3. Objetos Distribuidos

3.1 Evolución del Paradigma: Del Mensaje al Objeto

Para entender dónde estamos, primero debemos contrastar este nuevo modelo con lo que ya conoces (Sockets/Paso de Mensajes).

3.1.1 Paso de Mensajes (Nivel Bajo)

- **Enfoque:** Orientado a **datos**.
- **Funcionamiento:** Simula una conversación humana. Un proceso envía un mensaje formateado y espera que el otro lo interprete.
- **Acoplamiento:** Alto. Los procesos deben estar activos y sincronizados; si el enlace cae, la colaboración falla.
- **Limitación:** En aplicaciones complejas con miles de interacciones, interpretar mensajes y mantener protocolos se vuelve una tarea "inabordable".

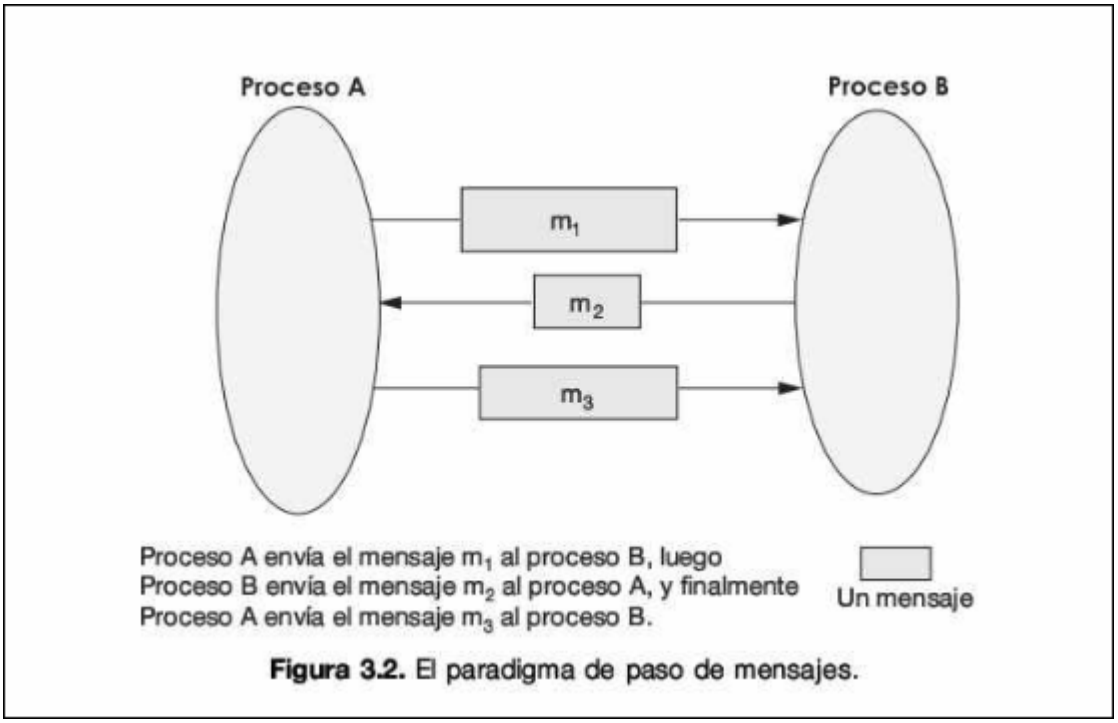

Figura 3.2. El paradigma de paso de mensajes.

> ⓘ **Info**
>
> **Orientado a datos** es como mandar una carta o formulario. El énfasis está en el **contenido** del mensaje. Los procesos intercambian paquetes de datos con un formato acordado y el receptor debe abrir el paquete, leer los datos y decidir qué

hacer con ellos. Es menos intuitivo porque te obliga a pensar en protocolos de comunicación en lugar de en la lógica del programa.

3.1.2 Objetos Distribuidos (Nivel Alto)

- **Enfoque:** Orientado a **acciones**.
- **Abstracción:** Trata los recursos de la red como objetos encapsulados. Los datos pasan a un segundo plano (son meros parámetros).
- **Ventaja:** Es más natural para el desarrollo de software moderno (POO), aunque conceptualmente sea menos intuitivo para los humanos que "enviar un mensaje".

ⓘ Info

Orientado a acciones es como dar una orden directa. El énfasis está en **invocar una operación** (un verbo, una acción). No piensas en "enviar datos", sino en ejecutar una función específica (método) sobre un objeto, pasando los datos simplemente como argumentos. Aunque ocurre red por debajo, para ti como programador es "pedir que se haga algo".

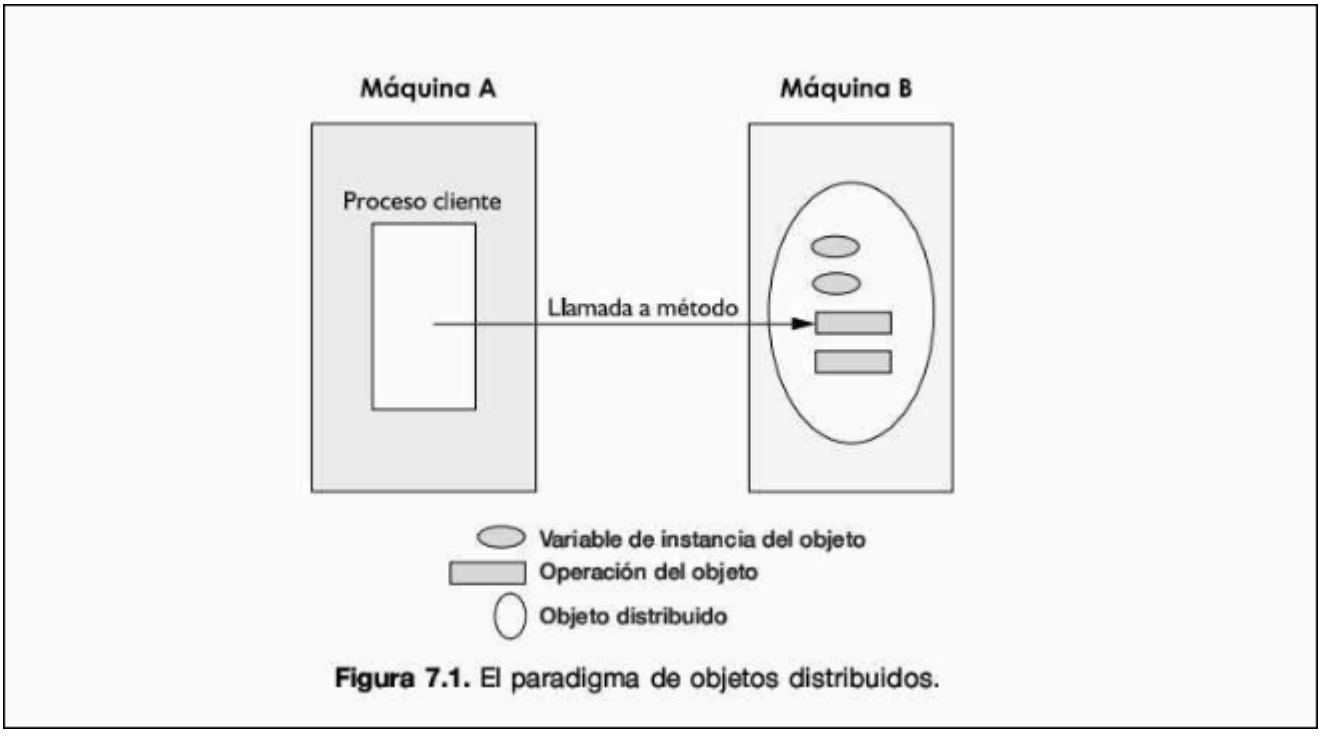

Figura 7.1. El paradigma de objetos distribuidos.

3.2 Conceptos Fundamentales

Antes de escribir código, debemos definir rigurosamente la terminología arquitectónica.

3.2.1 Tipos de Objetos

- **Objeto Local:** sus métodos solo pueden ser invocados por un proceso que se ejecuta en el mismo computador (mismo espacio de direcciones).
- **Objeto Distribuido (Remoto):** sus métodos pueden ser invocados por un proceso situado en un ordenador distintos conectado por red

3.2.2 La arquitectura de intermediarios (Proxies)

Dado que el cliente no puede tocar físicamente la memoria del servidor, necesitamos intermediarios que creen la ilusión de localidad.

- **Objeto Servidor:** es la entidad que exporta el servicio y ejecuta la lógica real.
- **Referencia:** un puntero lógico que permite localizar al objeto en la red
- **Registro de Objetos:** un servicio de directorio (como una guía telefónica) donde el servidor registra sus objetos y el cliente busca referencias

> ⓘ **Info**
>
> En el contexto de RMI, **exportar un servicio** es el acto de **hacer visible y accesible** un objeto local para el resto del mundo.

El Mecanismo de Sub y Skeleton

La comunicación no es directa. A nivel físico, ocurre lo siguiente:

- **Cliente:** invoca el método sobre un **Proxy de Cliente (Stub)**.
- **Marshalling (Empaquetado):** el Stub empaqueta los argumentos y la petición y los pasa al soporte de ejecución (runtime).
- **Red:** los datos viajan al servidor
- **Skeleton (Proxy del Servidor):** el soporte de ejecución del servidor recibe los datos, los desempaqueta (**Unmarshalling**) y se los pasa al Skeleton.
- **Invocación Real:** el Skeleton invoca al método en el objeto real, obtiene el resultado, lo empaquea y lo envía de vuelta.

A nivel **lógico**, parece que el cliente llama directamente al servidor. A nivel **físico**, los datos atraviesan capas de proxies, soporte de ejecución y red

> ⓘ **Info**
>
> Un **proxy** es un intermediario. Es un componente de software que se hace pasar por otra cosa para ocultar la complejidad de la red (en este caso para entenderlo, aunque su definición rigurosa es otra). Su trabajo es que el cliente crea que está

hablando con el objeto real, cuando en realidad está halando con un representante local que gestiona el envío de la petición a través de la red.

Imagina que quieres hablar con un gran empresario (el Objeto Servidor) que vive en otro país. No puedes gritarle directamente.

- **El Stub (tu representante local):** es como un **doble** del empresario que está en tu oficina. Tú le hablas a él como si fuera el real. Él no sabe la respuesta, pero toma nota de tu pregunta, la mete en un sobre (**marshalling**) y la echa al correo. Para tu código, el Stub es el servidor
- **El Skeleton (El secretario del empresario):** está en la oficina del empresario real. Su trabajo es recibir el sobre del correo, abrirlo (**unmarshalling**), leerle la pregunta al empresario real, apuntar la respuesta, meterla en otro sobre y enviarla de vuelta.

El Stub simula ser el servidor en el lado del cliente. El Skeleton recibe los mensajes de red y llama al servidor real

Si el Stub y Skeleton son los que escriben y leen las cartas, el **Soporte de Ejecución** es el **servicio de correos** y el sistema de transporte.

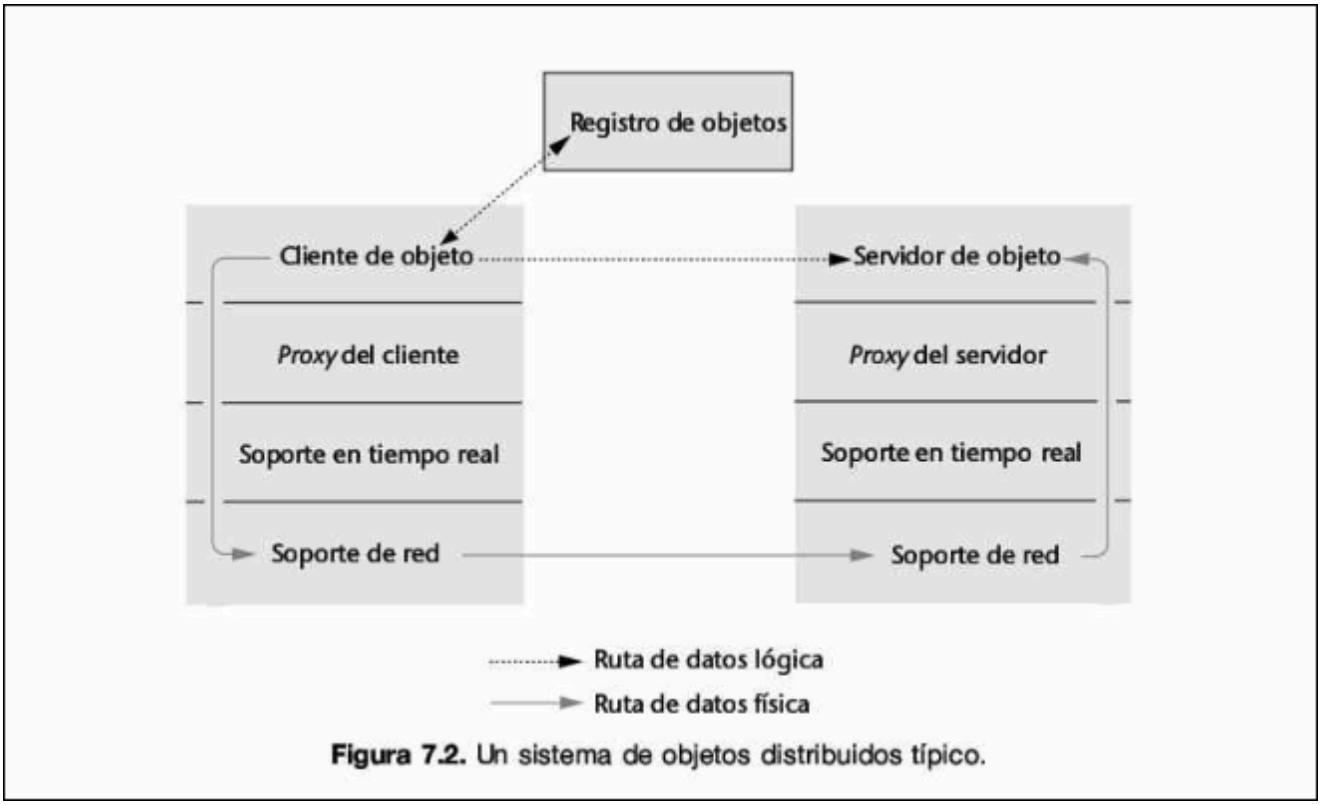

Figura 7.2. Un sistema de objetos distribuidos típico.

3.3 De RPC a RMI: El salto Tecnológico

3.1 Remote Procedure Call (RPC)

RMI no nació de la nada. Su ancestro es **RPC**, popularizado en los años 80.

- En RPC, un proceso llama a un **procedimiento** (función C) en otro proceso
- Utiliza herramientas como `rpcgen` para crear los stubs automáticamente.

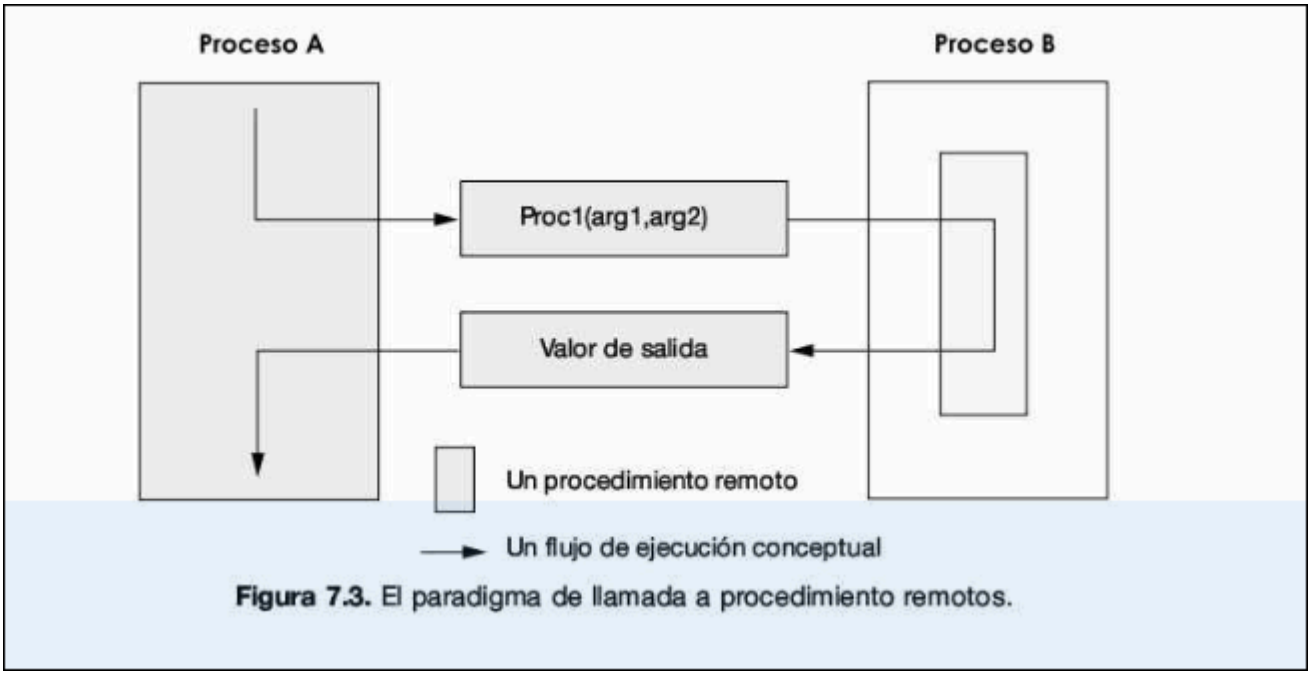

Figura 7.3. El paradigma de llamada a procedimiento remotos.

3.2 Java RMI (Remote Method Invocation)

RMI es la evolución **Orientada a Objetos de RPC**, exclusiva para el ecosistema Java

- Permite pasar objetos completos como parámetros o valores de retorno, no solo tipos de datos primitivos.
- Utiliza un **Registro RMI** (`rmiregistry`) para la localización de servicios.

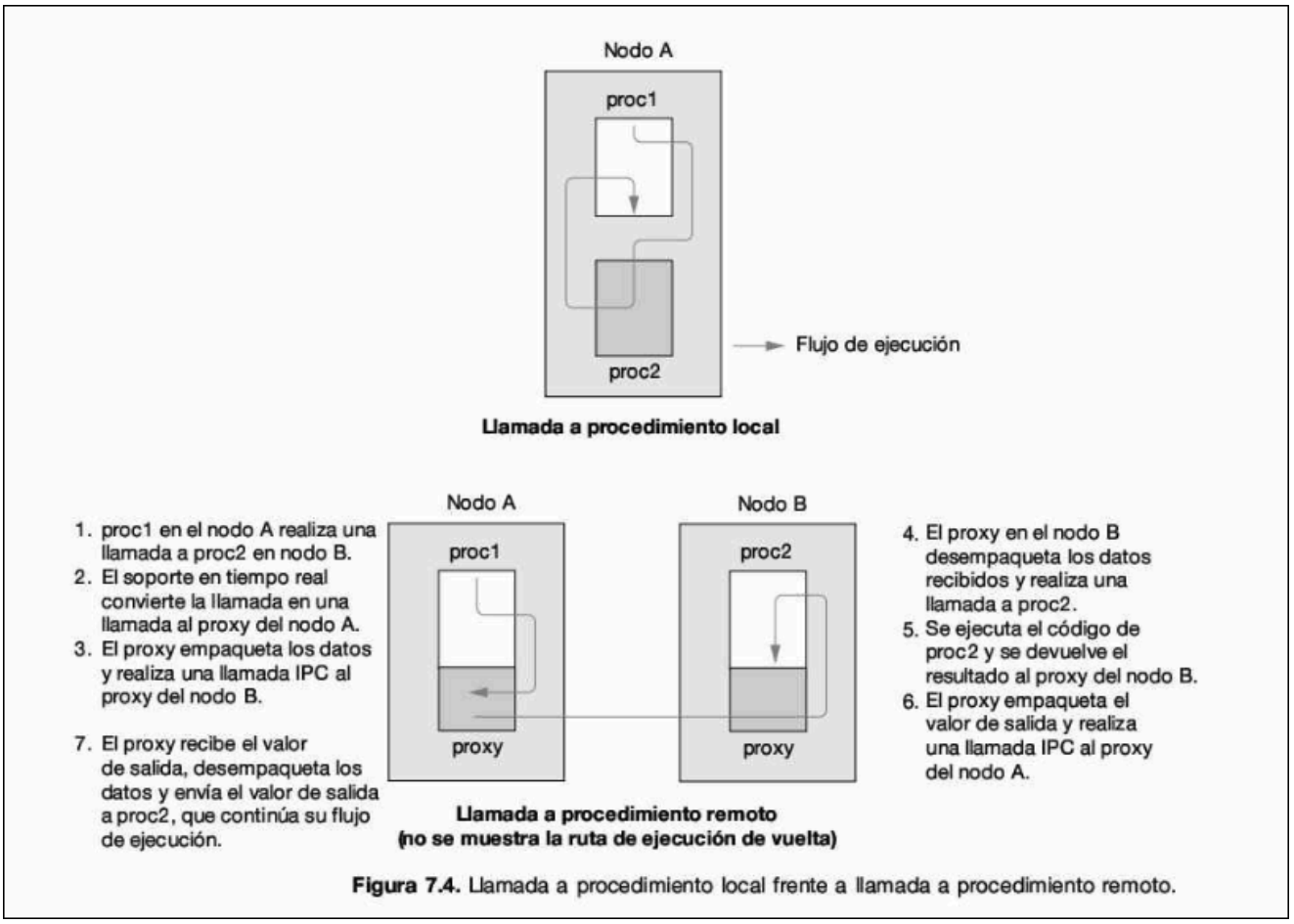

Figura 7.4. Llamada a procedimiento local frente a llamada a procedimiento remoto.

3.4 Implementación en Java RMI

Para construir una aplicación distribuida, Java RMI impone una metodología estricta basada en interfaces.

3.4.1 La interfaz Remota

Es el contrato entre cliente y servidor. Define qué se puede hacer, pero no cómo.

- Debe extender `java.rmi.REmote`
- **Regla de Oro:** todos los métodos deben declara que lanzan `java.rmi.RemoteException`

> **¿Por qué RemoteException?** Porque en sistemas distribuidos, las cosas fallan. La red puede caerse, el servidor puede no existir, o el stub puede perderse. Java obliga al programador a manejar estos fallos explícitamente.

JAVA

```java
public interface SomeInterface extends java.rmi.Remote {
    public String someMethod1() throws java.rmi.RemoteException;
}
```

3.4.2 La Implementación del Servidor

Es la clase que hace el trabajo real.

- Debe implementar la interfaz remota definida anteriormente
- Generalmente hereda de `UnicastRemoteObject`. Esto facilita que el objeto sea "exportable" a la red.
- Debe tener un constructor que lance `RemoteException` (porque la clase padre lo hace).

3.4.3 Generación de Stubs y Skeletons

Una vez compiladas las clases, se usa el compilador RMI para generar los proxies:

- Comando: `rmic NOmbreDeLaImplementacion`
- Resultado: Generado dos archivos: `_Stub.class` y `_Skel.class`
  - **Importante:** El archivo `_Stub.class` debe copiarse manualmente a la máquina del cliente (o descargarse dinámicamente) para que este sepa cómo comunicarse.

> **Info**
>
> A día de hoy se generan de forma automática a no ser que uses una versión inferior a java 5 (que debe ser la puta mierda que usa el profesor, el pavo no actualiza su asignatura desde 2008).

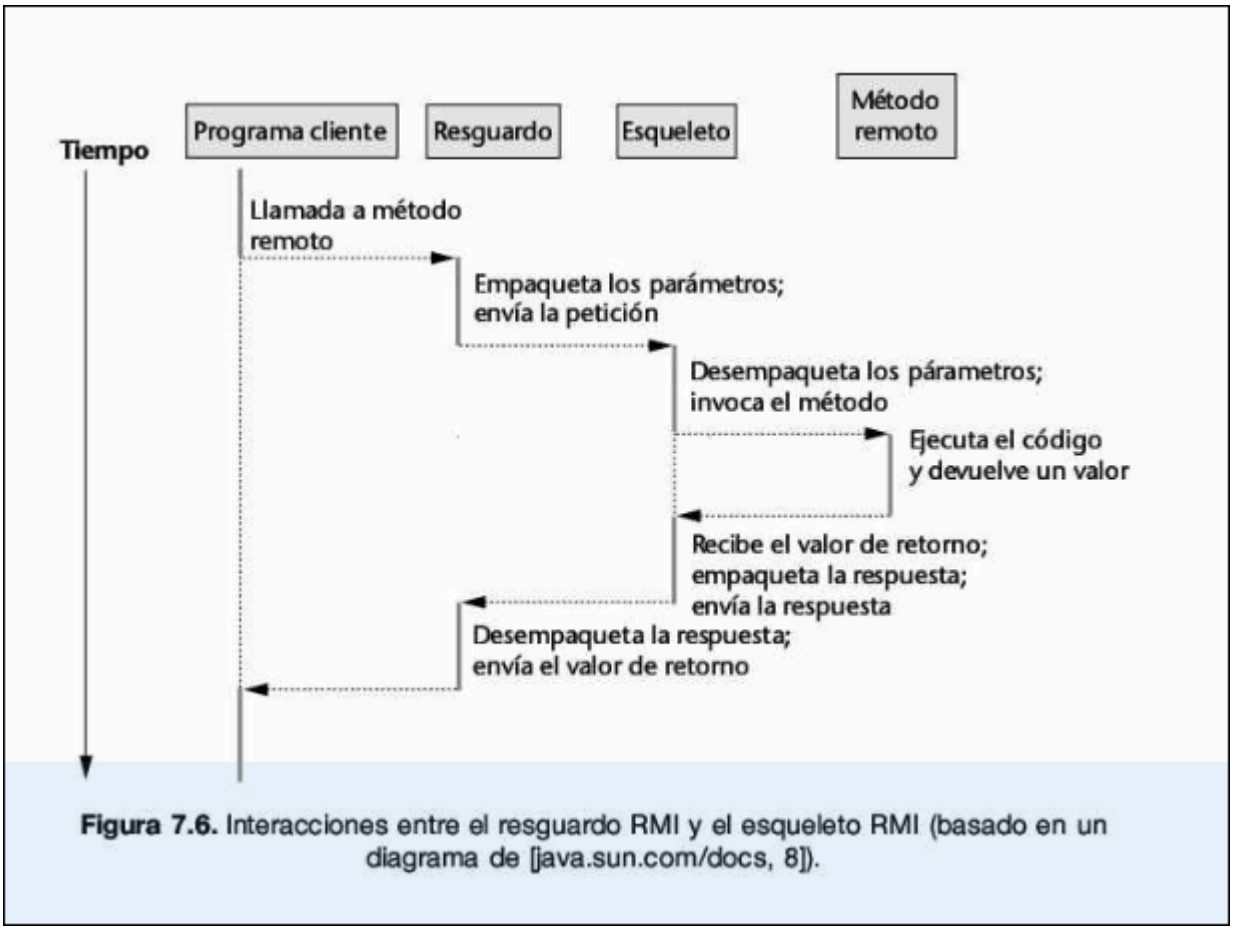

Figura 7.6. Interacciones entre el resguardo RMI y el esqueleto RMI (basado en un diagrama de [java.sun.com/docs, 8]).

3.5 El Registro y la Puesta en Marcha

3.5.1 El RMI Registry

Es un servicio simple que se ejecuta por defecto en el puerto TCP 1099. Actúa como un mapa de `String` → `Objeto`.

- Se puede iniciar desde consola: `rmiregistry`
- O desde código Java: `LocateRegistry.createRegistry(puerto)`

3.5.2 Lógica del Servidor (Publicación)

El servidor debe instanciar el objeto y "publicarlo" usando la clase `Naming`.

- **URL RMI:** `rmi://host:port/nombre_servicio`.
- **Métodos Clave:**
  - `bind(url,objeto)`: registra el objeto. Falla si ya existe ese nombre
  - `rebind(url,objeto)`: registra o sobreescribe si ya existe (más seguro/común)

3.5.3 Lógica del Cliente (Búsqueda)

El cliente necesita localizar el objeto para obtener una referencia.

- Construye la URL de búsqueda
- Usa `Naming.lookup(url)`.

> **Advertencia Crítica:** `Naming.lookup` devuelve un objeto tipo `Remote`. Debes hacer un **cast** a la **Interfaz Remota**, NUNCA a la clase de implementación del servidor. El cliente no conoce la implementación, solo la interfaz.

3.6 Ejemplo Completo

Paso 1: La Interfaz Remota

Define el contrato. Debe extender `Remote` y lanzar `RemoteException`.

JAVA

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

// Interfaz compartida por Cliente y Servidor
public interface Calculadora extends Remote {
    // Método que queremos invocar remotamente
    public int sumar(int a, int b) throws RemoteException;
}
```

Paso 2: El Servidor (Implementación y Publicación)

Implementa la lógica y se registra en el `rmiregistry`.

JAVA

```java
import java.rmi.server.UnicastRemoteObject;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;

// La implementación real
public class ServidorCalculadora extends UnicastRemoteObject implements
Calculadora {

    protected ServidorCalculadora() throws RemoteException {
        super(); // Necesario para exportar el objeto
    }

    @Override
    public int sumar(int a, int b) throws RemoteException {
        System.out.println("Me han pedido sumar: " + a + " + " + b);
        return a + b;
    }

    public static void main(String[] args) {
        try {
            // 1. Iniciar el registro RMI en el puerto 1099 (para no usar
consola)
            LocateRegistry.createRegistry(1099);

            // 2. Crear el objeto
            ServidorCalculadora objetoRemoto = new ServidorCalculadora();

            // 3. Registrarlo con un nombre ("Naming.rebind")
            Naming.rebind("rmi://localhost:1099/MiCalculadora",
objetoRemoto);

            System.out.println("Servidor listo y esperando...");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Paso 3: El Cliente

Busca el objeto y lo usa.

```java
import java.rmi.Naming;

public class ClienteCalculadora {
    public static void main(String[] args) {
        try {
            // 1. Buscar el objeto en el registro (Lookup)
            // OJO: Hacemos casting a la INTERFAZ, no a la clase del
servidor
            Calculadora calc = (Calculadora)
Naming.lookup("rmi://localhost:1099/MiCalculadora");

            // 2. Usar el objeto como si fuera local
            int resultado = calc.sumar(5, 10);

            System.out.println("El resultado de la suma remota es: " +
resultado);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

3.7 Algoritmo de Desarrollo Paso a Paso

Para no perderte en la práctica, sigue este "algoritmo" riguroso extraído del profesor (gilipollez histórica, programa con lógica y pista, no vas a hacerte el código sin haber hecho antes la interfaz):

1. **Definir Interfaz:** Crear `Interfaz.java` (extiende `Remote`) y compilar.
2. **Implementar:** Crear `Impl.java` (implementa `Interfaz`, extiende `UnicastRemoteObject`) y compilar.
3. **Generar Proxies:** Ejecutar `rmic Impl`. Obtendrás el Stub y el Skeleton.
4. **Desarrollar Servidor:** Crear `Server.java` que instancia `Impl` y hace `Naming.rebind`.
5. **Desarrollar Cliente:** Crear `Client.java` que hace `Naming.lookup` y usa la interfaz.
6. **Despliegue de Ficheros:**
   - **Servidor:** Necesita `Interfaz.class`, `Impl.class`, `Skeleton.class`, `Server.class`.
   - **Cliente:** Necesita `Interfaz.class`, `Client.class`, `Stub.class`.

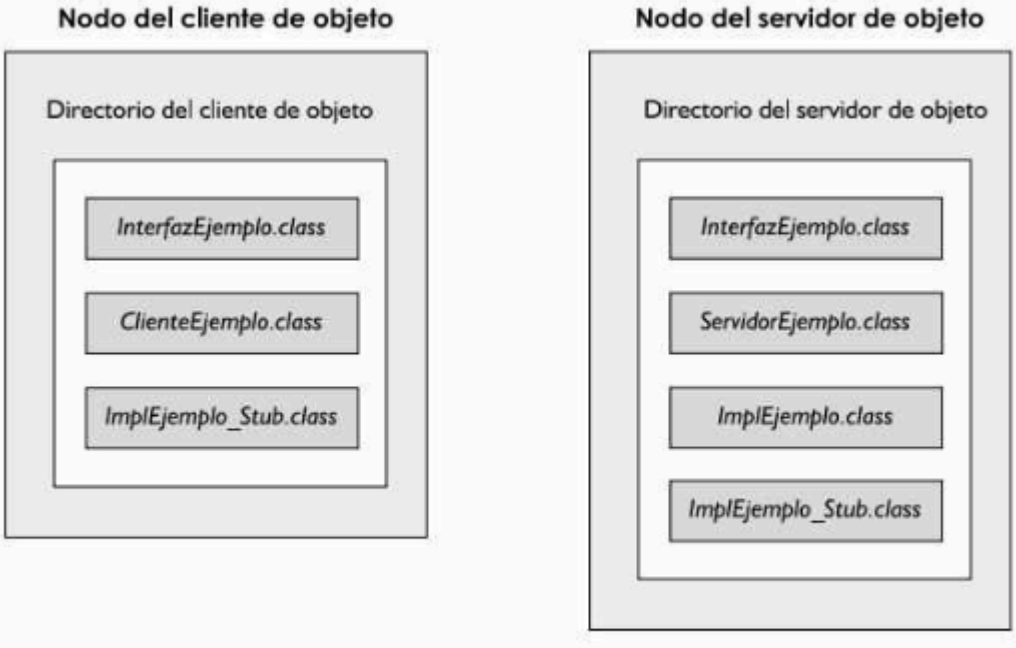

Figura 7.17. Colocación de los ficheros en una aplicación RMI.