3. Objetos Distribuidos

3.1 Paso de Mensajes frente a Objetos Distribuidos

3.1.1 Paradigma de paso de mensajes

El paradigma de paso de mensajes es un modelo natural a usar en la computación distribuida ya que simula la comunicación entre personas. Es apropiado para los servicios de red porque estos procesos interactúan mandando y recibiendo mensajes. Sin embargo, no proporciona la abstracción necesaria para aplicaciones de red complejas por varios motivos:

- **Los procesos están fuertemente acoplados:** los procesos deben comunicarse directamente entre ellos y si la comunicación se pierde la colaboración falla.
- **Está orientado a datos:** los mensajes contienen datos en un formato determinado y se interpretan como peticiones o respuestas. Que sea orientado a datos está bien para servidores de red o aplicaciones de red sencillas pero si es una aplicación compleja que requiera un gran número de peticiones y respuestas sería una tarea imposible tratar con la interpretación de los mensajes.

3.1.2 Paradigma de objetos distribuidos

El paradigma de objetos distribuidos proporciona una mayor abstracción que el anterior. Está basado en objetos existentes en un sistema distribuido (en diferentes máquinas). Los objetos son entidades significativas para la aplicación y contienen:

- El estado o datos de la entidad (en java son variables)
- Las operaciones de la entidad que modifican el estado de la entidad (en Java son los métodos).

Podemos clasificar los objetos en:

- **Objetos locales:** objetos cuyos métodos solo se pueden invocar desde un proceso **local** (proceso en la misma máquina en donde se encuentra el objeto)
- **Objetos distribuidos:** objetos cuyos métodos pueden invocarse por un proceso **remoto** (proceso en distinta máquina de la que se encuentra el objeto).

El paradigma de objetos distribuidos es orientado a acciones (se hace hincapié en la invocación de métodos mientras que los datos tienen un papel secundario). Además es más natural para el desarrollo de software orientado a objetos. Sin embargo, el paradigma de paso de mensajes es orientado a datos. Los recursos de red son objetos

distribuidos y si un proceso quiere solicitar un servicio de uno de estos recursos debe invocar a uno de sus métodos pasándole parámetros (datos). El método se ejecuta en la máquina remota y la respuesta se le envía al proceso solicitante.

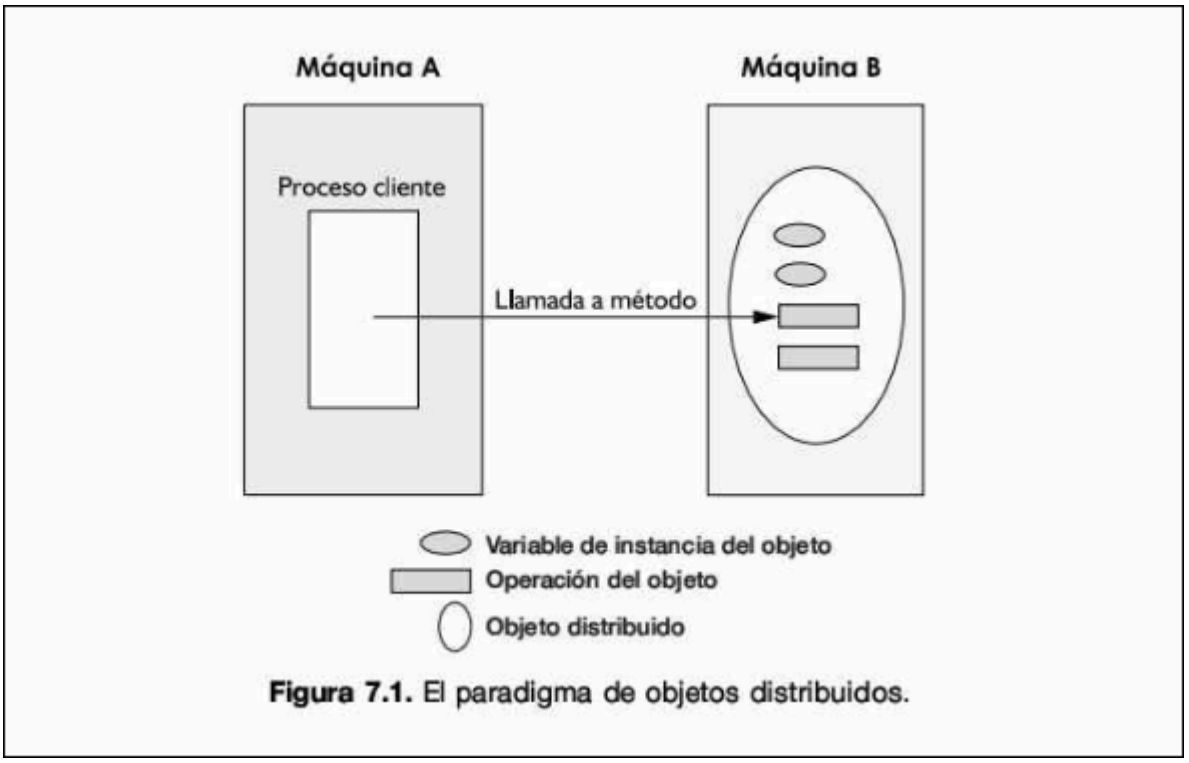

Figura 7.1. El paradigma de objetos distribuidos.

Un proceso que se ejecuta en la máquina A realiza una llamada a un método (con los datos necesarios) de un objeto distribuido de la máquina B. Esta llamada invoca una acción realiza por el método en la máquina A y una salida que se pasa de A a B.

Arquitectura

El objeto distribuido exportado por un proceso se llama **servidor de objeto**. Es necesario que exista un **registro de objetos** para registrar los objetos distribuidos. Para acceder a un objeto distribuido el **cliente de objeto** busca en el registro la referencia al objeto y la utiliza para realizar llamadas a métodos del objeto remoto (**métodos remotos**). Un componente software se encarga de gestionar la llamada, este componente se llama **proxy de cliente**. Se encarga de interactuar con el software en la máquina cliente para proporcionar **soporte en tiempo de ejecución** para el sistema de objetos distribuidos. Y este soporte se encarga de la comunicación, entre procesos, necesaria para la llamada a la máquina remota y el comportamiento de los argumentos.

En la parte del servidor el soporte en tiempo de ejecución gestiona la recepción de mensajes y envía la llamada al componente software llamado **proxy de servidor**. Este componente invoca la llamada al método local en el objeto distribuido (pasándole los datos desempaquetados como argumentos). El resultado de la ejecución del método se empaqueta y se envía desde el proxy del servidor al proxy del cliente a través del soporte en tiempo de ejecución y el soporte de red de ambas partes.

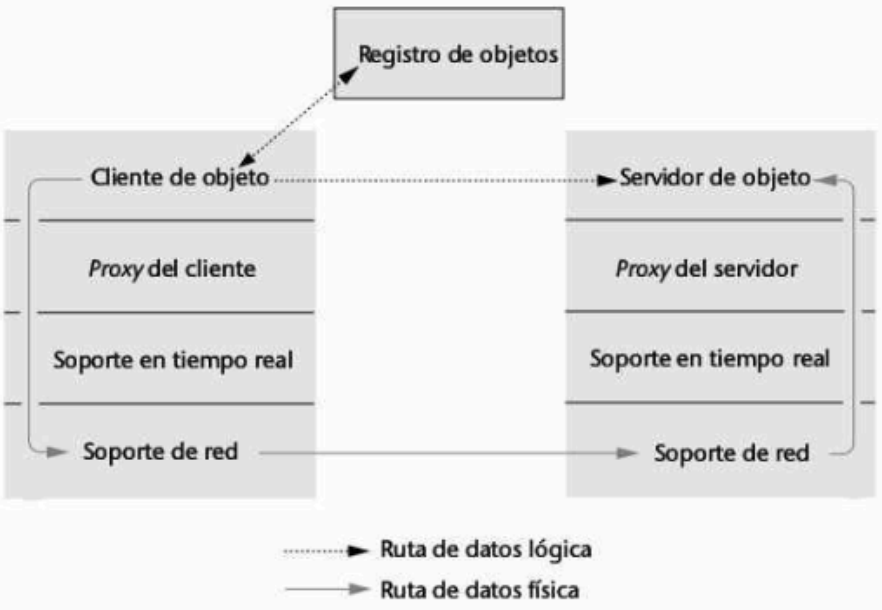

Figura 7.2. Un sistema de objetos distribuidos típico.

Ejemplo sencillo

- El cliente quiere sumar dos números usando un objeto remoto.
- Busca el objeto "Calculadora" en el registro de objetos.
- Obtiene la referencia y llama al método `sumar(2,3)` en su proxy de cliente.
- El proxy de cliente manda esa petición al servidor por la red.
- El soporte en tiempo de ejecución en ambas máquinas gestiona la comunicación.
- El proxy de servidor recibe la petición y llama al método real `sumar(2,3)` en el objeto calculadora del servidor.
- El resultado (5) se empaqueta y se envía de vuelta al cliente.
- El cliente recibe el resultado como si fuera una llamada local.

Sistemas

Existe un gran número de herramientas basadas en este paradigma:

- **Java RMI** (Remote Method Invocation), la más sencilla de todas
- Sistemas basados en **CORBA** (Common Object Request Broker Architecture)
- **Modelo de componentes distribuidos o DCOM**
- Herramientas y API para el protocolo **SOAP**

3.2 Llamadas a procedimientos remotos vs invocación de métodos remotos

3.3.1 Llamada a procedimiento remoto

En este paradigma un proceso realiza una llamada a un procedimiento de otro proceso (en otra máquina). Como siempre, los datos se pasan a través de argumentos. Cuando

un proceso recibe una llamada, se ejecuta la acción codificada en ese procedimiento. Después se le informa al proceso que invocó la llamada de que invocó la llamada de que ha finalizado y si existe un valor de retorno se le envía.

El modelo es orientado a procedimiento.

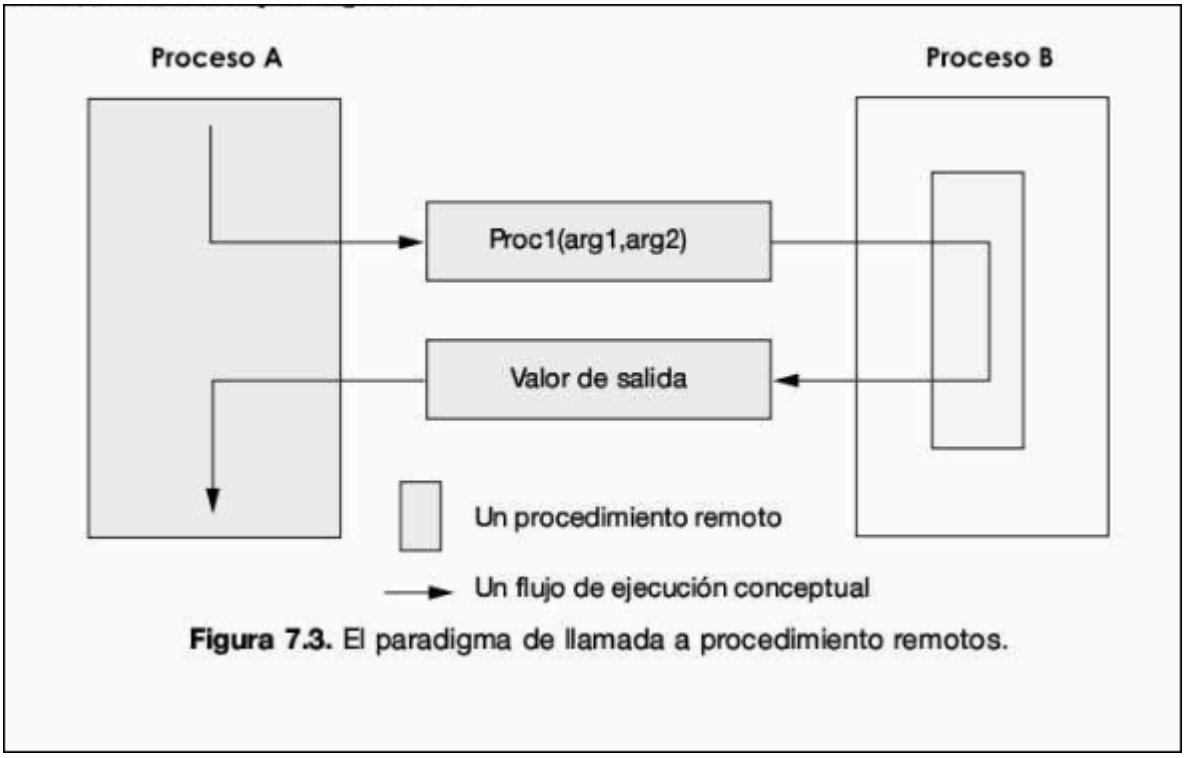

Figura 7.3. El paradigma de llamada a procedimiento remotos.

Llamada a procedimiento local y a procedimiento remoto

El modelo RPC se ha utilizado mucho en las aplicaciones de red. Existen dos APIs que prevalecen en este paradigma:

- **Open Networks Comuting Remote Procedure Call:** evolución del API de RPC que desarrolló Sun Microsystems
- **Open Group Distributed Computing ENviroment** RPC

Estas dos interfaces incorporan las herramientas `rpcgen`, que transforma las llamadas a procedimientos remotos en llamadas a procedimientos locales.

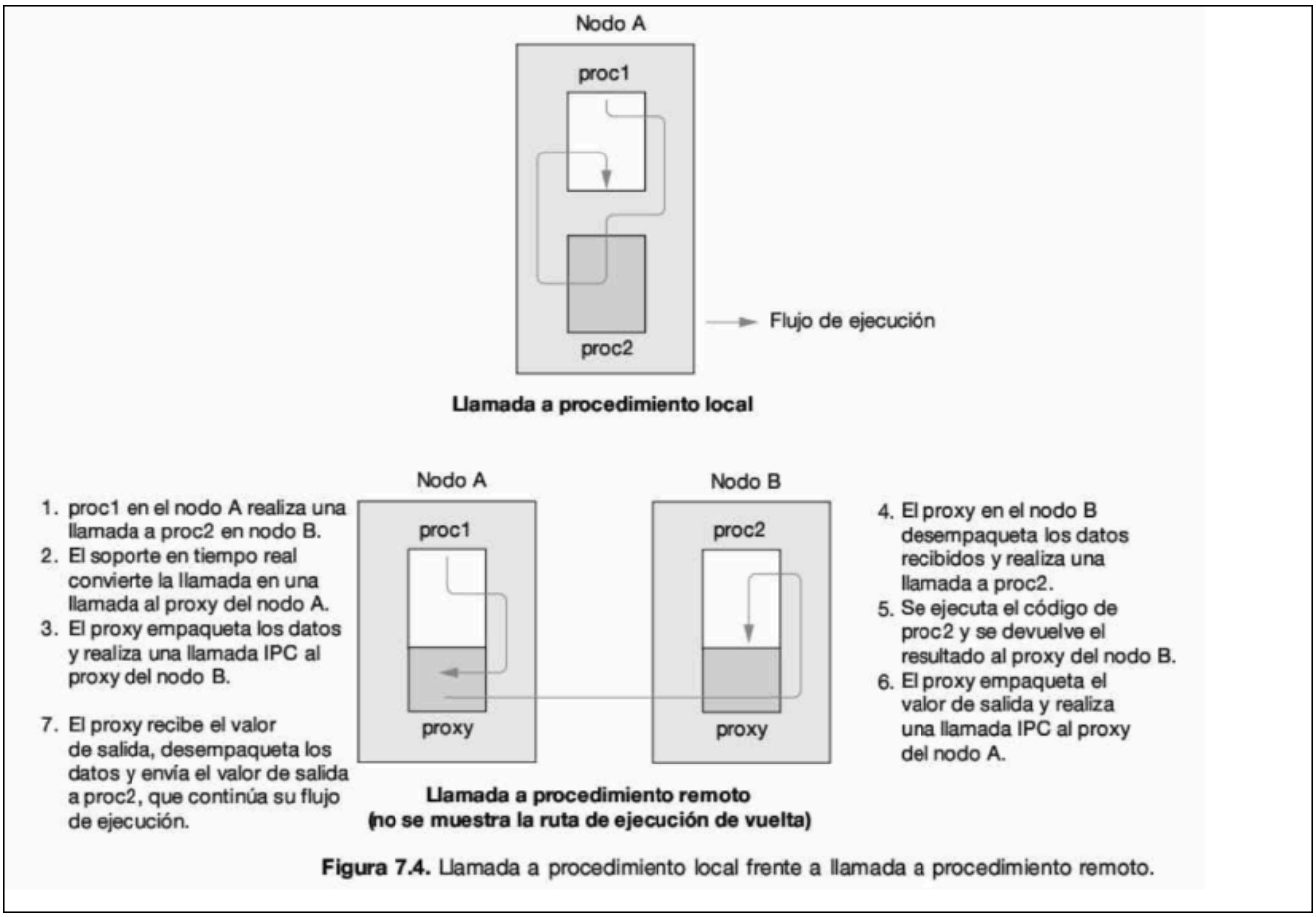

Figura 7.4. Llamada a procedimiento local frente a llamada a procedimiento remoto.

3.2.2 Remote Method Invocation

El paradigma RMI es una implementación orientada a objetos del modelo RPC. Además es una API exclusiva para programas Java.

En este paradigma un servidor de objeto exporta un objeto remoto y lo registra en el servicio de directorios. El objeto proporciona métodos remotos que pueden invocar los clientes.

El objeto remoto se declara como una interfaz remota (extensión de la interfaz Java). El servidor de objeto implementa la interfaz remota y el cliente accede al objeto gracias a la invocación de sus métodos.

Arquitectura

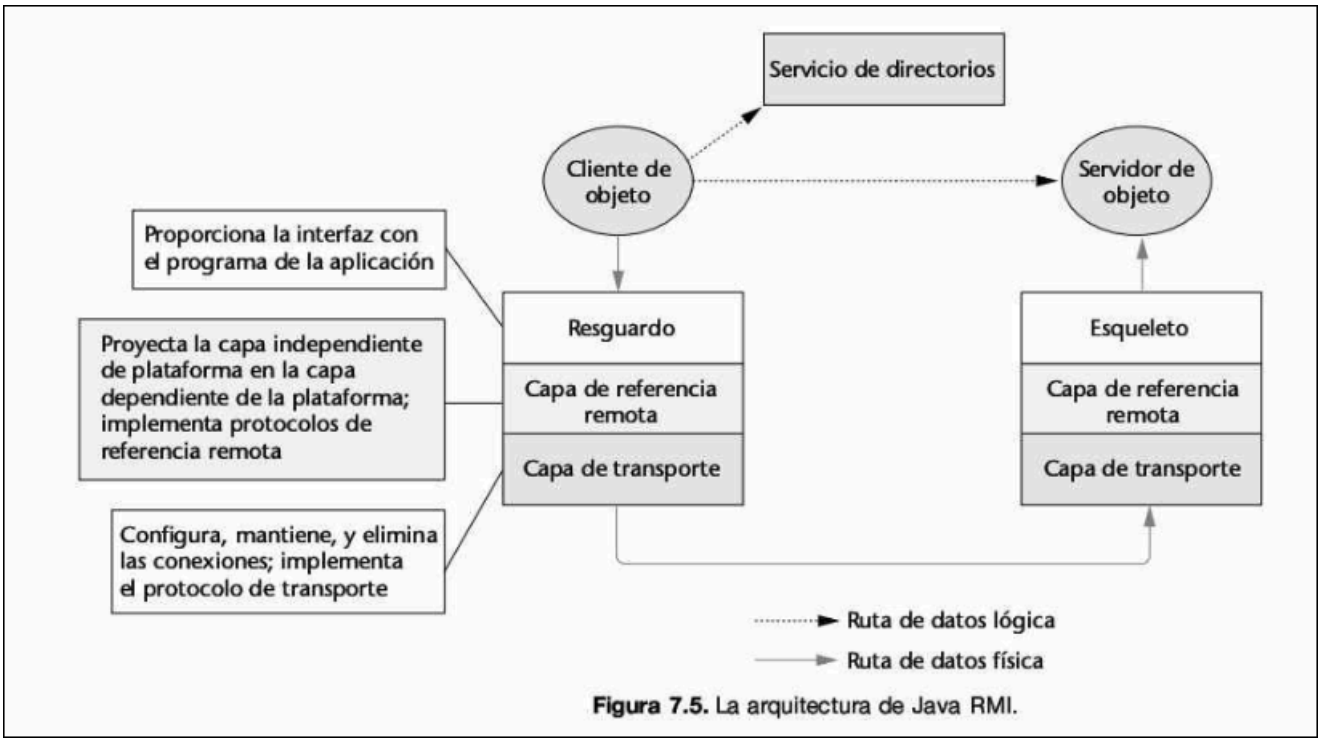

Figura 7.5. La arquitectura de Java RMI.

Parte del Cliente

- La **capa de resguardo o stub:** un proceso cliente invoca un método remoto y esta invocación es dirigida a un objeto proxy (**stub**). Esta capa de resguardo está justo debajo de la aplicación y sirve para interceptar las invocaciones a métodos remotos. Una vez interceptada la invocación se envía a la capa inferior, la **capa de referencia remota**.
- La **capa de referencia remota** interpreta y gestiona referencias a los objetos hechas por los clientes e invoca las operaciones entre procesos de la siguiente capa, la **capa de transporte**, para transmitir las llamadas a la máquina remota.
- La **capa de transporte** (basada en TCP) es orientada a conexión. Esta capa y el resto de arquitectura de la conexión entre procesos y transmiten los datos (llamada al método) a la máquina remota.

Parte del servidor

- La **capa, esqueleto o skeleton** está justo debajo de la capa de aplicación y se utiliza para interactuar con la capa de resguardo de la parte cliente.
- La **capa de referencia remota** gestiona y transforma la referencia remota originada por el cliente, en una referencia local
- La **capa de transporte** es orientada a conexión igual que la del cliente

Registro de objetos

- El API de RMI posibilita el uso de uso de diferentes servicios de directorios para registrar un objeto distribuido (uno de ellos es la **interfaz de nombrado y directorios** de Java=JNDI, es más general que el registro RMI ya que lo pueden usar aplicaciones que no usan el API RMI).

- El registro RMI, **rmiregestry**, es un servicio de directorios sencillo que proporciona el **kit** de desarrollo de software Java (SDK). Este registro es un servicio cuyo servidor se ejecuta en una **máquina del servidor del objeto** y utiliza el puerto **1099**.

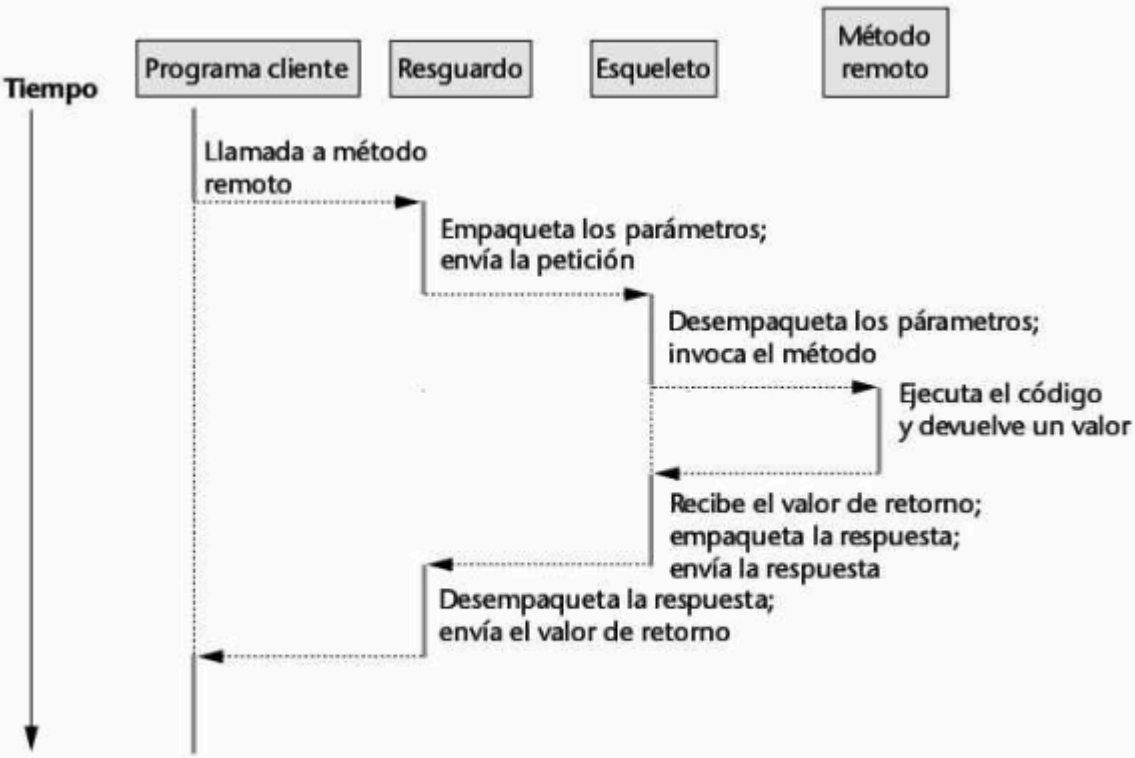

Figura 7.6. Interacciones entre el resguardo RMI y el esqueleto RMI (basado en un diagrama de [java.sun.com/docs, 8]).

3.3 El API de Java RMI

3.3.1 Interfaz remota

Este es el punto inicial para crear un objeto distribuido. 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. Es una interfaz que hereda de la clase `remote` que permite implementar la interfaz utilizando sintaxis RMI. Aparte de la extensión que se hace de esta clase y de que todas las declaraciones de los métodos deben especificar la clase `RemoteException`, una interfaz remota utiliza la misma sintaxis que una interfaz Java local.

Cada declaración de un método debe especificar la excepción (líneas 9 y 12). Cuando ocurre un error durante el procesamiento de la invocación del método remoto se lanza la excepción que debe ser gestionada en el programa del método que lo invoca.

Estos errores pueden ser: problemas en la comunicación entre los procesos (fallos de acceso, fallos de conexión) o problemas asociados a la invocación de métodos remotos (no encontrar un objeto, el resguardo o el esqueleto).

Figura 7.7. Un ejemplo de interfaz remota Java.

```
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
```

3.3.2 Software del servidor

Un objeto que proporciona los métodos y la interfaz de un objeto distribuido. Cada servidor debe:

- Implementar cada uno de los métodos remotos especificados en la interfaz
- Registrar en un servicio de directorios un objeto que contiene la implementación

Implementación interfaz remota

Se debe crear una clase que implemente la interfaz remota. La sintaxis es similar a una clase que implementa una interfaz local.

Las importaciones son necesarias para usar 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. Se debe definir un constructor de la clase. La primera línea del código debe ser una sentencia (llamada a `super()`) que invoque al constructor de la clase base. A continuación, debe aparecer la implementación de cada método remoto.

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
```

Generación del resguardo y del esqueleto (stub y skeleton)

Un objeto distribuido requiere un proxy por cada uno de los servidores y clientes del objetos (**skeleton y stub**). Estos proxies se generan utilizando el comipilador RM **rmic**: `rmic <nombre de la clase de la implementación de la interfaz remota>`.

Se generarán entonces: `(nombre de la clase)_skel.class` y `(nombre de la clase)_stub.class`. El fichero del stub y el de la interfaz remota deben compartirse con cada cliente del objeto (son imprescindibles para que el cliente compile). Además, una copia de cada fichero debe colocarse manualmente en la parte del cliente.

El servidor de objeto

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

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; después, se **exportará** la referencia a este objeto.

Exportación del objeto. Las líneas 20-23 exportan al objeto. Para esta exportación se debe registrar la referencia del objeto en un servicio de directorios (**remiregistry**). Es `rmregistry` debe ejecutarse en la máquina del servidor que exporta el objeto y está localizado en el puerto 1099. Además puede ejecutarse dinámicamente por el servidor.

En un sistema de producción donde se utilice el servidor de registro RMI por defecto y esté ejecutando continuamente la llamada `arrancarRegistro` y el método pueden

omitirse.

La clase `Naming` proporciona métodos para almacenar y obtener referencias del registro. (`rebind` = almacenar en el registro una referencia a un objeto con una URL). El método `rebind` sobreescribe cualquier referencia en el registro asociada al nombre de la referencia (para no sobreescribir usar `bind`). El nombre de la máquina debe corresponder con el nombre del servidor (o usar localhost). EL nombre de la referencia debe ser único en el registro.

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. Este servidor es concurrente (cada solicitud de un cliente de objeto se procesa a través de un hilo independiente del servidor, por lo que es importante que la implementación del objeto sea thread-safe).

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
```

3.3.3 Software del cliente

Es como cualquier otra clase Java. Para hacer uso de RMI suponge:

- Localizar el registro RMI en el nodo servidor.
- Buscar la referencia para el servidor del objeto. Se realizará un cast de la referencia a la clase de la interfaz remota y se invocarán los métodos remotos.

Sentencias de importación. Las sentencias de importación se necesitan para que el programa pueda compilar.

Búsqueda del objeto remoto. El cogios 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).

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

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
```

3.4 Diferencais entre RMI y el API de sockets

El API de RMI es una herramienta eficiente para construir aplicaciones de red. Puede utilizarse en lugar del API de `sockets` (que representa el paradigma e paso de mensajes) para construir una aplicación de red rápidamente.

- El API de `sockets` está más cercano al SO, 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.
- 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 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 otro.

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

3.5 Anexos

3.5.1 Algoritmo para desarrollar el software de la parte servidora

- Crear un directorio donde se almacenen todos los ficheros generados por la aplicación
- Especificar la interfaz remota del servidor en `InterfazEjemplo.java`. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
- Implementar la interfaz en `ImplEjemplo.java`. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
- 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.
- Crear un programa servidor de objeto `ServidorEjemplo.java`. Compilarlo y revisarlo hasta que no exista ningún error de sintaxis
- Activar el servidor de objeto: `java ServidorEjemplo`

3.5.2 Algoritmo para desarrollar el software de la parte cliente

- Crear un directorio donde se almacenen todos los ficheros generador por la aplicación.
- 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 genera el fichero `class` de la interfaz.
- Obtener una copia del fichero de resguardo para la implementación de la interfaz, `ImplEjemplo_Stub.class`
- Desarrollar el programa cliente `ClienteEjemplo.java`. Compilarlo y revisarlo hasta que no exista ningún error de sintaxis.
- Activar el cliente: `java ClienteEjemplo`

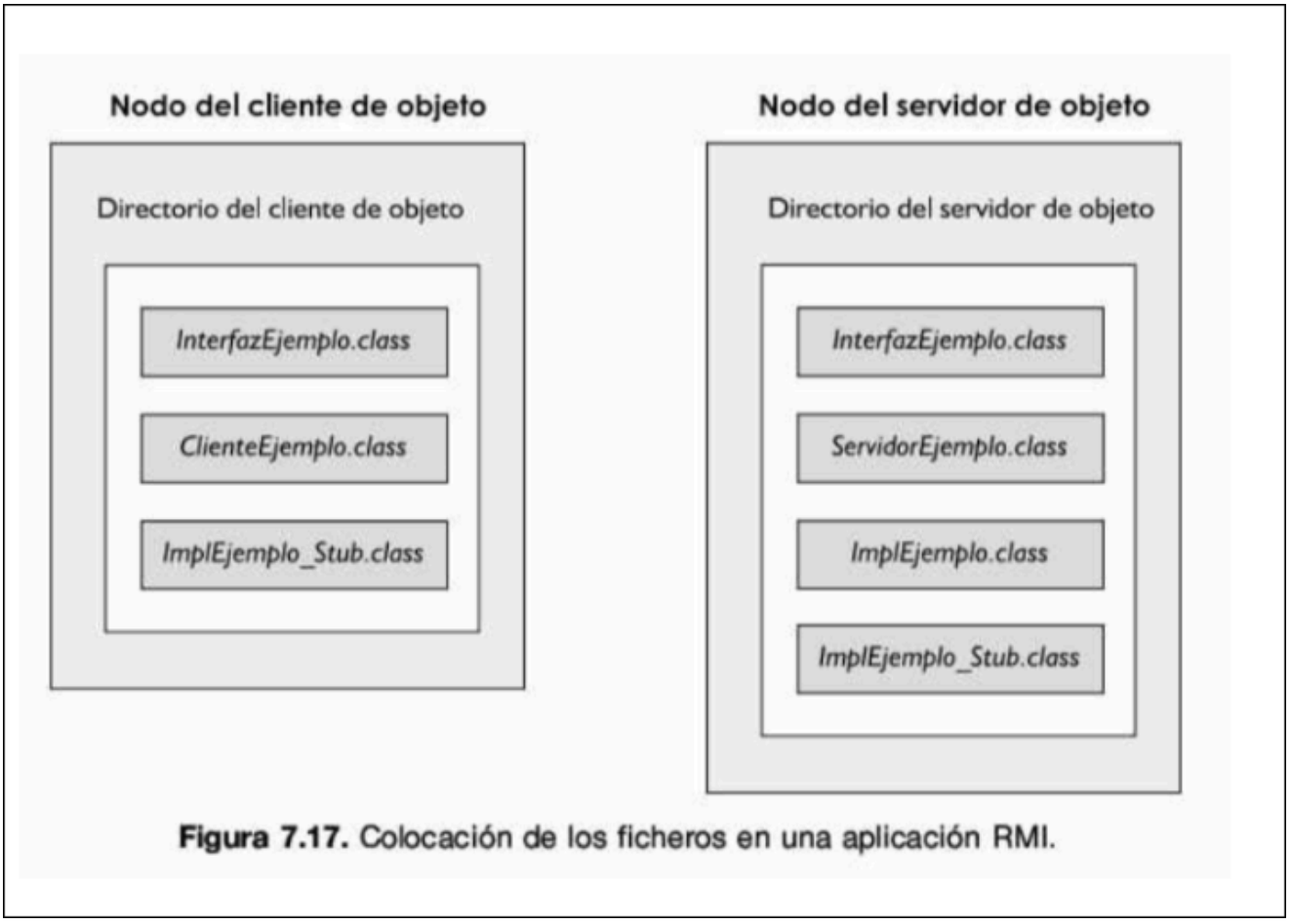

Figura 7.17. Colocación de los ficheros en una aplicación RMI.