Comunicación en Sistemas Distribuidos

Fernando Pérez Costoya

Índice

- Paso de mensajes
  - Comunicación punto-a-punto (*unicast*)
    - Repaso de sockets en tema independiente (**prácticas individuales**)
  - Comunicación de grupo
  - Sistemas de colas de mensajes
- Llamadas a procedimientos remotos (RPC)
- Invocación de métodos remotos (RMI)
  - Java RMI (**prácticas en grupo**)
- Servicios web
- Memoria compartida distribuida

Tema para apoyo a las prácticas
No entra en examen

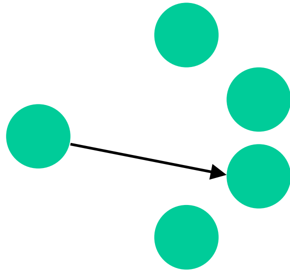
Paso de mensajes

Introducción al paso de mensajes

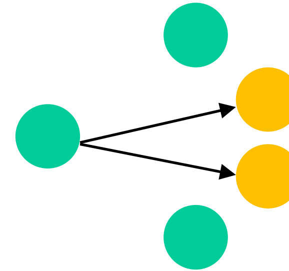
- API para envío/recepción de mensajes. Alternativas de diseño:
  - Cardinalidad: *unicast*, *anycast*, *multicast*
  - Persistencia: ¿Se guardan mensajes hasta que aparezca receptor?
    - Posibilita el desacoplamiento temporal
- Paradigmas de comunicación por paso de mensajes
  - *unicast* no persistente: sockets (repaso en tema independiente)
    - No desacoplamiento espacial ni temporal
  - *multicast* no persistente: comunicación de grupo
  - *anycast* o *multicast* persistente: sistemas colas de mensajes
- ¿Sistema de paso de mensajes maneja heterogeneidad?
  - Corresponde a nivel de presentación OSI
  - Si no (p.e. sockets), debe hacerlo la aplicación
    - La aplicación debe realizar la *serialización* de datos

Cardinalidad

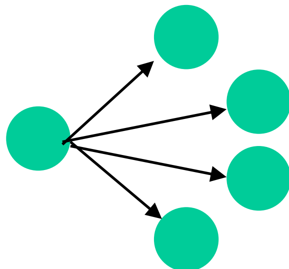
unicast



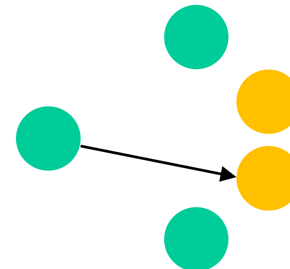
multicast



broadcast



anycast



Serialización de datos

- "Serializar": convertir estructuras de datos en
  - secuencias de bytes para transmitir (o almacenar en un fichero)
  - y también transformar formato de datos para que nodos se entiendan
  - En envío, *serializar*: formato local → formato convenido
  - En recepción, *deserializar*: formato convenido → formato local
- Procesadores, lenguajes, compiladores difieren en
  - Orden de bytes en tipos numéricos (*endian*)
  - Organización estructuras datos (compactación, alineamientos…)…
- Alternativas en formato de representación de datos enviados
  - Texto vs binario, solo datos o también tipos y nombre de campos…
- Ejemplos de formato de *serialización*:
  - XDR (RFC 1832): binario; solo transmite datos (RPC de Sun)
  - XML: texto
  - JSON: texto
  - *Protocol Buffers* (Google): binario

Se estudia en Sistemas orientados a servicios

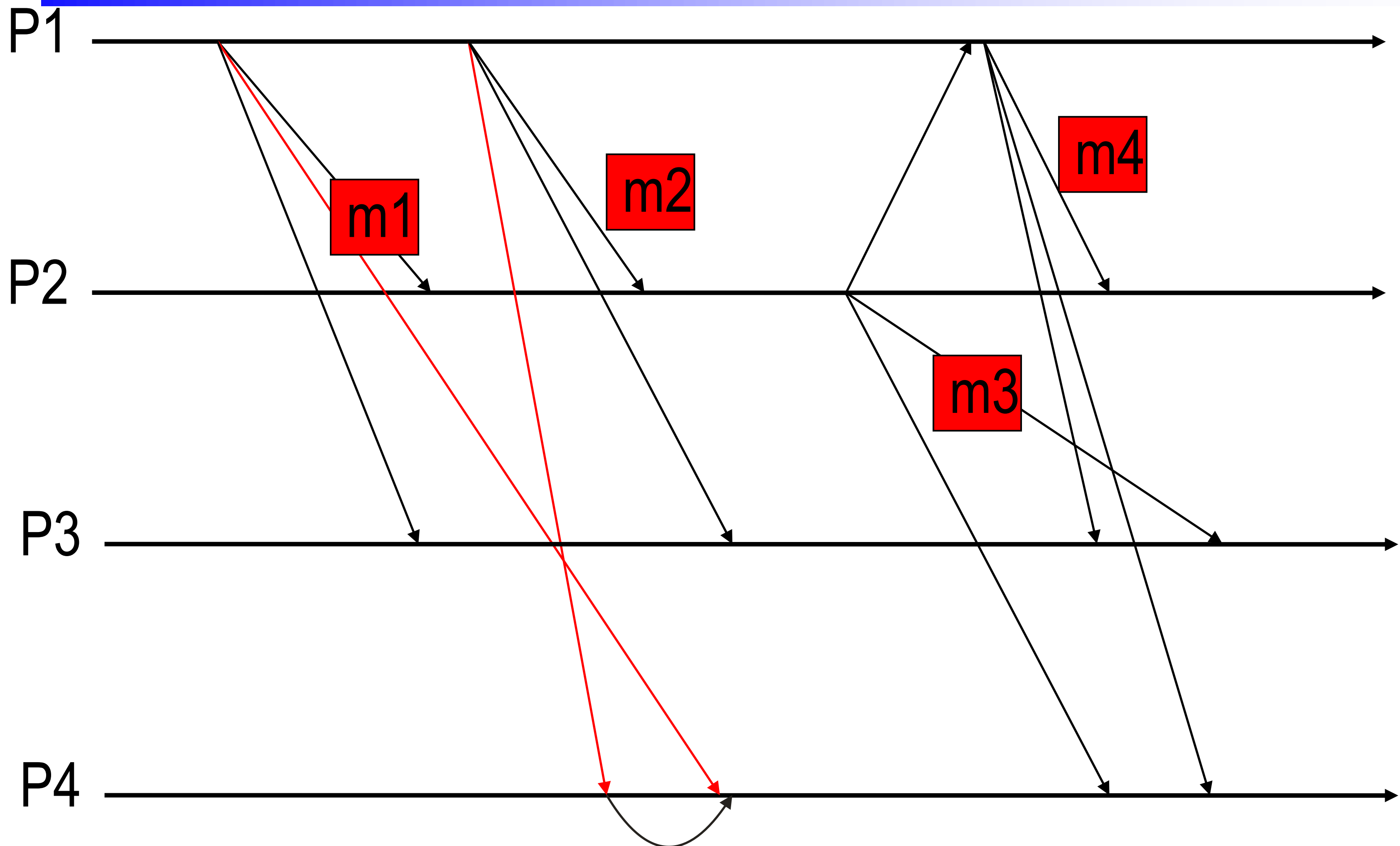
Multidifusión: comunicación de grupo

- Destino de mensaje → grupo de procesos
  - Procesos se hacen miembros del grupo y se dan de baja
  - Desacoplamiento espacial pero no temporal
- Ofrece mucha mayor funcionalidad que *multicast IP*
  - aunque, normalmente, se implementa sobre ese mecanismo
- Trabajo seminal: ISIS (posteriores Horus, Ensemble, Jgroups)
- Simplifica desarrollo de diversas aplicaciones:
  - Facilita gestión de réplicas
    - Réplicas son miembros del mismo grupo
    - Cliente envía operaciones al grupo
  - Posibilita la implementación de modelo editor/subscriptor
    - Uso de 1 grupo/tema
    - Subscriptor se hace miembro del grupo
    - Editor envía mensajes al grupo

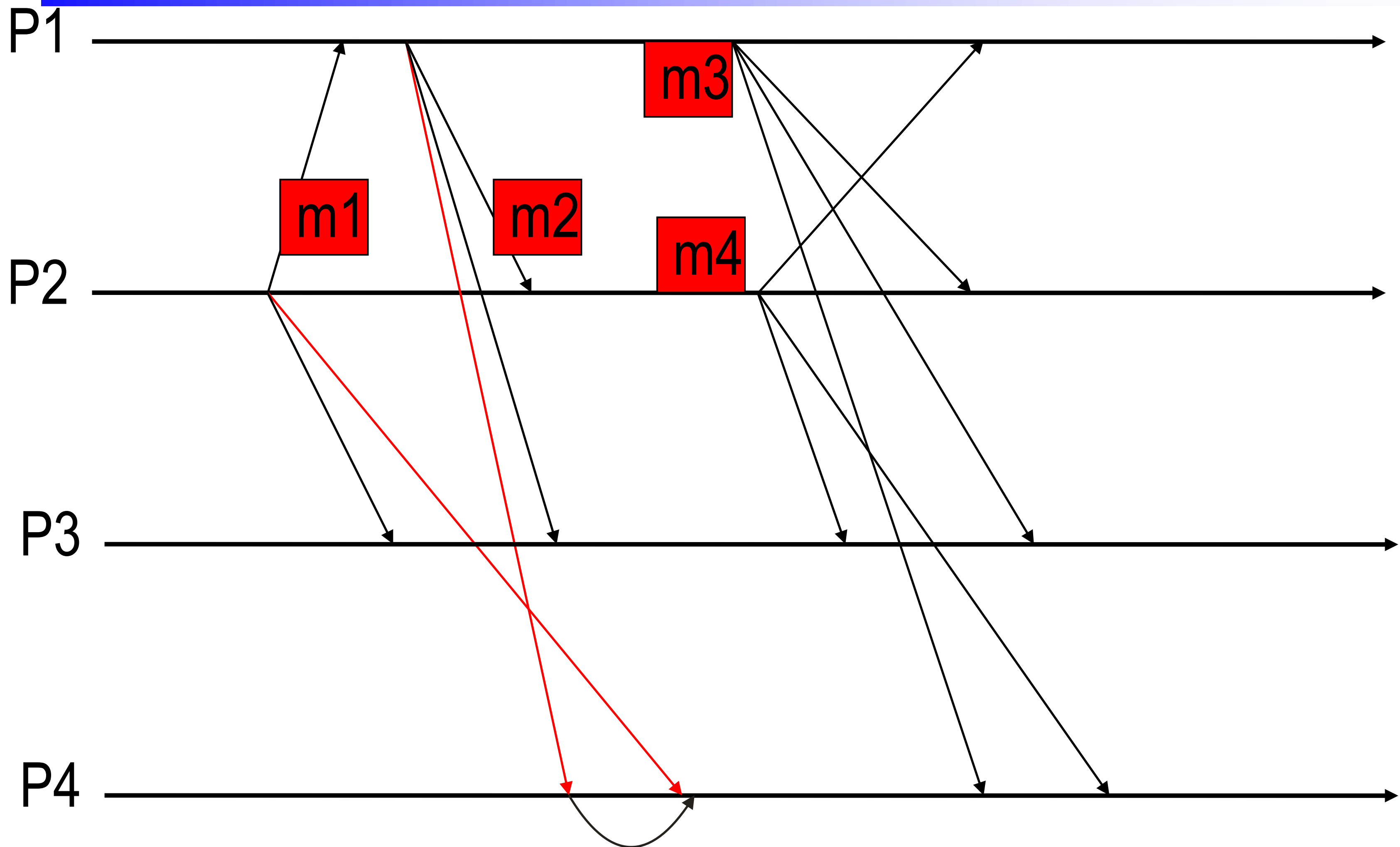
Funcionalidad de com. de grupo

- Garantiza atomicidad: o todos reciben el mensaje o ninguno
  - *multicast* IP no: usa UDP, que puede perder mensajes
- Permite seleccionar el orden de recepción de los mensajes
  - *multicast* IP no garantiza ningún orden
  - Orden más estricto → mayor sobrecarga
  - **FIFO**: mensajes de misma fuente llegan en orden de envío
    - No garantía sobre mensajes de distintos emisores
  - **Causal**: entrega respeta relación “causa-efecto”
    - Si no hay relación, no garantiza ningún orden de entrega
  - **Total**: Todos los mensajes recibidos en mismo orden por todos
- Gestiona carácter dinámico del grupo
  - La pertenencia se coordina con la comunicación
  - *multicast* IP no garantiza esa coordinación

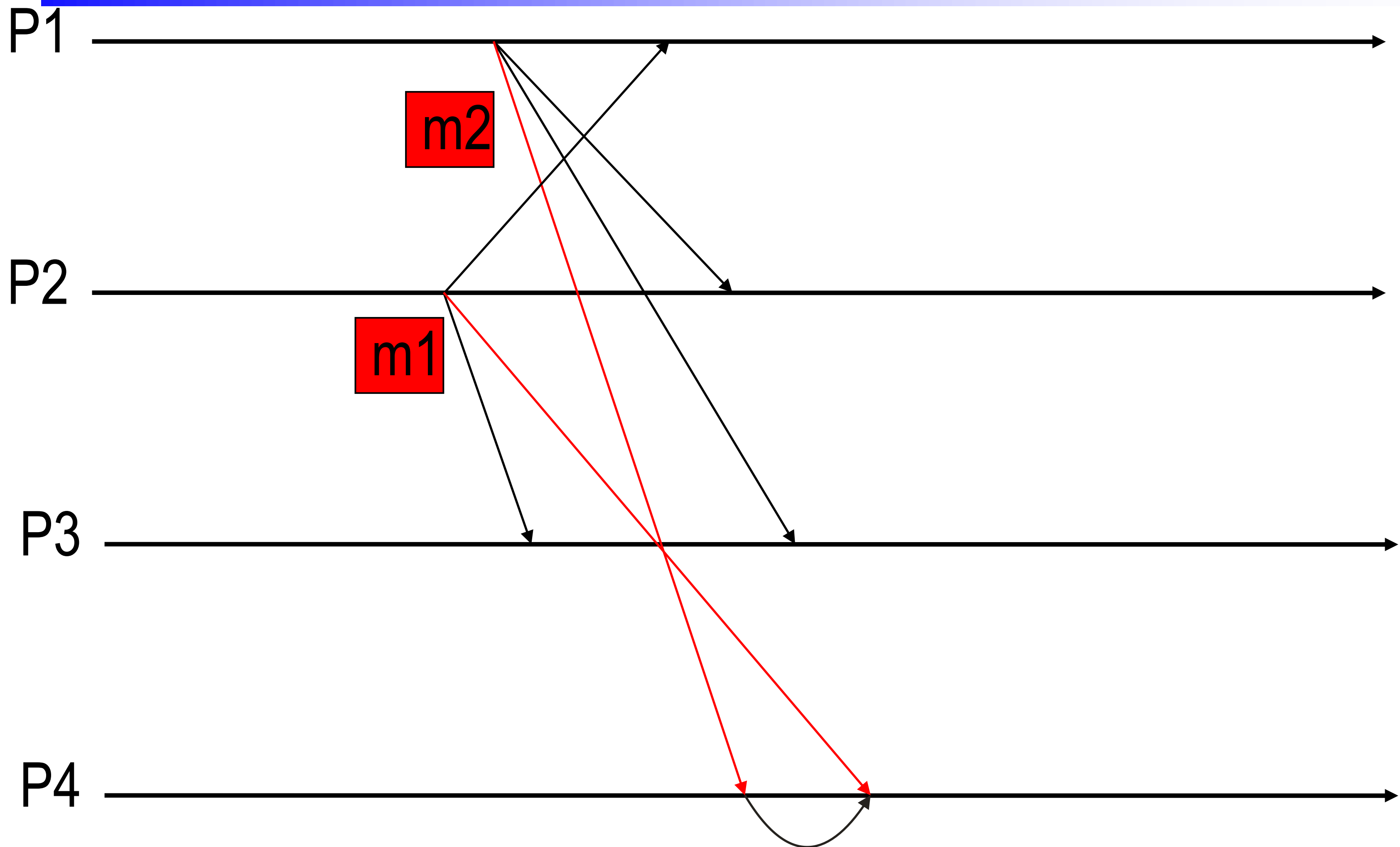
Orden FIFO



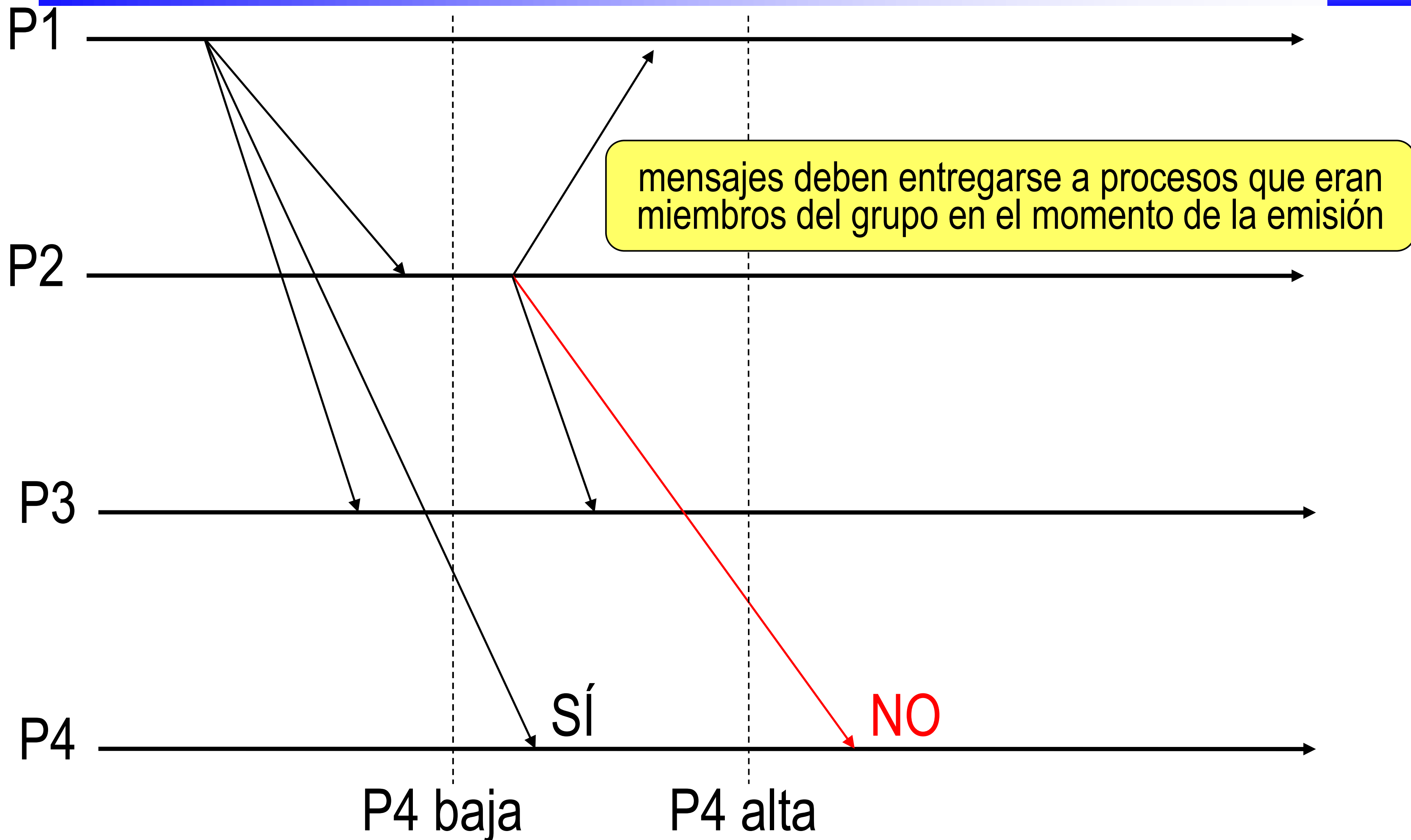
Orden causal



Orden total



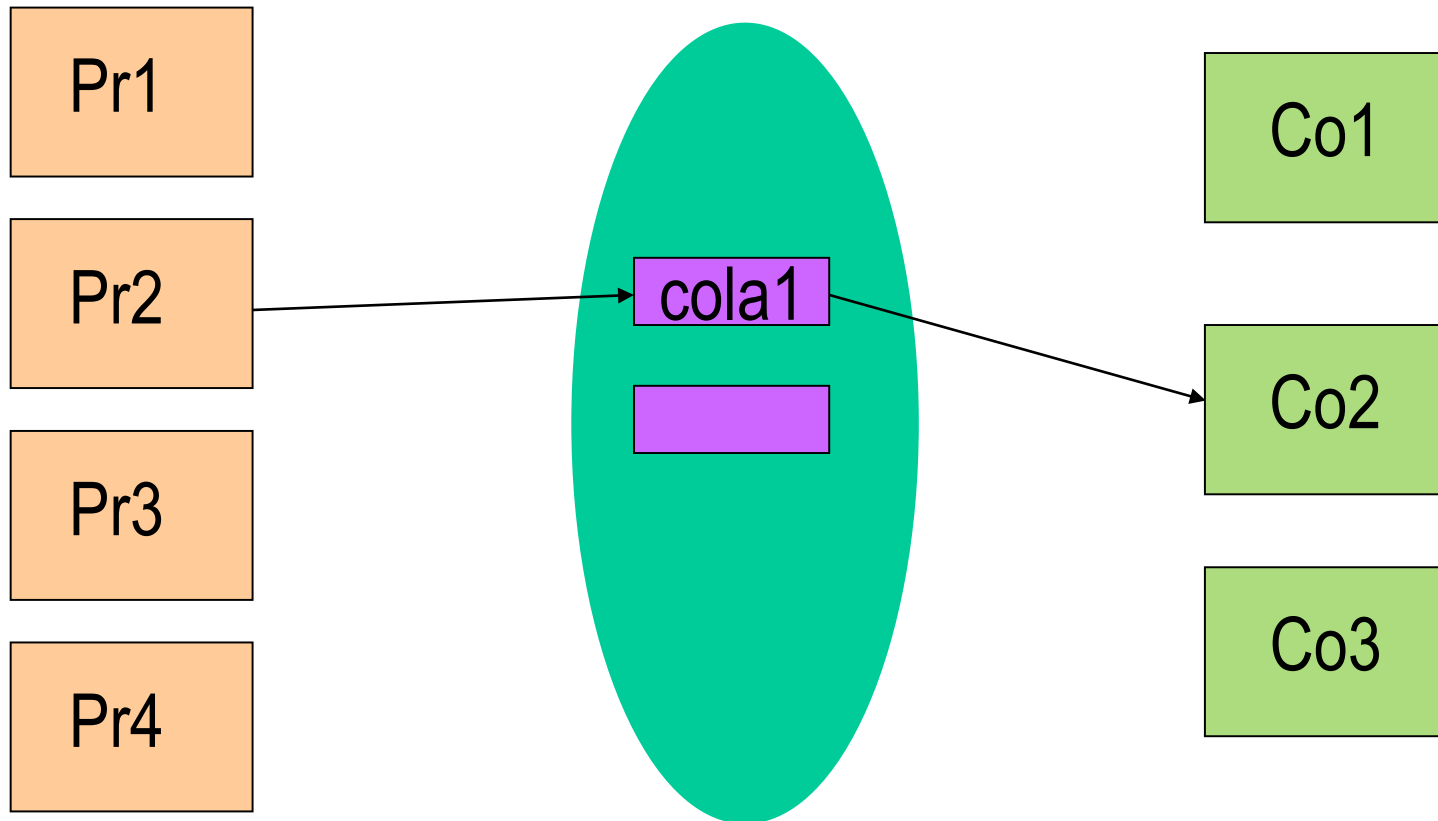
Gestión de miembros



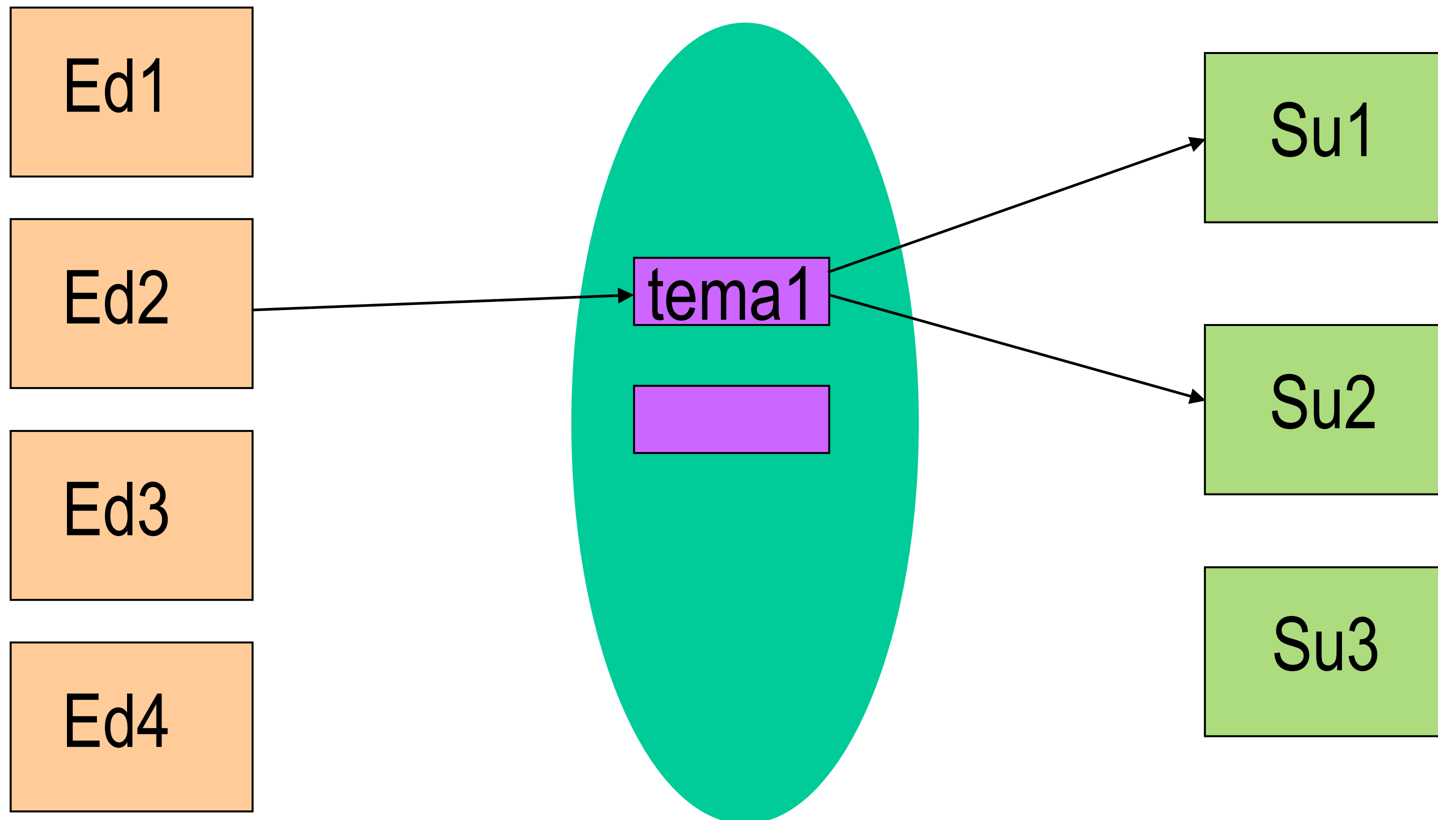
Sistemas de colas de mensajes

- Es un tipo de *Message-oriented middleware* (MOM)
  - P.e. *ActiveMQ, RabbitMQ* (basado en el estándar *ISO AMQP*)
- Procesos envían y reciben mensajes de colas
  - Uso de cola → desacoplamiento espacial
  - Mensajes se guardan hasta ser leídos → desacoplamiento temporal
- Soporte directo a modelo productor/consumidor (*anycast*)
- Algunos sistemas ofrecen extensión para editor/subscriptor
  - Permiten asociar tema al crear una cola
  - Editores envían mensaje al tema asociado a una cola
  - Mensaje es recibido por **todos** los subscriptores de ese tema

Modelo productor/consumidor



Modelo editor/subscriptor



RPC: Llamada a procedimiento remoto

Fernando Pérez Costoya

Provisión de servicios en local

Aplicación

```
int main(...) {
    ....
    r=op1(p, q, ...);
    .........
}
```

Biblioteca

```
t1 op1(ta a, tb b, ...) {
    .....
    return r1;
}
t2 op2(tx x, ty y, ...) {
    .....
    return r2;
}
...........
```

Provisión de servicios (C/S) en S. Dist.

Cliente

```
int main(...) {
  Mensaje msj,resp;
  ..........
  dir_srv=busca(IDservicio);
  msj.op=OP1;
  msj.arg1=p;
  msj.arg2=q;
  serializacion(&msj);
  envío(msj,dir_srv);
  recepción(&resp, NULL);
  deserialización(&resp);
  r=resp.r;
  .........
}
```

Servidor

```
int main(...) {
  Mensaje msj,resp;
  alta(IDservicio,dir_srv);
  while (TRUE) {
    recepción(&msj,&dir_clie);
    deserialización(&msj);
    switch(msj.op) {
    case OP1:
      resp.r=op1(msj.arg1,...);
    case OP2:
      resp.r=op2(msj.arg1,...);
    serialización(&resp);
    envío(resp,dir_clie);
  }
}
t1 op1(ta a, tb b, ...) {
}
...........
```

Fundamento de las RPC

- Código añadido a provisión de servicios en SD
  - Es independiente de la implementación del cliente y del servidor
  - Solo depende de la interfaz de servicio
  - Puede generarse automáticamente a partir de la misma
- Objetivo de las RPC: Provisión de servicios igual que en local
  - Como en local, solo se programa bibliotecas de servicio y aplicaciones
  - Código restante generado automáticamente (resguardos: *stubs*)
    - Se encarga de mensajería, serialización, interacción con *binder*, autenticación, comportamiento ante fallos, gestión de concurrencia…
  - Lograr semántica convencional de llamadas a procedimiento en SD
  - Invocación de funciones remotas no debe ser totalmente transparente
    - Tratamiento de errores de comunicación
    - Ser conscientes de que llamada tiene latencia y consume ancho banda
- Surge en Xerox PARC (1981)

Provisión de servicios en SD con RPC

aplicación

```
int main(...) {
    ....
    r=op1(p, q, ...);
    .........
}
```

resguardo cliente

```
init() {
  dir_srv=busca(IDservicio);
}
t1 op1(ta a, tb b, ...) {
  Mensaje msj,resp;
  msj.op=OP1;
  msj.arg1=a; msj.arg2=b;
  serializacion(&msj);
  envío(msj,dir_srv);
  recepción(&resp, NULL);
  deserialización(&resp);
  return resp.r;
}
```

resguardo servidor

```
int main(...) {
  Mensaje msj,resp;
  alta(IDservicio,dir_srv);
  while (TRUE) {
    recepción(&msj,&dir_clie);
    deserialización(&msj);
    switch(msj.op) {
    case OP1:
      resp.r=op1(msj.arg1,...);
    case OP2:
      resp.r=op2(msj.arg1,...);
    serialización(&resp);
    envío(resp,dir_clie);
  }
}
```

biblioteca

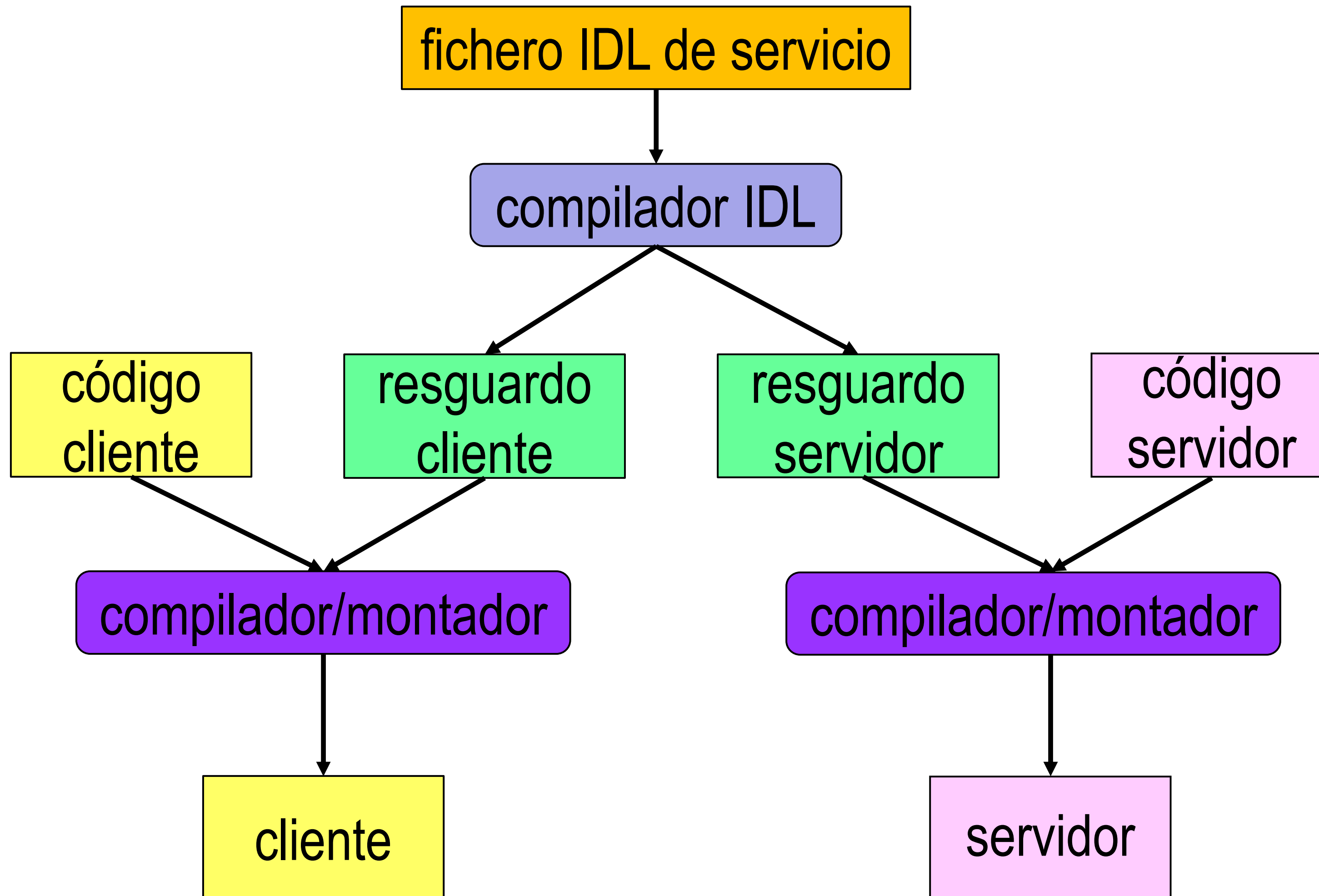
```
t1 op1(ta a, tb b, ...) {
}
...........
```

Modo de operación de RPC

- Punto de partida: fichero con especificación de servicio
  - Puede estar escrito en lenguaje de programación convencional
  - O usar un lenguaje específico para definir interfaces de servicio:
    - *Interface Definition Language* (IDL)
    - Directivas específicas (por ejemplo, marcar función como idempotente)
- En tiempo de construcción del programa:
  - Generador automático de código: compilador IDL
    - Definición de interfaz de servicio → Resguardos
    - Heterogeneidad: genera resguardos para múltiples lenguajes
    - *App* se enlaza con el resguardo de cliente y biblioteca con el del servidor
- Sun: *Open Network Computing* (1985; RFC 1831)
  - RPC de ONC es la base para servicios NFS o NIS
  - XDR: define IDL y formato serialización binario que solo envía datos
  - *rpcgen*: compilador IDL
  - Uso de un *binder* local (en Linux demonio *portmap* o *rpcbind*)

  http://laurel.datsi.fi.upm.es/~ssoo/SOD.dir/practicas/guiarpc.html

Generación de programas que usan RPCs



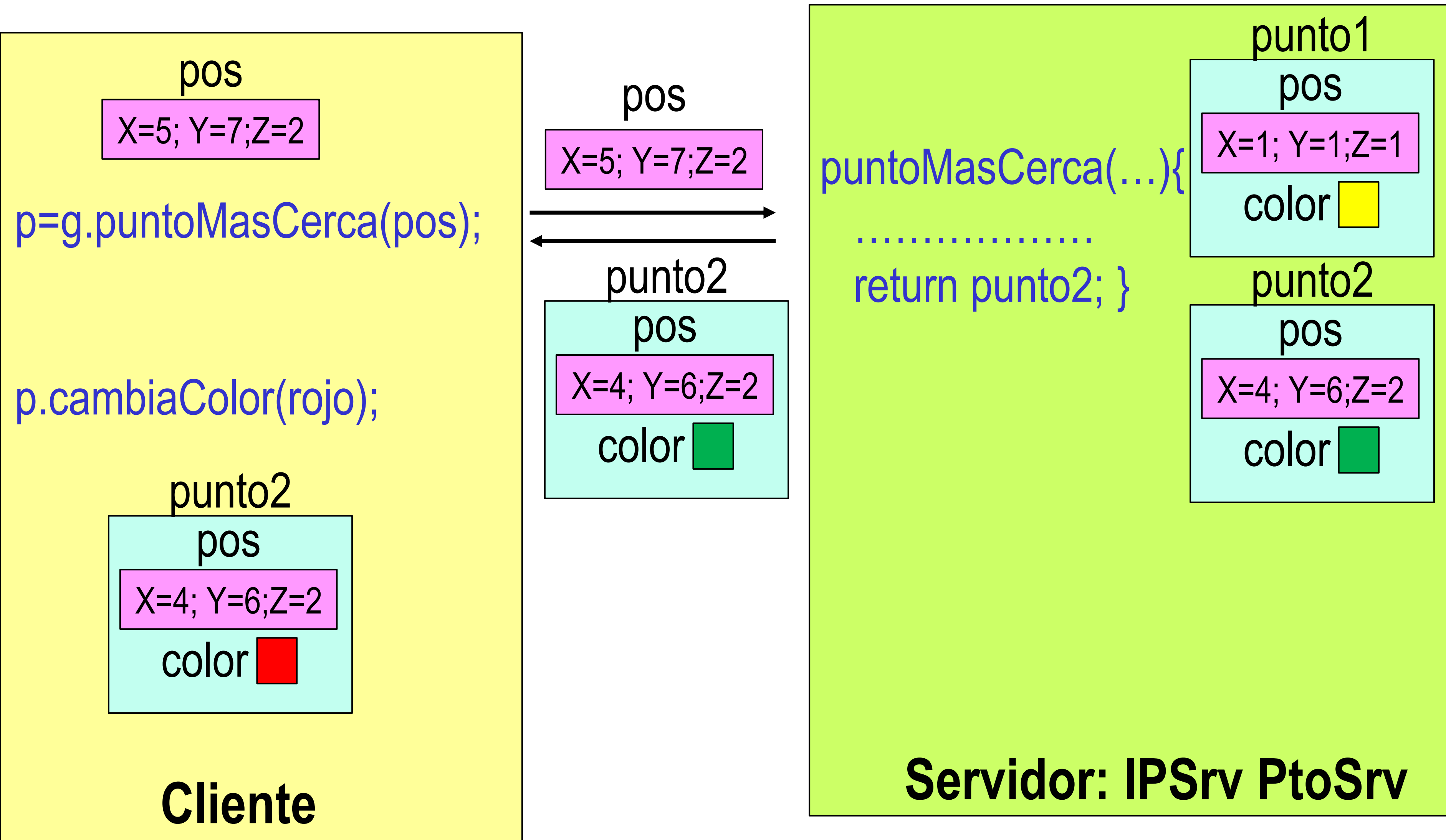
RMI: Invocación de método remoto

- Java RMI

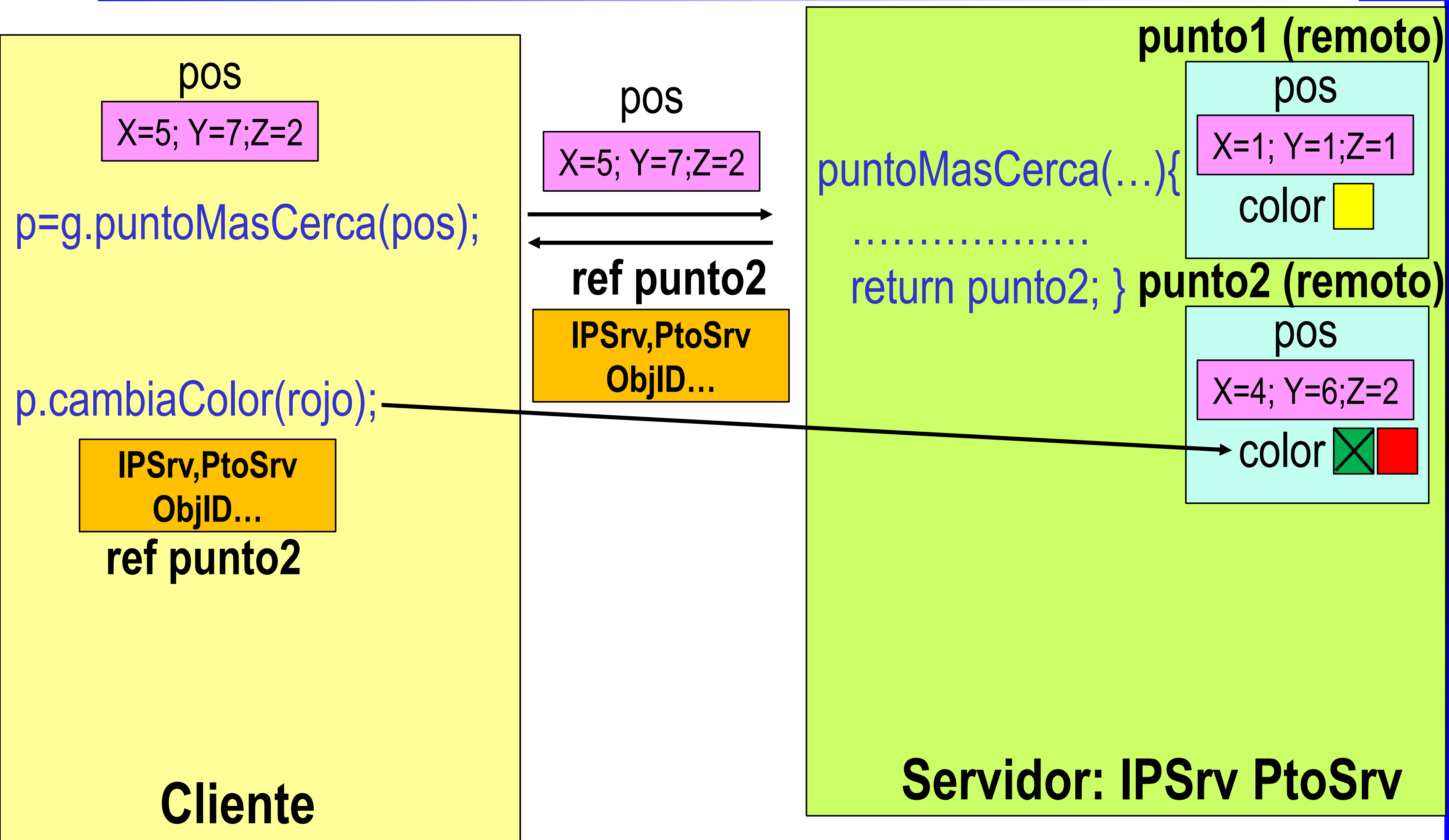
De RPC a RMI

- Evolución natural: RPC → RMI
  - Programación procedimental → Programación orientada a objetos
  - Extensión de la potencia de la POO al sistema distribuido
- Aplicable todo lo comentado sobre RPCs
- Serialización de objetos es una extensión de la de estructuras:
  - En invocación: envía copias de objetos usados como parámetros
  - En retorno: envía de vuelta copias de objetos y excepciones devueltos
- Pero hay una importante diferencia: los objetos remotos
  - Si un objeto (una clase) se marca como remoto
  - Cuando se pasa como parámetro en invocación de método
  - o cuando se devuelve como resultado en retorno
    - No se envía copia del objeto sino una referencia remota al objeto
  - Cuando se llama a método usando una referencia remota
    - Se produce la invocación remota

Serialización de objetos



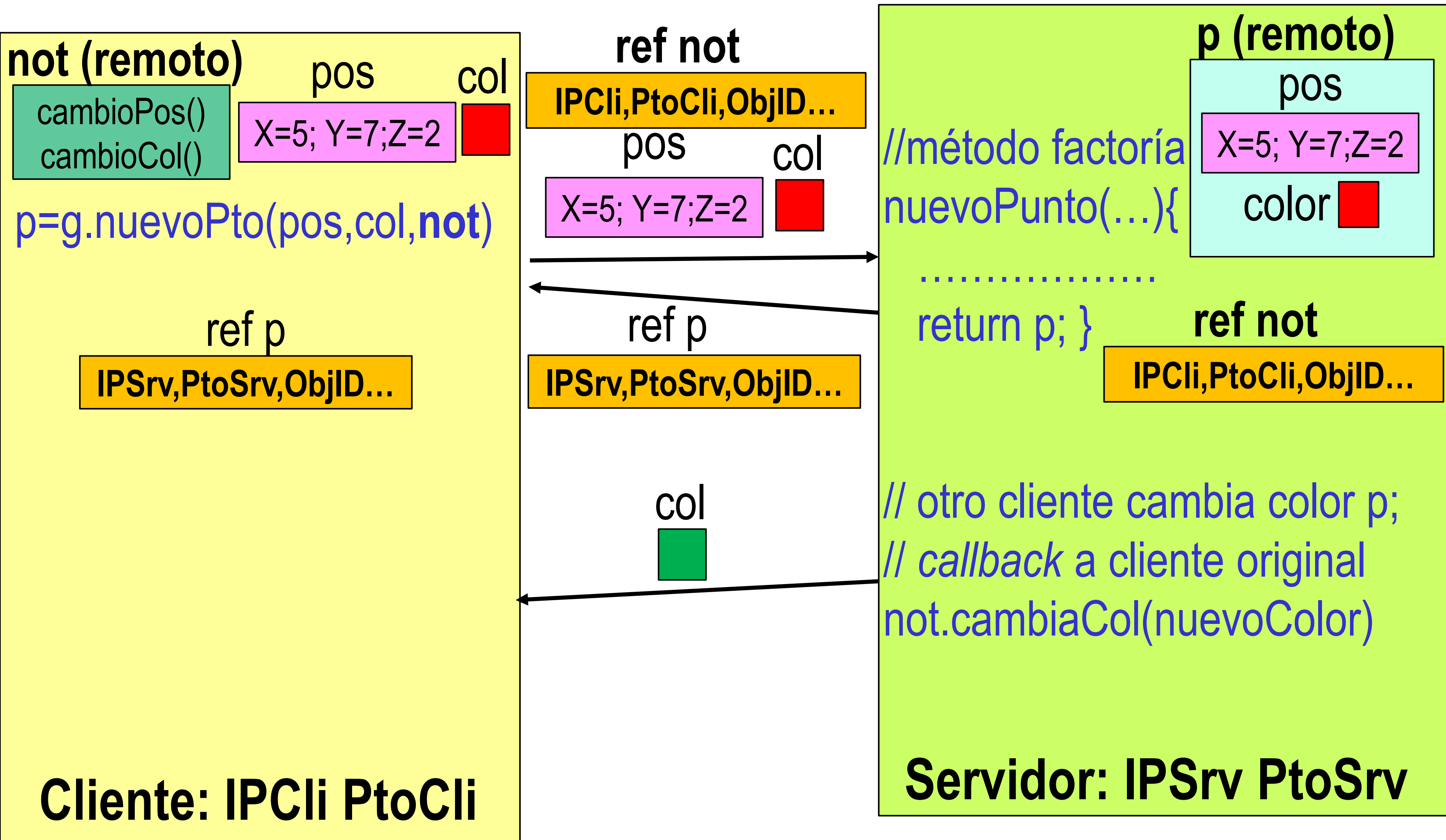
Serialización de objetos remotos



Objetos remotos

- Como en RPC, punto de partida: definición de interfaz de servicio
  - Especifica conjunto de métodos que pueden invocarse de forma remota
- Clase implementa la interfaz: instancias son objetos remotos
- Serialización de objeto remoto ya sea parámetro o valor retorno
  - Genera referencia a objeto remoto: ID único que permite acceso remoto
    - p.e. *IP servidor | puerto servidor | ID-objeto | ID-interfaz-servicio*
- Invocación método con referencia remota: invocación resguardo
  - método del resguardo usa info referencia remota para contactar servidor
- ¿Cómo obtener 1ª referencia remota?: Uso de *binder*
  - *RPC: IDservicio→IP|puerto; RPC: IDservicio→IP|puerto|IDobj|IDinterfaz*
- Facilita implementar distintos patrones habituales:
  - *Callback* (servidor invoca método de cliente)
  - Creación remota de objetos remotos: método factoría (o fábrica)
- Necesidad de recolector de basura distribuido

Ejemplo método factoría y *callback*



Java RMI

- Implementación de RMI en la distribución estándar de Java
- Solución restringida a entornos distribuidos basados en Java
- Aprovecha funcionalidades avanzadas de Java
  - Como, por ejemplo, reflexión y descarga dinámica de código de clases
- Generación automática resguardos: *proxy* (clnt) y *skeleton* (srv)
  - En versiones iniciales, era necesario usar compilador IDL (*rmic*)
- Definición de clase que proporciona acceso remoto:
  - Se especifica una interfaz que hereda de la interfaz *Remote*
    - Solo serán accesibles de forma remota métodos definidos en esa interfaz
  - Se crea una clase que implementa esa interfaz
- Por defecto, usa un *binder* local: *RMI Registry*
  - Asocia nombre (URL) con objeto que implementa una interfaz remota

http://laurel.datsi.fi.upm.es/~ssoo/SD.dir/practicas/guiarmi.html

Ejemplo servicio Eco: interfaz de servicio

ServicioEco.java

```
import java.rmi.*;

interface ServicioEco extends Remote {
        String eco (String s) throws RemoteException;
}
```

Ej. servicio Eco: implementación servicio

ServicioEcoImpl.java

```
import java.rmi.*;
import java.rmi.server.*;
class ServicioEcoImpl extends UnicastRemoteObject
                implements ServicioEco {
   ServicioEcoImpl() throws RemoteException { } // constructor
   public String eco(String s) throws RemoteException {
      return s.toUpperCase();
   }
}
```

Alternativa: **NO** subclase de *UnicastRemoteObject*
Uso de método estático *exportObject* de *UnicastRemoteObject*

Fernando Pérez Costoya

Ejemplo servicio Eco: servidor

ServidorEco.java

Instancia y da de alta la implementación

```
import java.rmi.*;
import java.rmi.server.*;
class ServidorEco  {
    static public void main (String args[]) {
      if (args.length!=1) {System.err.println("Uso: ServidorEco numPuertoRegistro"); return;}
       if (System.getSecurityManager() == null) {
           System.setSecurityManager(new SecurityManager());   }
       try {
           ServicioEco srv = new ServicioEcoImpl();
           Naming.rebind("rmi://localhost:" + args[0] + "/Eco", srv);  }
       catch (RemoteException e) {
           System.err.println("Error de comunicacion: " + e.toString()); System.exit(1);  }
       catch (Exception e) {
           System.err.println("Excepcion en ServidorEco:"); e.printStackTrace(); System.exit(1);  }
    } // main
} // clase
```

División arbitraria entre implementación de servicio y servidor

P.e. podría usarse única clase que fusione *ServicioEcoImpl* y *ServidorEco*

Ejemplo servicio Eco: cliente

ClienteEco.java

```
import java.rmi.*;
import java.rmi.server.*;
class ClienteEco {
    static public void main (String args[]) {
        if (args.length<2) { System.err.println("Uso: ClienteEco hostReg numPuertoReg ...");
            return; }
        if (System.getSecurityManager() == null)
            System.setSecurityManager(new SecurityManager());
        try {
            ServicioEco srv = (ServicioEco) Naming.lookup("//"+ args[0] + ":" + args[1] + "/Eco");
            for (int i=2; i<args.length; i++) System.out.println(srv.eco(args[i])); // RMI }
        catch (RemoteException e) {
            System.err.println("Error de comunicacion: " + e.toString()); }
        catch (Exception e) {
            System.err.println("Excepcion en ClienteEco:"); e.printStackTrace(); }
    } // main
} // clase
```

Ejemplo servicio Eco: políticas seguridad

cliente.permisos

servidor.permisos

```
grant {
    permission java.security.AllPermission;
};
```

- Para simplificar, permitimos todas las operaciones
  - Tanto en el cliente como en el servidor

Ej. servicio Eco: compilación y ejecución

servidor

```
cd servidor
javac *.java
rmiregistry 54321 & // por defecto, usa el puerto 1099
java -Djava.security.policy=servidor.permisos  ServidorEco 54321
```

---

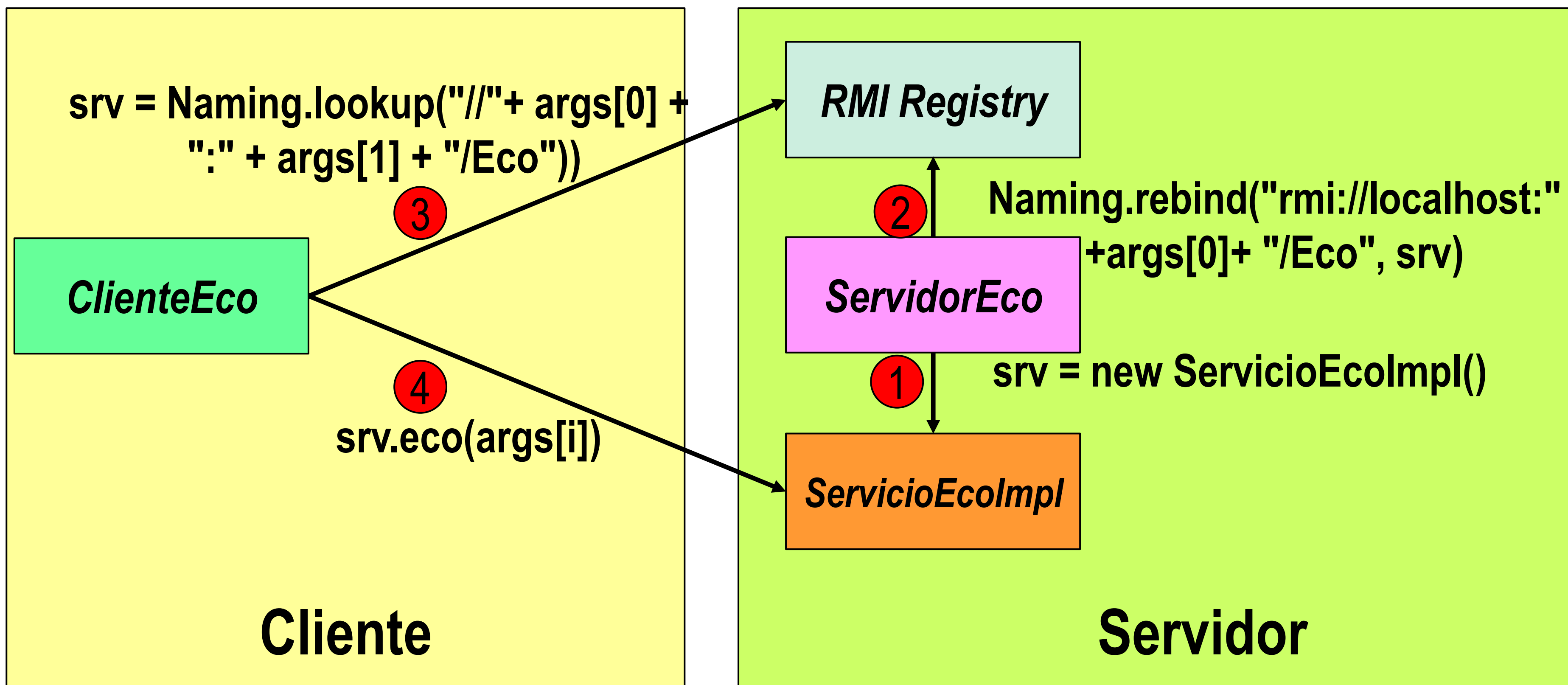
cliente

```
cd cliente
javac *.java
java -Djava.security.policy=cliente.permisos  ClienteEco localhost 54321 hola adios
```

HOLA

ADIOS

Modo de operación de *Registry*



Sistemas Distribuidos

Servicios web

Fernando Pérez Costoya

Servicios web

- Básicamente, RPC sobre HTTP
- Uso de HTTP integra la operación en la plataforma web
  - Múltiples beneficios: p.e. compatibilidad con cortafuegos o *proxies*
- Tipos de servicios web
  - Servicios con APIs específicas (SOAP)
    - Solución “natural”: API de servicio con un método por cada operación
    - Formato de representación XML
  - Servicios con APIs orientadas a recursos (REST)
    - API de servicio siempre con mismos métodos: CRUD
    - Pero aplicados a los distintos objetos de la aplicación
    - Diseño de servicios sin estado
    - Formato de representación más frecuente JSON

Se estudian en Sistemas orientados a servicios

Memoria compartida distribuida

Fernando Pérez Costoya

Memoria compartida distribuida

- Falta de memoria compartida obliga a buscar nuevas soluciones
  - Para gestionar procesos, sincronizarlos, manejar ficheros…
  - Es la razón de esta asignatura
- Alternativa: capa que cree memoria compartida en SD
  - Permitiría aplicar soluciones tradicionales (p.e. semáforo para sincro.)
- ¿Cómo implementarla? Esbozo de posible solución:
  - Fallo de página contacta con nodo remoto para descargar página
  - Se sustituyen mensajes explícitos entre los procesos
  - por mensajes internos que mueven páginas entre los nodos
- Difícil implementación eficiente
  - Uso limitado en entornos homogéneos de aplicaciones paralelas
  - Ejemplo: biblioteca *Treadmarks*

Memoria compartida distribuida (MCD)