>");
      salida.println("Intervalo de Inactividad de la sesion: "
        + sesion.getMaxInactiveInterval());
      salida.println("ID de sesion en peticion: "
        + req.getRequestedSessionId() + "<br>");
      salida.println("ID de sesion desde Cookie: "
        + req.isRequestedSessionIdFromCookie()
        + "<br>");
      salida.println("Es el ID de la session del URL: "
        + req.isRequestedSessionIdFromURL()
        + "<br>");
      salida.println("Es ID de session valido: "
        + req.isRequestedSessionIdValid()
        + "<br>");
      salida.println("</BODY>");
      salida.close();
    }
    public String getServletInfo() {
      return "Un servlet de seguimiento de sesion";
    }
} ///:~
```

Dentro del método **service( )**, se invoca a **getSession( )** para el objeto petición, que devuelve el objeto **Session** asociado con esta petición. El objeto **Session** no viaja a través de la red, sino que en vez de ello, vive en el servidor asociado con un cliente y sus peticiones.

El método **getSession( )** viene en dos versiones: la de sin parámetros, usada aquí, y **getSession(boolean)**. Usar **getSession(true)** equivale a **getSession( )**. La única razón del **boolean** es para establecer si se desea crear el objeto sesión si no es encontrado. La llamada más habitual es **getSession(true)**, razón de la existencia de **getSession( )**.

El objeto **Session**, si no es nuevo, nos dará detalles sobre el cliente provenientes de sus visitas anteriores. Si el objeto **Session** es nuevo, el programa comenzará a recopilar información sobre las actividades del cliente en esta visita. La captura de esta información del cliente se hace mediante los métodos **setAttribute( )** y **getAttribute( )** del objeto de sesión.

```
java.lang.Object getAtribute(java.lang.String)
void setAttribute(java.lang.String nombre,
                  java.lang.Object valor)
```

El objeto **Session** utiliza un emparejamiento nombre-valor simple para cargar información. El nombre es un **String**, y el valor puede ser cualquier objeto derivado de **java.lang.Object**. **SeguirSesion** mantiene un seguimiento de las veces que ha vuelto el cliente durante esta sesión. Esto se hace con un objeto **Integer** denominado **sesspeek.cntr**. Si no se encuentra el nombre se crea un **Integer** de valor uno, si no, se crea un **Integer** con el valor incrementado respecto del **Integer** anteriormente guardado. Si se usa la misma clave en una llamada a **setAttribute( )**, el objeto nuevo sobreescribe el viejo. El contador incrementado se usa para mostrar el número de veces que ha vistado el cliente durante esta sesión.

El método **getAttributeNames( )** está relacionado con **getAttribute( )** y **setAttribute( )**; devuelve una enumeración de los nombres de objetos vinculados al objeto **Session**. Un bucle **while** en **SeguirSesion** muestra este método en acción.

Uno podría preguntarse durante cuánto tiempo puede permanecer inactivo un objeto **Session**. La respuesta depende del contenedor de servlets que se esté usando; generalmente vienen por defecto a 30 minutos (1.800 segundos), que es lo que debería verse desde la llamada a **SeguirSesion** hasta **getMaxInactiveInterval( )**. Las pruebas parecen producir resultados variados entre contenedores de servlets. En ocasiones, el objeto **Session** puede permanecer inactivo durante toda la noche, pero nunca hemos visto ningún caso en el que el objeto **Session** desaparezca en un tiempo menor al especificado por el intervalo de inactividad. Esto se puede probar estableciendo el valor de este intervalo con **setMaxInactiveInterval( )** a 5 segundos y ver si el objeto **Session** se cuelga o es eliminado en el tiempo apropiado. Éste puede constituir un atributo a investigar al seleccionar un contenedor de servlets.

Ejecutar los ejemplos de servlets

Si el lector no está trabajando con un servidor de aplicaciones que maneje automáticamente las tecnologías servlet y JSP de Sun, puede descargar la implementación Tomcat de los servlets y JSPs de Java, que es una implementación gratuita, de código fuente abierto, y que es la implementación de referencia oficial de Sun. Puede encontrarse en *jakarta.apache.org*.

Siga las instrucciones de instalación de la implementación Tomcat, después edite el archivo **server.xml** para que apunte a la localización de su árbol de directorios en el que se ubicarán los servlets. Una vez que se arranque el programa Tomcat, se pueden probar los programas de servlets.

Ésta sólo ha sido una somera introducción a los servlets; hay libros enteros sobre esta materia. Sin embargo, esta introducción debería proporcionarse las suficientes ideas como para que se inicie. Además, muchas ideas de la siguiente sección son retrocompatibles con los servlets.

Java Server Pages

Las *Java Server Pages* (JSP) son una extensión del estándar Java definido sobre las Extensiones de servlets. La meta de las JSP es la creación y gestión simplificada de páginas web dinámicas.

La implementación de referencia Tomcat anteriormente mencionada y disponible gratuitamente en *jakarta.apache.org* soporta JSP automáticamente.

Las JSP permite combinar el HTML de una página web con fragmentos de código Java en el mismo documento. El código Java está rodeado de etiquetas especiales que indican al contenedor de JSP que debería usar el código para generar un servlet o parte de uno. El beneficio de las JSP es que se puede mantener un documento único que representa tanto la página como el código Java que habilita. La pega es que el mantenedor de la página JSP debe dominar tanto HTML como Java (sin embargo, los entornos constructores de IGU para JSP deberían aparecer en breve).

La primera vez que el contenedor de JSP carga un JSP (que suele estar asociado con, o ser parte de, un servidor web) se genera, compila y carga automáticamente en el contenedor de servlets el código servlet necesario para cumplimentar las etiquetas JSP. Las porciones estáticas de la página HTML se producen enviando objetos **String** estáticos a **write( )**. Las porciones dinámicas se incluyen directamente en el servlet.

A partir de ese momento, y mientras el código JSP de la página no se modifique, se comporta como si fuera una página HTML estática con servlets asociados (sin embargo, el servlet genera todo el código HTML). Si se modifica el código fuente de la JSP, ésta se recompila y recarga automáticamente la siguiente vez que se solicite esa página. Por supuesto, debido a todo este dinamismo, se apreciará una respuesta lenta en el acceso por primera vez a una JSP. Sin embargo, dado que una JSP suele usarse mucho más a menudo que ser cambiada, normalmente uno no se verá afectado por este retraso.

La estructura de una página JSP está a caballo entre la de un servlet y la de una página HTML. Las etiquetas JSP empiezan y acaban con "<" y ">", como las etiquetas HTML, pero las etiquetas también incluyen símbolos de porcentaje, de forma que todas las etiquetas JSP se delimitan por:

```
<% código JSP aquí%>
```

El signo de porcentaje precedente puede ir seguido de otros caracteres que determinen el tipo específico de código JSP de la etiqueta.

He aquí un ejemplo extremadamente simple que usa una llamada a la biblioteca estándar Java para lograr la hora actual en milisegundos, que es después dividida por mil para producir la hora en segundos. Dado que se usa una *expresión JSP* (la <%=), el resultado del cálculo se fuerza a un **String** y se coloca en la página web generada:

```
//:! C15:jsp:MostrarSegundos.jsp
<html><body>
<H1>El tiempo en segundos es:
<%= System.currentTimeMillis()/1000 %></H1>
```

```
</body></html>
///:~
```

En los ejemplos de JSP de este libro no se incluyen la primera y última líneas en el archivo de código extraído y ubicado en el árbol de códigos fuente del libro.

Cuando el cliente crea una petición de la página JSP hay que haber configurado el servidor web para confiar en la petición del contenedor de JSP que posteriormente invoca a la página. Como se mencionó arriba, la primera vez que se invoca la página, el contenedor de JSP genera y compila los componentes especificados por la página como uno o más servlets. En el ejemplo de arriba, el servlet contendrá código para configurar el objeto **HttpServletResponse**, producir un objeto **PrintWriter** (que siempre se denomina **out**), y después convertir el cómputo de tiempo en un **String** que es enviado a **out**. Como puede verse, todo esto se logra con una sentencia muy sucinta, pero el programador HTML/diseñador web medio no tendrá las aptitudes necesarias para escribir semejante código.

Objetos implícitos

Los servlets incluyen clases que proporcionan utilidades convenientes, como **HttpServletRequest**, **HttpServletResponse**, **Session**, etc. Los objetos de estas clases están construidos en la especificación JSP y automáticamente disponibles para ser usados en un JSP, sin tener que escribir ninguna línea extra de código. Los objetos implícitos de un JSP se detallan en la tabla siguiente:

Variable implícita	De tipo (javax.servlet)	Descripción	Ámbito
request	Subtipo de protocolo dependiente de **HttpServletRequest**	La petición que dispara la invocación del servicio.	petición
response	Subtipo de protocolo dependiente de **HttpServletResponse**	La respuesta a la petición.	página
pageContext	**jsp.PageContext**	El contexto de la página encapsula facetas dependientes de la implementación y proporciona métodos de conveniencia y acceso de espacio de nombres a este JSP.	página
session	Subtipo de protocolo dependiente de **http.HttpSession**	El objeto sesión creado para el cliente que hace la petición. Ver el objeto servlet Session.	sesión

Variable implícita	De tipo (javax.servlet)	Descripción	Ámbito
application	**ServletContext**	El contexto de servlet obtenido del objeto de configuración del servlet (por ejemplo, **getServletConfig( ), getContext( )**).	aplicación
out	**jsp.JspWriter**	El objeto que escribe en el flujo de salida.	página
config	**ServletConfig**	El **ServletConfig** de este JSP.	página
page	**java.lang.Object**	La instancia de la clase de implementación de esta página que procese la petición actual.	página

El ámbito de cada objeto puede variar significativamente. Por ejemplo, el objeto **session** tiene un ámbito que excede al de una página, pues puede abarcar varias peticiones de clientes y páginas. El objeto **application** puede proporcionar servicios a un grupo de páginas JSP que representan juntas una aplicación web.

Directivas JSP

Las directivas son mensajes al contenedor de JSP y se delimintan por la "@":

```
<%@ directiva {atr="valor"}* %>
```

Las directivas no envían nada al flujo **out**, pero son importantes al configurar los atributos y dependencias de una página JSP con el contenedor de JSP. Por ejemplo, la línea:

```
<% page language="java" %>
```

dice que el lenguaje de escritura de guiones que se está usando en la página JSP es Java. De hecho, la especificación de Java *sólo* describe las semánticas de guiones para atributos de lenguaje iguales a "Java". La intención de esta directiva es aportar flexibilidad a la tecnología JSP. En el futuro, si hubiera que elegir otro lenguaje, como Python (un buen lenguaje de escritura de guiones), entonces este lenguaje debería soportar el Entorno de Tiempo de Ejecución de Java, exponiendo el modelo de objetos de la tecnología Java al entorno de escritura de guiones, especialmente las variables implícitas definidas arriba, las propiedades de los JavaBeans y los métodos públicos.

La directiva más importante es la directiva de página. Define un número de atributos dependientes de la página y comunica estos atributos al contenedor de JSP. Entre estos atributos se incluye: **language**, **extends**, **import**, **session**, **buffer**, **autoFlush**, **isThreadSafe**, **info** y **errorPage**. Por ejemplo:

```
<%@ page session="true" import="java.util.*" %>
```

La primera línea indica que la página requiere participación en una sesión HTTP. Dado que no hemos establecido directiva de lenguaje, el contenedor de JSP usa por defecto Java y la variable de lenguaje de escritura denominada **session** es de tipo **javax.servlet.http.HttpSession**. Si la directiva hubiera sido falsa, la variable implícita **session** no habría estado disponible. Si no se especifica la variable **session**, se pone a "true" por defecto.

El atributo **import** describe los tipos disponibles al entorno de escritura de guiones. Este atributo se usa igual que en el lenguaje de programación Java, por ejemplo, una lista separada por comas de expresiones **import** normales. Esta lista es importada por la implementación de la página JSP traducida y está disponible para el entorno de escritura de guiones. De nuevo, esto sólo está definido verdaderamente cuando el valor de la directiva de lenguaje es "java".

Elementos de escritura de guiones JSP

Una vez que se han usado las directivas para establecer el entorno de escritura de guiones se puede usar los elementos del lenguaje de escritura de guiones. JSP 1.1 tiene tres elementos de lenguaje de escritura de guiones —*declaraciones*, *scriptlets* y *expresiones*. Una declaración declarará elementos, un scriptlet es un fragmento de sentencia y una expresión es una expresión completa del lenguaje. En JSP cada elemento de escritura de guiones empieza por "<%". La sintaxis de cada una es:

```
<%! declaracion %>
<% scriptlet %>
<%= expresión %>
```

El espacio en blanco tras "<%!", "<%", "<%=" y antes de "%>" es opcional.

Todas estas etiquetas se basan en XML; se podría incluso decir que una página JSP puede corresponderse con un documento XML. La sintaxis equivalente en XML para los elementos de escritura de guiones de arriba sería:

```
<jsp:declaracion> declaracion  </jsp:declaracion>
<jsp:scriptlet>  scriptlet   </jsp:scriptlet>
<jsp:expresion>  expresion </jsp:expresion>
```

Además, hay dos tipos de comentarios:

```
<%--comentario jsp --%>
<!--comentario html  -->
```

La primera forma permite añadir comentarios a las páginas fuente JSP que no aparecerán de ninguna forma en el HTML que se envía al cliente. Por supuesto, la segunda forma de comentario no es específica de los JSP —es simplemente un comentario HTML ordinario. Lo interesante es que se puede insertar código JSP dentro de un comentario HTML, y el comentario se producirá en la página resultante, incluyendo el resultado del código JSP.

Las declaraciones se usan para declarar variables y métodos en el lenguaje de escritura de guiones (actualmente sólo Java) usado en una página JSP. La declaración debe ser una sentencia Java com-

pleta y no puede producir ninguna salida en el flujo **salida**. En el ejemplo **Hola.jsp** de debajo, las declaraciones de las variables **cargaHora**, **cargaFecha** y **conteoAccesos** son sentencias Java completas que declaran e inicializan nuevas variables.

```
//:! C15:jsp:Hola.jsp
<%-- Este comentario JSP no aparecera en el HTML generado --%>
<%-- Esto es una directiva JSP: --%>
<%@ page import="java.util.*" %>
<%-- Estas son declaraciones: --%>
<%!
     long cargaHora= System.currentTimeMillis();
     Date cargaFecha = new Date();
     int conteoAccesos = 0;
%>
<html><body>
<%-- Las siguientes lineas son el resultado de una
expression JSP insertada en el html generado;
el '=' indica una expresion JSP --%>
<H1>Esta pagina fue cargada el <%= cargaFecha %> </H1>
<H1>¡Hola, Mundo! Hoy es <%= new Date() %></H1>
<H2>He aqui un objeto: <%= new Object() %></H2>
<H2>Esta pagina ha estado activa
<%= (System.currentTimeMillis()-cargaHora)/1000 %>
segundos</H2>
<H3>Esta pagina ha sido accedida <%= ++conteoAccesos %>
veces desde <%= cargaFecha %></H3>
<%-- Un "scriptlet" que escribe a la consola
servidora y a la pagina cliente.
Notese que se require el ';': --%>
<%
    System.out.println("Adios");
    out.println("Cheerio");
%>
</body></html>
///:~
```

Cuando se ejecute este programa se verá que las variables **cargaHora**, **cargaFecha** y **conteoAccesos** guardan sus valores entre accesos a la página, por lo que son claramente campos y no variables locales.

Al final del ejemplo hay un scriptlet que escribe "Adios" a la consola servidora web y "Cheerio" al objeto **JspWriter** implícito **out**. Los scriptlets pueden contener cualquier fragmento de código que sean sentencias Java válidas. Los scriptlets se ejecutan bajo demanda en tiempo de procesamiento. Cuando todos los fragmentos de scritplet en un JSP dado se combinan en el orden en que aparecen en la página JSP, deberían conformar una sentencia válida según la definición del lenguaje de programación Java. Si producen o no alguna salida al flujo **out** depende del código del scriptlet. Uno de-

bería ser consciente de que los scriptlets pueden producir efectos laterales al modificar objetos que son visibles para ellos.

Las expresiones JSP pueden entremezclarse con el HTML en la sección central de **Hola.jsp**. Las expresiones deben ser sentencias Java completas, que son evaluadas, convertidas a un **String** y enviadas a **out**. Si el resultado no puede convertirse en un **String**, se lanza una **ClassCastException**.

Extraer campos y valores

El ejemplo siguiente es similar a uno que se mostró anteriormente en la seccion de servlets. La primera vez que se acceda a la página, se detecta que no se tienen campos y se devuelve una página que contiene un formulario, usando el mismo código que en el ejemplo de los servlets, pero en formato JSP. Cuando se envía el formulario con los campos rellenos al mismo URL de JSP, éste detecta los campos y los muestra. Ésta es una técnica brillante pues permite tener tanto la página que contiene el formulario para que el usuario la rellene como el código de respuesta para esa página en un único archivo, facilitando así la creación y mantenimiento.

```
//:! c15:jsp:MostarDatosFormulario.jsp
<%-- Tomando los datos de un formulario HTML. --%>
<%-- Este JSP tambien genera el formulario. --%>
<%@ page import="java.util.*" %>
<html><body>
<H1>MostarDatosFormulario</H1><H3>
<%
  Enumeration campos = request.getParameterNames();
  if(!campos.hasMoreElements()) { // No hay campos %>
    <form method="POST"
    action="MostarDatosFormulario.jsp">
<%  for(int i = 0; i < 10; i++) {  %>
      Campo<%=i%>: <input type="text" size="20"
      name="Campo<%=i%>" value="Valor<%=i%>"><br>
<%  } %>
    <INPUT TYPE=submit name=someter
    value="Someter"></form>
<%} else {
    while(campos.hasMoreElements()) {
      String campo = (String)campos.nextElement();
      String valor = request.getParameter(campo);
%>
      <li><%= campo %> = <%= valor %></li>
<%  }
  } %>
</H3></body></html>
///:~
```

El aspecto más interesante de este ejemplo es que demuestra cómo puede entremezclarse el código scriptlet con el código HTML incluso hasta el punto de generar HTML dentro de un bucle **for** de Java. Esto es especialmente conveniente para construir cualquier tipo de formulario en el que, de lo contrario, se requeriría código HTML repetitivo.

Atributos JSP de página y su ámbito

Merodeando por la documentación HTML de servlets y JSP, se pueden encontrar facetas que dan información sobre el servlet o el JSP actualmente en ejecución. El ejemplo siguiente muestra uno de estos fragmentos de datos:

```
//:! c15:jsp:ContextoPagina.jsp
<%--Viendo los atributos de ContextoPagina--%>
<%-- Notese que se puede incluir cualquier cantidad de codigo
dentro de las etiquetas de scriptlet --%>
<%@ page import="java.util.*" %>
<html><body>
NombreServlet: <%= config.getServletName() %><br>
El contenedor de servlets soporta la version:
<% out.print(application.getMajorVersion() + "."
+ application.getMinorVersion()); %><br>
<%
  session.setAttribute("Mi perro", "Ralph");
  for(int ambito = 1; ambito <= 4; ambito++) {   %>
    <H3>Ambito: <%= ambito %> </H3>
<%  Enumeration e =
      pageContext.getAttributeNamesInScope(ambito);
    while(e.hasMoreElements()) {
      out.println("\t<li>" +
        e.nextElement() + "</li>");
    }
  }
%>
</body></html>
///:~
```

Este ejemplo también muestra el uso tanto del HTML embebido como de la escritura en **out** para sacar la página HTML resultante.

El primer fragmento de información que se produce es el nombre del servlet, que probablemente será simplemente "JSP", pero depende de la implementación. También se puede descubrir la versión actual del contenedor de servlets usando el objeto aplicación. Finalmente, tras establecer un atributo de sesión, se muestran los "nombres de atributo" de un ámbito en particular. Los ámbitos no se usan mucho en la mayoría de programas JSP; simplemente se muestran aquí para añadir interés al ejemplo. Hay cuatro ámbitos de atributos, que son: el *ámbito de página* (ámbito 1), el *ám-*

bito de petición (ámbito 2), el *ámbito de sesión* (ámbito 3) —aquí, el único elemento disponible en ámbito de sesión es "Mi perro", añadido justo antes del bucle **for**, y el *ámbito de aplicación* (ámbito 4), basado en el objeto **ServletContext**. Sólo hay un **ServletContext** por "aplicación web" por cada Máquina Virtual Java. (Una "aplicación web" es una colección de servlets y contenido instalados bajo un subconjunto del espacio de nombres URL del servidor, como /catalog. Esto se suele establecer utilizando un archivo de configuración.) En el ámbito de aplicación se verán objetos que representan rutas para el directorio de trabajo y el directorio temporal.

Manipular sesiones en JSP

Las sesiones se presentaron en la sección anterior de los servlets, y también están disponibles dentro de los JSP. El ejemplo siguiente ejercita el objeto **session** y permite manipular la cantidad de tiempo antes de que la sesión se vuelva no válida.

```
//:! c15:ObjetoSesion.jsp
<%--Recuperando y estableciendo valores de objetos session --%>
<html><body>
<H1>IDSesion : <%= session.getId() %></H1>
<H3><li>Esta sesion se creo el
<%= session.getCreationTime() %></li></H1>
<H3><li>Intervalo Maximo de Inactividad anterior =
  <%= session.getMaxInactiveInterval() %></li>
<% session.setMaxInactiveInterval(5); %>
<li>Nuevo intervalo maximo de inactividad=
  <%= session.getMaxInactiveInterval() %></li>
</H3>
<H2>Si el objeto sesion "Mi perro" sigue vivo,
este valor sera distinto de null: <H2>
<H3><li>Valor de sesion para "Mi perro" =
<%= session.getAttribute("Mi perro") %></li></H3>
<%-- Ahora añadir el objeto sesion "Mi perro" --%>
<% session.setAttribute("Mi perro",
                        new String("Ralph")); %>
<H1>El nombre de mi perro es
<%= session.getAttribute("Mi perro") %></H1>
<%-- Ver si "Mi perro" pasa a otra forma --%>
<FORM TYPE=POST ACTION=ObjetoSesion2.jsp>
<INPUT TYPE=submit name=someter
Value="Invalidar"></FORM>
<FORM TYPE=POST ACTION=ObjetoSesion3.jsp>
<INPUT TYPE=submit name=Someter
Value="Mantener"></FORM>
</body></html>
///:~
```

El objeto **session** se proporciona por defecto, por lo que está disponible sin necesidad de codificación extra. Las llamadas a **getId( )**, **getCreationTime( )** y **getMaxInactiveInterval( )** se usan para mostrar información sobre este objeto **session**.

Cuando se trae por primera vez esta sesión se verá un **MaxInactiveInterval** de, por ejemplo, 1800 segundos (30 minutos). Esto dependerá de la forma en que esté configurado el contenedor de JSP/servlets. El **MaxInactiveInterval** se acorta a 5 segundos para que las cosas parezcan interesantes. Si se refresca la página antes de que expire el intervalo de 5 segundos, se verá:

```
Valor de sesion para "Mi perro" = Ralph
```

Pero si se espera más que eso, "Ralph" se convertirá en **null**.

Para ver cómo se puede traer la información de sesión a otras páginas, y para ver también el efecto de invalidar un objeto de sesión *frente a* simplemente dejar que expire, se crean otros dos JSP. El primero (al que se llega presionando el botón "invalidar" de **ObjetoSesion.jsp**) lee la información de sesión y después invalida esa sesión explícitamente:

```
//:! c15:jsp:ObjetoSesion2.jsp
<%--El objeto sesion se arrastra --%>
<html><body>
<H1>ID Sesion: <%= session.getId() %></H1>
<H1>Valor de sesion para "Mi perro"
<%= session.getValue("Mi perro") %></H1>
<% session.invalidate(); %>
</body></html>
///:~
```

Para experimentar con esto, refresque **ObjetoSesion.jsp**, y después pulse inmediatamente en el objeto "invalidar" para ir a **ObjetoSesion2.jsp**. En este momento se verá "Ralph", y después (antes de que haya expirado el intervalo de 5 segundos), refresque **ObjetoSesion2.jsp** para ver que la sesión ha sido invalidada a la fuerza y que "Ralph" ha desaparecido.

Si se vuelve a **ObjetoSesion.jsp**, refresque la página, de forma que se tenga un nuevo intervalo de 5 segundos, desués presione el botón "Mantener", que le llevará a la siguiente página, **ObjetoSesion3.jsp**, que NO invalida la sesión:

```
//:! c15:jsp:ObjetoSesion3.jsp
<%--El objeto sesion se arrastra --%>
<html><body>
<H1>ID Sesion: <%= session.getId() %></H1>
<H1>Valor de sesion para "Mi perro"
<%= session.getValue("Mi perro") %></H1>
<FORM TYPE=POST ACTION=ObjetoSesion.jsp>
<INPUT TYPE=submit name=someter Value="volver">
</FORM>
</body></html>
```

```
///:~
```

Dado que esta página no invalida la sesión, "Ralph" merodeará por ahí mientras se siga refrescando la página, antes de que expire el intervalo de 5 segundos. Esto es semejante a una mascota "Tomagotchi" —en la medida en que se juega con "Ralph", sigue vivo, si no, expira.

Crear y modificar cookies

Las cookies se presentaron en la sección anterior relativa a servlets. De nuevo, la brevedad de los JSP hace que jugar con las cookies aquí sea mucho más sencillo que con el uso de servlets. El ejemplo siguiente muestra esto cogiendo las cookies que vienen con la petición, leyendo y modificando sus edades máximas (fechas de expiración) y adjuntando una Cookie a la respuesta saliente:

```
//:! c15:jsp:Cookies.jsp
<%--¡Este programa tiene distintos comportamientos con
 los distintos navegadores! --%>
<html><body>
<H1>IDsesion: <%= session.getId() %></H1>
<%
Cookie[] cookies = request.getCookies();
for(int i = 0; i < cookies.length; i++) { %>
  Cookie nombre: <%= cookies[i].getName() %> <br>
  valor: <%= cookies[i].getValue() %><br>
  Vieja edad maxima en segundos:
  <%= cookies[i].getMaxAge() %><br>
  <% cookies[i].setMaxAge(5); %>
  Nueva edad maxima en segundos:
  <%= cookies[i].getMaxAge() %><br>
<% } %>
<%! int conteo = 0; int conteop = 0; %>
<% response.addCookie(new Cookie(
     "Bob" + conteo++, "Perro" + conteop++)); %>
</body></html>
///:~
```

Dado que cada navegador almacena las cookies a su manera, se pueden ver distintos comportamientos con distintos navegadores (no puede asegurarse, pero podría tratarse de algún tipo de error solucionado para cuando se lea el presente texto). También podrían experimentarse resultados distintos si se apaga el navegador y se vuelve a arrancar en vez de visitar simplemente una página distinta y volver a **Cookies.jsp**. Nótese que usar objetos sesión parece ser más robusto que el uso directo de cookies.

Tras mostrar el identificador de sesión, se muestra cada cookie del array de cookies que viene con el objeto **request**, junto con su edad máxima. Después se cambia la edad máxima y se muestra de nuevo para verificar el nuevo valor. A continuación se añade una nueva cookie a la respuesta. Sin embargo, el navegador puede parecer ignorar la edad máxima; merece la pena jugar con este pro-

grama y modificar el valor de la edad máxima para ver el comportamiento bajo distintos navegadores.

Resumen de JSP

Esta sección sólo ha sido un recorrido breve por los JSP, e incluso con lo aquí cubierto (junto con lo que se ha aprendido en el resto del libro, y el conocimiento que cada uno tenga de HTML) se puede empezar a escribir páginas web sofisticadas vía JSP. La sintaxis de JSP no pretende ser excepcionalmente profunda o complicada, por lo que si se entiende lo que se ha presentado en esta sección, uno ya puede ser productivo con JSP. Se puede encontrar más información en los libros más actuales sobre servlets o en *java.sun.com*.

Es especialmente bonito tener disponibles los JSP, incluso si la meta es producir servlets. Se descubrirá que si se tiene una pregunta sobre el comportamiento de una faceta servlet, es mucho más fácil y sencillo escribir un programa de pruebas de JSP para responder a esa cuestión, que escribir un servlet. Parte del beneficio viene de tener que escribir menos código y de ser capaz de mezclar el HTML con el código Java, pero la mayor ventaja resulta especialmente obvia cuando se ve que el contenedor JSP maneja toda la recompilación y recarga de JSP automáticamente siempre que se cambia el código fuente.

Siendo los JSP tan terroríficos, sin embargo, merece la pena ser conscientes de que la creación de JSP requiere de un nivel de talento más elevado que el necesario para simplemente programar en Java o crear páginas web. Además, depurar una página JSP que no funciona no es tan fácil como depurar un programa Java, puesto que (actualmente) los mensajes de error son más oscuros. Esto podría cambiar al mejorar los sistemas de desarrollo, pero también puede que veamos otras tecnologías construidas sobre Java y la Web que se adapten mejor a las destrezas del diseñador de sitios web.

RMI (Invocation Remote Method)

Los enfoques tradicionales a la ejecución de código en otras máquinas a través de una red siempre han sido confusos a la vez que tediosos y fuentes de error a la hora de su implementación. La mejor forma de pensar en este problema es que algún objeto resulte que resida en otra máquina, y que se pueda enviar un mensaje al objeto remoto y obtener un resultado exactamente igual que si el objeto viviera en la máquina local. Esta simplificación es exactamente lo que permite hacer el *Remote Method Invocation* (RMI) de Java. Esta sección recorre los pasos necesarios para que cada uno cree sus propios objetos RMI.

Interfaces remotos

RMI hace un uso intensivo de las interfaces. Cuando se desea crear un objeto remoto, se enmascara la implementación subyacente pasando una interfaz. Por consiguiente, cuando el cliente obtiene una referencia a un objeto remoto, lo que verdaderamente logra es una referencia a una interfaz,

que resulta estar conectada a algún fragmento de código local que habla a través de la red. Pero no hay que pensar en esto, sino simplemente en enviar mensajes vía la referencia a la interfaz.

Cuando se cree una interfaz remota, hay que seguir estas normas:

1. La interfaz remota debe ser **public** (no puede tener "acceso package" es decir, no puede ser "amigo". De otra forma, el cliente obtendría un error al intentar cargar un objeto remoto que implemente la interfaz remota.
2. La interfaz remota debe extender la interfaz **java.rmi.Remote**.
3. Cada método de la interfaz remota debe declarar **java.rmi.RemoteException** en su cláusula **throws,** además de cualquier excepción específica de la aplicación.
4. Todo objeto remoto pasado como parámetro o valor de retorno (bien directamente o bien embebido en un objeto local) debe declararse como la interfaz remota, no como la clase implementación.

He aquí una interfaz remota simple que representa un servicio de tiempo exacto:

```
//: c15:rmi:ITiempoPerfecto.java
// La interfaz remota TiempoPerfecto.
package c15.rmi;
import java.rmi.*;

interface ITiempoPerfecto extends Remote {
  long obtenerTiempoPerfecto() throws RemoteException;
} ///:~
```

Tiene el mismo aspecto que cualquier otra interfaz, excepto por extender **Remote** y porque todos sus métodos lanzan **RemoteException**. Recuérdese que una **interfaz** y todos sus métodos son automáticamente **public**.

Implementar la interfaz remota

El servidor debe contener una clase que extienda **UnicastRemoteObject** e implementar la interfaz remota. Esta clase también puede tener métodos adicionales, pero sólo los métodos de la interfaz remota están disponibles al cliente, por supuesto, dado que el cliente sólo obtendrá una referencia al interfaz, y no a la clase que lo implementa.

Hay que definir explícitamente el constructor para el objeto remoto, incluso si sólo se está definiendo un constructor por defecto que invoque al constructor de la clase base. Hay que escribirlo puesto que debe lanzar **RemoteException**.

He aquí la implementación de la interfaz remota **ITiempoPerfecto:**

```
//: c15:rmi:TiempoPerfecto.java
// La implementación del objeto
```

```
// remoto TiempoPerfecto.
package c15.rmi;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;

public class TiempoPerfecto
    extends UnicastRemoteObject
    implements ITiempoPerfecto {
  // Implementación de la interfaz:
  public long obtenerTiempoPerfecto()
      throws RemoteException {
    return System.currentTimeMillis();
  }
  // Debe implementar el constructor
  // para lanzar RemoteException:
  public TiempoPerfecto() throws RemoteException {
    // super(); // Invocado automáticamente
  }
  // Registro para el servicio RMI. Lanza
  // excepciones a la consola.
  public static void main(String[] args)
  throws Exception {
    System.setSecurityManager(
    new RMISecurityManager());
    TiempoPerfecto tp = new TiempoPerfecto();
    Naming.bind(
      "//pepe:2005/TiempoPerfecto", tp);
    System.out.println("Preparado para dar la hora");
  }
} ///:~
```

Aquí, **main( )** maneja todos los detalles de establecimiento del servidor. Cuando se sirven objetos RMI, en algún momento del programa hay que:

1. Crear e instalar un gestor de seguridad que soporte RMI. El único disponible para RMI como parte de la distribución JAVA es **RMISecurityManager**.

2. Crear una o más instancias de un objeto remoto. Aquí, puede verse la creación del objeto **TiempoPerfecto**.

3. Registrar al menos uno de los objetos remotos con el registro de objetos remotos RMI para propósitos de reposición. Un objeto remoto puede tener métodos que produzcan referencias a otros objetos remotos. Esto permite configurarlo de forma que el cliente sólo tenga que ir al registro una vez para lograr el primer objeto remoto.

Configurar el registro

Aquí se ve una llamada al método **static Naming.bind( )**. Sin embargo, esta llamada requiere que el registro se esté ejecutando como un proceso separado en el computador. El nombre del servidor de registros es **rmiregistry**, y bajo Windows de 32 bits se dice:

```
start rmiregistry
```

para que arranque en segundo plano. En Unix, el comando es:

```
rmiregistry &
```

Como muchos programas de red, el **rmiregistry** está ubicado en la dirección IP de cualquier máquina que lo arranque, pero también debe estar escuchando por un puerto. Si se invoca al **rmiregistry** como arriba, sin argumentos, el puerto del registro por defecto será 1099. Si se desea que esté en cualquier otro puerto, se añade un argumento a la línea de comandos para especificar el puerto. Para este ejemplo, el puerto está localizado en el 2005, de forma que bajo Windows de 32 bits el **rmiregistry** debería empezarse así:

```
start rmiregistry 2005
```

o en el caso de Unix:

```
rmiregistry 2005 &
```

La información sobre el puerto también debe proporcionarse al comando **bind( )**, junto con la dirección IP de la máquina en la que está ubicado el registro. Pero esto puede ser un problema frustrante si se desea probar programas RMI de forma local de la misma forma en que se han probado otros programas de red hasta este momento en el presente capítulo. En la versión 1.1 del JDK, hay un par de problemas[5]:

1. **localhost** no funciona con RMI. Por consiguiente, para experimentar con RMI en una única máquina, hay que proporcionar el nombre de la máquina. Para averiguar el nombre de la máquina bajo Windows de 32 bits, se puede ir al panel de control y seleccionar "Red". Después, se selecciona la solapa "Identificación", y se dispondrá del nombre del computador. En nuestro caso, llamamos a mi computador "Pepe". El uso de mayúsculas y minúsculas parece ignorarse.

2. RMI no funcionará a menos que el computador tenga una conexión TCP/IP activa, incluso si todos los componentes simplemente se comunican entre sí en la máquina local. Esto significa que hay que conectarse al proveedor de servicios de Internet antes de intentar ejecutar el programa o se obtendrán algunos mensajes de excepción siniestros.

Con todo esto en mente, el comando **bind( )** se convierte en :

```
Naming.bind("//pepe:2005/TiempoPerfecto", tp);
```

[5] Para descubrir esta información fueron muchas las neuronas que sufrieron una muerte agónica.

Si se está usando el puerto por defecto, el 1099, no hay que especificar un puerto, por lo que podría decirse:

```
Naming.bind("//pepe/TiempoPerfecto", tp);
```

Se deberían poder hacer pruebas locales usando sólo el identificador:

```
Naming.bind("TiempoPerfecto", tp);
```

El nombre del servicio es arbitrario; resulta que en este caso es TiempoPerfecto, exactamente igual que el nombre de la clase, pero se le podría dar el nombre que se desee. Lo importante es que sea un nombre único en el registro que el cliente conozca, para buscar el objeto remoto. Si el nombre ya está en el registro, se obtiene una **AlreadyBoundException**. Para evitar esto, se puede usar siempre **rebind( )** en vez de **bind( )**, puesto que **rebind( ),** o añade una nueva entrada o reemplaza una ya existente.

Incluso aunque exista **main( )**, el objeto se ha creado y registrado, por lo que se mantiene vivo por parte del registro, esperando a que venga un cliente y lo solicite. Mientras se esté ejecutando el **rmiregistry** y no se invoque a **Naming.unbind( )** para ese nombre, el objeto estará ahí. Por esta razón, cuando se esté desarrollando código hay que apagar el **rmiregistry** y volver a arrancarlo al compilar una nueva versión del objeto remoto.

Uno no se ve forzado a arrancar **rmiregistry** como un proceso externo. Si se sabe que una aplicación es la única que va a usar el registro, se puede arrancar dentro del programa con la línea:

```
LocateRegistry.createRegistry(2005);
```

Como antes, 2005 es el número de puerto que usamos en este ejemplo. Esto equivale a ejecutar **rmiregistry** 2005 desde la línea de comandos, pero a menudo puede ser más conveniente cuando se esté desarrollando código RMI, pues elimina los pasos adicionales de arrancar y detener el registro. Una vez ejecutado este código, se puede invocar **bind( )** usando **Naming** como antes.

Crear stubs y skeletons

Si se compila y ejecuta **TiempoPerfecto.java**, no funcionará incluso aunque el **rmiregistry** se esté ejecutando correctamente. Esto se debe a que todavía no se dispone de todo el marco para RMI. Hay que crear primero los *stubs* y *skeletons* que proporcionan las operaciones de conexión de red y que permiten fingir que el objeto remoto es simplemente otro objeto local de la máquina.

Lo que ocurre tras el telón es complejo. Cualquier objeto que se pase o que sea devuelvo por un objeto remoto debe **implementar Serializable** (si se desea pasar referencias remotas en vez de objetos enteros, los parámetros objeto pueden **implementar Remote**), por lo que se puede imaginar que los *stubs y skeletons* están llevando a cabo operaciones de serialización y deserialización automáticas, al ir mandando todos los parámetros a través de la red, y al devolver el resultado. Afortunadamente, no hay por qué saber nada de esto, pero *sí* que hay que crear los *stubs* y *skeletons*. Este proceso es simple: se invoca a la herramienta **rmic** para el código compilado, y ésta crea los archivos necesarios. Por tanto, el único requisito es añadir otro paso al proceso de compilación.

Sin embargo, la herramienta **rmic** tiene un comportamiento particular para paquetes y *classpaths*. **TiempoPerfecto.java** está en el **package c15.rmi**, e incluso si se invoca a **rmic** en el mismo directorio en el que está localizada **TiempoPerfecto.class**, **rmic** no encontrará el archivo, puesto que busca el *classpath*. Por tanto, hay que especificar las localizaciones distintas al *classpath*, como en:

```
rmic c15.rmi.TiempoPerfecto
```

No es necesario estar en el directorio que contenga **TiempoPerfecto.class** cuando se ejecute este comando, si bien los resultados se colocarán en el directorio actual.

Cuando **rmic** se ejecuta con éxito, se tendrán dos nuevas clases en el directorio:

```
TiempoPerfecto_Stub.class
TiempoPerfecto_Skel.class
```

correspondientes al *stub* y al *skeleton*. Ahora, ya estamos listos para que el cliente y el servidor se intercomuniquen.

Utilizar el objeto remoto

Toda la motivación de RMI es simplificar el uso de objetos remotos. Lo único extra que hay que hacer en el programa cliente es buscar y capturar la interfaz remota desde el servidor. A partir de ese momento, no hay más que programación Java ordinaria: envío de mensajes a objetos. He aquí el programa que hace uso de **TiempoPerfecto**:

```
//: c15:rmi:MostrarTiempoPerfecto.java
// Usa el objeto remoto TiempoPerfecto.
package c15.rmi;
import java.rmi.*;
import java.rmi.registry.*;

public class MostrarTiempoPerfecto {
  public static void main(String[] args)
  throws Exception {
    System.setSecurityManager(
      new RMISecurityManager());
    ITiempoPerfecto t =
      (ITiempoPerfecto)Naming.lookup(
        "//pepe:2005/TiempoPerfecto");
    for(int i = 0; i < 10; i++)
      System.out.println("TiempoPerfecto = " +
        t.obtenerTiempoPerfecto());
  }
} ///:~
```

La cadena de caracteres ID es la misma que la que se usó para registrar el objeto con **Naming**, y la primera parte representa al URL y al número de puerto. Dado que se está usando una URL también se puede especificar una máquina en Internet.

Lo que se devuelve de **Naming.lookup( )** hay que convertirlo a la interfaz remota, *no* a la clase. Si se usa la clase en su lugar, se obtendrá una excepción.

En la llamada a método:

```
t.obtenerTiempoPerfecto()
```

puede verse que una vez que se tiene una referencia al objeto remoto, la programación con él no difiere de la programación con un objeto local (con una diferencia: los métodos remotos lanzan **RemoteException**).

CORBA

En aplicaciones distribuidas grandes, las necesidades pueden no verse satisfechas con estos enfoques que acabamos de describir. Por ejemplo, uno podría querer interactuar con almacenes de datos antiguos, o podría necesitar servicios de un objeto servidor independientemente de su localizaron física. Estas situaciones requieren de algún tipo de *Remote Procedure Call* (RPC), y posiblemente de independencia del lenguaje. Es aquí donde CORBA puede ser útil.

CORBA no es un aspecto del lenguaje; es una tecnología de integración. Es una especificación que los fabricantes pueden seguir para implementar productos de integración compatibles con CORBA. Éste es parte del esfuerzo de la OMG para definir un marco estándar para interoperabilidad de objetos distribuidos independientemente del lenguaje.

CORBA proporciona la habilidad de construir llamadas a procedimientos remotos en objetos Java y no Java, y de interactuar con sistemas antiguos de forma independiente de la localización. Java añade soporte a redes y un lenguaje orientado a objetos perfecto para construir aplicaciones gráficas o no. El modelo de objetos de Java y el de la OMG se corresponden perfectamente entre sí; por ejemplo, ambos implementan el concepto de interfaz y un modelo de objetos de referencia.

Fundamentos de CORBA

A la especificación de la interoperabilidad entre objetos desarrollada por la OMG se le suele denominar la Arquitectura de Gestión de Objetos (OMA, *Object Management Architecture*). La OMA define dos conceptos: el *Core Object Model* y la *OMA Reference Architecture*. El primero establece los conceptos básicos de un objeto, interfaz, operación, etc. (CORBA es un refinamiento del *Core Object Model*). La OMA *Reference Architecture* define una infraestructura de servicios y mecanismos subyacentes que permiten interoperar a los objetos. Incluye el *Object Request Broker* (ORB), *Object Services* (conocidos también como *CORBA services*) y facilidades generales.

El ORB es el canal de comunicación a través del cual unos objetos pueden solicitar servicios a otros, independientemente de su localización física. Esto significa que lo que parece una llamada a un mé-

todo en el código cliente es, de hecho, una operación compleja. En primer lugar, debe existir una conexión con el objeto servidor, y para crear la conexión el ORB debe saber dónde reside el código que implementa ese servidor. Una vez establecida la conexión, hay que pasar los parámetros del método, por ejemplo, convertidos en un flujo binario que se envía a través de la red. También hay que enviar otra información como el nombre de la máquina servidora, el proceso servidor y la identidad del objeto servidor dentro de ese proceso. Finalmente, esta información se envía a través de un protocolo de bajo nivel, se decodifica en el lado servidor y se ejecuta la llamada. El ORB oculta toda esta complejidad al programador y hace la operación casi tan simple como llamar a un método de un objeto local. No hay ninguna especificación que indique cómo debería implementarse un nucleo ORB, pero para proporcionar compatibilidad básica entre los ORB de los diferentes vendedores, la OMG define un conjunto de servicios accesibles a través de interfaces estándar.

Lenguaje de Definición de Interfaces CORBA (IDL)

CORBA está diseñado para lograr la transparencia del lenguaje: un objeto cliente puede invocar a métodos de un objeto servidor de distinta clase, independientemente del lenguaje en que estén implementados. Por supuesto, el objeto cliente debe conocer los nombres y signaturas de los métodos que expone el objeto servidor. Es aquí donde interviene el IDL. El CORBA IDL es una forma independiente del lenguaje de especificar tipos de datos, atributos, operaciones, interfaces y demás. La sintaxis IDL es semejante a la de C++ o Java. La tabla siguiente muestra la correspondencia entre algunos de los conceptos comunes a los tres lenguajes, que pueden especificarse mediante CORBA IDL:

CORBA IDL	Java	C++
Módulo	Paquete	Espacio de nombre
Interfaz	Interfaz	Clase abstracta pura
Método	Método	Función miembro

También se soporta el concepto de herencia, utilizando el operador "dos puntos" como en C++. El programador escribe una descripción IDL de los atributos, métodos e interfaces implementados y usados por el servidor y los clientes. Después, se compila el IDL mediante un compilador IDL/Java proporcionado por un fabricante, que lee el fuente IDL y genera código Java.

El compilador IDL es una herramienta extremadamente útil: no genera simplemente un código fuente Java equivalente al IDL, sino que también genera el código que se usará para pasar los parámetros a métodos y para hacer llamadas remotas. A estos códigos se les llama *stub* y *skeleton*, y están organizados en múltiples archivos fuente Java, siendo generalmente parte del mismo paquete Java.

El servicio de nombres

El servicio de nombres es uno de los servicios CORBA fundamentales. A un objeto CORBA se accede a través de una referencia, un fragmento de información que no tiene significado para un lector humano. Pero a las referencias pueden asignarse nombres o cadenas de caracteres definidas por el programador. A esta operación se le denomina *encadenar la referencia*, y uno de los componentes de OMA, el Servicio de Nombres, se dedica exclusivamente a hacer conversiones y correspondencias cadena y objeto-a-cadena. Puesto que el servicio de nombres actúa como un directorio de teléfonos y, tanto servidores como clientes pueden consultarlo y manipularlo, se ejecuta como un proceso aparte. A la creación de una correspondencia objeto-a-cadena se le denomina vinculación de un objeto y a la eliminación de esta correspondencia de le denomina *desvinculación*. A la obtención de una referencia de un objeto pasando un String se le denomina *resolución de un nombre*.

Por ejemplo, al arrancar, una aplicación servidora podría crear un objeto servidor, establecer una correspondencia en el servicio de nombres y esperar después a que los clientes hagan peticiones. Un cliente obtiene primero una referencia al objeto servidor, resuelve el string, y después puede hacer llamadas al servidor haciendo uso de la referencia.

De nuevo, la especificación del servicio de nombres es parte de CORBA, pero la aplicación que lo proporciona está implementada por el fabricante del ORB. La forma de acceder a la funcionalidad del Servicio de Nombres puede variar de un fabricante a otro.

Un ejemplo

El código que se muestra aquí no será muy elaborado ya que distintos ORB tienen distintas formas de acceder a servicios CORBA, por lo que los ejemplos son específicos de los fabricantes. (El ejemplo de debajo usa JavaIDL, un producto gratuito de Sun que viene con un ORB ligero, un servicio de nombres, y un compilador de IDL a Java.) Además, dado que Java es un lenguaje joven y en evolución, no todas las facetas de CORBA están presentes en los distintos productos Java/CORBA.

Queremos implementar un servidor, en ejecución en alguna máquina, al que se pueda preguntar la hora exacta. También se desea implementar un cliente que pregunte por la hora exacta. En este caso, se implementarán ambos programas en Java, pero también se podrían haber usado dos lenguajes distintos (lo cual ocurre a menudo en situaciones reales).

Escribir el IDL fuente

El primer paso es escribir una descripción IDL de los servicios proporcionados. Esto lo suele hacer el programador del servidor, que es después libre de implementar el servidor en cualquier lenguaje en el que exista un compilador CORBA IDL. El archivo IDL se distribuye al programador de la parte cliente y se convierte en el puente entre los distintos lenguajes.

El ejemplo de debajo muestra la descripción IDL de nuestro servidor **TiempoExacto**:

```
//: c15:corba:TiempoExacto.idl
//# Hay que instalar el idltojava.exe de
//# java.sun.com y configurarlo para usar el
```

```
//# preprocesador C local para que compile
//# este archivo. Ver documentos de java.sun.com.
module tiemporemoto {
    interface TiempoExacto {
        string getTime();
    };
}; ///:~
```

Ésta es la declaración de una interfaz **TiempoExacto** de dentro del espacio de nombres **tiemporemoto**. La interfaz está formada por un único método que devuelve la hora actual en formato **string**.

Crear stubs y skeletons

El segundo paso es compilar el IDL para crear el código *stub* y el *skeleton* de Java que se usará para implementar el cliente y el servidor. La herramienta que viene con el producto JavaIDL es **idltojava**:

```
idltojava tiemporemoto.idl
```

Esto generará automáticamente el código, tanto para el *stub* como para el *skeleton*. **Idltojava** genera un **package** Java cuyo nombre se basa en el del módulo IDL, **tiemporemoto**, y los archivos Java generados se ponen en el subdirectorio **tiemporemoto**. El *skeleton* es **_TiempoExactoImplBase.java**, que se usará par implementar el objeto servidor, y **_TiempoExactoStub.java** se usará para el cliente. Hay representaciones Java de la interfaz IDL en **TiempoExacto.java** y un par de otros archivos de soporte que se usan, por ejemplo, para facilitar el acceso a operaciones de servicio de nombres.

Implementar el servidor y el cliente

Debajo puede verse el código correspondiente al lado servidor. La implementación del objeto servidor está en la clase **ServidorTiempoExacto**. El **ServidorTiempoRemoto** es la aplicación que crea un objeto servidor, lo registra en el ORB, le da un nombre a la referencia al objeto, y después queda a la espera de peticiones del cliente.

```
//: c15:corba:ServidorTiempoRemoto.java
import tiemporemoto.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.*;
import java.text.*;

// Implementación del objeto servidor
class ServidorTiempoExacto extends _TiempoExactoImplBase {
  public String obtenerTiempo(){
    return DateFormat.
        getTimeInstance(DateFormat.FULL).
          format(new Date(
              System.currentTimeMillis()));
```

```
    }
}

// Implementación de la aplicación remota
public class ServidorTiempoRemoto {
  // Lanza excepciones a la consola:
  public static void main(String[] args)
  throws Exception {
    // Creación e inicialización del ORB:
    ORB orb = ORB.init(args, null);
    // Crear el objeto servidor y registrarlo:
    ServidorTiempoExacto refObjServidorTiempo =
      new ServidorTiempoExacto();
    orb.connect(refObjServidorTiempo);
    // Conseguir el contexto de raíz de los nombres:
    org.omg.CORBA.Object refObj =
      orb.resolve_initial_references(
        "NameService");
    NamingContext ncRef =
      NamingContextHelper.narrow(refObj);
    // Asignar un string a la
    // referencia a objetos (ubicación):
    NameComponent nc =
      new NameComponent("TiempoExacto", "");
    NameComponent[] ruta = { nc };
    ncRef.rebind(ruta, refObjServidorTiempo);
    // Esperar peticiones de clientes:
    java.lang.Object sinc =
      new java.lang.Object();
    synchronized(sinc){
      sinc.wait();
    }
  }
} ///:~
```

Como puede verse, implementar el objeto servidor es simple; es una clase normal Java que se hereda del código *skeleton* generado por el compilador IDL. Las cosas se complican cuando hay que interactuar con el ORB y otros servicios CORBA.

Algunos servicios CORBA

He aquí una breve descripción de lo que está haciendo el código JavaIDL (ignorando principalmente la parte del código CORBA dependiente del vendedor). La primera línea del método **main( )** arranca el ORB, y por supuesto, esto se debe a que el objeto servidor necesitará interactuar con él. Justo después de la inicialización del ORB se crea un objeto servidor. De hecho, el término correcto sería un *objeto sirviente transitorio*: un objeto que recibe peticiones de clientes, cuya longevidad

es la misma que la del proceso que lo crea. Una vez que se ha creado el objeto sirviente transitorio, se registra en el ORB, lo que significa que el ORB sabe de su existencia y que puede ahora dirigirle peticiones.

Hasta este punto, todo lo que tenemos es **RefObjServidorTiempo**, una referencia a objetos conocida sólo dentro del proceso servidor actual. El siguiente paso será asignarle un nombre en forma de string a este objeto sirviente; los clientes usarán ese nombre para localizarlo. Esta operación se logra haciendo uso del Servicio de Nombres. Primero, necesitamos una referencia a objetos al Servicio de Nombres; la llamada a **resolve_initial_references( )** toma la referencia al objeto pasado a string del Servicio de Nombres, que es "NameService", en JavaIDL, y devuelve una referencia al objeto. Ésta se convierte a una referencia **NamingContext** específica usando el método **narrow( )**. Ahora ya pueden usarse los servicios de nombres.

Para vincular el objeto sirviente con una referencia a objetos pasada a string, primero se crea un objeto **NameComponent**, inicializado con "TiempoExacto", la cadena de caracteres que se desea vincular al objeto sirviente. Después, haciendo uso del método **rebind( )**, se vincula la referencia pasada a string a la referencia al objeto. Se usa **rebind( )** para asignar una referencia, incluso si ésta ya existe, mientras que **bind( )** provoca una excepción si la referencia ya existe. En CORBA un nombre está formado por varios NameContexts —por ello se usa un array para vincular el nombre a la referencia al objeto.

Finalmente, el objeto sirviente ya está listo para ser usado por los clientes. En este punto, el proceso servidor entra en un estado de espera. De nuevo, esto se debe a que es un sirviente transitorio, por lo que su longevidad podría estar confinada al proceso servidor. JavaIDL no soporta actualmente objetos persistentes —objetos que sobreviven a la ejecución del proceso que los crea.

Ahora que ya se tiene una idea de lo que está haciendo el código servidor, echemos un vistazo al código cliente:

```
//: c15:corba:ClienteTiempoRemoto.java
import tiemporemoto.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class ClienteTiempoRemoto {
  // Lanzar excepciones a la consola:
  public static void main(String[] args)
  throws Exception {
    // Creación e inicialización del ORB:
    ORB orb = ORB.init(args, null);
    // Obtener el contexto de nombrado raíz:
    org.omg.CORBA.Object refObj =
      orb.resolve_initial_references(
        "NameService");
    NamingContext ncRef =
      NamingContextHelper.narrow(refObj);
    // Lograr (resolver) la referencia del objeto
```

```
      // pasado a String para el servidor de hora:
      NameComponent nc =
        new NameComponent("TiempoExacto", "");
      NameComponent[] ruta = { nc };
      TiempoExacto refObjTiempo =
        TiempoExactoHelper.narrow(
          ncRef.resolve(ruta));
      // Hacer peticiones al objeto servidor:
      String tiempoExacto = refObjTiempo.obtenerTiempo();
      System.out.println(tiempoExacto);
    }
} ///:~
```

Las primeras líneas hacen los mismo que en el proceso servidor: se inicializa el ORB y se resuelve una referencia al servidor del Servicio de Nombres. Después se necesita una referencia a un objeto para el objeto sirviente, por lo que le pasamos una referencia a objeto pasada a String al método **resolve( )**, y convertimos el resultado en una referencia a la interfaz **TiempoExacto** usando el método **narrow( )**. Finalmente, invocamos a **obtenerTiempo( )**.

Activar el proceso de servicio de nombres

Finalmente, tenemos una aplicación servidora y una aplicación cliente listas para interoperar. Hemos visto que ambas necesitan el servicio de nombres para vincular y resolver referencias a objetos pasadas a String. Hay que arrancar el proceso de servicio de nombres antes de ejecutar o el servidor o el cliente. En JavaIDL el servicio de nombres es una aplicación Java que viene con el paquete del producto, pero puede ser distinta en cada producto. El servicio de nombres JavaIDL se ejecuta dentro de una instancia de la JVM y escucha por defecto al puerto 900 de red.

Activar el servidor y el cliente

Ahora ya se pueden arrancar las aplicaciones servidor y cliente (en este orden, pues el servidor es temporal). Si todo se configura bien, lo que se logra es una única línea de salida en la ventana de consola del cliente, con la hora actual. Por supuesto, esto puede que de por sí no sea muy excitante, pero habría que tener algo en cuenta: incluso si están en la misma máquina física, la aplicación cliente y la servidora se están ejecutando en máquinas virtuales distintas y se pueden comunicar vía una capa de integración subyacente, el ORB y el Servicio de Nombres.

Éste es un ejemplo simple, diseñado para trabajar sin red, pero un ORB suele estar configurado para buscar transparencia de la localización. Cuando el servidor y el cliente están en máquinas distintas, el ORB puede resolver referencias pasadas a Strings, remotas, utilizando un componente denominado el *Repositorio de implementaciones*. Aunque éste es parte CORBA, casi no hay ninguna especificación relativa al mismo, por lo que varía de fabricante en fabricante.

Como puede verse, hay mucho más de CORBA que lo que se ha cubierto hasta aquí, pero uno ya debería haber captado la idea básica. Si se desea más información sobre CORBA un buen punto de partida es el sitio web de OMG, en *http://www.omg.org*. Ahí hay documentación, *white papers*, procedimientos y referencias a otras fuentes y productos relacionados con CORBA.

Applets de Java y CORBA

Los *applets* de Java pueden actuar como clientes CORBA. De esta forma, un *applet* puede acceder a información remota y a servicios expuestos como objetos CORBA. Pero un *applet* sólo se puede conectar con el servidor desde el que fue descargado, por lo que todos los objetos CORBA con los que interactúe el *applet* deben estar en ese servidor. Esto es lo contrario a lo que intenta hacer CORBA: ofrecer una transparencia total respecto a la localización.

Éste es un aspecto de la seguridad de la red. Si uno está en una Intranet, una solución es disminuir las restricciones de seguridad en el navegador. O establecer una política de *firewall* para la conexión con servidores externos.

Algunos productos Java ORB ofrecen soluciones propietarias a este problema. Por ejemplo, algunos implementan lo que se denomina *Tunneling HTTP*, mientras que otros tienen sus propias facetas *firewall*.

Éste es un tema demasiado complejo como para cubrirlo en un apéndice, pero definitivamente es algo de lo que habría que estar pendientes.

CORBA frente a RMI

Se ha visto que una de las facetas más importantes de CORBA es el soporte RPC, que permite a los objetos locales invocar a métodos de objetos remotos. Por supuesto, ya hay una faceta nativa Java que hace exactamente lo mismo: RMI (véase Capítulo 15). Mientras que RMI hace posibles RPC entre objetos Java, CORBA las hace posibles entre objetos implementados en cualquier lenguaje. La diferencia es, pues, enorme.

Sin embargo, RMI puede usarse para invocar a servicios en código remoto no Java. Todo lo que se necesita es algún tipo de envoltorio de objeto Java en torno al código no Java del lado servidor. El objeto envoltorio se conecta externamente a clientes Java vía RMI, e internamente se conecta al código no Java usando una de las técnicas mostradas anteriormente, como JNI o J/Direct.

Este enfoque requiere la escritura de algún tipo de capa de integración, que es exactamente lo que hace CORBA automáticamente, pero de esta forma no se necesitaría un ORB elaborado por un tercero.

Enterprise JavaBeans

Supóngase[6] que hay que desarrollar una aplicación multicapa para visualizar y actualizar registros de una base de datos a través de una interfaz web. Se puede escribir un aplicación de base de datos usando JDBC, una interfaz web usando JSP/servlets, y un sistema distribuido usando CORBA/RMI.

---

[6] Sección a la que ayudó Robert Castaneda, ayudado a su vez por Dave Bartlett.

Pero, ¿qué consideraciones extra hay que hacer al desarrollar un sistema de objetos distribuido además de simplemente conocer las API? He aquí los distintos aspectos:

Rendimiento: los objetos distribuidos que uno crea deben ejecutarse con buen rendimiento, pues potencialmente podrían servir a muchos clientes simultáneamente. Habrá que usar técnicas de optimización como la captura y organización de recursos como conexiones a base de datos. También habrá que gestionar el ciclo de vida de los objetos distribuidos.

Escalabilidad: los objetos distribuidos deben ser también escalables. La escalabilidad en una aplicación distribuida significa que se pueda incrementar el número de instancias de los objetos distribuidos, trasladándose a máquinas adicionales sin necesidad de modificar ningún código.

Seguridad: un objeto distribuido suele tener que gestionar la autorización de los clientes que lo acceden. Idealmente, se pueden añadir nuevos usuarios y roles al mismo sin tener que recompilar.

Transacciones distribuidas: un objeto distribuido debería ser capaz de referenciar a transacciones distribuidas de forma transparente. Por ejemplo, si se está trabajando con dos base de datos separadas, habría que poder actualizarlas simultáneamente dentro de la misma transacción, o deshacer una transacción si no se satisface determinado criterio.

Reusabilidad: el objeto distribuido ideal puede moverse sin esfuerzo a otro servidor de aplicaciones de otro vendedor. Sería genial si se pudiera revender un componente objeto distribuido sin tener que hacerle modificaciones especiales, o comprar un componente de un tercero y usarlo sin tener que recompilarlo ni reescribirlo.

Disponibilidad: si una de las máquinas del sistema se cae, los clientes deberían dirigirse automáticamente a copias de respaldo de esos objetos, residentes en otras máquinas.

Estas consideraciones, además del problema de negocio que se trata de solucionar, pueden hacer un proyecto inabordable. Sin embargo, todos los aspectos *excepto* los del problema de negocio son redundantes —hay que reinventar soluciones para cada aplicación de negocio distribuida.

Sun, junto con otros fabricantes de objetos distribuidos líderes, se dio cuenta de que antes o después todo equipo de desarrollo estaría reinventando estas soluciones particulares, por lo que crearon la especificación de los *Enterprise JavaBeans* (EJB). Esta especificación describe un modelo de componentes del lado servidor que toma en consideración todos los aspectos mencionados utilizando un enfoque estándar que permite a los desarrolladores crear componentes de negocio denominados EJBs, que se aíslan del código "pesado" de bajo nivel y se enfocan sólo en proporcionar lógica de negocio. Dado que los EJB están definidos de forma estándar, pueden ser independientes del fabricante.

JavaBeans frente a EJB

Debido a la similitud de sus nombres, hay mucha confusión en la relación entre el modelo de componentes JavaBeans y la especificación de los *Enterprise JavaBeans*. Mientras que, tanto unos como otros comparten los mismos objetivos de promocionar la reutilización y portabilidad del código Java entre las herramientas de desarrollo y diseminación con el uso de patrones de diseño estándares,

los motivos que subyacen tras cada especificación están orientados a solucionar problemas diferentes.

Los estándares definidos en el modelo de comportamiento de los JavaBeans están diseñados para crear componentes reusables que suelen usarse en herramientas de desarrollo IDE y comúnmente, aunque no exclusivamente, en componentes visuales.

La especificación de los JavaBeans define un modelo de componentes para desarrollar código Java del lado servidor. Dado que los EJBs pueden ejecutarse potencialmente en distintas plataformas de la parte servidor —incluyendo *servidores* que no tienen presentación visuales— un EJB no puede hacer uso de las bibliotecas gráficas, como la AWT o Swing.

La especificación EJB

La especificación de *Enterprise JavaBeans* describe un modelo de componentes del lado servidor. Define seis papeles que se usan para llevar a cabo las tareas de desarrollo y distribución además de definir los componentes del sistema. Estos papeles se usan en el desarrollo, distribución, y ejecución de un sistema distribuido. Los fabricantes, administradores y desarrolladores van jugando los distintos papeles, para permitir la división del conocimiento técnico y del dominio. El fabricante proporciona un marco de trabajo de sonido en su parte técnica, y los desarrolladores crean componentes específicos del dominio; por ejemplo, un componente "contable". Una misma parte puede llevar a cabo uno o varios papeles. Los papeles definidos en la especificación de EJB se resumen en la tabla siguiente:

Papel	Responsabilidad
Suministrador de *Enterprise Bean*	El desarrollador responsable de crear componentes EJB reusables. Estos componentes se empaquetan en un archivo jar especial (archivo ejb-jar).
Ensamblador de aplicaciones	Crea y ensambla aplicaciones a partir de una colección de archivos ejb-jar. Incluye la escritura de aplicaciones que usan la colección de EJB (por ejemplo, servlets, JSP, Swing, etc.,etc.).
Distribuidor	Toma la colección de archivos ejb-jar del ensamblador y/o suministrador de Beans y los distribuye en un entorno de tiempo de ejecución: uno o más contenedores EJB.
Proveedor de contenedores/ servidores de EJB	Proporciona un entorno de tiempo de ejecución y herramientas que se usan para distribuir, administrar y ejecutar componentes EJB.
Administrador del sistema	Gestiona los distintos componentes y servicios, de forma que estén configurados e interactúen correctamente, además de asegurar que el sistema esté activo y en ejecución.

Componentes EJB

Los componentes EJB son elementos de lógica de negocio reutilizable que se adhieren a estándares estrictos y patrones de diseño definidos en la especificación EJB. Esto permite la portabilidad de los componentes. También permite poder llevar a cabo otros servicios —como la seguridad, la gestión de cachés, y las transacciones distribuidas— por parte de los componentes. El responsable del desarrollo de componentes EJB es el *Enterprise Bean Provider.*

Contenedor & Servidor EJB

El *Contenedor EJB* es un entorno de tiempo de ejecución que contiene y ejecuta componentes EJB y les proporciona un conjunto de servicios estándares. Las responsabilidades del contenedor de EJB están definidas de forma clara en la especificación, en aras de lograr neutralidad respecto del fabricante. El contenedor de EJB proporciona el "peso pesado" de bajo nivel del EJB, incluyendo transacciones distribuidas, seguridad, gestión del ciclo de vida de los *beans*, gestión de caché, hilado y gestión de sesiones. El proveedor de contenedor EJB es el responsable de su provisión.

Un *Servidor EJB* se define como un servidor de aplicación que contiene y ejecuta uno o más contenedores de EJB. El Proveedor de Servicios EJB es responsable de proporcionar un Servidor EJB. Generalmente se puede asumir que el Contenedor de EJBs y el Servidor de EJBs son el mismo.

Interfaz Java para Nombres y Directorios (JNDI)[7]

Interfaz usado en los *Enterprise JavaBeans* como el servicio de nombres para los componentes EJB de la red y para otros servicios de contenedores, como las transacciones. JNDI establece una correspondencia nuy estricta con otros estándares de nombres y directorios como *CORBA CosNaming* y puede implementarse, realmente, como un envoltorio realmente de éstos.

Java Transaction API / Java Transaction Service (JTA/JTS)

JTA/JTS se usa en las *Enterprise JavaBeans* como el API transaccional. Un proveedor de *Enterprise Beans* puede usar JTS para crear código de transacción, aunque el contenedor EJB suele implementar transacciones EJB de parte de los componentes EJB. El distribuidor puede definir los atributos transaccionales de un componente EJB en tiempo de distribución. El Contenedor de EJB es el responsable de gestionar la transacción, sea local o distribuida. La especificación JTS es la correspondencia Java al CORBA OTS (*Object Transaction Service*).

CORBA y RMI/IIOP

La especificación EJB define la interoperabilidad con CORBA a través de la compatibilidad con protocolos CORBA. Esto se logra estableciendo una correspondencia entre servicios EJB como JTS y JNDI con los servicios CORBA correspondientes, y la implementación de RMI sobre el protocolo IIOP de CORBA.

---

[7] N. del traductor: *Java Naming and Directory Interface.*

El uso de CORBA y RMI/IIOP en *Enterprise JavaBeans* está implementado en el contenedor de EJB y es el responsable del proveedor de contenedores EJB. El uso de CORBA y de RMI/IIOP sobre el contenedor de EJB está oculto desde el propio componente EJB. Esto significa que el proveedor de *Enterprise Beans* puede escribir su componente EJB y distribuirlo a cualquier contenedor de EJB sin que importe qué protocolo de comunicación se esté utilizando.

Las partes de un componente EJB

Un EJB está formado por varias piezas, incluyendo el propio Bean, la implementación de algunas interfaces, y un archivo de información. Todo se empaqueta junto en un archivo jar especial.

Enterprise Bean

El Enterprise Bean es una clase Java que desarrolla el *Enterprise Bean Provider*. Implementa una interfaz de *Enterprise Bean* y proporciona la implementación de los métodos de negocio que va a llevar a cabo el componente. La clase no implementa ninguna autorización, autentificación, multihilo o código transaccional.

Interfaz local

Todo *Enterprise Bean* que se cree debe tener su interfaz local asociado. Esta interfaz se usa como fábrica del EJB. Los clientes lo usan para encontrar una instancia del EJB o crear una nueva.

Interfaz remota

Se trata de una interfaz Java que refleja los métodos del *Enterprise Bean* que se desea exponer al mundo exterior. Esta interfaz juega un papel similar al de una interfaz CORBA IDL.

Descriptor de distribución

El descriptor de distribución es un archivo XML que contiene información sobre el EJB. El uso de XML permite al distribuidor cambiar fácilmente los atributos del EJB. Los atributos configurables definidos en el descriptor de distribución incluyen:

- Los nombres de interfaz Local y Remota requeridos por el EJB.
- El nombre a publicar en el JNDI para la interfaz local del EJB.
- Atributos transaccionales para cada método de EJB.
- Listas de control de acceso para la autenticación.

Archivo EJB-Jar

Es un archivo jar normal que contiene el EJB, las interfaces local y remota, y el descriptor de distribución.

Funcionamiento de un EJB

Una vez que se tiene un archivo EJB-Jar que contiene el Bean, las interfaces Local y Remota, y el descriptor de distribución, se pueden encajar todas las piezas y a la vez entender por qué se necesitan las interfaces Local y Remota, y cómo los usa el contendor de EJB.

El Contenedor implementa las interfaces Local y Remoto del archivo EJB-Jar. Como se mencionó anteriormente, la interfaz Local proporciona métodos para crear y localizar el EJB. Esto significa que el contenedor de EJB es responsable de la gestión del ciclo de vida del EJB. Este nivel de indirección permite que se den optimizaciones. Por ejemplo, 5 clientes podrían solicitar simultáneamente la creación de un EJB a través de una interfaz Local, y el contenedor de EJB respondería creando sólo un EJB y compartiéndolo entre los 5 clientes. Esto se logra a través de la interfaz Remota, que también está implementado por el Contenedor de EJB. El objeto Remoto implementado juega el papel de objeto *proxy* al EJB.

Todas las llamadas al EJB pasan por el *proxy* a través del contenedor de EJB vía las interfaces Local y Remota. Esta indirección es la razón por la que el contenedor de EJB puede controlar la seguridad y el comportamiento transaccional.

Tipos de EJB

La especificación de *Enterprise JavaBeans* define distintos tipos de EJB con características y comportamientos diferentes. En la especificación se han definido dos categorías de EJB: *Beans* de sesión y *Beans* de entidad, y cada categoría tiene sus propias variantes.

Beans de Sesión

Se usan para representar minúculos casos de uso o flujo de trabajo por parte de un cliente. Representan operaciones sobre información persistente, pero no a la información persistente en sí. Hay dos tipos de Beans de sesión: los que no tienen estado y los que lo tienen. Todos los Beans de sesión deben implementar la interfaz **javax.ejb.SessionBean**. El contenedor de EJB gobierna la vida de un Bean de Sesión.

Beans de Sesión sin estado: son el tipo de componente EJB más simple de implementar. No mantienen ningún estado conversacional con clientes entre invocaciones a métodos por lo que son fácilmente reusables en el lado servidor, y dado que pueden ser almacenados en cachés, se escalan bien bajo demanda. Cuando se usen, hay que almacenar toda información de estado fuera del EJB.

Beans de Sesión con estado: mantienen el estado entre invocaciones. Tienen una correspondencia lógica de uno a uno con el cliente y pueden mantener el estado entre ellas. El responsable de su almacenamiento y paso a caché es el Contenedor de EJB, que lo logra mediante su *Pasivación* y *Activación*. Si el contenedor de EJB falla, se perdería toda la información de los Beans de sesión con estado. Algunos Contenedores de EJB avanzados proporcionan posibilidad de recuperación para estos beans.

Beans de entidad

Son componentes que representan información persistente y comportamiento de estos datos. Estos Beans los pueden compartir múltiples clientes, de la misma forma que puede compartirse la información de una base de datos. El Contenedor de EJB es el responsable de gestionar en la caché los Beans de Entidad, y de mantener su integridad. La vida de estos beans suele ir más allá de la vida del Contenedor de EJB por lo que si éste falla, se espera que el Bean de Entidad siga disponible cuando el contenedor vuelva a estar activo.

Hay dos tipos de Beans de Entidad: los que tienen Persistencia Gestionada por el Contenedor y los de Persistencia Gestionada por el Bean.

Persistencia Gestionada por el Contenedor (CMP o *Container Managed Persistence*). Un Bean de Entidad CMP tiene su persistencia implementada en el Contenedor de EJB. Mediante atributos especificados en el descriptor de distribución, el Contenedor de EJB establecerá correspondencias entre los atributos del Bean de Entidad y algún almacenamiento persistente (generalmente —aunque no siempre— una base de datos). CMP reduce el tiempo de desarrollo del EJB, además de reducir dramáticamente la cantidad de código necesaria.

Persistencia Gestionada por el Bean (BMP o *Bean Managed Persistence*). Un Bean de entidad BMP tiene su persistencia implementada por el proveedor de Enterprise Beans. Éste es responsable de implementar la lógica necesaria para crear un nuevo EJB, actualizar algunos atributos de los EJB, borrar un EJB, y encontrar un EJB a partir de un almacén persistente. Esto suele implicar la escritura de código JDBC para interactuar con una base de datos u otro almacenamiento persistente. Con BMP, el desarrollado tiene control total sobre la gestión de la persistencia del Bean de Entidad.

BMP también proporciona flexibilidad allí donde puede no es posible implementar un CMP. Por ejemplo, si se deseara crear un EJB que envolviera algún código existente en un servidor, se podría escribir la persistencia usando CORBA.

Desarrollar un EJB

Como ejemplo, se implementará como componente EJB el ejemplo "Tiempo Perfecto" de la sección RMI anterior. El ejemplo será un sencillo Bean de Sesión sin Estado.

Como se mencionó anteriormente, los componentes EJB consisten en al menos una clase (el EJB) y dos interfaces: el Remoto y el Local. Cuando se crea una interfaz remota para un EJB, hay que seguir las siguientes directrices:

1. La interfaz Remota debe ser **public**.
2. La interfaz Remota debe extender a la interfaz **javax.ejb.EJBObject.**
3. Cada método de la interfaz Remota debe declarar **java.rmi.RemoteException** en su cláusula **throws** además de cualquier excepción específica de la aplicación.

4. Cualquier objeto que se pase como parámetro o valor de retorno (bien directamente o embebido en algún objeto local) debe ser un tipo de datos RMI-IIOP válido (incluyendo otros objetos EJB).

He aquí una interfaz remota simple para el EJB TiempoPerfecto:

```
//: c15:ejb:TiempoPerfecto.java
//# Para compilar este archivo hay que instalar
//# la J2EE Java Enterprise Edition de
//# java.sun.com y añadir j2ee.jar al
//# CLASSPATH. Ver detalles en java.sun.com.
// Interfaz Remota de BeanTiempoPerfecto
import java.rmi.*;
import javax.ejb.*;

public interface TiempoPerfecto extends EJBObject {
  public long obtenerTiempoPerfecto()
    throws RemoteException;
} ///:~
```

La interfaz Local es la fábrica en la que se creará el componente. Puede definir métodos *create*, para crear instancias de EJB, o métodos *finder* que localizan EJB existentes y se usan sólo para Beans de entidad. Cuando se crea una interfaz Local para un EJB hay que seguir estas directrices:

1. La interfaz Local debe ser **public**.
2. La interfaz Local debe extender el interfaz **javax.ejb.EJBHome**.
3. Cada método *create* de la interfaz Local debe declarar **java.rmi.RemoteException** en su cláusula **throws** además de una **javax.ejb.CreateException**.
4. El valor de retorno de un método *create* debe ser una interfaz Remota.
5. El valor de retorno de un método *finder* (sólo para Beans de Entidad) debe ser una interfaz Remota o **java.util.Enumeration** o **java.util.Collection**.
6. Cualquier objeto que se pase como parámetro (bien directamente o embebido en un objeto local) debe ser un tipo de datos RMI-IIOP válido (esto incluye otros objetos EJB).

La convención estándar de nombres para las interfaces Locales es tomar el nombre de la interfaz Remoto y añadir "Home" al final. He aquí la interfaz Local para el EJB TiempoPerfecto:

```
//: c15:ejb:TiempoPerfectoHome.java
// Interfaz Local de BeanTiempoPerfecto.
import java.rmi.*;
import javax.ejb.*;

public interface TiempoPerfectoHome extends EJBHome {
```

```
  public TiempoPerfecto create()
    throws CreateException, RemoteException;
} ///:~
```

Ahora se puede implementar la lógica de negocio. Cuando se cree la clase de implementación del EJB, hay que seguir estas directrices, (nótese que debería consultarse la especificación de EJB para lograr una lista completa de las directrices de desarrollo de *Enterprise JavaBeans*):

1. La clase debe ser **public**.
2. La clase debe implementar una interfaz EJB (o **javax.ejb.SessionBean** o **javax.ejb.EntityBean**).
3. La clase debería definir métodos que se correspondan directamente con los métodos de la interfaz Remota. Nótese que la clase no implementa la interfaz Remota; hace de espejo de los métodos de la interfaz Remota pero *no* lanza **java.rmi.RemoteException**.
4. Definir uno o más métodos **ejbCreate( )** para inicializar el EJB.
5. El valor de retorno y los parámetros de todos los métodos deben ser tipos de datos RMI-IIOP válidos.

```
//: c15:ejb:BeanTiempoPerfecto.java
// Bean Simple de Sesión sin estado
// que devuelve la hora actual del sistema.
import java.rmi.*;
import javax.ejb.*;

public class BeanTiempoPerfecto
  implements SessionBean {
  private SessionContext contextoSesion;
  //devuelve el tiempo actual
  public long obtenerTiempoPerfecto() {
      return System.currentTimeMillis();
  }
  //  métodos EJB
  public void ejbCreate()
  throws CreateException {}
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void
  SetSessioncontext(SessionContext ctx) {
    TcontextoSesion = ctx;
  }
}///:~
```

Dado que este ejemplo es muy sencillo, los métodos EJB (**ejbCreate( )**, **ejbRemove( )**, **ejbActivate( )**, **ejbPassivate( )**) están todos vacíos. Estos métodos son invocados por el Contenedor de EJB y se usan para controlar el estado del componente. El método **setSesionContext( )** pasa un objeto **javax.ejb.SessionContext** que contiene información sobre el contexto del componente, como información de la transacción actual y de seguridad.

Tras crear el *Enterprise JavaBean*, tenemos que crear un descriptor de distribución. Éste es un archivo XML que describe el componente EJB, y que debería estar almacenado en un archivo denominado **ebj-jar.xml**.

```
//:! c15:ejb:ejb-jar.xml
<?xml version="1.0" encoding="Cp1252"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>
  <description>Ejemplo del Capitulo 15</description>
  <display-name></display-name>
  <small-icon></small-icon>
  <large-icon></large-icon>
  <enterprise-beans>
    <session>
      <ejb-name>TiempoPerfecto</ejb-name>
      <home>TiempoPerfectoHome</home>
      <remote>TiempoPerfecto</remote>
      <ejb-class>BeanTiempoPerfecto</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <ejb-client-jar></ejb-client-jar>
</ejb-jar>
///:~
```

Pueden verse el componente, la interfaz Remota y la interfaz Local definidos dentro de la etiqueta <**session**> de este descriptor de distribución. Estos descriptores pueden ser generados automáticamente usando las herramientas de desarrollo de EJB.

Junto con el descriptor de distribución estándar **ejb-jar.xml**, la especificación EJB establece que cualquier etiqueta específica de un vendedor debería almacenarse en un archivo separado. Esto pretende lograr una alta portabilidad entre componentes y distintos tipos de contenedores de EJB.

Los archivos deben guardarse dentro de un Archivo Java estándar (JAR). Los descriptores de distribución deberían ubicarse dentro del subdirectorio **/META-INF** del archivo Jar.

Una vez que se ha definido el componente EJB dentro del descriptor de distribución, el distribuidor debería distribuir el componente EJB en el Contendor de EJB. Al escribir esto, el proceso de dis-

tribución era bastante "intensivo respecto a IGU" y específico de cada contendor de EJB individual, por lo que este repaso no documenta ese proceso. Todo contenedor de EJB, sin embargo, tendrá un proceso documentado para la distribución de un EJB.

Dado que un componente EJB es un objeto distribuido, el proceso de distribución también debería crear algunos *stubs* de cliente para invocar al componente. Estas clases deberían ubicarse en el *classpath* de la aplicación cliente. Dado que los componentes EJB pueden implementarse sobre RMI-IIOP (CORBA) o sobre RMI-JRMP, los *stubs* generados podrían variar entre contenedores de EJB; de cualquier manera, son clases generadas.

Cuando un programa cliente desea invocar a un EJB, debe buscar el componente EJB dentro de la JNDI y obtener una referencia al interfaz local del componente EJB. Esta interfaz se usa para crear una instancia del EJB.

En este ejemplo, el programa cliente es un programa Java simple, pero debería recordarse que podría ser simplemente un servlet, un JSP o incluso un objeto distribuido CORBA o RMI.

```
//: c15:ejb:ClienteTiempoPerfecto.java
// Programa cliente para BeanTiempoPerfecto

public class ClienteTiempoPerfecto {
public static void main(String[] args)
throws Exception {
   // Lograr un contexto JNDI usando
   // el servicio de nombre de JNDI:
   javax.naming.Context contexto =
      new javax.naming.InitialContext();
   // Buscar la interfaz local en
   // el JNDI Naming service:
   Object ref = contexto.lookup("tiempoPerfecto");
   // Convertir el objeto remoto la interfaz local:
   TiempoPerfectoHome home = (TiempoPerfectoHome)
      javax.rmi.PortableRemoteObject.narrow(
         ref, TiempoPerfectoHome.class);
   // Crear un objeto remoto para la interfaz local:
   TiempoPerfecto tp = home.create();
   // Invocar obtenerTiempoPerfecto()
   System.out.println(
      "EJB Tiempo Perfecto invocado, son las: " +
      tp.obtenerTiempoPerfecto() );
   }
} ///:~
```

La secuencia de este ejemplo se explica mediante comentarios. Nótese el uso del método **narrow( )** para llevar a cabo una conversión del objeto antes de llevar a cabo una conversión Java autentica.

Esto es muy semejante a lo que ocurre en CORBA. Nótese también que el objeto Home se convierte en una fábrica de objetos **TiempoPerfecto**.

Resumen de EJB

La especificación de *Enterprise JavaBeans* es un terrible paso adelante de cara a la estandarización y simplificación de la computación de objetos distribuidos. Es una de las piezas más importantes de la plataforma *Java 2 Enterprise Edition* (J2EE) y está recibiendo mucho soporte de la comunidad de objetos distribuidos. Actualmente hay muchas herramientas disponibles, y si no lo estarán en breve, para ayudar a acelerar el desarrollo de componentes EJB.

Este repaso sólo pretendía ser un breve *tour* por los EJB. Para más información sobre la especificación de EJB puede acudirse a la página oficial de los *Enterprise JavaBeans* en *http://java.sun.com/products/ejb* donde puede descargarse la última especificación y la implementación de referencia J2EE. Éstas pueden usarse para desarrollar y distribuir componentes EJB propios.

Jini: servicios distribuidos

Esta sección[8] da un repaso a la tecnología Jini de Sun Microsystems. Describe algunas de las ventajas e inconvenientes de Jini y muestra cómo su arquitectura ayuda a afrontar el nivel de abstracción en programación de sistemas distribuidos, convirtiendo de forma efectiva la programación en red en programación orientada a objetos.

Jini en contexto

Tradicionalmente, se han diseñado los sistemas operativos con la presunción de que un computador tendría un procesador, algo de memoria, y un disco. Cuando se arranca un computador lo primero que hace es buscar un disco. Si no lo encuentra, no funciona como computador. Cada vez más, sin embargo, están apareciendo computadores de otra guisa: como dispositivos embebidos que tienen un procesador, algo de memoria, y una conexión a la red —pero no disco. Lo primero que hace un teléfono móvil al arrancarlo, por ejemplo, es buscar una red de telefonía. Si no la encuentra, no puede funcionar como teléfono móvil. Esta tendencia en el entorno hardware, de centrados en el disco a centrados en la red, afectará a la forma que tenemos de organizar el software —y es aquí donde Jini cobra sentido.

Jini es un intento de replantear la arquitectura de los computadores, dada la importancia creciente de la red y la proliferación de procesadores en dispositivos sin unidad de disco. Estos dispositivos, que vendrán de muchos fabricantes distintos, necesitarán interactuar por una red. La red en sí será muy dinámica —regularmente se añadirán y eliminarán dispositivos y servicios. Jini proporciona mecanismos para permitir la adición, eliminación y búsqueda de directorios y servicios de forma sencilla a través de la red. Además, Jini proporciona un modelo de programación que facilita a los programadores hacer que sus dispositivos se comuniquen a través de la red.

---

[8] Esta sección se llevó a cabo con la ayuda de Bill Venners (http://www.artima.com).

Construido sobre Java, la serialización de objetos, y RMI (que permiten, entre todos, que los objetos se muevan por la red de máquina virtual en máquina virtual), Jini intenta extender los beneficios de la programación orientada a objetos a la red. En vez de pedir a fabricantes de dispositivos que se pongan de acuerdo en protocolos de red para que sus dispositivos interactúen, Jini habilita a los dispositivos para que se comuniquen mutuamente a través de interfaces con objetos.

¿Qué es Jini?

Jini es un conjunto de API y protocolos de red que pueden ayudar a construir y desplegar sistemas distribuidos organizados como *federaciones de servicios*. Un *servicio* puede ser cualquier cosa que resida en la red y que esté listo para llevar a cabo una función útil. Los dispositivos hardware, el software, los canales de comunicación —e incluso los propios seres humanos— podrían ser servicios. Una unidad de disco habilitada para Jini, por ejemplo, podría ofrecer un servicio de "almacenamiento". Una impresora habilitada para Jini podría ofrecer un servicio de "impresión". Por consiguiente, una federación de servicios es un conjunto de servicios disponible actualmente en la red, que un cliente (entendiendo por tal un programa, servicio o usuario) puede juntar para que le ayuden a lograr alguna meta.

Para llevar a cabo una tarea, un cliente lista la ayuda de varios servicios. Por ejemplo, un programa cliente podría mandar a un servidor fotos del almacén de imágenes de una cámara digital, descargar las fotos a un servicio de almacenamiento persistente ofrecido por una unidad de disco, y enviar una página de versiones del tamaño de una de las fotos al servicio de impresión de una impresora a color. En este ejemplo, el programa cliente construye un sistema distribuido que consiste en sí mismo, el servicio de almacenamiento de imágenes, el servicio de almacenamiento persistente, y el servicio de impresión en color. El cliente y los servicios de este sistema distribuido trabajan juntos para desempeñar una tarea: descargar y almacenar imágenes de una cámara digital e imprimir una página de copias diminutas.

La idea que hay tras la palabra *federación* es que la visión de Jini de la red no implique una autoridad de control central. Dado que no hay ningún servicio al mando, el conjunto de todos los servicios disponibles en la red conforman una federación —un grupo formado por iguales. En vez de una autoridad central, la infraestructura de tiempo de ejecución de Jini simplemente proporciona una forma para que los clientes y servicios se encuentren entre sí (vía un servicio de búsqueda que almacena un directorio de los servicios actualmente disponibles). Una vez que los servicios se localizan entre sí ya pueden funcionar. El cliente y sus servicios agrupados llevan a cabo su tarea independientemente de la infraestructura de tiempo de ejecución de Jini. Si el servicio de búsqueda de Jini se cae, cualquier sistema distribuido conformado vía el servicio de búsqueda antes de que se produjera el fallo puede continuar con su trabajo. Jini incluso incluye un protocolo de red que pueden usar los clientes para encontrar servicios en la ausencia de un servicio de búsqueda.

Cómo funciona Jini

Jini define una *infraestructura de tiempo de ejecución* que reside en la red y proporciona mecanismos que permiten a cada uno añadir, eliminar, localizar y acceder a servicios. La infraestructura de tiempo de ejecución reside en tres lugares: en servicios de búsqueda que residen en la red, en los proveedores de servicios (como los dispositivos habilitados para Jini), y en los clientes. Los *Servicios de bús-*

queda son el mecanismo de organización central de los sistemas basados en Jini. Cuando aparecen más servicios en la red, se registran a sí mismos con un servicio de búsqueda. Cuando los clientes desean localizar un servicio para ayudar con alguna tarea, consultan a un servicio de búsqueda.

La infraestructura de tiempo de ejecución usa un protocolo de nivel de red denominado *discovery*, y dos protocolos de nivel de red, llamados *join* y *lookup*. *Discovery* permite a los clientes y servicios localizar los servicios de búsqueda. *Join* permite a un servicio registrarse a sí mismo en un servicio de búsqueda. *Lookup* permite a un cliente preguntar por los servicios que pueden ayudar a lograr sus metas.

El proceso de discovery

Discovery funciona así: imagínese que se tiene una unidad de disco *habilitada para Jini* que ofrece un servicio de almacenamiento persistente. Tan pronto como se conecte el disco a la red, éste lanza en multidifusión un *anuncio de presencia* depositando un paquete de multidifusión en un puerto bien conocido. En este anuncio de presencia va incluida la dirección IP y el número de puerto en el que el servicio de búsqueda puede localizar la unidad de disco.

Los servicios de búsqueda monitorizan el puerto bien conocido en espera de paquetes de anuncio de presencia. Cuando un servicio de búsqueda recibe un anuncio de presencia, lo abre e inspecciona el paquete. Éste contiene información que permite al servicio de búsqueda determinar si debería o no contactar al emisor del paquete. En caso afirmativo, contacta con el emisor directamente haciendo una conexión TCP a la dirección IP y número de puerto extraídos del paquete. Usando RMI, el servicio de búsqueda envía un objeto denominado *registrador de servicios*, a través de la red, a la fuente del paquete. El propósito del servicio registrador de objetos es facilitar una comunicación más avanzada con el servicio de búsqueda. Al invocar a los métodos de este objeto, el emisor del paquete de anuncio puede unirse y buscar en el servicio de búsqueda. En el caso de la unidad de disco, el servicio de búsqueda haría una conexión TCP a la unidad de disco y le enviaría un objeto registrador de servicios, a través del cual registraría después la unidad de disco su servicio de almacenamiento persistente vía el proceso *join*.

El proceso join

Una vez que un proveedor de servicio tiene un objeto registrador de servicios, como resultado del *discovery*, está listo para hacer una *join* —convertirse en parte de la federación de servicios registrados en el servicio de búsqueda. Para ello, el proveedor del servicio invoca al método **register( )** del objeto registrador de servicios, pasándole como parámetro un objeto denominado un elemento de servicio, un conjunto de objetos que describen el servicio. El método **register( )** envía una copia del elemento de servicio al servicio de búsqueda, donde se almacena el elemento de servicio.

Una vez completada esta operación, el proveedor del servicio ha acabado el proceso *join*: su servicio ya está registrado en el servicio de búsqueda.

Este elemento de servicio es un contenedor de varios objetos, incluyendo uno llamado un *objeto de servicio*, que pueden usar los clientes para interactuar con el servicio. El elemento de servicio tam-

bién puede incluir cualquier número de *atributos*, que pueden ser cualquier objeto. Algunos atributos potenciales son iconos, clases que proporcionan IGU para el servicio, y objetos que proporcionan más información del servicio.

Los objetos de servicio suelen implementar uno o más interfaces a través de los cuales los clientes interactúan con el servicio. Por ejemplo, un servicio de búsqueda es un servicio Jini, y su objeto de servicio está en el registrador de servicios. El método **register( )** invocado por los proveedores de servicio durante el join se declara en la interfaz **ServiceRegistrar** (miembro del paquete **net.jini.core.lookup**), que implementan todos los objetos registradores de servicios. Los clientes y proveedores de servicios se comunican con el servicio de búsqueda a través del objeto registrador de servicios invocando a los métodos declarados en la interfaz **ServiceRegistrar**. De manera análoga, una unidad de disco proporcionaría un objeto servicio que implementaba alguna interfaz de servicio de almacenamiento bien conocido. Los clientes buscarían e interactuarían con la unidad de disco a través de esta interfaz de servicio de almacenamiento.

El proceso lookup

Una vez que se ha registrado un servicio con un servicio lookup vía el proceso join, ese servicio está disponible para ser usado por clientes que pregunten al servicio de búsqueda. Para construir un sistema distribuido que trabaje en conjunción para desempeñar alguna tarea un cliente debe localizar y agrupar la ayuda de servicios individuales. Para encontrar un servicio, los clientes preguntan a servicios de búsqueda vía un proceso llamado *lookup*.

Para llevar a cabo una búsqueda, un cliente invoca al método **lookup( )** de un objeto registrador de servicios. (Un cliente, como un proveedor de servicios, logra un registrador de servicios a través del proceso de *discovery* anteriormente descrito). El cliente pasa una plantilla de servicio como parámetro al **lookup( )**. Esta plantilla es un objeto que sirve como criterio de búsqueda para la consulta. La plantilla puede incluir una referencia a un array de objetos **Class**. Estos objetos **Class** indican al servicio de búsqueda el tipo (o tipos) Java del objeto servicio deseados por el cliente. La plantilla también puede incluir un *ID de servicio*, que identifica un servicio de manera unívoca, y atributos, que deben casar exactamente con los atributos proporcionados por el proveedor de servicios en el elemento de servicio. La plantilla de servicio también puede contener comodines para alguno de estos campos. Un comodín en el campo ID del servicio, por ejemplo, casará con cualquier ID de servicio. El método **lookup( )** envía la plantilla al servicio *lookup*, que lleva a cabo la consulta y devuelve cero a cualquier objeto de servicio que case. El cliente logra una referencia a los objetos de servicio que casen como valor de retorno del método **lookup( )**.

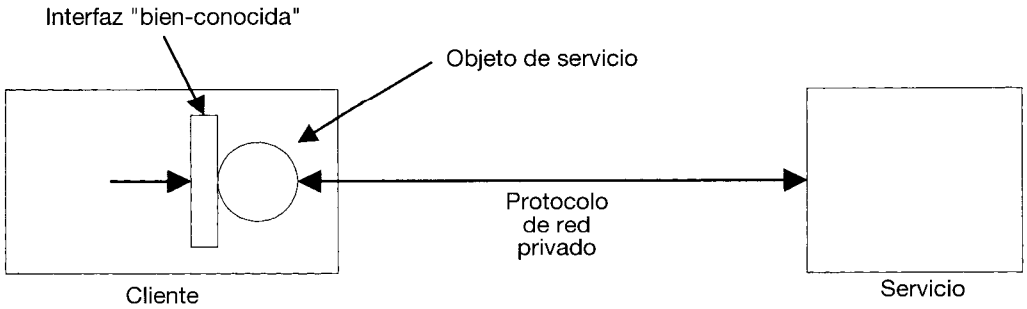
En el caso general, un cliente busca un servicio por medio de un tipo Java, generalmente una interfaz. Por ejemplo, si un cliente quisiese usar una impresora, compondría una plantilla de servicio que incluiría un objeto **Class** para una interfaz bien conocida. El servicio de búsqueda devolvería un objeto (o varios) de servicio que implementa esta interfaz. En la plantilla de servicio pueden incluirse atributos para reducir el número de coincidencias para una búsqueda de este tipo. El cliente usaría el servicio de impresión invocando a los métodos de la interfaz de impresión bien conocida del objeto de servicio.

Separación de interfaz e implementación

La arquitectura de Jini acerca la programación orientada a la red permitiendo a los servicios de red tomar ventaja de uno de los aspectos fundamentales de los objetos: la separación de interfaz e implementación. Por ejemplo, un objeto de servicio puede garantizar a los clientes acceso al servicio de muchas formas. El objeto puede representar, de hecho, todo el servicio, que es descargado al cliente durante la búsqueda, y después ejecutado localmente. Después, cuando el cliente invoca a métodos del objeto servicio, envía las peticiones al servidor a través de la red, que hace el trabajo real. Una tercera opción es que el objeto servicio local y un servidor remoto hagan parte del trabajo cada uno.

Una consecuencia importante de la arquitectura de Jini es que el protocolo de red usado para comunicarse entre un objeto de servicio intermediario y un servidor remoto no tiene por qué ser conocido por el cliente. Como se muestra en la siguiente figura, el protocolo de red es parte de la implementación del servicio. El cliente puede comunicarse con el servicio vía el protocolo privado porque el servicio inyecta parte de su propio código (el objeto servicio) en el espacio de reacciones del cliente. El objeto de servicio inyectado podría comunicarse con el servicio vía RMI, CORBA, DCOM, algún protocolo casero construido sobre sockets y flujos, o cualquier otra cosa. El cliente simplemente no tiene que preocuparse de protocolos de red, puede comunicarse con la interfaz bien conocida que implementa el objeto de servicio. El objeto de servicio se encarga de cualquier comunicación necesaria por la red.

Distintas implementaciones de una misma interfaz de servicio pueden usar enfoques y protocolos de red completamente diferentes. Un servicio puede usar hardware especializado para completar las solicitudes del cliente, o puede hacer todo su trabajo vía software. De hecho, el enfoque de implementación tomado por un único servicio puede evolucionar con el tiempo. El cliente puede estar seguro de tener un objeto de servicio que entiende la implementación actual del servicio, porque el cliente recibe el objeto de servicio (por parte del servicio de búsqueda) del propio proveedor de servicios. Para el cliente, un servicio tiene el aspecto de una interfaz bien conocida, independientemente de cómo esté implementado el servicio.



El cliente se comunica con el servicio a través de una interfaz bien conocida.

Abstraer sistemas distribuidos

Jini intenta elevar el nivel de abstracción para programación de sistemas distribuidos, desde el nivel del protocolo de red al nivel de interfaz del objeto. En la proliferación emergente de dispositivos embebidos conectados a redes puede haber muchas piezas de un sistema distribuido que provengan de distintos fabricantes. Jini hace innecesario que los fabricantes de dispositivos se pongan de acuerdo en los protocolos de nivel de red que permitan a sus dispositivos interactuar. En vez de ello, los fabricantes deben estar de acuerdo en las interfaces Java a través de las cuales pueden interactuar sus dispositivos. Los procesos de *discovery*, *join*, y *lookup* proporcionados por la infraestructura de tiempo de ejecución de Jini permiten a los dispositivos localizarse entre sí a través de la red. Una vez que se localizan, los dispositivos pueden comunicarse mutuamente a través de interfaces Java.

Resumen

Junto con Jini para redes de dispositivos locales, este capítulo ha presentado algunos, pero no todos los componentes a los que Sun denomina J2EE: la *Java 2 Enterprise Edition*. La meta de J2EE es construir un conjunto de herramientas que permitan a un desarrollador Java construir aplicaciones basadas en servidores mucho más rápido, y de forma independiente de la plataforma. Construir aplicaciones así no sólo es difícil y consume tiempo, sino que es especialmente difícil construirlas de forma que puedan ser portadas fácilmente a otras plataformas, y además mantener la lógica de negocio separada de los detalles de implementación subyacentes. J2EE proporciona un marco de trabajo para ayudar a crear aplicaciones basadas en servidores; estas aplicaciones tienen gran demanda hoy en día, y esa demanda parece, además, creciente.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com*.

1. Compilar y ejecutar los programas **ServidorParlante** y **ClienteParlante** de este capítulo. Ahora, editar los archivos para eliminar todo *espacio de almacenamiento intermedio* en las entradas y salidas, después compilar y volver a ejecutarlos para observar los resultados.

2. Crear un servidor que pida una contraseña, después un archivo y enviarlo a través de la conexión de red. Crear un cliente que se conecte al servidor, proporcionar la contraseña adecuada, y después capturar y salvar el archivo. Probar los dos programas en la misma máquina usando el **localhost** (la dirección IP de bucle local **127.0.0.1** producida al invocar a **InetAddress.getByName(null)**).

3. Modificar el servidor del Ejercicio 2, de forma que use el multihilo para manejar múltiples clientes.

4. Modificar **ClienteParlante.java** de forma que no se dé el vaciado de la salida y se observe su efecto.

5. Modificar **ServidorMultiParlante** de forma que use *hilos cooperativos*. En vez de lanzar un hilo cada vez que un cliente se desconecte, el hilo debería pasar a un "espacio de hilos disponibles". Cuando se desee conectar un nuevo cliente, el servidor buscará en este espacio un hilo que gestione la petición, y si no hay ninguno disponible, construirá uno nuevo. De esta forma, el número de hilos necesario irá disminuyendo en cuanto a la cantidad necesaria. La aportación de hilos cooperativos es que no precisa de sobrecarga para la creación y destrucción de hilos nuevos por cada cliente.

6. A partir de **MostrarHTML.java**, crear un *applet* que sea una pasarela protegida por contraseña a una porción particular de un sitio web.

7. Modificar **CrearTablasCID.java,** de forma que lea las cadenas de caracteres SQL de un texto en vez de **CIDSQL**.

8. Configurar el sistema para que se pueda ejecutar con éxito **CrearTablasCID.java** y **CargarBD.java**.

9. Modificar **ReglasServlets.java** superponiendo el método **destroy( )** para que salve el valor de **i** a un archivo, y el método **init( )** para que restaure el valor. Demostrar que funciona volviendo a arrancar el contenedor de servlets. Si no se tiene un contenedor de servlets, habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache.org* para poder ejecutar servlets.

10. Crear un servlet que añada un *cookie* al objeto respuesta, almacenándolo en el lado cliente. Añadir al servlet el código que recupera y muestra el *cookie*. Si no se tiene un contenedor de servlets habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache,org* para poder ejecutar servlets.

11. Crear un servlet que use un objeto **Session** para almacenar la información de sesión que se seleccione. En el mismo servlet, retirar y mostrar la información de sesión. Si no se tiene un contenedor de servlets habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache.org* para poder ejecutar servlets.

12. Crear un servlet que cambie el intervalo inactivo de una sesión a 5 segundos invocando a **getMaxInactiveInterval( )**. Probar que la función llegue a expirar tras 5 segundos. Si no se tiene un contenedor de servlets habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache.org* para poder ejecutar servlets.

13. Crear una página JSP que imprima una línea de texto usando la etiqueta <H1>. Poner el color de este texto al azar, utilizando código Java embebido en la página JSP. Si no se tiene un contenedor de JSP habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache.org* para poder ejecutar servlets.

14. Modificar el valor de la edad máxima de **Cookies.jsp** y observar el comportamiento bajo dos navegadores diferentes. Notar también la diferencia entre revisitar la página y apagar y volver a arrancar el navegador. Si no se tiene un contenedor de JSP habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache.org* para poder ejecutar servlets.

15. Crear un JSP con un campo que permita al usuario introducir el tiempo de expiración de la sesión y un segundo campo que guarde la fecha almacenada en la sesión. El botón de envío refresca la página y captura la hora de expiración actual además de la información de la sesión y las coloca como valores por defecto en los campos antes mencionados. Si no se tiene un contenedor de JSP habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache.org* para poder ejecutar servlets.

16. (Más complicado) tomar el programa **BuscarV.java** y modificarlo, de forma que al hacer clic en el nombre resultante tome automáticamente el nombre y lo copie al portapapeles (de forma que se pueda simplemente pegar en el correo electrónico). Habrá que echar un vistazo al Capítulo 13 para recordar como usar el portapapeles en JFC.