Programación Distribuida con RMI

1. Introducción

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.

2. 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:

- El paso de mensajes 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 la colaboración falla.
- 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.

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.

El **paradigma de objetos distribuidos** es un paradigma que proporciona mayor abstracción que el modelo de paso de mensajes. 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**.

Aunque ya 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.

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).

Un proceso que utiliza objetos distribuidos se dice que es un **proceso cliente** de eso 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.

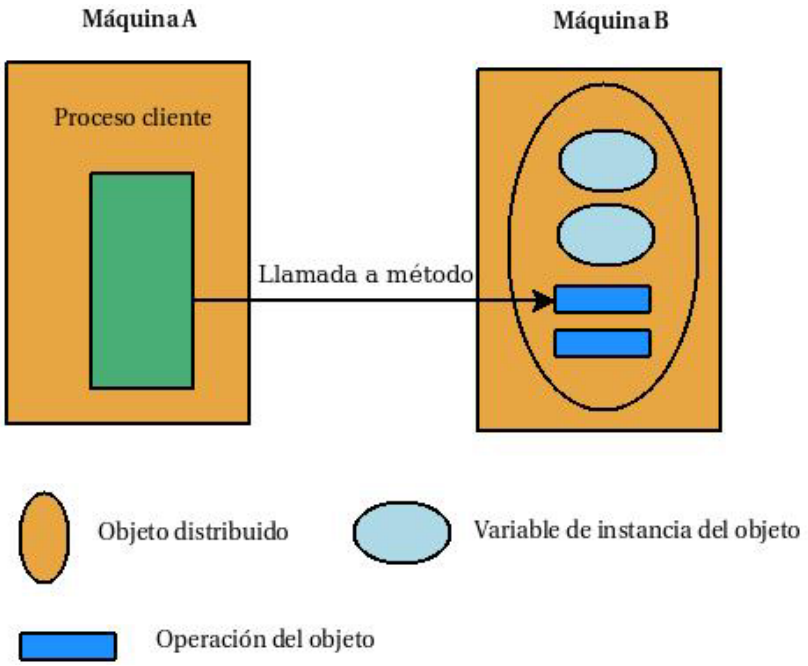

Figura 1. El paradigma de objetos distribuidos.

3. 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.

Al objeto distribuido proporcionando 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.

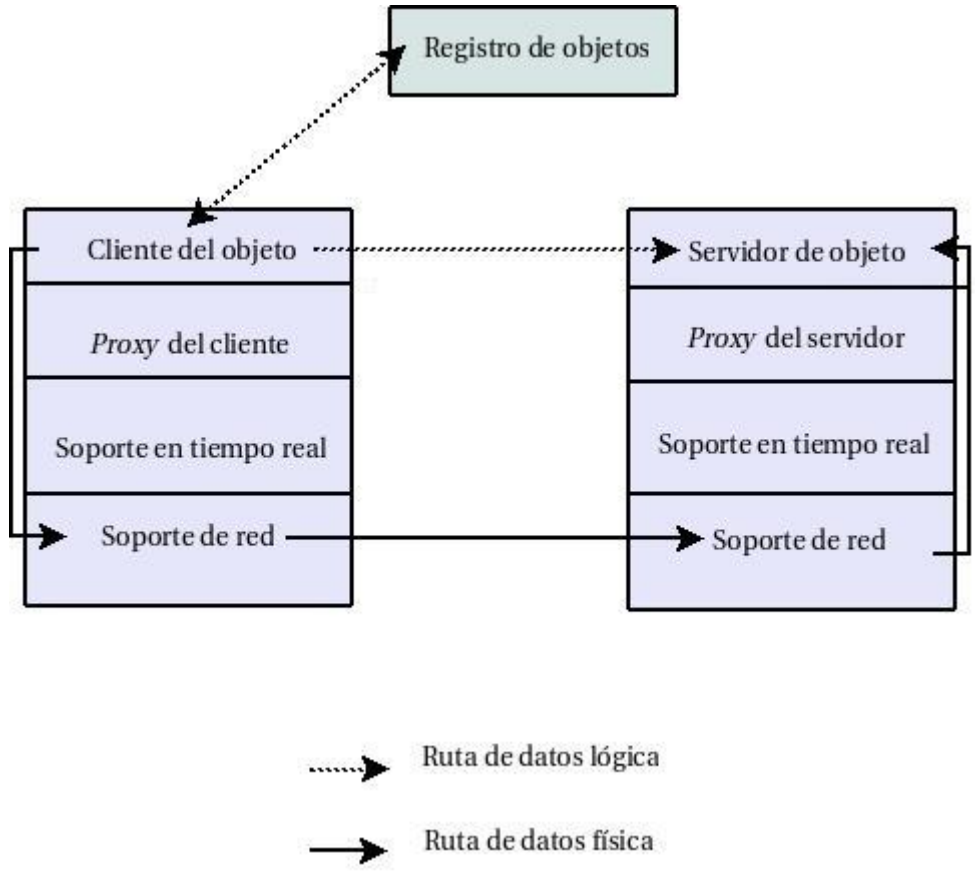

Figura 2. Un sistema de objetos distribuidos típico.

Realmente, un componente software se encarga de gestionar esta llamada. Este componente se denomina ***proxy* de cliente** 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.

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 ***proxy* de servidor**. El *proxy* de servidor 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.

4. 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 encuentran:

- 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 Objetct Access Protocol).

De todas estas herramientas, la más sencilla es Java RMI.

5. 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 e 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. 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.

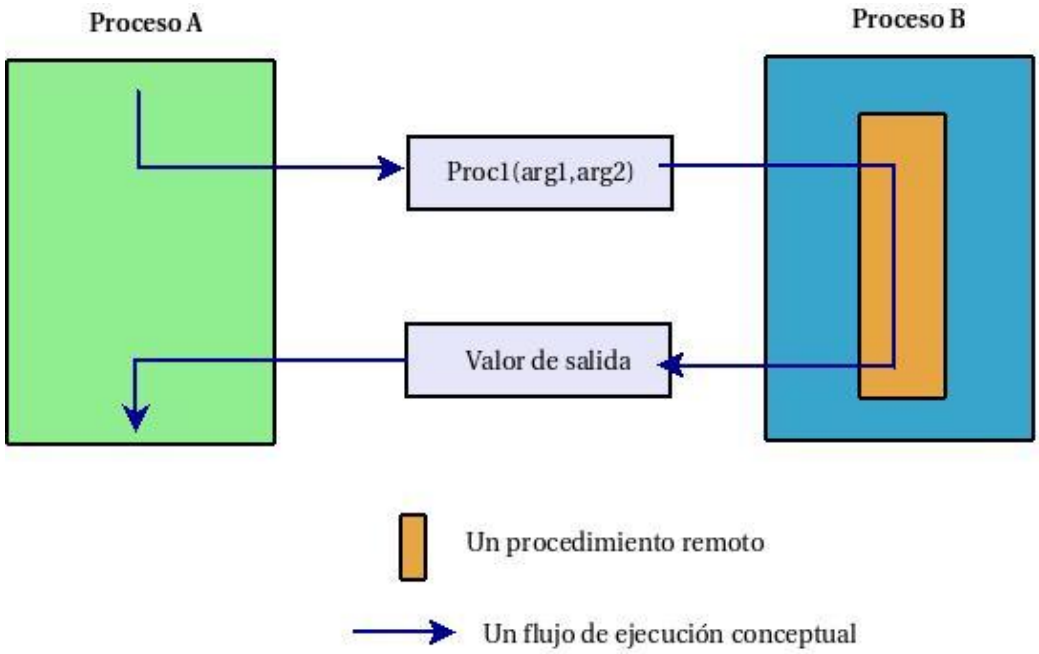

Figura 3. El paradigma de llamada a procedimiento remotos.

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***.

En el otro extremo, un *proxy* recibe el mensaje y lo transforma en una llamada a procedimiento local al procedimiento remoto.

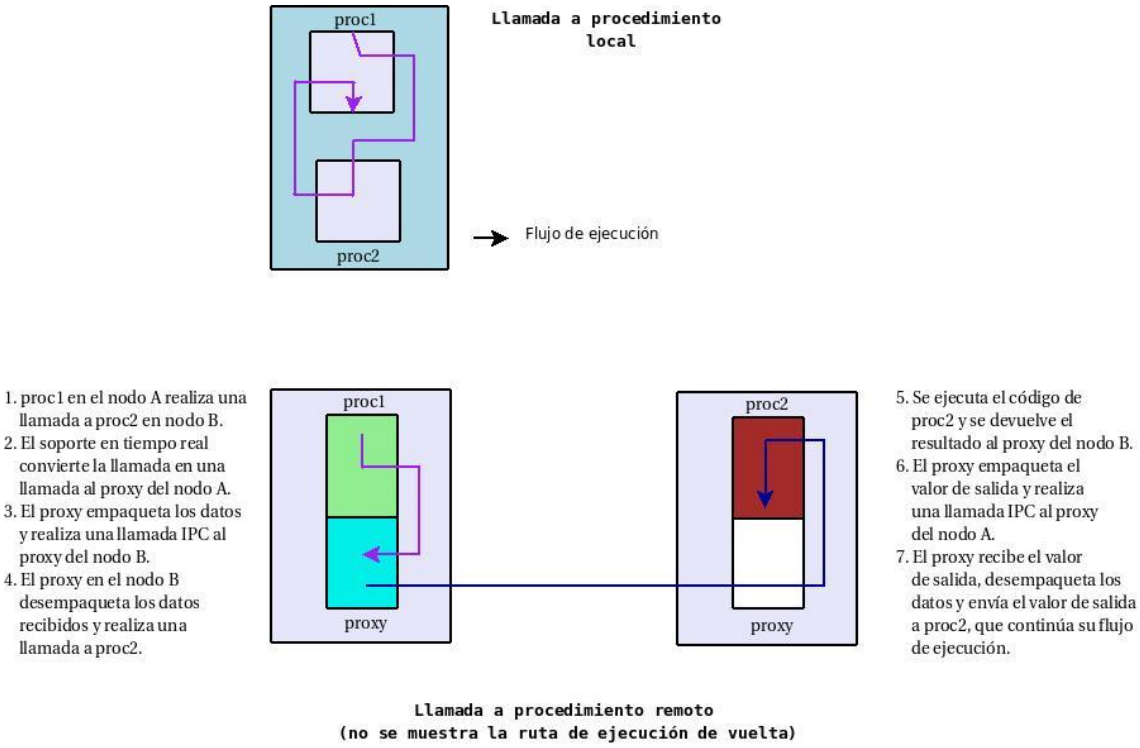

Figura 4. Llamada a procedimiento local frente a llamada a procedimiento remoto.

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 Remote Procedure Call***, es una evolución del API del RPC que desarrolló originalmente Sun Microsystems. La otra API popular es ***Open Group Distributed Computing Enviroment*** (DCE). Ambas interfaces proporcionan una herramienta, **rpcgen**, para transformar las llamadas a procedimientos remotos en llamadas a procedimientos locales al resguardo.

6. 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.

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.

7. La arquitectura 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 subyacente.

7.1 Parte cliente de la arquitectura

1. La **capa resguardo o *stub***. La invocación de un método remoto por parte de un proceso cliente es dirigida 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, e invoca las operaciones entre procesos de 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.

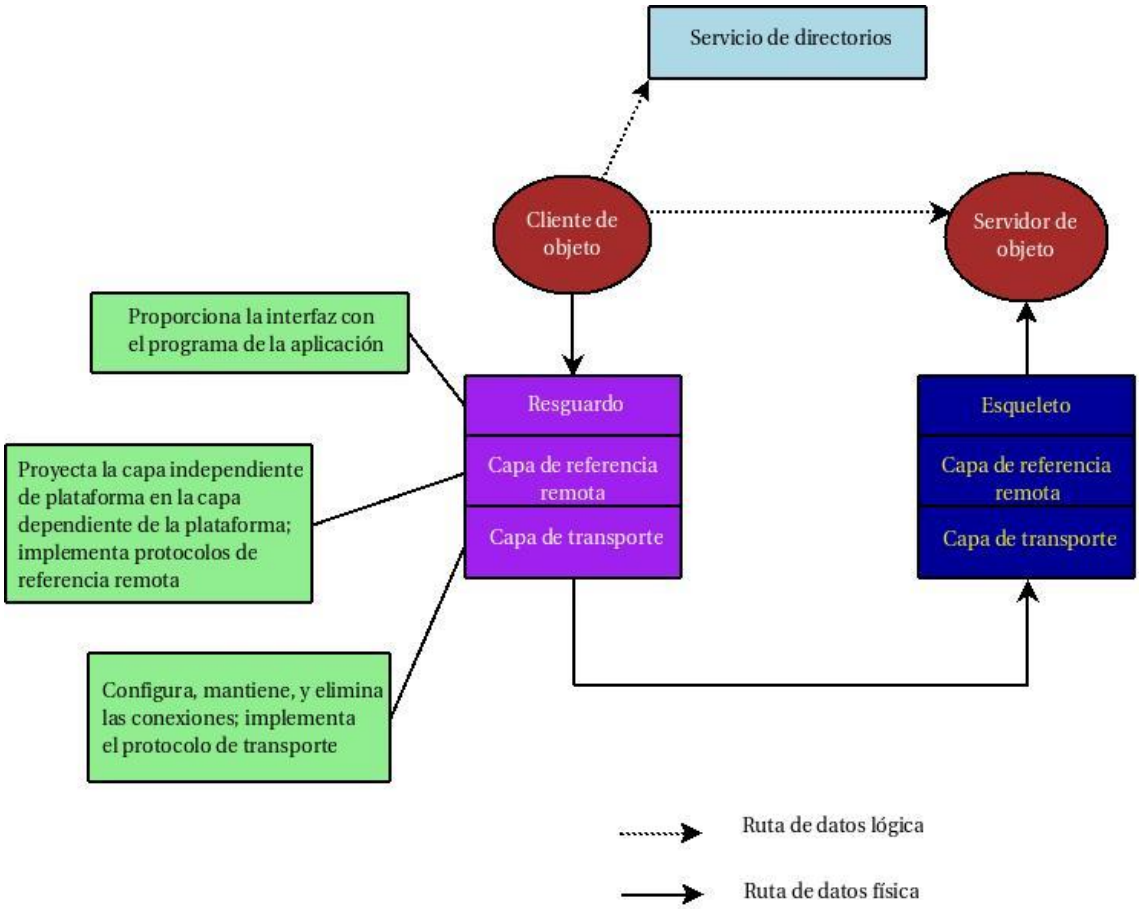

Figura 5. La arquitectura de Java RMI.

7. 2 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.

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.

6. 7. 3 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**, que es más general que el registro RMI. Este registro 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.

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 resguardos y esqueletos en tiempo de ejecución, dando lugar a la transmisión de datos a través de la red.

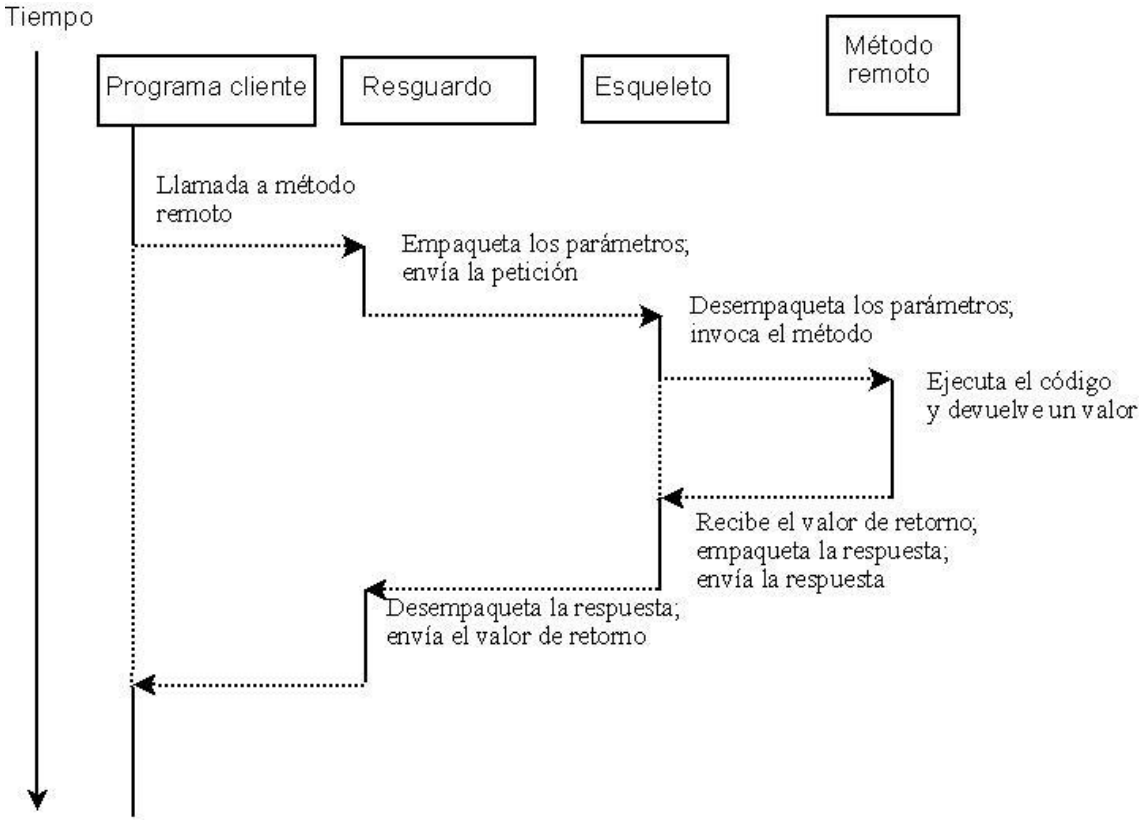

Figura 6. Interacciones entre el resguardo RMI y el esqueleto RMI.

8. API de Java RMI

En esta sección se introduce un subconjunto del API de Java RMI.

8. 1. 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.

Figura 7. Un ejemplo de interfaz remota Java.

```
// fichero: InterfazEjemplo.java
// implementada por una clase servidor Java RMI

import java.rmi.*;

public interface InterfazEjemplo extendí Remote {
  // cabecera del primer método remoto
  public String metodoEj1( )
    throws java.rmi.RemoteException;
  // cabecera del Segundo método remoto
  public int metodoEj2(float parámetro)
     throws java.rmi.RemoteException;
  // cabeceras de otros métodos remotos
}  // fin interfaz
```

En este ejemplo, se declara una interfaz denominada *InterfazEjemplo*. La interfaz extiende o hereda la clave Java *remote*, convirtiéndose de este modo en una interfaz remota.

Dentro del bloque que se encuentra entre las llaves se encuentran las declaraciones de los **métodos remotos**, que se llaman *metodoEj1* y *metodoEj2*, respectivamente.

Cada declaración de un método debe especificar la excepción *java.rmi.RemoteException* en la sentencia *throws*. 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.

8. 2. 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 implementar cada uno de los métodos remotos especificados en la interfaz, y registrar en un servicio de directorios un objeto que contiene la implementación.

8. 3. 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.

Figura 8. Sintaxis de un ejemplo de implementación de interfaz remota.

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

/**
Esta clase implementa la interfaz remota InterfazEjemplo.
*/

public class ImplEjemplo extends UnicastRemoteObject
   implements InterfazEjemplo {

   public ImplEjemplo( ) throws RemoteException {
     super( );
   }

   public metodoEj1() throws RemoteException {
     // código del método
   }

   public metodoEj2() throws RemoteException {
     // código del método
   }
}  // fin class
```

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

La cabecera de la clase debe especificar que es una subclase de la clase Java *UnicastRemoteObject*, y que implementa una interfaz remota específica, llamada *InterfazEjemplo* en la plantilla.

Se debe definir un constructor de la clase. 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.

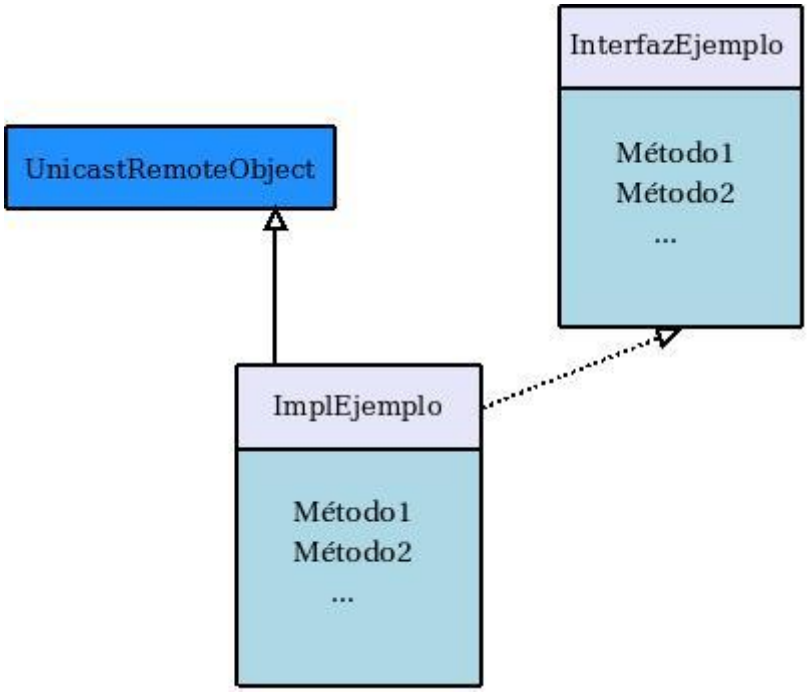

Figura 9. El diagrama de clases UML para ImplEjemplo.

8. 4. Generación del resguardo y del esqueleto

En RMI, un objeto distribuido requiere un *proxy* por cada uno de los servidores y clientes 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>
```

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 implementación.

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. 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.

8. 5. El servidor de objetos

La clase del servidor de objeto instancia y exporta un objeto de la implementación de la interfaz remota.

Figura 10. Sintaxis de un ejemplo de un servidor de objeto.

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.Registry;
import java.rmi.registry.LocatRegistry;
import java.net.*;
import java.io.*;

/**
*  Esta clase representa el servidor de un objeto
*  distribuido de la clase ImplEjemplo, que implementa la
*  interfaz remota InterfazEjemplo.
*/

public class ServidorEjemplo {
   public static void main(String args[]) {
      String numPuertoRMI, URLRegistro;
      try{
         // código que permite obtener el valor del número de puerto
         ImplEjemplo objExportado = new ImplEjemplo();
         arrancarRegistro(numPuertoRMI);
         // registrar el objeto bajo el nombre "ejemplo"
         URLRegistro = "rmi://localhost:" + numPuerto + "/ejemplo";
         Naming.rebind(URLRegistro, objExportado);
         System.out.println("Servidor ejemplo preparado.");
      }  // fin try
      catch (Exception excr) {
         // fin catch
   } // fin main

// Este método arranca un registro RMI en la máquina
// local, si no existe en el número de puerto especificado

   private static void arrancarRegistro (int numPuertoRMI)
           throws RemoteException {
       try {
           Registry registro = LocateRegistry(numPuertoRMI);
           registo.list();
           // El método anterior lanza una excepción
           // si el registro no existe.
       }  //fin try
       catch (RemoteException exc) {
           // No existe un registro válido en este puerto.
           System.out.println(
              "El registro RMI no se puede localizar en el puerto:
              + RMIPortNum);
           Registry registro =LocateRegistry.createRegistry(numPuertoRMI);
           System.out.println(
              "Registro RMI creado en el puerto " + RMIPortNum);
       } // fin catch
   } // fin arrancarRegistro


}  // fin class
```

Creación de un objeto de la implementación de la interfaz remota. 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.

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.

En un sistema de producción, debería existir un servidor *rmiregistry*, ejecutando de forma continua, presumiblemente en el puerto por defecto 1099.

En la plantilla del servidor del objeto, se implementa el método estático *arrancar Registro()*, que arranca un servidor de registro RMI si no está actualmente en ejecución, en un número de puerto especificado por el usuario:

```
arrancarRegistro(numPuertoRMI);
```

En un sistema 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 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* sobrescribe cualquier referencia en el registro asociada al nombre de la referencia. Si no se desea sobrescribir, 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.

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.

8. 6. 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.

Figura 61. Plantilla para un cliente de objeto.

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.Registry;
import java.rmi.registry.LocatRegistry;

/**
*  Esta clase representa el cliente de un objeto
*  distribuido de la clase ImplEjemplo, que implementa la
*  interfaz remota InterfazEjemplo
*/

public class ClienteEjemplo {
   public static void main(String args[ ]) {
      try {
        int puertoRMI;
        String nombreNodo;
        String numPuerto;
        // Código que permite obtener el nombre del nodo y
        // el número de puerto del registro

        // Búsqueda del objeto remoto y cast de la
        // referencia con la correspondiente clase
        // de la interfaz remota – reemplazar <<localhost>> por el
        // nombre del nodo del objeto remoto.

        String URLRegistro =
          "rmi://localhost:" + numPuerto + "/ejemplo";
        InterfazEjemplo h =
          (InterfazEjemplo) Naming.lookup(URLRegistro);
        // invocar el o los métodos remotos
        String mensaje = h.metodoEj1();
        System.out.prinln(mensaje);
        // el método metodoEj2 puede invocarse del mismo modo
       }// fin try
       catch (Exception exc) {
          exc.printStackTrace();
```

```
            } // fin match
          } // fin main
         // Posible definición de otros métodos de la clase
    }  // fin class
```

Las **sentencias de importación**. Las sentencias de importación 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.

```
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.

```
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 método locales.

9. Una aplicación RMI de ejemplo

El servidor exporta un objeto que contiene un único método, denominado *decirHola*.

Figura 12. HolaMundoInt.java.

```
// Un ejemplo sencillo de interfaz RMI
import java.rmi.*;

/**
*  Interfaz remota
*/

public interface HolaMundoInt extends Remote {
/**
*  Este método remoto devuelve un mensaje.
*  @para name – una cadena de caracteres con un nombre.
*  @return – una cadena de caracteres.
*/

   public String decirHola(String nombre)
      throws java.rmi.RemoteException;
   }
```

Figura 13. HolaMundoImpl.java.

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

/**
*  Esta clase implementa la interfaz remota
*  HolaMundoInt
*/

public class HolaMundoImpl extends UnicastRemoteObject
implements HolaMundoInt {

   public HolaMundoImpl() throws RemoteException {
      super();
   }

   public String decirHola(String nombre)
        throws RemoteException {
      return "Hola mundo" + nombre;
   }
} // fin class
```

Figura 14. HolaMundoServidor.java.

```
import java.rmi.*;
import java.rmi.server*;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.net.*;
import java.io.*;

/**
*  Esta clase representa el servidor de un objeto
*  de la clase HolaMundo, que implementa la
*  interfaz remota HolaMundoInterfaz.
*/

public class HolaMundoServidor {
      public static void main (String args[]) {
         IntputStreamReader ent = new InputStreamReader(System.in));
         BufferReader buf = new BufferedReader(ent);
         String numPuerto, URLRegistro;
         try {
            System.out.println("Introducir el número de puerto del
            registro RMI:");
            numPuerto = buf.readLine().trim();
            int numPuertoRMI = Integer.parseInt(numPuerto);
            arrancarRegistro(numPuertoRMI);
            HolaMundoImpl objExportado = new HolaMundoImpl();
            URLRegistro = "rmi://localhost:" + numPuerto + "/holaMundo";
            Naming.rebind(URLRegistro, objExportado);
   /**/     System.out.prinln
   /**/     ("Servidor registrado. El registro contiene actualmente:");
   /**/     // lista de los nombres que se encuentran en el registro
            // actualmente
   /**/     listaRegistro(URLRegistro);
            System.out.println("Servidor HolaMundo preparado.");
          } // fin try
          catch (Exception Exc.) {
             System.out.println("Excepción en HolaMundoServidor.main:" + excr);
          } // fin catch
      } // fin main
         // Este método arranca un registro RMI en la máquina
         // local, si no existe en el número de puerto especificado.
         private static void arrancarRegistro(int numPuertoRMI)
              throws RemoteException {
            try {
              Registry registro = LocateRegistry.getRegistry(numPuertoRMI);
```

```
                registro.list(); //Esta llamada lanza
                // una excepción si el registro no existe
              }
              catch (RemoteException e) {
                // Registro no válido en este puerto
    /**/        System.out.println
    /**/        ("El registro RMI no se puede localizar en el puerto"
    /**/        + numPuertoRMI);
                Registry registro =
                  LocateRegistry.createRegistry(numPuertoRMI);
    /**/        System.out.println(
    /**/           "Registro RMI creado en el Puerto" + numPuertoRMI);
              } // fin catch
      } // fin arrancarRegistro

      // Este método lista los nombres registrados con un objetos Registry
      private static void listaRegistro(String URLRegistro)
           throws RemoteException, MalformedURLException {
         System.out.println("Registro " + URLRegistro * " contiene: ");
         String [] nombres = Naming.list(URLRegistro);
         for (int i=0, i<nombres.length; i++)
           System.out.println(names[i]);
      } // fin listaRegistro
} // fin class
```

Figura 15. HolaMundoCliente.java.

```
import java.io.*;
import java.rmi.*;

/**
*  Esta clase representa el cliente de un objeto
*  distribuido de clase HolaMundo, que implementa la
*  interfaz remota HolaMundoInterfaz
*/

public class HolaMundoCliente {

   public static void main(String args[]) {
     try {
       int numPuertoRMI;
       String nombreNodo;
       InputStreamReader ent = new InputStreamReader(System.in);
       BufferedReader buf = new BufferedReader(ent);
       System.out.prinln("Introducir el nombre del nodo del
       Registro RMI:");
       nombreNodo = buf.readLine();
       System.out.println("Introducir el número de puerto del
       Registro RMI:");
       String numPuerto = buf.readLine();
       numPuertoRMI = Integer.parseInt(numPuerto);
     String URLRegistro =
       "rmi://" + nombreNodo + ":" + numPuerto + "/holaMundo";
     // Búsqueda del objeto remoto y cast del objeto de la interfaz
       HolaMundoInterfaz h =
          (HolaMundoInterfaz)Naming.lookup(URLRegistro);
       System.out.prinln("Búsqueda completa");
       // Invocar el método remoto
       String mensaje = h.decirHola("Pato Donald");
       System.out.println("HolaMundoCliente: " + mensaje);
     } // fin try
     catch (Exception e) {
       System.out.println("Excepción en HolaMundoCliente: " + e);
     } // fin match
   } // fin main
} // fin class
```

Una vez comprendida la estructura básica del ejemplo de aplicación RMI presentado, se 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.

10. Pasos para construir una aplicación RMI

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.

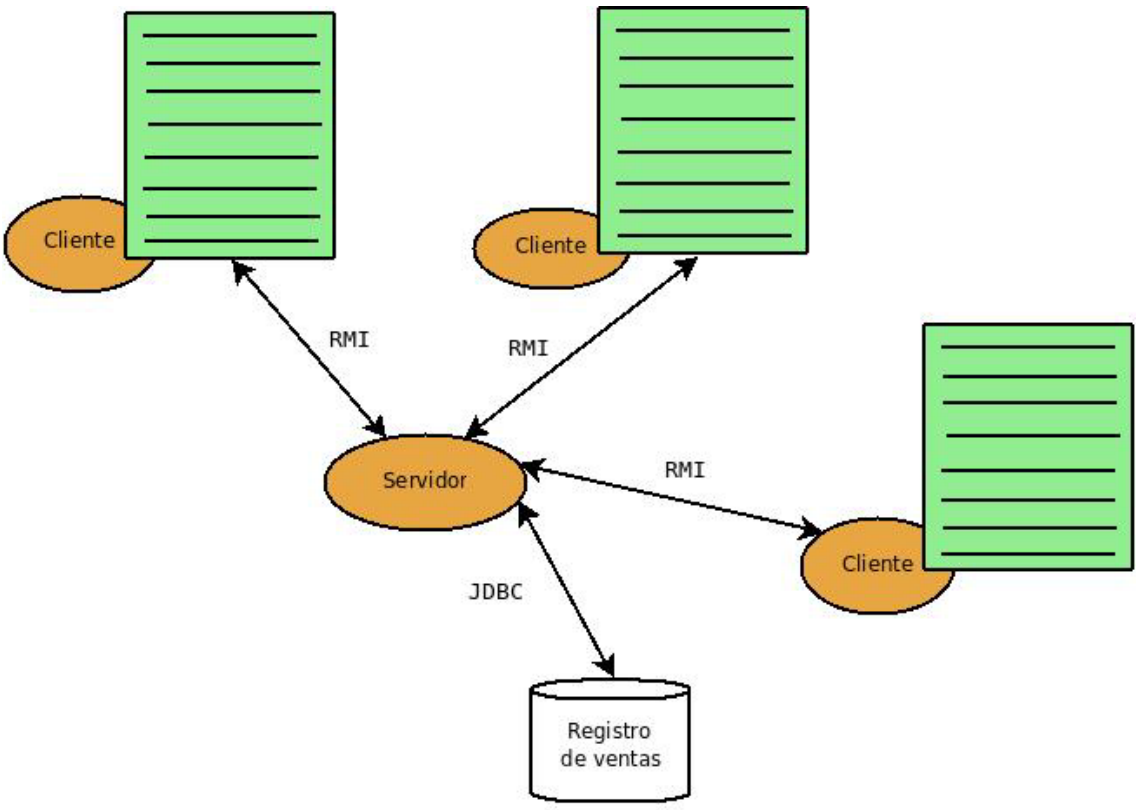

Figura 16. Un ejemplo de aplicación RMI.

10. 1. 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
   ```

   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 revisarlo hasta que no exista ningún error de sintaxis.
6. Activar el servidor de objeto.

   ```
   java ServidorEjemplo
   ```

10. 2. 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.
4. Desarrollar el programa cliente *ClienteEjemplo.java*. Compilarlo y revisarlo hasta que no exista ningún error de sintaxis.
5. Activar el cliente.

   ```
   Java ClienteEjemplo
   ```

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 ficheros class de la interfaz, del servidor de objeto, de la implementación de la interfaz y del esqueleto para el objeto remoto.

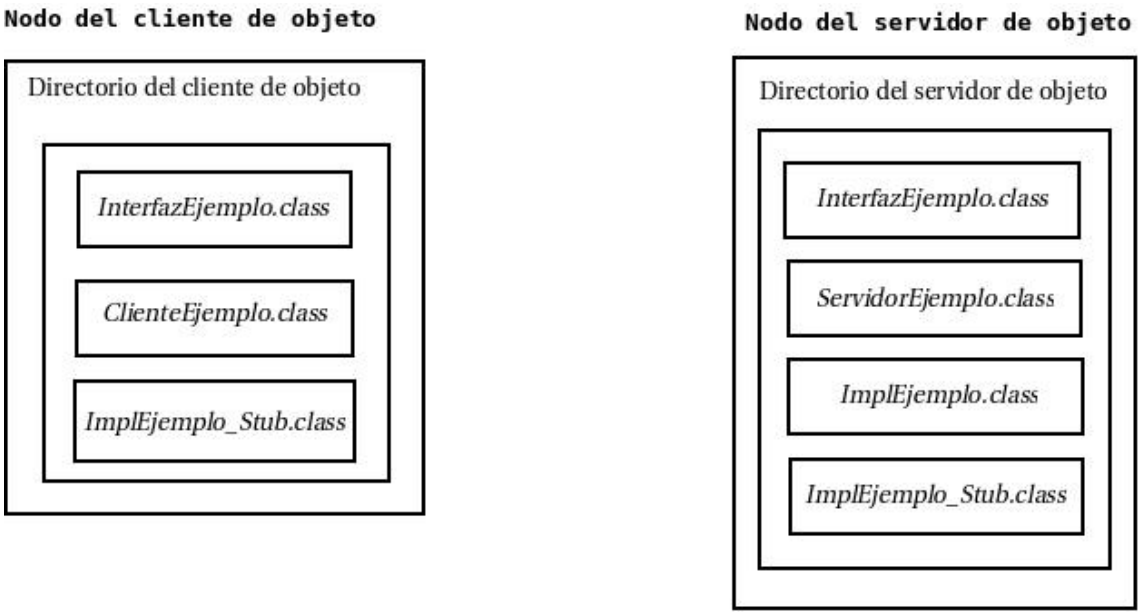

Figura 17. Colocación de los ficheros en una aplicación RMI.

11. 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 retorno.
2. Añadir una declaración cada vez a la interfaz.
3. Rellenar la definición de cada método remoto uno a uno.
4. Después de todos los métodos remotos se han probado detalladamente, crear la aplicación cliente utilizando una técnica incremental.
5. Distribuir los programas en máquinas separadas. Probarlos y depurarlos.

12. 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* para construir una aplicación de red rápidamente.

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.
- 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.

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 otro.