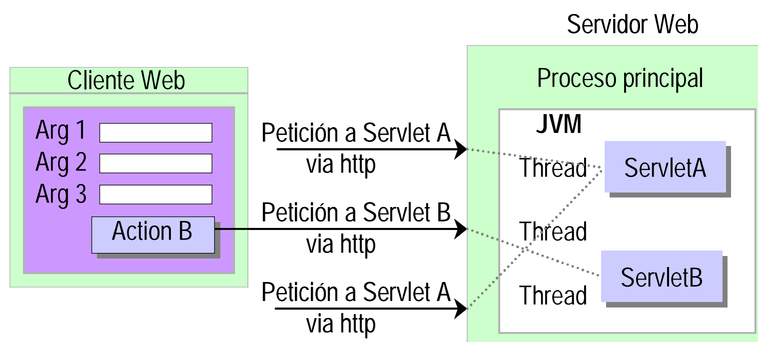
Interacción entre Aplicaciones: objetos distribuidos e invocación remota

En la anterior práctica se ha visto cómo extender la funcionalidad de un servidor web incorporando servlets que atienden peticiones web. Usan los métodos de http GET, con los argumentos codificados en el URL (urlencoded), y POST, donde los argumentos viajan en el cuerpo de la petición. En general, los servlets servirían para extender la funcionalidad de cualquier servidor añadiendo nuevos servicios o "comandos" que se invocan desde un formulario html.



Una ventaja importante de los servlets respecto a los CGI está en que se puede seleccionar la política de servicio (threads e instancias): una vez instanciado un servlet en la máquina virtual Java (JVM) puede servir varias peticiones usando un thread (procesos "ligeros") para cada una o por el contrario, un objeto (con un solo thread) para cada petición. Un servlet puede también guardar información que persiste durante la vida del objeto o conexiones a bases de datos. Además, la librería de servlets facilita y abstrae el paso de parámetros y su codificación, el seguimiento de sucesivas visitas de un mismo cliente (sesiones), la transformación de juegos de caracteres entre cliente y servidor, etc.

El modelo cliente web+formulario html / servidor web+extensiones (cgi o servlet) es adecuado para aplicaciones en que el usuario utiliza un cliente web y rellena un formulario. Se invoca una única operación en el servidor con un conjunto de parámetros textuales y sin estructura.

Sin embargo, si se desea comunicar procesos u objetos arbitrarios, intercambiar objetos o estructuras de datos (hay que "serializarlos" para que puedan pasar por la red), invocaciones bidireccionales, etc. el modelo anterior no lo permite. En todo caso, si se pretende interactuar con un objeto remoto de forma similar a como se interactúa con un objeto local, lo anterior no sirve.

En esta práctica se va a experimentar con objetos distribuidos: un mecanismo de invocación remota de los métodos de un objeto casi como si estuviera en la misma máquina. RMI (Invocación Remota de Métodos) permite localizar un objeto remoto, enviar y recibir por un canal de comunicación los argumentos y resultados de una invocación, y tratar los fallos como excepciones de Java. De todo esto se encarga la máquina virtual y el programador sólo tiene que introducir unos pocos cambios a su programa para distribuirlo. Por tanto, se trata de un mecanismo de comunicación entre aplicaciones que no se basan en el uso de un cliente web, ni en formularios html, ni en extensiones de un servidor web, ni en el uso de una conexión http.

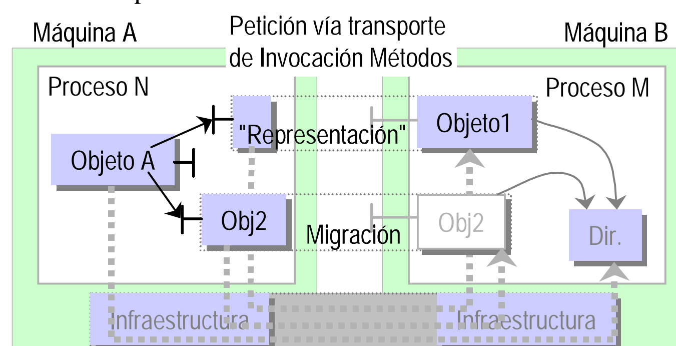


Fig. 1: Mecanismos de invocación remota entre objetos

En realidad, también pueden encontrarse objetos que corren dentro de un proceso cliente web (Applets, controles Active-X, u otras extensiones), que se comunican con un proceso servidor en otra máquina que además o en lugar de comunicarse con una extensión del servidor web, puede hacerlo con otro objeto servidor remoto.

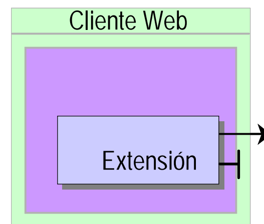


Fig. 2: Un entorno cliente (raramente servidor) puede ser un objeto que extiende un cliente web

Se trata por tanto de objetos o procesos que se comunican entre dos máquinas separadas por una red, que intercambian invocaciones de operaciones o métodos que llevan como argumentos de entrada y/o salida objetos, estructuras de datos y referencias a otros objetos.

Estos mecanismos suelen intentar ocultar la separación al programador (transparencia), para ello las invocaciones siempre se hacen a un objeto local (próximo al cliente del servicio). Este objeto puede ser:

- un representante del objeto distante (un objeto representante o stub o referente, que sólo se encarga de interactuar e intercambiar información con su representado)
- un objeto local que ha venido de lejos para atender una petición (ha migrado o se ha transferido por valor).

El intercambio de información y la invocación tiene lugar sobre un transporte de datos que puede ser TCP/IP y utiliza un formato de representación de datos (serialización y deserialización) especial y distinto a las codificaciones textuales de los métodos GET y POST de http.

La representación de información e invocación puede ser específica de un lenguaje como en Java-RMI (Invocación Remota de Métodos), o independiente de cada lenguaje como en CORBA-IIOP (Arquitectura común de tratamiento de peticiones entre objetos) lo que permite comunicar partes escritas en lenguajes distintos.

Invocación Remota de Métodos (RMI)

RMI forma parte de Java estándar y viene incluido en los JDK de Sun desde la versión 1.1. En esta práctica vamos a usar el soporte RMI que incluye la máquina virtual Java2 del JDK 1.2 o 1.3 de Sun

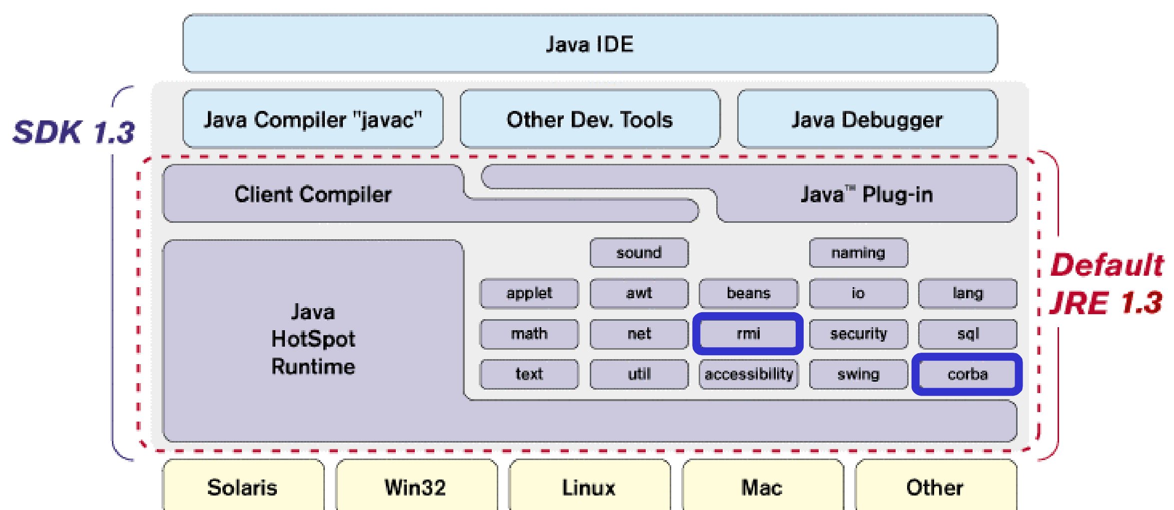


Fig. 4: La arquitectura de "Java 2 SDK, Standard Edition v. 1.3" que incluye rmi y corba.

Las ventajas principales de este modelo son las siguientes:

- Transparencia: unas mínimas modificaciones al código fuente permiten distribuir objetos entre varias máquinas. El entorno de ejecución Java (JRE) oculta y facilita la invocación, localización, activación, intercambio de datos y objetos por la red (serialización).
- Eficiencia: la invocación remota es más eficiente que una transferencia http por la forma en que se codifican los datos para la serialización, por la utilización de los canales de comunicación (usualmente conexiones TCP/IP), por la eficiencia y sencillez del stub servidor respecto a un servidor web.
- Seguridad: el gestor de seguridad "security manager" y otros componentes pueden controlar qué objetos y qué métodos puede invocar cada máquina cliente remota, así como restringir qué acciones puede llevar a cabo un objeto sobre la máquina en que corre y evitar que los fallos de un proceso pueda afectar a la seguridad e integridad de la máquina.
- Funcionalidad: se obtiene casi la misma funcionalidad que ofrece el lenguaje a la comunicación entre objetos remotos que entre objetos locales. Además, se puede elegir la forma en que un proceso servidor atiende las peticiones, se activa (se instancia), se elimina (por falta de referencias, como un objeto local).

El modelo de programación distribuída con RMI

En primer lugar se presenta el caso más sencillo de comunicación entre dos objetos que corren en distintas máquinas o al menos que corren en distintas máquinas virtuales Java (JVM) aunque sea el mismo computador. Es decir, se va a estudiar el mecanismo de invocación de métodos entre objetos remotos Java: Java-RMI.

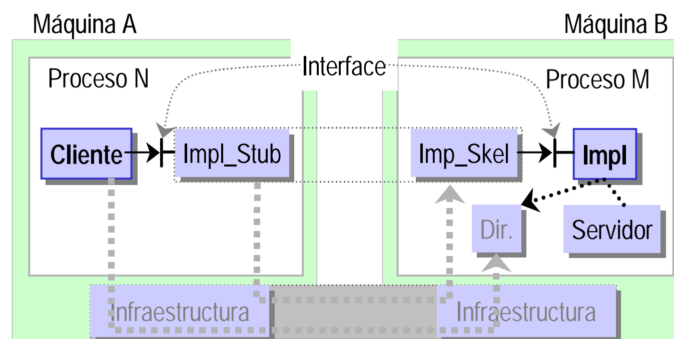


Fig. 3: Un objeto cliente invoca métodos de un objeto remoto.

Cuando un objeto quiere comunicarse con un objeto remoto:

1) ha de conocerlo o localizarlo previamente. Algún proceso en la máquina remota habrá instanciado o al menos registrado un objeto que ofrece cierto servicio. Para ello se usa un servicio de directorio: el registro rmi (rmiregistry).

```
X c = (X)Naming.lookup("rmi://servidor/ServicioX");
```

2) en realidad lo que obtiene es una referencia o representante local (stub) del objeto remoto. Cuando se invoca un método del stub, éste pasa la petición a un representante remoto (skeleton) que invoca localmente al objeto que implementa el método. Este objeto (Impl) o bien estará esperando peticiones o será activado automáticamente al llegar una invocación (similar a un servlet). El resultado de la invocación sigue el mismo camino de vuelta.

De cara al objeto cliente, todo es casi invisible: ha invocado un método de un objeto local (el stub), y ha obtenido una respuesta local. Lo que ha ocurrido entre stub cliente – stub servidor y objeto implementación es invisible y mágico para él.

Lo único diferente es la forma en que se ha instanciado el objeto (con `Naming.lookup()` en lugar de `new()`) y que una invocación a ese objeto puede generar alguna excepción nueva por los fallos que pueda introducir la red (como `java.rmi.RemoteException`).

RMI usa clases e interfaces del paquete: `java.rmi`. A continuación se especifica en java el interface de un programa ejemplo que se va a mostrar:

```
 public interface X extends java.rmi.Remote {
   public long incr(long a) throws java.rmi.RemoteException;
   public String mesg(String a) throws java.rmi.RemoteException;
 }
```

Cada operación puede fallar y generar la excepción `java.rmi.RemoteException`.

Una clase java ha de implementar el interface anterior. Para indicar que va a poder ser usada remotamente, hay que indicar que extiende `java.rmi.server.UnicastRemoteObject`. También se ha de declarar un constructor para que conste que puede generar la excepción `java.rmi.RemoteException`. Por lo demás, se trata de una clase normal:

```
public class Ximpl
extends java.rmi.server.UnicastRemoteObject implements X {
 // Constructor para declarar la excepción "RemoteException"
 public XImpl() throws java.rmi.RemoteException { super(); }

 public long incr(long a) throws java.rmi.RemoteException {
   return a+1;
 }
 public String mesg(String a) throws java.rmi.RemoteException {
   return "!Hola Amigo " + a + "!";
 }
}
```

El objeto cliente ha de instanciar la clase remota utilizando el servicio de nombres (`Naming.lookup`) y es conveniente tratar las excepciones que puedan producirse, al menos `RemoteException`:

```
 import java.rmi.Naming;
 import java.rmi.RemoteException;
 import java.net.MalformedURLException;
 import java.rmi.NotBoundException;
 [...]
  try {
   X c = (X)Naming.lookup("rmi://servidor/ServicioX");
  }
  catch (RemoteException re) { }          // El servidor puede fallar
  catch (MalformedURLException nbe) { } // Puede fallar la ref
  catch (NotBoundException nbe) { }    // Puede fallar la búsqueda
```

A continuación puede verse el código de un programa en la máquina servidor que instancia la clase `XImpl` y la publica en el servicio de nombres (`Naming.rebind` la anuncia, y sustituye una anterior si hubiere).

```
import java.rmi.Naming;

public class XServidor {
  public XServidor() { // El método constructor
   try {
    X c = new XImpl();
    Naming.rebind("rmi://localhost:1099/ServicioX", c);
   } catch (Exception e) {
    System.out.println("Problema: " + e);
   }
  }

  public static void main(String args[]) {
   new XServidor();
  }
}
```

En resumen, los pasos a seguir para diseñar una aplicación distribuida con RMI son:

1. Definir las funciones de la clase remota como un interface Java.
2. Escribir la clase implementación.
    a. Declarar que implementa al menos un interface remoto.
    b. Definir el constructor del objeto remoto.
    c. Proporcionar implementaciones para los métodos declarados en el interface.
3. Escribir la clase servidor que instancia y anuncia el objeto que implementa el interface.
    a. Crear e instalar un "security manager" (no se hace en estos ejemplos).
    b. Crear una o más instancias del objeto remoto.
    c. Registrar al menos un objeto remoto en el registro RMI.
4. Escribir un programa cliente que usa el servicio remoto.

Una vez comprobado que la JVM funciona (se puede ejecutar `javac y rmic`)`se puede` copiar el programa anterior y probar que funciona correctamente. Puede encontrarse en el web del laboratorio de AAD (http://www.ac.upc.es/docencia/FIB/AAD/lab)

Pasos a seguir:

1) Crear un directorio de trabajo. Traer y descompactar el paquete zip con el programa de ejemplo.

2) Compilar el código Java: `javac *.java`
   Generar el stub y skel del objeto implementación: `rmic Ximpl`

3) Tras ello, se ha de encontrar lo siguiente:

Origen	*Resultado*
`X.java`	`X.class`
`XImpl.java`	`XImpl.class`
`Xservidor.java`	`XServidor.class`
`Xcliente.java`	`XCliente.class`
rmic `XImpl`	`XImpl_Stub.class`
	`XImpl_Skel.class`

4) Arrancar el registro de rmi:
   `rmiregistry &` (unix) / `start rmiregistry` (windows).

5) Ejecutar el programa servidor, que instancia y anuncia un objeto de la clase XImpl:
   `java XServidor &` (unix) / `start java XServidor` (windows)

6) Ejecutar el programa cliente, que localiza una instancia remota, e invoca sus métodos (en realidad los del objeto stub local):
   `java XCliente` (unix) / `java XCliente` (windows)

Debería responder con el mensaje que se haya programado en la clase XImpl.

Persistencia y activación de los objetos

Un objeto accesible a distancia lo instancia y anuncia en el servicio de nombres un programa que arranca en la máquina servidor. También podría sólo anunciarlo y activarse automáticamente si el objeto Impl se declara como:

```
public class Ximpl
extends java.rmi.server.Activatable implements X {
```

El objeto servidor responde a peticiones y vive hasta que nadie lo referencie por lo que es “reciclado” por el garbage collector (distribuido). El ciclo de vida es:

1. Crear e inicializar el objeto implementación,
2. Anunciar el objeto en el registro (rmiregistry),
3. Responder a invocaciones de métodos de otros objetos locales o remotos,
4. Quitar el anuncio y morir o esperar a ser reciclado (garbage collection).

Sigue RMI un ciclo de vida similar a los servlet.

Ejercicio 1:

Modificar `Ximpl.java` del ejemplo anterior para probar el efecto de invocar varias veces el método desde el programa cliente. Se va a introducir un contador de visitas, y cada vez que se invoque cualquier método de un objeto de clase `Ximpl`, se incrementará un contador. El método `mesg` ha de informar además del número de visitas:

```
>java Xcliente Luis
5
!Hola Amigo Luis! (visitas 2)
```

Ejercicio 2:

Escribir un programa distribuido para una tienda de muebles. Se trata de modificar el ejemplo anterior para construir un programa cliente y un programa servidor que puedan correr en distintas máquinas físicas o al menos en distintas máquinas virtuales.

Pasos a seguir:

1) Definir el interface java que implemente la clase `tiendaDeMuebles`.
2) Escribir la implementación de la clase `tiendaDeMuebles` (se puede partir del código de la práctica anterior)
3) Modificar el programa servidor de ejemplo para que instancie y anuncie un objeto de la clase `tiendaDeMuebles`.
4) Modificar el programa cliente de ejemplo para que localice e invoque métodos de un objeto de la clase `tiendaDeMuebles`. (las órdenes pueden pasarse como prefieras: línea comandos o menú texto).

Referencias

1. Java Remote Method Invocation (RMI); http://java.sun.com/products/jdk/rmi/

2. jGuru: Remote Method Invocation: Introduction; http://developer.java.sun.com/developer/onlineTraining/rmi/

3. RMI y CORBA (RMI sobre IIOP); http://developer.java.sun.com/developer/earlyAccess/idlc/