

Tema 3. Objetos distribuidos

SCS – Sistemas Cliente/Servidor
4º informática

<http://ccia.ei.uvigo.es/docencia/SCS>

octubre 2008

3.1 Modelo de objetos distribuidos

Objetivo: Extender el paradigma de orientación a objetos al desarrollo de sistemas distribuidos

- Modelo RPC se ajusta bien a arquitecturas *2-tier*
 - 2 papeles bien diferenciados $\left\{ \begin{array}{l} \text{cliente: realiza peticiones} \\ \text{servidor: recibe peticiones} \end{array} \right.$
 - La lógica de aplicación está en el cliente o en el servidor
- Difícil extender RPCs en desarrollo arquitecturas *3-tier* o *n-tier*
 - Componentes del sistema pueden actuar a la vez como clientes y servidores
 - En *n-tier* la lógica de aplicación se divide en componentes modulares reutilizables $\Rightarrow$ se ajustan mejor a un modelo orientado a objetos.
- **Finalidad:** Diseñar e implementar sistemas distribuidos que se estructuren como colecciones de componentes modulares (objetos) que puedan ser manejados facilmente y organizados en capas para ocultar la complejidad del diseño.
 - Posibilidad de invocar y pasar como argumento diferentes "*comportamientos*"
 - Posibilidad de aplicar patrones de diseño orientados a objetos en la implementación del sistema distribuido

Idea base: Extender el concepto RPC para su uso en un modelo de computación distribuida orientado a objetos

- Un objeto es una entidad identificable de forma única en todo el sistema
- Objetos encapsulan los recursos disponibles en el sistema distribuido
 - **Estado:** representado por los atributos del objeto
 - **Comportamiento:** representado por los métodos del objeto
- La "partición" de la aplicación y su distribución se basa en el uso de esos objetos
- La comunicación se lleva a cabo mediante la invocación de los métodos de esos objetos

3.1.1 Aspectos claves

(1) Definición interfaces vs. implementación métodos remotos

- **Interfaz remoto:** define el comportamiento del objeto remoto
 - Conjunto de definiciones de métodos accesibles por los demás objetos del sistema
 - Separación entre *definición* del interfaz remoto e *implementación* del comportamiento
- Alternativas:
 1. Mismo lenguaje para definición e implementación (ej.: Java RMI)
 2. Uso de un IDL (*interface definition language*) independiente del lenguaje de implementación (ej.: CORBA)
- Generación automática de elementos complementarios a partir de la definición del interfaz (*stubs*, código de partida para las implementaciones)

(2) Referencias a objetos remotos (referencia remota)

- Objetos remotos residen en la máquina que los crea
 - es donde almacenan su estado (atributos)
 - es donde ejecutan su comportamiento (métodos)
- En los procesos cliente se manejan *referencias remotas* a esos objetos
 - Determinan de forma unívoca la ubicación de un objeto en el sistema distribuido
 - $\approx$ "*puntero*" que referencia un objeto distribuido residente en otra máquina
- Información típica asociada a una *referencia remota*
 - dirección IP de la máquina donde reside el objeto remoto
 - n° de puerto de escucha asignado por el sistema de objetos
 - identificador único del objeto (único dentro de la máquina que lo crea)
- Normalmente su contenido no es accesible directamente
 - gestionado por los *stub* de los clientes y por el servicio de *binding*
- Uso de las *referencias remotas*
 - en la invocación remota de los métodos exportados por el objeto remoto
 - como argumento que se pasa en una invocación a otro objeto

(3) Invocación transparente de métodos remotos

- Modelos de invocación
 - **Invocación estática:** la interfaz del objeto remoto es conocida en tiempo de compilación
 - Basado en la generación/uso de representantes (*stubs* y *skeleton*)
 - Cliente invoca el *stub* correspondiente, que gestiona la invocación "real" del objeto remoto
 - **Invocación dinámica:** la interfaz del objeto remoto no es conocida en tiempo de compilación
 - Requiere una serie de pasos previos para *descubrir el interfaz*
- Paso de parámetros
 - Para *tipos básicos*: paso por valor (copia-restauración)
 - Para *objetos locales*: paso por valor de objetos serializados (copia-restauración)
 1. Serializar: convertir objeto en una cadena de bytes
 2. Transferir copia serializada del objeto
 3. Deserializar: reconstruir el objeto en la máquina de destino
 - Para *objetos remotos*: paso de *referencias remotas*
- Ubicación de objetos remotos (*binding*)
 - Necesidad de mecanismos para obtener *referencias remotas*
 - *binding* estático: en tiempo de compilación
 - ◇ la ubicación del objeto remoto es fija y conocida *a priori*
 - ◇ el compilador genera directamente la *referencia remota*
 - *binding* dinámico: en tiempo de ejecución
 - ◇ el objeto remoto tiene asociado un *nombre*
 - ◇ la *referencia remota* se recupera de un *servidor de nombres*
 - *binding* dinámico $\Rightarrow$ uso de *servicios de nombres*
 - Mantiene una B.D. con pares (NOMBRE OBJETO, REFERENCIA REMOTA)
 - Funciones básicas: $\left\{ \begin{array}{l} \text{registrar pares nombre-referencia (bind())} \\ \text{localizar referencia asociada a un nombre (lookup())} \end{array} \right.$
 - Nombres pueden ser $\left\{ \begin{array}{l} \text{transparentes a la ubicación} \\ \text{no transparentes a la ubicación} \end{array} \right.$
 - Espacio de nombres puede ser $\left\{ \begin{array}{l} \text{plano} \\ \text{jerárquico (en árbol)} \end{array} \right.$

3.2 Java-RMI (*Remote Method Invocation*)

Entorno de invocación de métodos remotos disponible en las máquinas virtuales Java (JVM) (desde la versión 1.1 de Java)

- Provee un modelo de objetos distribuidos similar al de Java
 - **Objetivo final:** conseguir que desarrollo de sistemas distribuidos se parezca lo más posible al desarrollo de aplicaciones Java locales
- Da soporte a las necesidades típicas de los sistemas basados en objetos distribuidos
 1. Localización de los objetos remotos

RMI contempla 2 mecanismos para obtener *referencias a objetos remotos*

 - Registro y localización en *rmiregistry* (servicio de nombre plano)
 - *referencias remotas* recibidas como parámetro o valor de retorno
 2. Comunicación con los objetos remotos
 - RMI oculta la comunicación con los objetos remotos
 - Para el programador la invocación de métodos remotos es idéntica a la invocación local
 3. Carga del código (*bytecodes*) de los objetos pasados como parámetro o valor de retorno
 - RMI proporciona mecanismos para la carga de los datos y el código de los objetos pasados como parámetro o valor de retorno en las invocaciones remotas
- Permite $\left\{ \begin{array}{l} \text{invocar métodos de objetos remotos} \\ \text{pasar referencias a objetos remotos como argumento} \end{array} \right.$
- Dos tipos de objetos en Java
 - objeto local:** sus métodos se invocan dentro de su propia máquina virtual (JVM)
 - invocados por el proceso que creó al objeto
 - objeto remoto:** sus métodos se invocan por procesos residentes en otras JVMs
 - la ejecución si se realiza en su propia JVM
- Para que un objeto ofrezca métodos fuera de su máquina virtual debe implementar una **interfaz remota**
 - interfaz que extiende `java.rmi.Remote`
 - recoge la declaración Java de los métodos accesibles desde máquinas virtuales externas

- Clientes interactúan con las interfaces remotas, no directamente con las clases que implementan los objetos remotos

Concepto clave: separación interfaz remoto vs. implementación

- Modelo basado en el uso de representantes (*stubs*) por parte de los clientes
 - *stubs* implementan el mismo *interfaz remoto* y trasladan las invocaciones al objeto remoto que las ejecutará realmente
 - *stubs* almacenan y manejan las *referencias remotas* a los objetos
- El servicio de nombres se encarga de registrar, almacenar y distribuir esas *referencias remotas*
 - Clientes reciben una copia serializada del *stub* al hacer *lookup()* en el servicio de nombres

3.2.1 Características generales de Java RMI

Objeto remoto: Objeto cuyos métodos pueden invocarse desde otras Máquinas Virtuales

- Descrito por una o más interfaces remotas en las que se declaran los métodos que pueden ser invocados por objetos de otras JVM

Características de los objetos remotos

- Las *referencias a objetos remotos* pueden pasarse como argumento o valor de retorno en cualquier invocación de métodos (local o remota)
- Las *referencias a objetos remotos* pueden convertirse (*cast*) a cualquiera de los interfaces remotos que implemente el objeto remoto al que referencia,
 - El operador *instanceof* puede ser usado para comprobar que interfaces remotos soporta un objeto
- Clientes interactúan con interfaces remotas, nunca con las clases de implementación directamente
- Objetos remotos deben cumplir ciertas restricciones adicionales respecto a algunos métodos genéticos de la clase `java.lang.Object` (*hashCode()*, *equals()*, *clone()*, *toString()*)
 - Normalmente serán subclases de `java.rmi.server.RemoteObject` que ya ofrece implementaciones adecuadas
- Mayores posibilidades de fallos en invocaciones remotas $\Rightarrow$ clientes tienen que gestionar nuevos tipos de excepciones (`java.rmi.RemoteException` y derivadas)

Paso de parámetros en invocaciones remotas

- Los objetos Java no remotos que actúen como *argumentos* o *valor de retorno* se pasan **por valor** (copia), no por referencia
 - Referencias a objetos locales sólo tienen sentido en la JVM que los creó ($\approx$ punteros)
 - Sólo se pueden pasar objetos locales que sean serializables
 - Se crea un nuevo objeto en la JVM que recibe el parámetro y él se copia el objeto serializado
 - Proceso *serialización-envío-deserialización* transparente (gestionado por RMI)

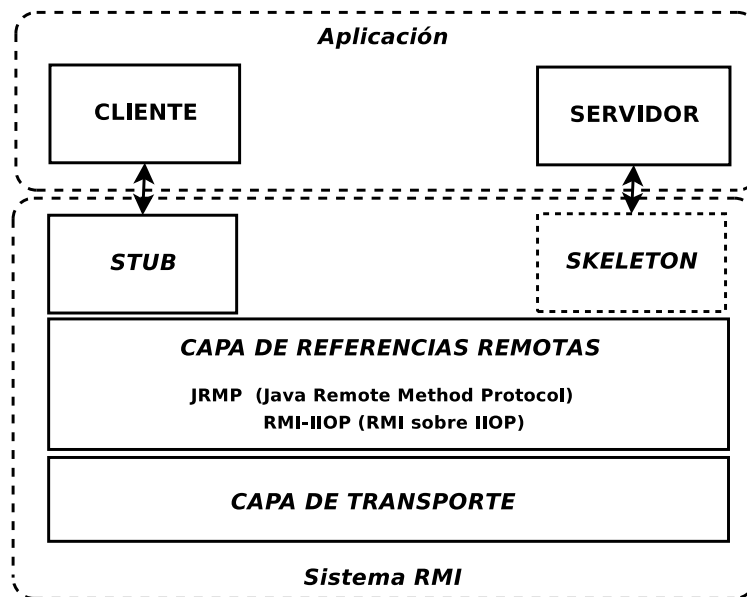
Nota: Objetos serializables implementan interfaz `java.io.Serializable`

- No define métodos, sirve para "marcar" una clase como serializable
- Serialización es genérica
 - Soportada por las clases `ObjectInputStream` y `ObjectOutputStream` mediante los métodos `readObject()` y `writeObject()`
 - Por defecto de cada objeto se guarda:
 - ◇ info. sobre la clase del objeto
 - ◇ valores de la variables no `static` ni `transient`
 - ◇ se serializan de forma recursiva los objetos referenciados
 - Puede reimplementarse $\left\{ \begin{array}{l} \text{readObject()} \\ \text{writeObject()} \end{array} \right.$ para modificar la serialización por defecto
- Los tipos básicos se pasan **por valor** (copia serializada)
- Los objetos remotos que actúen como argumento o valor de retorno se pasan por referencia (uso *referencias remotas*)
 - **Importante:** no se copia el objeto remoto
 - Se pasan copias serializadas del *stub* que a su vez conoce la *referencia remota*
 - *referencia remota* contiene toda la información necesaria para identificar al objeto remoto en la máquina donde reside
 - Creada al exportar el objeto remoto, contiene $\left\{ \begin{array}{l} \text{dir.IP máquina donde reside} \\ \text{nº de puerto de escucha} \\ \text{ID del objeto: único en su JVM} \\ \text{clase del objeto remoto} \end{array} \right.$
 - *Stub* implementa el mismo interfaz remoto que el objeto remoto pasado como parámetro
 - Para quien lo recibe se comporta como el objeto remoto
 - ◇ interfaz determina que métodos remotos se podrán invocar sobre él
 - El *stub* deriva las invocaciones al objeto remoto real, usando la info. contenida en la *referencia remota*

Diferencias entre objetos locales y objetos remotos RMI

	objeto local	objeto remoto
definición del objeto	en la propia clase Java	en un interfaz que extiende <i>java.rmi.Remote</i> (sólo el comportamiento exportado)
implementación del objeto	en la propia clase Java	en una clase Java que implemente el interfaz remoto
creación del objeto	constructor del objeto (operador <i>new()</i> en JVM local)	constructor del objeto en la máquina remota (cliente "obtiene" <i>referencias remotas</i> como valor de retorno en llamadas remotas o mediante consultas al servicio de nombres)
acceso al objeto	acceso directo mediante una variable de referencia	acceso indirecto a través de los métodos del <i>stub</i> local
referencias	variable de referencia que apunta al objeto en memoria	<i>referencia remota</i> gestionada el <i>stub</i> local, que contie info. para conectarse con el objeto remoto real
vida (validez)	vivo mientras exista al menos una var. de referencia que lo apunte	vivo mientras existan <i>referencias remotas</i> activas que no hayan expirado (timeout, caidas)
recolección de basura (<i>gc</i>)	liberación cuando no hay referencias apuntando al objeto	liberación cuando no hay ni referencias locales ni <i>referencias remotas</i> al objeto (colaboración <i>gc</i> local y remota)

3.3.2 Arquitectura de Java RMI



(a) Estructura Java RMI

Clientes y servidores. Objetos que implementan la lógica de la aplicación

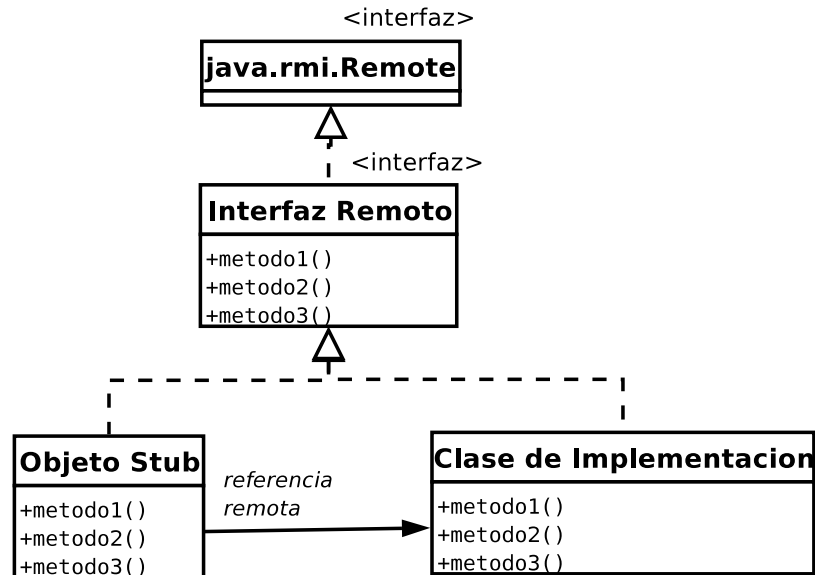
- **"Servidores"**: crean instancias de los objetos remotos y los ponen a disposición de los clientes para recibir peticiones
 - Crear objetos remotos
 - Crear referencias remotas y exportarlas
 - Nombrar y registrar los objetos remotos en el servicio de nombres
 - Espera de invocaciones
 - RMI mantiene activo el proceso servidor mientras existan *referencias remotas* a alguno de los objetos remotos creados por él (incluidas las referencias del servicio de nombres)
 - Cuando un objeto no tiene referencias desde ningún cliente, ni desde el servidor, queda disponible para la recolección de basura (*Garbage Collection*)

■ Clientes

- Declaran objetos de la interfaz remota y obtienen referencias a objetos remotos (*stubs*)
 - mediante $\left\{ \begin{array}{l} \text{consultas al servidor de nombres} \\ \text{referencias remotas recibidas como valor de retorno} \end{array} \right.$
- Invocación de métodos remotos (mediante el *stub*)
- Un mismo proceso/objeto puede actuar a la vez como servidor (exporta un objeto remoto) y como cliente (invoca un método de otro objeto remoto)

Capa de stubs. intercepta las llamadas de los clientes y las redirecciona al objeto remoto

- Objetos locales que actúan como representantes de los objetos remotos
 - Conocen la localización "real" del objeto remoto (interfaz *RemoteRef*)
 - dirección IP de la máquina
 - identificador único del objeto (*ObjID*)
 - puerto de escucha de peticiones en la máquina remota
 - clase de implementación del objeto remoto (+ localización de sus *bytecodes*)
 - Empaquetan/dempaquetan invocaciones+argumentos+resultados (método *invoke()*)



- Basada en el uso del patrón de diseño *Proxy*
 - Dos objetos implementan la misma interfaz remota (extienden `java.rmi.Remote`)
 - uno implementa el servicio y se ejecuta en el servidor
 - otro actúa como intermediario (*proxy*) para el servicio remoto y se ejecuta en el cliente
 - el *proxy* reenvía las peticiones de invocación al objeto remoto que las implementa
 - Cada objeto que implementa un interfaz remoto requiere una clase *stub*
- Referencia a objeto remoto por parte de cliente es realmente referencia local al objeto *stub*
- Desde la versión Java 1.2 no se usan clases *Skeleton*
 - La propia infraestructura RMI hace ese papel
- Desde la versión Java 1.5 no es necesario construir explícitamente las clases *stub*
 - Se generan bajo demanda y de forma automática en tiempo de ejecución a partir de las clases de implementación los objetos remotos
 - Uso de mecanismos de reflexión (*API Java Reflection*)
 - Recogida y procesamiento de información en tiempo de ejecución sobre las clases, métodos, etc
 - En versiones anteriores *stubs* (y *skeletons*) se generan con el compilador de interfaces `rmic` a partir de las clases de implementación.
 - Generación de *stubs* sólo necesaria si se atienden clientes de versiones anteriores a 1.5

- Los *stub* se construyen automáticamente o manualmente [`rmic`] a partir de las clases de implementación de los interfaces remotos
 - *stubs* extienden `java.rmi.server.RemoteStub`
 - contienen una referencia remota al objeto (`dir.IP+puerto+ObjID`) que implementa el interfaz `RemoteRef`
 - usan el método `invoke()` del interfaz `RemoteRef` para realizar la invocación al objeto remoto "real"

Capa de referencias remotas. interpreta y administra las referencias a objetos remotos

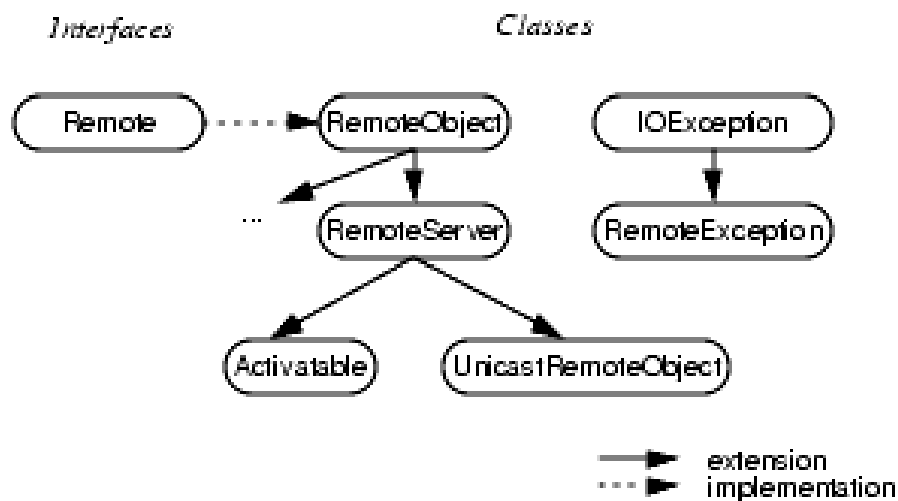
- Resuelve las referencias a objetos remotos gestionadas por los *stubs* para acceder e invocar los métodos de los respectivos objetos remotos
- Responsable de enviar/recibir paquetes empaquetados (*marshalling*) con peticiones/respuestas a través de la capa de transporte
- Internamente, se usa un *ObjID* (identificador de objeto) para identificar los *objetos remotos* de cada JVM exportados por el runtime RMI
 - Cuando se exporta un objeto remoto se le asigna (explícitamente o implícitamente) una identificación (gestionada por objetos que implementan el interfaz `java.rmi.server.RemoteRef`)
 - Información típica

{	dirección IP de la máquina que ejecuta la JVM identificador único del objeto dentro de la JVM puerto de escucha de la implementación
---	--
- Dos alternativas
 - JRMP (*Java Remote Method Protocol*): protocolo propietario definido por Sun para el manejo de las referencias remotas y la invocación de métodos
 - Es el usado por defecto
 - RMI-IIOP (RMI sobre IIOP): usa el protocolo IIOP (*Internet Inter-ORB Protocol*) de CORBA para la comunicación entre ORBs (*Object Request Brokers*) (ORB gestiona las referencias remotas entre objetos CORBA)
 - Usado en Java EE (*Java Enterprise Edition*)

Capa de transporte. ofrece conectividad entre las JVMs

- Soporta el intercambio de mensajes entre *stubs* y objetos remotos
- Basado en conexiones TCP/IP

(b) Paquetes, clases e interfaces básicos



- **java.rmi:** clases, interfaces y excepciones manejadas típicamente por los clientes
 - Interfaz `Remote`: interfaz base a implementar/extender por los objetos remotos
 - Excepción `RemoteException`: base de las excepciones lanzadas por los objetos remotos
 - Clase `Naming`: clase de apoyo para acceder al servicio de nombres *rmiregistry*
 - Clase `RMISecurityManager`: gestor de seguridad para aplicaciones RMI
- **java.rmi.server:** clases, interfaces y excepciones manejadas típicamente por los servidores
 - Clase `RemoteObject`: implementación de métodos básicos para objetos remotos (`hashCode()`, `equals()`)
 - Clase `RemoteServer`: superclase base para las implementaciones de objetos remotos
 - Clase `UnicastRemoteObject`: usada para crear y exportar *objetos remotos* con JRMP y obtener un *stub* que se comunicará con ese objeto remoto
 - Clase `RMIClassLoader`: cargador de clases (*bytecodes*) desde orígenes remotos
 - necesario para deserializar *stubs* y objetos serializados recibidos como parámetro
 - Clase `ObjID`: clase que encapsula un identificador único para objetos
 - Clase `RemoteStub`: superclase de la que heredan las implementaciones de *stubs*
 - Interface `RemoteRef`, `ServerRef`: gestión de *referencias remotas*
- **java.rmi.registry:** clases, interfaces y excepciones necesarios para registrar y localizar objetos remotos
 - Interface `Registry`: definición del interfaz remoto de *rmiregistry*
 - Clase `LocateRegistry`: clase con métodos para creación y consulta de servidores de nombres
- **java.rmi.dgc:** clases, interfaces y excepciones para dar soporte a la recolección de basura distribuida
- **java.rmi.activation:** clases, interfaces y excepciones para dar soporte a la activación de objetos remotos
 - Clase `Activatable`: usadas en la creación y exportación de objetos remotos activables

(c) Elementos adicionales

Servicio de nombres. Asocia nombres lógicos a los objetos remotos exportados (*stubs*)

- Servidor: asocia un nombre al objeto remoto (registro) [bind()]
- Cliente: obtiene una referencia al objeto remoto (*stub*) a partir de ese nombre [lookup()]

Objetivo: ofrecer transparencia de ubicación

- Evitar que el cliente tenga que ser recompilado si el objeto remoto pasa a ejecutarse en otra máquina virtual (JVM)

Almacena y mantiene asociación entre $\left\{ \begin{array}{l} \text{nombre de objeto remoto} \\ \text{referencia remota (*stub*) serializada} \end{array} \right.$

Funcionamiento del proceso bind()

- Toma un objeto de la clase que implementa el interfaz Remote
- Si es necesario se construye internamente una instancia de RemoteStub para ese objeto $\left\{ \begin{array}{l} \text{automáticamente [Java 1.5]} \\ \text{desde clase *stub* generada por rmic)} \end{array} \right.$
- Serializa esa instancia de RemoteStub y la envía al registro asociada a un nombre

En RMI se ofrece el *RMIregistry*: servidor que implementa una interfaz de registro/localización de nombres de objetos sencilla

- Servidor *rmiregistry*

- Contiene un objeto remoto que implementa el interfaz remoto

`java.rmi.registry.Registry`

Métodos: $\left\{ \begin{array}{l} \text{void bind(String name, Remote obj)} \\ \text{String[] list()} \\ \text{Remote lookup(String name)} \\ \text{void rebind(String name, Remote obj)} \\ \text{void unbind(String name)} \end{array} \right.$

- Las invocaciones a `bind()` y `lookup()` son invocaciones remotas RMI

- Por defecto escucha peticiones en el puerto 1099
- Debe estar en ejecución antes de que sean lanzados los servidores y clientes

- Servicio de nombres plano (sin jerarquía)

- No persistente (sólo mantiene pares [NOMBRE, OBJETO] durante la ejecución del servidor)

- Servidores y clientes acceden al servicio *RMIregistry* empleando los métodos estáticos de las clases `java.rmi.Naming` ó `java.rmi.registry.LocateRegistry`

Clase `java.rmi.Naming` (implementa `java.rmi.registry.Registry`)

- Clase de conveniencia con métodos estáticos para acceso al servicio de nombres *RMIregistry*
- Sintaxis de nombrado basada en urls: `url://maquina:puerto/nombre`
- Emplea los métodos de `java.rmi.registry.LocateRegistry`

Nota: Existe la posibilidad de utilizar otros servidores de nombres genéricos para almacenar la asociación (NOMBRE, REFERENCIA REMOTA) empleando el API de Java JNDI (*Java Naming and Directory Interface*)

- JNDI es independiente de la infraestructura de servicio de nombrado empleada
- Posibilidad de usar servicios de nombres jerárquicos y persistentes (por ejemplo: servidor LDAP)
- Utilizado en la especificación Java EE (*Java Enterprise Edition*)

Cargador de clases. Necesidad de carga dinámica de clases

- Cliente debe tener acceso a los *bytecodes* (ficheros *.class*) de las clases que implementan los *stubs* (referencias remotas)
 - generadas en el servidor (automáticamente o mediante *rmic*)
 - almacenadas (en forma serializada) junto a su *nombre* en el registro RMI
- Cliente (y servidor) debe tener acceso a los *bytecodes* (ficheros *.class*) de las clases cuyos objetos serializados se pasan como parámetro o valor de retorno en las invocaciones remotas

Opción 1. incluir (copiar) esas clases (ficheros *.class*) en el CLASSPATH del cliente

Opción 2. el cargador de clases del cliente las descargará de forma automática desde la localización que indique el servidor

- Uso de la propiedad *java.rmi.codebase* de la JVM:
 - { url apuntando a la localización
 - { análogo al CLASSPATH

```
$ java -Djava.rmi.codebase=file://directorio_clases/ Servidor
```

```
$ java -Djava.rmi.codebase=http://url_clases/ Servidor
```

```
$ java -Djava.rmi.codebase=ftp://url_clases/ Servidor
```

- Ese *codebase* se almacena en cada clase *stub* al ser serializada para ser incluida en el RMIregistry

Gestor de seguridad. La necesidad de realizar carga remota de clases (*bytecodes*) en RMI abre una potencial brecha de seguridad.

- Posibilidad de carga (y ejecución) de *stubs* malignos

RMI no descargará clases de localizaciones remotas si no se ha establecido un *gestor de seguridad*

- RMI ofrece el gestor de seguridad (`java.rmi.RMISecurityManager`)
 - Interpreta e implementa las políticas de seguridad
 - Permite especificar si se permitirá o no realizar una operación potencialmente peligrosa antes de realizarla

- Establecimiento del gestor de seguridad en el cliente

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

- La política de seguridad se configura en un fichero de propiedades (*properties*)

- Por defecto se utiliza el fichero `java.policy` incluido en el JDK
- Se puede establecer otra política indicando en la propiedad `java.security.policy` de la JVM cliente un fichero *properties* distinto

```
$ java -Djava.security.policy=nueva.policy EjemploCliente
```

- Ejemplo: fichero `nueva.policy`

```
grant { // permite conectar con el puerto 80 (descarga http)
    permission java.net.SocketPermission "*:1024-65535", "connect,accept";
    permission java.net.SocketPermission "*:80", "connect";
};
```

3.3.2 Desarrollo de aplicaciones RMI

Secuencia de pasos típicos {

- (1) definir interfaz remoto
- (2) crear clase que lo implemente
- (3) crear programa servidor
- (4) crear programa cliente

(a) Definición de interfaces remotos Interfaces remotos deben extender directamente o indirectamente el interfaz `java.rmi.Remote`

- No define métodos
- Sirve como "marcador" de una clase que define objetos remotos

Definición de métodos

- Deben declarar la excepción `java.rmi.RemoteException`
- Los tipos de los parámetros y valor de retorno del método pueden ser
 - tipos básicos → se pasan por valor (copias serializadas)
 - objetos locales serializables → se pasan por valor (copias serializadas)
 - objetos remotos → se pasan por referencia
 - cuando cliente invoca método remoto que devuelve referencia a objeto remoto → cliente recibe una instancia de su *stub*
 - cuando cliente invoca método remoto pasando referencia a objeto remoto → servidor recibe una instancia de su *stub*
- Sólo los métodos definidos en interfaz remota estarán disponibles para su invocación desde clientes

Ejemplo: `Calculadora.java`

```
public interface Calculadora extends java.rmi.Remote {
    int sumar(int a, int b) throws java.rmi.RemoteException;
    int restar(int a, int b) throws java.rmi.RemoteException;
    int multiplicar(int a, int b) throws java.rmi.RemoteException;
    int dividir(int a, int b) throws java.rmi.RemoteException;

    int getContadorPeticiones() throws java.rmi.RemoteException;
}
```

Compilación: `$ javac Calculadora.java`

(b) Implementación del interfaz remoto

Clase de implementación debe de implementar **todos** los métodos definidos en la interfaz remota

Dos tipos de objetos remotos en RMI

objetos remotos transitorios. El servidor que los crea y exporta debe estar en ejecución durante toda su "vida"

- Residen siempre en memoria
 - Viven mientras viva el servidor o hasta que el servidor los destruya
- Heredan de `UnicastRemoteObject` o son exportados con los métodos estáticos `UnicastRemoteObject.exportObject()`

objetos remotos activables. Objetos que tienen "vida" independiente del servidor que los crea y exporta

- Sólo residen en memoria durante el tiempo en que son utilizados
- Heredan de `Activatable` o son exportados con `Activatable.exportObject()`
- Cuando se invoca uno de sus métodos por un cliente se *activan* (vuelven a estar disponibles en memoria)
- Se pueden *desactivar* (salen de memoria pero no se destruyen) cuando dejan de ser usados por los clientes
- Uso de los mecanismos de serialización para "guardar" el estado del objeto en disco al desactivarlo y para "recuperar" su estado al activarlo
- **Finalidad:** ahorro de recursos en aplicaciones que deban gestionar grandes cantidades de objetos remotos
- Hacen uso del servidor de objetos activables `rmid`

Exportación de objetos remotos: Proceso mediante el que se notifica a la infraestructura RMI la existencia de un objeto remoto

- Objeto remoto queda a la espera de llamadas en un puerto anónimo (> 1024)
 - Tareas realizadas:
 - Creación del puerto de escucha hacia el que el *stub* enviará las peticiones de invocación de métodos remotos
 - Creación de una instancia de un objeto *stub* asociado a cada objeto remoto
 - automáticamente (java 1.5)
 - a partir de la clase *stub* generada con el compilador `rmic`
- El objeto *stub* contiene una *referencia remota* al objeto exportado
- *referencia remota* implementa `java.rmi.server.RemoteRef` y contiene info. que identifica al objeto de forma única (dir. IP+puerto+ObjID)
- Después de la exportación el objeto remoto queda listo para recibir invocaciones

En este tema trabajaremos con objetos remotos transitorios

Hay **dos opciones** posibles para implementar los objetos remotos

1. Clase de implementación extiende `UnicastRemoteObject`

- Esquema más sencillo para implementar objetos remotos con RMI
- `UnicastRemoteObject` proporciona todo lo necesario para que un objeto remoto se integre en la infraestructura RMI
- El propio objeto, en su constructor, se exporta a si mismo (*exportación implícita*)
 - Las instancias de `UnicastRemoteObject` almacenan referencias a su propio *stub*
- **Importante:** hay que definir un constructor de la clase de implementación que llame a `super()` (constructor de `UnicastRemoteObject`) para hacer efectiva la auto-exportación
 - Además, ese constructor debe declarar la excepción `RemoteException`

Ejemplo: `CalculadoraImpl.java` (opción 1)

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class CalculadoraImpl extends UnicastRemoteObject implements Calculadora {
    private int contadorPeticones;
    public CalculadoraImpl() throws RemoteException {
        super();
        contadorPeticones = 0;
    }
    public int sumar(int a, int b) throws RemoteException {
        contadorPeticones++;
        return(a+b);
    }
    ...
}
```

Compilación: `$ javac CalculadoraImpl.java`

2. Clase de implementación extiende cualquier otra clase

- Se define como una clase normal, único requisito: implementar métodos de interfaz remota
- Se usan los métodos estáticos de `UnicastRemoteObject` para exportarla *explícitamente*
 - Llamados por el servidor que crea el objeto o por el propio objeto en su constructor o en algún método de inicialización propio.
 - `UnicastRemoteObject.exportObject(obj)`: lo exporta en un puerto aleatorio
 - `UnicastRemoteObject.exportObject(obj,puerto)`: lo exporta en el puerto indicado

Ejemplo: `CalculadoraImpl.java` (opción 2)

```
import java.rmi.RemoteException;

public class CalculadoraImpl implements Calculadora {
    private int contadorPeticiones;
    public CalculadoraImpl() throws RemoteException {
        contadorPeticiones = 0;
    }
    public int sumar(int a, int b) throws RemoteException {
        contadorPeticiones++;
        return(a+b);
    }
    ...
}
```

Compilación: `$ javac CalculadoraImpl.java`

Paso extra: Generación de clases *stub* (y *skeleton*)

- Uso del compilador `rmic` con el fichero `.class` de la clase de implementación del interfaz remoto

```
$ rmic CalculadoraImpl (genera CalculadoraImpl_Stub.class)
```
- Sólo necesario si los clientes que usarán el objeto remoto se compilaron con versiones anteriores a Java 1.5
 - Desde Java 1.5 la infraestructura RMI genera los objetos *stub* adecuados en tiempo de ejecución usando los mecanismos de reflexión de java (API *Java Reflection*)

(c) Implementación del "servidor"

Tareas típicas

1. Crear los objetos remotos que implementan las interfaces remotas
2. *Exportar* objetos remotos $\left\{ \begin{array}{l} \text{implícitamente: extienden } \text{UnicastRemoteObject} \\ \text{explícitamente: métodos estáticos de } \text{UnicastRemoteObject} \end{array} \right.$
3. Registrar las referencias a objetos remotos (los *stubs*) resultantes de la exportación en el servidor de nombres (*rmiregistry*) asociándoles un nombre (métodos `bind(nombre, stub)` o `rebind(nombre, stub)`)

Ej.: EjemploServidor.java (opción 1: CalculadoraImpl extiende UnicastRemoteObject)

```
import java.rmi.Naming;
public class EjemploServidor {
    public static void main(String[] args){
        try {
            Calculadora calcStub = new CalculadoraImpl();
            Naming.rebind("rmi://localhost/Calculadora", calcStub);
            System.out.println("Servidor en espera ... ");
        } catch (Exception e) {
            System.out.println("Error en servidor:"+e);
        }
    }
}
```

Ej.: EjemploServidor.java (opción 2: CalculadoraImpl no extiende UnicastRemoteObject)

```
import java.rmi.Naming;
import java.rmi.server.UnicastRemoteObject;
public class EjemploServidor {
    public static void main(String[] args){
        try {
            Calculadora calc = new CalculadoraImpl();
            Calculadora calcStub =
                (Calculadora) UnicastRemoteObject.exportObject(calc, 0);
            Naming.rebind("rmi://localhost/Calculadora", calcStub);
            System.out.println("Servidor en espera ... ");
        } catch (Exception e) {
            System.out.println("Error en servidor:"+e);
        }
    }
}
```

Compilación: `$ javac EjemploServidor.java`

(d) Implementación del cliente

Tareas típicas

1. Establecer el gestor de seguridad `RMISecurityManager` (opcional)
 - Necesario si se van a utilizar objetos remotos residentes en una máquina física distintaEstablecerlo antes de cualquier invocación RMI (incluida llamada a `RMIregistry`)

```
System.setSecurityManager(new RMISecurityManager());
```
2. Obtener las referencias al objeto remoto (*stubs*) consultando el servicio de nombres (*rmiregistry*) (método `lookup(nombre)`)
3. Invocar los métodos remotos llamando a los métodos del *stub* con los parámetros precisos

Ejemplo: EjemploCliente.java

```
import java.rmi.Naming;

public class EjemploCliente {
    public static void main(String[] args) {
        try {
            String urlCalculadora = "rmi://" + args[0] + "/Calculadora";
            Calculadora calcStub = (Calculadora) Naming.lookup(urlCalculadora);

            calcStub.sumar(30, 5);
            calcStub.restar(30, 5);
            calcStub.multiplicar(30, 5);
            calcStub.dividir(30, 5);
            System.out.println("Contador peticiones: " +
                               calcStub.getContadorPeticiones());
        } catch (Exception e) {
            System.out.println("Error: " + e);
        }
    }
}
```

Compilación: `$ javac EjemploCliente.java`

Pasos para la ejecución

1. Previamente debe estar en ejecución el *rmiregistry*

```
$ rmiregistry &
```

2. Lanzar el servidor (quedará a la espera)

```
$ java EjemploServidor &
```

Opcional: establecer la propiedad *java.rmi.codebase*

- Necesario si cliente no dispone de los *bytecodes*
 - de las clases *stub* (o si se usan *stub* autogenerados de java 1.5)
 - de las clases de los parámetros que se intercambian de forma serializada
- Indicar una *url* donde estarán disponibles esos *bytecodes* para su descarga por el cargador de clases del cliente

```
$ java -Djava.rmi.codebase=file://path_directorio_clases/ EjemploServidor &
```

3. Lanzar el cliente (en otra máquina o terminal)

```
$ java EjemploCliente localhost
```

Opcional: establecer la propiedad *java.security.policy* si la política por defecto no es adecuada (objetos remotos en otra máquina)

```
$ java -Djava.security.policy=fichero.policy EjemploCliente host_remoto &
```