



1. Introducción
2. Problemas de diseño
3. Implementación de la LPR
4. Implementación de la IMR
5. *Java RMI*



Introducción y objetivos



- Programa distribuido:
 - usuarios interactuando con aplicaciones
 - las aplicaciones son clientes de los servicios
 - los servidores son clientes de otros servicios
- Patrón de comunicación: cliente-servidor
 - basado en un protocolo solicitud-respuesta
 - sincronización: el cliente siempre espera
- La invocación remota integra a los programas distribuidos
 - con los lenguajes de programación
 - incluso los no distribuidos



La Lógica de Mediación (*middleware*)




- Proporciona:
 - un modelo de programación de alto nivel
 - transparencia de ubicación e
 - independencia de:
 - protocolos, sistemas operativos, hardware



- IR modelada a imagen de IL, pero:
 - se ejecuta en **distinto proceso** (ordenador)
- Programa = conjunto de módulos que se pueden comunicar
- En el nivel de IR, **servicio = módulo**:
 - exporta un conjunto de procedimientos o métodos, que
 - proveen de un **conjunto completo de operaciones**
 - sobre una clase de recursos
 - una **interfaz** específica qué operaciones y variables son accesibles desde otros módulos



- Se intenta mantener la semántica de la invocación local
 - pero el entorno es radicalmente diferente 
- **Movimiento** de los argumentos
 - argumentos de *entrada* y/o *salida*
 - argumentos de entrada = por valor
 - ¿argumentos por referencia?
 - Necesidad de un lenguaje de definición de interfaz
 - un módulo que corre en un proceso no puede acceder a las variables de un módulo de otro proceso
 - no se pueden pasar punteros



- a) IR integrada en un **lenguaje específico**
 - que incluye la notación para definir interfaces
 - Ej.: *Java RMI* (Sun), *Argus* (MIT)
 - el lenguaje se puede ocupar de los especiales problemas de la IR

- b) Uso de un **lenguaje de definición de interfaz** (LDI)
 - sobre lenguajes convencionales
 - *Sun RPC* → LDI para LPR
 - *CORBA IDL* → LDI para IMR
 - Ventaja: no está atado a un entorno particular de lenguaje



- La definición de interfaz especifica:
 - las características de los métodos:
 - provistos por el/los servidor(es)
 - accesibles a sus clientes
- Es decir:
 - los **nombres** de los métodos
 - los **tipos** de sus argumentos
 - el **movimiento**: entrada y/o salida
 - qué argumentos van en las solicitudes y en las respuestas



```
// En el fichero Persona.idl

struct Persona {
    string nombre;
    string lugar;
    int anho;
};

interface ListaPersonas {
    readonly attribute string nombreLista;
    void metePersona(in Persona p);
    void damePersona(in string nombre, out Persona p);
    long numero();
};
```



La comunicación entre objetos distribuidos



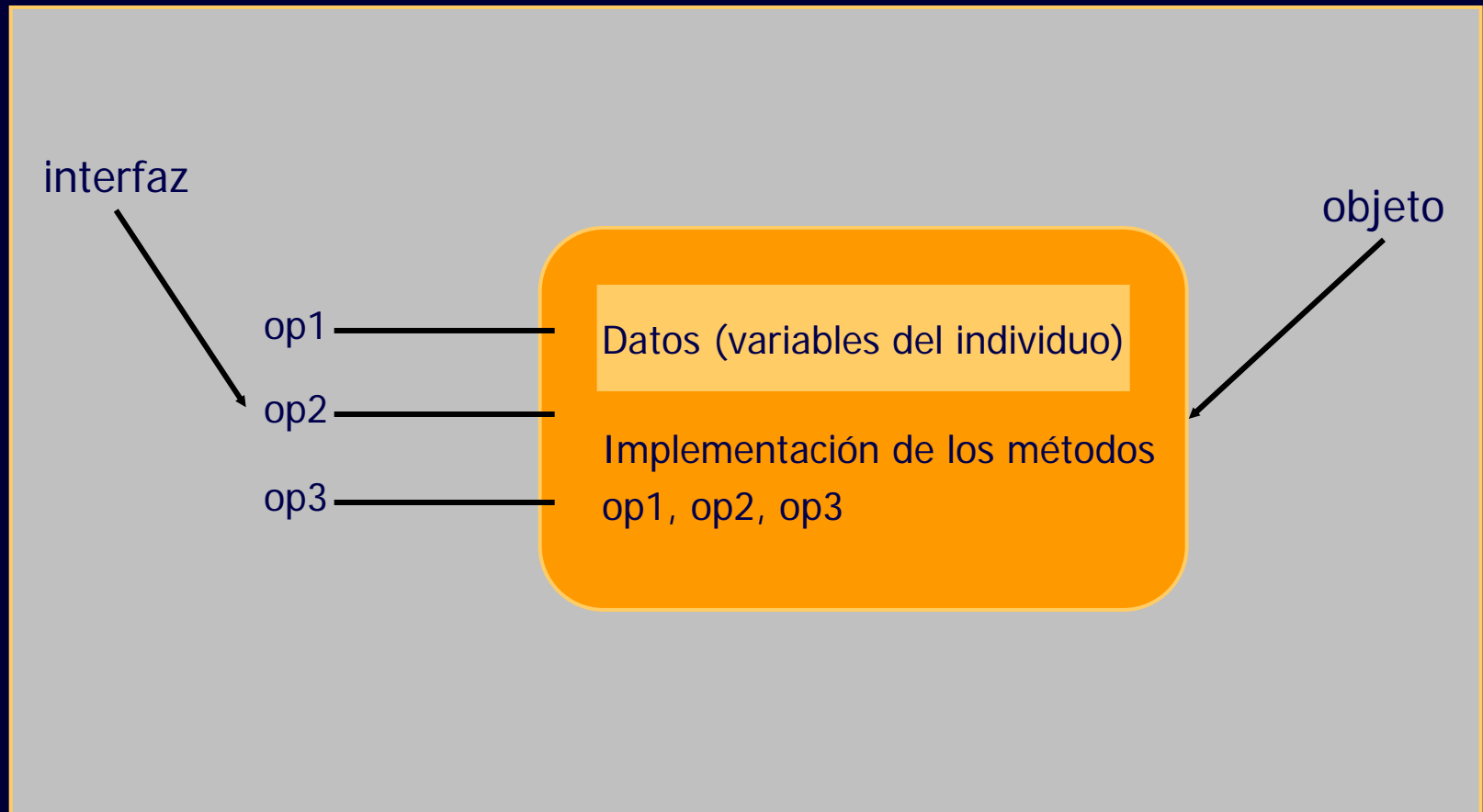
- Un programa escrito en un lenguaje orientado a objetos (*Java*, *C++*, ...) consiste en:
 - una colección de **objetos** que interactúan. Cada uno:
 - da un servicio especificado por su **interfaz**
- Comunicación entre objetos: envío de **mensajes**
 - mensaje = solicitud de realización de un **método**
 - el receptor determina **cómo** realizarlo
- Los objetos encapsulan:
 - sus **datos**: definen su estado
 - sus **métodos**: definen su comportamiento

⇒ se puede alterar el formato de los datos y la implementación de los métodos de forma transparente.





Un objeto y su interfaz





- **RO** = referencia a un objeto
 - a través de ellas se accede a los objetos
 - permite: ponerlo como destino de un mensaje, comprobar si 2 *ROs* se refieren al mismo objeto, pasarlo como argumento ...
- **Acción** = invocación + ejecución + retorno
 - la ejecución puede causar 2 efectos:
 - cambiar el estado del objeto e
 - invocar otros métodos de otros objetos
 - ⇒ una acción es una **cadena** de invocaciones relacionadas
 - la interpretación del mensaje (qué operación y sus argumentos) depende de la definición del objeto



- Clase = conjunto potencialmente infinito de individuos similares
- Objeto = un **individuo** (*instance*) de una clase
- Una clase define:
 - las signaturas de los métodos
 - las variables del individuo y la implementación de los métodos
 - cómo crear nuevos individuos (*new*)
- Todos los individuos comparten el código
 - pero cada uno tiene sus variables con sus valores



- Ante un error o condición inesperada en tiempo de ejecución:
 - el tratamiento *manual* perjudica la claridad
- **Excepciones** = mecanismo automatizado para tratar condiciones de error de forma más limpia:
 - cualquier bloque de código puede declarar que **lanza** (*throw*) determinadas excepciones
 - la cabecera de cada método lista las excepciones que puede lanzar
 - en cualquier bloque de código se pueden **capturar** (*catch*) excepciones
 - el control pasa a otro bloque de código



- Detecta cuándo un individuo ya no es accesible y libera la memoria que ocupaba
- si no existe (*C++*), operación manual (*delete*)
- si existe (*Java*), evita graves (y frecuentes) errores de programación



Arquitectura de los sistemas de objetos distribuidos

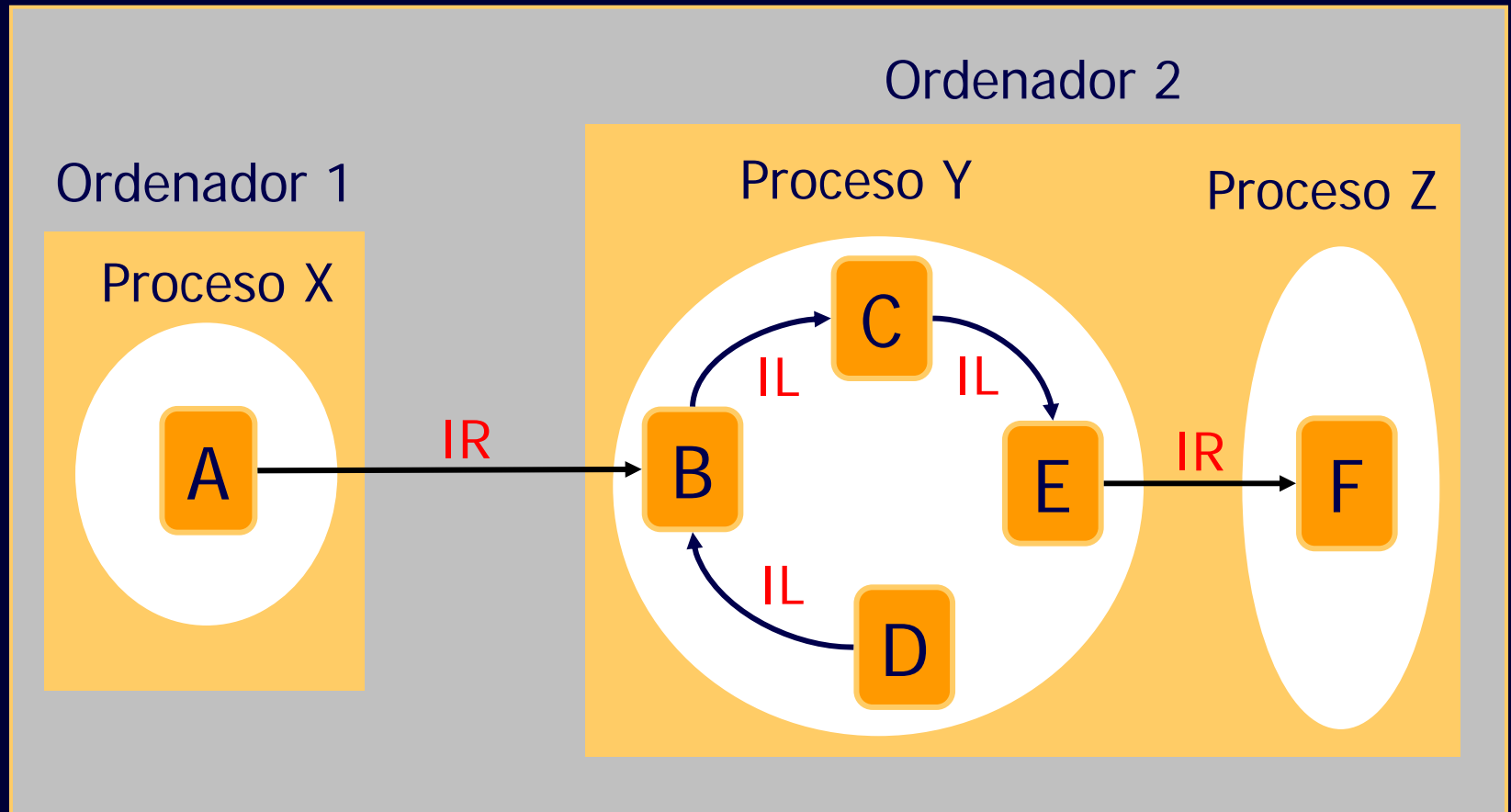
- Los sistemas de objetos distribuidos suelen adoptar la arquitectura **cliente/servidor**:
 - los servidores gestionan los recursos en la forma de objetos
 - los clientes invocan sus métodos a través de delegados (*proxies*)
 - cadenas de invocaciones relacionadas:
 - un objeto de un servidor puede ser cliente de otros objetos



- SD \Rightarrow distribución física de los objetos
- Estado de un objeto = valores de las variables del individuo
 - accesible sólo a través de sus métodos
- **ROR** = Referencia a un objeto remoto:
 - identificador global de un objeto en un SD
 - se pueden asignar a variables, pasar como argumentos, comparar, usar para conectar unos objetos a otros, ...

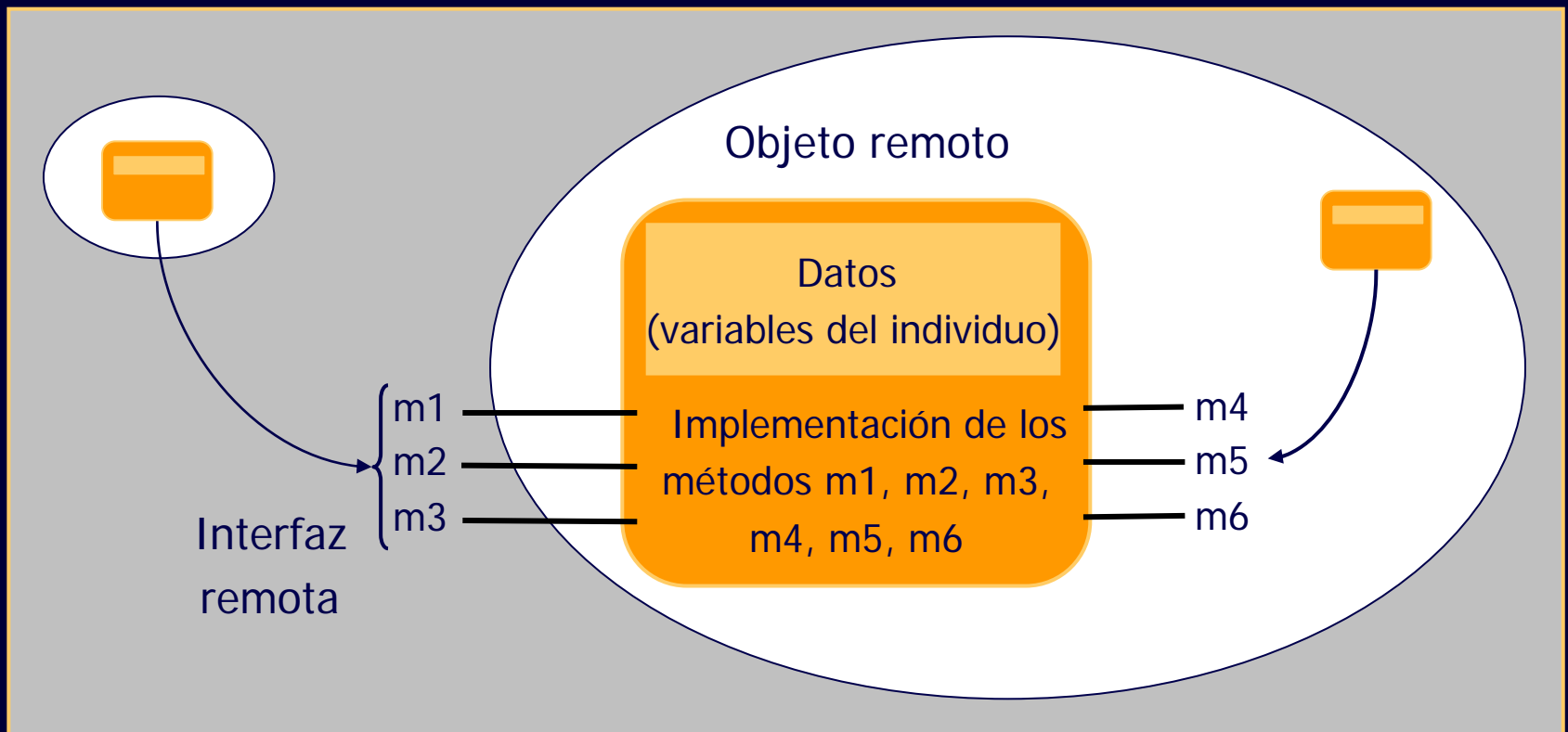


Invocaciones locales y remotas





- La Interfaz remota de un objeto especifica cuáles de sus métodos se pueden invocar remotamente







La recolección de residuos y excepciones en SD

- La recolección es difícil de aplicar a objetos distribuidos:
 - puede haber referencias al objeto en diferentes sitios
- La recolección distribuida implica la colaboración entre la recolección local y
 - un módulo añadido, basado en una cuenta de referencias
- En una invocación remota pueden surgir nuevas excepciones:
 - las relacionadas con la distribución: temporizaciones, ...



Cuestiones de diseño en la invocación remota



- *Semántica* de invocación = fiabilidad de una IR:
 - desde el punto de vista del llamante 
- Semántica *tal-vez* (maybe)
 - temporización y salida
 - si cayó el servidor o se perdió la solicitud: método no ejecutado
 - si se perdió la respuesta: método ejecutado
 - el cliente *no sabe*
 - generalmente no es aceptable 



- Son consecuencia de las decisiones de diseño en la implementación del módulo de comunicaciones:

<i>Garantías de entrega</i>			<i>Semánticas de IR</i>
<i>Reintentar la solicitud</i>	<i>Filtrado de duplicados</i>	<i>Re-ejecutar o retransmitir</i>	
No	--	--	<i>Tal-vez</i>
Sí	No	Re-ejecutar el método	<i>Al-menos-una-vez</i>
Sí	Sí	Retransmitir la respuesta	<i>Como-mucho-una-vez</i>





- Semántica *al-menos-una-vez* (at-least-once)
 - el cliente reintenta al vencer la temporización
 - pero el servidor no filtra los duplicados
 - el cliente recibe:
 - o una excepción (“posible” fallo)
 - o una respuesta correcta
 - el cliente no sabe *cuántas veces* se ha ejecutado
 - sólo es aceptable si *todas* las operaciones del servidor son *idempotentes*



- Semántica *como-mucho-una-vez* (at-most-once)
 - reintento + filtrado de duplicados
 - historial + retransmisión de respuesta
 - para operaciones no idempotentes se necesitaría semántica *exactamente-una-vez* = imposible
 - como-mucho-una-vez:
 - si el servidor no cae y el cliente recibe una respuesta
 - el método se ha ejecutado exactamente una vez
 - en caso contrario: excepción
 - el método se ha ejecutado cero o una vez





- ¿Hasta qué punto las IR idénticas a las IL?
 - Los métodos locales disponen de un tiempo ilimitado
 - Los métodos remotos son más **vulnerables a fallos** que los locales
 - la red, otro ordenador, otro proceso
 - deben manejar errores nuevos
- **CORBA**: a medio camino
 - IR lanzan excepciones ante problemas de comunicación
 - los clientes deben saber tratarlas
- Los LDI pueden permitir especificar la semántica de llamada



- Práctica actual:
 - IR misma sintaxis que IL, pero
 - las interfaces deben expresar la diferencia entre objetos locales y remotos
- *Java RMI*: los objetos que admiten invocaciones remotas
 - implementan interfaces que extienden a la interfaz *Remote*
 - lanzan *RemoteException*
 - Además, la implementación de un objeto que puede ser accedido remotamente debe ocuparse de mantener la consistencia ante accesos concurrentes.



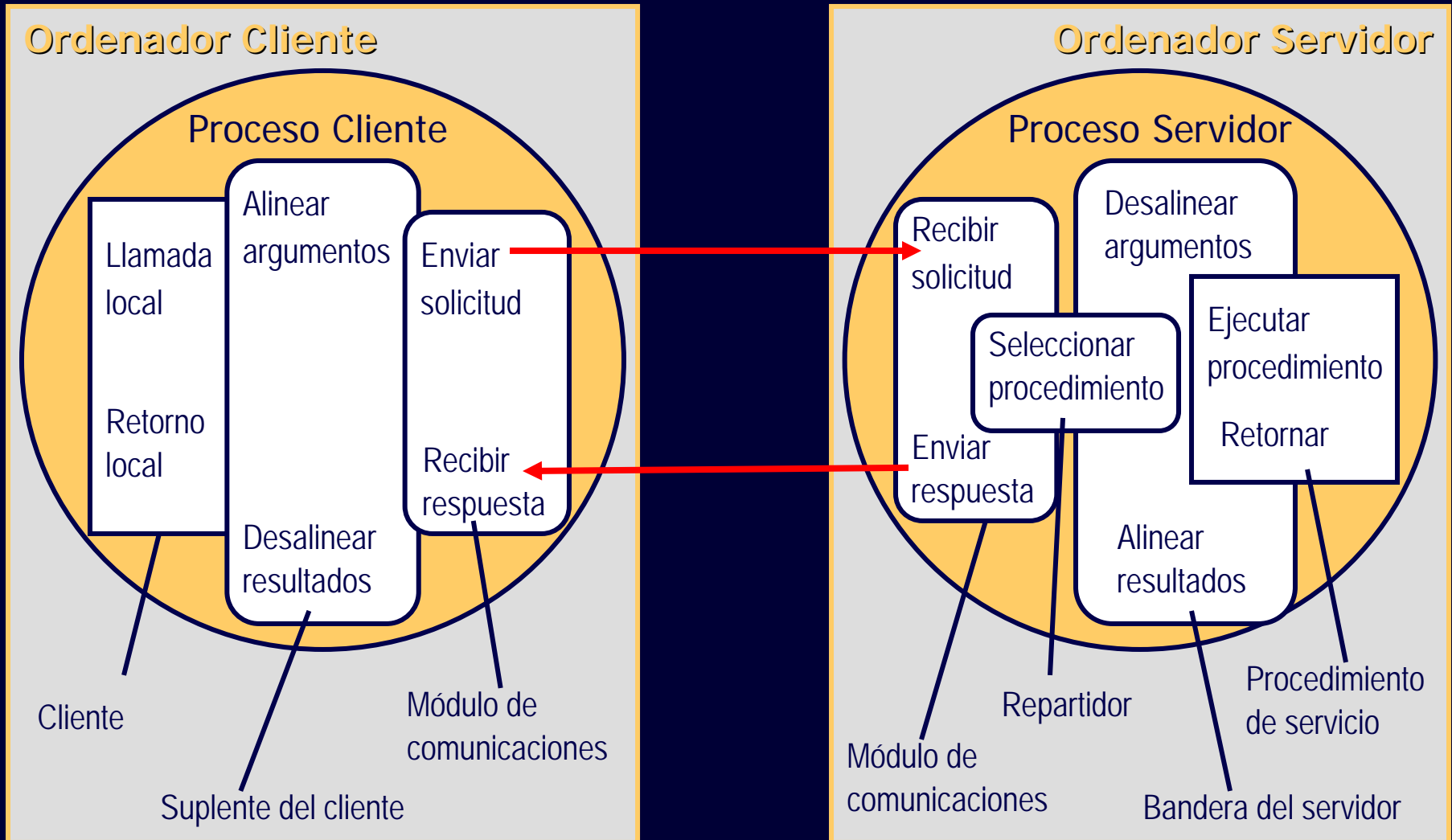
Implementación de la LPR



- La lógica de LPR tiene 3 partes:
 - **procesado de la interfaz**
 - a partir de la definición de interfaz se generan componentes lógicos adicionales
 - que permiten integrar la LPR con los programas escritos en lenguajes convencionales
 - incluyen la alineación y el reparto (*dispatch*)
 - módulo de **tratamiento de la comunicación**
 - usa un protocolo solicitud-respuesta
 - preparado para *linkarlo* al cliente y al servidor
 - **enlace dinámico** de clientes y servidores



La película completa





Procesado de la interfaz: Componentes lógicos generados

- Procedimientos de **resguardo** (*stubs*)
 - procedimiento **suplente** (*resguardo* del cliente):
 - alinea los argumentos de entrada
 - utiliza el módulo de comunicaciones para:
 - enviar la solicitud,
 - esperar una respuesta y, al llegar ésta
 - desalinea los argumentos de salida
 - procedimiento **bandera** (*resguardo* del servidor)
 - desalinea los argumentos de entrada
 - llama al procedimiento correspondiente
 - alinea los argumentos de salida, y
 - retorna al repartidor
- Procedimiento **repartidor** (*dispatcher*)
 - utiliza el módulo de comunicaciones para esperar una solicitud
 - averigua la identidad del procedimiento remoto,
 - llama al *bandera* correspondiente, y, al retornar éste,
 - utiliza el módulo de comunicaciones para enviar la respuesta



- Procesa una definición de interfaz
 - está escrita en un LDI
 - cada procedimiento tiene asignado un identificador
- Genera los componentes lógicos necesarios
- Asegura la conformidad:
 - llamadas ↔ procedimientos
- Resultado de compilar una definición de interfaz:
 - Un *suplente* para cada signatura de la interfaz
 - se compilarán y *linkarán* con el programa del cliente
 - Un *bandera* para cada signatura de la interfaz
 - se compilarán y *linkarán* con el programa del servidor
 - Un *repartidor*
 - se compilará y *linkará* con el programa del servidor
 - Las operaciones de *alineación* en los *resguardos*
 - Las *cabeceras* de los procedimientos de servicio
 - el programador del servicio provee los cuerpos



- **Binding** = establecer una correspondencia:
 - nombre textual servicio ↔ identificador de comunicación
- ¿Por qué necesitamos un **binder** de LPR?
 - el cliente debe **encontrar un servidor** que ejecute el procedimiento remoto
 - transparencia de ubicación, movilidad, replicación, ...
 - debemos asegurar la **consistencia** de los argumentos
 - nos lo garantiza la interfaz de LPR
 - pero el cliente y el servidor se compilan por **separado**:
 - ¿cómo saber que han usado **la misma** definición de interfaz?
 - necesitamos un servicio separado que:
 - mantenga las correspondencias
 - permita que el servidor las **exporte** y el cliente las **importe**



- **Asocia**, a solicitud de un servidor:
 - el nombre de su interfaz de servicio
 - el identificador de su puerto de servicio
 - su nº de versión
- **Busca**, a solicitud de un cliente:
 - el puerto de un servidor cuya interfaz sea de la misma versión que la usada para compilar el cliente
- Si un servidor **migra** y los identificadores de puerto no son independientes de la ubicación:
 - el servidor debe volver a registrarse
 - el cliente debe volver a buscarlo



Interfaz de servicio de un *binder*

void registrar (String nombreServicio, IdPuerto puertoServidor, int version);

hace que el *binder* anote en su tabla a un servidor: el nombre del servicio, junto con el puerto del servidor y su número de versión

void retirar (String nombreServicio, IdPuerto puertoServidor, int version);

hace que el *binder* elimine a un servidor de su tabla

IdPuerto buscar (String nombreServicio, int version);

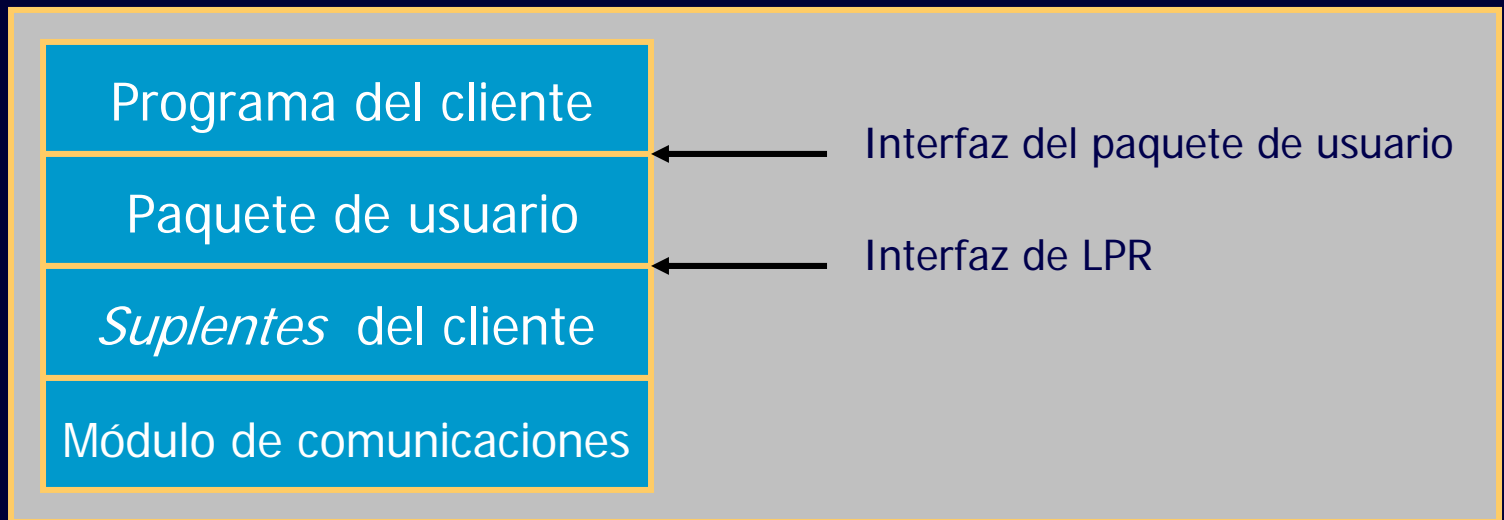
el *binder* busca en su tabla a los servidores cuyo nombre coincida con el dado y retorna la dirección de aquél o aquéllos cuyo número de versión coincida con el dado.



- ¿Cómo encontramos el identificador del puerto del *binder*?
- Alternativas (comunes en UNIX):
 - corre en un **puerto fijo** y conocido
 - migración \Rightarrow compilación del cliente
 - el **S.O. informa** de dónde está
 - p.ej. mediante una variable de entorno
 - migración \Rightarrow señalización al cliente
 - el cliente lo busca enviando una solicitud por **difusión**
 - migración \Rightarrow nueva difusión



- Procedimientos remotos:
 - **diferentes** de los procedimientos locales,
 - escritos en una **notación** especial y
 - definidos para servir al más amplio **rango** de posibles clientes.
 - Luego: interfaz de LPR **incómoda** para clientes
- Paquete de usuario:
 - **librería** de procedimientos convencionales
 - hacen las llamadas a los PRs
 - presentan una interfaz más amistosa al cliente



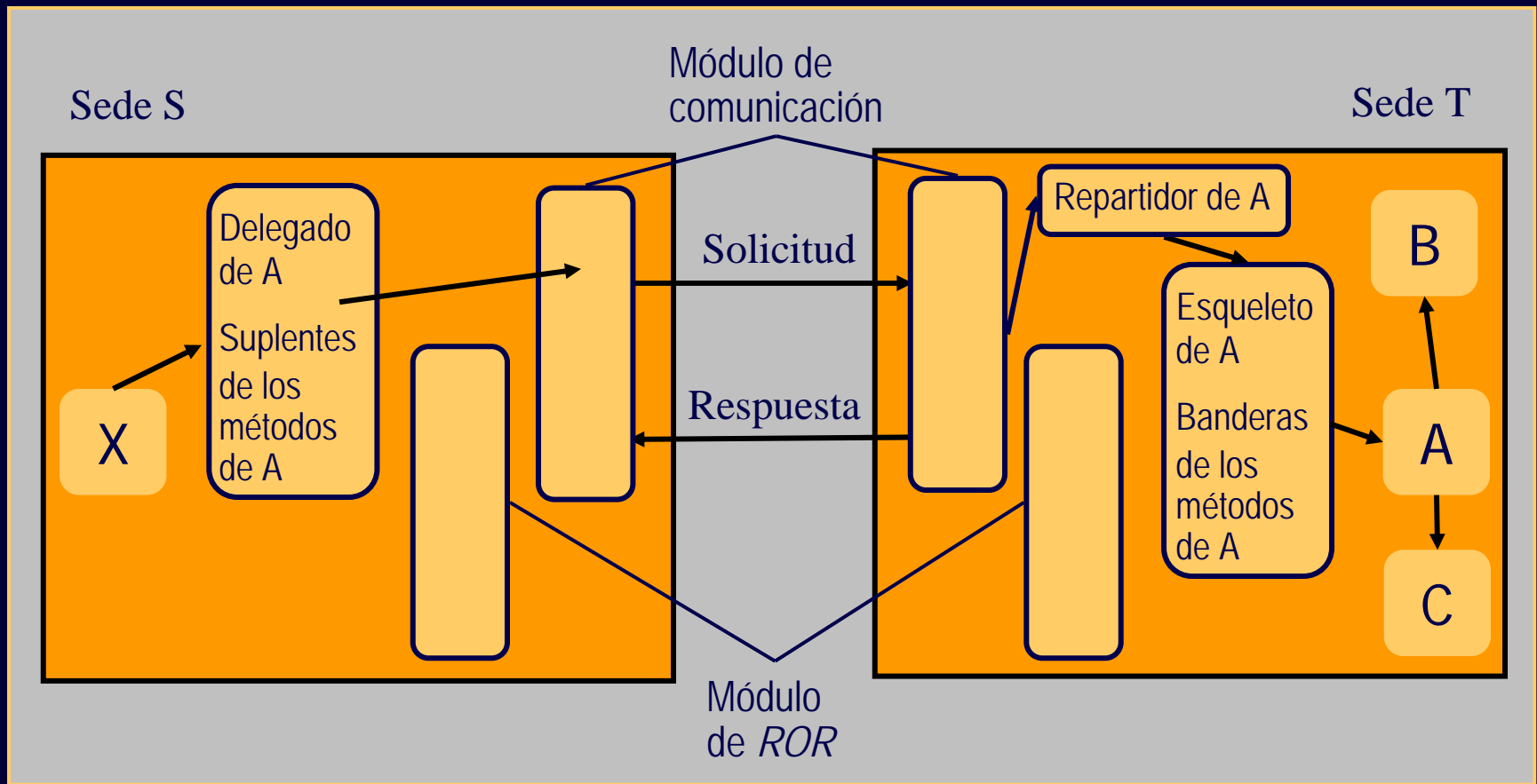
- Es una parte del servicio que reside en el cliente
 - pero es una parte **no fiable**
- Análogos a la librería de entrada/salida estándar de UNIX



La implementación de la IMR




La implementación de la IMR. 1





La implementación de la IMR. 2

- Módulos de comunicación:
 - implementan un protocolo de comunicación y proporcionan una semántica de invocación
 - usan *tipoMensaje* e *idInvocacion* de los mensajes
 - a partir de *refObjeto* selecciona al repartidor adecuado 
- Para cada objeto remoto que se pueda invocar desde un objeto local se crea un **delegado** (*proxy*):
 - para el emisor se comporta como un objeto local, pero
 - en vez de ejecutar el método, se lo pasa al objeto remoto
 - el objeto remoto lo ejecuta y responde
 - sin percibir que la respuesta se envía a un ordenador remoto
 - La clase del delegado asegura que las IMRs concuerdan con los tipos de los objetos remotos:
 - implementa los métodos de la interfaz del objeto remoto
 - los implementa como métodos **suplentes**



La implementación de la IMR. 3

- Toda clase que representa a objetos a los que se pueda acceder remotamente tiene:
 - un **esqueleto** (*skeleton*), que:
 - implementa los métodos del objeto como métodos **bandera**
 - un **repartidor** (*dispatcher*), que:
 - invoca el método **bandera** adecuado basándose en *idMetodo*
- Compilador de interfaz:
 - genera la clase del delegado, el repartidor y el esqueleto
 - En *Orbix*: las interfaces se definen en *CORBA IDL*
 - el CI genera dichas clases en *C++*, *Java*, etc
 - En *Java RMI*: los métodos *ofrecidos* por el objeto remoto se definen como una interfaz *Java*
 - la clase del objeto remoto implementa dicha interfaz,
 - y a partir de esa clase, el compilador genera las del delegado, el repartidor y el esqueleto



Una acción remota completa

- El objeto emisor invoca un método de un OR
 - se invoca un método suplente en el delegado del OR
- el método suplente invocado en el delegado
 - tiene la misma signatura que el método del OR
 - alinea la *refObjeto* del OR, la *idMetodo* y sus argumentos y envía todo en un mensaje de Solicitud
- el módulo de comunicaciones del receptor
 - basándose en *refObjeto*, le pasa la solicitud al repartidor del OR
 - que, a su vez, invoca el método bandera en el esqueleto del OR
- el método bandera
 - desalinea los argumentos e invoca el método del OR
 - luego, alinea el resultado de dicha invocación y envía un mensaje de Respuesta al delegado
- el método suplente invocado en el delegado
 - desalinea la respuesta, y
 - le pasa el resultado al objeto emisor







El programa del cliente y del servidor

- Programa del servidor:
 - clases de los repartidores y los esqueletos
 - clases **servientes** (*servant*). Implementan:
 - las clases de todos los objetos remotos que soporta
 - sección de inicialización
 - crea e inicializa al menos uno de los objetos remotos
 - registra a algún objeto en el servidor de nombres
- Programa del cliente:
 - clases de los delegados
 - usa el servidor de nombres para buscar RORs
- Métodos **factoría**:
 - específicos para crear objetos remotos a petición de los clientes
 - los objetos remotos no se pueden crear mediante constructores



- Gestiona los delegados y las *RORs*
- En la sede del OR, cuando un método retorna la *referencia* de otro objeto, su método bandera:
 - le pide al módulo de *ROR* la correspondiente *ROR*
 - si no existe, el módulo de *ROR* la crea
 - incluye la *ROR* en su mensaje de Respuesta 
- Un delegado se crea cuando se necesita:
 - cuando llega una *ROR* en un mensaje de Respuesta,
 - el MROR comprueba si ya existe un delegado para el OR,
 - si no, crea uno de la clase adecuada
 - al método emisor se le da la *referencia* del delegado 



- El módulo de ROR se basa en una **tabla de objetos remotos** que registra:
 - correspondencias entre identificadores locales y remotos
- Si una entrada se refiere a un objeto local que admite invocaciones remotas:
 - registra la correspondencia entre la *RO* con la que se accede localmente y la *ROR* con la que se accede remotamente a dicho objeto
- Si se refiere a un objeto remoto al que pueden acceder los objetos locales:
 - registra la correspondencia entre la *ROR* del objeto remoto y la *RO* de su delegado local



- La mayoría de las RORs se obtienen por IMR
- Pero hace falta un servicio que registre:
 - la correspondencia entre nombres textuales y referencias de los objetos remotos
- El resultado de buscar por su nombre un objeto remoto es:
 - la creación de un delegado que conoce la *refObjeto* pertinente
- *CORBA: Naming Service*
- *Java: RMIregistry*



La recolección de residuos. 1

- Implica la colaboración entre el recolector local y
 - un módulo añadido, basado en una cuenta de referencias
- Objetivo: mientras exista una ROR existirá el objeto
 - en cuanto nadie mantenga la ROR, el objeto se libera
 - siempre que tampoco existan referencias locales
- Cuando llega una ROR a un cliente,
 - se informa de esta ROR al servidor donde reside el OR
 - durante algún tiempo, lo usará algún objeto del cliente
- Cuando la ROR se deja de usar,
 - se informa al servidor





- La recolección distribuida de residuos colabora con el módulo de *ROR*:
 - en un cliente **informa** a los servidores de la incorporación o eliminación de las *RORs* de objetos de dichos servidores
 - en un servidor **registra** cuántos clientes tienen *RORs* de los objetos del servidor
 - Ej.: *Java RMI*. El módulo de *ROR* de la *MVJ* colabora con la recolección local para proporcionar recolección distribuida
 - Préstamo temporal renovable



- Uno que debe sobrevivir entre activaciones de los procesos
- gestionado por almacenes de objetos persistentes:
 - cuando deja de ser necesario, se **desactiva**
 - se guarda en disco de forma **aplanada**
 - la desactivación debe ser transparente
 - a veces, se salva en disco cuando aún es necesario
 - para conseguir tolerancia a fallos
 - se **activa** cuando alguien lo vuelve a invocar
 - la activación de un objeto también debe ser **transparente**
 - el llamante no debe percibir si ya estaba en memoria o se ha activado
 - optimización: salvar sólo los que se han modificado desde que se activaron





Java RMI



- *Java RMI* extiende el modelo de objeto de *Java* para tratar con objetos distribuidos
 - invocar métodos de objetos remotos
 - de forma *casi* transparente:
 - misma sintaxis, chequeo de tipos, ...
 - el que invoca métodos remotos debe manejar *RemoteExceptions*
 - el objeto remoto debe implementar una interfaz que extiende a *Remote*
 - no necesidad de un LDI
- Ejemplo de aplicación:
 - pizarra electrónica compartida



La Alineación de Objetos en *Java RMI*

- La *Object Serialization* permite pasar como argumentos y resultados de las invocaciones de los métodos:
 - objetos y valores de datos primitivos 
- Para que los individuos de una clase sean alineables:
 - debe implementar la interfaz *Serializable*
 - del paquete *java.io*
- Si un objeto contiene referencias a otros objetos, éstos también se alinean
 - de forma recursiva
- Vale tanto para la RMI como para el almacenamiento en disco 



Ejemplo de *Java Object Serialization*

```
public class ObjetoGrafico implements Serializable {
    String tipo;
    Rectangulo region;
    boolean relleno;

    public Persona (String elTipo,
                   Rectangulo elArea,
                   boolean elRelleno) {
        tipo = elTipo;
        region = elArea;
        relleno = elRelleno;
    }

    // seguido de métodos para acceder a las variables del
    // individuo
}
```





Las interfaces remotas en *Java RMI*

- Se definen extendiendo la interfaz *Remote*
 - del paquete *java.rmi*
- Sus métodos deben lanzar *RemoteException*
 - además de excepciones de la aplicación
- En los argumentos y resultados puede haber:
 - valores de tipos primitivos,
 - objetos *normales* y
 - objetos remotos
 - denotados por el nombre de su interfaz remota
- La semántica es *como-mucho-una-vez*



Ejemplos de interfaces remotas en *Java RMI*

```
import java.rmi.*;
import java.util.Vector;


public interface Figura extends Remote {
    int dameVersion() throws RemoteException;
    ObjetoGrafico dameEstado() throws RemoteException;
}

public interface Pizarra extends Remote {
    Figura creaFigura(ObjetoGrafico g) throws RemoteException;
    Vector dameLasFiguras() throws RemoteException;
    int dameVersion() throws RemoteException;
}
```





Paso de argumentos y resultados

- Todos los argumentos son de entrada
- El resultado es el único de salida
- Todos los tipos primitivos y los objetos remotos son alineables
- Los objetos remotos se pasan por referencia:
 - como referencias a objetos remotos (*ROR*)
- Los objetos no remotos se pasan por valor
 - ⇒ copia (👁️) 
- al alinear:
 - un objeto que implementa una interfaz que extiende a *Remote*, se le sustituye por su *ROR*
 - cualquier objeto, se anota la ubicación (*URL*) de su clase, para permitir que el receptor la descargue



- Al pasar un objeto de una *MVJ* a otra:
 - se descarga de forma automática el código pertinente:
 - si se pasa por valor y el receptor no posee la clase,
 - si es remoto (pasado por referencia) y el receptor no posee la clase del delegado
- Ventajas:
 - no se necesita que todos los usuarios tengan todas las clases
 - los clientes y los servidores pueden usar de forma transparente los individuos de clases nuevas





El servidor de nombres: *RMIregistry*

- Es el *binder* de *Java RMI*
- No da servicio a nivel de red
 - un *RMIregistry* en cada ordenador que aloje objetos accesibles remotamente
- mantiene una tabla con correspondencias
 - para cada objeto alojado:
 - referencia textual ↔ ROR
- accedido por métodos de la clase *Naming*
 - argumento: una cadena con forma de *URL*
`rmi://nombreOrdenador:puerto/nombreObjeto`
 - donde *nombreOrdenador* y *puerto* son la ubicación del *RMIregistry*



La clase *Naming* del *RMIregistry* de *Java*

```
void rebind(String name, Remote obj);
```

Método usado por los servidores para registrar por nombre el identificador de un objeto remoto.

```
void bind(String name, Remote obj);
```

Igual que rebind, pero si el nombre ya está enlazado a una referencia a un objeto remoto, se lanza una excepción

```
void unbind(String name, Remote obj);
```

Deshace un enlace.

```
Remote lookup(String name);
```

Método usado por los clientes para buscar por nombre un objeto remoto. Retorna una referencia a un objeto remoto.

```
String [] list();
```

Retorna un array de cadenas que contiene los nombres enlazados en el *RMIregistry*.



La construcción de programas



Ejemplo de programa servidor de pizarra electrónica

- Representa cada figura como un objeto remoto que implementa la interfaz *Figura*
 - mantiene su estado y su número de versión
- representa una colección de figuras como un objeto remoto que implementa la interfaz *Pizarra*
 - mantiene un *vector* de figuras
- Consiste en un método *main* y clases sirvientes que implementan las interfaces remotas
 - *main* sólo crea un individuo de *SirvientePizarra*
 - los individuos de *SirvienteFigura* se crean bajo demanda
- Gestor de seguridad: *RMI SecurityManager*
 - asegura que las clases remotas descargadas son fiables
 - sin él, sólo se pueden usar clases locales



Clase *ServidorPizarra* con método *main*

```
import java.rmi.*;

public class ServidorPizarra {
    public static void main(String args []) {
        System.setSecurityManager(new RMISecurityManager());
        try{
            SirvientePizarra unaPizarra = new SirvientePizarra();
            Naming.rebind("Pizarra", unaPizarra);
            System.out.println("Servidor de Pizarra preparado");
        } catch(Exception e) {
            System.out.println("Main del servidor de Pizarra " +
                e.getMessage());
        }
    }
}
```



Clase *SirvientePizarra* que implementa la interfaz *Pizarra*

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

public class SirvientePizarra extends UnicastRemoteObject
                                implements Pizarra {
    private Vector laLista;    // Contiene la lista de Figuras
    private int version;
    public SirvientePizarra() { ... }
    public Figura creaFigura(ObjetoGrafico g)
                                throws RemoteException {
        version++;
        SirvienteFigura s = new SirvienteFigura(g, version);
        laLista.addElement(s);
        return s;
    }
    public Vector dameLasFiguras() throws RemoteException { ... }
    public int dameVersion() throws RemoteException { ... }
}
```



- También establece un gestor de seguridad
- Usa el *RMIregistry* para conseguir una referencia a un objeto remoto
- A partir de ahí puede invocar métodos de dicho objeto o de otros que vaya descubriendo
 - después de invocar *dameLasFiguras*, puede mostrarlas en pantalla
 - tras crear una nueva figura, puede invocar *creaFigura* para añadirla a la pizarra compartida
 - periódicamente puede invocar *dameVersion* y, si hay figuras nuevas, pedir las y mostrarlas



Cliente *Java* de Pizarra

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

public class ClientePizarra {
    public static void main(String args []) {
        System.setSecurityManager(new RMISecurityManager( ));
        Pizarra unaPizarra = null;
        try{
            unaPizarra = (Pizarra) Naming.lookup("rmi://kraken/Pizarra");
            Vector fLista = unaPizarra.dameLasFiguras();
            ...
        } catch (RemoteException e) {
            System.out.println(e.getMessage());
        } catch (Exception e) {
            System.out.println("Cliente: " + e.getMessage());
        }
    }
}
```



- Permite que los servidores informen a los clientes de la ocurrencia de eventos
 - en vez de obligar a los clientes a sondear
- Método:
 - el cliente crea un objeto remoto que implementa una interfaz con un método al que pueden llamar los servidores: **objeto de retrollamada** (*callback object*)
 - la interfaz remota implementada por el servidor incorpora métodos que permiten a los clientes pasar la *ROR* de su objeto de retrollamada
 - cuando se produce un evento, el servidor avisa a los clientes registrados



- Interfaz del objeto de retrollamada del cliente:

```
public interface RetroPizarra extends Remote {  
    void avisa(int version) throws RemoteException;  
}
```

- Métodos adicionales en la interfaz *Pizarra*

```
public interface Pizarra extends Remote {  
    Figura creaFigura(ObjetoGrafico g) throws RemoteException;  
    Vector dameLasFiguras() throws RemoteException;  
    int dameVersion() throws RemoteException;  
    int registra(RetroPizarra retro) throws RemoteException;  
    void daDeBaja(int idRetro) throws RemoteException;  
}
```