

## 4. RMI Avanzado

Copyright (c) 2025 Adrián Quiroga Linares Lectura y referencia permitidas;  
reutilización y plagio prohibidos

### 4.1 Cuestiones Avanzadas

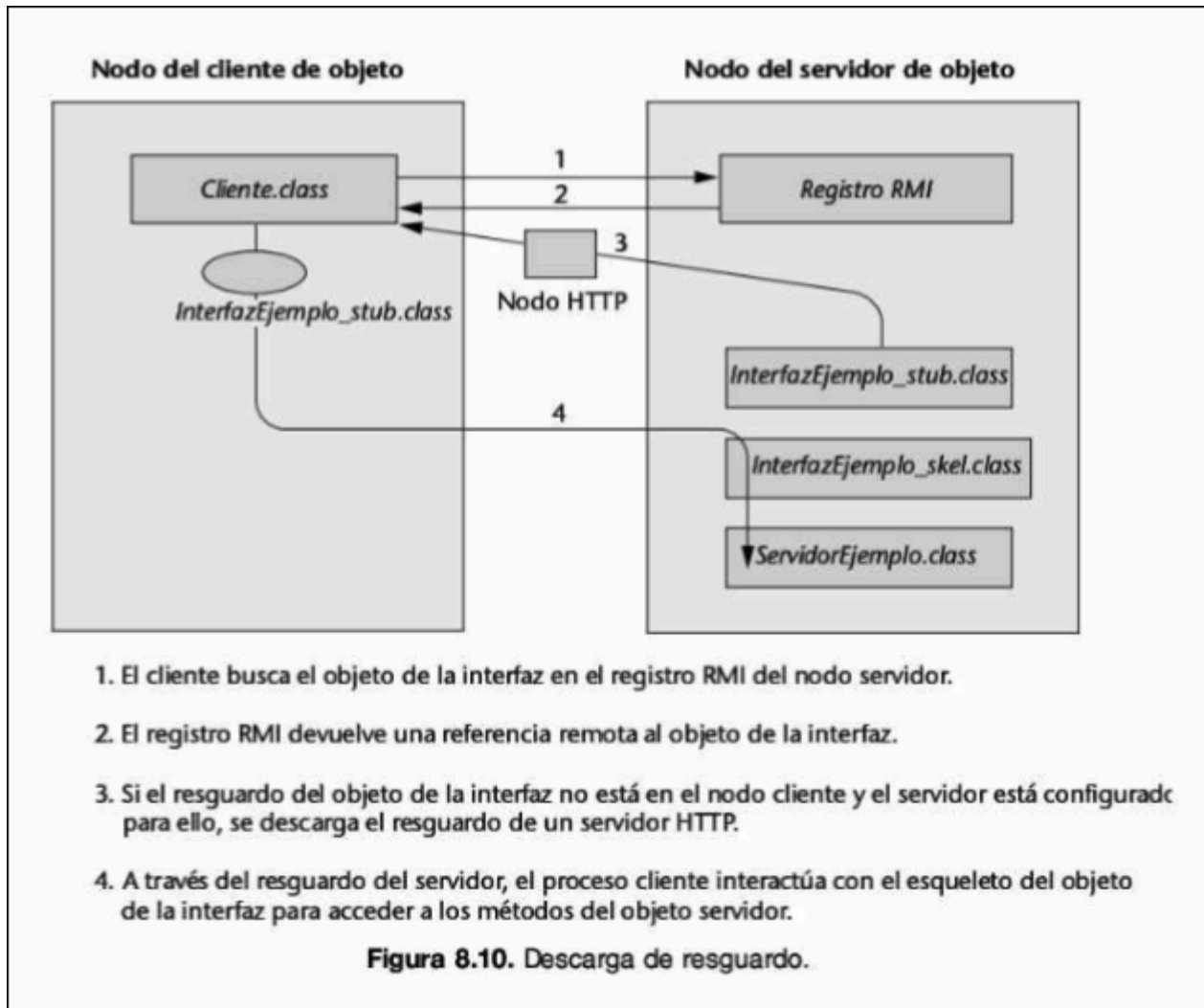
RMI avanzado incorpora nuevos mecanismos que pueden ser útiles para los desarrolladores de aplicaciones:

- **Descarga de resguardo** (stub downloading)
- **Gestiones de seguridad** (security manager)
- **Callback** de cliente
- **Serialización** y envío de objetos

### 4.2 Stub Downloading

Como ya vimos en el tema anterior Java RMI incorpora un método para que los clientes obtengan dinámicamente el **stub**. A este método se le llama descarga dinámica de resguardo. Se puede obtener dinámicamente la clase resguardo de un servidor HTTP de forma que puede interactuar con el cliente de objeto.

## Interacción entre cliente de objeto, servidor de objeto y el registro RMI cuando se utiliza descarga de resguardo



## Uso de descarga de resguardo y un fichero de políticas de seguridad

- Si debe descargarse el resguardo de un servidor HTTP, transfiera la clase resguardo a un directorio apropiado del servidor HTTP y es necesario asegurarse de que tiene los permisos de acceso adecuados
- Cuando activamos el servidor se debe especificar las siguientes opciones:

```
java -Djava.rmi.server.codbase=<URL>
    -Djava.security.policy=
    <ruta completa del fichero de políticas de seguridad>
```

donde **<URL>** es el URL del directorio donde se encuentra la clase resguardo, por ejemplo, <http://www.miempresa.com/resguardos/>.

Obsérvese la barra del final del URL, que indica que el URL especifica un **directorio**, no un fichero.

`<ruta completa del fichero de políticas de seguridad>` especifica el fichero de políticas de seguridad de la aplicación, por ejemplo, `java.security`, si el fichero `java.security` se encuentra en el directorio actual.

## Fichero `java.policy`

El fichero de políticas de seguridad de Java es un fichero de texto que contiene códigos que permiten la concesión de permisos específicos. Es recomendable hacer una copia del fichero en el directorio tanto de la máquina del cliente como en la del servidor.

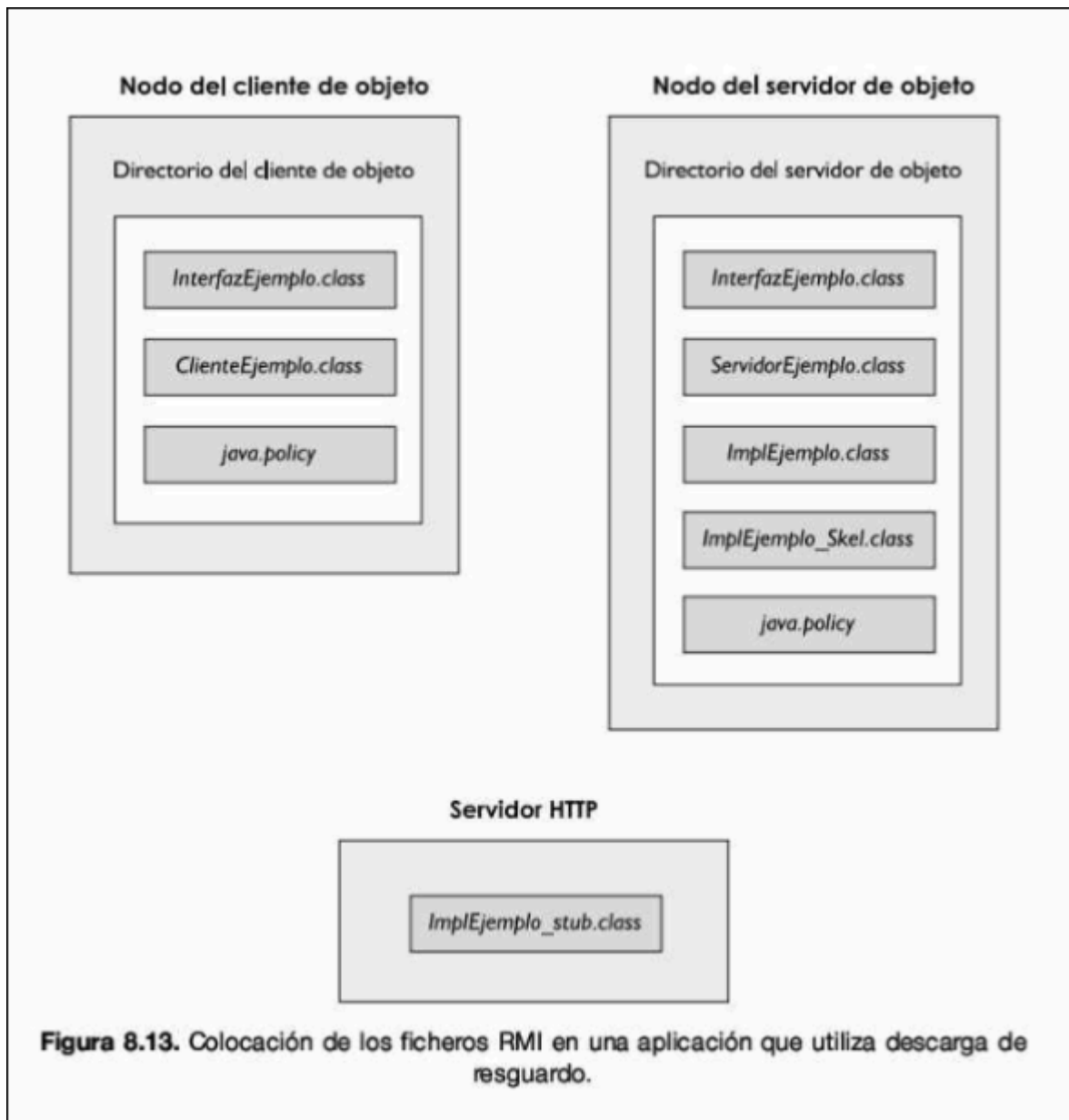
Cuando activemos el cliente se debe utilizar la siguiente opción para especificar los permisos que debe tener el cliente de objeto, definidos en el fichero de políticas:

```
java -Djava.security.policy=java.policy ClienteEjemplo
```

Lo mismo para el servidor

```
java -Djava.security.policy=java.policy ServidorEjemplo
```

## Colocación de los ficheros



### 4.3 RMI Security Manager

Puesto que RMI involucra el acceso desde/a una máquina remota y posiblemente la descarga de objetos, es importante que tanto el servidor como el cliente se protejan ante accesos inadecuados o no permitidos.

**RMISecurityManager** es una clase de Java que puede ser instanciada tanto en el cliente como en el servidor para limitar los privilegios de acceso. Para escribir nuestro propio gestor de seguridad podemos utilizar:

```
System.setSecurityManager(new RMISecurityManager());
```

JAVA

# Algoritmo para construir aplicación RMI

## Lado Servidor

- Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
- Especificar la interfaz remota y compilarla para generar el archivo .class de la interfaz
- Construir el servidor remoto implementando la interfaz y compilarlo hasta que no haya ningún error
- Usar **rmic** para procesar la clase del servidor y generar un fichero .class de **stub** y un fichero .class de **skeleton**: `rmic SomeServer`
- Si se desea **stub downloading**, copiar el fichero **stub** al directorio apropiado del servidor HTTP
- Activar el **RMIRegistry** en el caso de que no haya sido activado previamente.
- Construir un fichero de políticas de seguridad para la aplicación llamado `java.policy`
- Activar el servidor especificando:
  - Campo `codebase` si se utiliza stub downloading
  - Nombre del servidor
  - Fichero de políticas de seguridad

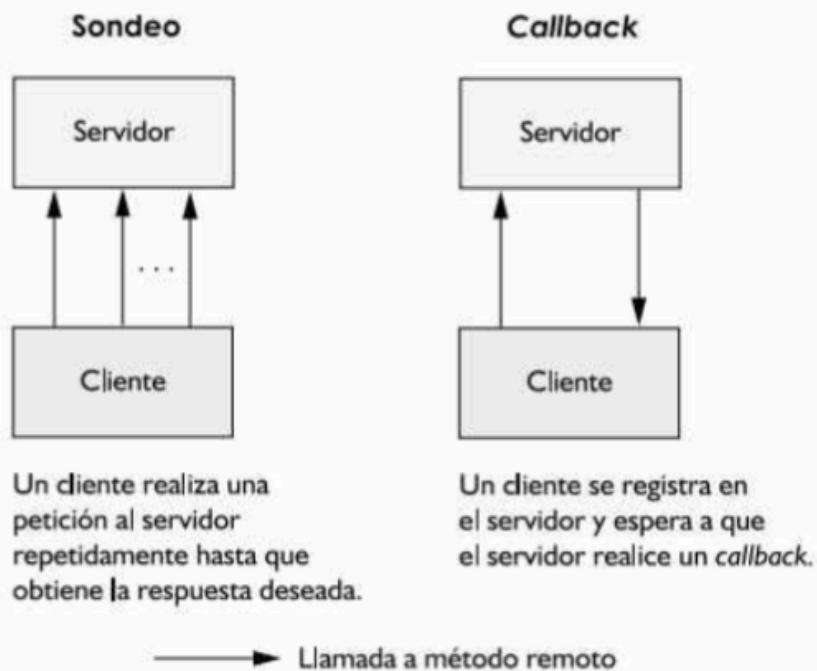
## Lado Cliente

- Crear un directorio donde se almacenen todos los ficheros generados por la aplicación
- Implementar el programa cliente o applet y compilarlo para genera la clase cliente.
- Si no se puede usar **stub downloading**, copiar el fichero class de **stub** a mano.
- Especificar el fichero de políticas de seguridad **java.policy**
- Activar el cliente especificado:
  - nombre p() del servidor
  - fichero con las políticas de seguridad

## 4.4 RMI Callbacks

En el modelo cliente-servidor, el servidor es pasivo (la comunicación IPC es iniciada por el cliente, el servidor espera por la llegada de las peticiones y proporciona las respuestas). Pero algunas aplicaciones necesitan que el **servidor inicie la comunicación** ante la ocurrencia de determinados eventos (juegos, subastas, ...)

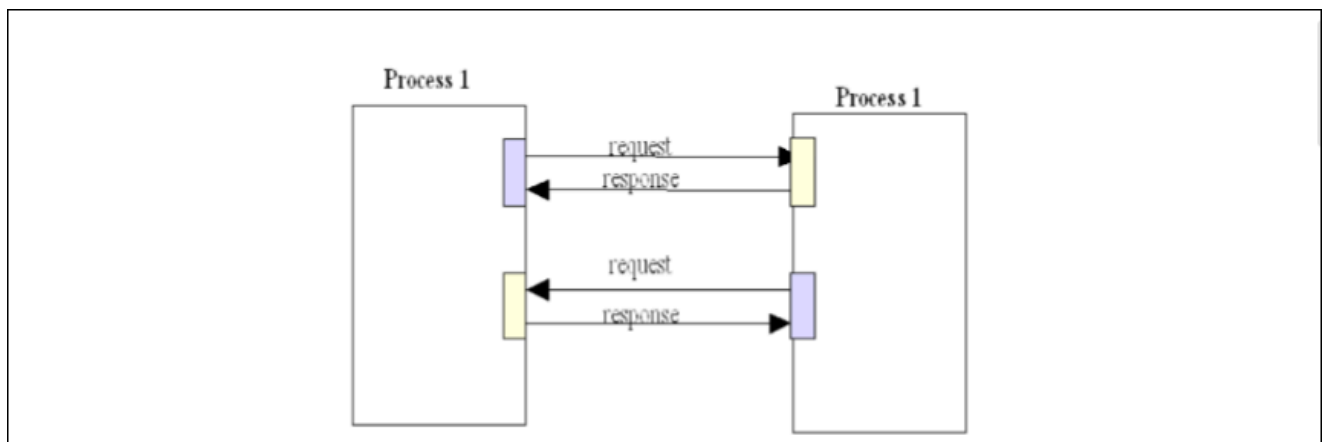
Una forma de llevar a cabo la transmisión de información es que cada proceso cliente realice un **sondeo** (pulling) a un servidor pasivo repetidas veces sin necesitar ser notificado de que un evento ha ocurrido en el servidor.



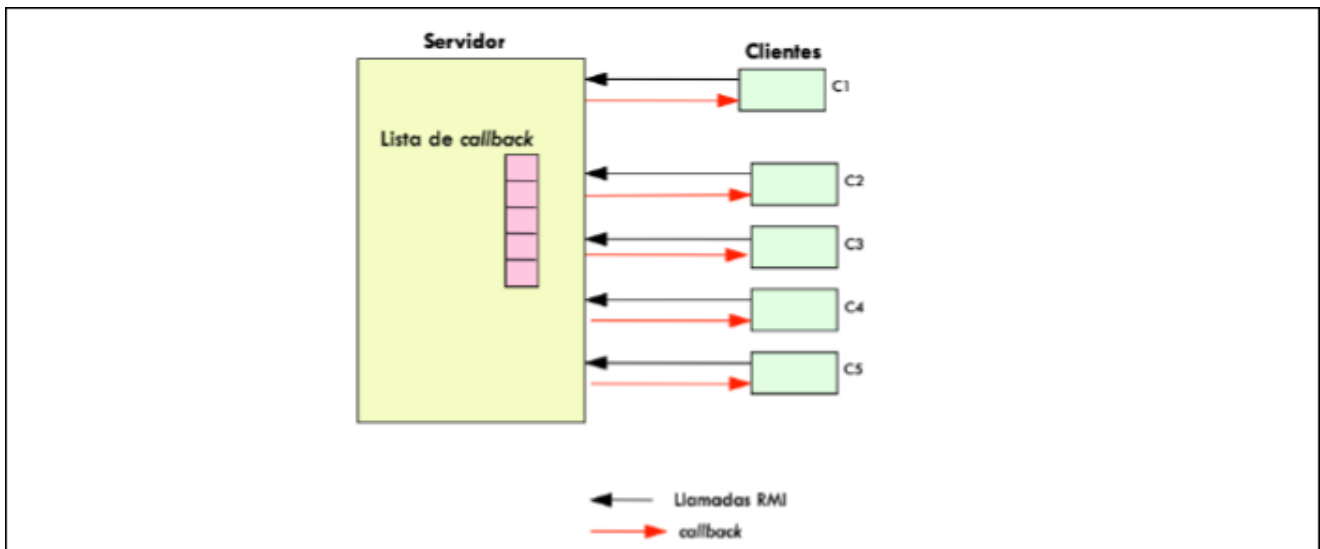
**Figura 8.1.** Sondeo (*polling*) frente a *callback*.

## Comunicaciones en ambos sentidos vs con Callbacks

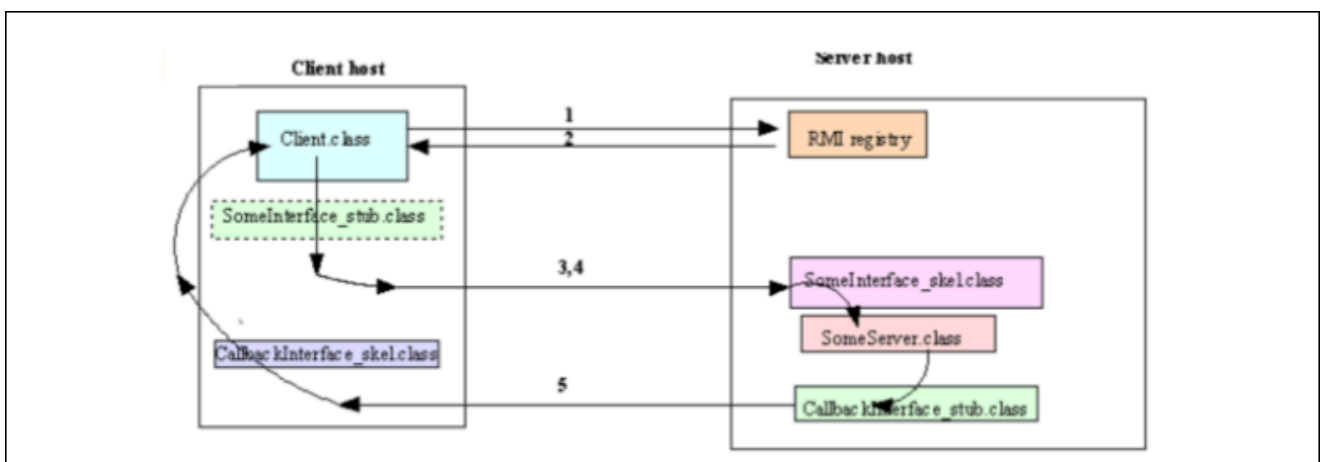
Algunas aplicaciones necesitan que ambos lados puedan iniciar una comunicación IPC. Al usar sockets (dos en cada lado) podemos conseguir una comunicación dúplex. Con sockets orientados a conexión cada lado actúa tanto como cliente como servidor.



Sin embargo un cliente callback se registra en un servidor RMI. El servidor realiza el callback a cada cliente registrado ante la ocurrencia de un determinado evento.

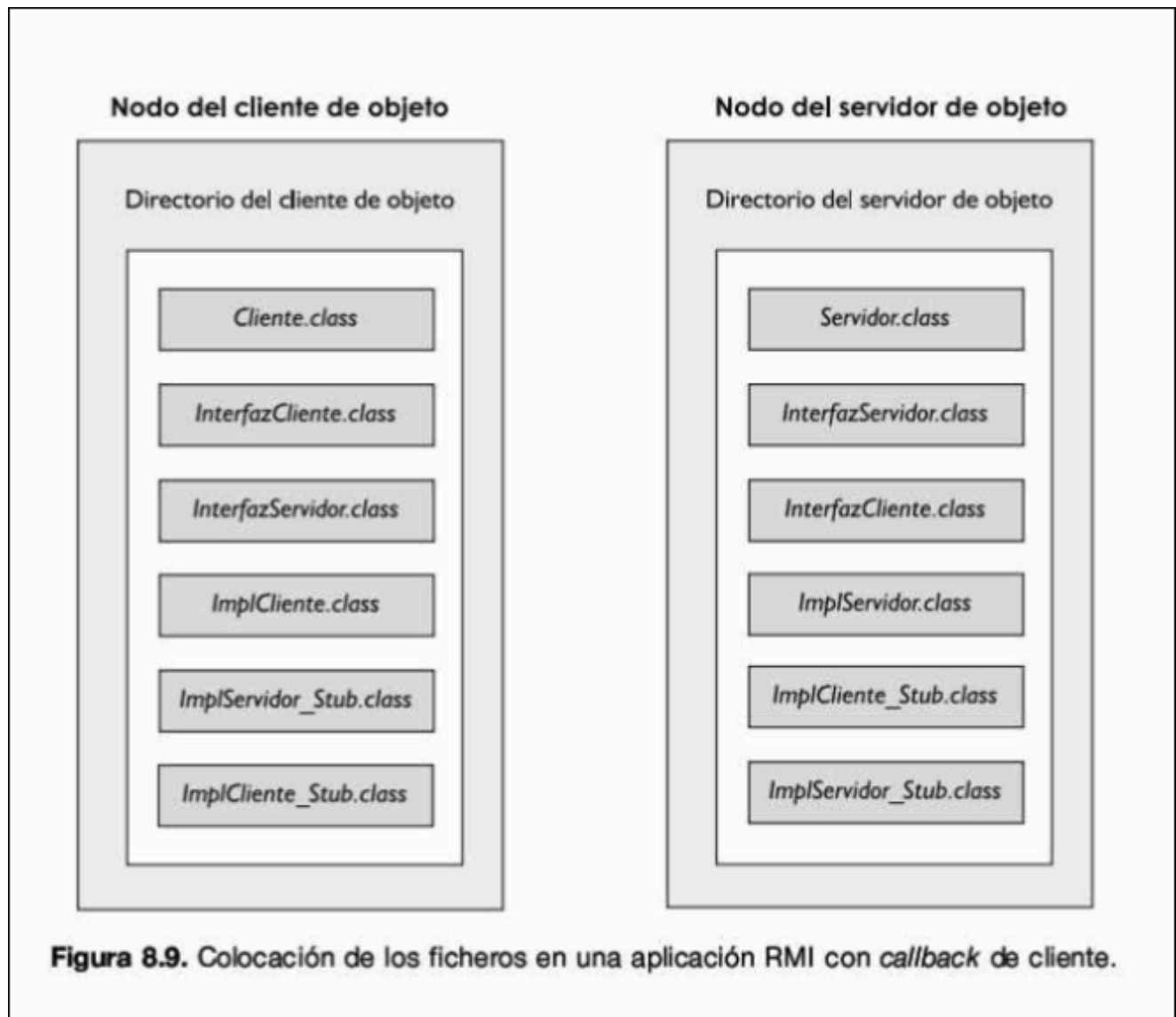


## Interacciones Cliente-Servidor con Callback



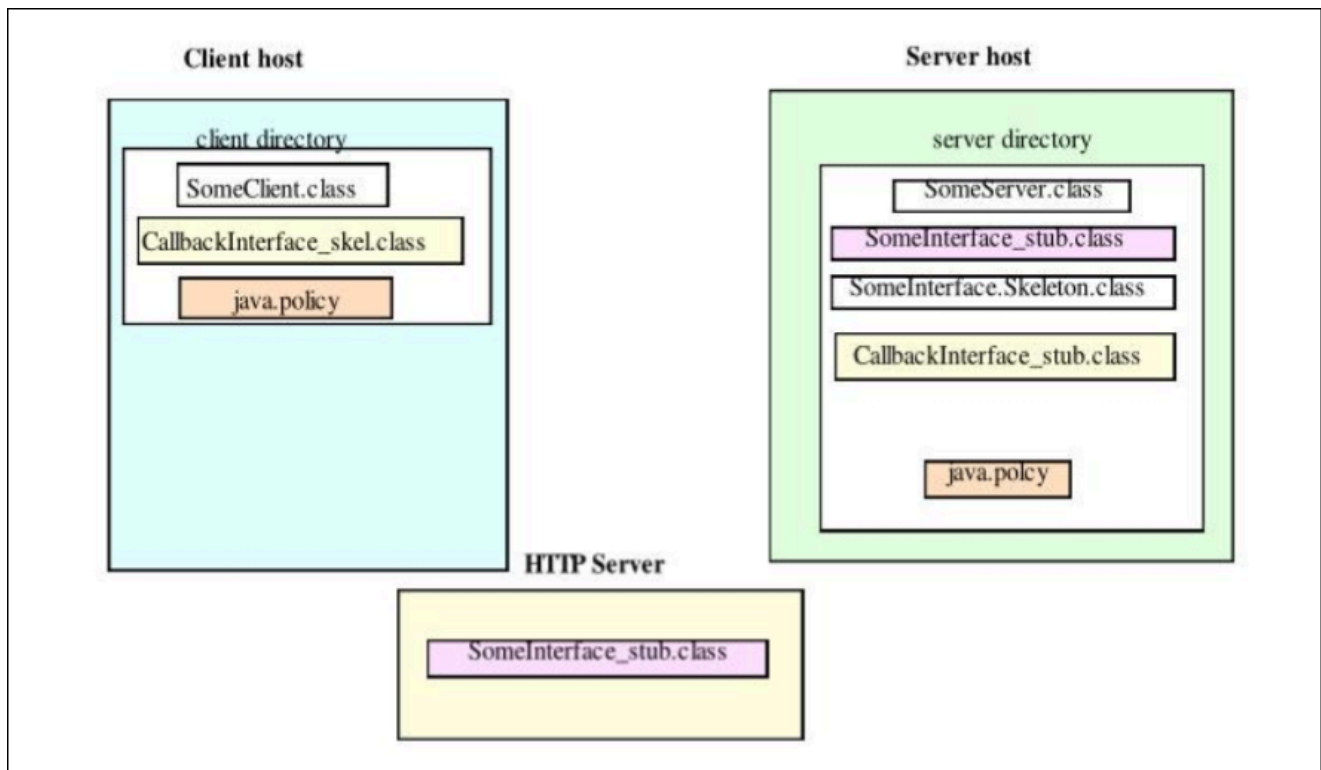
- El cliente busca el objeto interfaz en el registro RMI del host del servidor
- El registro RMI le devuelve una referencia remota del objeto interfaz
- EL cliente invoca al método de registrarse con callback gracias al stub del servidor. Le pasa una referencia remota de sí mismo y el servidor guarda la referencia en su lista de callback
- Gracias al stub del servidor el cliente interactúa con el skeleton de objeto interfaz para acceder a los métodos de este objeto
- Cuando el evento ocurre el servidor hace el callback a cada uno de sus clientes registrados por callback mediante stub y el skeleton del cliente.

## Ficheros de una aplicación Callback





## Colocación de los ficheros en una RMI Callback



## Interfaz RMI de Callback

El servidor proporciona un método remoto que permite al cliente registrarse para recibir callbacks. Para esto es necesario una interfaz remota para callback además de la interfaz de servidor. Esta interfaz especifica un método para aceptar llamadas de un servidor.

El programa cliente es una subclase de `RemoteObject` e implementa la interfaz callback (incluido el método de callback). El cliente se registra en su método main para el callback y el servidor invoca al método remoto del cliente ante la ocurrencia de un determinado evento.

## Algoritmo para construir aplicación RMI con Callback

### Lado servidor

- Crear un directorio donde se almacenen todos los ficheros generados por la aplicación
- Especificar la interfaz remota de servidor y compilarla para generar el fichero `.class` de la interfaz
- Construir la clase remota del servidor implementando el interfaz y compilarla hasta que no exista ningún error de sintaxis
- Utilizar **rmic** para procesar la clase del servidor y generar un fichero `.class` de **stub** y otro fichero `.class` de **skeleton**
- Si se requiere **stub downloading**, copiar el fichero **stub** al directorio apropiado del servidor HTTP.

- Activar el registro de RMI, si no estaba previamente activo
- Establecer la política de seguridad en el archivo **java.policy**
- Activar el servidor especificando:
  - El **codebase** si requiere **stub downloading**
  - nombre del servidor
  - fichero con la política de seguridad
- Obtener el **CallbackInterface**. Compilarlo con **javac** y usar **rmic** para genera el fichero de **stub** para la **callback**

## Lado Cliente

- Crear un directorio donde se almacenen todos los ficheros generados por la aplicación
- Implementar el programa cliente o applet y compilarlo para generar la clase cliente
- Si no está activo el **stub downloading**, copiar el fichero .class del **stub** correspondiente al interfaz del servidor a mano.
- Implementar la interfaz de **callback**. Compilarla usando **javac** y usando **rmic** generar los ficheros .class correspondientes al **stub** y el **skeleton**.
- Establecer la política de seguridad en el fichero **java.policy**
- Activar el cliente especificando:
  - nombre del servidor
  - fichero con la política de seguridad

## 4.5 Serialización y Envío de Objetos

A veces, resulta necesario pasar como argumento a un método de un objeto remoto tipos de datos complejos (objetos que hayamos creado nosotros). Es posible gracias a la **serialización**: es un mecanismo que encapsula el contenido de un objeto (código + datos) en una cadena de bytes que puede ser enviada a través de la red –canal de comunicación– o almacenada en una base de datos.

Java nos garantiza que será correcta la reconstrucción del objeto recibido y que funcionará sin problemas. Para pasar como argumento en una invocación remota un objeto es indispensable que dicho objeto sea **serializable**.

Ejemplo: **integración numérica** (programa iterativo para calcular una suma, aproximación numérica de integrales, regla del punto medio).