Diseño Y Aplicaciones de Sistemas Distribuidos

Comunicación en sistemas distribuidos (i):
del cliente/servidor al modelo de objetos

Joan Vila

DISCA / UPV

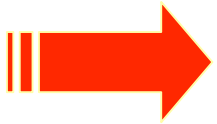
Departament d'Informàtica de Sistemes i Computadors

Universitat Politècnica de València

DISCA
DEPARTAMENT D'INFORMÀTICA DE SISTEMES I COMPUTADORS

Comunicación en sistemas distribuidos

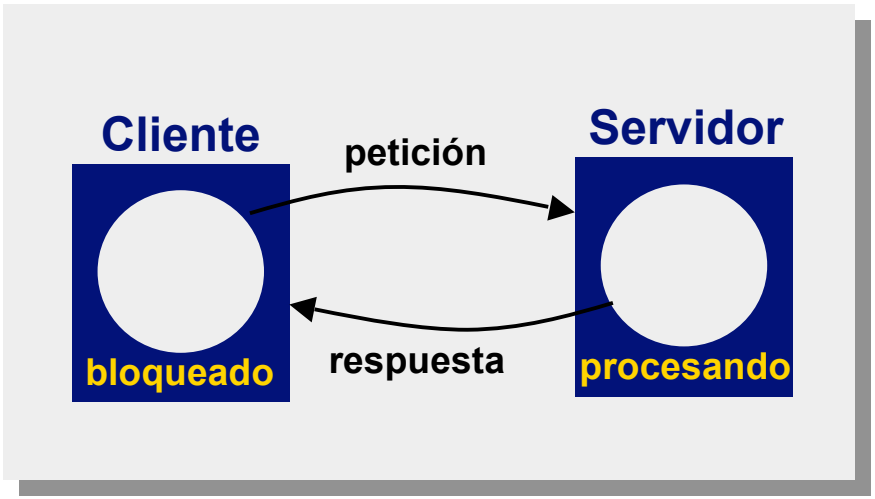
Indice

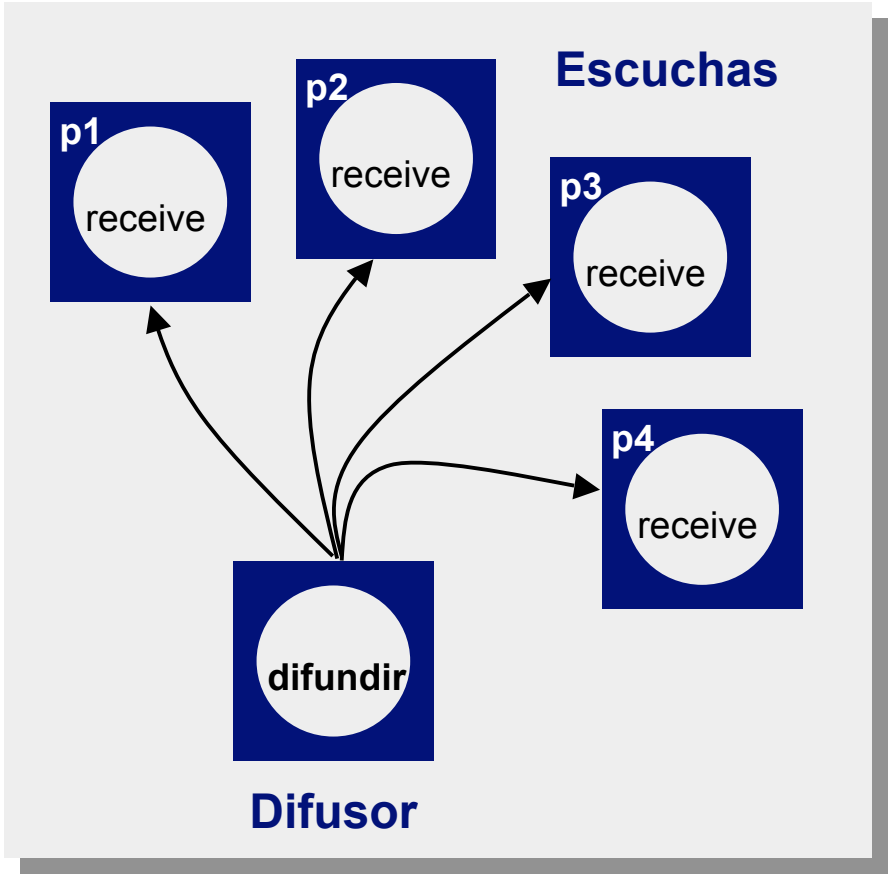


- Introducción
- El modelo cliente-servidor
- Llamadas a procedimientos remotos (RPC's)
- El modelo de objetos distribuido
- La arquitectura CORBA

Introducción

- **Comunicación en sistemas distribuidos**
  - La comunicación en sistemas distribuidos sigue, fundamentalmente, dos patrones:
    - Cliente-Servidor
    - Difusor-Escucha

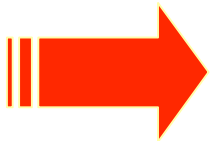




Introducción

- **Comunicación en sistemas distribuidos**
  - **Cliente - servidor**: la comunicación es iniciada por un proceso, *cliente* que realiza una petición a otro proceso *servidor*, el cual responde. Puede seguir diversos modelos:
    - Basada en **mensajes**: *sockets*
    - Basada en **llamadas a procedimientos remotos** o RPC's (*Remote Procedure Calls*): *RPC de Sun*
    - Basada en **invocaciones a objetos remotos**: *RMI en Java, CORBA*
  - **Difusión a grupos**: la comunicación es iniciada por un proceso *difusor* que envia una información a un grupo de procesos *escuchas*.

Indice



- Introducción
- El modelo cliente-servidor
- Llamadas a procedimientos remotos (RPC's)
- El modelo de objetos distribuido
- La arquitectura CORBA

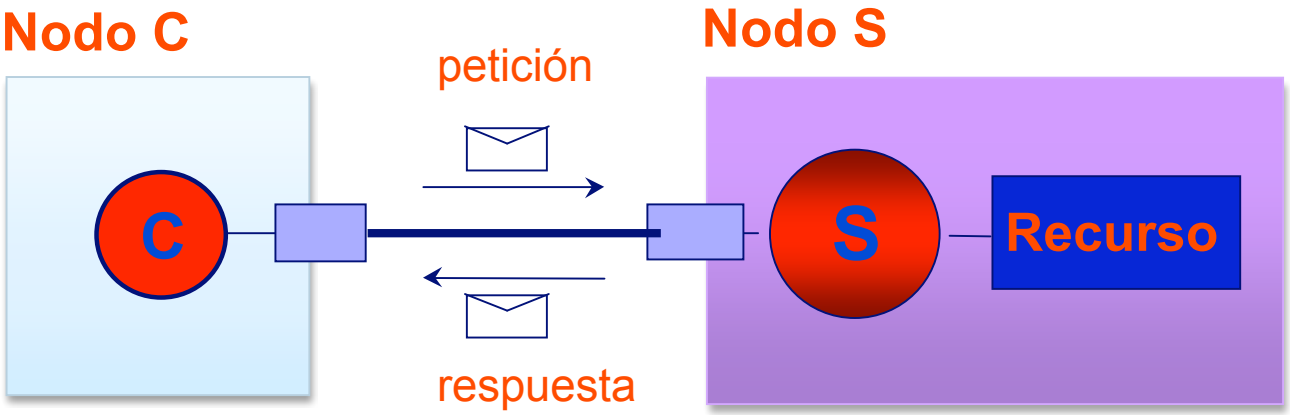
El modelo cliente-servidor

- **El modelo cliente servidor**
  - Basado en asociar un *proceso servidor* a los recursos que quieren compartirse o hacerse visibles a otros nodos.

  Existen procesos con dos patrones de comportamiento diferenciados:
  - **Servidores:** gestionan un recurso y ofrecen servicios relacionados con este recurso a otros procesos.
    - Reciben peticiones de *procesos clientes*, vía mensajes
    - las ejecutan en su nombre y
    - devuelven una respuesta.
  - **Clientes:** formulan peticiones de servicio a los servidores.

El modelo cliente-servidor

- **El modelo cliente servidor**



CLIENTE

```
…
send(serv, peticion, …);
recv(serv, respuesta, …);
...
```

SERVIDOR

```
repeat
        recv(any, peticion, …);
        respuesta= trabajar(peticion);
        ret=remite(peticion);
        send(ret, respuesta, …);
forever;
```

El modelo cliente-servidor

- **Tipos de servidores**

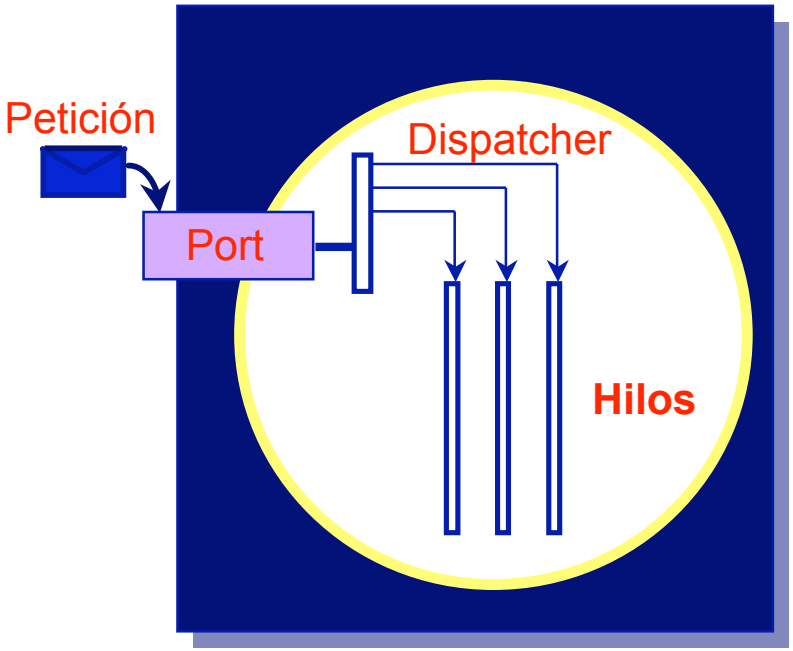
  Un proceso servidor puede adoptar dos estrategias básicas para la gestión del recurso que custodia:

  – **Servidores secuenciales:** Atienden las peticiones secuencialmente.
    - Adecuados para recursos reutilizables en serie
    - Pueden introducir cuellos de botella

  – **Servidores multi-hilo:** pueden servir varias peticiones simultáneamente creando un nuevo hilo (*thread*) para atender cada nueva petición que reciben.
    - Precisan facilidades de procesos concurrentes: “threads”
      – Ver transparencias de Java
    - Adecuado para recursos instanciables

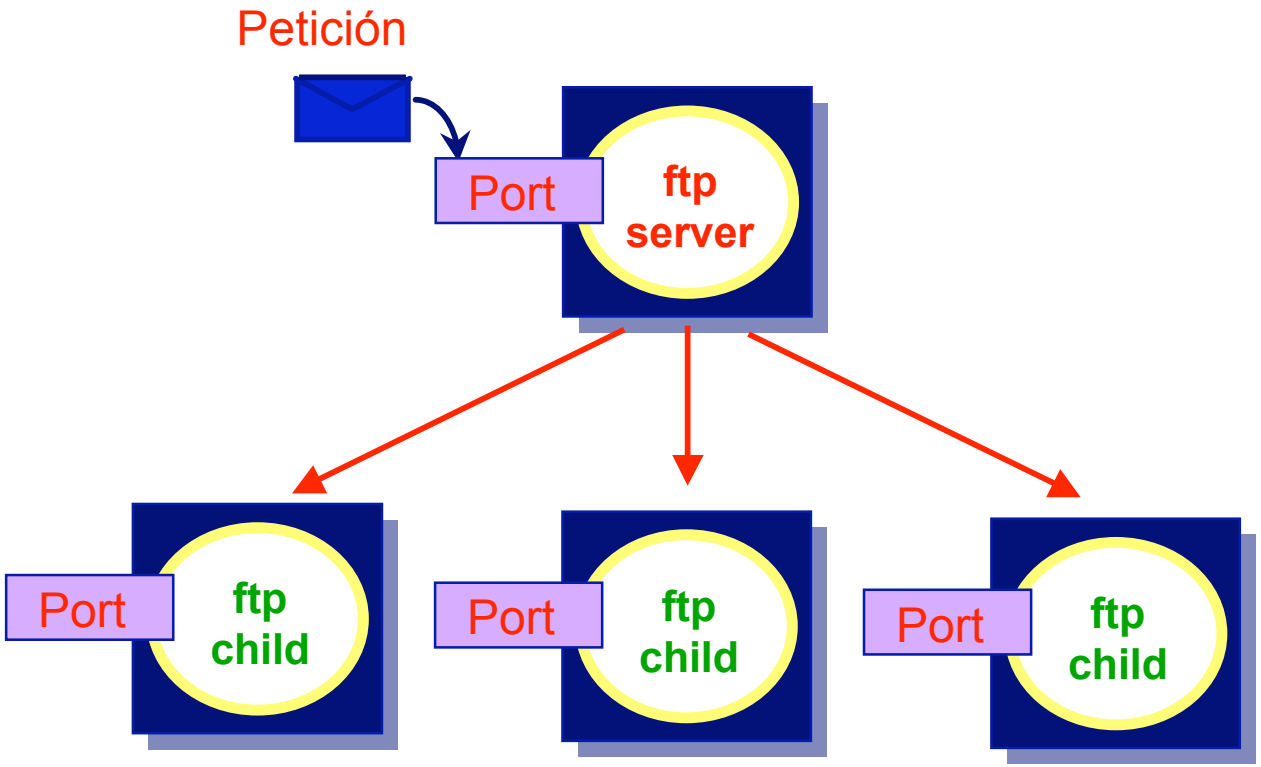
El modelo cliente-servidor

- **Servidores multi-hilo**
  - Estructura:
    - Un distribuidor (*dispatcher*) de peticiones.
    - Una hilo dedicado (*worker*) por cada petición.



El modelo cliente-servidor

- **Servidores concurrentes**
  - **Modelo de Unix**: no se crean hilos para atender las peticiones, sino nuevos procesos con fork().

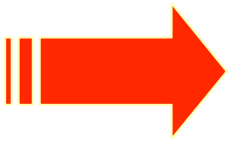


El modelo cliente-servidor

- **Sistemas de mensajes**
  - Para realizar una aplicación cliente-servidor es necesario disponer de un sistema de mensajería entre procesos, o en otras palabras, un *nivel de transporte*.
  - Existen dos modelos básicos de servicios de mensajería:
    - Comunicación orientada a **conexión**.
    - Comunicación sin conexión o **datagrama**.
  - **Ejemplo:** sockets TCP/IP, UDP/IP
- **La interfaz socket**
  - La interfaz socket en Java
    - Ver transparencias de Java

Indice

- Introducción
- El modelo cliente-servidor
- Llamadas a procedimientos remotos (RPC's)
- El modelo de objetos distribuido
- La arquitectura CORBA

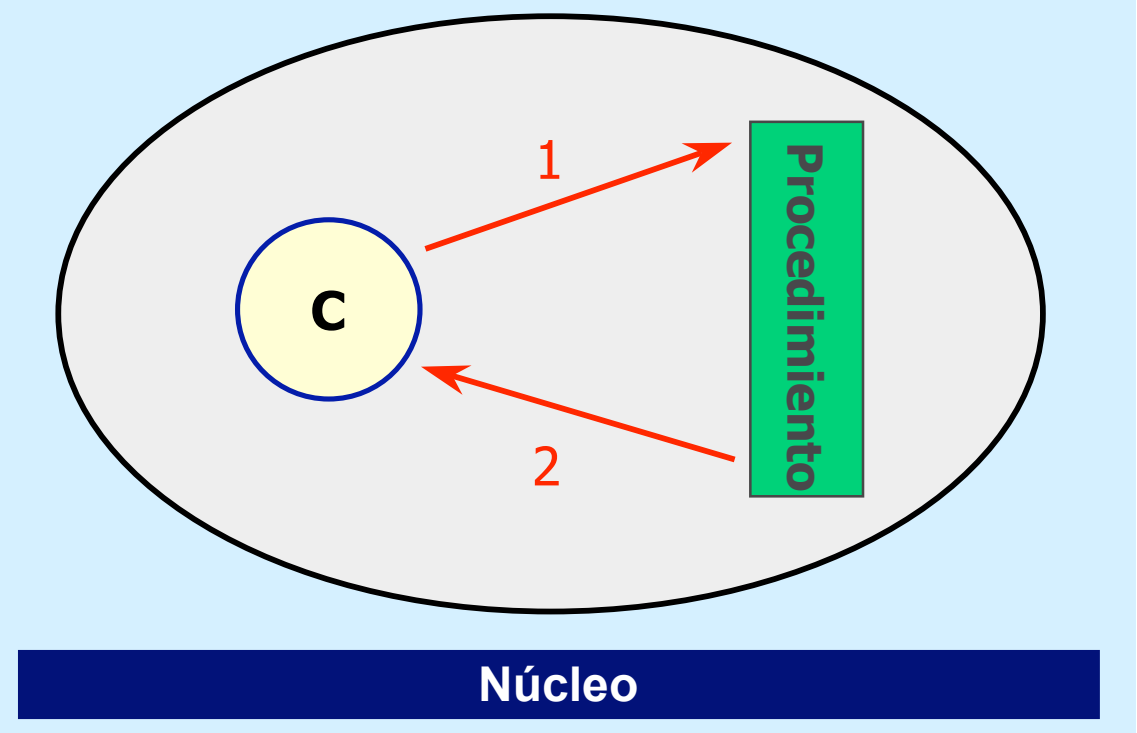


RPC

- **Concepto de RPC**
  - **RPC (*Remote Procedure Call*):** llamada a procedimiento remoto.
    - Es una forma de comunicación en sistemas distribuidos basada en extender el uso de *llamadas a procedimiento* para invocar servicios sobre objetos o recursos remotos en lugar de mensajes.
  - **Características básicas:**
    - Oculta el sistema de mensajes: hace transparente su uso
    - Hace transparente la invocación de servicios locales y remotos

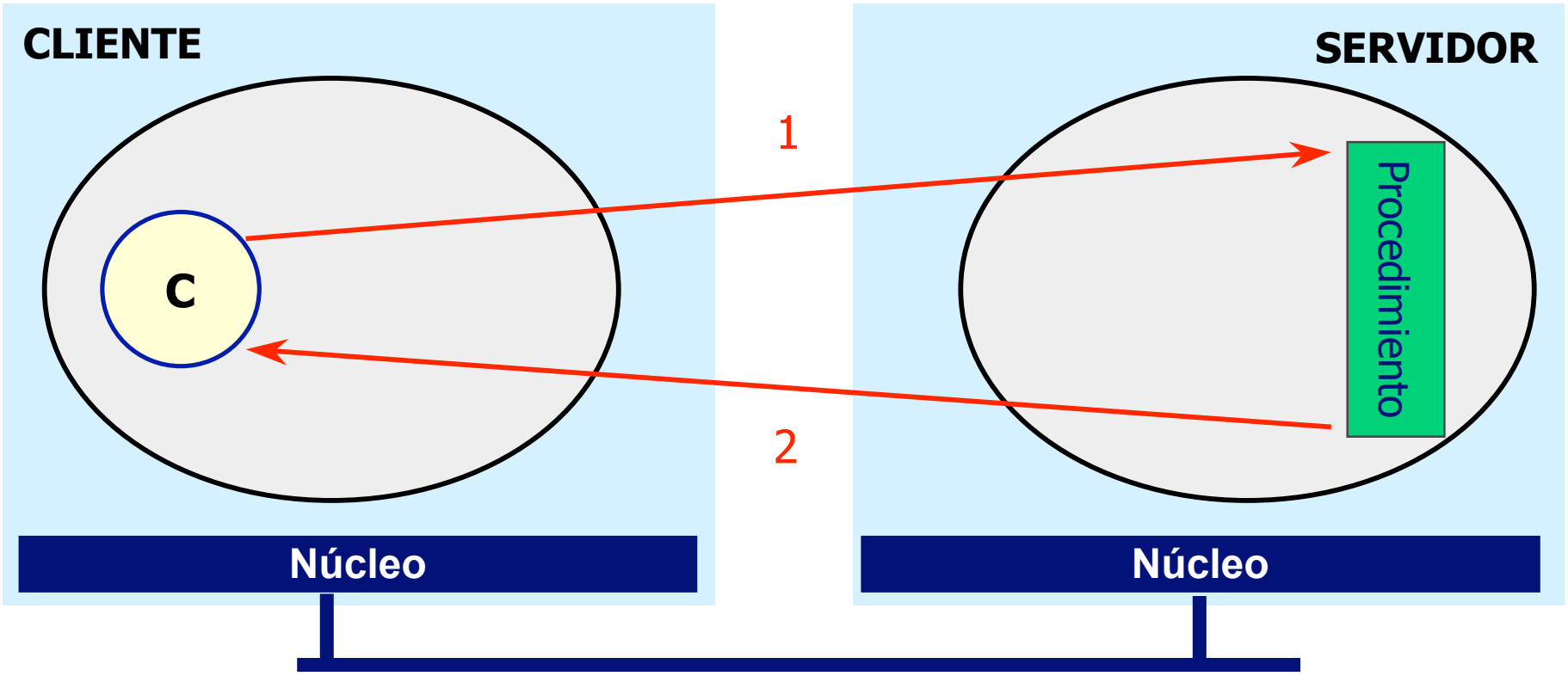
RPC

- **Concepto de RPC**
  - Esquema de una llamada a procedimiento local



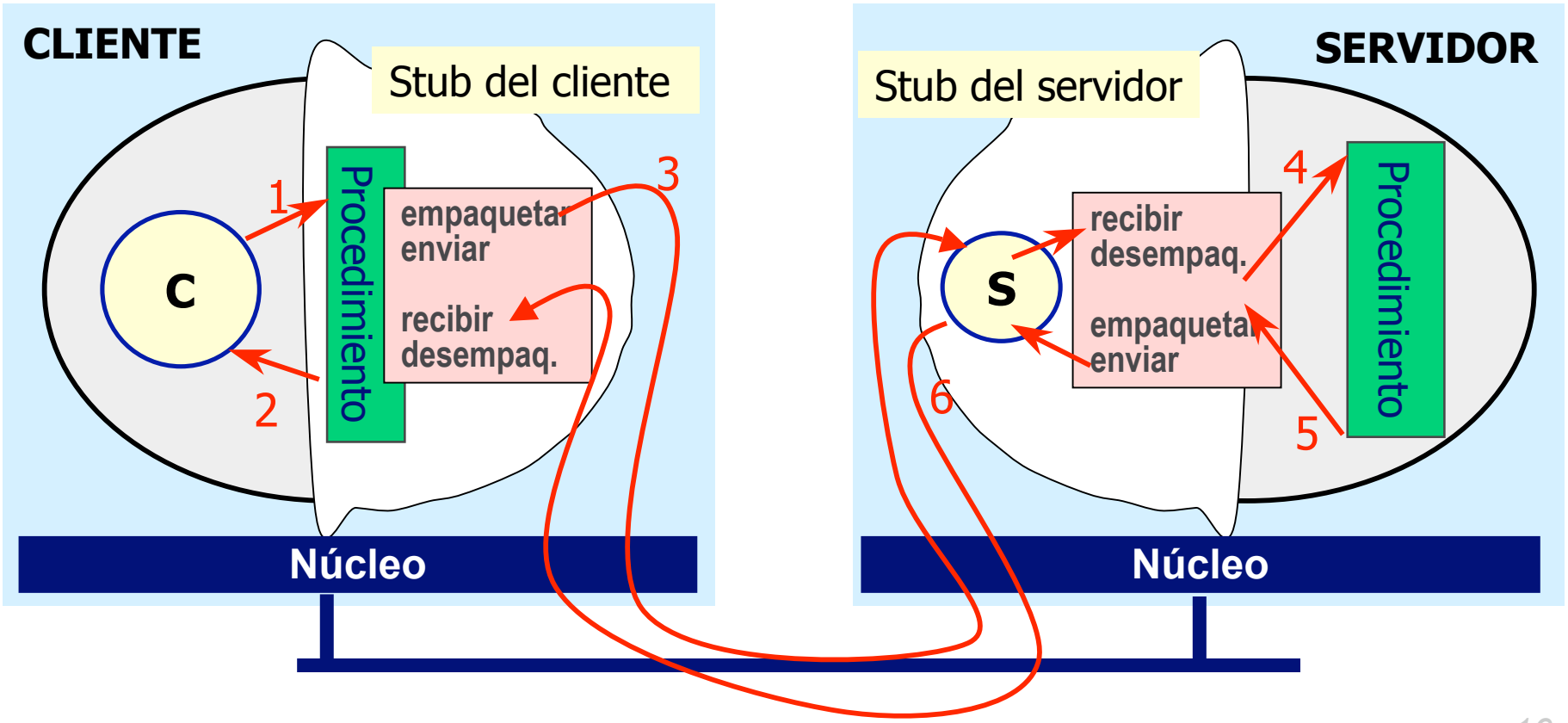
RPC

- **Concepto de RPC**
  - Esquema de una llamada a procedimiento remota



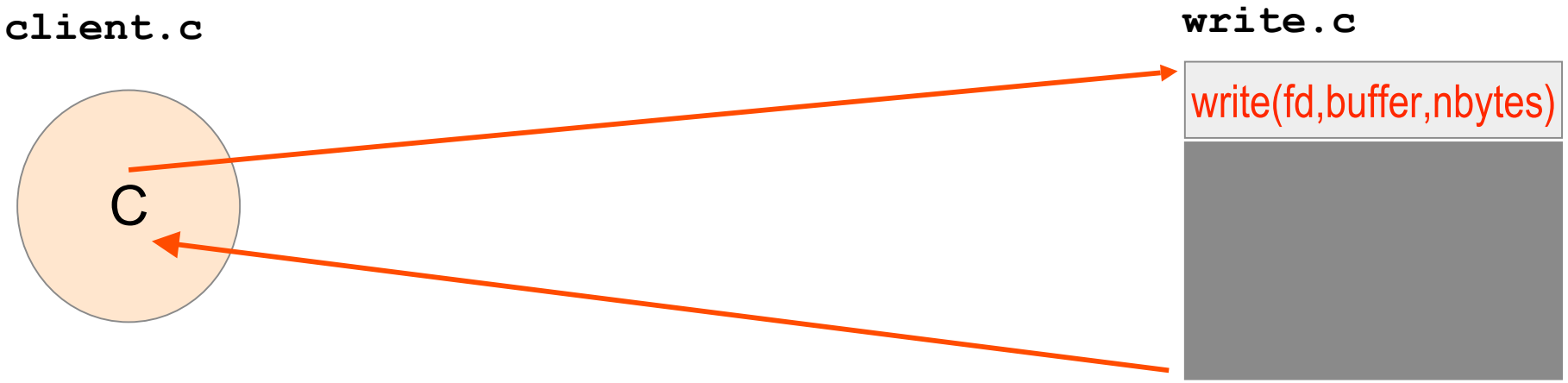
RPC

- **Implementación de la RPC**
  - El método de los *stubs*



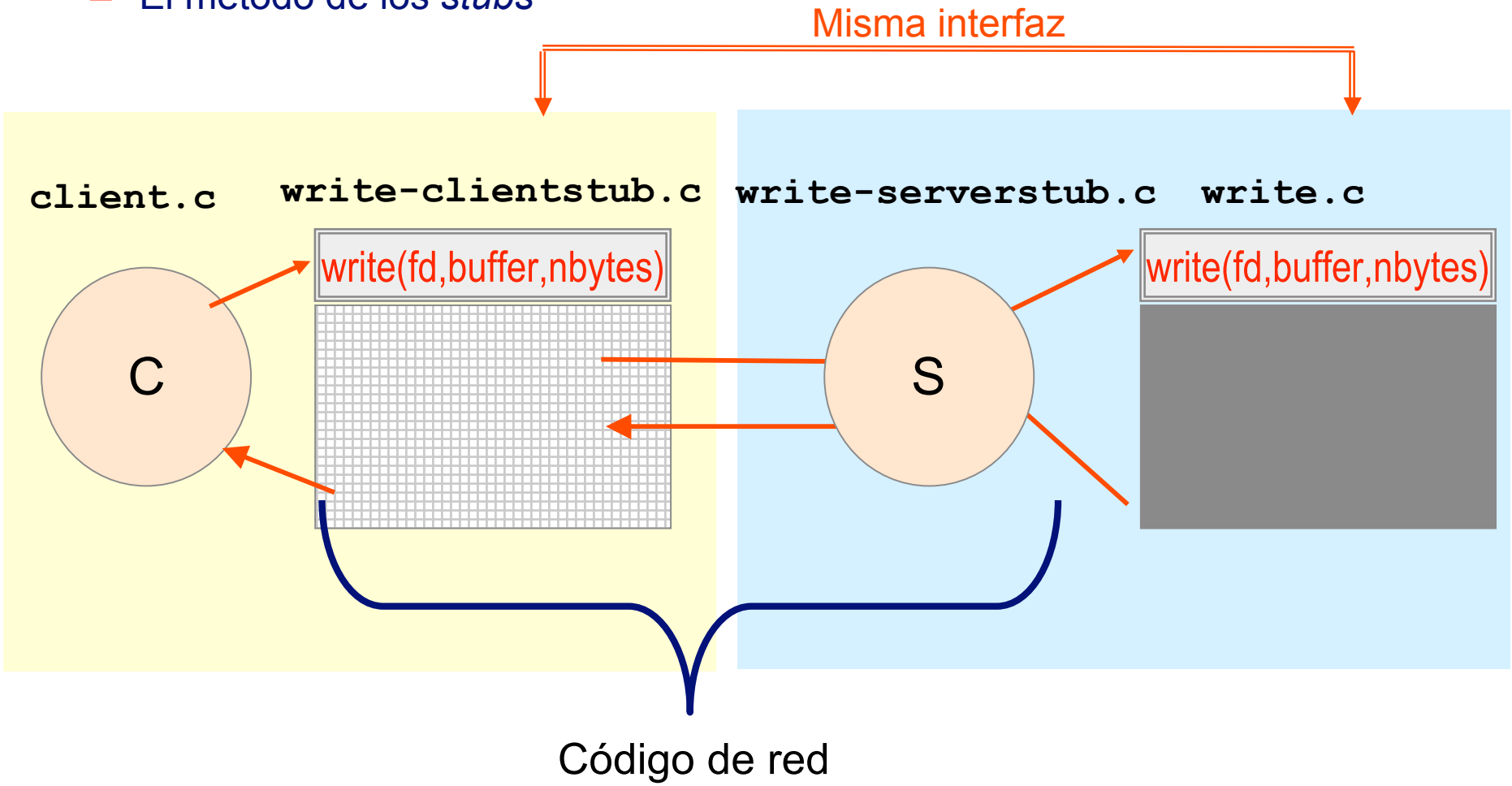
RPC

- **Implementación de la RPC**
  - El método de los *stubs*



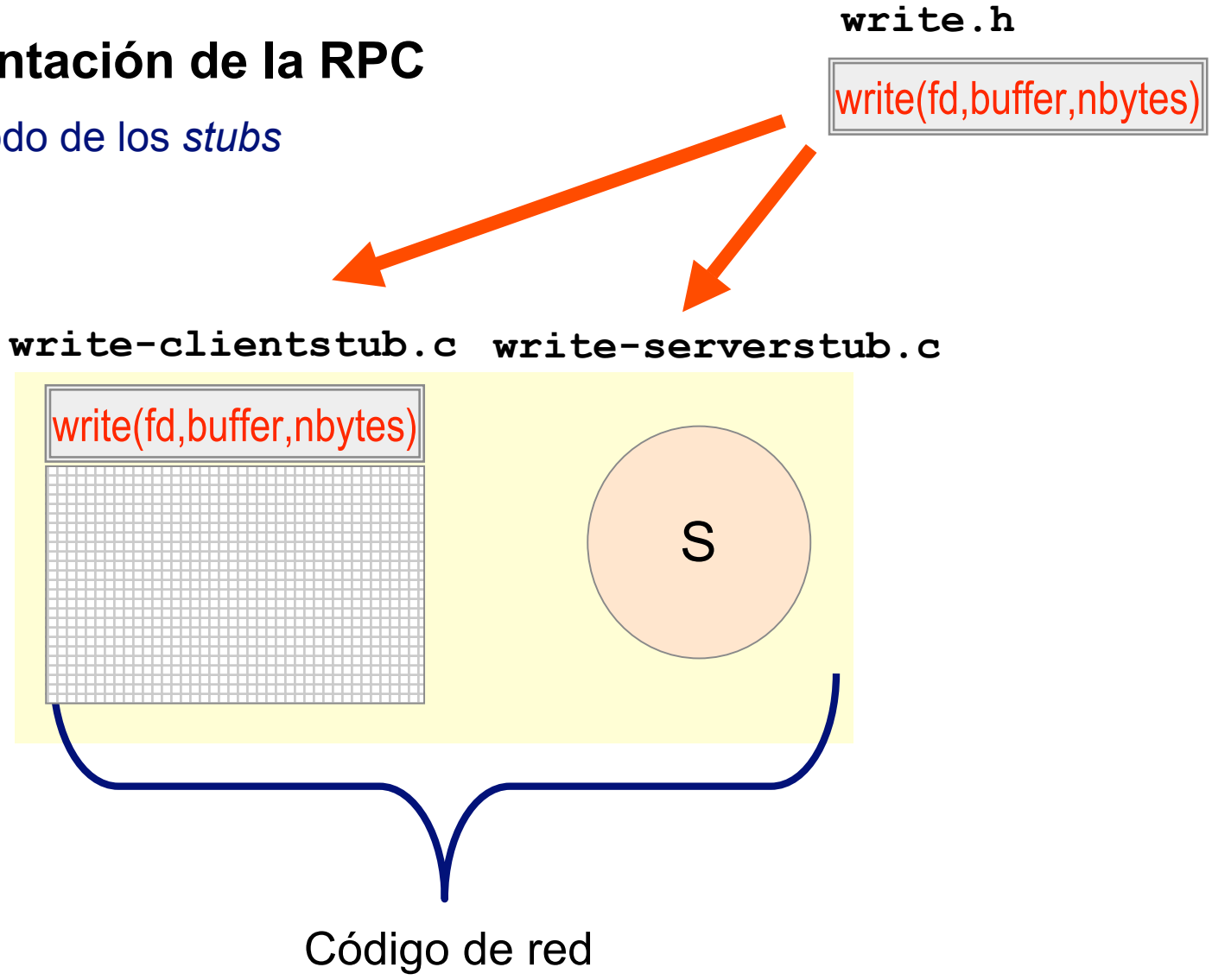
RPC

- **Implementación de la RPC**
  - El método de los *stubs*



RPC

- **Implementación de la RPC**
  - El método de los *stubs*



RPC

- **Implementación de la RPC: *stubs***
  - **Concepto de stub:** función de biblioteca que oculta el sistema de mensajes y proporciona a la comunicación remota una semántica de llamada a procedimiento.
  - **El stub del cliente:** es un representante del procedimiento remoto en el nodo del cliente que tiene su misma interfaz (nombre, argumentos, semántica) pero que no ejecuta el procedimiento (no realiza la funcionalidad del procedimiento).
    - La funcionalidad del stub del cliente es la siguiente:
      - Empaquetar los parámetros en un mensaje (paramater marshalling)
      - Envíar el mensaje al servidor
      - Esperar la respuesta
      - Desempaquetar la respuesta

RPC

- **Implementación de la RPC: *stubs***
  - **El stub del servidor:** es el soporte de ejecución de un procedimiento o conjunto de procedimientos remotos. Normalmente es proceso servidor en vez de una biblioteca (como en el caso del cliente).
  - La funcionalidad del stub del servidor es similar a la del cliente pero, adicionalmente, se encarga de:
    - **Creación de hilos**: en caso de servidor con múltiples hilos, crear un hilo para gestionar cada invocación.
    - ***Dispatching***: seleccionar el procedimiento remoto solicitado (si hay varios) e invocarlo, actuando como hilo representante del cliente.

RPC

- **Generación de *stubs***

  Existen dos enfoques básicos para la generación de *stubs*:

  – Generación automática por un preprocesador de RPC
    - Ejemplo: rpcgen de Sun

  – Generación “a mano” por el propio programador
    - Constituye un “buen estilo” de programación separar el código independiente de red y el código dependiente de la red en procedimientos separados; éste último código lo constituyen los *stubs* y *skeletons*.

RPC

- **Empaquetado de parámetros en mensajes**
  - Supone convertir los datos a una representación serie, adecuada para ser encapsulados en un mensaje (*parameter marshalling*):
  - Los datos que se pasan por valor son fáciles de empaquetar.
  - En los datos que se pasan por referencia se suele utilizar la técnica de *copy-in copy-out*: hacer una copia del parámetro que se referencia sobre el mensaje y referenciar la copia.
    - Esta técnica puede fallar en procedimientos como “intercambiar x e y”.
    - swap(&x, &y)
  - Pasar listas enlazadas u otro tipo de estructuras recursivas puede resultar prácticamente imposible.

RPC

- **Representación estándar de datos**

  En sistemas heterogéneos, los componentes de un sistema distribuido pueden tener diferentes representaciones de los datos.

  Existen dos formas de abordar la heterogeneidad en la representación de datos:

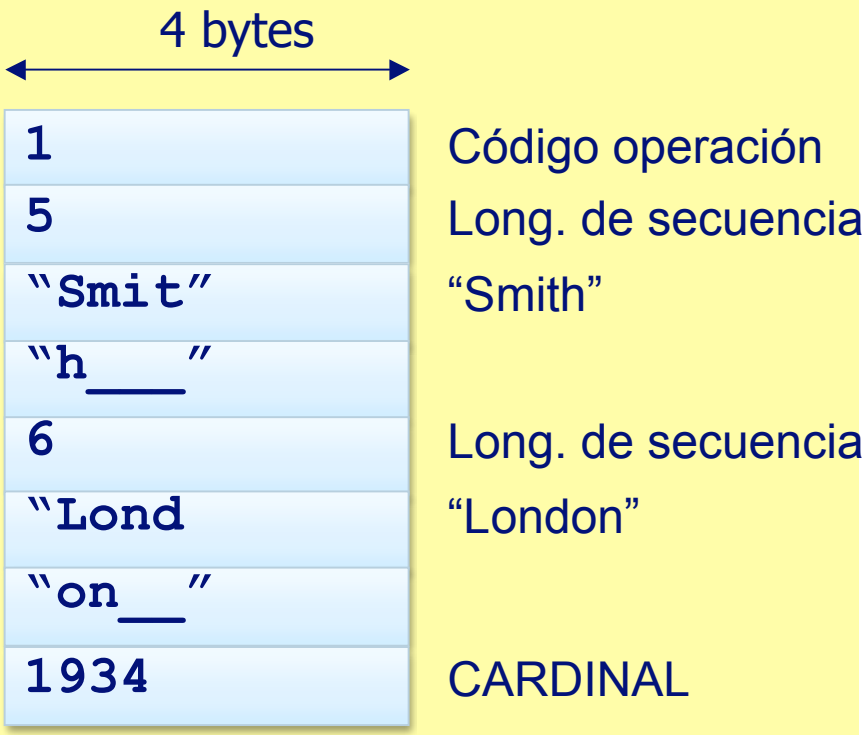
  – Los valores son convertidos a una *representación externa* de red estándar para transmitirlos y vueltos a convertir a *representación interna* al recibirlos. Ej: XDR (eXternal Data Representation de SUN) Courier (Xerox), ASN.1 (CCITT)

  – Los valores son transmitidos en su representación original, junto con una etiqueta de la arquitectura y, en caso necesario, son convertidos por el receptor (Ej: XML, s.o. Mach)

RPC

Ejemplo de message en XDR

El mensaje es:
"Smith", "London", 1934



- **Empaquetado de parámetros**

Ejemplo de empaquetado "a mano"

```
int proc1(char *name; char *place; int year) {
        char * message[1024];
        sprintf(message, "%d %d %s %d %s %d",
                PROC1, strlen(name), name, "   ",
                strlen(place), place, "  ",year);
        msglen = strlen(name) + strlen(place) + ...
        send(socketcli, message,msglen);
        recv(socketcli, message,msglen);
        return (int) message;
}
```

Indice

- Introducción
- El modelo cliente-servidor
- Llamadas a procedimientos remotos (RPC's)
- El modelo de objetos distribuido
- La arquitectura CORBA

El modelo de objetos distribuido

- **Fundamentos**

  Se basa en extender el paradigma de objetos a sistemas distribuidos:

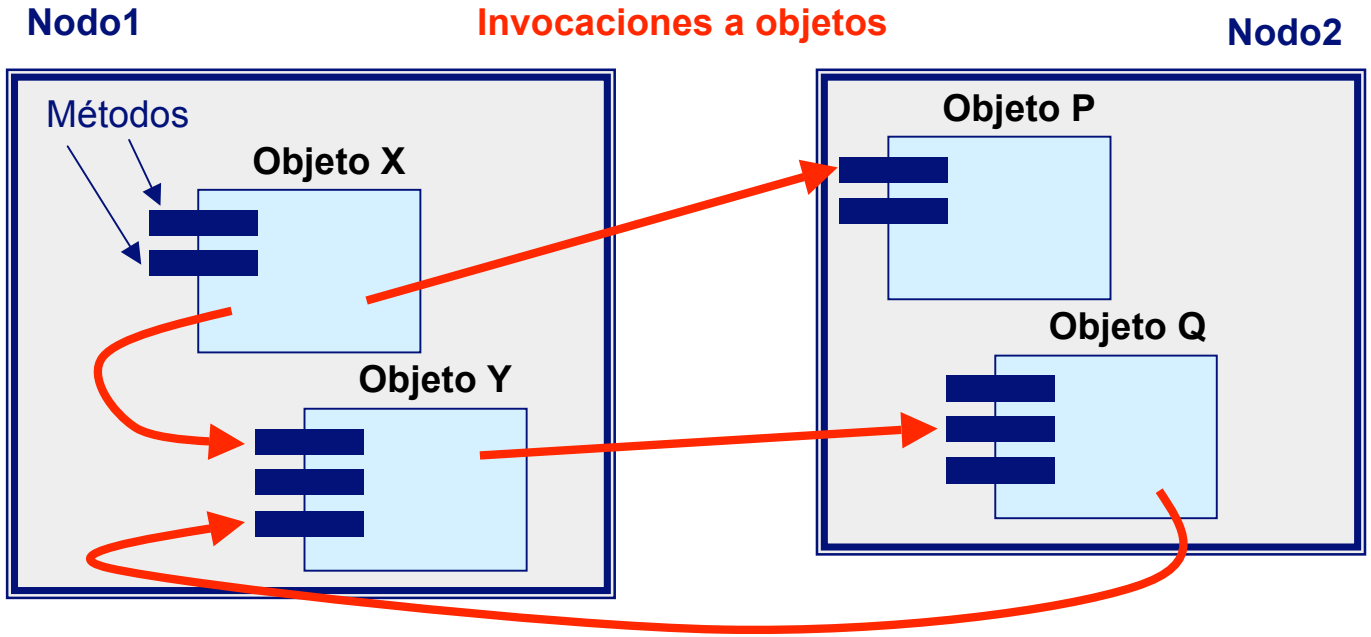
  – Objetos como forma de **encapsular los recursos**

  – Objetos como **unidad de distribución**

  - La particion de una aplicación y su distribución se hace en base a objetos

  – Comunicación remota basada en **invocaciones a métodos de objetos**.

  - Transparencia de invocación.



El modelo de objetos distribuido

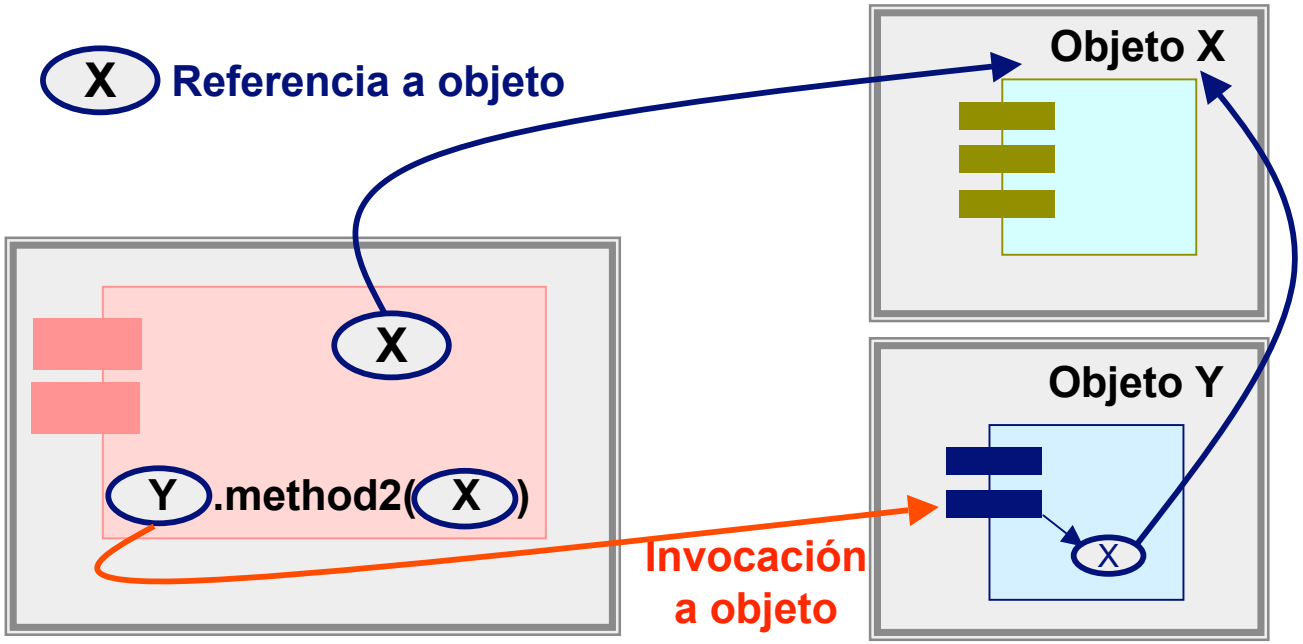
- **Aspectos clave del modelo de objetos distribuido**
  - **Referencias** a objetos remotos
    - Extender el concepto de **referencia a objeto** a objetos remotos
    - **Servicio de nombres**
  - **Invocación de métodos remotos**
    - **Extender el concepto de RPC** a objetos
    - Implementar **paso por valor de objetos** (**"migrar objetos"**)
    - Invocaciones estáticas e **invocaciones dinámicas**
  - Definición y procesamiento de **interfaces**
    - Definición de interfaces en **sistemas heterogéneos**
    - **Generación de *stubs***

Referencias a objetos

- **Referencias a objetos**
  - **Concepto de referencia a objeto**:
    - Determina de forma unívoca la ubicación de un objeto en un sistema distribuido.
    - Se usa para referenciar un “objeto distribuido”.
    - Extiende el concepto de puntero para que también se puedan referenciar objetos remotos.
  - **Implementación de referencias a objetos**.
    - Las referencias se manejan de forma abstracta, siendo oculta su implementación.
    - La implementación es dependiente de la plataforma, pero algunas plataformas como CORBA la estandarizan en aras de la interoperabilidad.
    - La implementación de una referencia normalmente contiene la siguiente información:
      - Dirección IP de máquina
      - Número de port dedicado al sistema de objetos
      - Número de objeto dentro de los objetos de un nodo

Referencias a objetos

- **Referencias a objetos**
  - **Utilización de las referencias a objetos**: si "X" es una referencia a un objeto remoto, entonces se pueden utilizar de los siguientes modos:
    - **Invocación estática de métodos remotos**: X.method1(arg1,arg2,...)
    - **Pasar la referencia a otro objeto**: Y.method3(X);



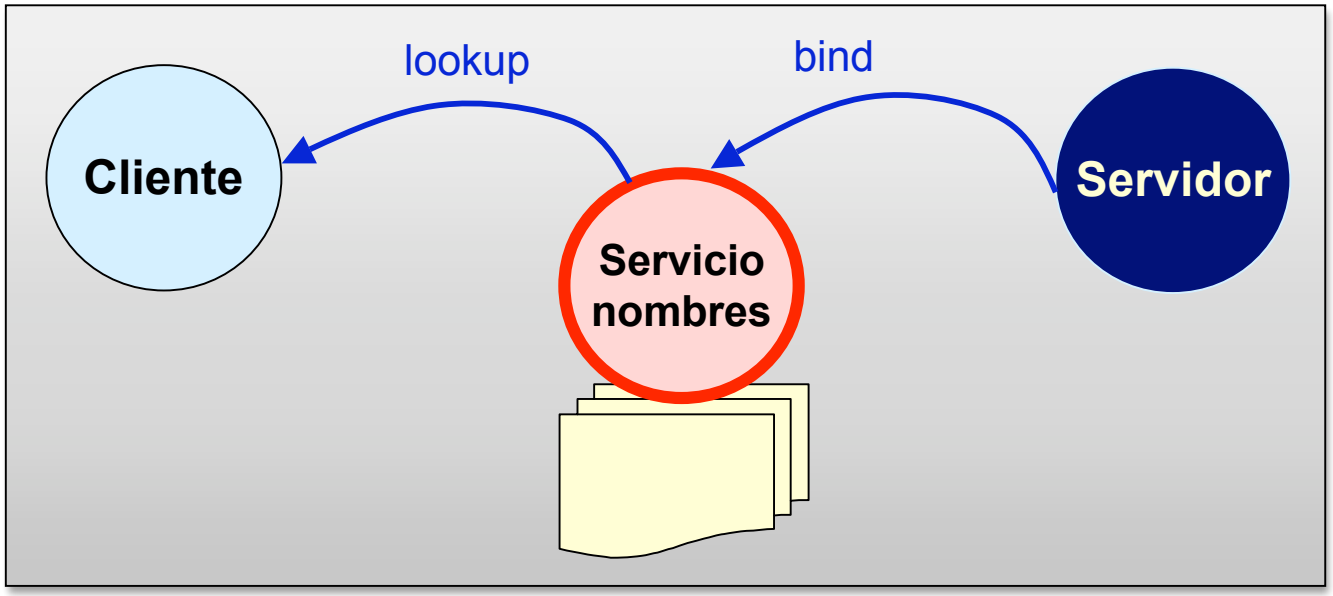
Referencias a objetos

- **El problema de *binding***
  - **Problema**: ¿Como obtener una referencia a un objeto?
  - **Posibles enfoques**:
    - ***Binding* estático**: se resuelve en tiempo de compilación. La ubicación del objeto es conocida y fija y la referencia es generada por el propio compilador.
    - ***Binding* dinámico** (*late binding*): se resuelve en tiempo de ejecución. Requiere la utilización de un *catálogo de servicios* que proporcione esta referencia en tiempo de ejecución a partir del nombre simbólico o algún otro atributo del servicio.
- **Catálogos de servicios**
  - Existen, fundamentalmente, dos tipos de catálogos:
    - **Naming service**: servicio de *páginas blancas*. Proporciona la referencia en función del nombre del objeto.
    - **Trading service:** servicio de *páginas amarillas*. Proporciona la referencia en función del servicio que proporciona el objeto.

Servicio de nombres

- **Servicio de nombres**
  - Proporciona una base de datos con tuplas:

    (nombreObjeto, ReferenciaAObjeto)
  - Se implementa como un objeto del sistema con dos métodos fundamentales:
    - **Registrar una nueva tupla: bind**(nombreObjeto, ReferenciaAObjeto)
    - **Buscar una referencia**: ReferenciaAObjeto = **lookup**(nombreObjeto)



Servicio de nombres

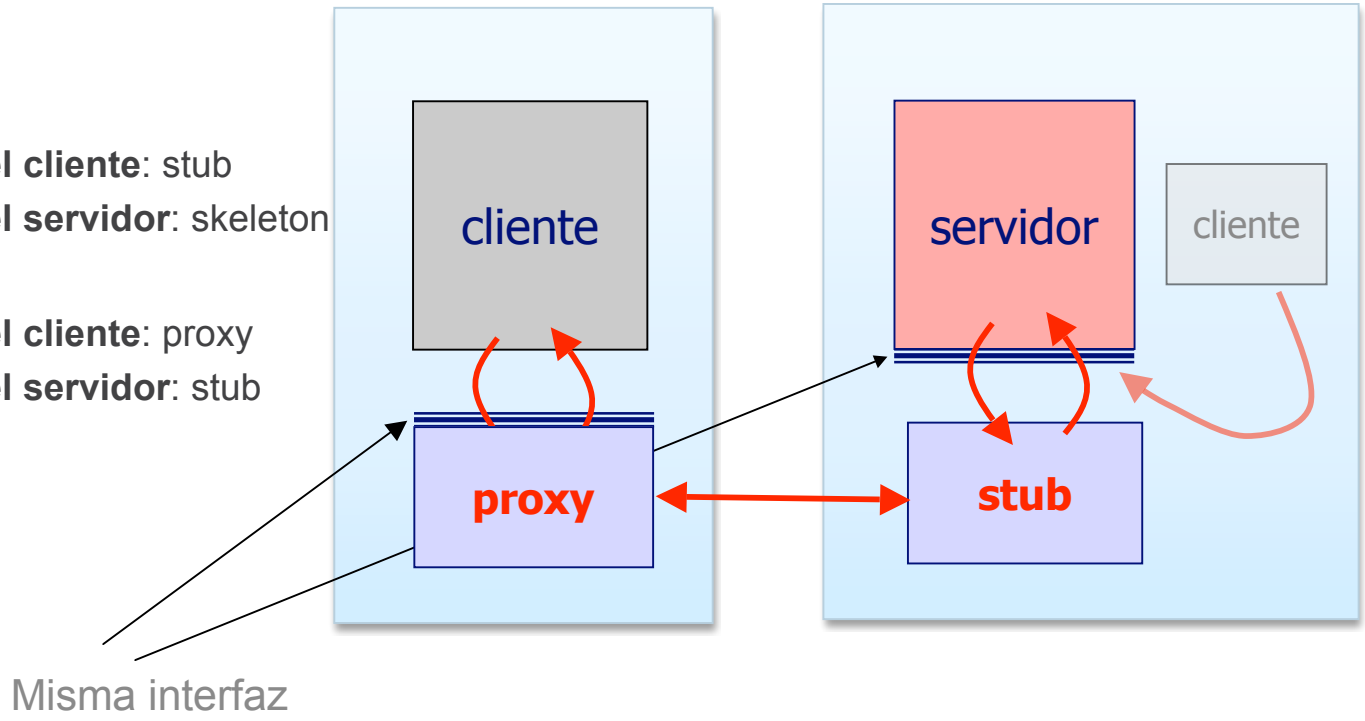
- **Nombres de objetos**

  Existen diferentes opciones para el espacio de nombres de objetos:

  - Nombres no transparentes a la ubicación:
    - **Ejemplo**: “unaMaquina:unNombre deObjeto”
  - Nombres transparentes a la ubicación:
    - Nombres planos:
      - **Ejemplo**: “object1”, “printer3”, ...
    - Espacio de nombres con estructura de árbol:
      - **Ejemplo**: “/europe/spain/upv/anObject”

Invocación de métodos remotos

- **Invocación de métodos remotos.**
  - Basado en **extender el concepto de RPC** para invocar **métodos remotos** (en vez de procedimientos)
    - Método de los *stubs*.
  - Terminología
    - Java-RMI:
      - **Stub del cliente**: stub
      - **Stub del servidor**: skeleton
    - CORBA:
      - **Stub del cliente**: proxy
      - **Stub del servidor**: stub



Invocación de métodos remotos

- **Paso de objetos por valor en invocaciones de métodos**

  Existen dos posibles esquemas:

  - **Paso por referencia**: se pasa una referencia (puntero) al objeto.
  - **Paso por valor**: implica
    - **Serializar** el objeto: convertir el objeto que se pasa como argumento en una cadena de bytes.
    - **Migrar** el objeto serializado al objeto invocado (el objeto no se “mueve”, se envía una copia serializada)
    - **Reconstruir el objeto** en el *heap* del nodo de destino (des- serializarlo).

- **Serialización de objetos**

  - La serialización de objetos es mucho más compleja cuando existe *herencia*, ya que ello implica serializar, recursivamente, todos los atributos que el objeto hereda de sus antecesores.

Invocación de métodos remotos

- **Invocación de objetos remotos**

  La invocación de objetos remotos puede adoptar dos formas:

  - **Invocación estática**: la interfaz del objeto es conocida en tiempo de compilación. La invocación del objeto es resuelta por el compilador que genera los *stubs* correspondientes.
    - Se invoca el objeto a la manera usual:
      - Obtener una referencia a un objeto de ese tipo: objRef
      - Realizar la invocación: objRef.metodo(args)
    - Proporciona una comprobación de tipos mas robusta.

  - **Invocación dinámica**: La interfaz del objeto no es conocida en tiempo de compilación. La invocación del objeto es resuelta en tiempo de ejecución. Consta de los siguientes pasos:
    - Averiguar el interfaz del objeto y seleccionar el método a invocar
    - Obtener una referencia a un objeto de ese tipo: objRef
    - Construir un objeto de tipo *invocación:* thisReq=objRef._cretate_request()
    - Incluir todos los argumentos del método a invocar: thisReq.add_value()
    - Ejecutar la *invocación:* thisReq.Invoke()

Invocación de métodos remotos

- **Invocación estática e invocación dinámica: un ejemplo**
  - Invocación estática

```
//Obtener una referencia al objeto a través de un servicio de nombres
org.omg.CORBA.Object conversorRef = nc.resolve(path);
// Invocar
conversorRef.cambiar(“euros”,”dollar”, 565)
```

  - Invocación dinámica

```
//Construir la lista de argumentos
NVList listaArgumentos = myORB.create_list(3);
Any arg1 = myORB.create_any();
arg1.insert_string("euro");
NamedValue nvArg = argList.add_value("monedaOrig", arg1, org.omg.CORBA.ARG_IN.value);

Any arg2 = myORB.create_any();
arg2.insert_string("dollar");
NamedValue nvArg = argList.add_value("monedaDest", arg2, org.omg.CORBA.ARG_IN.value);
 // etc...
 // Crear un objeto Any para albergar el resultado y empaquetarlo a en un NamedValue
Any resultado = myORB.create_any();
result.insert_string("dummy");
NamedValue resultVal = myORB.create_named_value("resultado", resultado,
                                                org.omg.CORBA.ARG_OUT.value);
// Invocar
Request thisReq = conversorRef._create_request(ctx,“cambiar", argList, resultVal);
thisReq.invoke();
```

Interfaces

- **Interfaces en sistemas de objetos distribuidos**
  - **Interfaz**: define el protocolo de comportamiento de un objeto remoto. Consta de una lista de firmas que describen los métodos que ofrece un objeto con sus correspondientes argumentos.
- **Importancia de las interfaces en sistemas distribuidos**
  - Algunos sistemas distribuidos son **heterogéneos** por naturaleza: componentes implementados en diferentes lenguajes de programación.
    - Esto hace necesario la utilización de un "lenguaje neutral" para declarar las interfaces de los servicios y posibilitar su interoperabilidad.
  - Algunos componentes pueden ser obtenidos directamente a partir del **procesamiento de interfaces.**

Interfaces

- **Lenguajes de definición de interfaces**
  - Son lenguajes declarativos que permiten especificar el interfaz de un objeto de una manera estándar e independiente de su lenguaje de implementación.
  - Establecen la naturaleza de las posibles interacciones entre objetos (atributos, métodos, excepciones, ...)
  - Estandariza los tipos de datos (enteros, reales, ...) y su implementación.
- **IDL (*Interface Definition Language*)**
  - Lenguaje de especificación de interfaces muy próximo a C++ pero sin punteros
    - Ver transparencias IDL
  - Definido originalmente en el proyecto DCE y estandarizado por CORBA
  - Existen *bindings* definidos para C, C++, Ada, Smalltalk, ...

Interfaces

- **Procesamiento de interfaces**
  - Puede generar de forma automática:
    - *Stubs* de cliente y servidor
    - Entradas para catálogos de servicios
    - Interfaz en otro lenguaje de programación