Objetos distribuidos

Hasta ahora este libro se ha centrado en el uso del paradigma de paso de mensajes en la computación distribuida. A través del paradigma de paso de mensajes, los procesos intercambian datos y, mediante el uso de determinados protocolos, colaboran en la realización de tareas. Las interfaces de programación de aplicaciones basadas en este paradigma, tales como el API de sockets de unidifusión y multidifusión de Java, proporcionan una abstracción que permite esconder los detalles de la comunicación de red a bajo nivel, así como escribir código de comunicación entre procesos (IPC), utilizando una sintaxis relativamente sencilla. Este capítulo introduce un paradigma que ofrece aún una mayor abstracción, los objetos distribuidos.

Paso de mensajes frente a objetos distribuidos

El paradigma de paso de mensajes es un modelo natural para la computación distribuida, en el sentido de que imita la comunicación entre humanos. Se trata de un paradigma apropiado para los servicios de red, puesto que estos procesos interactúan a través del intercambio de mensajes. Pero este paradigma no proporciona la abstracción necesaria para algunas aplicaciones de red complejas, por los siguientes motivos:

- El paso de mensaje básico requiere que los procesos participantes estén **fuertemente acoplados**. A través de esta interacción, los procesos deben comunicarse directamente entre ellos. Si la comunicación se pierde entre los procesos (debido a fallos en el enlace de comunicación, en el sistema o en uno de los procesos), la colaboración falla. Por ejemplo, considérese una sesión del protocolo *Echo*: si la comunicación entre el cliente y el servidor es interrumpida, la sesión no puede continuar.
- El paradigma de paso de mensajes está **orientado a datos**. Cada mensaje contiene datos con un formato mutuamente acordado, y se interpreta como una petición o respuesta de acuerdo al protocolo. La recepción de cada mensaje desencadena una acción en el proceso receptor.
  Por ejemplo, en el protocolo *Echo*, el receptor de un mensaje del proceso *p* solicita esta acción al servidor *Echo*: un mensaje conteniendo los mismos datos se envía al proceso *p*. En el mismo protocolo, el receptor de un mensaje del servidor *Echo* por el proceso *p* desencadena esta acción: un nuevo mensaje es solicitado por el usuario y el mensaje se envía al servidor *Echo*.
  Mientras que el hecho de que el paradigma sea orientado a datos es apropiado para los servicios de red y aplicaciones de red sencillas, no es adecuado para aplicaciones complejas que impliquen un gran número de peticiones y respuestas entremezcladas. En dichas aplicaciones, la interpretación de los mensajes se puede convertir en una tarea inabordable.

El **paradigma de objetos distribuidos** es un paradigma que proporciona mayor abstracción que el modelo de paso de mensajes. Como su nombre indica, este paradigma está basado en objetos existentes en un sistema distribuido. En programación orientada a objetos, basada en un lenguaje de programación

orientado a objetos, tal como Java, los objetos se utilizan para representar entidades significativas para la aplicación. Cada objeto encapsula

- el **estado** o datos de la entidad – en Java, dichos datos se encuentran en las **variables de instancia** de cada objeto;
- las **operaciones** de la entidad, a través de las cuales se puede acceder o modificar el estado de la entidad – en Java, estas operaciones se denominan **métodos**.

Para ilustrar estos conceptos, considérese los objetos de la clase *MensajeDatagrama* presentada en la Figura 5.12 (en el Capítulo 5). Cada objeto instanciado de esta clase contiene tres variables de estado: un mensaje, la dirección del emisor y el número de puerto del emisor. Además, cada objeto contiene tres operaciones: (1) un método *fijaValor*, que permite modificar los valores de las variables de estado, (2) un método *obtieneMensaje*, que permite obtener el valor actual del mensaje, y (3) un método *obtieneDireccion*, que permite obtener la dirección del emisor.

Aunque en este libro se han utilizado objetos en capítulos anteriores, tales como el objeto *MensajeDatagrama*, se trata de objetos **locales**, en lugar de objetos **distribuidos**. Los objetos locales son objetos cuyos métodos sólo se pueden invocar por un **proceso local**, es decir, un proceso que se ejecuta en el mismo computador del objeto. Un objeto distribuido es aquel cuyos métodos pueden invocarse por un **proceso remoto**, es decir, un proceso que se ejecuta en un computador conectado a través de una red al computador en el cual se encuentra el objeto. En un paradigma de objetos distribuidos, los recursos de la red se representan como objetos distribuidos. Para solicitar un servicio de un recurso de red, un proceso invoca uno de sus métodos u operaciones, pasándole los datos como parámetros al método. El método se ejecuta en la máquina remota, y la respuesta es enviada al proceso solicitante como un valor de salida. Comparado con el paradigma de paso de mensajes, el paradigma de objetos distribuidos es **orientado a acciones**: Hace hincapié en la invocación de las operaciones, mientras que los datos toman un papel secundario (como parámetros y valores de retorno). Aunque es menos intuitivo para los seres humanos, el paradigma de objetos distribuidos es más natural para el desarrollo de software orientado a objetos.

La Figura 7.1 ilustra el paradigma. Un proceso que se ejecuta en la máquina A realiza una llamada a un método de un objeto distribuido de la máquina B, pasando los datos necesarios, en caso de existir, mediante argumentos. La llamada al método invoca una acción realizada por el método en la máquina A, y un valor de salida, en caso de que exista, se pasa desde la máquina A a la máquina B. Un proceso que utiliza objetos distribuidos se dice que es un **proceso cliente** de ese objeto, y los métodos del objeto se denominan **métodos remotos** (por contraposición a los métodos locales, o métodos pertenecientes a un objeto local) del proceso cliente.

En el resto del capítulo se va a proceder a presentar una arquitectura genérica que da soporte al paradigma de objetos distribuidos; a continuación se explorará un ejemplo de este tipo de arquitecturas: **Java RMI (*Remote Method Invocation*)**.

Una arquitectura típica de objetos distribuidos

La premisa de todo sistema de objetos distribuidos es minimizar las diferencias de programación entre las invocaciones de métodos remotos y las llamadas a métodos locales, de forma que los métodos remotos se puedan invocar en una aplicación utilizando una sintaxis similar a la utilizada en la invocación de los métodos locales. Realmente existen diferencias, porque la invocación de métodos remotos implica una comunicación entre procesos independientes, y por tanto deben tratarse aspectos tales como el empaquetamiento de los datos (*marshaling*), así como la sincronización de los eventos. Estas diferencias quedan ocultas en la arquitectura.

La Figura 7.2 presenta una arquitectura típica de una utilidad que dé soporte al paradigma de objetos distribuidos.

Al objeto distribuido proporcionado o **exportado** por un proceso se le denomina **servidor de objeto**. Otra utilidad, denominada **registro de objetos**, o simplemente **registro**, debe existir en la arquitectura para registrar los objetos distribuidos.

Para acceder a un objeto distribuido, un proceso, el **cliente de objeto**, busca en el registro para encontrar una **referencia** al objeto. El cliente de objeto utiliza esta referencia para realizar llamadas a los métodos del objeto remoto, o **métodos remotos**. Lógicamente, el cliente de objeto realiza una llamada directamente al método remoto. Realmente, un componente software se encarga de gestionar esta llamada. Este componente se denomina **cliente *proxy*** y se encarga de interactuar con el software en la máquina cliente con el fin de proporcionar **soporte en tiempo de ejecución** para el sistema de objetos distribuidos. De esta forma, se lleva a cabo la comunicación entre procesos necesaria para transmitir la llamada a la máquina remota, incluyendo el empaquetamiento de los argumentos que se van a transmitir al objeto remoto.

Una arquitectura similar es necesaria en la parte del servidor, donde el soporte en tiempo de ejecución para el sistema de objetos distribuidos gestiona la recepción de los mensajes y el desempaquetado de los datos, enviando la llamada a un componente software denominado **servidor *proxy***. El servidor *proxy* invoca la llamada al método local en el objeto distribuido, pasándole los datos desempaquetados como argumentos. La llamada al método activa la realización de determinadas tareas en el servidor. El resultado de la ejecución del método es empaquetado y enviado por el servidor *proxy* al cliente *proxy*, a través del soporte en tiempo de ejecución y el soporte de red de ambas partes de la arquitectura.

Sistemas de objetos distribuidos

El paradigma de objetos distribuidos se ha adoptado de forma extendida en las aplicaciones distribuidas, para las cuales existe un gran número de herramientas disponibles basadas en este paradigma. Entre las herramientas más conocidas se encuentra:

- Java RMI (*Remote Method Invocation*),
- sistemas basados en CORBA (*Common Object Request Broker Architecture*),
- el modelo de objetos de componentes distribuidos o DCOM (*Distributed Component Object Model*), y
- herramientas y API para el protocolo SOAP (*Simple Object Access Protocol*).

De todas estas herramientas, la más sencilla es Java RMI [java.sun.com/products, 7; java.sun.com/doc, 8; developer.java.sun.com, 9; java.sun.com/marketing, 10], que se describe detalladamente en este capítulo. CORBA [corba.org, 1] y sus implementaciones son detalladas en el Capítulo 9. SOAP [w3.org, 2] es un protocolo basado en la Web y se introducirá en el Capítulo 11, al analizar las aplicaciones basadas en la Web. DCOM [microsoft.com, 3; Grimes, 4] se encuentra fuera del ámbito de este libro; los lectores interesados en DCOM deben consultar las referencias.

No es posible cubrir todas las utilidades de objetos distribuidos existentes, y además seguramente seguirán emergiendo nuevas herramientas que den soporte a este paradigma. Familiarizarse con el API de Java RMI proporciona los fundamentos y prepara al lector para aprender los detalles de utilidades similares.

Llamadas a procedimientos remotos

RMI tiene su origen en un paradigma denominado **Llamada a procedimientos remotos** o **RPC** (***Remote Procedure Call***).

La **programación procedimental** precede a la programación orientada a objetos. En la programación procedimental, un procedimiento o función es una estructura de control que proporciona la abstracción correspondiente a una acción. La acción de una función se invoca a través de una llamada a función. Para permitir usar diferentes variables, una llamada a función puede ir acompañada de una lista de datos, conocidos como argumentos. El valor o la referencia de cada argumento se le pasa a la función, e incluso podría determinar la acción que realiza dicha función. La llamada a procedimiento convencional es una llamada a un procedimiento que reside en el mismo sistema que el que la invoca, y, por tanto, se denomina **llamada a procedimiento local**.

En el modelo de llamada a procedimiento remoto, un proceso realiza una llamada a procedimiento de otro proceso, que posiblemente resida en un sistema remoto. Los datos se pasan a través de argumentos. Cuando un proceso recibe una llamada, se ejecuta la acción codificada en el procedimiento. A continuación, se notifica la finalización de la llamada al proceso que invoca la llamada y, si existe

un valor de retorno o salida, se le envía a este último proceso desde el proceso invocado. La Figura 7.3 muestra el paradigma RPC.

Basándose en el modelo RPC, han aparecido un gran número de interfaces de programación de aplicaciones. Estas API permiten realizar llamadas a procedimientos remotos utilizando una sintaxis y una semántica similar a las de las llamadas a procedimientos locales. Para enmascarar los detalles de la comunicación entre procesos, cada llamada a procedimiento remoto se transforma mediante una herramienta denominada **rpcgen** en una llamada a procedimiento local dirigida a un módulo software comúnmente denominado **resguardo** (***stub***) o, más formalmente, ***proxy***. Los mensajes que representan la llamada al procedimiento y sus argumentos se pasan a la máquina remota a través del *proxy*.

En el otro extremo, un *proxy* recibe el mensaje y lo transforma en una llamada a procedimiento local al procedimiento remoto. La Figura 7.4 muestra paso a paso la transformación de una llamada a procedimiento remoto en una llamada a procedimiento local y el paso de mensajes necesario.

Hay que destacar que hay que emplear un *proxy* a cada lado de la comunicación para proporcionar el soporte en tiempo de ejecución necesario para la comunicación entre procesos, llevándose a cabo el correspondiente empaquetado y desempaquetado de datos, así como las llamadas a sockets necesarias.

Desde su introducción a principios de los años 80, el modelo RPC se ha utilizado ampliamente en las aplicaciones de red. Existen dos API que prevalecen en este paradigma. La primera, el API ***Open Network Computing Remote Procedure Call*** [ietf.org, 5], es una evolución del API de RPC que desarrolló originalmente Sun Microsystems, Inc., a principios de los años 80. La otra API popular es ***Open Group Distributed Computing Environment*** (DCE) RPC [opennc.org, 6]. Ambas interfaces proporcionan una herramienta, **rpcgen**, para transformas las llamadas a procedimientos remotos en llamadas a procedimientos locales al resguardo.

A pesar de su importancia histórica, este libro no analiza en detalle RPC, por los siguientes motivos:

- RPC, como su nombre indica, es orientado a procedimiento. Las API de RPC emplean una sintaxis que permite realizar llamadas a procedimiento o función. Por tanto, son más adecuadas para programas escritos en un lenguaje procedimental, tal como C. Sin embargo, no son adecuadas para programas escritos en Java, el lenguaje orientado a objetos adoptado en este libro.
- En lugar de RPC, Java proporciona el API RMI, que es orientado a objetos y tiene una sintaxis más accesible que RPC.

RMI (*Remote Method Invocation*)

RMI es una implementación orientada a objetos del modelo de llamada a procedimientos remotos. Se trata de una API exclusiva para programas Java,

aunque debido a su relativa simplicidad, se trata de un buen comienzo para los estudiantes que estén aprendiendo a utilizar objetos distribuidos en aplicaciones de red.

En RMI, un servidor de objeto exporta un objeto remoto y lo registra en un servicio de directorios. El objeto proporciona métodos remotos, que pueden invocar los programas clientes.

Sintácticamente, un objeto remoto se declara como una **interfaz remota**, una extensión de la interfaz Java. El servidor de objeto implementa la interfaz remota. Un cliente de objeto accede al objeto mediante la invocación de sus métodos, utilizando una sintaxis similar a las invocaciones de los métodos locales.

El resto del capítulo se encargará de explorar en detalle el API de Java RMI.

La arquitectura de Java RMI

La Figura 7.5 muestra la arquitectura del API de Java RMI. Al igual que las API de RPC, la arquitectura de Java RMI utiliza módulos de software *proxy* para dar el soporte en tiempo de ejecución necesario para transformar la invocación del método remoto en una llamada a un método local y gestionar los detalles de la comunicación entre procesos subyacentes. Cada uno de los extremos de la arquitectura, cliente y servidor, está formado por tres capas de abstracción. A continuación se describen los dos extremos de la arquitectura.

Parte cliente de la arquitectura

1. La **capa resguardo o *stub*.** La invocación de un método remoto por parte de un proceso cliente es dirigido a un objeto *proxy*, conocido como **resguardo**. Esta capa se encuentra debajo de la capa de aplicación y sirve para interceptar las invocaciones de los métodos remotos hechas por los programas clientes; una vez interceptada la invocación es enviada a la capa inmediatamente inferior, la capa de referencia remota.
2. La **capa de referencia remota** interpreta y gestiona las referencias a los objetos de servicio remoto hechas por los clientes, así como las operaciones entre procesos relacionadas con la capa siguiente, la capa de transporte, a fin de transmitir las llamadas a los métodos a la máquina remota.
3. La **capa de transporte** está basada en TCP y, por tanto, es orientada a conexión. Esta capa y el resto de la arquitectura se encargan de la conexión entre procesos, transmitiendo los datos que representan la llamada al método a la máquina remota.

Parte servidora de la arquitectura

Conceptualmente, la parte servidora de la arquitectura también está formada por tres capas de abstracción, aunque la implementación varía dependiendo de la versión de Java.

1. La **capa esqueleto o *skeleton*** se encuentra justo debajo de la capa de aplicación y se utiliza para interactuar con la capa resguardo en la parte cliente. Como se cita en [java.sun.com/products, 7],
   "La capa esqueleto se encarga de conversar con la capa resguardo; lee los parámetros de la invocación al método del enlace, realiza la llamada al objeto que implementa el servicio remoto, acepta el valor de retorno, y a continuación devuelve dicho valor al resguardo"
2. La **capa de referencia remota**. Esta capa gestiona y transforma la referencia remota originada por el cliente en una referencia local, que es capaz de comprender la capa esqueleto.
3. La **capa de transporte**. Al igual que en la parte cliente, se trata de una capa de transporte orientada a conexión, es decir, TCP en la arquitectura de red TCP/IP.

Registro de los objetos
El API de RMI hace posible el uso de diferentes servicios de directorios para registrar un objeto distribuido. Uno de estos servicios de directorios es la **Interfaz de Nombrado y Directorios de Java** (JNDI, *Java Naming and Directory Interface)*, que es más general que el registro RMI que se utilizará en este capítulo, en el sentido de que lo pueden utilizar aplicaciones que no usan el API RMI. El registro RMI, *rmiregistry*, es un servicio de directorios sencillo proporcionado por el *kit* de desarrollo de software Java (SDK, *Java Software Development Kit*). El registro RMI es un servicio cuyo servidor, cuando está activo, se ejecuta en la **máquina del servidor del objeto**. Por convención, utiliza el puerto TCP 1099 por defecto.

Lógicamente, desde el punto de vista del desarrollador de software, las invocaciones a métodos remotos realizadas en un programa cliente interactúan directamente con los objetos remotos en un programa servidor, de la misma forma que una llamada a un método local interactúa con un objeto local. Físicamente, las invocaciones del método remoto se transforman en llamadas a los resguardo y esqueleto en tiempo de ejecución, dando lugar a la transmisión de datos a través de la red. La Figura 7.6 es un diagrama de tiempos y eventos que describe la interacción entre el resguardo y el esqueleto.

API de Java RMI

En esta sección se introduce un subconjunto del API de Java RMI. (Por simplicidad, la presentación de este capítulo no cubre los **gestores de seguridad**, que es muy recomendable utilizarlos en todas las aplicaciones RMI. Los gestores de seguridad se explican en la Sección 8.3 del próximo capítulo.) Esta sección cubre las siguientes áreas: la **interfaz remota**, el **software de la parte servidora** y el **software de la parte cliente**.

La interfaz remota
En el API de RMI, el punto inicial para crear un objeto distribuido es una **interfaz remota** Java. Una interfaz Java es una clase que se utiliza como plantilla para otras clases: contiene las declaraciones de los métodos que deben implementar las clases que utilizan dicha interfaz.

Una interfaz remota Java es una interfaz que hereda de la clase Java ***Remote***, que permite implementar la interfaz utilizando sintaxis RMI. Aparte de la extensión que se hace de la clase *Remote* y de que todas las declaraciones de los métodos deben especificar la excepción *RemoteException*, una interfaz remota utiliza la misma sintaxis que una interfaz Java local. A fin de mostrar la sintaxis básica, la Figura 7.7 muestra un ejemplo de interfaz remota.

En este ejemplo, se declara una interfaz denominada *InterfazEjemplo*. La interfaz extiende o hereda la clase Java *Remote* (línea 6), convirtiéndose de este modo en una interfaz remota.

Dentro del bloque que se encuentra entre las llaves (líneas 6-14) se encuentran las declaraciones de los dos **métodos remotos**, que se llaman *metodoEj1* (líneas 8-9) y *metodoEj2* (líneas 11-12), respectivamente.

Como se puede observar, *metodoEj1* no requiere argumentos (de ahí la lista de parámetros vacía detrás del nombre del método) y devuelve un objeto de tipo *String*. El método *metodoEj2* requiere un argumento de tipo *float* y devuelve un valor de tipo *int*.

Obsérvese que un objeto *serializable*, tal como un objeto *String* o un objeto de otra clase, puede ser un argumento o puede ser devuelto por un método remoto. Al método remoto se le pasa una copia del elemento (específicamente, una copia del objeto), sea éste un tipo primitivo o un objeto. El valor devuelto se gestiona de la misma forma, pero en la dirección contraria.

Cada declaración de un método debe especificar la excepción *java.rmi.RemoteException* en la sentencia *throws* (líneas 9 y 12). Cuando ocurre un error durante el procesamiento de la invocación del método remoto, se lanza una excepción de este tipo, que debe ser gestionada en el programa del método que lo invoca. Las causas que originan este tipo de excepción incluyen los errores que pueden ocurrir durante la comunicación entre los procesos, tal como fallos de acceso y fallos de conexión, así como problemas asociados exclusivamente a la invocación de métodos remotos, como por ejemplo no encontrar el objeto, el resguardo o el esqueleto.

Software de la parte servidora
Un servidor de objeto es un objeto que proporciona los métodos y la interfaz de un objeto distribuido. Cada servidor de objeto debe (1) implementar cada uno de los métodos remotos especificados en la interfaz, y (2) registrar en un servicio de

directorios un objeto que contiene la implementación. Se recomienda que las dos partes se realicen en clases separadas, como se ilustra a continuación.

La implementación de la interfaz remota

Se debe crear una clase que implemente la interfaz remota. La sintaxis es similar a una clase que implementa una interfaz local. La Figura 7.8 muestra un esquema o plantilla de la implementación.

Las sentencias de importación (*import*) (líneas 1-2) son necesarias para que el código pueda utilizar las clases *UnicastRemoteObject* y *RemoteException*.

La cabecera de la clase (línea 8) debe especificar (1) que es una subclase de la clase Java *UnicastRemoteObject*, y (2) que implementa una interfaz remota específica, llamada *InterfazEjemplo* en la plantilla. (Nota: Un objeto *UnicastRemoteObject* da soporte a RMI *unicast*, es decir, RMI utilizando comunicación unidifusión entre procesos. Presumiblemente, se puede implementar una clase *MulticastRemoteObject*, que dé soporte a RMI con comunicación multidifusión.)

Se debe definir un constructor de la clase (líneas 11-13). La primera línea del código debe ser una sentencia (la llamada *super()*) que invoque al constructor de la clase base. Puede aparecer código adicional en el constructor si se necesita.

A continuación, debe aparecer la implementación de cada método remoto (líneas 15-21). La cabecera de cada método debe coincidir con la cabecera de dicho método en el fichero de la interfaz.

La Figura 7.9 muestra un diagrama UML para la clase *ImplEjemplo*.

Generación del resguardo y del esqueleto

En RMI, un objeto distribuido requiere un *proxy* por cada uno de los servidor y cliente de objeto, conocidos como esqueleto y resguardo del objeto, respectivamente. Estos *proxies* se generan a partir de la implementación de una interfaz remota utilizando una herramienta del SDK de Java: el compilador RMI ***rmic***. Para utilizar esta herramienta, se debe ejecutar el siguiente mandato en una interfaz de mandatos UNIX o Windows:

```
rmic <nombre de la clase de la implementación de la interfaz remota>
```

Por ejemplo:

```
rmic ImplEjemplo
```

Si la compilación se realiza de forma correcta, se generan dos ficheros *proxy*, cada uno de ellos con el prefijo correspondiente al nombre de la clase de la implementación. Por ejemplo, *ImplEjemplo_skel.class* y *ImplEjemplo_stub.class*.

El fichero del resguardo para el objeto, así como el fichero de la interfaz remota deben compartirse con cada cliente de objeto: estos ficheros son imprescindibles para que el programa cliente pueda compilar correctamente. Una copia de cada fichero debe colocarse manualmente en la parte cliente (es decir, debe colocarse una copia del fichero en un directorio apropiado del cliente). Adicionalmente, Java RMI dispone de una característica denominada **descarga de resguardo**, que consiste en que el cliente obtiene de forma dinámica el fichero de resguardo. Esta característica se estudiará en el Capítulo 8, donde se analizan varios temas avanzados de RMI.

El servidor de objeto

La clase del servidor de objeto instancia y exporta un objeto de la implementación de la interfaz remota. La Figura 7.10 muestra una plantilla para la clase del servidor de objeto.

En los siguientes párrafos se analizan las diferentes partes de esta plantilla.

Creación de un objeto de la implementación de la interfaz remota. En la línea 19, se crea un objeto de la clase que **implementa** la interfaz remota; a continuación, se **exportará** la referencia a este objeto.

Exportación del objeto. Las líneas 20-23 de la plantilla exportan el objeto. Para exportar el objeto, se debe registrar su referencia en un servicio de directorios. Como ya se ha mencionado anteriormente, en este capítulo se utilizará el servicio *rmiregistry* del SDK Java. Un servidor *rmiregistry* debe ejecutarse en el nodo del servidor de objeto para poder registrar objetos RMI.

Cada registro RMI mantiene una lista de objetos exportados y posee una interfaz para la búsqueda de estos objetos. Todos los servidores de objetos que se ejecutan en la misma máquina pueden compartir un mismo registro. Alternativamente, un proceso servidor individual puede crear y utilizar su propio registro si lo desea, en cuyo caso múltiples servidores *rmiregistry* pueden ejecutarse en diferentes números de puertos en la misma máquina, cada uno con una lista diferente de objetos exportados.

En un sistema de producción, debería existir un servidor *rmiregistry*, ejecutando de forma continua, presumiblemente en el puerto por defecto 1099. Para los ejemplos de este libro, se supone que existe un registro RMI siempre disponible, aunque se utiliza código para arrancar una copia del servidor bajo demanda en un puerto de la elección del lector, de forma que cada estudiante pueda utilizar una copia diferente del registro para sus experimentos y no existan colisiones de nombres.

En la plantilla del servidor de objeto, se implementa el método estático *arrancarRegistro()* (líneas 34-51), que arranca un servidor de registro RMI si no está actualmente en ejecución, en un número de puerto especificado por el usuario (línea 20):

```
arrancarRegistro(numPuertoRMI);
```

En un sistema de producción donde se utilice el servidor de registro RMI por defecto y esté ejecutando continuamente, la llamada *arrancarRegistro* – y por tanto el método *arrancarRegistro* – puede omitirse.

En la plantilla del servidor de objeto, el código para exportar un objeto (líneas 22-23) se realiza del siguiente modo:

```
// registrar el objeto con el nombre "ejemplo"
URLRegistro = "rmi://localhost:" + numPuerto + "/ejemplo";
Naming.rebind(URLRegistro, objExportado);
```

La clase *Naming* proporciona métodos para almacenar y obtener referencias del registro. En particular, el método *rebind* permite almacenar en el registro una referencia a un objeto con un URL de la forma

```
rmi://<nombre máquina>:<número puerto>/<nombre referencia>
```

El método *rebind* sobreescribe cualquier referencia en el registro asociada al nombre de la referencia. Si no se desea sobreescribir, existe un método denominado *bind*.

El nombre de la máquina debe corresponder con el nombre del servidor, o simplemente se puede utilizar "*localhost*". El nombre de la referencia es un nombre elegido por el programador y debe ser único en el registro.

El código del ejemplo comprueba primero si se está ejecutando actualmente un registro RMI en el puerto por defecto. Si no es así, se activa un registro RMI.

Alternativamente, se puede activar un registro RMI manualmente utilizando la utilidad *rmiregistry*, que se encuentra en el SDK, a través de la ejecución del siguiente mandato en el intérprete de mandatos:

```
rmiregistry <número puerto>
```

donde el número de puerto es un número de puerto TCP. Si no se especifica ningún puerto, se utiliza el puerto por defecto 1099.

Cuando se ejecuta un servidor de objeto, la exportación de los objetos distribuidos provoca que el proceso servidor comience a escuchar por el puerto y espere a que los clientes se conecten y soliciten el servicio del objeto. Un servidor de objeto RMI

es un servidor concurrente: cada solicitud de un cliente de objeto se procesa a través de un hilo independiente del servidor. Dado que las invocaciones de los métodos remotos se pueden ejecutar de forma concurrente, es importante que la implementación de un objeto remoto sea *thread-safe*. Los lectores pueden revisar este tema en “Programación concurrente” en la Sección 1.5 del Capítulo 1.

Software de la parte cliente
La clase cliente es como cualquier otra clase Java. La sintaxis necesaria para hacer uso de RMI supone localizar el registro RMI en el nodo servidor y buscar la referencia remota para el servidor de objeto; a continuación se realizará un *cast* de la referencia a la clase de la interfaz remota y se invocarán los métodos remotos. La Figura 7.11 presenta una plantilla para el cliente de objeto.

Las sentencias de importación. Las sentencias de importación (líneas 1-4) se necesitan para que el programa pueda compilar.

Búsqueda del objeto remoto. El código entre las líneas 24 y 27 permite buscar el objeto remoto en el registro. El método *lookup* de la clase *Naming* se utiliza para obtener la referencia al objeto, si existe, que previamente ha almacenado en el registro el servidor de objeto. Obsérvese que se debe hacer un *cast* de la referencia obtenida a la clase de la interfaz remota (*no* a su implementación).

```
String URLRegistro = “rmi://localhost:” + numPuerto +
“/ejemplo”;
InterfazEjemplo h =
(InterfazEjemplo)Naming.lookup(URLRegistro);
```

Invocación del método remoto. Se utiliza la referencia a la interfaz remota para invocar cualquiera de los métodos de dicha interfaz, como se muestra en las líneas 29-30 del ejemplo:

```
String mensaje = h.metodoEj1();
System.out.println(mensaje);
```

Obsérvese que la sintaxis utilizada para la invocación de los métodos remotos es igual que la utilizada para invocar métodos locales.

Una aplicación RMI de ejemplo

Desde la Figura 7.12 hasta la Figura 7.15 se muestra la lista completa de ficheros necesarios para crear la aplicación RMI *HolaMundo*. El servidor exporta un objeto que contiene un único método, denominado *decirHola*. Se recomienda al lector identificar en el código las diferentes partes que se han descrito en la sección anterior.

Una vez comprendida la estructura básica del ejemplo de aplicación RMI presentado, el lector debería ser capaz de utilizar esta sintaxis como plantilla para construir cualquier aplicación RMI, modificando la presentación y la lógica de la aplicación; la lógica del servicio (utilizando RMI) queda inalterada.

La tecnología RMI es un buen candidato como componente software en la capa de servicio. Un ejemplo de aplicación industrial es un sistema de informe de gastos de una empresa, que se muestra en la Figura 7.16 y que se describe en [java.sun.com/marketing, 8]. En la aplicación mostrada, el servidor de objeto proporciona métodos remotos que permiten a los clientes de objeto buscar o actualizar los datos en una base de datos de gastos. Los programas clientes del objeto proporcionan la lógica de aplicación o negocio necesaria para procesar los datos y la lógica de presentación para la interfaz de usuario.

Java RMI posee un gran número de características. Este capítulo ha presentado un conjunto muy básico de estas características, como ejemplo de un sistema de objetos distribuidos. Algunas de las características avanzadas de RMI más interesantes se describirán en el siguiente capítulo.

Pasos para construir una aplicación RMI

Una vez vistos algunos de los aspectos del API de RMI, se va a pasar a describir paso a paso el procedimiento para construir una aplicación RMI, de forma que el lector pueda experimentar con dicho paradigma. Se describe tanto la parte del servidor de objeto como del cliente de objeto. Hay que tener en cuenta que en un entorno de producción es probable que el desarrollo de software de ambas partes sea independiente.

El algoritmo es expresado en términos de una aplicación denominada *Ejemplo*. Los pasos se aplicarán para la construcción de cualquier aplicación, reemplazando el nombre *Ejemplo* por el nombre de la aplicación.

Algoritmo para desarrollar el software de la parte servidora

1. Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
2. Especificar la interfaz remota del servidor en *InterfazEjemplo.java*. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
3. Implementar la interfaz en *ImplEjemplo.java*. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
4. Utilizar el compilador de RMI ***rmic*** para procesar la clase de la implementación y generar los ficheros de resguardo y esqueleto para el objeto remoto:

```
rmic ImplEjemplo
```

Los ficheros generados se encontrarán en el directorio como *ImplEjemplo_Skel.class* e *ImplEjemplo_Stub.class*. Se deben repetir los pasos 3 y 4 cada vez que se realice un cambio a la implementación de la interfaz.
5. Crear el programa del servidor de objeto *ServidorEjemplo.java*. Compilarlo y revisarlo hasta que no exista ningún error de sintaxis.
6. Activar el servidor de objeto.

```
java ServidorEjemplo
```

Algoritmo para desarrollar el software de la parte cliente
1. Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
2. Obtener una copia del fichero *class* de la interfaz remota. Alternativamente, obtener una copia del fichero fuente de la interfaz remota y compilarlo utilizando ***javac*** para generar el fichero *class* de la interfaz.
3. Obtener una copia del fichero de resguardo para la implementación de la interfaz, *ImplEjemplo_Stub.class*.
4. Desarrollar el programa cliente *ClienteEjemplo.java*. Compilarlo y revisarlo hasta que no exista ningún error de sintaxis.
5. Activar el cliente.

```
java ClienteEjemplo
```

La Figura 7.17 muestra la colocación de los ficheros de una aplicación en las partes cliente y servidora. Los ficheros *class* de la interfaz remota y del resguardo para cada objeto remoto deben estar en la máquina donde se encuentra el cliente de objeto, junto con la clase del cliente de objeto. En la parte servidora se deben encontrar los fichero *class* de la interfaz, del servidor de objeto, de la implementación de la interfaz y del esqueleto para el objeto remoto.

Pruebas y depuración

Como cualquier tipo de programación de red, las pruebas y depuración de los procesos concurrentes no son triviales. Es recomendable utilizar los siguientes pasos incrementales a la hora de desarrollar una aplicación RMI:
1. Construir una plantilla para un programa RMI básico. Empezar con una interfaz remota que sólo contenga la declaración de un método, su implementación utilizando un resguardo, un programa servidor que exporte el objeto y un programa cliente con código que sólo invoque al método remoto. Probar la plantilla en una máquina hasta que se pueda ejecutar correctamente el método remoto.
2. Añadir una declaración cada vez a la interfaz. Con cada adición, modificar el programa cliente para que invoque al método que se ha añadido.
3. Rellenar la definición de cada método remoto uno a uno. Probar y depurar de forma detallada cada método añadido antes de incluir el siguiente.

4. Después de que todos los métodos remotos se han probado detalladamente, crear la aplicación cliente utilizando una técnica incremental. Con cada incremento, probar y depurar los programas.
5. Distribuir los programas en máquinas separadas. Probarlos y depurarlos.

Comparación entre RMI y el API de sockets

El API de RMI, como API representativa del paradigma de objetos distribuidos, es una herramienta eficiente para construir aplicaciones de red. Puede utilizarse en lugar del API de sockets (que representa el paradigma de paso de mensajes) para construir una aplicación de red rápidamente. Sin embargo, debería tenerse en cuenta que además de ventajas, también existen desventajas en esta opción.

Algunas de estas ventajas y desventajas se enumeran a continuación:

- El API de sockets está más cercano al sistema operativo, por lo que tiene menos sobrecarga de ejecución. RMI requiere soporte software adicional, incluyendo los *proxies* y el servicio de directorio, que inevitablemente implican una sobrecarga en tiempo de ejecución. Para aplicaciones que requieran alto rendimiento, el API de sockets puede ser la única solución viable.
- Por otro lado, el API de RMI proporciona la abstracción necesaria para facilitar el desarrollo de software. Los programas desarrollados con un nivel más alto de abstracción son más comprensibles y por tanto más sencillos de depurar.
- Debido a que el API de sockets opera a más bajo nivel, se trata de una API típicamente independiente de plataforma y lenguaje. Puede no ocurrir lo mismo con RMI. Java RMI, por ejemplo, requiere soportes de tiempo de ejecución específicos de Java. Como resultado, una aplicación implementada con Java RMI debe escribirse en Java y sólo se puede ejecutar en plataformas Java.

La elección de un paradigma y una API apropiados es una decisión clave en el diseño de una aplicación. Dependiendo de las circunstancias, es posible que algunas partes de la aplicación utilicen un paradigma o API y otras partes otros.

Debido a la relativa facilidad con la que las aplicaciones de red pueden desarrollarse utilizando RMI, RMI es un buen candidato para el desarrollo rápido de un prototipo de una aplicación.

Para pensar

El modelo de computación distribuida orientada a objetos presentado en este capítulo está basado en la visión de que, desde el punto de vista del programador, no hay una distinción esencial entre los objetos locales y los objetos remotos. Aunque esta visión es ampliamente aceptada como la base de los sistemas de objetos distribuidos, existen detractores de la misma. Los autores de [research.sun.com, 11], por ejemplo, afirman que esta visión, aunque conveniente, es inapropiada porque ignora las diferencias inherentes entre los objetos locales y remotos. El Ejercicio 11 al final del capítulo pide al lector profundizar más en el estudio de este argumento.

Resumen

Este capítulo ha introducido el paradigma de objetos distribuidos. A continuación se resumen algunos de los puntos claves:

- El paradigma de objetos distribuidos posee un nivel de abstracción más alto que el paradigma de paso de mensajes.
- Mediante el uso de este paradigma, un proceso invoca métodos de un objeto remoto, pasando los datos como argumentos y recibiendo un valor de retorno con cada llamada, de forma similar a las llamadas a los métodos locales.
- En un sistema de objetos distribuidos, un servidor de objeto consiste en un objeto distribuido que posee métodos que un cliente de objeto puede invocar. Cada extremo de la comunicación requiere un *proxy*, que interactúa con el soporte en tiempo de ejecución del sistema para llevar a cabo la comunicación entre procesos necesaria. Adicionalmente, debe existir un registro de objetos que permita a los objetos distribuidos registrarse y poder hacer búsquedas.
- Entre los protocolos de los sistemas de objetos distribuidos más conocidos se encuentran Java RMI (*Remote Method Invocation*), el modelo de objetos de componentes distribuidos DCOM, la arquitectura CORBA (*Common Object Request Broker Architecture*), y el protocolo SOAP (*Simple Object Access Protocol*).
- Java RMI es un sistema representativo de los sistemas de objetos distribuidos. Algunos de los temas relacionados con RMI que se han tratado en este capítulo son:
  - La arquitectura del API Java RMI incluye tres capas en ambos extremos, el cliente y el servidor. En la parte cliente, la capa resguardo accepta una invocación a un método remoto y la transforma en mensajes, que envía a la parte servidora. En la parte servidora, la capa esqueleto recibe los mensajes y los transforma en llamadas locales al método remoto. Para registrar un objeto, se debe utilizar un servicio de directorios, como JNDI o el registro RMI.
  - El software de una aplicación RMI incluye una interfaz remota, software en la parte servidora, y software en la parte cliente. Este capítulo presentó la sintaxis y los algoritmos recomendados para desarrollar este software.
- Se analizaron las diferencias entre el API de sockets y el API de Java RMI.

Ejercicios

1. Compare y contraste el paradigma de paso de mensajes y el paradigma de objetos distribuidos.
2. Compare y contraste una llamada a procedimiento local y una llamada a procedimiento remoto.
3. Describa la arquitectura de Java RMI. ¿Cuál es el papel del registro RMI?
4. Considérese una aplicación sencilla, donde un cliente envía dos valores enteros a un servidor, que suma los valores y devuelve el resultado al cliente.
   a. Describa cómo se implementaría la aplicación mediante el uso del API de sockets. Describa los mensajes intercambiados y las acciones correspondientes a cada mensaje.

b. Describa cómo se implementaría la aplicación mediante el uso del API RMI. Describa la interfaz, los métodos remotos, y la invocación de los métodos remotos en el programa cliente.

5. Este ejercicio utiliza el ejemplo *HolaMundo*.
   a. Cree un directorio para este ejercicio. Coloque los ficheros fuente del ejemplo *HolaMundo* en el directorio.
   b. Compile *HolaMundoInterfaz.java* y *HolaMundoImpl.java*.
   c. Utilice *rmic* para compilar *HolaMundoImpl*. Mire la carpeta para comprobar que se han generado las clases del *proxy*. ¿Cuáles son sus nombres?
   d. Compile *HolaMundoServidor.java*. Compruebe el contenido de la carpeta.
   e. Ejecute el servidor, especificando un número de puerto aleatorio para el registro RMI. Compruebe los mensajes que se muestran, incluyendo la lista de nombres actualmente registrados. ¿Se puede ver el nombre bajo el cuál se ha registrado el objeto remoto (tal y como se especifica en el programa)?
   f. Compile y ejecute *HolaMundoCliente.java*. Cuando se solicite, especifique "localhost" como nombre de la máquina y el número de puerto RMI previamente introducido. ¿Qué sucede? Explíquelo.
   g. Ejecute el programa cliente en una máquina separada. ¿La ejecución fue correcta?

6. Cree una nueva carpeta. Copie todos los ficheros fuente del ejemplo *HolaMundo* a la carpeta. Añada código al método *decirHola* de *HolaMundoImpl.java*, de forma que haya un retardo de 5 segundos antes de que el método devuelva el flujo de ejecución. Esto tiene el efecto de alargar artificialmente la latencia de cada invocación del método. Compile y arranque el servidor.
   En pantallas separadas, arranque dos o más clientes. Observe la secuencia de eventos en las pantallas. ¿Se puede afirmar si el servidor de objeto ejecuta las llamadas a los métodos concurrentemente o iterativamente? Explíquelo.

7. Cree una nueva carpeta. Copie todos los ficheros fuente del ejemplo *HolaMundo* a la carpeta. Modifique el método *decirHola* de forma que se le pase un argumento, una cadena de caracteres con un nombre, y que la cadena que devuelve sea la cadena "Hola Mundo" concatenada con el nombre pasado como argumento.
   a. Muestre las modificaciones del código.
   b. Recompile y ejecute el servidor y a continuación el cliente. Describa y explique lo que sucede.
   c. Ejecute *rmic* de nuevo para generar los nuevos *proxies* de la interfaz modificada, y a continuación ejecutar el servidor y el cliente.
   Entregue los listados fuentes y las salidas de la ejecución.

8. Utilice RMI para implementar un servidor y cliente *Daytime*.
9. Utilizando RMI, escriba una aplicación para un prototipo de un sistema de consultas de opinión. Asúmase que sólo un tema se va a encuestar. Los entrevistados pueden responder *sí*, *no* o *ns/nc*. Escriba una aplicación servidora, que acepte los votos, guarde el recuento (en memoria), y proporcione el recuento actual a aquellos que estén interesados.

a. Escriba el fichero de interfaz primero. Debería proporcionar métodos remotos para aceptar una respuesta a la encuesta, proporcionando el recuento actual (ej. “10 sí, 2 no, 5 ns/nc) sólo cuando el cliente lo requiera.
b. Diseñe e implemente un servidor que (i) exporte los métodos remotos, y (ii) mantenga información de estado (los recuentos). Obsérvese que las actualizaciones de los recuentos deben protegerse con exclusión mutua.
c. Diseñe e implemente una aplicación cliente que proporcione una interfaz de usuario para aceptar una respuesta y/o una petición, y para interactuar con el servidor apropiadamente a través de la invocación de métodos remotos.
d. Pruebe la aplicación ejecutando dos o más clientes en máquinas diferentes (preferiblemente en plataformas diferentes).
e. Entregue los listados de los ficheros, que deben incluir los ficheros fuente (el fichero de interfaz, los ficheros del servidor y los ficheros del cliente), y un fichero LEEME que explique los contenidos y las interrelaciones de los ficheros fuente, así como el procedimiento para ejecutar el trabajo.

10. Cree una aplicación para gestionar unas elecciones. El servidor exporta dos métodos:
    - *emitirVoto*, que acepta como parámetro una cadena de caracteres que contiene un nombre de candidato (Gore o Bush), y no devuelve nada, y
    - *obtenerResultado*, que devuelve, en un vector de enteros, el recuento actual para cada candidato.

    Pruebe la aplicación ejecutando todos los procesos en una máquina. A continuación pruebe la aplicación ejecutando el proceso cliente y servidor en máquinas *separadas*.
    Entregue el código fuente de la interfaz remota, el servidor y el cliente.
11. Lea la referencia [research.sun.com, 11]. Resuma las razones por las que los autores encuentran fallos en el modelo de objetos distribuidos, que minimiza las diferencias de programación entre las invocaciones local y remoto de métodos. ¿Está de acuerdo con los autores? ¿Cuál podría ser un modelo alternativo para objetos distribuidos que resolviera los problemas identificados por los autores? (*Pista*: Busque información sobre la tecnología *Jini* [sun.com, 12].)

Referencias

1. Object Management Group. Website de OMG Corba, http://www.corba.org
2. World Wide Web Consortium. SOAP versión 1.2 Parte 0: Primero, http://www.w3.org/TR/soap12-part0/
3. Modelo de Objetos de Componentes Distribuidos (DCOM) – Documentos, especificaciones, ejemplos y recursos de Microsoft DCOM, http://www.microsoft.com/com/tech/DCOM.asp, Microsoft.
4. Richard T. Grimes. *Professional DCOM Programming.* Chicago, IL: Wrox Press, Inc., 1997.
5. RFC 1831: Especificación del protocolo Llamada a Procedimientos Remotos, versión 2, Agosto 1995, http://www.ietf.org/rfc/rfc1831.txt

6. The Open Group, DCE 1.1: Remote Procedure Call, http://www.opennc.org/public/pubs/catalog/c706.htm
7. Java Remote Method Invocation, http://java.sun.com/products/jdk/rmi
8. RMI – Tutorial de Java, http://java.sun.com/docs/books/tutorial/rmi
9. Introducción a la computación distribuida con RMI, http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html
10. Java Remote Method Invocation – Computación distribuida para Java, http://java.sun.com/marketing/collateral/javarmi.html
11. Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. *"A Note on Distributed Computing"*, Informe TR-94-29, Sun Microsystems Laboratories, 1994, http://research.sun.com/research/techrep/1994/smli_tr-94-29.pdf
12. Jini Network Technology, *"An Executive Overview"*, white paper. Sun Microsystems, Inc., 2001, http://www.sun.com/software/jini/whitepapers/jini-execoverview.pdf

Figuras


```
 1   // fichero: InterfazEjemplo.java
 2   // implementada por una clase servidor Java RMI.
 3
 4   import java.rmi.*
 5
 6   public interface InterfazEjemplo extends Remote {
 7     // cabecera del primer método remoto
 8     public String metodoEj1( )
 9       throws java.rmi.RemoteException;
10     // cabecera del segundo método remoto
11     public int metodoEj2( float parametro)
12       throws java.rmi.RemoteException;
13     // cabeceras de otros métodos remotos
14   } // fin interfaz
```

Figura 7.8 Sintaxis de un ejemplo de implementación de interfaz remota.

```
 1   import java.rmi.*;
 2   import java.rmi.server.*;
```

```
 3
 4   /**
 5   Esta clase implementa la interfaz remota InterfazEjemplo.
 6   */
 7
 8   public class ImplEjemplo extends UnicastRemoteObject
 9     implements InterfazEjemplo {
10
11     public ImplEjemplo( ) throws RemoteException {
12       super( );
13     }
14
15     public metodoEj1() throws RemoteException {
16       // código del método
17     }
18
19     public metodoEj2() throws RemoteException {
20       // código del método
21     }
22
23   } // fin clase
```

Figura 7.9 El diagrama de clases UML para *ImplEjemplo*.

Figura 7.10 Sintaxis de un ejemplo de un servidor de objeto.

```
1    import java.rmi.*;
2    import java.rmi.server.*;
3    import java.rmi.registry.Registry;
4    import java.rmi.registry.LocateRegistry;
5    import java.net.*;
6    import java.io.*;
7
8    /**
9    * Esta clase representa el servidor de un objeto
10   * distribuido de la clase ImplEjemplo, que implementa la
11   * interfaz remota InterfazEjemplo.
12   */
13
14   public class ServidorEjemplo {
15     public static void main(String args[]) {
16       String numPuertoRMI, URLRegistro;
17       try{
18         // código que permite obtener el valor del número
de puerto
19         ImplEjemplo objExportado = new ImplEjemplo();
20         arrancarRegistro(numPuertoRMI);
21         // registrar el objeto bajo el nombre “ejemplo”
```

```
22          URLRegistro = "rmi://localhost:" + numPuerto +
            "/ejemplo";
23          Naming.rebind(URLRegistro, objExportado);
24          System.out.println("Servidor ejemplo preparado.");
25        } // fin try
26        catch (Exception excr) {
27          System.out.printl(
28          "Excepción en ServidorEjemplo.main: " + excr);
29        } // fin catch
30      } // fin main
31
32    // Este método arranca un registro RMI en la máquina
33    //local, si no existe en el número de puerto
  especificado
34    private static void arrancarRegistro (int numPuertoRMI)
35          throws RemoteException {
36        try {
37          Registry registro =
  LocateRegistry.getRegistry(numPuertoRMI);
38          registro.list();
39          // El método anterior lanza una excepción
40          // si el registro no existe.
41        }
42        catch (RemoteException exc) {
43          //No existe un registro válido en este puerto.
44          System.out.println(
45               "El registro RMI no se puede localizar en el
  puerto:
46               + RMIPortNum);
47          Registry registro
=LocateRegistry.createRegistry(numPuertoRMI);
48           System.out.println(
49             "Registro RMI creado en el puerto " +
RMIPortNum);
50        } // fin catch
51      } // fin arrancarRegistro
52
53    } // fin clase
```

Figura 7.11 Plantilla para un cliente de objeto

```
1   import java.io.*;
2   import java.rmi.*;
3   import java.rmi.registry.Registry;
4   import java.rmi.registry.LocateRegistry;
5
6   /**
```

```
 7   * Esta clase representa el cliente de un objeto
 8   * distribuido de la clase ImplEjemplo, que implementa la
 9   * interfaz remota InterfazEjemplo.
10   */
11
12   public class ClienteEjemplo {
13     public static void main(String args[ ]) {
14       try {
15         int puertoRMI;
16         String nombreNodo;
17         String numPuerto;
18         // Código que permite obtener el nombre del nodo y
19         el número de puerto del registro
20
21         // Búsqueda del objeto remoto y cast de la
22         // referencia con la correspondiente clase
23         // de la interfaz remota – reemplazar "localhost
           por el nombre del nodo del objeto remoto.
24         String URLRegistro =
25             "rmi://localhost:" + numPuerto + "/ejemplo";
26         InterfazEjemplo h =
27             (InterfazEjemplo) Naming.lookup(URLRegistro);
28         // invocar el o los métodos remotos
29         String mensaje = h.metodoEj1();
30         System.out.println(mensaje);
31         // el método metodoEj2 puede invocarse del mismo
           modo
32       }// fin try
33       catch (Exception exc) {
34         exc.printStackTrace();
35             } // fin catch
36     } // fin main
37     // Posible definición de otros métodos de la clase
38   } // fin clase
```

Figura 7.12 *HolaMundoInt.java*

```
 1   // Un ejemplo sencillo de interfaz RMI - M. Liu
 2   import java.rmi.*;
 3
 4   /**
 5   * Interfaz remota.
 6   * @author M. L. Liu
 7   */
 8
 9   public interface HolaMundoInt extends Remote {
10   /**
11   * Este método remoto devuelve un mensaje.
```

```
12   * @para name – una cadena de caracteres con un nombre.
13   * @return - una cadena de caracteres.
14   */
15     public String decirHola(String nombre)
16       throws java.rmi.RemoteException;
17
18   }
```

Figura 7.13 *HolaMundoImpl.java*

```
 1   import java.rmi.*;
 2   import java.rmi.server.*;
 3
 4   /**
 5   * Esta clase implementa la interfaz remota
 6   * HolaMundoInt.
 7   * @author M. L. Liu
 8   */
 9
10   public class HolaMundoImpl extends UnicastRemoteObject
11   implements HolaMundoInt {
12
13        public HolaMundoImpl() throws RemoteException {
14             super();
15        }
16
17        public String decirHola(String nombre)
18           throws RemoteException {
19             return "Hola mundo" + nombre;
20        }
21   } //fin clase
```

Figura 7.14 *HolaMundoServidor.java*

```
 1   import java.rmi.*;
 2   import java.rmi.server.*;
 3   import java.rmi.registry.Registry;
 4   import java.rmi.registry.LocateRegistry;
 5   import java.net.*;
 6   import java.io.*;
 7
 8    /**
 9    * Esta clase representa el servidor de un objeto
10    * de la clase HolaMundo, que implementa la
11    * interfaz remota HolaMundoInterfaz.
12    * @author M. L. Liu
13    */
14
15    public class HolaMundoServidor {
```

```
16            public static void main (String args[]) {
17                 InputStreamReader ent = new
   InputStreamReader(System.in));
18                 BufferedReader buf = new
   BufferedReader(ent);
19                 String numPuerto, URLRegistro;
20                 try {
21                      System.out.println("Introducir el
   número de puerto del registro RMI:");
22                      numPuerto = buf.readLine().trim();
23                      int numPuertoRMI =
   Integer.parseInt(numPuerto);
24                      arrancarRegistro(numPuertoRMI);
25                      HolaMundoImpl objExportado = new
   HolaMundoImpl();
26                      URLRegistro = "rmi://localhost:"+
   numPuerto + "/holaMundo";
27                      Naming.rebind(URLRegistro,
   objExportado);
28       /*   */        System.out.println
29   /*  */         ("Servidor registrado. El registro
   contiene actualmente:");
30   /*  */         // lista de los nombres que se encuentran
   en el registro actualmente
31   /*  */         listaRegistro(URLRegistro);
32                      System.out.println("Servidor
   HolaMundo preparado.");
33                 } // fin try
34                 catch (Exception excr) {
35                      System.out.println("Excepción en
   HolaMundoServidor.main: " + excr);
36                 } // fin catch
37            } // fin main
38
39            // Este método arranca un registro RMI en la
   máquina
40            // local, si no existe en el número de puerto
   especificado.
41            private static void arrancarRegistro(int
   numPuertoRMI)
42                      throws RemoteException {
43                 try {
44                      Registry registro =
   LocateRegistry.getRegistry(numPuertoRMI);
45                      registro.list(); // Esta llamada
   lanza
```

```
46                      // una excepción si el registro no
   existe
47                 }
48                 catch (RemoteException e) {
49                      // Registro no válido en este puerto
50   /*  */        System.out.println
51   /*  */        ("El registro RMI no se puede localizar
   en el puerto "
52   /*  */        + numPuertoRMI);
53                      Registry registro =
54
    LocateRegistry.createRegistry(numPuertoRMI);
55   /*  */        System.out.println(
56   /*  */        "Registro RMI creado en el puerto " +
   numPuertoRMI);
57                 } // fin catch
58            } // fin arrancarRegistro
59
60            // Este método lista los nombres registrados
   con un objeto Registry
61            private static void listaRegistro(String
   URLRegistro)
62                 throws RemoteException,
   MalformedURLException {
63                 System.out.println("Registro " +
   URLRegistro * " contiene: ");
64                 String [] nombres =
   Naming.list(URLRegistro);
65                 for (int i=0; i<nombres.length; i++)
66                      System.out.println(names[i]);
67            } // fin listaRegistro
68
69       } // fin clase
```

Figura 7.15 *HolaMundoCliente.java*

```
 1   import java.io.*;
 2   import java.rmi.*;
 3
 4   /**
 5   * Esta clase representa el cliente de un objeto
 6   * distribuido de clase HolaMundo, que implementa la
 7   * interfaz remota HolaMundoInterfaz.
 8   * @author M. L. Liu
 9   */
10
11   public class HolaMundoCliente {
```

```
12
13        public static void main(String args[]) {
14             try {
15                  int numPuertoRMI;
16                  String nombreNodo;
17                  InputStreamReader ent = new
    InputStreamReader(System.in);
18                  BufferedReader buf = new
    BufferedReader(ent);
19                  System.out.println("Introducir el nombre
    del nodo del registro RMI:");
20                  nombreNodo = buf.readLine();
21                  System.out.println("Introducir el número
    de puerto del registro RMI:");
22                  String numPuerto = buf.readLine();
23                  numPuertoRMI =
    Integer.parseInt(numPuerto);
24                  String URLRegistro =
25                       "rmi://" + nombreNodo + ":" +
    numPuerto + "/holaMundo";
26                  // Búsqueda del objeto remoto y cast del
    objeto de la interfaz
27                  HolaMundoInterfaz h =
28
    (HolaMundoInterfaz)Naming.lookup(URLRegistro);
29                  System.out.println("Busqueda completa ");
30                  // Invocar el método remoto
31                  String mensaje = h.decirHola("Pato
    Donald");
32                  System.out.println("HolaMundoCliente: " +
    mensaje);
33             } // fin try
34             catch (Exception e) {
35                  System.out.println("Excepcion en
    HolaMundoCliente: " + e);
36             } // fin catch
37        } // fin main
38   } // fin clase
```

Figura 7.16 Un ejemplo de aplicación RMI.
Figura 7.17 Colocación de los ficheros en una aplicación RMI.

Notas al margen

Página 206 (Anclado al segundo párrafo)

En un lenguaje de programación, una referencia es un “manejador” de un objeto; es decir, la representación a través de la cual se puede localizar dicho objeto en el computador donde reside.

Página 206 (Anclado al segundo párrafo)
El término *proxy*, en el contexto de la computación distribuida, se refiere a un componente software que sirve como intermediario de otros componentes software.

Página 210 (Anclado al último párrafo)
El esqueleto, el *proxy* de la parte servidora está *“deprecated”*, es decir, no se utiliza en las nuevas versiones, a partir de la versión Java 1.2. Su funcionalidad queda reemplazada por el uso de una técnica conocida como “reflexión”. Este libro continúa incluyendo el esqueleto en la arquitectura como concepto de la misma.

Página 211 (Anclado al segundo párrafo)
El SDK de Java se puede instalar en una máquina local. Permite el uso de bibliotecas de clases Java y herramientas, tales como el compilador de Java *javac*.

Página 213 (Anclado al penúltimo párrafo)
Un objeto *serializable* es un objeto de una clase que puede “aplanarse”, de forma que se puede empaquetar para su transmisión a través de la red.

Página 217 (Anclado al penúltimo párrafo)
Una colisión de nombres se produce cuando se intenta exportar un objeto cuyo nombre coincide con el nombre de otro objeto ya existente en el registro.

Página 227 (Anclado al penúltimo párrafo)
En ingeniería de software, un prototipo es una versión inicial que se realiza de forma rápida para mostrar la interfaz de usuario y las funciones de una aplicación propuesta.