Comunicación en SD 8

Sistemas Operativos y Distribuidos

Mg. Javier Echaiz
D.C.I.C. – U.N.S.
http://cs.uns.edu.ar/~jechaiz
je@cs.uns.edu.ar

Road Map

- Pasaje de Mensajes.
- Modelo Cliente-Servidor.
- Llamadas a Procedimiento Remoto (RPC).
- Comunicación en Grupo.

Comunicación en Sistemas Distribuidos

Permite la interacción entre aplicaciones y servicios del sistema.

Modelos de comunicación entre procesos:

- Memoria compartida (Sólo uni/multiprocesador no distribuido).
- Paso de mensajes.

El nivel de abstracción en la comunicación:

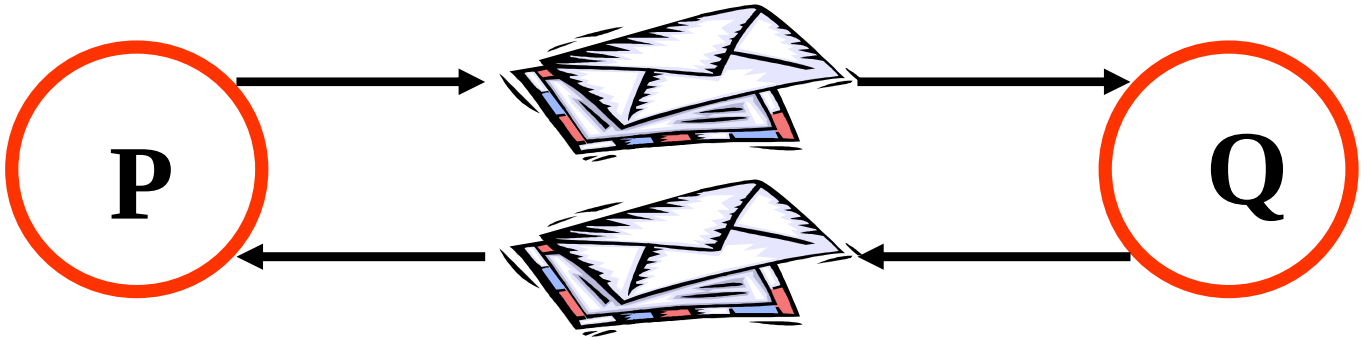
- Paso de mensajes puro (Cliente-Servidor).
- Llamadas a procedimientos remotos.
- Modelos de objetos distribuidos.

Comunicación en Sistemas Distribuidos

La comunicación entre procesos necesita compartir información:

a) datos compartidos

b) pasajes de mensajes o copias compartidas



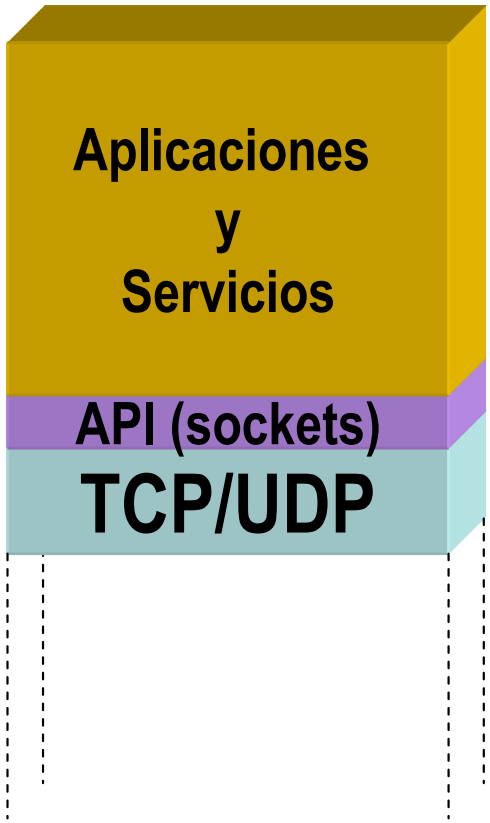
Factores de Comunicación

Los diferentes mecanismos de comunicación se caracterizan por los siguientes factores:

- Rendimiento: Latencia, ratio de transferencia, ancho de banda, ...
- Escalabilidad: Número de elementos activos.
- Fiabilidad: Pérdida de mensajes.
- Seguridad: Cifrado, certificación, ...
- Movilidad: Equipos móviles.
- Calidad de Servicio (*QoS*): Reserva y garantía de anchos de banda.
- Comunicación en grupo: *Multicast*.

Niveles de Comunicación

1) Pasaje de mensajes puro. Aplicaciones en red.



2) Funcionalidades de comunicación de bajo nivel. Sistemas Operativos Distribuidos.

3) Llamadas a procedimientos remotos y objetos distribuidos.

Primitivas de Comunicación

Cada una de las funciones de comunicación de una tecnología determinada. Las primitivas básicas son:

- Envío: **`send(`*`destino`*`,`*`mensaje`*`)`**.
- Recepción: **`receive(`*`fuente`*`,`*`mensaje`*`)`**.

Otras primitivas:

- Conexión: **`connect(`*`destino`*`)`**.
- Desconexión: **`close()`**.

Cada una de las primitivas tiene las siguientes características:

- Boqueantes vs No-bloqueantes.
- Síncronas vs Asíncronas.
- Fiables vs No-fiables.

Bloqueantes vs. No-bloqueantes

Las características de bloqueo son:

- **Primitivas bloqueantes:** La operación bloquea al elemento que la solicita hasta que ésta sea completada.
- **Primitivas no-bloqueantes:** La operación no detiene la ejecución del elemento que la solicita.

Las llamadas no bloqueantes tienen distinto sentido dependiendo de la primitiva que se trate:

- **Envío no bloqueante:** El emisor almacena el dato en un buffer del núcleo (que se encarga de su transmisión) y reanuda su ejecución.
- **Recepción no bloqueante:** Si hay un dato disponible el receptor lo lee, en otro caso indica que no había mensaje.

Síncronas vs. Asíncronas

Esta característica afecta no tanto a la primitiva como a la transmisión en sí.

- **Comunicación síncrona:** Envío y recepción se realizan de forma simultanea.
- **Comunicación asíncrona:** El envío no requiere que el receptor este esperando.

La comunicación asíncrona usa un *buffer* de almacenamiento.

Implica ciertas condiciones de bloque en envío y recepción.

Fiabilidad

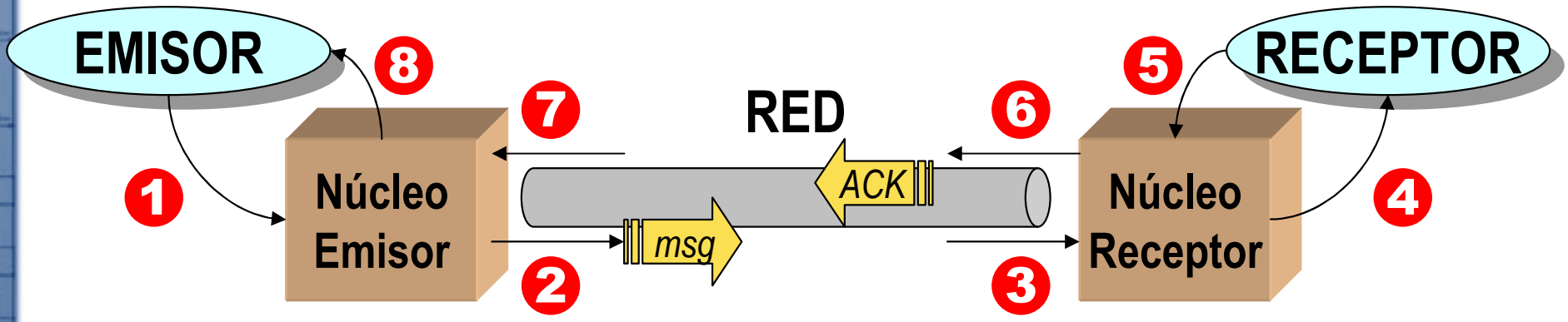
El envío fiable de datos garantiza que un mensaje enviado ha sido recibido por el receptor.

Implica la retransmisión de mensajes de validación (ACKs).

La fiabilidad la puede garantizar:

- El protocolo de comunicación.
  - TCP si y UDP no.
- Los elementos emisor y receptor.

Primitivas de Comunicación



Envío no bloqueante: [1:8] El emisor continua al pasar el mensaje al núcleo.

Envío bloqueante: [1:2:7:8] El emisor espera a que el núcleo transmita por red el mensaje.

Envío bloqueante fiable: [1:2:3:6:7:8]: El emisor espera a que el núcleo receptor recoge el mensaje.

Envío bloqueante explícito: [1:2:3:4:5:6:7:8]: Idem al anterior, pero es la aplicación receptora la que confirma la recepción.

Petición-Respuesta: [1:2:3:4:<servicio>:5:6:7:8]: El emisor espera a que el receptor procese la operación para reanudar la ejecución.

Direccionamiento

Información válida para la identificación de elementos del sistema. Posibles receptores de un mensaje.

Mecanismos:

- Dirección dependiente de la localización:
  - Por ejemplo: dirección máquina + dirección puerto local.
  - No proporciona transparencia.
- Dirección independiente de la localización (dir. lógica):
  - Facilita transparencia.
  - Necesidad de proceso de localización:
    - Mediante *broadcast*.
    - Uso de un servidor de localización que mantiene relaciones entre direcciones lógicas y físicas.
  - Uso de cache en clientes para evitar localización.

Comunicación en Grupo

Se habilita por medio de:

- Variantes de protocolos de red: *IP-multicast*.
- Emulándola por medio de protocolos de alto nivel o por las aplicaciones.

El direccionamiento se realiza por medio de una **dirección de grupo** (grupo al que pertenecen todos los receptores).

Modelos de grupos:

- Grupo abierto.
- Grupo abierto controlado.
- Grupo cerrado.

Comunicación en Grupo

Utilidad para los sistemas distribuidos:

- Ofrecer tolerancia a fallos basado en servicios replicados.
- Localizar objetos en sistemas distribuidos.
- Mejor rendimiento mediante datos replicados.
- Actualizaciones múltiples.
- Operaciones colectivas en cálculo paralelo.

Problemática:

- Comunicación fiable es difícil.
- Escalabilidad de las tecnologías (Internet con MBone).
- Gestión de grupos.
- Ruteo (*Flooding*, *Spanning Tree*, *RPB*, *TRPB*, *RPM*).

Ordenación en Comunicación en Grupo

De acuerdo a las garantías de ofrecidas en la recepción de mensajes de grupo se tienen:

- **Ordenación FIFO**: Los mensajes de una fuente llegan a cada receptor en el orden que son enviados.
- **Ordenación Causal**: Los mensajes enviados por dos emisores distintos so recibidos en el orden relativo en el que se han enviado.
- **Ordenación Total**: Todos los mensajes (de varias fuentes) enviados a un grupo son recibidos en el mismo orden por todos los elementos.

Cliente-Servidor

<Pasaje de Mensajes>

- **Berkeley Sockets**
- **Java Sockets**

Pasaje de Mensajes

Características deseables de un buen sistema de pasaje de mensajes

Simplicidad

- Simple y fácil de usar (uso directo).
- Hacer sin preocuparse de aspectos de la red/sistema.

Semántica uniforme en:

- Comunicaciones locales.
- Comunicaciones remotas.

Pasaje de Mensajes

Eficiencia

Si no la hay, las IPC son costosas.

Criterio: reducción del número de mensajes intercambiados.

Optimización incluye:

-Evitar el costo de establecer y terminar conexiones entre el mismo par de procesos y cada intercambio de mensajes entre ellos.

-Minimizar el costo de mantener la conexión.

-Optimizar los reconocimientos cuando hay una serie de mensajes entre el ***send*** y ***receive***.

Pasaje de Mensajes

Confiabilidad

La caída del nodo o enlace implica pérdida de mensaje.

Se usan *timeouts* (duplicación de mensajes).

Correctitud

Pueden enviarse *multicast*

- Atomicidad.
- Orden de despacho.
- Persistencia.

Pasaje de Mensajes

Flexibilidad

Deben permitir alguna clase de control de flujo entre procesos cooperativos, incluyendo *send/receive* sincrónicos y asincrónicos.

Seguridad

- Autenticación del receptor de un mensaje por el *emisor (sender).*
- Autenticación del emisor de un mensaje por el *receptor.*
- Encriptación del mensaje.

Pasaje de Mensajes

Portabilidad

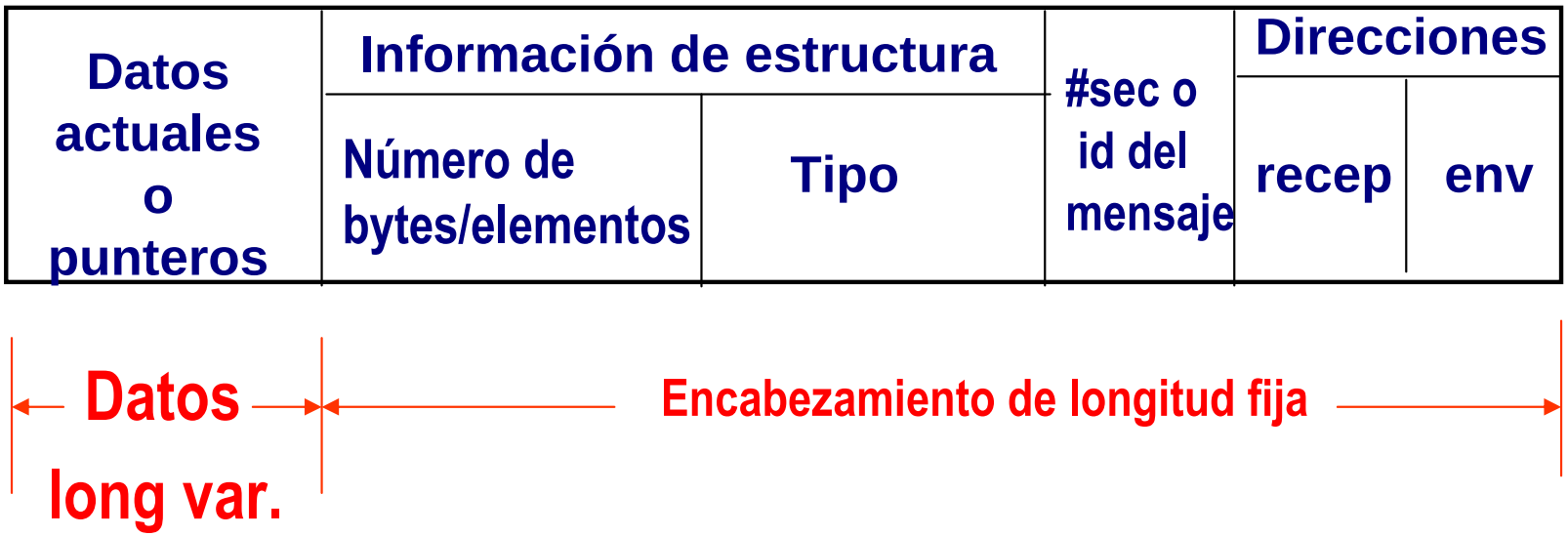
El sistema de pasaje de mensajes debe ser *portable* (posible construcción de protocolos de IPC reusando el mismo sistema de mensajes).

Heterogeneidad de máquinas ⇒ compatibilización de representación.

Pasaje de Mensajes

El pasaje de mensajes en la intercomunicación entre procesos

Una estructura de mensajes típica:



Pasaje de Mensajes

El *emisor* determina el contenido del mensaje.

El *receptor* tiene en cuenta como interpretar los datos.

Pasaje de Mensajes

En el diseño de un protocolo de intercomunicación entre procesos debe considerarse:

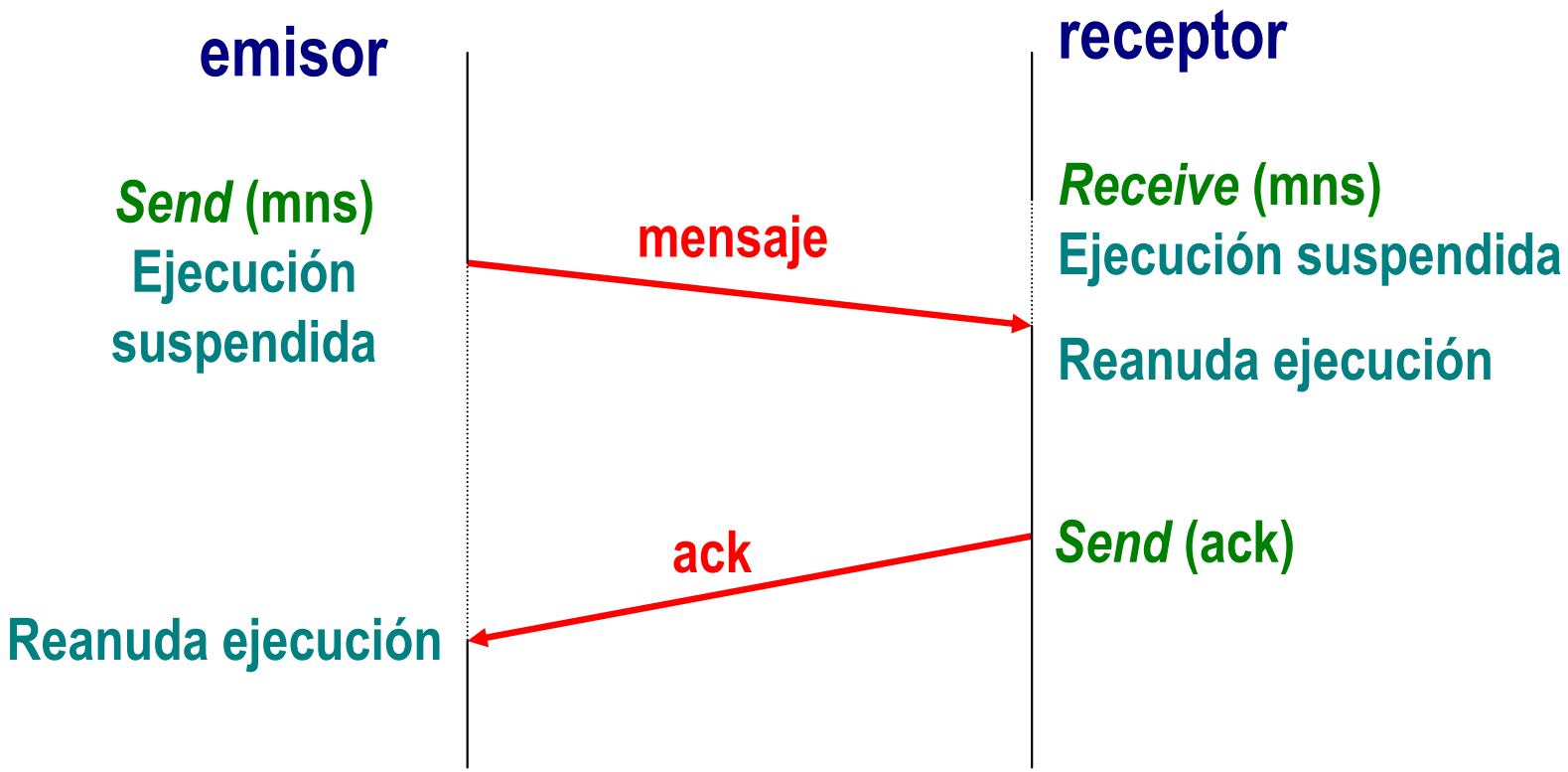
- ¿Quién envía?
- ¿Quién recibe?
- ¿Hay uno o varios *receptores*?
- ¿Está garantizado que el mensaje ha sido aceptado por el *receptor*?
- ¿Necesita el *send* esperar una respuesta?

Pasaje de Mensajes

- ¿Qué se debe hacer si falla el sitio y/o enlace?
- ¿Qué sucede si el *receptor* no está listo para recibir el mensaje?
- Si hay varios mensajes esperando en el *receptor*, ¿puede éste cambiar el orden?

Pasaje de Mensajes

Mensajes Bloqueantes vs. no bloqueantes



Pasaje de Mensajes

No bloqueante

El receptor conoce la llegada del mensaje

- Polling.
- Interrupción.

Bloqueantes $\Rightarrow$ *sincrónica*

Fácil de implementar pero poca concurrencia.

Pasaje de Mensajes

Buffering

De *buffer nulo* a *buffer* con capacidad ilimitada.

Sin *buffer*

- Cita (*rendez-vous*).
- Descarte.

***Buffer* simple**

Adecuado para transferencia sincrónica.

Capacidad infinita

Almacena todo lo que recibe (asincrónica).

Pasaje de Mensajes

Buffer límite finito

Puede haber *overflow* de *buffer*

- Comunicación no exitosa (lo hace menos confiable).
- Comunicación con flujo controlado (bloquea al emisor hasta que haya espacio).

Buffer múltiple

Mailbox o pórtico.

Pasaje de Mensajes

Mensajes multidatagrama

La mayoría tiene un límite superior en el tamaño del dato que puede ser transmitido en algún momento (MTU).

Esto implica que magnitudes mas grandes deben fragmentarse en paquetes.

El ensamblador y desensamblador es responsabilidad del sistema de pasaje de mensajes.

Pasaje de Mensajes

Codificación y decodificación de mensajes de datos

Un puntero absoluto pierde significado cuando es transmitido de un espacio a otro.

Diferentes programas objeto ocupan una cantidad de espacio variada.

Se usan, en general, dos representaciones:

- Representación etiquetada (MACH).
- Representación no etiquetada (SUN XDR).

Pasaje de Mensajes

Direccionamiento de los procesos (1)

Problema de nombres de las partes involucradas en una interacción.

Direccionamiento explícito

Send (process-id,msg)

Receive (process-id,msg)

Pasaje de Mensajes

Direccionamiento de los procesos (2)

Direccionamiento implícito

No se explicita el nombre del proceso.

Resulta útil para cliente-servidor: se menciona un servicio.

Send-any (service-id,msg)

Receive-any (proceso-mudo,msg)

Pasaje de Mensajes

Manejo de fallas

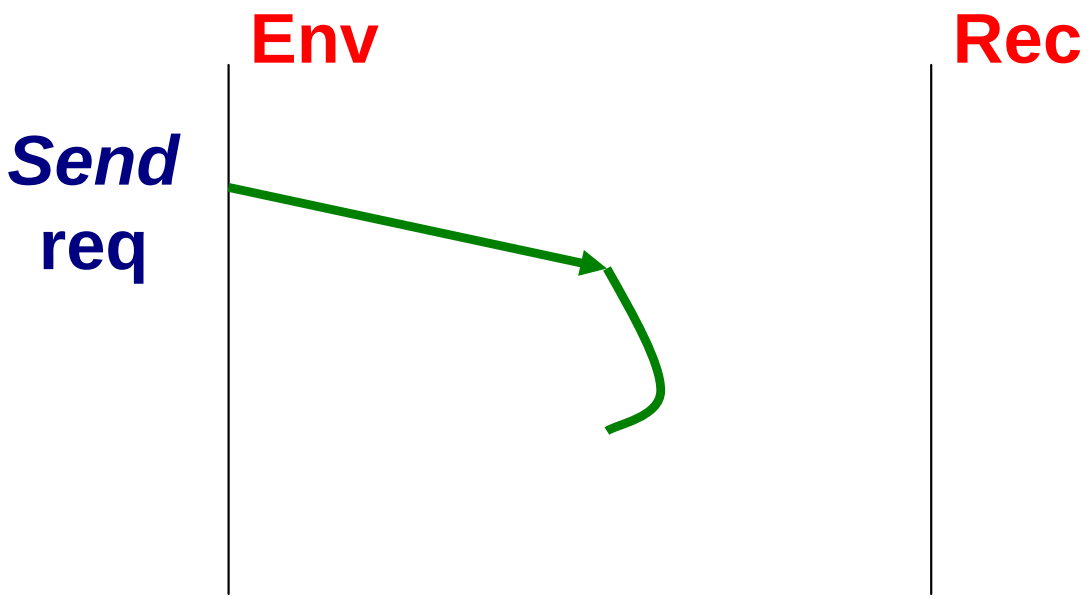
- Caída de sitio.
- Caída de enlace.

Problemas posibles:

a) Pérdida del mensaje de requerimiento.
b) Pérdida del mensaje de respuesta.
c) Ejecución del requerimiento no exitosa.

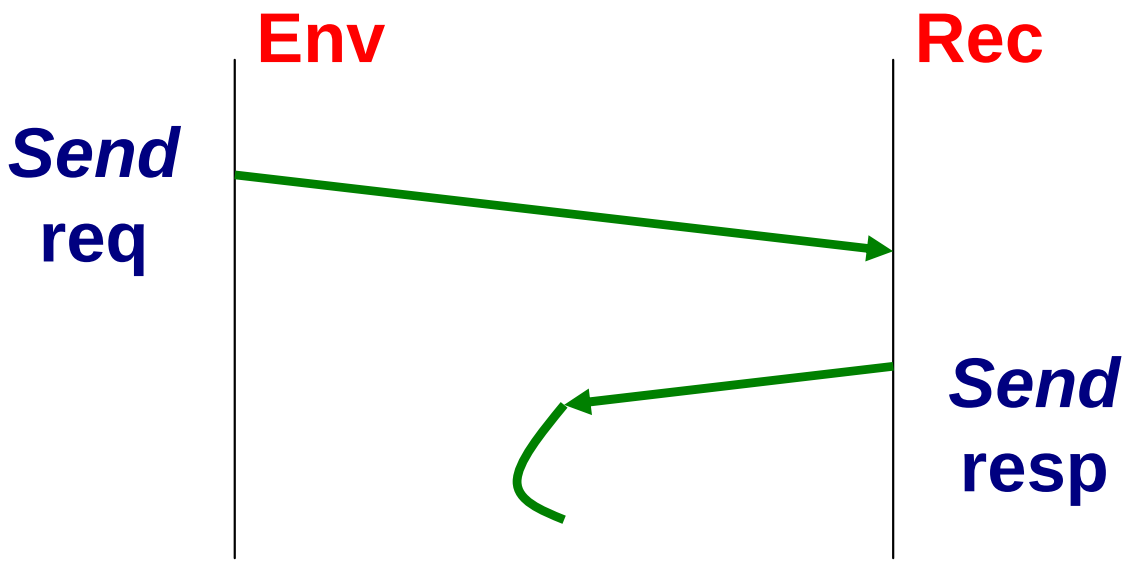
Pasaje de Mensajes

(a) Pérdida del mensaje de requerimiento



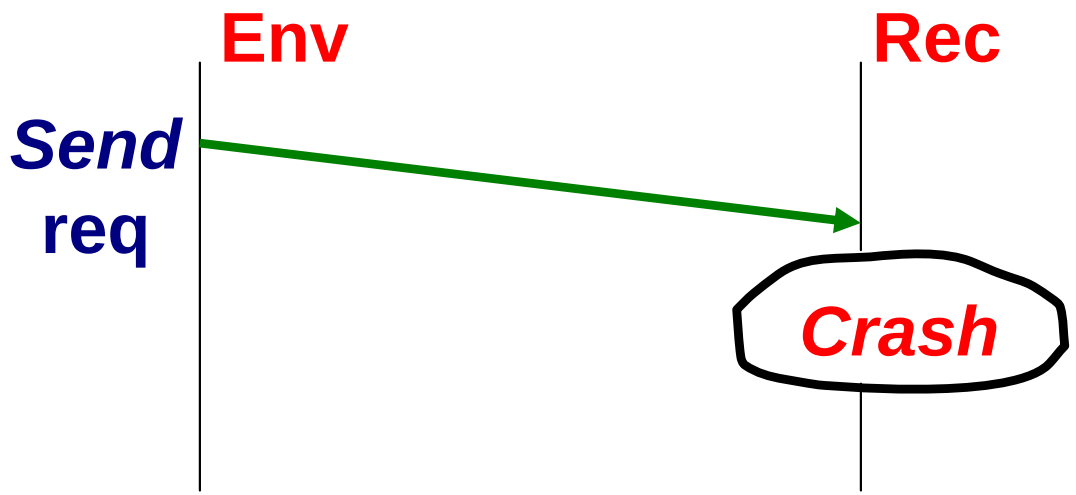
Pasaje de Mensajes

(b) Pérdida del mensaje de respuesta



Pasaje de Mensajes

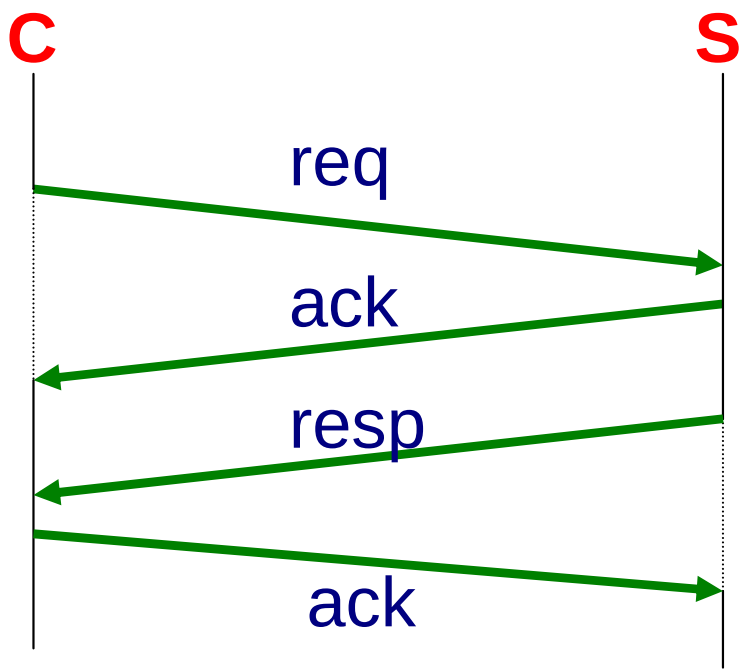
(c) Ejecución del requerimiento no exitosa



Pasaje de Mensajes

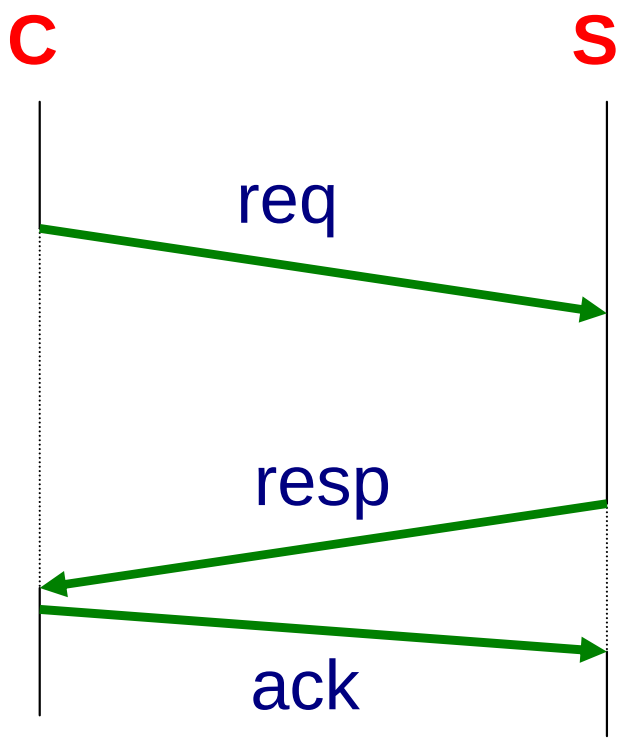
Protocolos de mensajes confiables

Cuatro mensajes



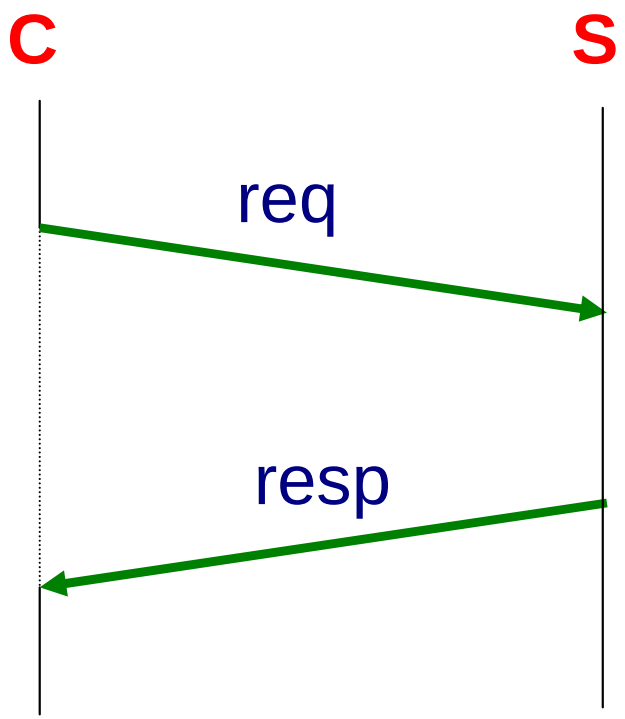
Pasaje de Mensajes

Tres mensajes



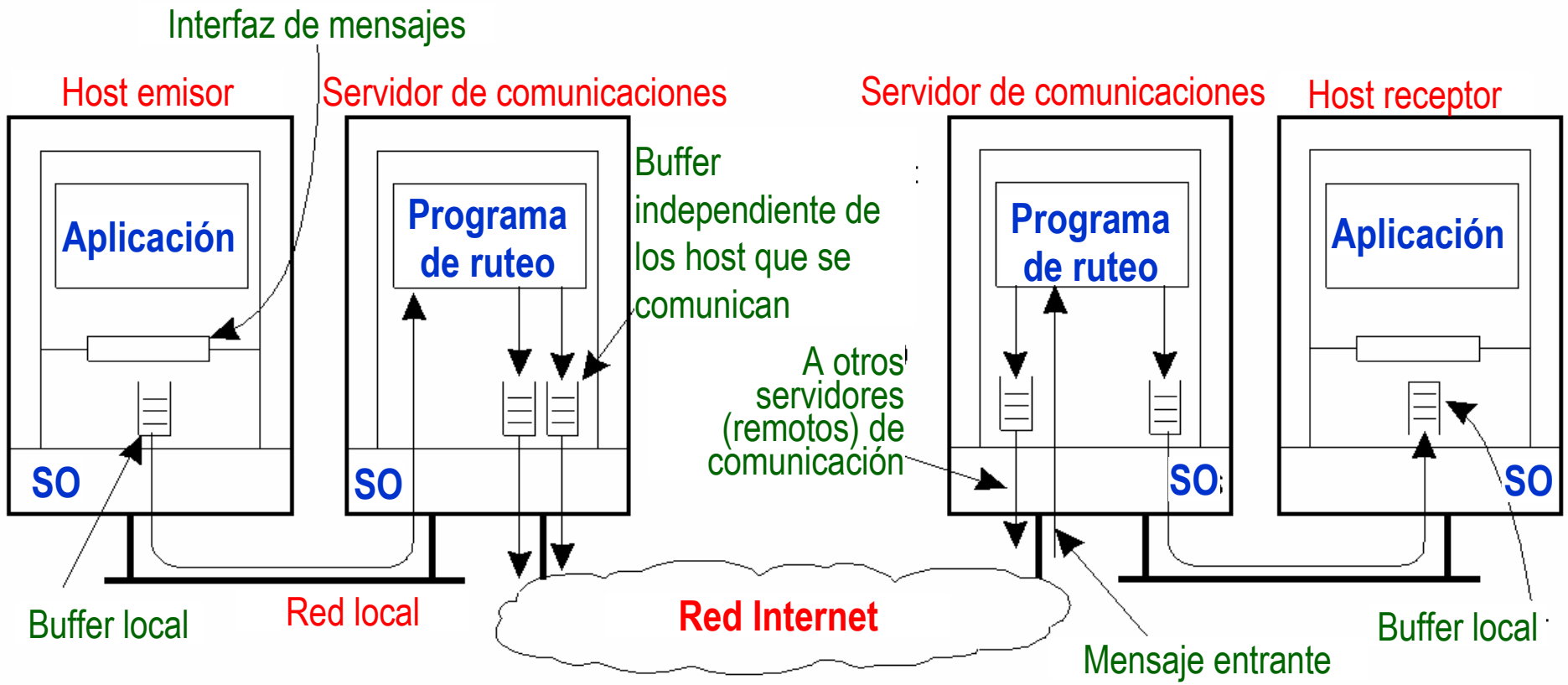
Pasaje de Mensajes

Dos mensajes



Pasaje de Mensajes

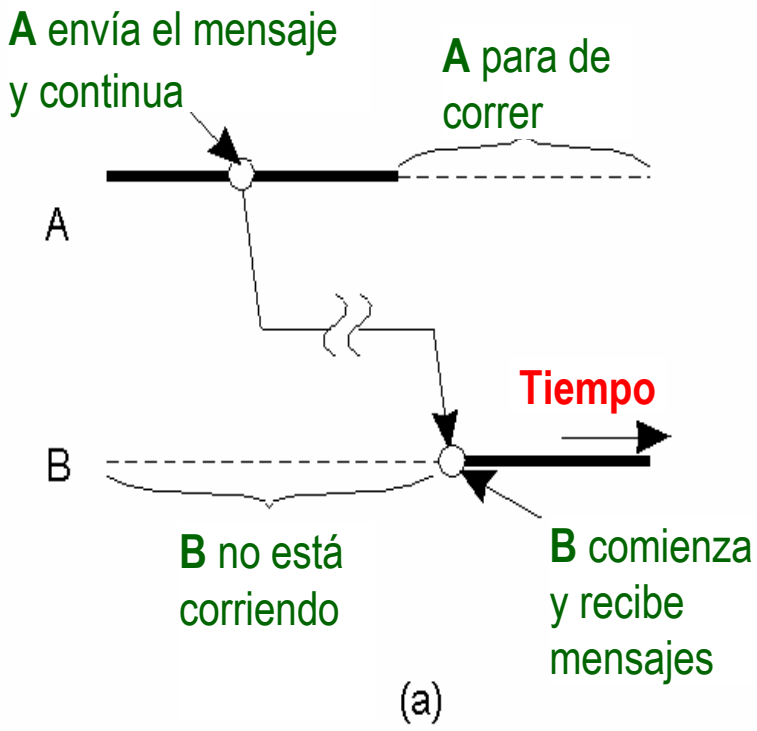
Persistencia y Sincronismo en Comunicaciones

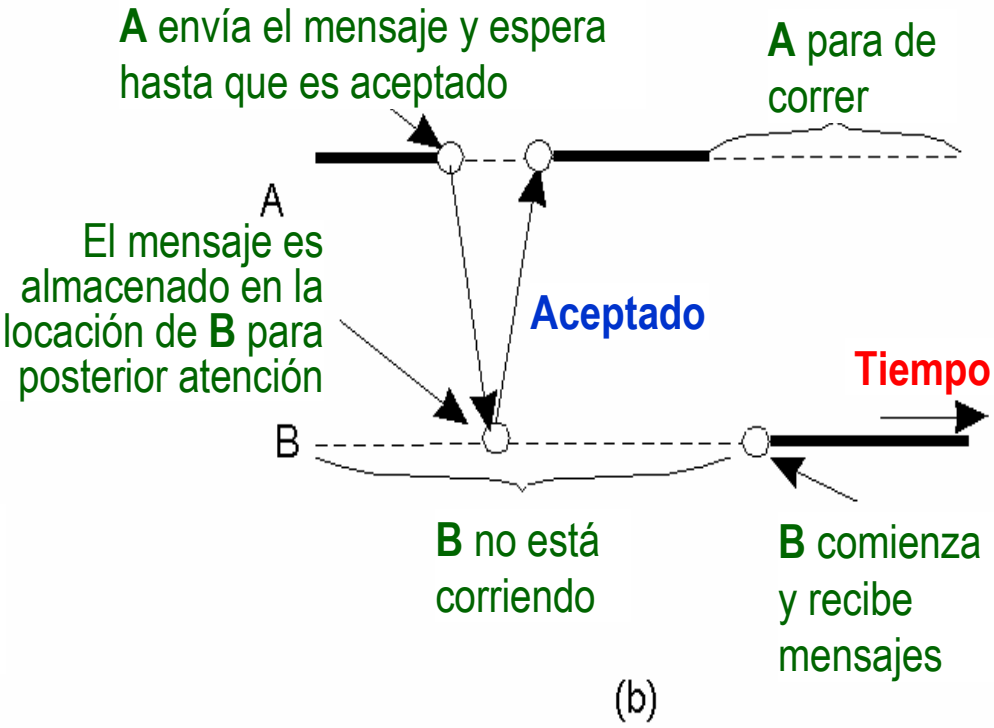


Organización general de un sistema de comunicaciones en el cual los hosts están conectados por una red

Pasaje de Mensajes

Persistencia y Sincronismo en Comunicaciones



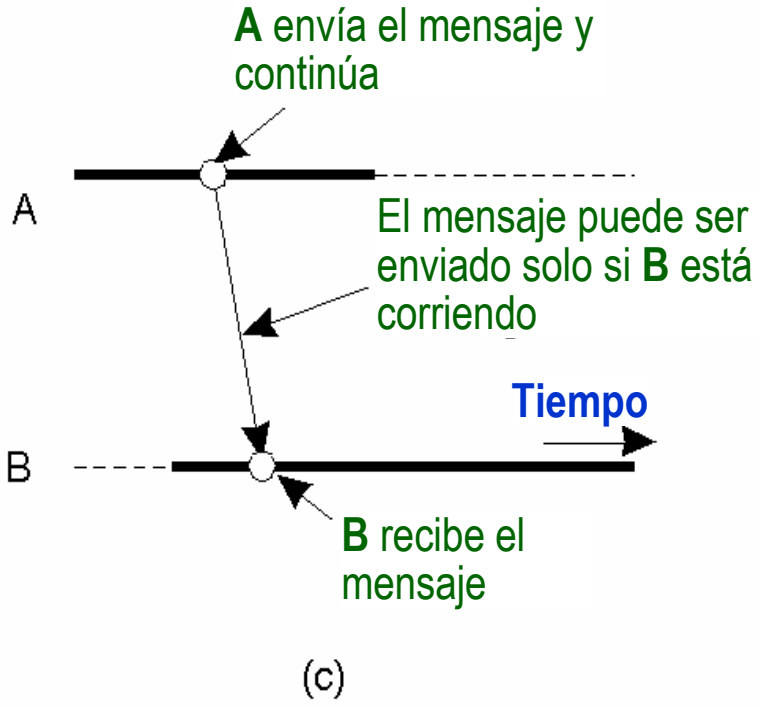


a) Persistencia en comunicación asincrónica.

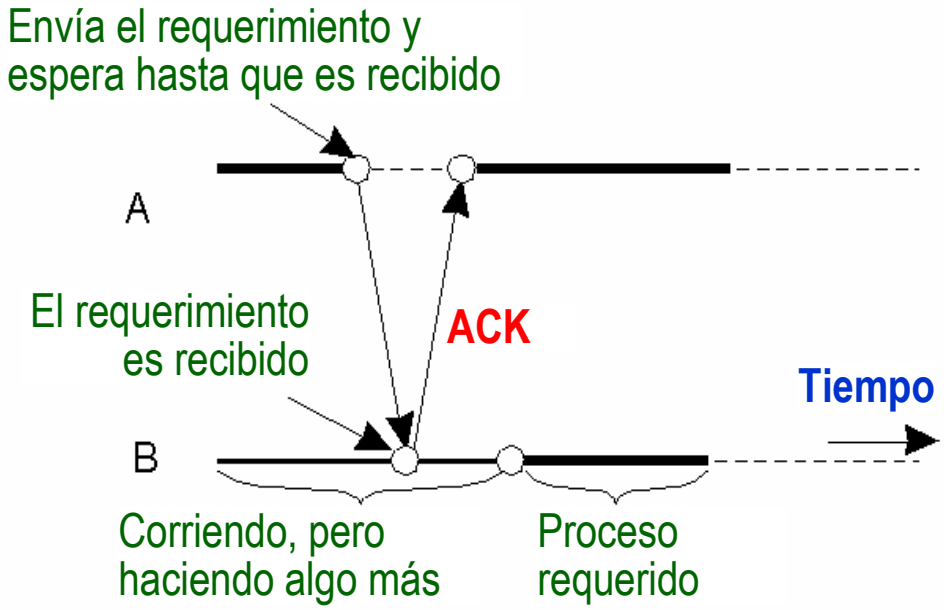
b) Persistencia en comunicación sincrónica.

Pasaje de Mensajes

Persistencia y Sincronismo en Comunicaciones



(c)



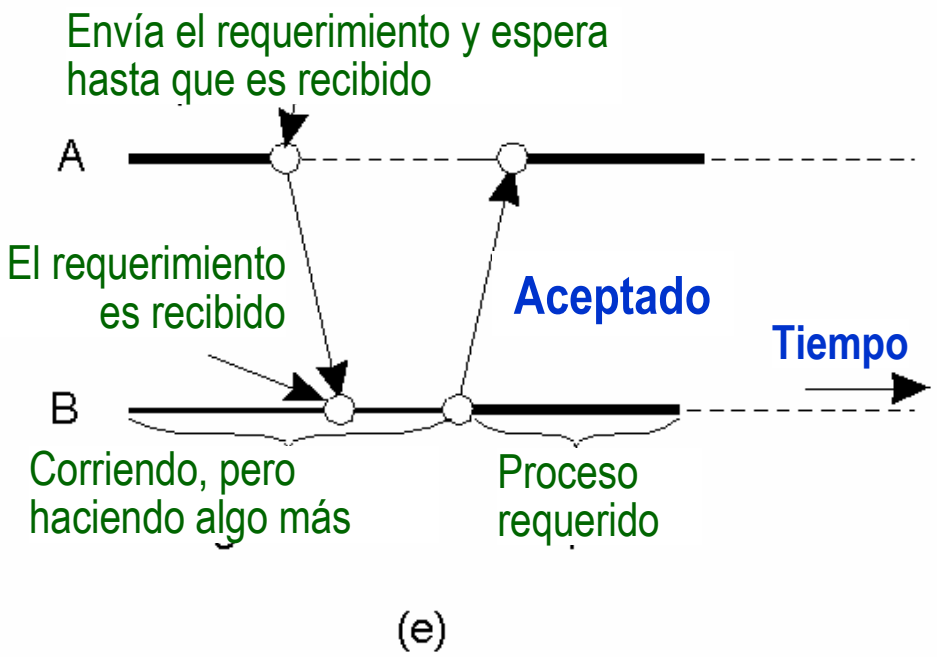
(d)

c) Comunicación asincrónica transitoria.

d) Comunicación sincrónica transitoria basada en recepción.

Pasaje de Mensajes

Persistencia y Sincronismo en Comunicaciones

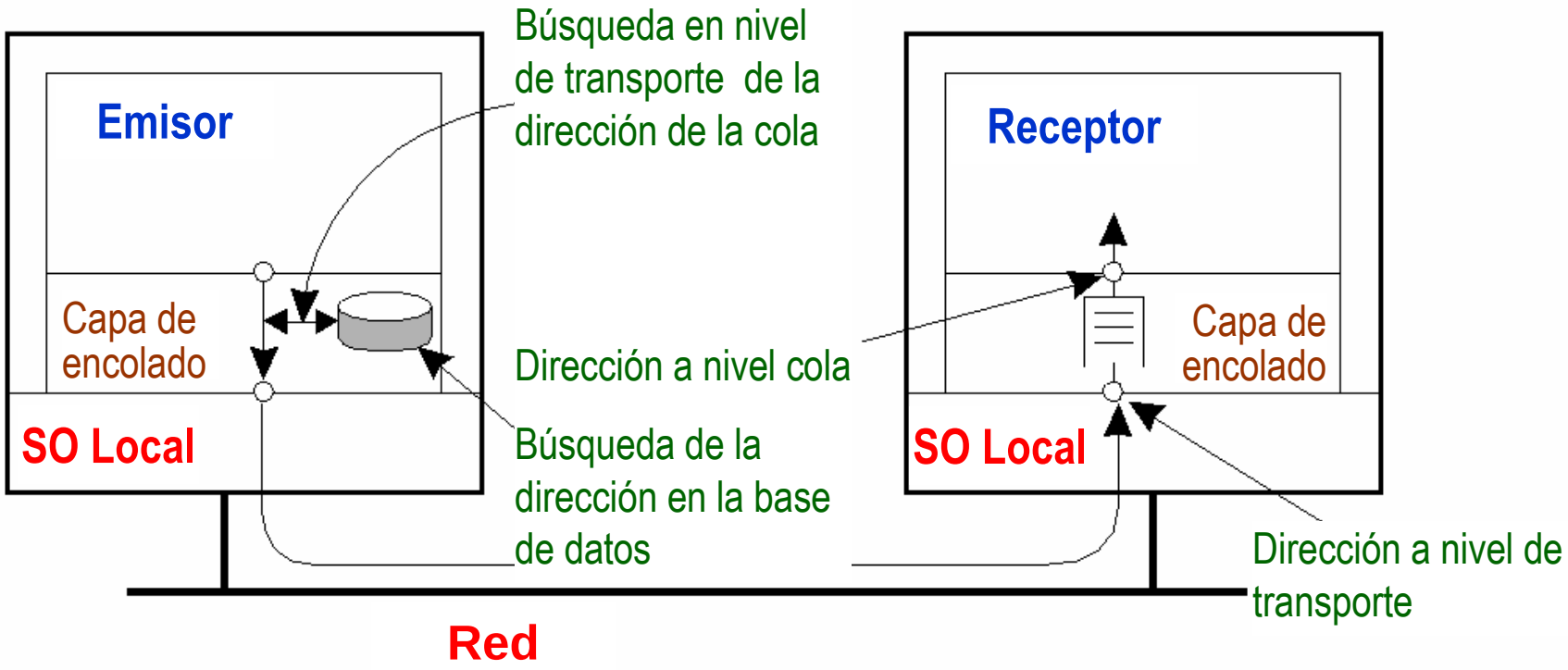


e) Comunicación sincrónica transitoria basado en el despacho de mensajes.

f) Comunicación sincrónica transitoria basado en respuesta.

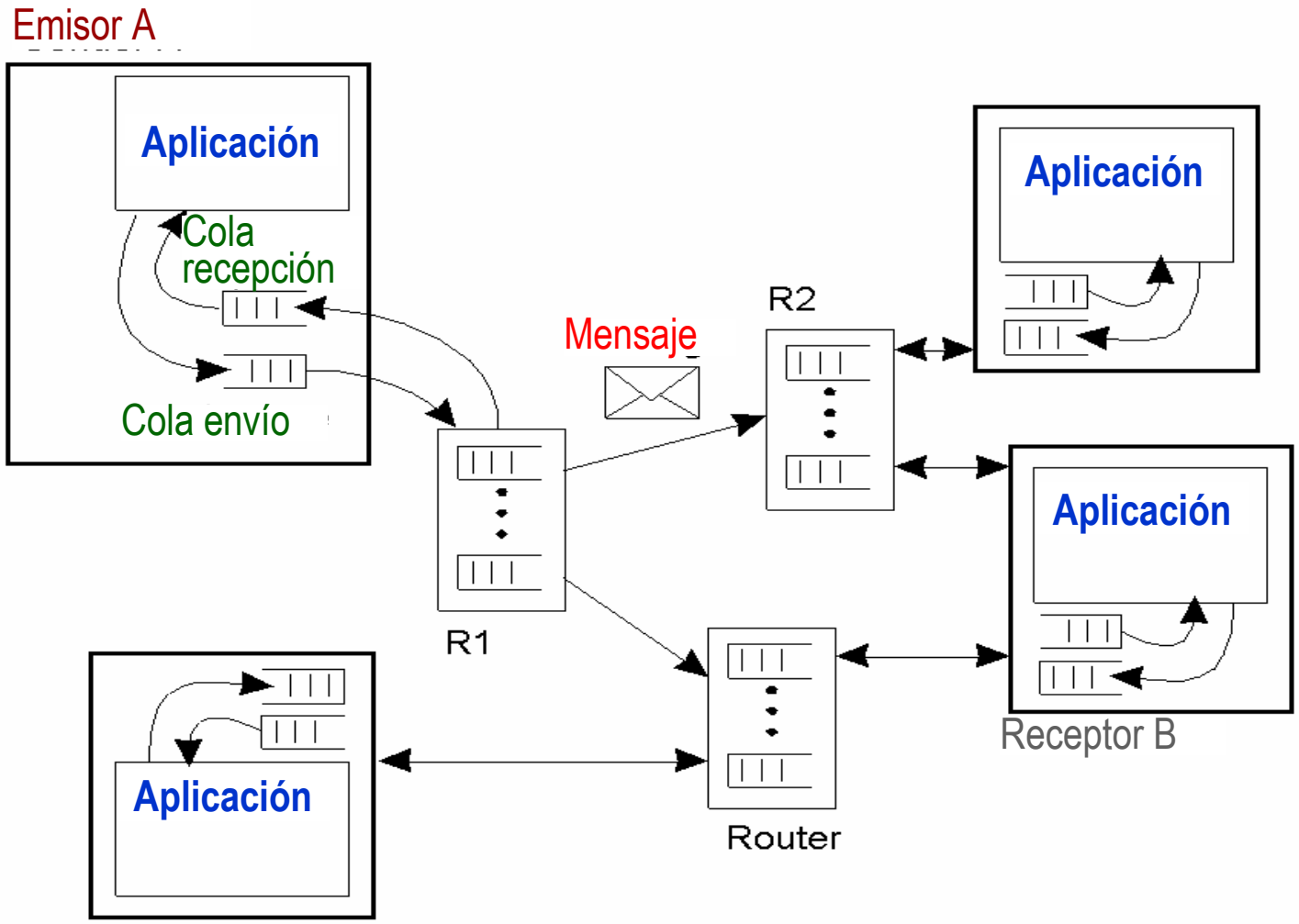
Pasaje de Mensajes

Arquitectura general de un sistema de mensajes encolados



Relación entre direcciones a nivel de colas y direcciones a nivel de red.

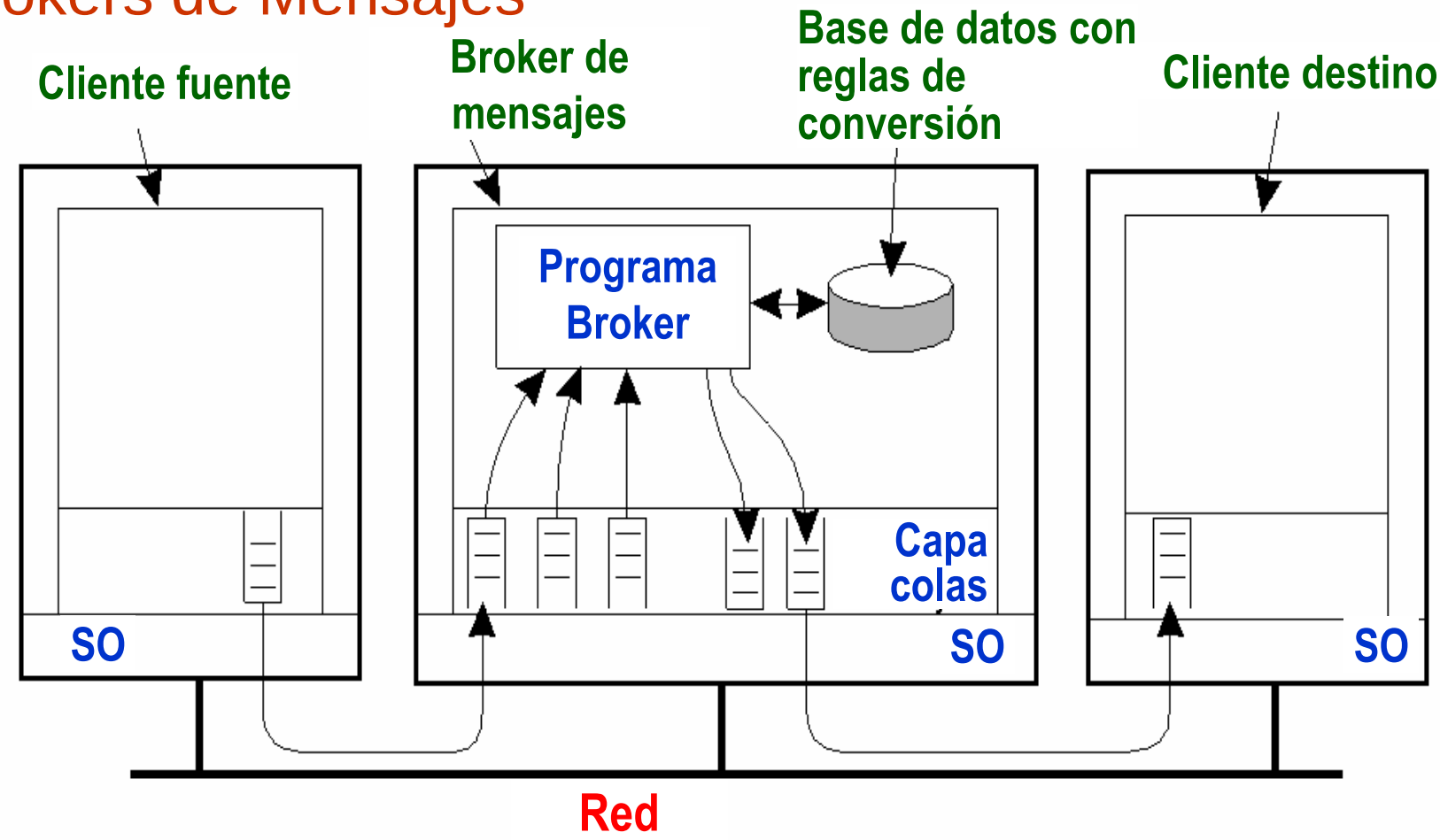
Pasaje de Mensajes



Organización general de un sistema de cola de mensajes con routers.

Pasaje de Mensajes

Brokers de Mensajes



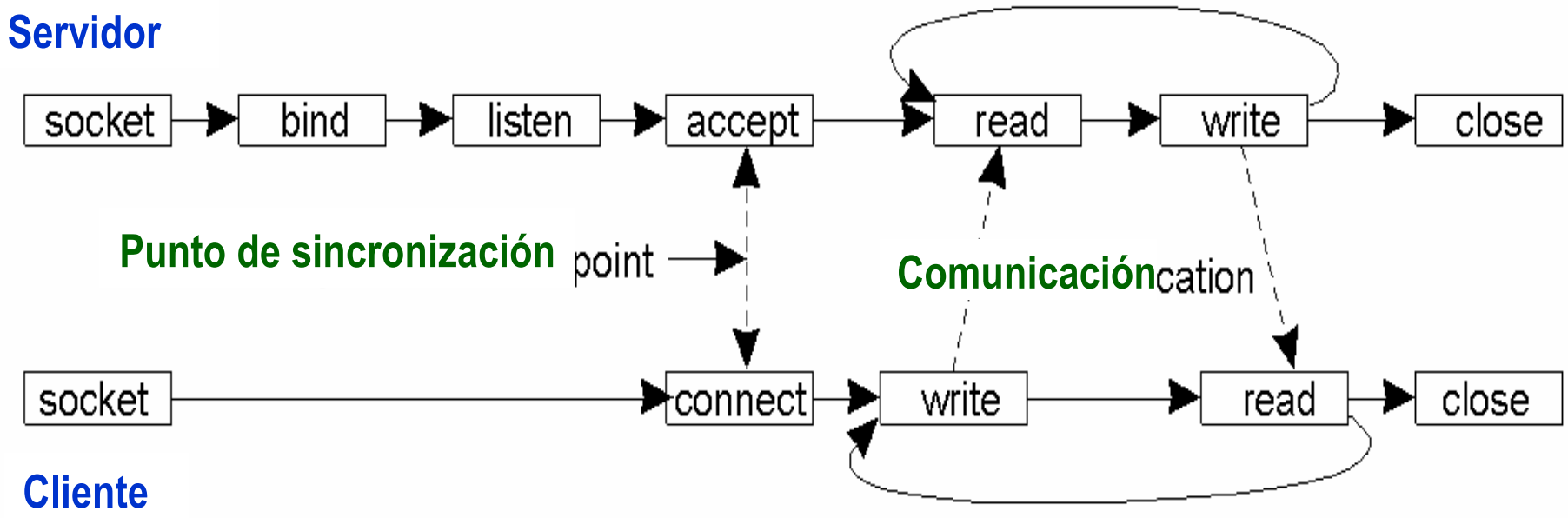
Organización general de un broker de mensajes en un sistema de mensajes encolados.

Pasaje de Mensajes

Primitivas para Sockets en TCP/IP

Primitiva	Significado
Socket	Crea un nuevo punto final de comunicación
Bind	Adjunta una dirección local a un socket
Listen	Anuncia el deseo de aceptar conexiones
Accept	Se bloquea el llamador hasta que llegue un requerimiento de conexión
Connect	Intenta activamente establecer una conexión
Send	Envía datos sobre la conexión
Receive	Recibe algunos datos sobre la conexión
Close	Libera la conexión

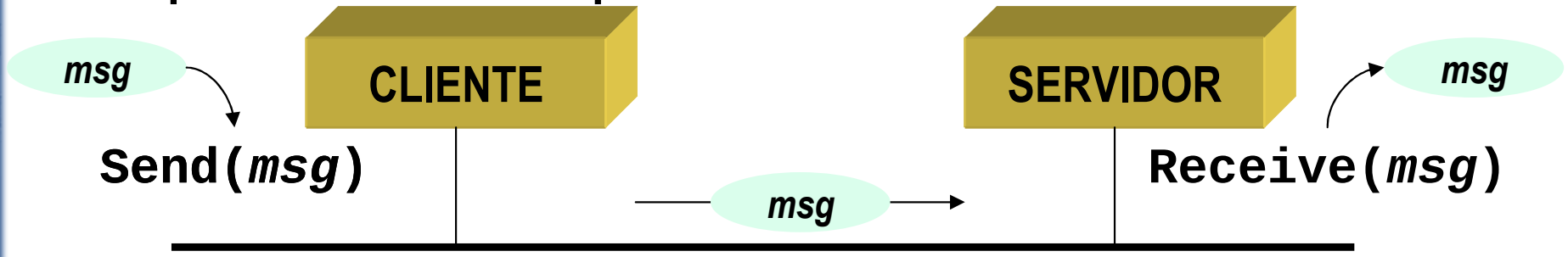
Pasaje de Mensajes



Modelo de comunicación orientado a conexión usando sockets.

Pasaje de Mensajes

Los modelos de comunicación basados en cliente-servidor con paso de mensajes responden al esqueleto:




```
Mensaje msg,reply;
msg=<dato a trasmitir>
send(msg);
receive(reply);
if(isOK(reply))
  <operación correcto>
else
  <error en operación>
...
```

```
Mensaje op,ack;
receive(op);
if(validOp(op))
  ack=<operación OK>
else
  ack=<operación ERROR>
send(ack);
...
```

Paso de Mensajes

Cada pareja **`send-receive`** transmite un mensaje entre cliente y servidor. Por lo general de forma asíncrona.

Habitualmente:

- **`Send`** no bloqueante.
- **`Receive`** bloqueante (pude hacerse no bloqueante).

Los mensajes intercambiados pueden ser:

- Mensajes de texto (por ejemplo: HTTP).
- Mensajes con formato (binarios).

Las aplicaciones definen el protocolo de comunicación: Petición-respuesta, recepción explícita, sin/con confirmación, ...

Mensajes Texto

Estructura del Mensaje:

- Cadenas de caracteres.
- Por ejemplo HTTP:

```
"GET //lisidi.cs.uns.edu.ar HTTP/1.1"
```

Envío del Mensaje:

```
send("GET //lisidi.cs.uns.edu.ar HTTP/1.1");
```

El emisor debe hacer un análisis de la cadena de caracteres transmitida.

Mensajes Binarios

Estructura del Mensaje:

```
struct mensaje_st
{
   unsigned int msg_tipo;
   unsigned int msg_seq_id;
   unsigned char msg_data[1024];
};
```

Envío del Mensaje:

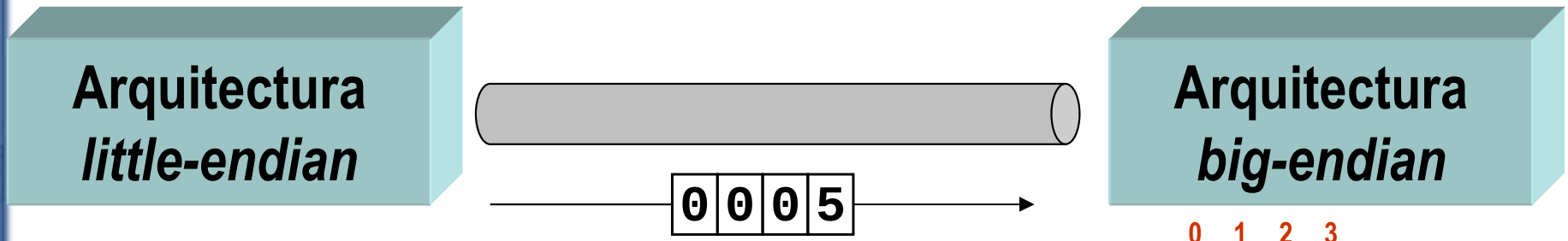
```
   struct mensaje_st confirm;
   confirm.msg_tipo=MSG_ACK;
   confirm.msg_seq_id=129;

   send(confirm);
```

Formatos de Representación

Para la transmisión de formatos binarios tanto emisor y receptor deben coincidir en la interpretación de los *bytes* transmitidos.

Problemática:

- Tamaño de los datos numéricos.
- Ordenación de *bytes*.
- Formatos de texto: ASCII vs Unicode.



Berkeley Sockets

Aparecieron en 1981 en UNIX BSD 4.2

- Intento de incluir TCP/IP en UNIX.
- Diseño independiente del protocolo de comunicación.

Un socket es punto final de comunicación (dirección IP y puerto).

Abstracción que:

- Ofrece interfaz de acceso a los servicios de red en el nivel de transporte.
- Representa un extremo de una comunicación bidireccional con una dirección asociada.

Berkeley Sockets

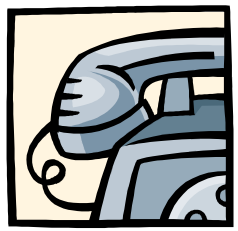
Sujetos a proceso de estandarización dentro de POSIX (POSIX 1003.1g).

Actualmente:

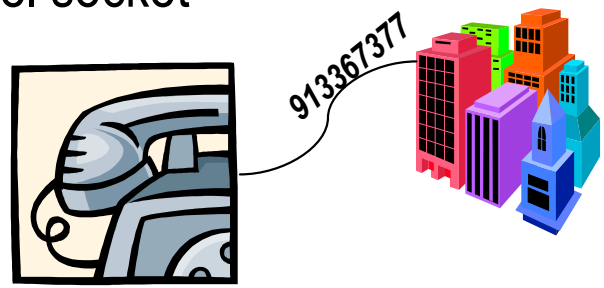
- Disponibles en casi todos los sistemas UNIX.
- En prácticamente todos los sistemas operativos:
  - WinSock: API de sockets de M$ Windows.
- En Java como clase nativa.

Conceptos Básicos sobre *Sockets*

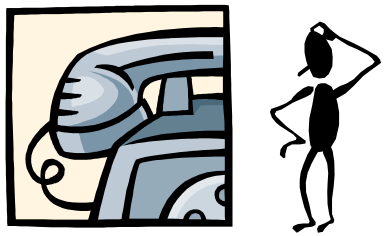
- Dominios de comunicación.
- Tipos de *sockets*.
- Direcciones de *sockets*.
- Creación de un *socket*.
- Asignación de direcciones.
- Solicitud de conexión.
- Preparar para aceptar conexiones.
- Aceptar una conexión.
- Transferencia de datos.



1.- Creación del socket



2.- Asignación de dirección



3.- Aceptación de conexión

Dominios de Comunicación

- Un dominio representa una familia de protocolos.
- Un *socket* está asociado a un dominio desde su creación.
- Sólo se pueden comunicar *sockets* del mismo dominio.
- Los servicios de *sockets* son independientes del dominio.

Algunos ejemplos:

- **`PF_UNIX`** (o **`PF_LOCAL`**): comunicación dentro de una máquina.
- **`PF_INET`**: comunicación usando protocolos TCP/IP.

Tipos de Sockets

- **Stream** (**`SOCK_STREAM`**):
  - Orientado a conexión.
  - Fiable, se asegura el orden de entrega de mensajes.
  - No mantiene separación entre mensajes.
  - Si **`PF_INET`** se corresponde con el protocolo TCP.
- **Datagrama** (**`SOCK_DGRAM`**):
  - Sin conexión.
  - No fiable, no se asegura el orden en la entrega.
  - Mantiene la separación entre mensajes.
  - Si **`PF_INET`** se corresponde con el protocolo UDP.
- **Raw** (**`SOCK_RAW`**):
  - Permite el acceso a los protocolos internos como IP.

Direcciones de Sockets

- Cada *socket* debe tener asignada una dirección única.
- Dependientes del dominio.
- Las direcciones se usan para:
  - Asignar una dirección local a un socket (**`bind`**).
  - Especificar una dirección remota (**`connect`** o **`sendto`**).
- Se utiliza la estructura genérica de dirección:
  - **`struct sockaddr`** ***`mi_dir;`***
- Cada dominio usa una estructura específica.
  - Uso de *cast* en las llamadas.
  - Direcciones en **`PF_INET`** (**`struct sockaddr_in`**).
  - Direcciones en **`PF_UNIX`** (**`struct sockaddr_un`**).

Direcciones de Sockets en PF_INET

Una dirección destino viene determinada por:

- Dirección del *host*: 32 bits.
- Puerto de servicio: 16 bits.

Estructura **`struct sockaddr_in`**:

- Debe iniciarse a 0 (**`bzero`**).
- **`sin_family`**: dominio (AF_INET).
- **`sin_port`**: puerto.
- **`sin_addr`**: dirección del *host*.

Una transmisión está caracterizada por cinco parámetros únicos:

- Dirección *host* y puerto origen.
- Dirección *host* y puerto destino.
- Protocolo de transporte (UDP o TCP).

Obtención de la Dirección del Host

Los usuarios manejan direcciones en forma de texto:

- decimal-punto: 200.49.226.22
- dominio-punto: lisidi.cs.uns.edu.ar

- Conversión a binario desde decimal-punto:

  `int inet_aton(char *str,struct in_addr *dir)`

  - **`str`**: contiene la cadena a convertir.
  - **`dir`**: resultado de la conversión en formato de red.
- Conversión a binario desde dominio-punto:

  `struct hostent *gethostbyname(char *str)`

  - **`str`**: cadena a convertir.
  - Devuelve la estructura que describe al *host*.

Creación de un Socket

La función **`socket`** crea uno nuevo:

`int socket(int dom,int tipo,int proto)`

- Devuelve un descriptor de archivo (igual que un **`open`** de archivo).
- Dominio (**`dom`**): **`PF`**_*XXX*
- Tipo de socket (**`tipo`**): **`SOCK`**_*XXX*
- Protocolo (**`proto`**): Dependiente del dominio y del tipo:
  - 0 elige el más adecuado.
  - Especificados en **`/etc/protocols`**.

El socket creado **no** tiene dirección asignada.

Asignación de Direcciones

La asignación de una dirección a un *socket* ya creado:

```
int bind(int s,struct sockaddr*
   dir,int tam)
```

- Socket (**s**): Ya debe estar creado.
- Dirección a asignar (**dir**): Estructura dependiendo del dominio.
- Tamaño de la dirección (**tam**): **sizeof()**.

Si no se asigna dirección (típico en clientes) se le asigna automáticamente (puerto efímero) en su primera utilización (**connect** o **sendto**).

Asignación de Direcciones (PF_INET)

Direcciones en dominio **`PF_INET`**

- Puertos en rango 0..65535.
- Reservados: 0..1023.
- Si se le indica el 0, el sistema elige uno.
- Host: una dirección IP de la máquina local.
  - **`INADDR_ANY`**: elige cualquiera de la máquina.

Si el puerto solicitado está ya asignado la función **`bind`** devuelve un valor negativo.

El espacio de puertos para *streams* (TCP) y datagramas (UDP) es independiente.

Solicitud de Conexión

Realizada en el cliente por medio de la función:

```
int connect(int s,struct sockaddr*
    d,int tam)
```

- Socket creado (**s**).
- Dirección del servidor (**d**).
- Tamaño de la dirección (**tam**).

Si el cliente no ha asignado dirección al socket, se le asigna una automáticamente.

Normalmente se usa con *streams*.

Preparar para Aceptar Conexiones

Realizada en el servidor *stream* después de haber creado (**`socket`**) y reservado dirección (**`bind`**) para el socket:

```
int listen(int sd, int baklog)
```

- Socket (**`sd`**): Descriptor de uso del socket.
- Tamaño del buffer (**`backlog`**): Número máximo de peticiones pendientes de aceptar que se encolarán (algunos manuales recomiendan 5)

Hace que el socket quede preparado para aceptar conexiones.

Aceptar una Conexión

Realizada en el servidor *stream* después de haber preparado la conexión (**`listen`**):

```
int accept(int s,struct sockaddr
  *d,int *tam)
```

- Socket (**`sd`**): Descriptor de uso del socket.
- Dirección del cliente (**`d`**): Dirección del socket del cliente devuelta.
- Tamaño de la dirección (**`tam`**): Parámetro valor-resultado
  - Antes de la llamada: tamaño de dir.
  - Después de la llamada: tamaño de la dirección del cliente que se devuelve.

Aceptar una Conexión

La semántica de la función **`accept`** es la siguiente:

- Cuando se produce la conexión, el servidor obtiene:
  - La dirección del socket del cliente.
  - Un nuevo descriptor (socket) que queda conectado al socket del cliente.
- Después de la conexión quedan activos dos sockets en el servidor:
  - El original para aceptar nuevas conexiones
  - El nuevo para enviar/recibir datos por la conexión establecida.
- Idealmente se pueden plantear servidores *multithread* para servicio concurrente.

Otras Funcionalidades

Obtener la dirección a partir de un descriptor:

- Dirección local: **`getsockname()`**.
- Dirección del socket en el otro extremo: **`getpeername()`**.

Transformación de valores:

- De formato *host* a red:
  - Enteros largos: htonl().
  - Enteros cortos: htons().
- De formato de red a *host*:
  - Enteros largos: ntohl().
  - Enteros cortos: ntohs().

Cerrar la conexión:

- Para cerrar ambos tipos de sockets: **`close()`**.
  - Si el socket es de tipo stream cierra la conexión en ambos sentidos.
- Para cerrar un único extremo: **`shutdown()`**.

Transferencia de Datos con *Streams*

Envío:

Puede usarse la llamada **`write`** sobre el descriptor de socket.

```
int send(int s,char *mem,int tam,int flags)
```

- Devuelve el número de bytes enviados.

Recepción:

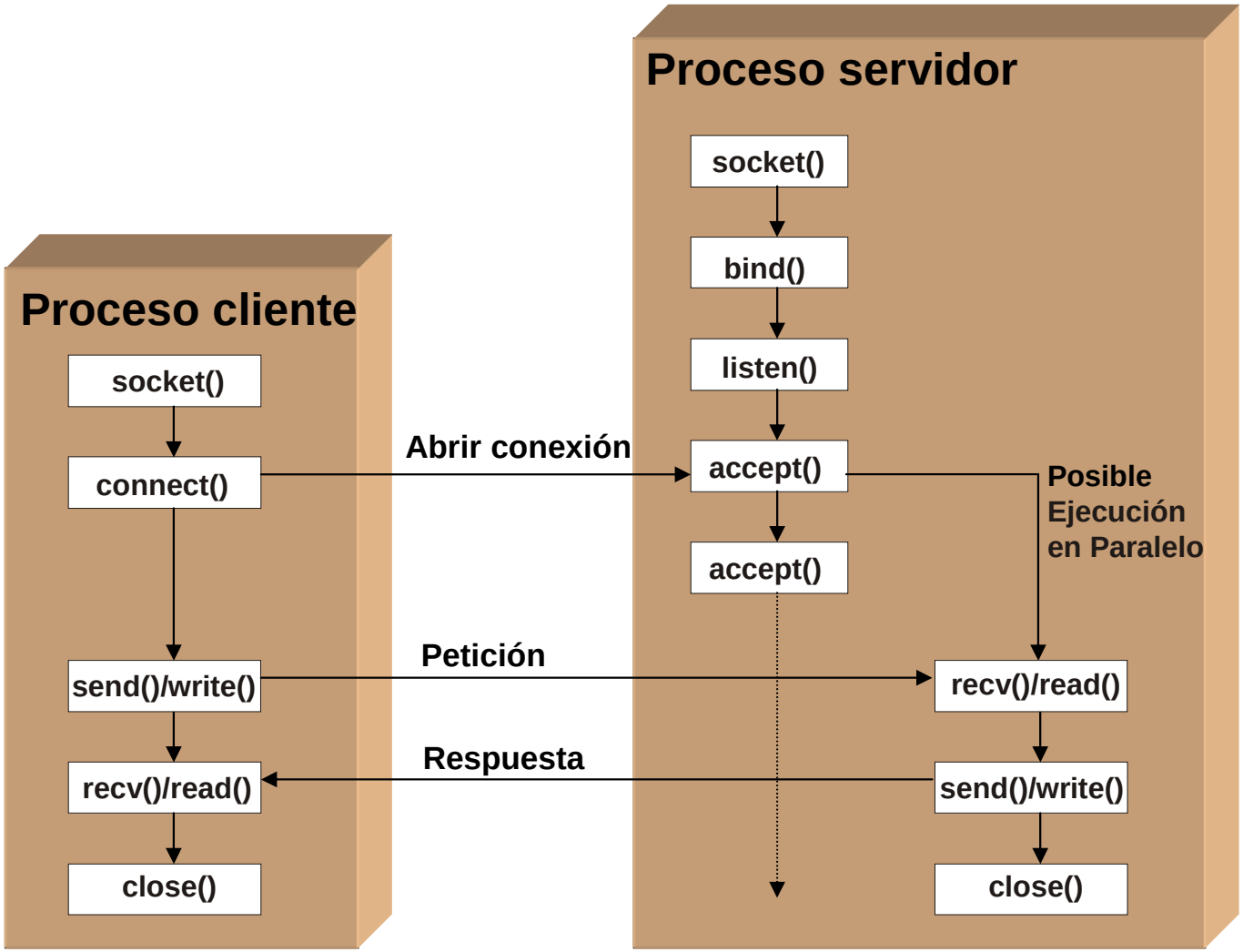
Puede usarse la llamada **`read`** sobre el descriptor de socket.

```
int recv(int s,char *mem,int tam,int flags)
```

- Devuelve el número de bytes recibidos.

Los flags implican aspectos avanzados, como enviar o recibir datos urgentes (*out-of-band*).

Escenario de Uso de Sockets *streams*



Transferencia de Datos con Datagramas

Envío:

```
int sendto(int s,char *mem,int tam,
           int flags,struct sockaddr
           *dir, int *tam)
```

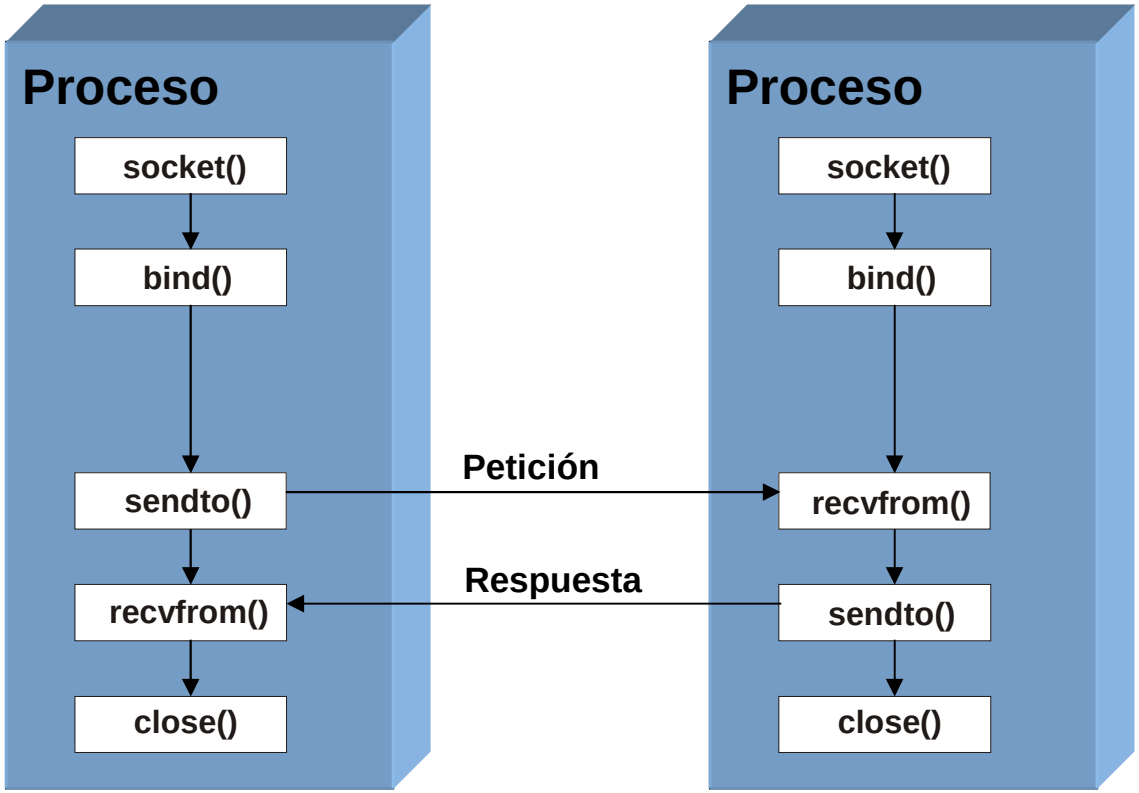
Recepción:

```
int recvfrom(int s,char *mem,int
           tam,int flags,struct
           sockaddr *dir, int *tam)
```

No se establece una conexión (**`connect`**/**`accept`**) previa.

Para usar un socket para transferir basta con crear el socket y reservar la dirección (**`bind`**).

Escenario de Uso de Sockets Datagrama



Configuración de Opciones

Existen varios niveles dependiendo del protocolo afectado como parámetro:

- **SOL_SOCKET**: opciones independientes del protocolo.
- **IPPROTO_TCP**: nivel de protocolo TCP.
- **IPPTOTO_IP**: nivel de protocolo IP.

Consultar opciones asociadas a un socket: **getsockopt()**

Modificar opciones asociadas a un socket: **setsockopt()**

Ejemplos (nivel **SOL_SOCKET**):

- **SO_REUSEADDR**: permite reutilizar direcciones.

Sockets en Java

- Engloba en objetos cada una de las estructuras de la comunicación. Las funciones se tratan como métodos de dichos objetos:
  - **`InetAddress`**
  - **`Socket DatagramSocket ServerSocket`**
  - **`Connection`**
  - **`DatagramPacket`**

- Define un nivel de abstracción mayor, proporcionando constructores que realizan parte del proceso de inicialización de los elementos.

Sockets en Java (Direccionamiento)

Las direcciones de Internet se asocian a objetos de la clase **`InetAddress`**. Estos objetos se construyen en base a métodos estáticos de la clase:

- **`static InetAddress getByName(String host)`**

  Obtiene una dirección IP en base al nombre (dominios o números).

- **`static InetAddress getLocalHost()`**

  Obtiene la dirección IP local.

Sockets en Java (UDP)

La información a trasmitir se asocia a un objetos de la clase **`DatagramPacket`**. Estos objetos se construyen con un *array* de *bytes* a transmitir:

- **`DatagramPacket(byte[] datos, int tam)`**

  Crea un *datagrama* para el vector de *bytes* a transmitir.

Adicionalmente se le puede pasar una dirección IP (**`InetAddress`**) y un puerto para indicar el destino de transmisión del paquete cuando se envíe.

Sockets en Java (UDP)

La comunicación vía UDP se realiza por medio de objetos de la clase **`DatagramSocket`**.

- **`DatagramSocket(int puerto,InetAddress dir)`**

  Crea un socket UDP con un bind a la dirección y puerto indicados. Dirección y puerto son opcionales (se elige uno libre).
- **`void send(DatagramPacket paquete)`**

  Envía el datagrama a la dirección del paquete.
- **`void receive(DatagramPacket paquete)`**

  Se bloquea hasta la recepción del datagrama.

Otros métodos:

- **`void close()`**: Cierra el socket.
- **`void setSoTimeout(int timeout)`**: Define el tiempo de bloqueo en un **`receive()`**.

Sockets en Java (TCP)

Se utilizan dos clases de socket (una para el cliente y otra para socket servidor).

Para el cliente:

- **`Socket(InetAddress dir, int puerto)`**

  Crea un *socket stream* para el **cliente** conectado con la dirección y puerto indicados. Existen otros constructores con diferentes argumentos.

Para el servidor:

- **`ServerSocket(int puerto)`**

  Crea un *socket stream* para el **servidor**. Existen otros constructores con diferentes argumentos.

- **`Socket accept()`**

  Prepara la conexión y se bloquea a espera de conexiones. Equivale a **`listen`** y **`accept`** de BSD Sockets. Devuelve un **`Socket`**.

Sockets en Java (TCP)

La lectura y la escritura sobre sockets TCP se realiza por medio de objetos derivados de las clases de Stream (en concreto subclases de **`InputStream`** y **`OutputStream`**).

- **`InputStream getInputStream()`**

  Obtiene el *stream* de lectura.

- **`OutputStream getOutputStream()`**

  Obtiene el *stream* de escritura.

Los objetos devueltos son transformados a la subclase apropiada para manejarlo (por ejemplo **`DataInputStream`**).

Sockets en Java

Para la confección de diferentes protocolos o variantes de los mismos Java proporciona un interfaz denominado **`SocketImplFactory`**.

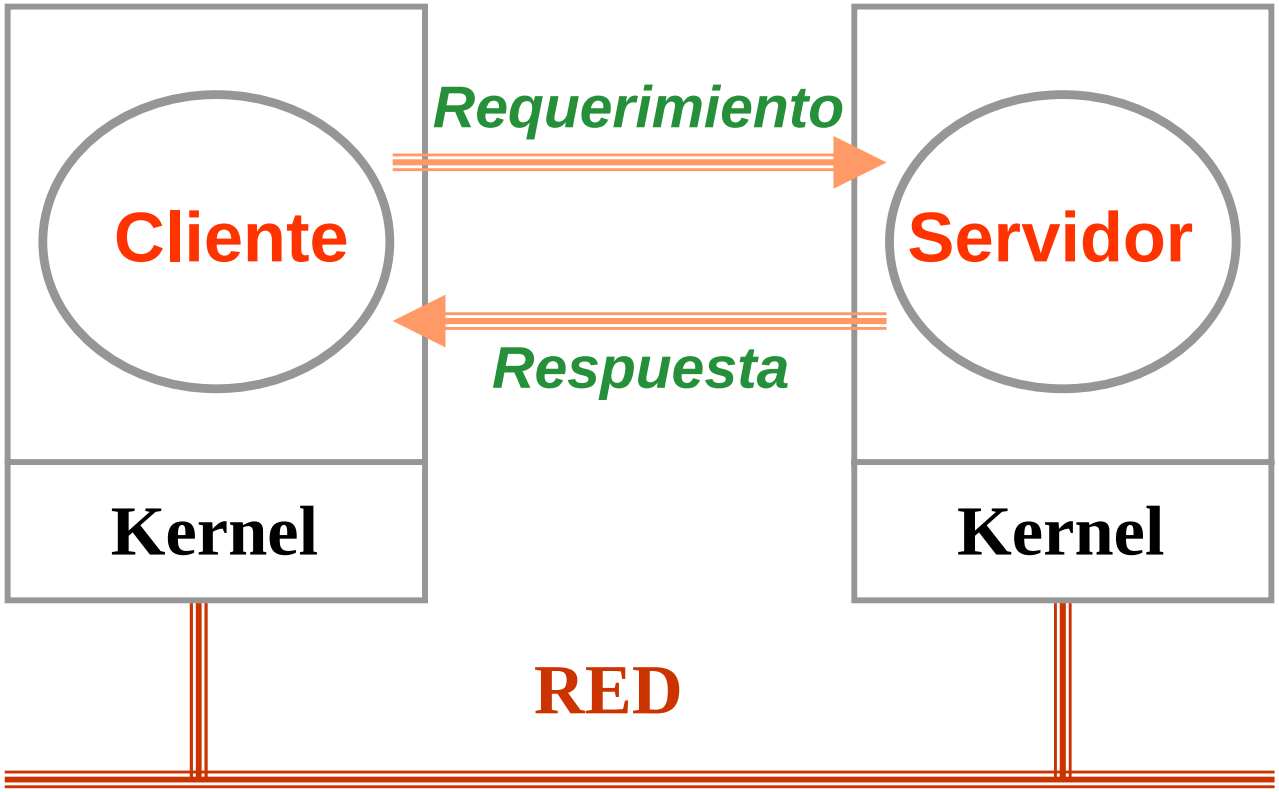
- **`SocketImpl createSocketImpl()`**

  Crea una instancia de la superclase de sockets.

`SocketImpl` es la superclase de todos las implementaciones de sockets. Un objeto de esta clase es usado como argumento para la creación de sockets.

Comunicación en Sistemas Distribuidos

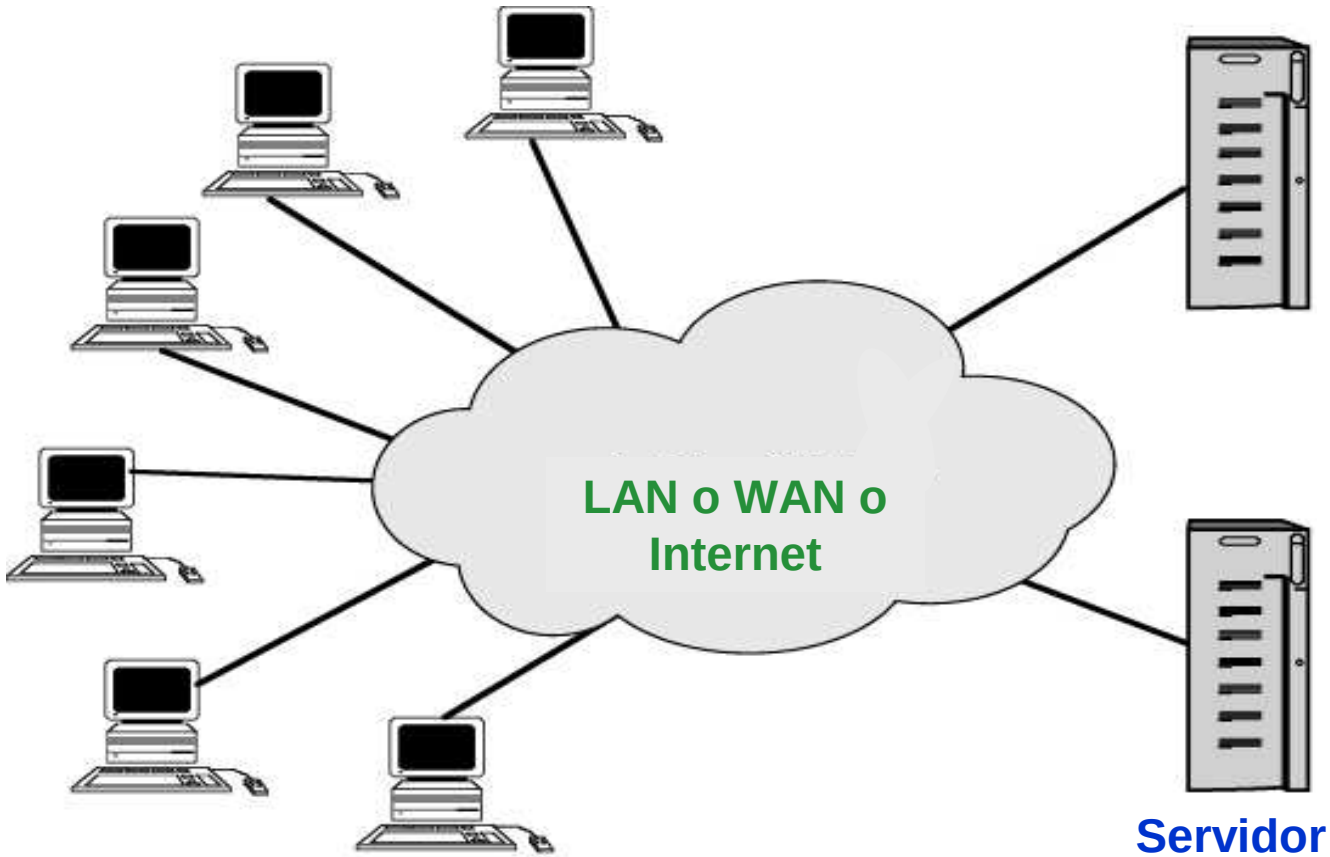
Modelo Cliente - Servidor



Modelo Cliente-Servidor

- Las máquinas *cliente* son, en general, PCs monousuario o puestos de trabajo que ofrecen una interfaz muy amigable al usuario final.
- Cada *servidor* ofrece una serie de servicios de usuario compartidos a los clientes.
- El servidor permite a los clientes compartir el acceso a la misma base de datos y permite el uso de un sistema de computación de alto rendimiento para gestionar la base de datos.

Modelo Cliente-Servidor

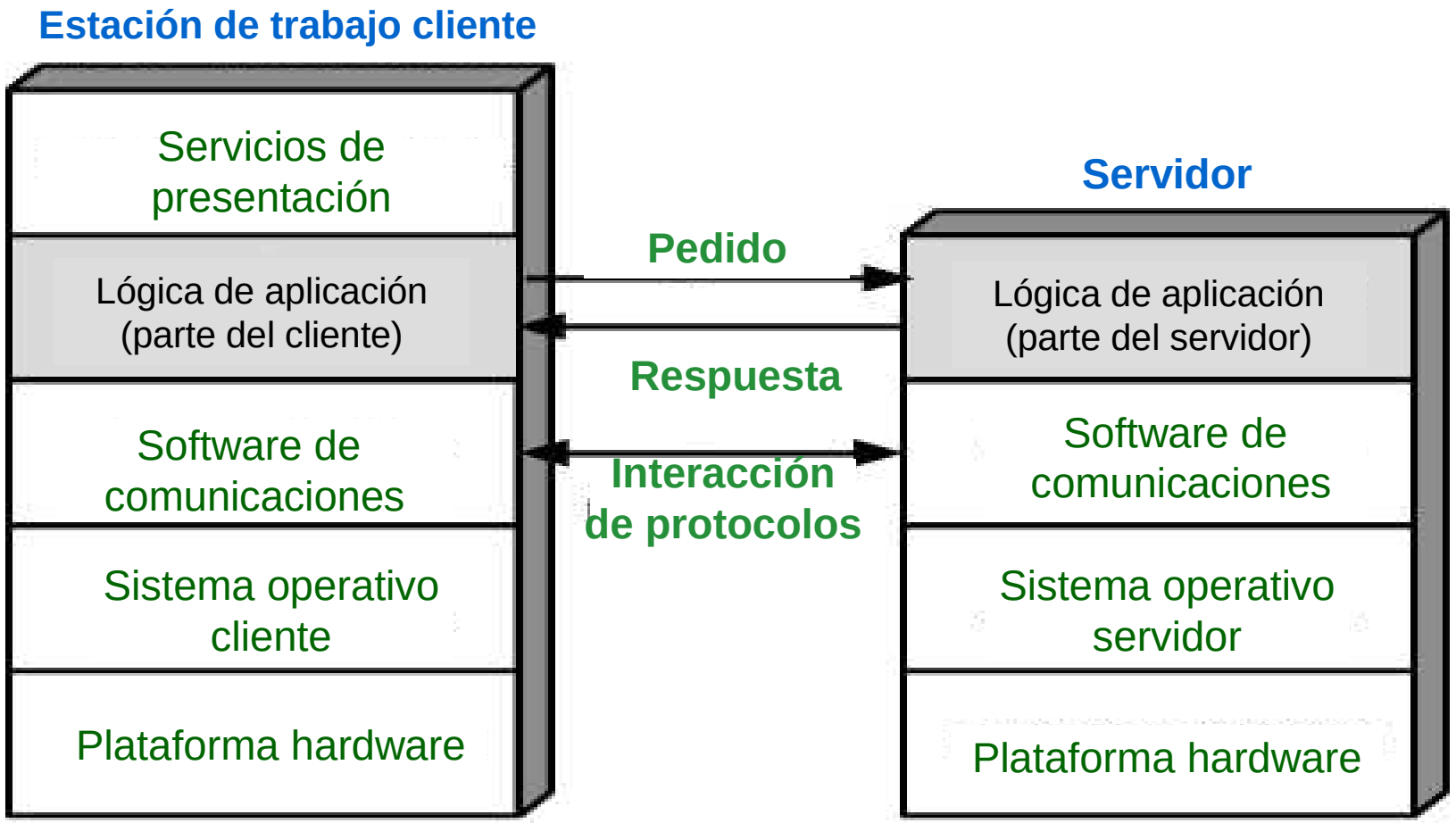


Entorno genérico cliente/servidor.

Modelo Cliente-Servidor

- El software básico es un sistema operativo que se ejecuta en la plataforma del hardware.
- Las plataformas y los sistemas operativos del cliente y del servidor pueden ser *diferentes*.
- Estas diferencias de niveles inferiores no son relevantes en tanto que un cliente y un servidor compartan los mismos protocolos de comunicación y soporten las mismas aplicaciones.

Modelo Cliente-Servidor



Arquitectura genérica Cliente-Servidor

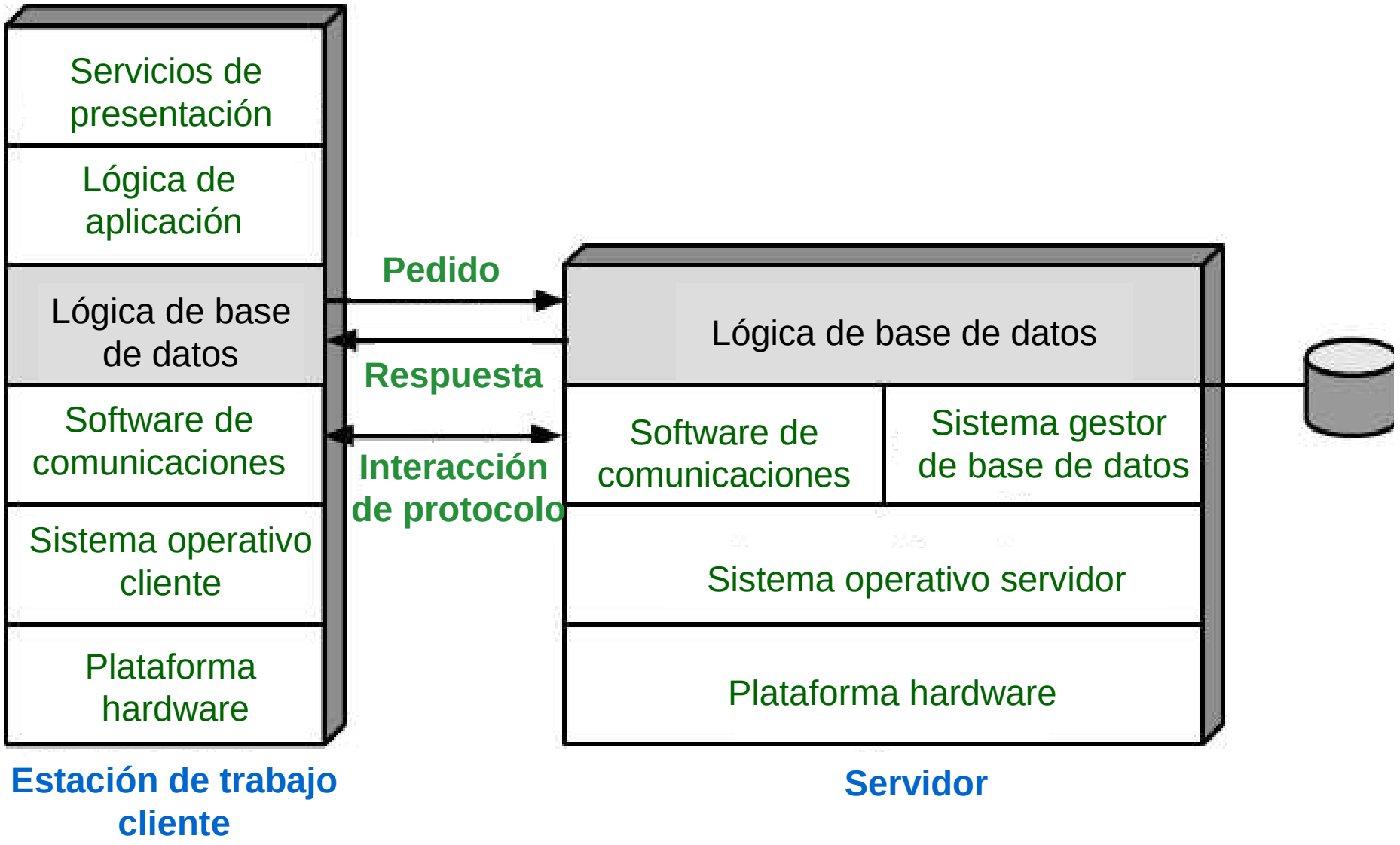
Modelo Cliente-Servidor

- Las funciones reales de la aplicación pueden repartirse entre cliente y servidor de forma que:
  - Se optimicen los recursos de la red y de la plataforma.
  - Se optimice la capacidad de los usuarios para realizar varias tareas.
  - Se optimice la capacidad para cooperar el uno con el otro en el uso de recursos compartidos.

Aplicaciones de Bases de Datos

- El servidor es un servidor de base de datos.
- La interacción entre el cliente y el servidor se hace en forma de transacciones:
  - El cliente realiza una petición a la base de datos y recibe una respuesta de aquella.
- El servidor es responsable de mantener la base de datos.

Aplicaciones de Bases de Datos



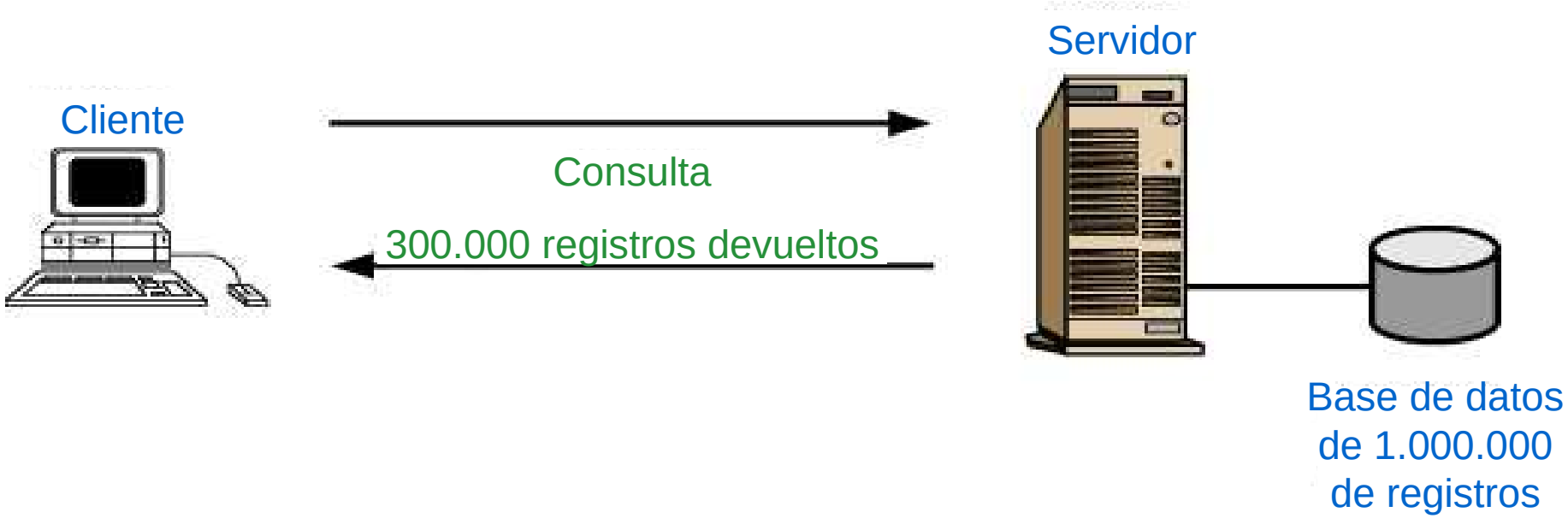
Arquitectura cliente/servidor para aplicaciones de base de datos.

Aplicaciones de Bases de Datos

(a) Cliente/servidor bien empleado

Utilización de bases de datos cliente/servidor.

Aplicaciones de Bases de Datos

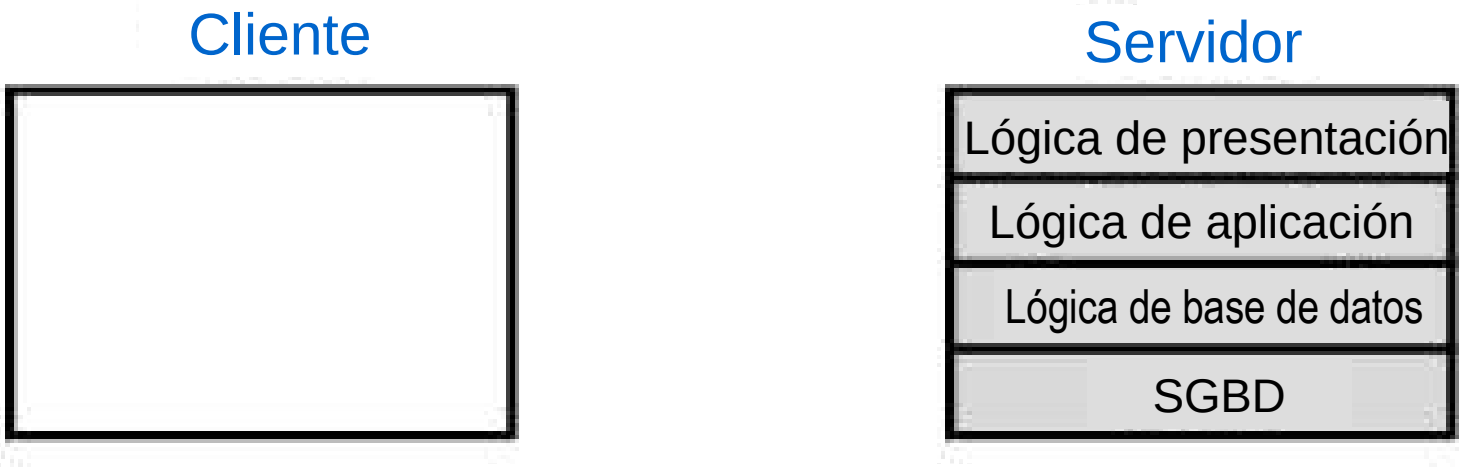


(b) Cliente/servidor mal empleado

Utilización de bases de datos cliente/servidor.

Clases de aplicaciones cliente/servidor

- Proceso basado en una máquina central:
  - No es realmente un proceso cliente/servidor.
  - Entorno tradicional de grandes sistemas.



(a) Proceso basado en una máquina central

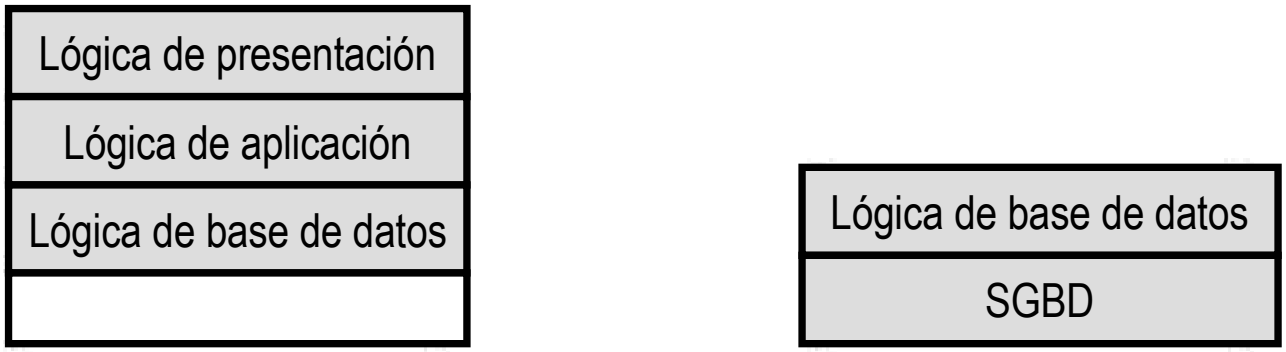
Clases de aplicaciones cliente/servidor

- Proceso basado en el servidor:
  - Todo el tratamiento se hace en el servidor.
  - Los puestos de trabajo de los usuarios ofrecen una interfaz de usuario gráfica.

(b) Proceso basado en el servidor

Clases de aplicaciones cliente/servidor

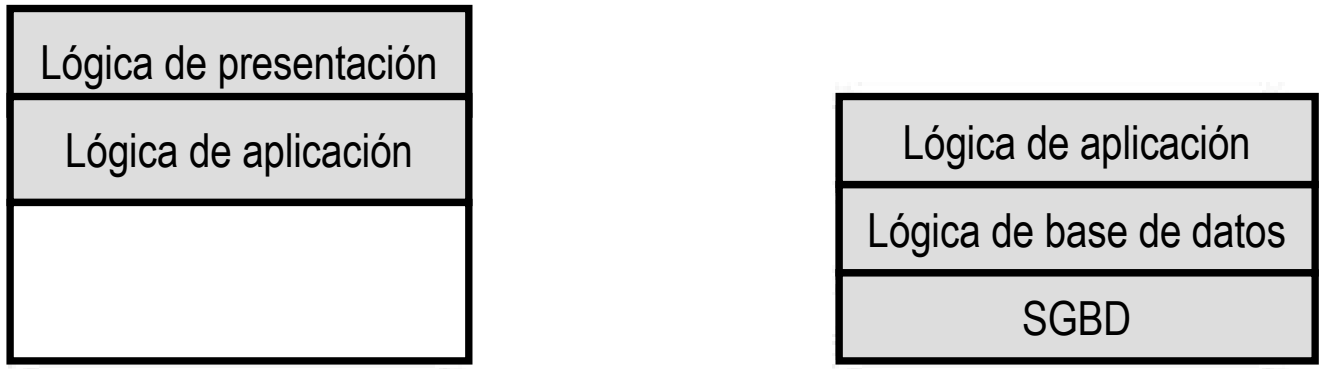
- Proceso basado en el cliente:
  - Casi todo el proceso de la aplicación se hace en el cliente.
  - Las rutinas de validación de datos y otras funciones lógicas de la base de datos se realizan en el servidor.



(d) Proceso basado en el cliente

Clases de aplicaciones cliente/servidor

- Proceso cooperativo:
  - El proceso de la aplicación se lleva a cabo de forma optimizada.
  - Compleja de instalar y mantener.

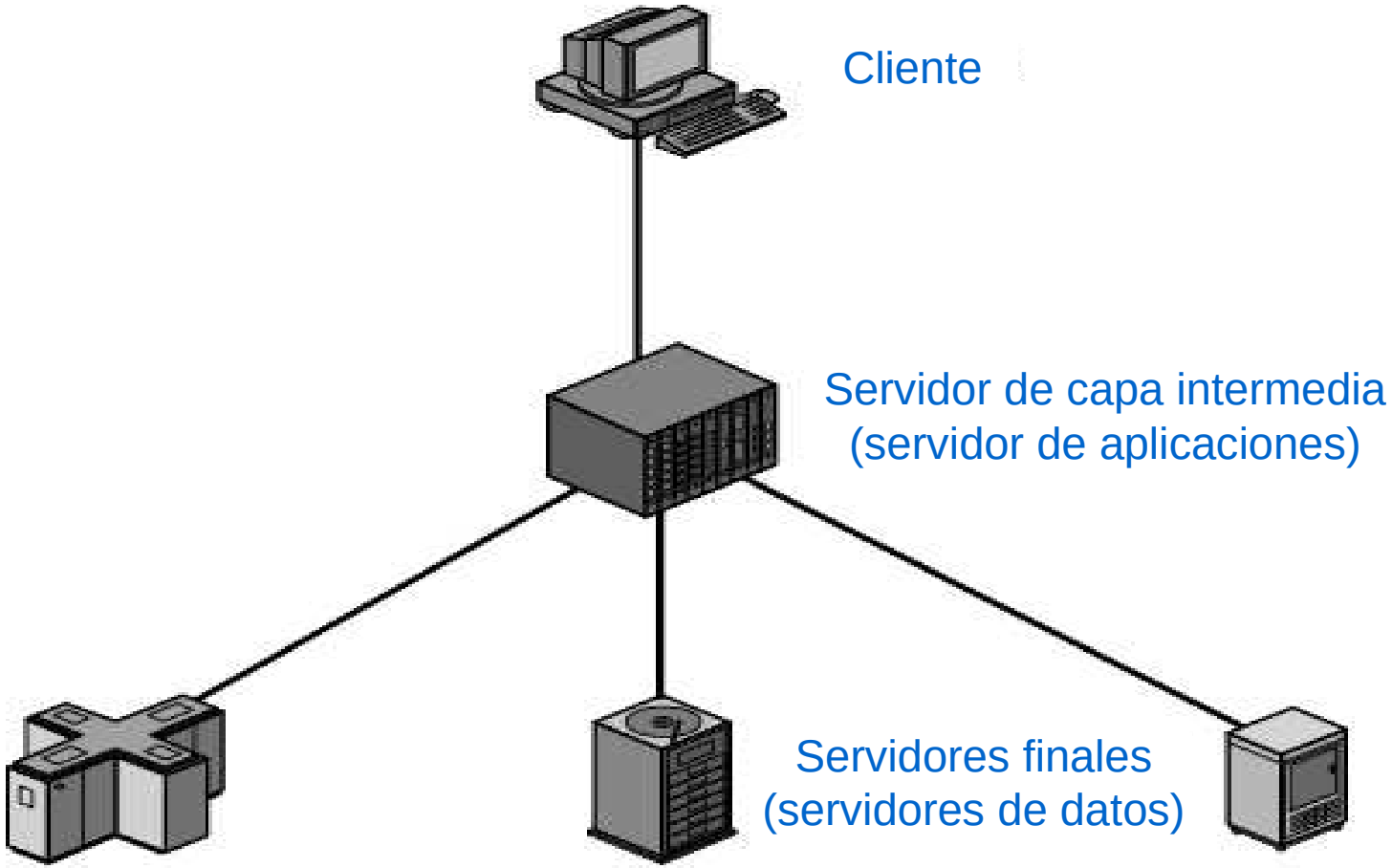


(c) Proceso cooperativo

Arquitectura cliente/servidor de tres capas

- El software de aplicación está distribuido entre tres tipos de máquinas:
  - Máquina de usuario:
    - Cliente
  - Servidor de capa intermedia:
    - Pasarelas.
    - Convierte protocolos.
    - Mezcla e integra resultados de distintas fuentes de datos.
  - Servidor final (*backend*).

Arquitectura cliente/servidor de tres capas



Arquitectura cliente/servidor de tres capas.

Llamadas a Procedimientos Remotos

<RPC>

- Sun RPCs

Llamadas a Procedimientos Remotos (RPC)

Llamadas a Procedimientos Remotos (RPC)

(RPC: Remote Procedure Call)

Es un caso especial del modelo general de pasaje de mensajes.

Es un mecanismo ampliamente aceptado para la intercomunicación de procesos en sistemas distribuidos.

Llamadas a Procedimientos Remotos (RPC)

El modelo RPC

Es similar al bien conocido y entendido modelo de llamadas a procedimientos usado para transferir control y datos.

El mecanismo de RPC es una extensión del anterior porque habilita a hacer una llamada a un procedimiento que no reside en el mismo espacio de direcciones.

Llamadas a Procedimientos Remotos (RPC)

La facilidad de RPC usa un esquema de pasaje de mensajes para intercambiar información entre los procesos *llamador* (proceso cliente) y *llamado* (proceso servidor).

Normalmente el proceso servidor *duerme*, esperando la llegada de un mensaje de requerimiento.

El proceso cliente se bloquea cuando envía el mensaje de requerimiento hasta recibir la respuesta.

Llamadas a Procedimientos Remotos (RPC)

Transparencia de RPC

Transparencia sintáctica: una llamada a procedimiento remoto debe tener la misma sintaxis que una llamada local.

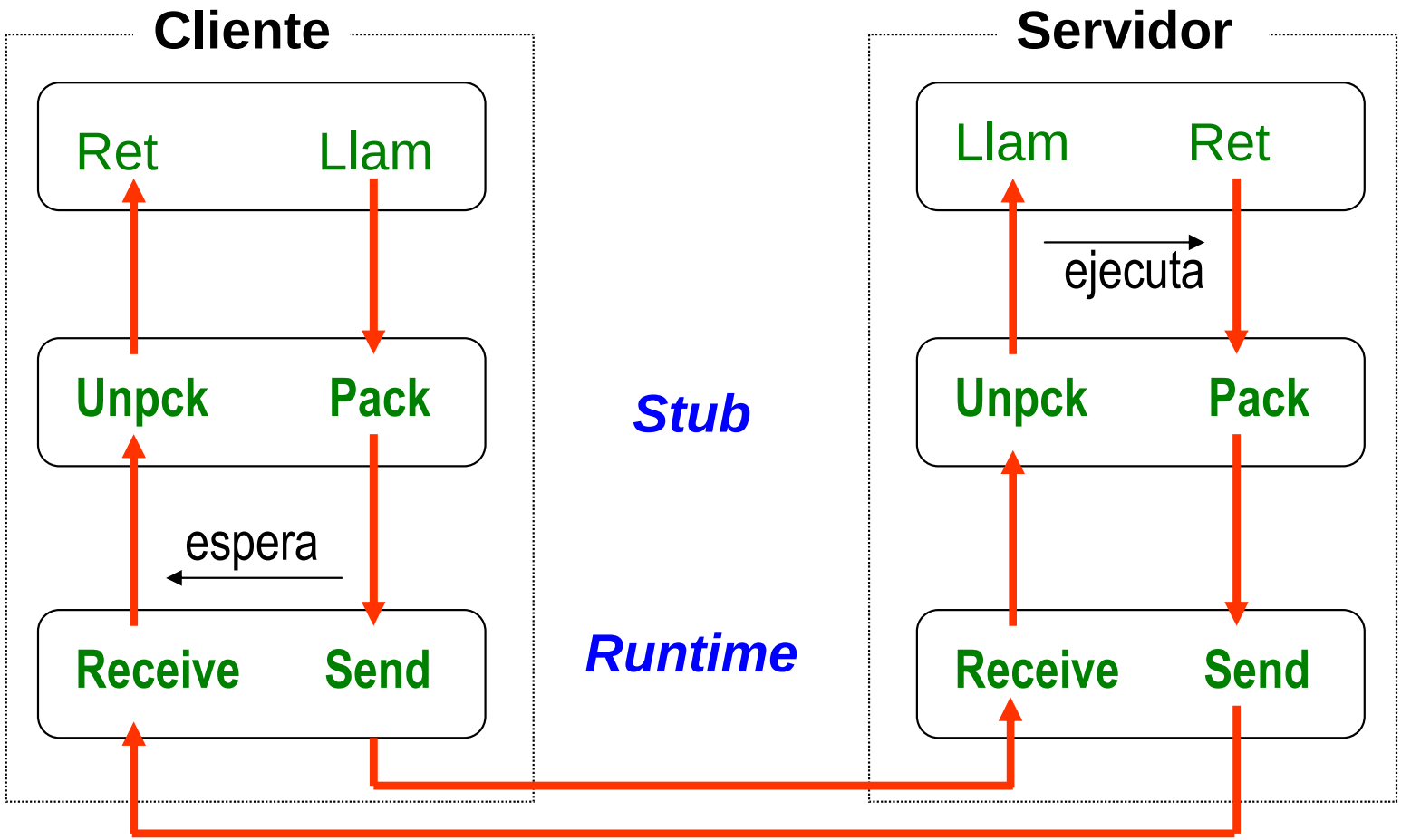
Transparencia semántica: la semántica de una RPC es la misma que para una llamada local.

Llamadas a Procedimientos Remotos (RPC)

Implementación del mecanismo de RPC

- El cliente
- El *stub* cliente
- El runtime RPC
- El *stub* servidor
- El servidor

Llamadas a Procedimientos Remotos (RPC)



Llamadas a Procedimientos Remotos (RPC)

Cliente

Es el que inicia el RPC. Hace una llamada que invoca al *stub*.

Stub cliente

Realiza las siguientes tareas:

a) Empaqueta la especificación del procedimiento objetivo y sus argumentos en un mensaje y pide al *runtime* local que lo envíe al *stub* servidor.

Llamadas a Procedimientos Remotos (RPC)

b) En la recepción de los resultados de la ejecución del proceso, desempaqueta los mismos y los pasa al cliente.

Runtime RPC

Maneja la transmisión de mensajes a través de la red entre las máquinas cliente y servidor.

Llamadas a Procedimientos Remotos (RPC)

Stub servidor

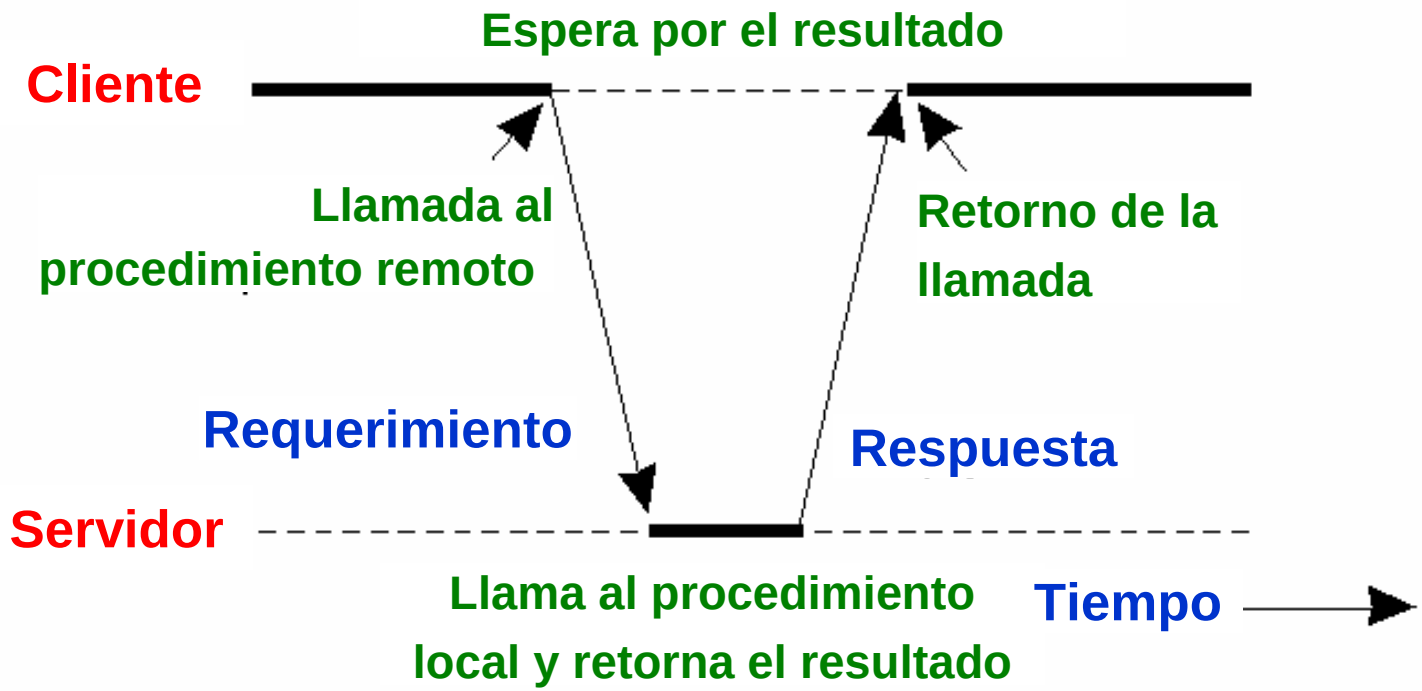
Trabaja en forma simétrica a como lo hace el *stub* cliente.

Servidor

Cuando recibe un requerimiento de llamada del *stub* servidor, ejecuta el procedimiento apropiado y retorna el resultado de la misma al *stub* servidor.

Llamadas a Procedimientos Remotos (RPC)

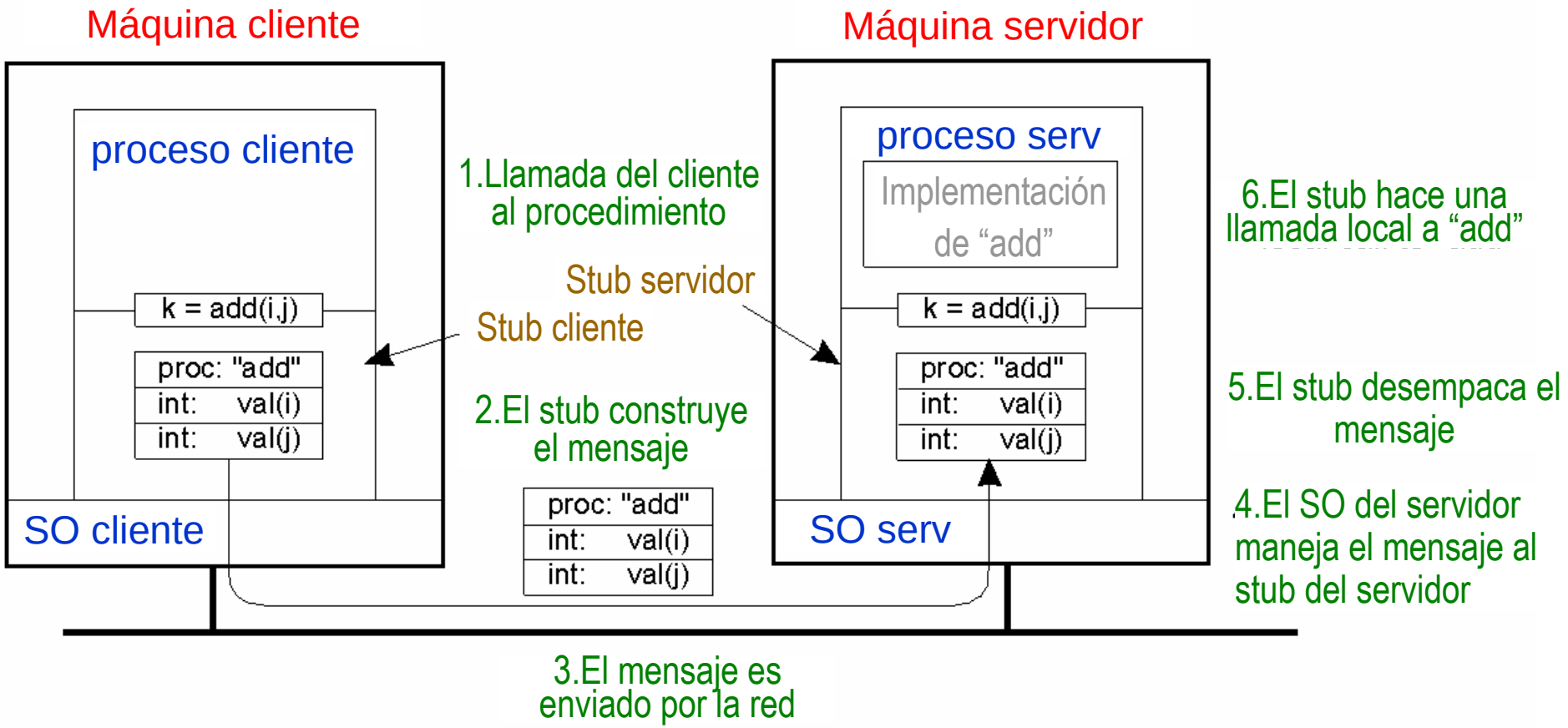
Stubs de Cliente y Servidor



Principio de RPC entre un cliente y el programa servidor.

Llamadas a Procedimientos Remotos (RPC)

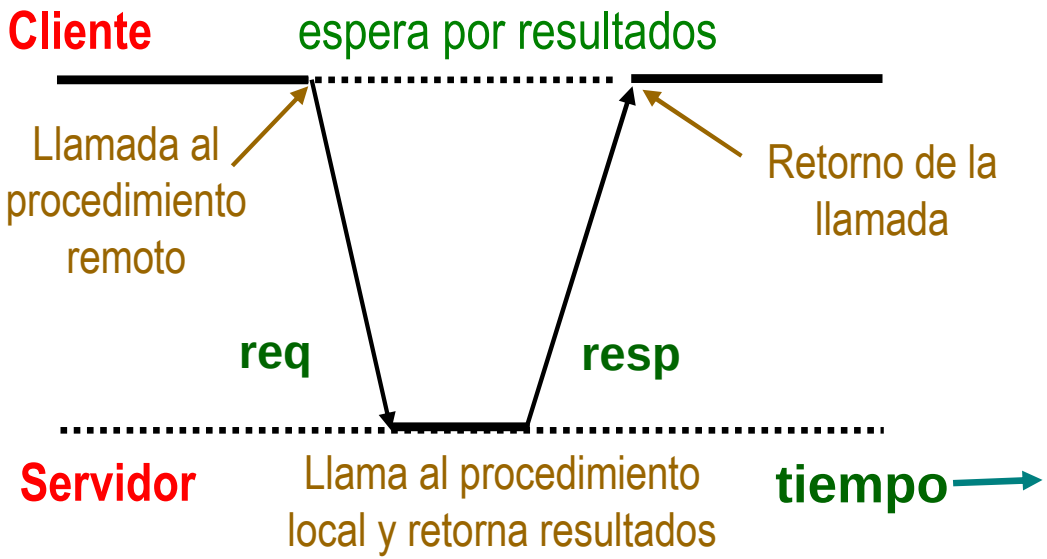
Pasaje de Parámetros por Valor



Pasos que involucra hacer una computación remota por medio de RPC.

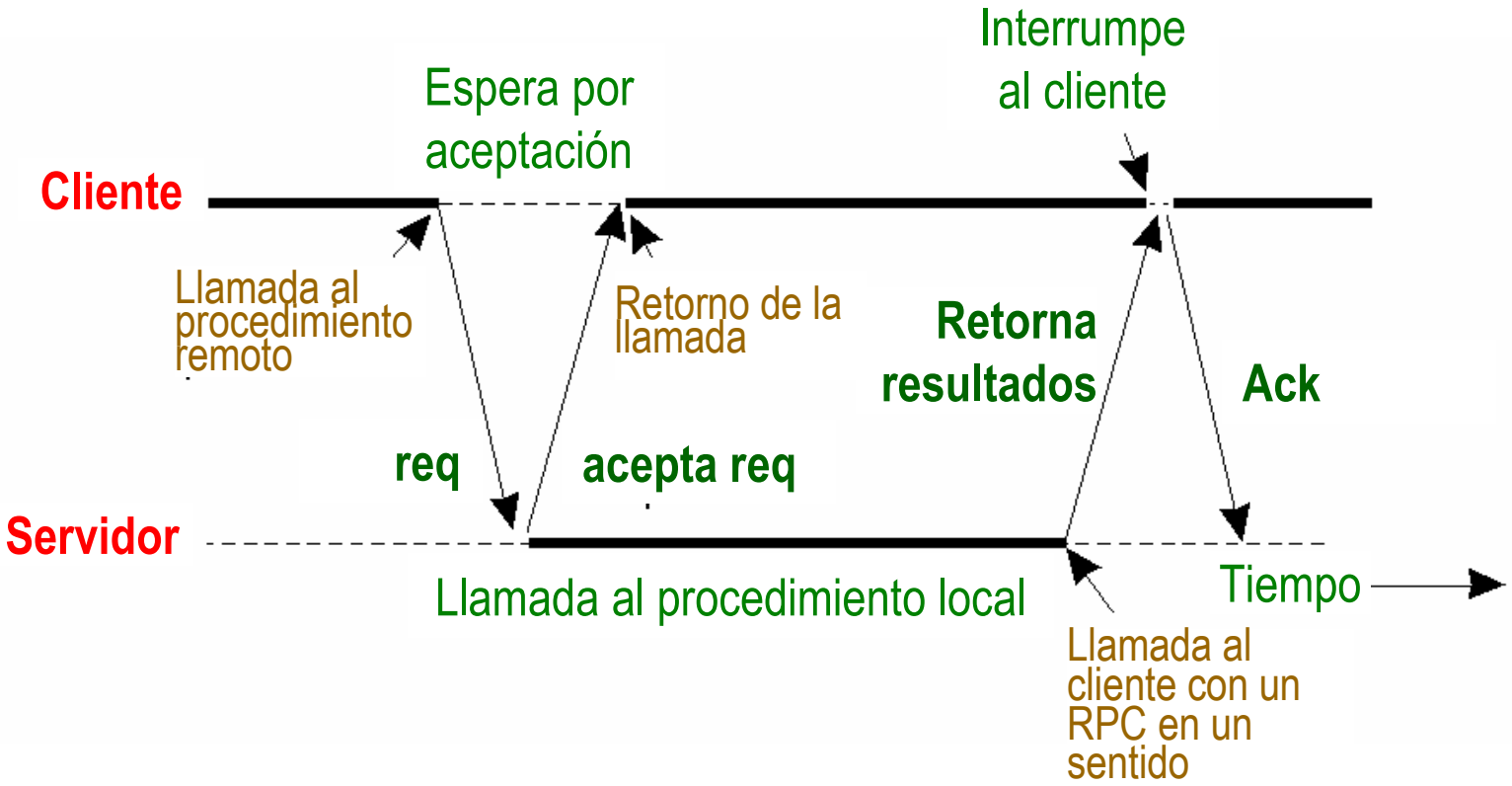
RPC Asincrónico

La interconexión entre cliente y servidor en un RPC tradicional

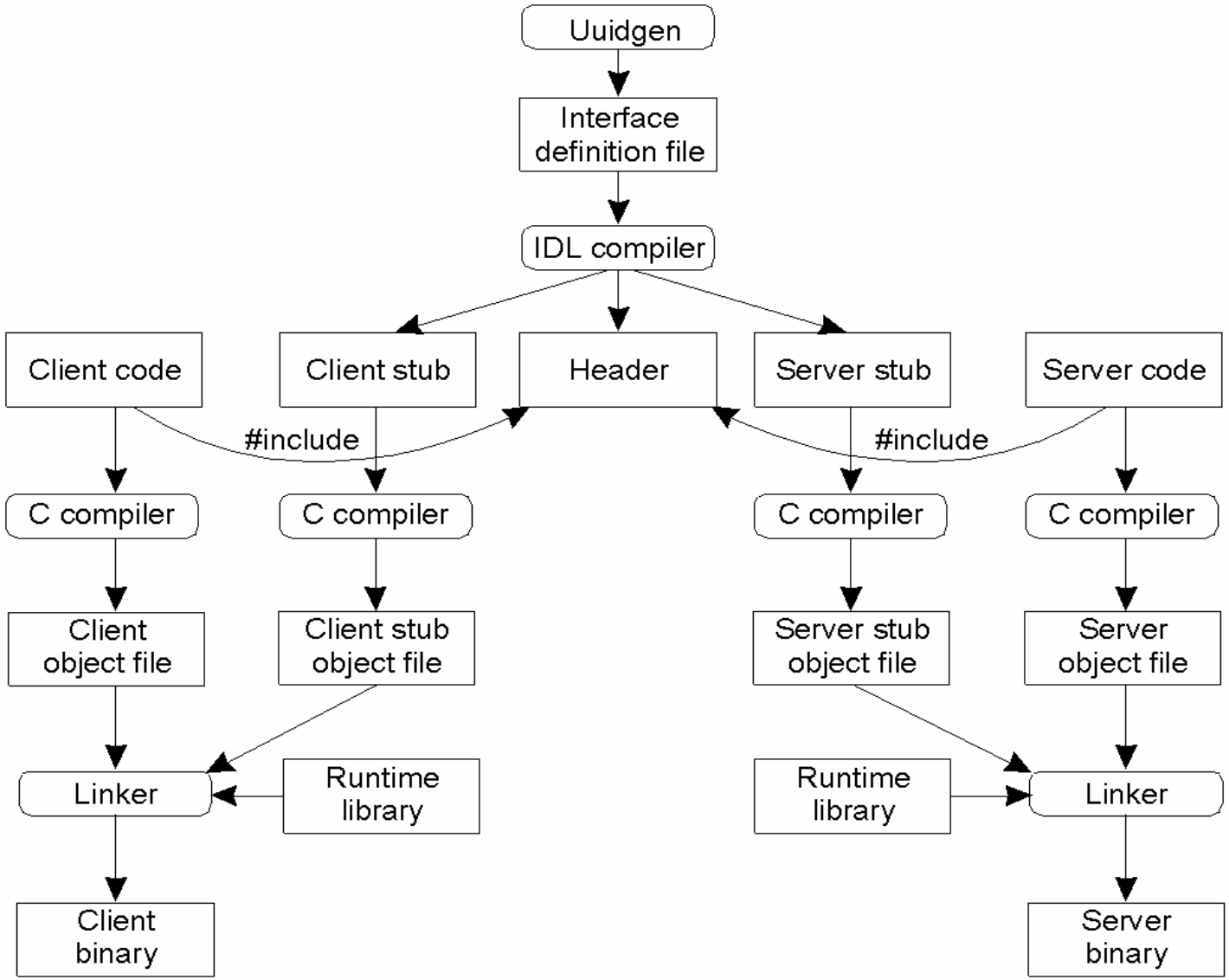


RPC Asincrónico

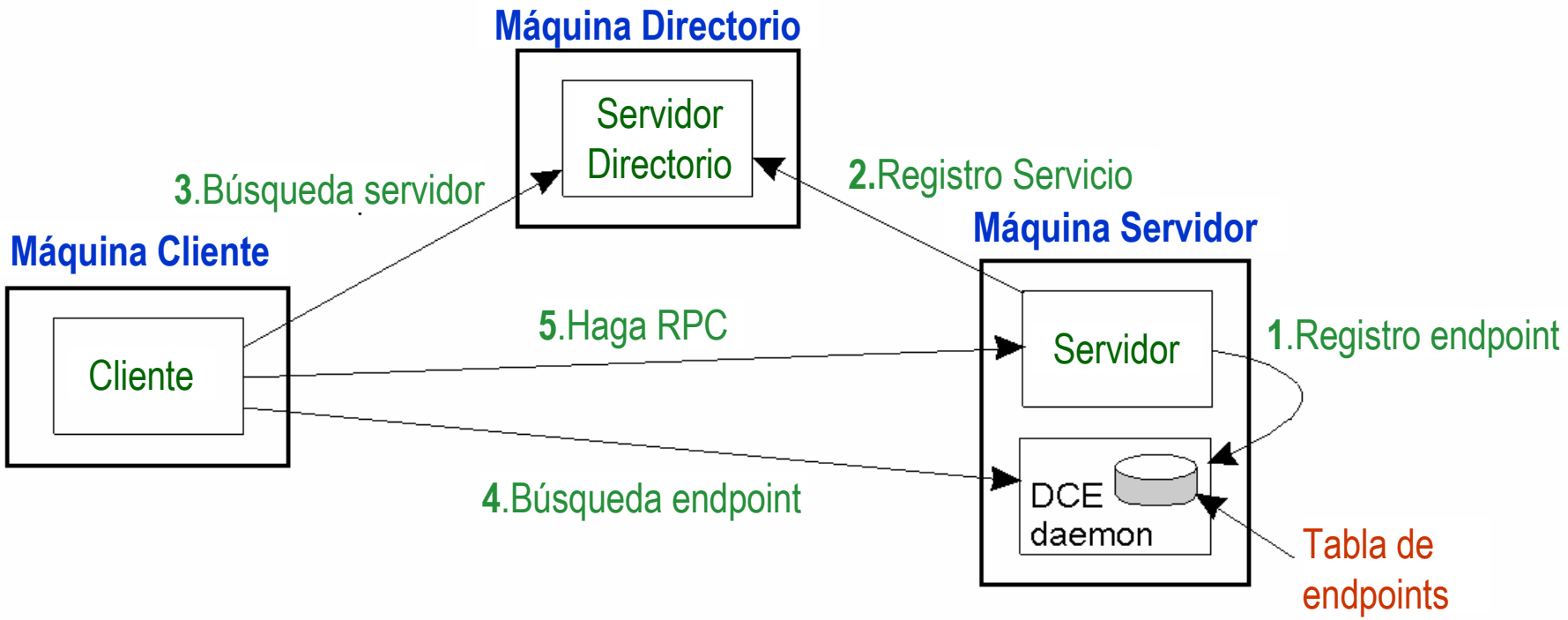
Un cliente y servidor interactuando con dos RPCs asincrónicos.



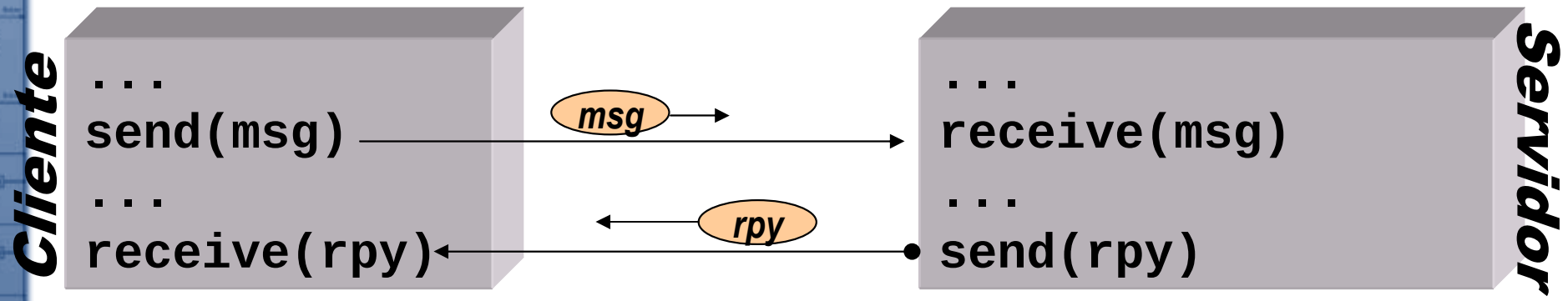
Escribiendo un Cliente y un Servidor para RPC



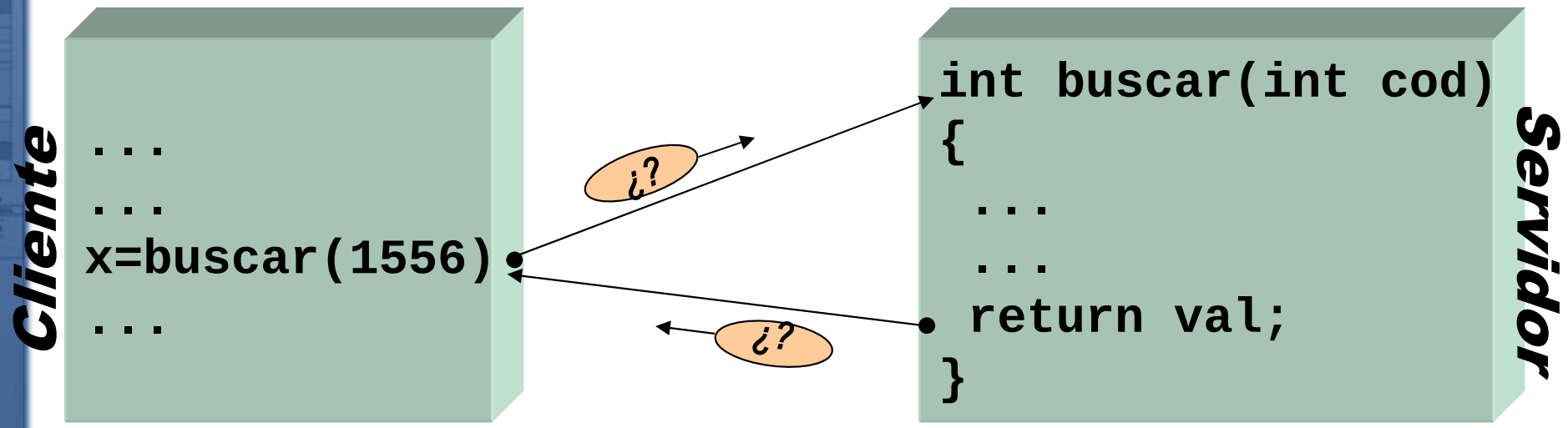
Enlace de un Cliente y un Servidor en RPC



Llamadas a Procedimientos Remotos



Paso de mensajes (visión de bajo nivel)



Llamadas a procedimientos remotos (más alto nivel) ⟶ *Comodidad*

Llamadas a Procedimientos Remotos

Remote Procedure Call: RPC.

Evolución:

- Propuesto por Birrel y Nelson en 1985.
- Sun RPC es la base para varios servicios actuales (NFS o NIS).
- Llegaron a su culminación en 1990 con DCE (*Distributed Computing Environment*) de OSF.
- Han evolucionado hacia orientación a objetos: invocación de métodos remotos (CORBA, RMI).

Lo veremos más adelante

Funcionamiento General de RPC

Cliente:

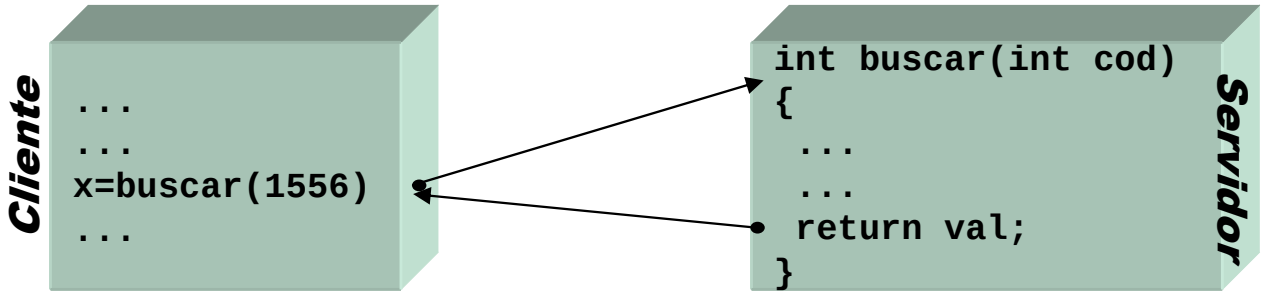
- El proceso que realiza una la llamada a una función.
- Dicha llamada empaqueta los argumentos en un mensaje y se los envía a otro proceso.
- Queda la espera del resultado.

Servidor:

- Se recibe un mensaje consistente en varios argumentos.
- Los argumentos son usados para llamar una función en el servidor.
- El resultado de la función se empaqueta en un mensaje que se retransmite al cliente.

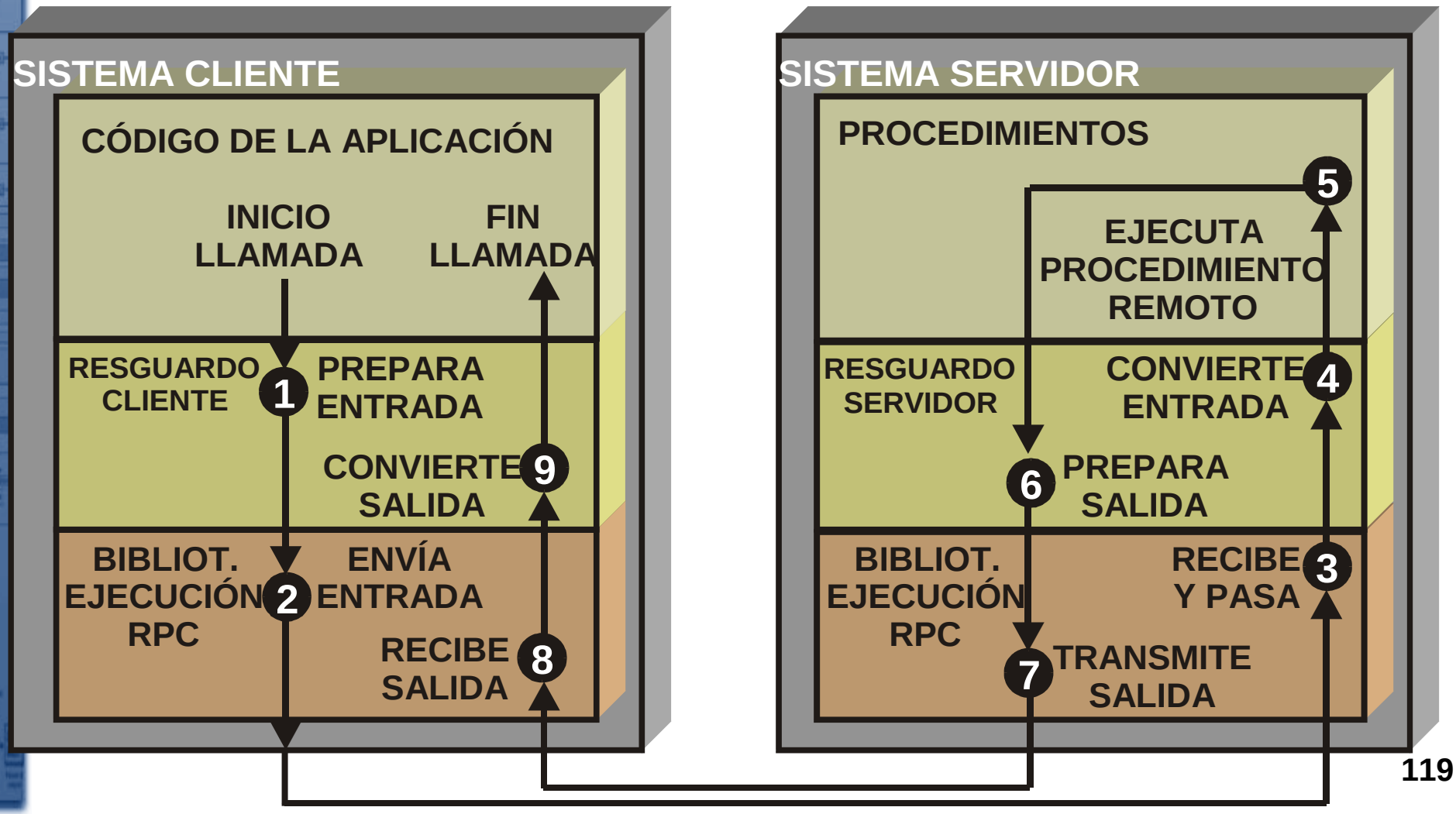
Elementos Necesarios

- Código cliente.
- Código del servidor.
- Formato de representación.
- Definición del interfaz.
- Localización del servidor.
- Semánticas de fallo.



Código Cliente/Código Servidor

Las funciones de abstracción de una llamada RPC a intercambio de mensajes se denominan resguardos (*stubs*).



Resguardos (*stubs*)

Los *stubs* se generan automáticamente por el software de RPC en base a la interfaz del servicio.

- Son independientes de la implementación que se haga del cliente y del servidor. Sólo dependen de la interfaz.

Tareas que realizan:

- Localizan al servidor.
- Empaquetan los parámetros y construyen los mensajes.
- Envían el mensaje al servidor.
- Espera la recepción del mensaje y devuelven los resultados.

Se basan en una librería de funciones RPC para las tareas más habituales.

Formato de Representación

Una de las funciones de los *stubs* es empaquetar los parámetros en un mensaje: aplanamiento (*marshalling*).

Problemas en la representación de los datos

- Servidor y cliente pueden ejecutar en máquinas con arquitecturas distintas.
- XDR (*external data representation*) es un estándar que define la representación de tipos de datos.
- Pasaje de parámetros (entrada/salida):
  - Problemas con los punteros: Una dirección sólo tiene sentido en un espacio de direcciones.

Definición de Interfaces: IDL

IDL (*Interface Definition Language*) es un lenguaje de representación de interfaces:

- Hay muchas variantes de IDL:
  - Integrado con un lenguaje de programación (Cedar, Argus).
  - Específico para describir las interfaces (RPC de Sun y RPC de DCE).
- Define procedimientos y argumentos (No la implementación).
- Se usa habitualmente para generar de forma automática los resguardos (*stubs*).

```
Server ServidorTickets
{
        procedure void reset();
        procedure int  getTicket(in string ident);
        procedure bool isValid(in int ticket);
}
```

Localización del Servidor

La comunicación de bajo nivel entre cliente y servidor por medio de paso de mensajes (por ejemplo sockets). Esto requiere:

- Localizar la dirección del servidor: tanto dirección IP como número de puerto en el caso de sockets.
- Enlazar con dicho servidor (verificar si esta sirviendo).

Estas tareas las realiza el resguardo cliente.

En el caso de servicios cuya localización no es estándar se recurre al enlace dinámico (*dynamic binding*).

Enlace Dinámico

Enlace dinámico: permite localizar objetos con nombre en un sistema distribuido, en concreto, servidores que ejecutan las RPC.

- Tipos de enlace:
  - Enlace no persistente: la conexión entre el cliente y el servidor se establece en cada llamada RPC.
  - Enlace persistente: la conexión se mantiene después de la primera RPC:
    - Útil en aplicaciones con muchas RPC repetidas.
    - Problemas si lo servidores cambian de lugar o fallan.

Enlazador Dinámico

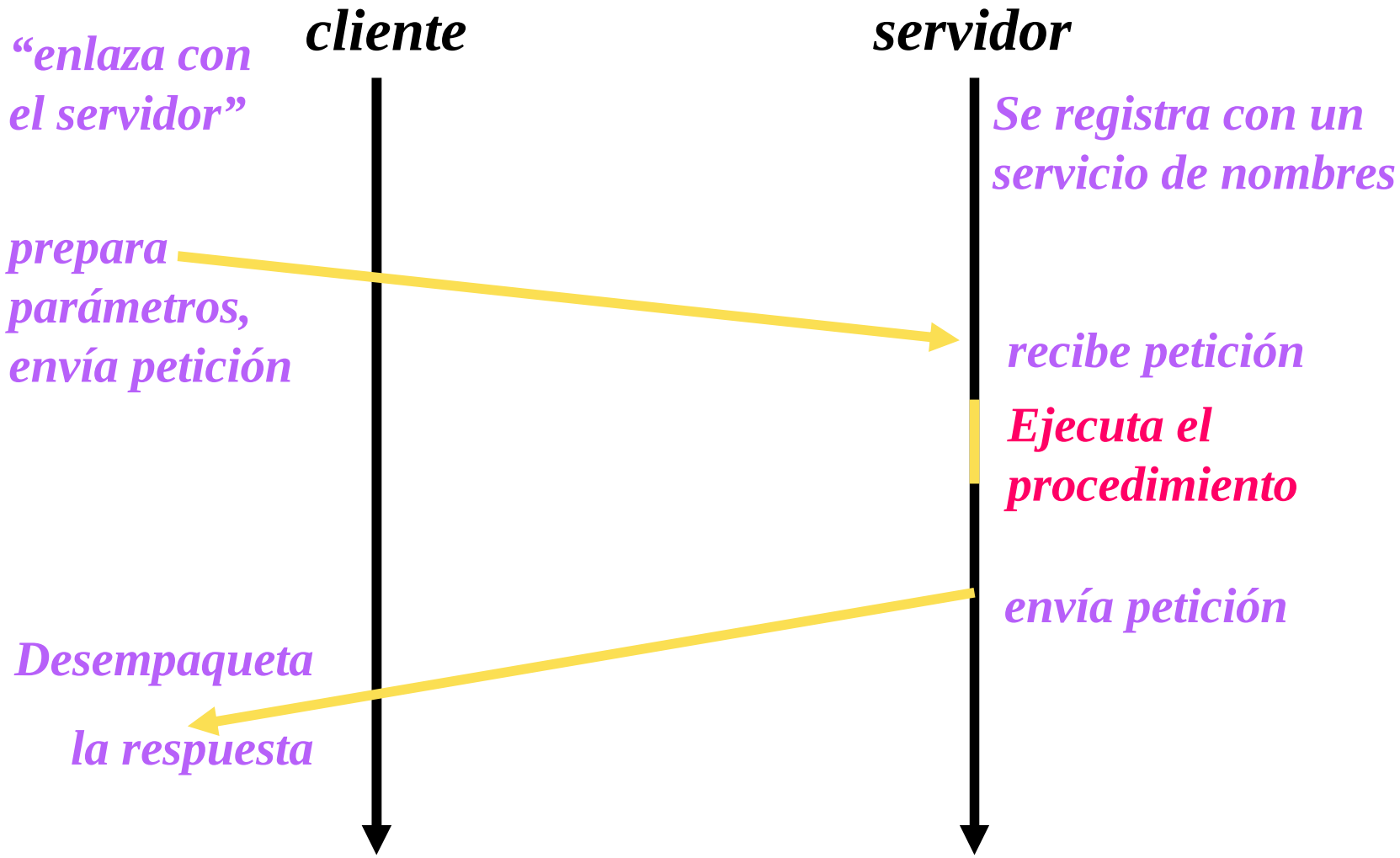
Enlazador dinámico (binder): Es el servicio que mantiene una tabla de traducciones entre nombres de servicio y direcciones. Incluye funciones para:

- Registrar un nombre de servicio (versión).
- Eliminar un nombre de servicio.
- Buscar la dirección correspondiente a un nombre de servicio.

Como localizar al enlazador dinámico:

- Ejecuta en una dirección fija de un computador fijo.
- El sistema operativo se encarga de indicar su dirección.
- Difundiendo un mensaje (*broadcast*) cuando los procesos comienzan su ejecución.

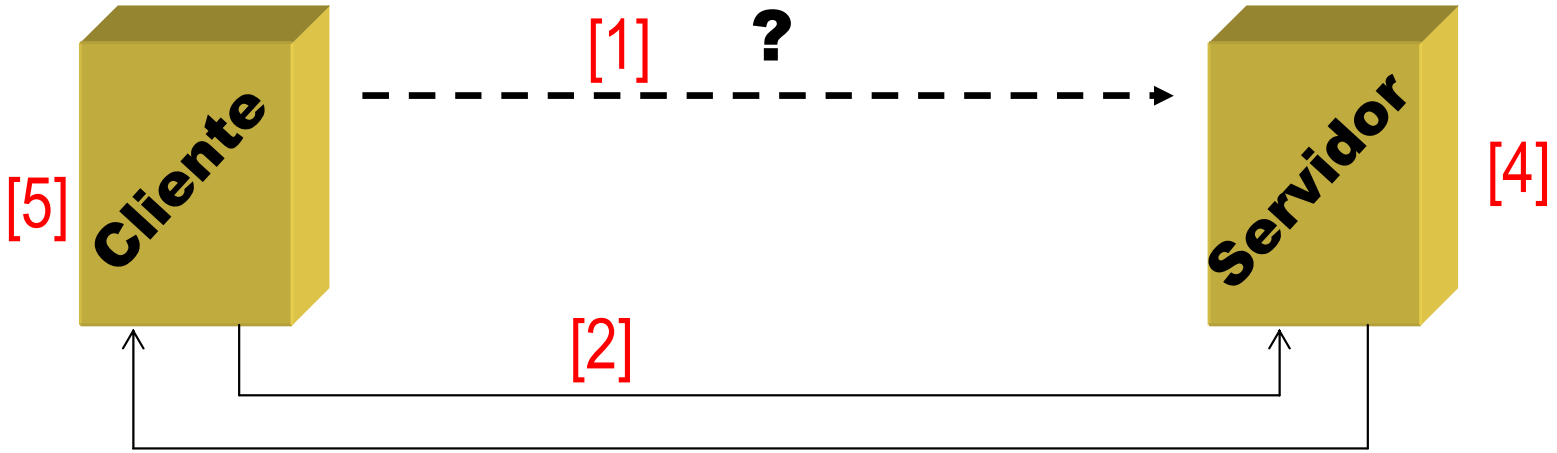
RPC: Protocolo Básico



Semántica Fallos

Problemas que pueden plantear las RPC:

- El cliente no es capaz de localizar al servidor. [1]
- Se pierde el mensaje de petición del cliente al servidor. [2]
- Se pierde el mensaje de respuesta del servidor al cliente. [3]
- El servidor falla después de recibir una petición. [4]
- El cliente falla después de enviar una petición. [5]



Cliente no Puede Localizar al Servidor

- El servidor puede estar caído.
- El cliente puede estar usando una versión antigua del servidor.
- La versión ayuda a detectar accesos a copias obsoletas.
- Cómo indicar el error al cliente
  - Devolviendo un código de error (-1)
    - No es transparente
      - Ejemplo: sumar(a,b)
  - Elevando una excepción
    - Necesita un lenguaje que soporte excepciones.

Pérdida de Mensajes del Cliente

- Es la más fácil de tratar.
- Se activa una alarma (*timeout*) después de enviar el mensaje.
- Si no se recibe una respuesta se retransmite.
- Depende del protocolo de comunicación subyacente.

Pérdidas de Mensajes de Respuesta

- Más difícil de tratar.
- Se pueden emplear alarmas y retransmisiones, pero:
  - ¿Se perdió la petición?
  - ¿Se perdió la respuesta?
  - ¿El servidor va lento?
- Algunas operaciones pueden repetirse sin problemas (operaciones idempotentes)
  - Una transferencia bancaria no es idempotente.
- Solución con operaciones no idempotentes es descartar peticiones ya ejecutadas.
  - Un número de secuencia en el cliente.
  - Un campo en el mensaje que indique si es un pedido original o una retransmisión.

Fallos en los Servidores

- El servidor no ha llegado a ejecutar la operación
  - Se podría retransmitir.
- El servidor ha llegado a ejecutar la operación.
- El cliente no puede distinguir los dos
- ¿Qué hacer?
  - No garantizar nada.
  - Semántica al menos una vez
    - Reintentar y garantizar que la RPC se realiza al menos una vez.
    - No vale para operaciones no idempotentes.
  - Semántica a lo más una vez
    - No reintentar, puede que no se realice ni una sola vez.
  - Semántica de exactamente una
    - Sería lo deseable.

Fallos en los Clientes

- La computación está activa pero ningún cliente espera los resultados (computación huérfana).
  - Gasto de ciclos de CPU.
  - Si cliente rearranca y ejecuta de nuevo la RPC se pueden crear confusiones.

Aspectos de Implementación

- Protocolos RPC
  - Orientados a conexión
    - Fiabilidad se resuelve a bajo nivel, peor rendimiento.
  - No orientados a conexión.
  - Uso de un protocolo estándar o un específico
    - Algunos utilizan TCP o UDP como protocolos básicos.
- Costo de copiar información aspecto dominante en rendimiento:
  - buffer del cliente → buffer del SO local → red → buffer del SO remoto + buffer del servidor.
  - Puede haber más copias en cliente para añadir cabeceras.
  - *scatter-gather*: puede mejorar rendimiento.

RPC de Sun

Utiliza como lenguaje de definición de interfaz IDL:

- Una interfaz contiene un número de programa y un número de versión.
- Cada procedimiento específica un nombre y un número de procedimiento.
- Los procedimientos sólo aceptan un parámetro.
- Los parámetros de salida se devuelven mediante un único resultado.
- El lenguaje ofrece una notación para definir:
  - Constantes.
  - Definición de tipos.
  - Estructuras, uniones.
  - Programas.

RPC de Sun

- *rpcgen* es el compilador de interfaces que genera:
  - Resguardo del cliente
  - Resguardo del servidor y procedimiento principal del servidor.
  - Procedimientos para el aplanamiento (*marshalling*)
  - archivo de cabecera (.h) con los tipos y declaración de prototipos.
- *Enlace dinámico*
  - El cliente debe especificar el *host* donde ejecuta el servidor
  - El servidor se registra (número de programa, número de versión y número de puerto) en el *port mapper* local
  - El cliente envía una petición al *port mapper* del *host* donde ejecuta el servidor.

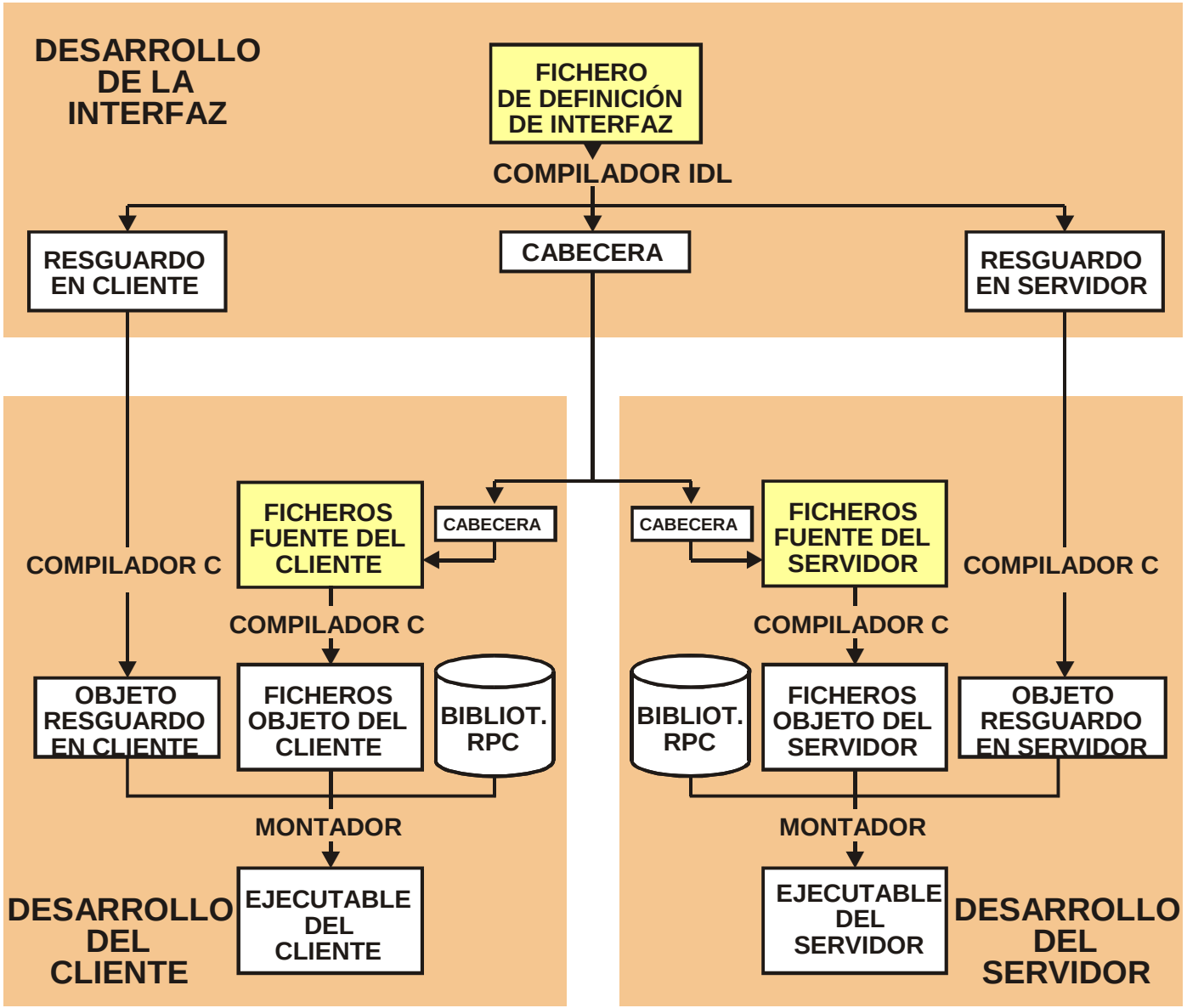
Ejemplo de Archivo IDL (Sun RPC)

```
struct peticion {
  int a;
  int b;
};

program SUMAR {
  version SUMAVER {
      int SUMA(peticion) = 1;
  } = 1;
} = 99;
```

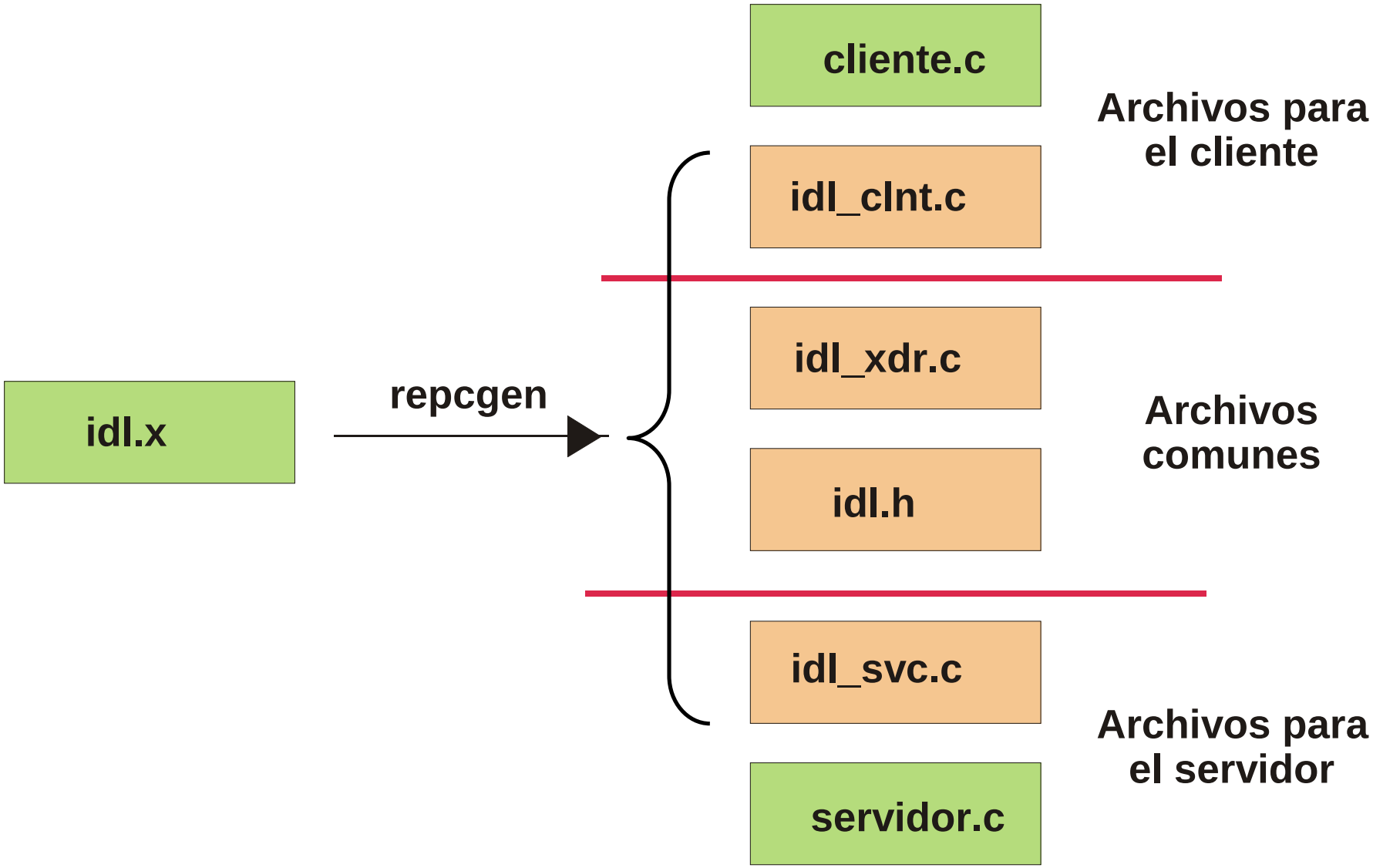
Programación con un Paquete de RPC

- El programador debe proporcionar:
  - La definición de la interfaz (archivo idl)
    - Nombres de las funciones.
    - Parámetros que el cliente pasa al servidor.
    - Resultados que devuelve el servidor al cliente.
  - El código del cliente.
  - El código del servidor.
- El compilador de IDL proporciona:
  - El resguardo del cliente.
  - El resguardo del servidor.

Programación con RPC



Esquema de una Aplicación



Entornos de Objetos Distribuidos

<Objetos Remotos>

- Java RMI
- CORBA

Motivación

La extensión de los mecanismos de RPC a una programación orientada a objetos dio lugar a los modelos de objetos distribuidos.

Ventajas:

- Los métodos remotos están asociados a objetos remotos.
- Más natural para desarrollo orientado a objetos.
- Admite modelos de programación orientada a eventos.

Problemas:

- El concepto de referencia a objeto es fundamental.
- Objetos volátiles y objetos persistentes.

Objetos-Distribuidos

Características:

- Uso de un *Middleware*: Nivel de abstracción para la comunicación de los objetos distribuidos. Oculta:
  - Localización de objetos.
  - Protocolos de comunicación.
  - Hardware de computadora.
  - Sistemas Operativos.
- Modelo de objetos distribuidos: Describe los aspectos del paradigma de objetos que es aceptado por la tecnología: Herencia, Interfaces, Excepciones, Polimorfismo, ...
- Recogida de basura (*Garbage Collection*): Determina los objetos que no están siendo usados para a liberar recursos.

Tecnologías de Objetos Distribuidos

Actualmente existen tres tecnologías de desarrollo de sistemas distribuidos basados en objetos:

- ANSA (1989-1991) fue el primer proyecto que intentó desarrollar una tecnología para modelar sistemas distribuidos complejos con objetos.
- DCOM de Microsoft.
- CORBA de OMG.
- Tecnologías Java de Sun Microsytems:
  - *Remote Method Invocation* (RMI).
  - *Enterprise Java Beans* (EJB).
  - Jini.
- Diferentes entornos de trabajo propietarios.

Java RMI

El soporte para RMI en Java está basado en las interfaces y clases definidas en los paquetes:

- *java.rmi*
- *java.rmi.server*

Características de Java RMI:

- Los argumentos y resultados se pasan mediante RMI por valor (nunca por referencia).
- Un objeto remoto se pasa por referencia.
- Es necesario tratar mayor número de excepciones que en el caso de invocación de métodos locales.

Java RMI

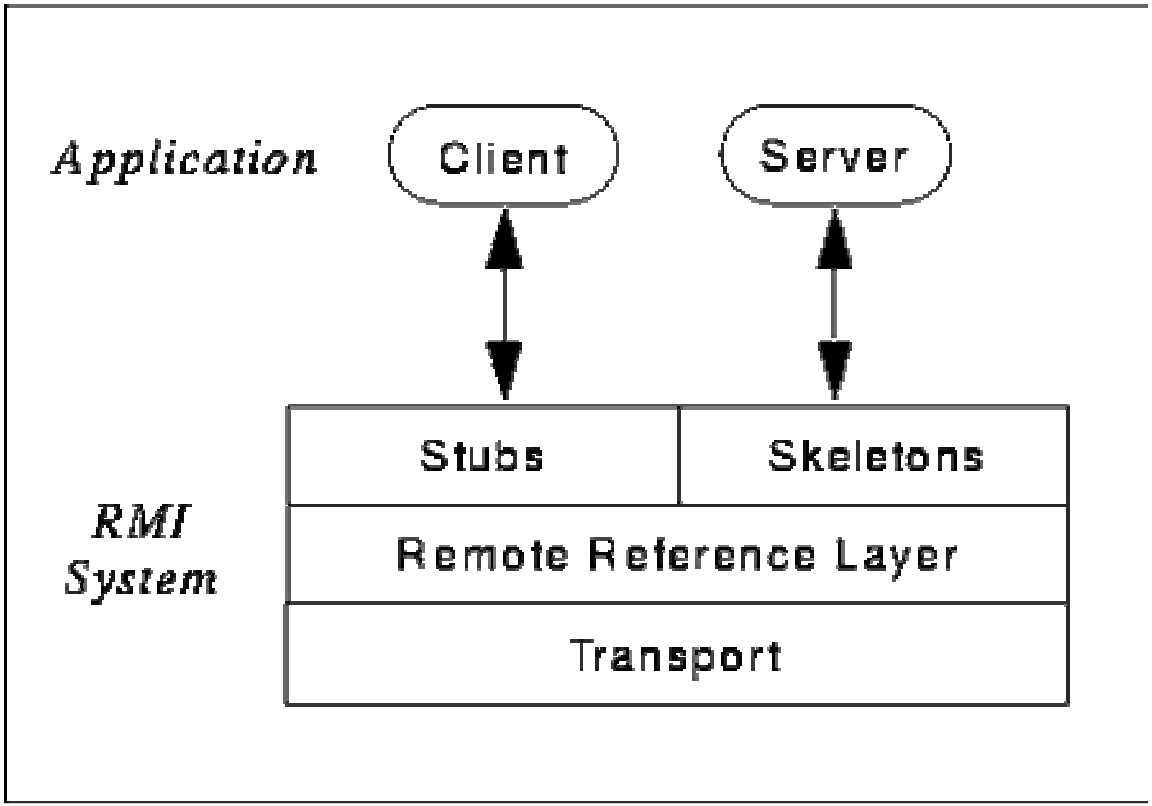
Localización de objetos remotos:

- Servidor de nombres: *java.rmi.Naming*

Ejemplo:

```
Cuenta cnt = new CuentaImpl();
String url = "rmi://java.Sun.COM/cuenta";
// enlazamos una url a un objeto remoto
java.rmi.Naming.bind(url, cnt);
      ....
// búsqueda de la cuenta
cnt=(Cuenta)java.rmi.Naming.lookup(url);
```

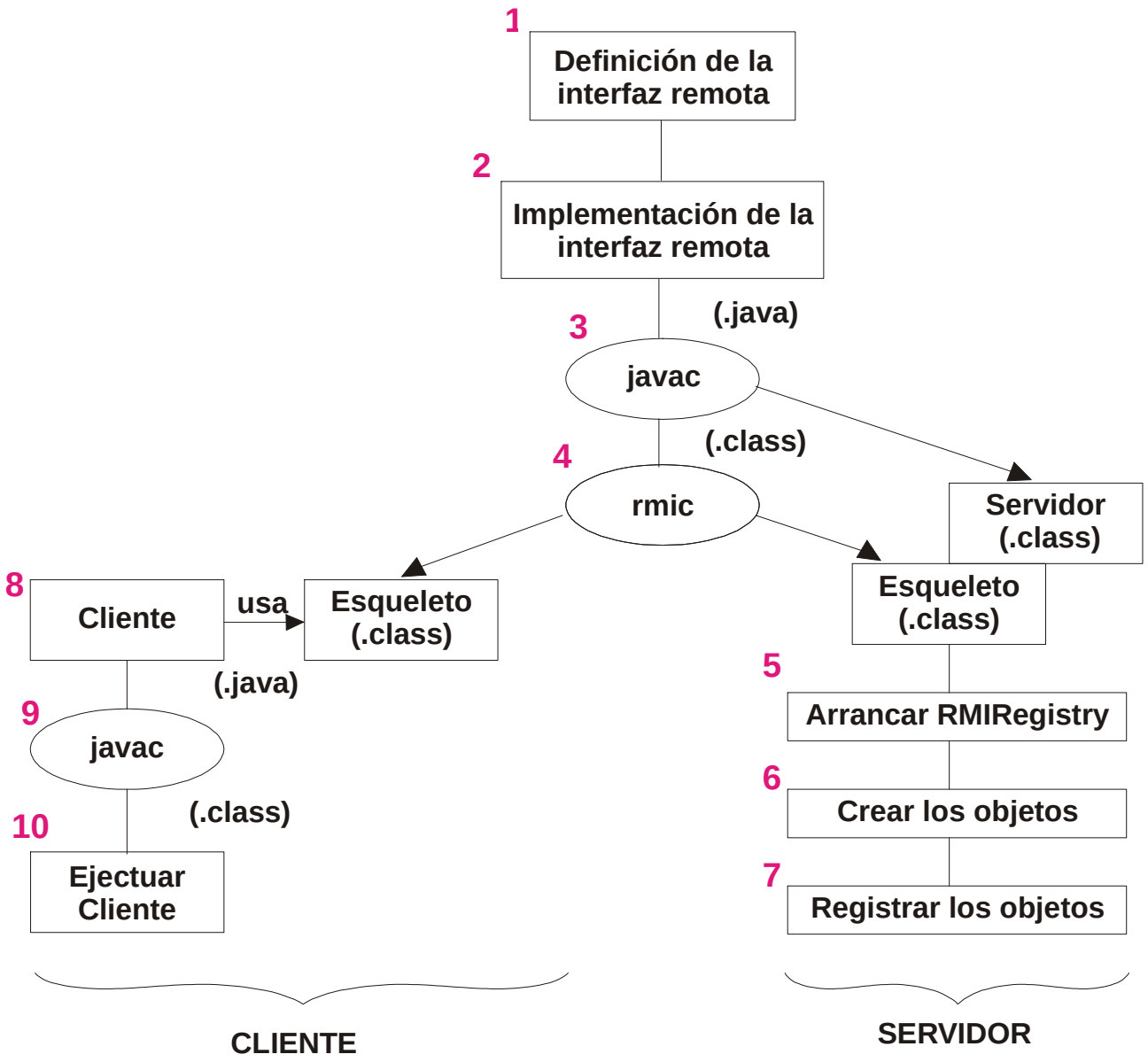
Arquitectura de Java RMI



Arquitectura de Java RMI

- ***Nivel de transporte***: se encarga de las comunicaciones y de establecer las conexiones necesarias
- ***Nivel de gestión de referencias remotas***: trata los aspectos relacionados con el comportamiento esperado de las referencias remotas (mecanismos de recuperación, etc.)
- ***Nivel de resguardo/esqueleto*** (*proxy/skeleton*) que se encarga del aplanamiento (serialización) de los parámetros
  - *proxy*: resguardo local. Cuando un cliente realiza una invocación remota, en realidad hace una invocación de un método del resguardo local.
  - Esqueleto (*skeleton*): recibe las peticiones de los clientes, realiza la invocación del método y devuelve los resultados.

Desarrollo de Aplicaciones RMI



Registro de Objetos

Cualquier programa que quiera instanciar un objeto de esta clase debe realizar el registro con el servicio de nombrado de la siguiente forma:

```
Cuenta mi_cuenta=
(Cuenta)Naming.lookup("rmi://"+host+"/"+”MiCuenta");
```

Antes de arrancar el cliente y el servidor, se debe arrancar el programa *rmiregistry* en el servidor para el servicio de nombres

OMG

OMG = Object Management Group

Conjunto de organizaciones que cooperan en la definición de estándares para la interoperabilidad en entornos heteregéneos.

Fundado en 1989, en la actualidad lo componen más de 700 empresas y otros organismos.

OMA

OMA = Object Management Architecture

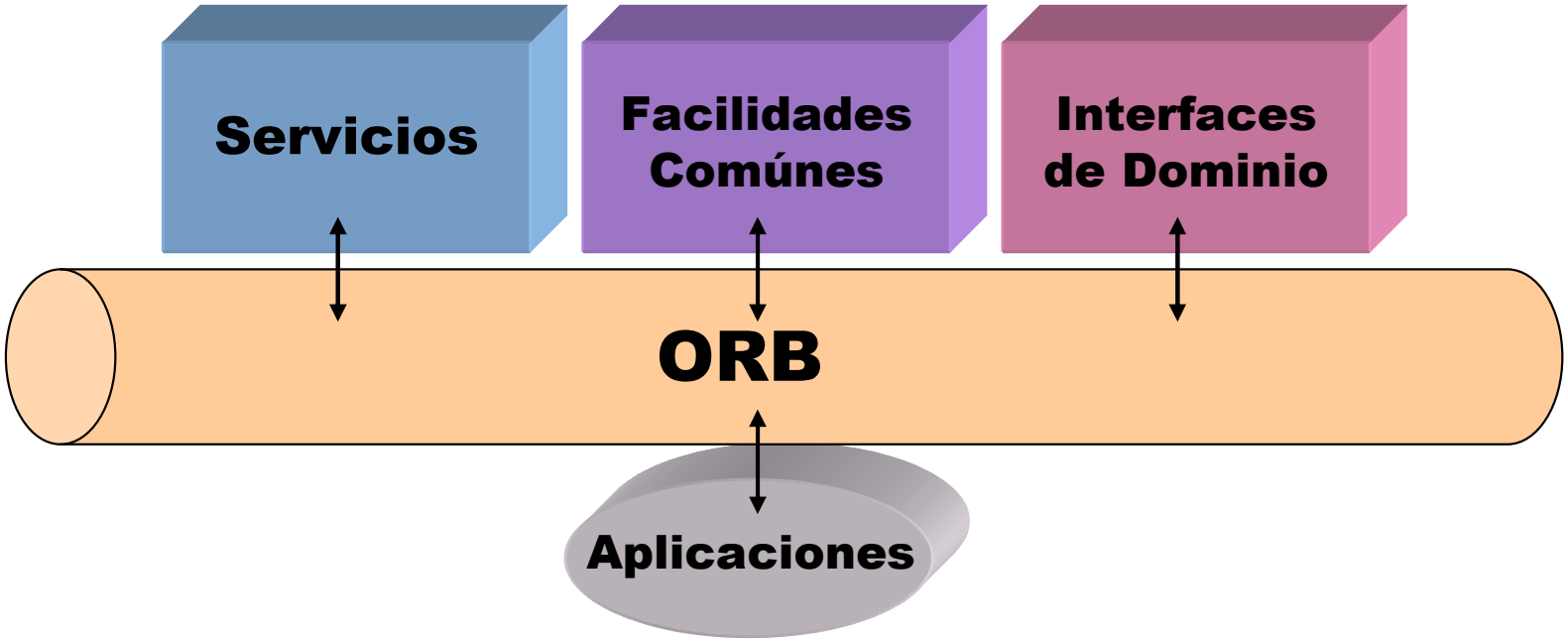
Arquitectura de referencia sobre cual se pueden definir aplicaciones distribuidas sobre un entorno heterogéneo. CORBA es la tecnología asociada a esta arquitectura genérica.

Formalmente esta dividida en una serie de modelos:

- Modelo de Objetos
- Modelo de Interacción
- ...

OMA

Una aplicación definida sobre OMA esta compuesta por una serie de objetos distribuidos que cooperan entre si. Estos objetos se clasifican en los siguientes grupos:



OMA

Servicios:

- Proporcionan funciones elementales necesarias para cualquier tipo de entorno distribuido, independientemente del entorno de aplicación.
- Los tipos de funciones proporcionados son cuestiones tales como la resolución de nombres, la notificación asíncrona de eventos o la creación y migración de objetos.
- Concede un valor añadido sobre otras tecnologías (por ejemplo RMI).
- Están pensados para grandes sistemas.

OMA

Facilidades Comunes:

- Proporcionan funciones, al igual que los servicios válidas para varios dominios pero más complejas. Están orientadas a usuarios finales (no al desarrollo de aplicaciones).

- Un ejemplo de este tipo de funciones es el DDCF (*Distributed Document Component Facility*) formato de documentación basado en OpenDoc.

- (También denominadas **Facilidades Horizontales**)

OMA

Interfaces de Dominio:

- Proporcionan funciones complejas, al igual que las Facilidades, pero restringidas a campos de aplicación muy concretos. Por ejemplo, telecomunicaciones, aplicaciones médicas o financieras, etc.
- Muchos grupos de interés (SIGs) trabajan sobre estas especificaciones.
- (También denominadas Facilidades Verticales)

OMA

Aplicaciones:

- El resto de funciones requeridas por una aplicación en concreto. Es el único grupo de objetos que OMG no define, pues esta compuesto por los objetos propios de cada caso concreto.

- Estos son los objetos que un sistema concreto tiene que desarrollar. El resto (servicios, facilidades) pueden venir dentro del entorno de desarrollo.

OMA

ORB:

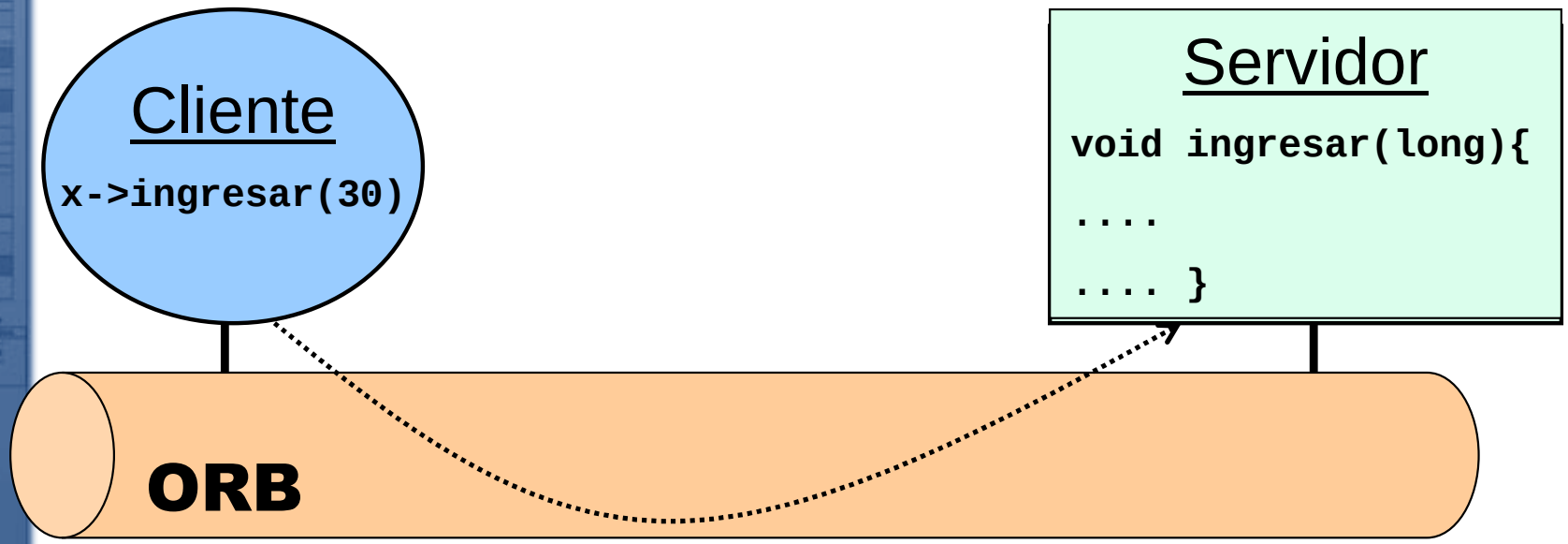
- (*Object Request Broker*)

- Es el elemento central de la arquitectura. Proporciona las funcionalidades de interconexión entre los objetos distribuidos (servicios, facilidades y objetos de aplicación) que forman una aplicación.

- Representa un bus de comunicación entre objetos.

ORB

Para posibilitar la comunicación entre dos objetos cualesquiera de una aplicación se realiza por medio del ORB. El escenario de aplicación elemental es:



IDL de CORBA

(*Interface Definition Language*)

Es el lenguaje mediante el cual se describen los métodos que un determinado objeto del entorno proporciona al resto de elementos del mismo.

```
interface Cuenta
{
  void ingresar(in long cantidad);
  void retirar(in long cantidad);
  long balance();
};
```

IDL de CORBA

Language Mappings:

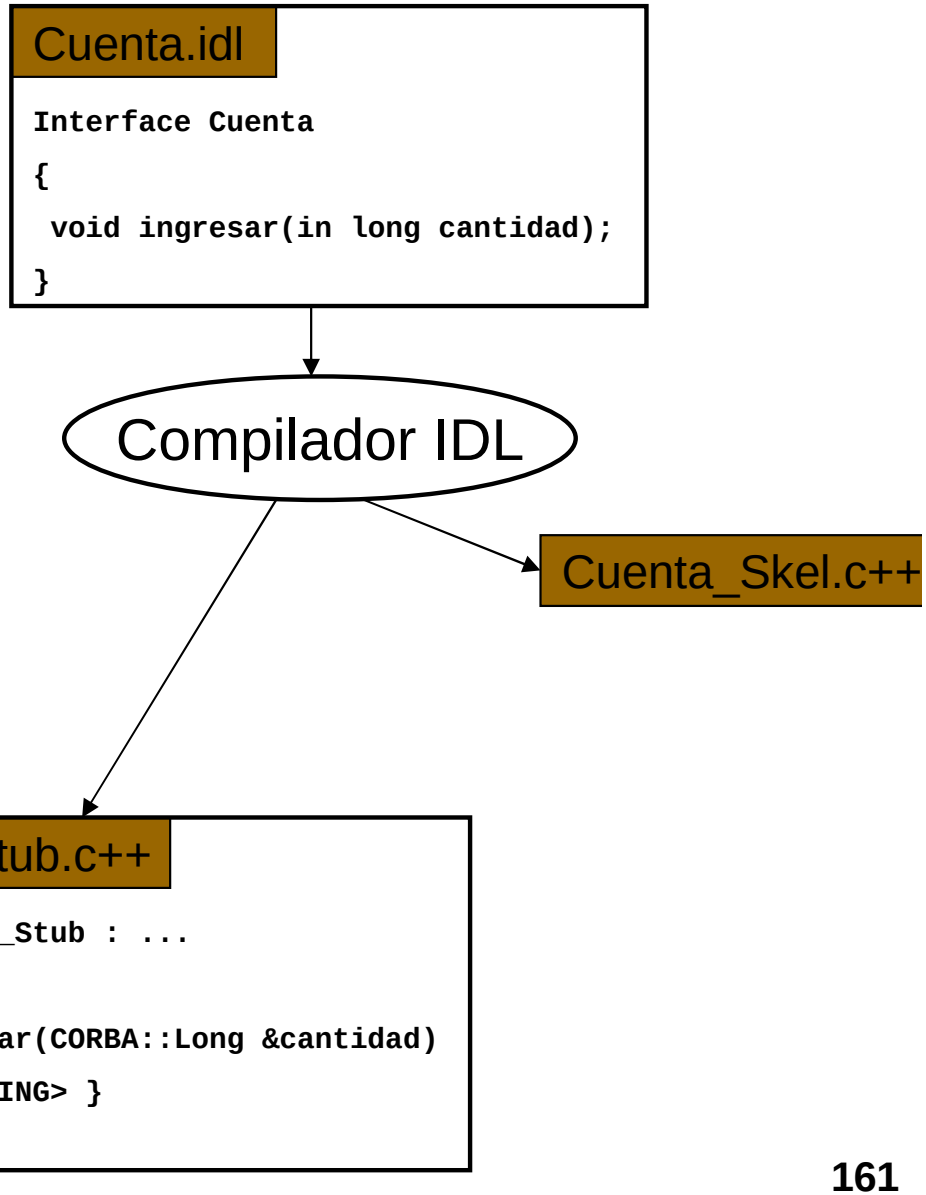
- Traducen la definición IDL a un lenguaje de programación como:
  - C++,
  - Ada,
  - COBOL,
  - SmallTalk,
  - Java,
  - ...

- Este proceso genera dos fragmentos de código denominados *Stub* y *Skeleton* que representan el código de cliente y servidor respectivamente.

IDL de CORBA

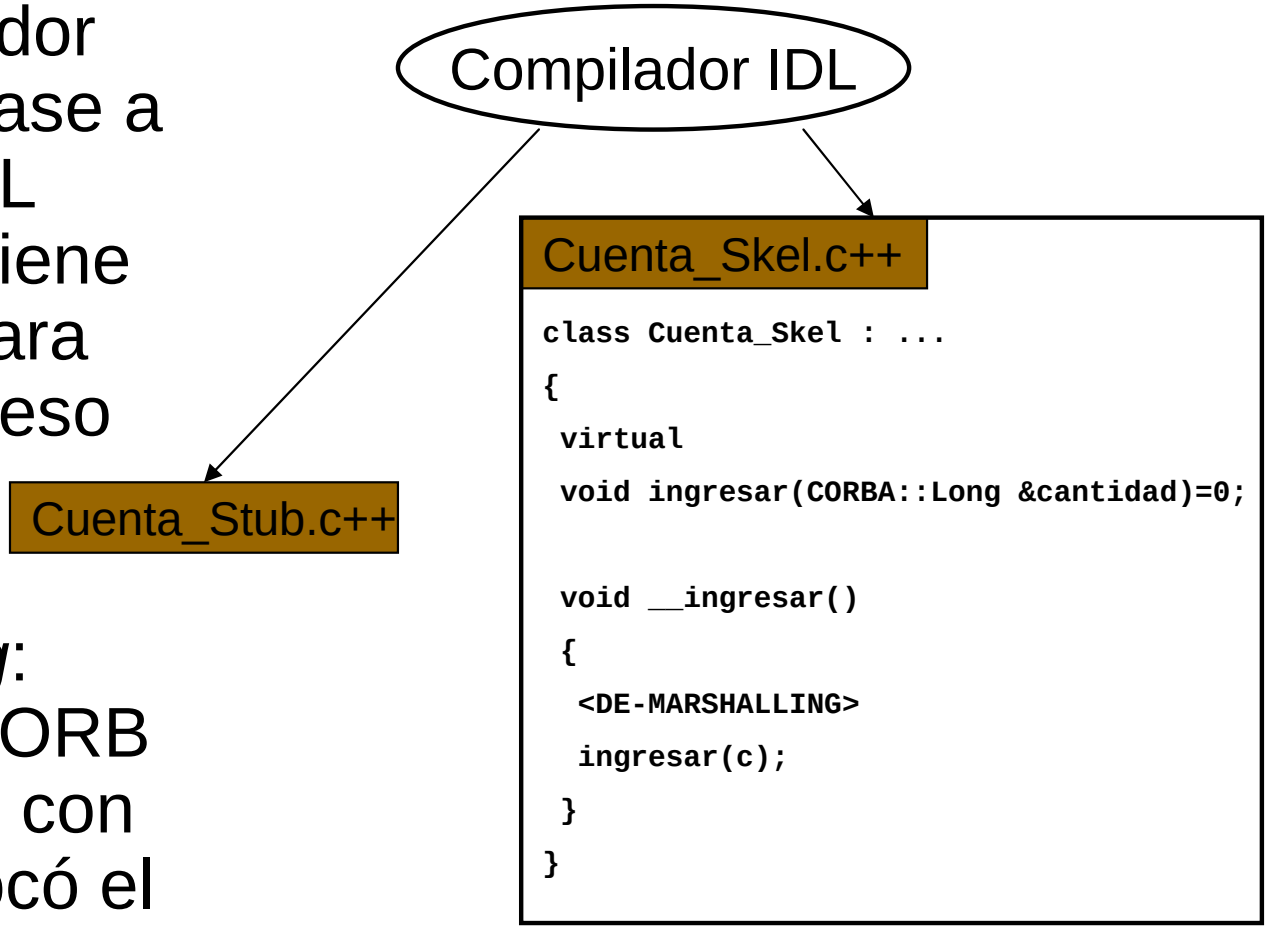
El código cliente generado en base a la definición IDL (*stub*) contiene las llamadas para realizar el proceso de *marshalling*.

Marshalling: Traducción de los argumentos a un formato intermedio y pedir al ORB su ejecución.



IDL de CORBA

El código servidor generado en base a la definición IDL (*skeleton*) contiene las llamadas para realizar el proceso inverso (*de-marshalling*).



De-marshalling: Recuperar del ORB los parametros con los que se invocó el método, construir la llamada y realizar la petición.

IDL de CORBA

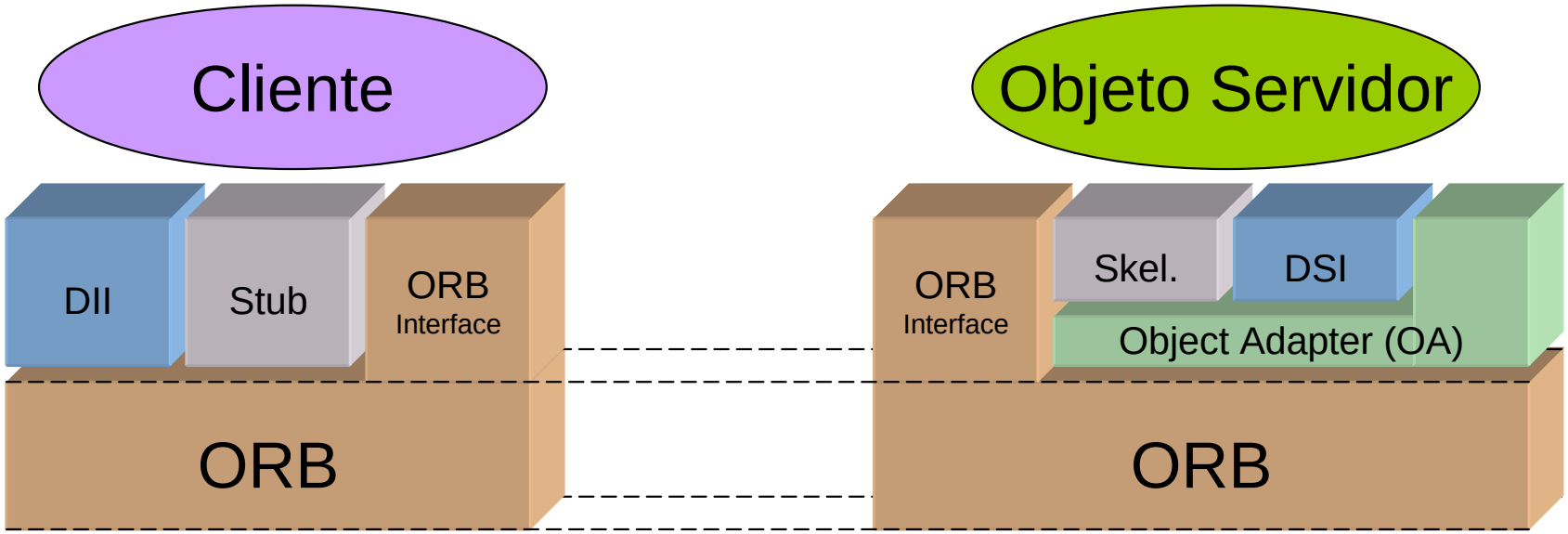
Implementación del Objeto:

- La implementación del objeto es invocada por los métodos definidos en el *Skeleton*. Por lo general se trata de una clase derivada del mismo.

```
class Cuenta_Impl: public Cuenta_Skel
{
  void ingresar(CORBA::Long &cantidad)
  {
     dinero += cantidad;
  }
};
```

Componentes de un ORB

La arquitectura completa de comunicaciones de CORBA es la siguiente:



Componentes de un ORB

Stub:

- Código cliente asociado al objeto remoto con el que se desea interactuar. Simula para el cliente los métodos del objeto remoto, asociando a cada uno de los métodos una serie de funciones que realizan la comunicación con el objeto servidor.

Skeleton:

- Código servidor asociado al objeto. Representa el elemento análogo al stub del cliente. Se encarga de simular la petición remota del cliente como una petición local a la implementación real del objeto.

Componentes de un ORB

DII:

- (Dynamic Invocation Interface)
- Alternativa al uso de stubs estáticos que permite que un cliente solicite peticiones a servidores cuyos interfaces se desconocían en fase de compilación.

DSI:

- (*Dynamic Skeleton Interface*)
- Alternativa dinámica al uso de skeletons estáticos definidos en tiempo de compilación del objeto. Es usado por servidores que durante su ejecución pueden arrancar diferentes objetos que pueden ser desconocidos cuando se compiló el servidor.

Componentes de un ORB

ORB/Interface ORB:

- Elemento encargado de (entre otras) las tareas asociadas a la interconexión entre la computadora cliente y servidor, de forma independiente de las arquitecturas hardware y SSOO.
- Debido a que tanto clientes como servidores pueden requerir de ciertas funcionalidades del ORB, ambos son capaces de acceder a las mismas por medio de un interfaz.

Las dos principales responsabilidades del ORB son:

- Localización de objetos: El cliente desconoce la computadora donde se encuentra el objeto remoto.
- Comunicación entre cliente y servidor: De forma independiente de protocolos de comunicación o características de implementación (lenguaje, sistema operativo, ...)

Componentes de un ORB

Adaptado de Objetos:

- En este elemento se registran todos los objetos que sirven en un determinado nodo. Es el encargado de mantener todas las referencias de los objetos que sirven en una determinada computadora de forma que cuando llega una petición a un método es capaz de redirigírla al código del skeleton adecuado.

Existen dos tipos de Adaptadores de Objetos especificados por OMG:

- BOA: (Basic Object Adapter).
- POA: (Portable Object Adapter).

Componentes de un ORB

Las principales tareas del Adaptador de Objetos son:

- Multiplexar a dos niveles (obnjeto y método) las llamadas.
- Mantiene información (almacenada en el Repositorio de Implementaciones) sobre los objetos servidos, siendo el encargado de activarlos si al llegar una petición no se encontraban en ejecución.
- Permite diferente modos de activación de los objetos:
  - **Persistente**: El estado del objeto se almacena entre varias ejecuciones.
  - **Compartido**: Todos los clientes comparten la instancia de objeto.
  - **No-compartido**: Cada cliente accede a una instancia diferente del objeto.
  - **Por-método**: Cada método solicitado es servido por una instancia de objeto diferente.
- Genera las referencias de los objetos dentro del entorno. Esta referencia es única para todos los objetos.

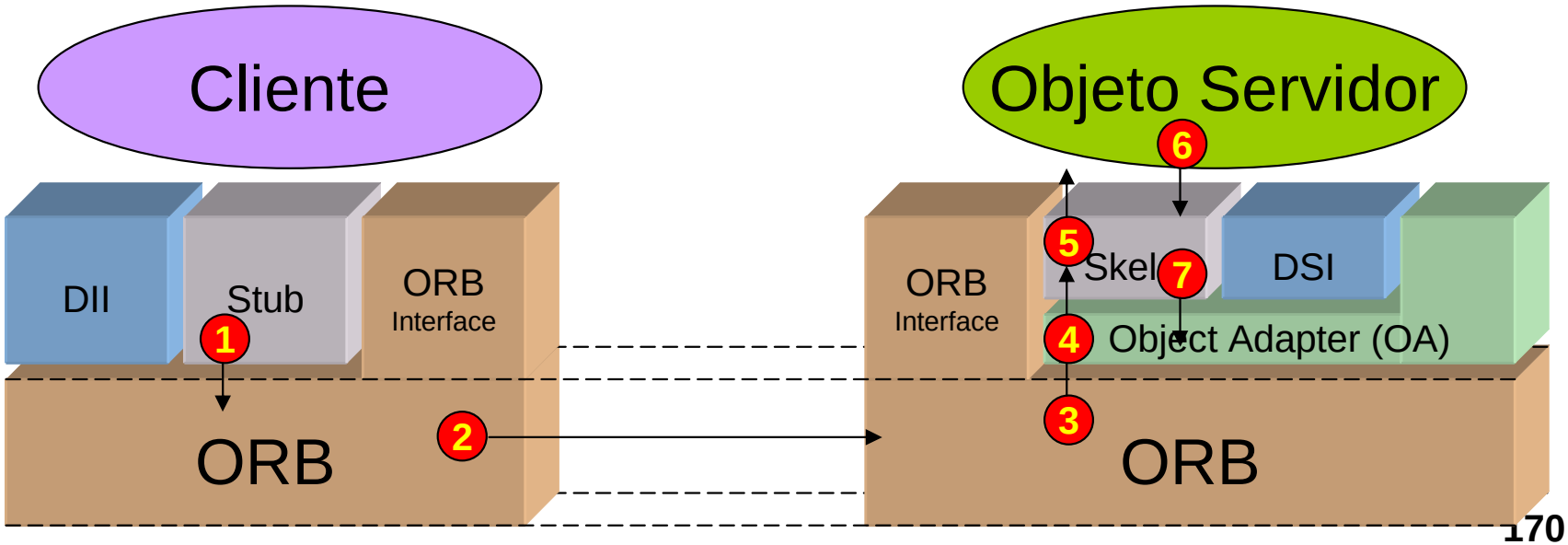
Comunicación vía CORBA

Pasos de una comunicación:

1- El cliente invoca el método asociado en el stub que realiza el proceso de marshalling. (Stub cliente)

2- El stub solicita del ORB la transmisión de la petición hacia el objeto servidor. (ORB cliente)

3- El ORB del servidor toma la petición y la transmite el Adaptador de Objetos asociado, por lo general sólo hay uno. (ORB servidor)



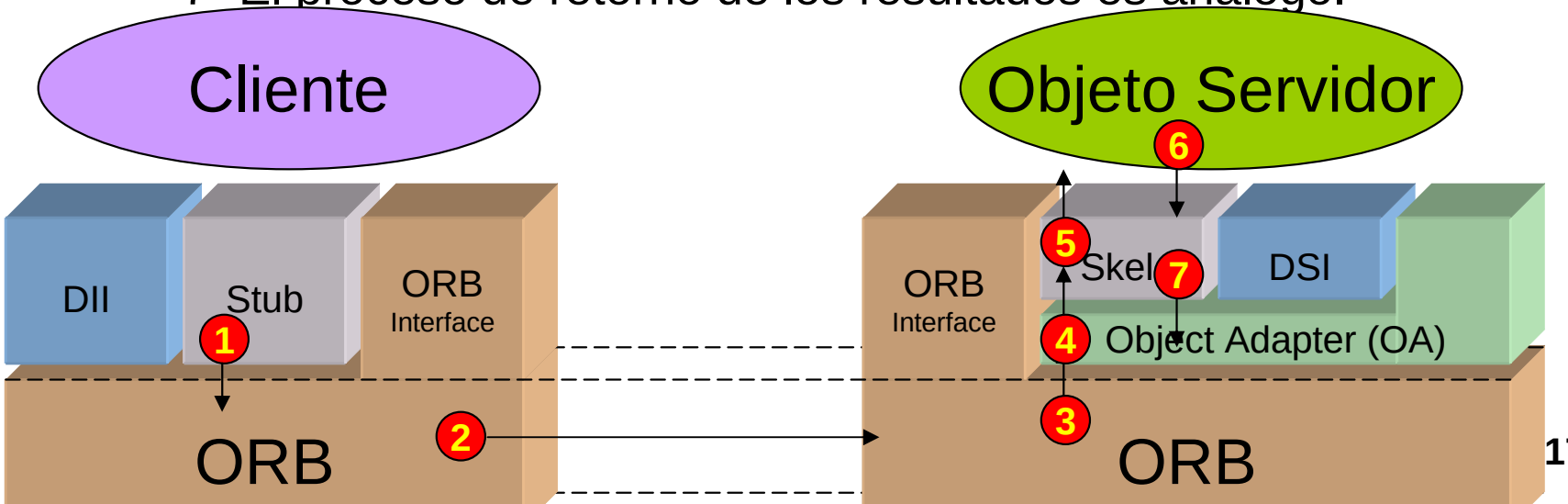
Comunicación vía CORBA

4- El Adaptador de Objetos resuelve cuál es el objeto invocado, y dentro de dicho objeto cuál es el método solicitado (Adaptador de Objetos)

5- El skeleton del servidor realiza el proceso de de-marshalling de los argumentos e invoca a la implementación del objeto. (Skeleton servidor)

6- La implementación del objeto se ejecuta y los resultados y/o parámetros de salida se retornan al skeleton. (Implementación del objeto)

7- El proceso de retorno de los resultados es análogo.



Implementación de un ORB

El ORB representa a nivel lógico el bus de objetos que comparten tanto clientes como servidores. A nivel de práctico puede estar implementado como:

- Residente cliente/servidor: Código que tanto clientes como objetos tiene que enlazar.
- Demonio del sistema: Un servicio del sistema encargado de centralizar las peticiones.
- Interno al sistema: Integrado dentro del SO.
- Librería: Usado cuando tanto clientes como servidores residen dentro del mismo espacio de memoria.

Localización de Objetos

- Los objetos de servicio de una aplicación CORBA se encuentran identificados por medio de una referencia única (Identificador de Objeto).
- Esta referencia es generada al activar un objeto en el Adaptador de Objetos.
- Por medio de esta referencia el ORB es capaz de localizar la computadora y el Adaptador de Objetos donde se encuentra, y éste último es capaz de identificar el objeto concreto dentro del Adaptador.

Localización de Objetos

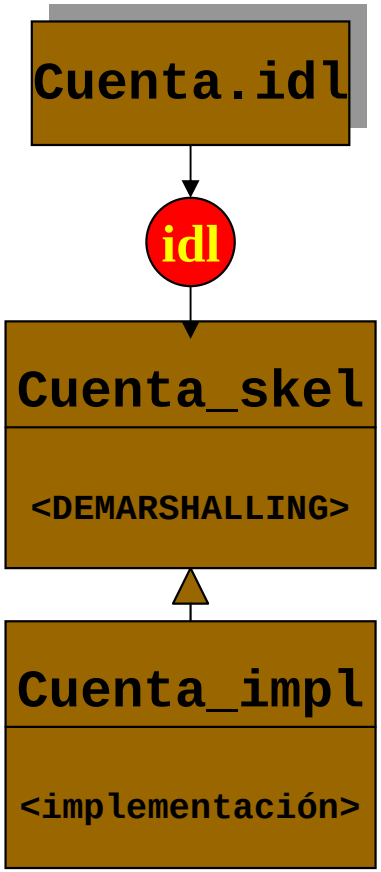
El ORB proporciona mecanismos para transformar a cadena de caracteres y de cadena de caracteres a dicha referencia :

`object_to_string, string_to_object`

Ejemplo:

**IOR:010000000f00000049444c3a4375656e746
13a312e3000000200000000000003000000001
0100001600000007175696e6f2e64617473692e6
6692e75706d2e65730041040c00000424f418a
640965000009f4030100000240000000100000
0010000000100000140000000100000010001**

Implementación del Servidor



Clase generada automáticamente

Implementación del objeto

La implementación del objeto se diseña como una subclase de la clase generada por el compilador (el *skeleton*) de IDL en base a la definición.

Si el lenguaje usado para la implementación no soporta objetos el mecanismo es diferente.

Tareas Típicas de un Servidor

El servidor debe realizar las siguientes tareas:

- Inicializar el ORB (obtiene el interfaz con el ORB).

  ```
  CORBA::ORB_init
  ```

- Obtener la referencia del Adaptador de objetos.

  ```
  orb->BOA_init
  ```

- Crear un un objeto (de la clase Cuenta_impl).

  ```
  new Cuenta_impl()
  ```

- Activar el objeto.

  ```
  boa->impl_is_ready(...)
  ```

- Iniciar el bucle de servicio.

  ```
  orb->run()
  ```

Tareas Típicas de un Cliente

El cliente debe realizar las siguientes tareas:

- Inicializar el ORB (obtiene el interfaz con el ORB).

  `CORBA::ORB_init`

- Obtener la referencia del Adaptador de objetos.

  `orb->BOA_init`

- Obtener la referencia al objeto (desde un *string*).

  `orb->string_to_object(...)`

- Cambiar la clase del objeto obtenido (*down-casting*).

  `Cuenta::_narrow(obj)`

- Realizar las llamadas al objeto.

  `cc->...`

Otros Modos de Activación

Como alternativa al proceso de arrancar cada uno de los objetos de servicio, existe la posibilidad de indicar al Adaptador de Objetos otros modos de activación:

- Esto permite no arrancar la instancia hasta que un cliente la solicite o sea activada explícitamente.

- Esta alternativa requiere un ORB del tipo demonio o interno al sistema.

Otros Modos de Activación

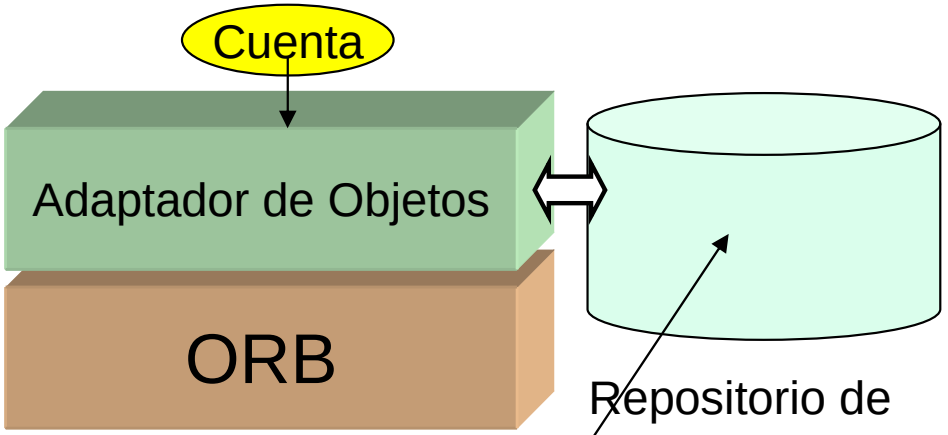
1- En primer lugar es necesario arrancar el demonio.

Para la implementación MICO es:

```
micod -ORBIIOPAddr <direc.>
```

2- Se registra en el repositorio de implementaciones un nuevo objeto, indicando el mandato para ejecutarlo.

```
imr create <nombre> <modo>
    <programa>
```



Nombre	Estado	Referencia
CuentaCorriente	active	IOR:0f.....

Servicios CORBA

Conjunto de objetos o grupos de objetos, que proporcionan una serie de funcionalidades elementales. Estos objetos esta definidos de forma estándar (interfaces IDL concretos).

- Sus especificaciones se encuentran recogidas en los documentos COSS (Common Object Services Specifications).
- Los servicios definidos son:

- Servicio de Nombres
- Servicio de Eventos
- Servicio de Ciclo de Vida
- Servicio de Objetos Persistentes
- Servicio de Transacciones
- Servicio de Control de Concurrencia
- Servicio de Relación
- Servicio de Externalización
- Servicio de Consulta
- Servicio de Licencias
- Servicio de Propiedad
- Servicio de Tiempo
- Servicio de Seguridad
- Servicio de Negociación
- Servicio de Colección de Objetos

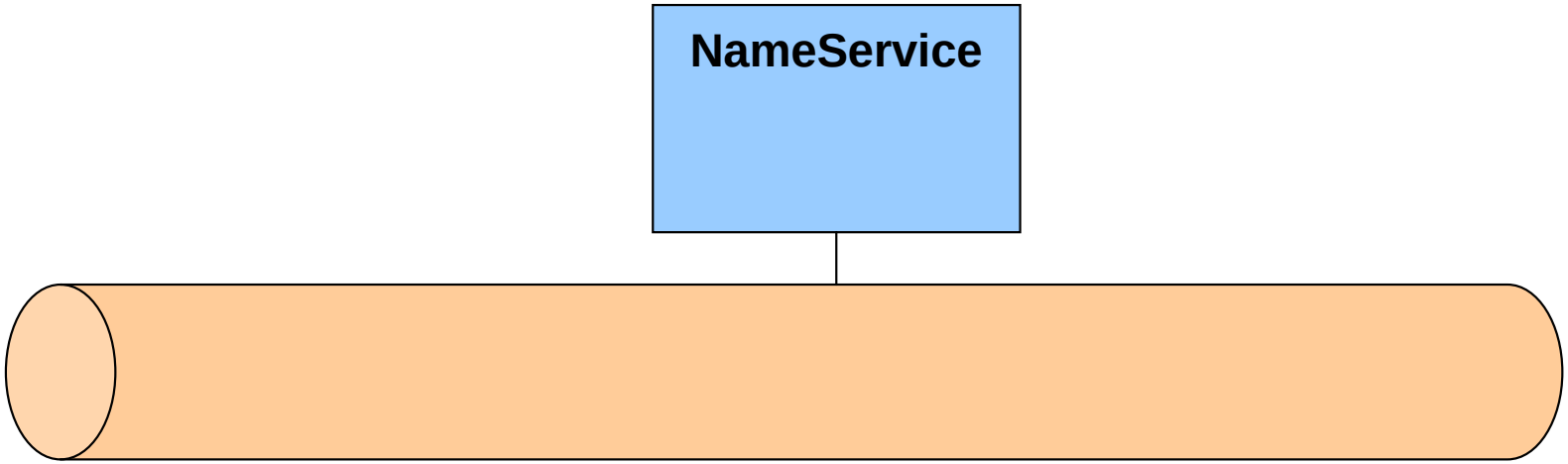
Uso del Servicio de Nombres

Permite asociar un nombre a una referencia de objeto. De esta forma los objetos al activarse pueden darse de alta en el servidor, permitiendo que otros objetos los obtengan su referencia en base a dicho nombre.

Los nombres de los objetos se encuentran organizados en una jerarquía de *contextos de nombre*.

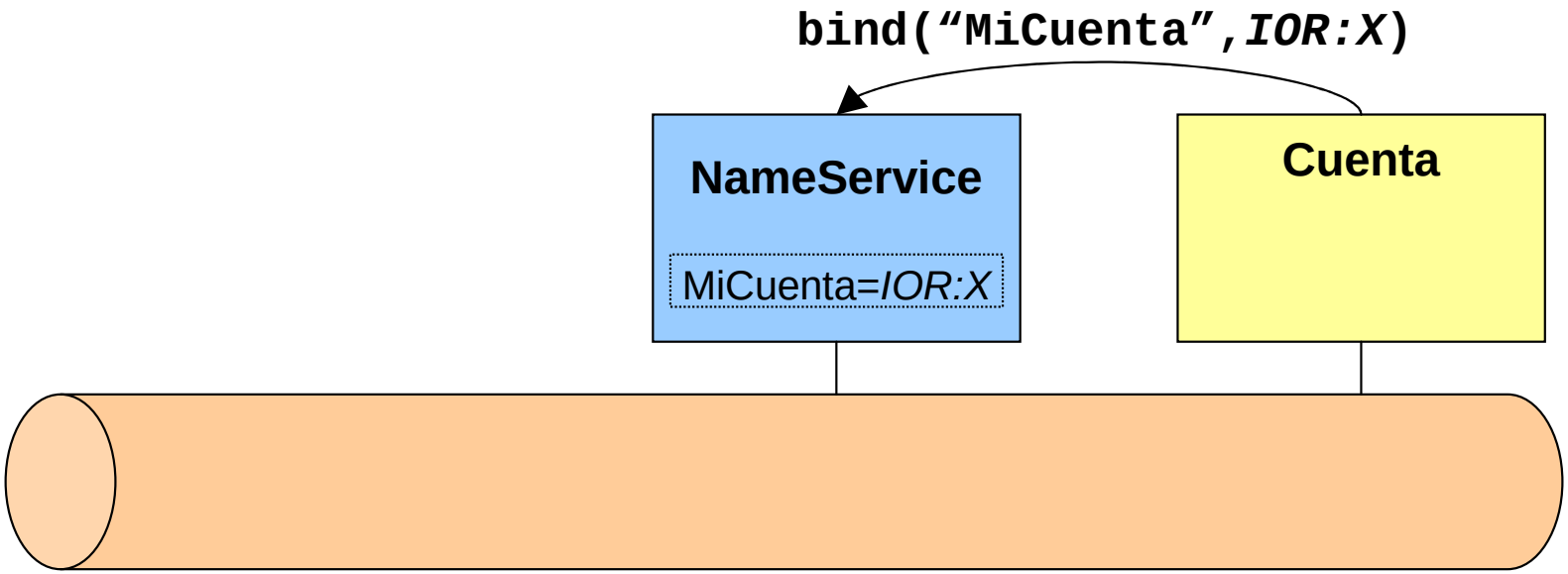
Servicio de Nombres

El Servidos de Nombres, al igual que todo objeto del sistema se encuentra previamente activo.



Servicio de Nombres

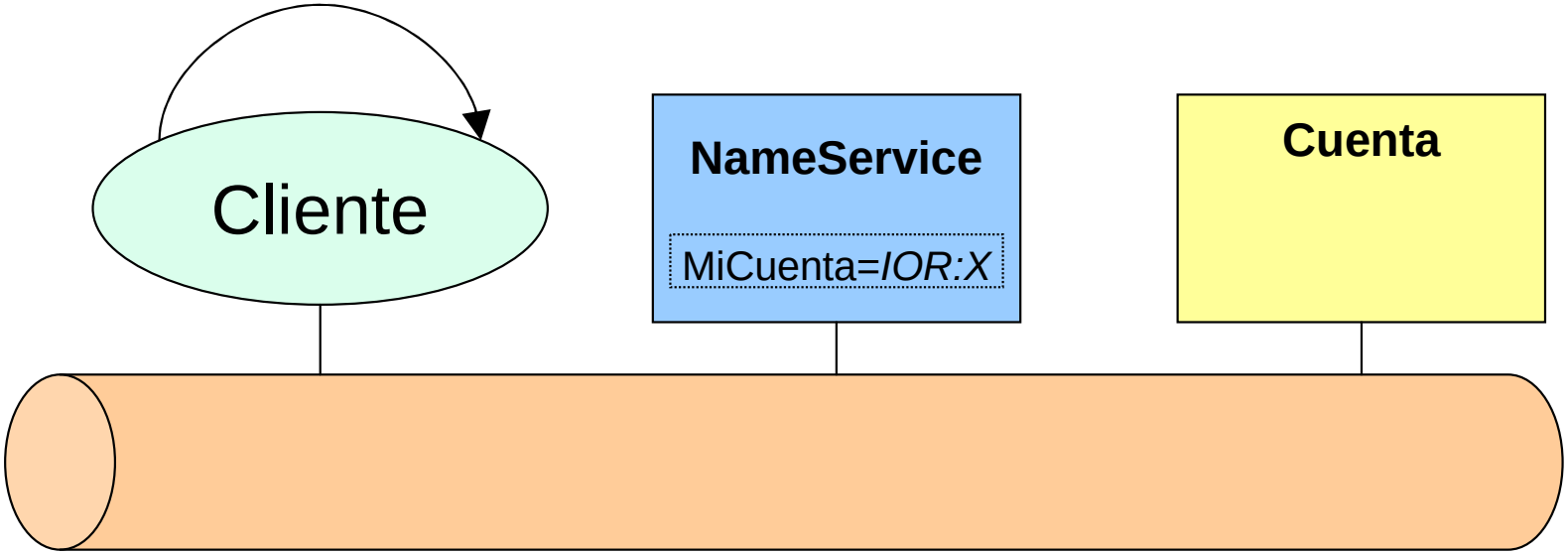
Un nuevo objeto se arranca y se registra en el servidor de nombres:



Servicio de Nombres

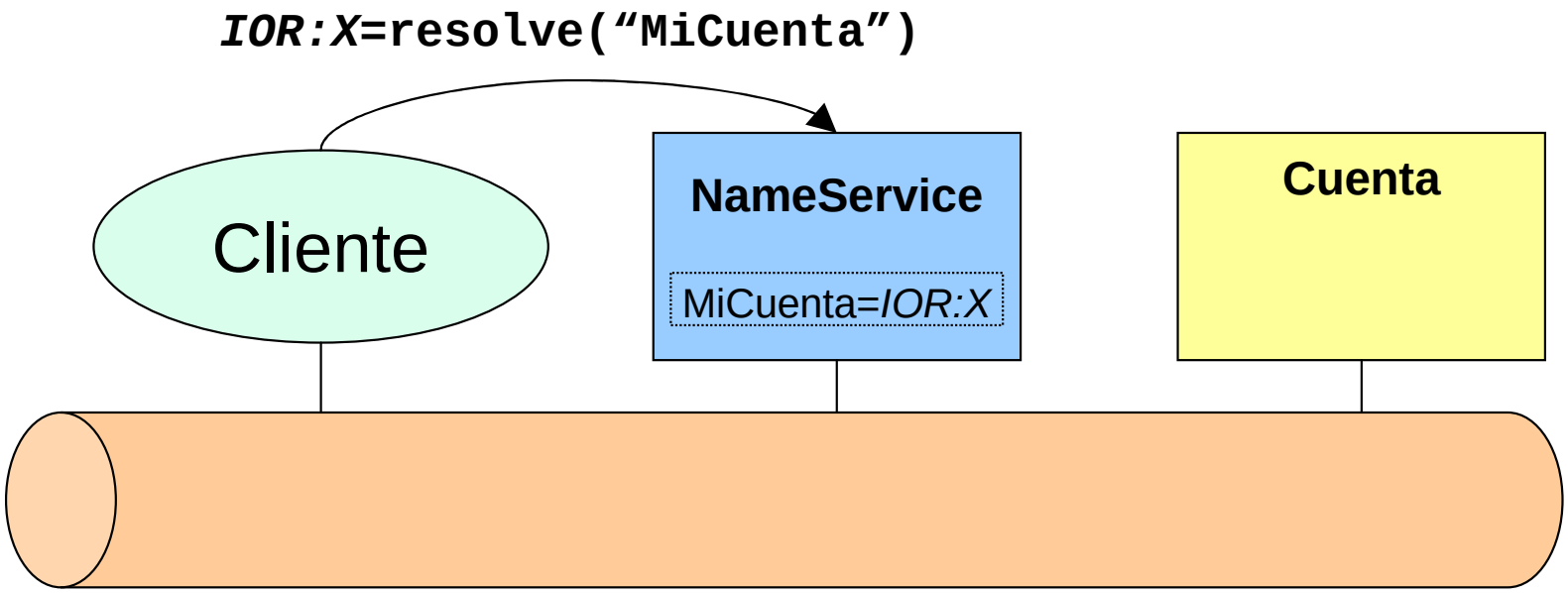
Un cliente localiza al servidor de nombres. Suele existir una función interna del ORB para localizar al servidor de nombres (**`resolve_initial_references`**) .

```
IOR:NS=resolve_initial_references("NameService")
```



Servicio de Nombres

El cliente le pide al servidor de nombres que resuelva el nombre. Así obtiene la referencia al objeto buscado.



Servicio de Negociación

Este servicio también permite obtener referencias a objetos usando otra información:

- Los objetos se exportan en el servidor con una serie de características del servicio que proporcionan.
- Los clientes consultan con el servidor cuáles son los objetos ofertados con una serie de características.

Un cliente que buscase un servicio de impresión podría construir una consulta del tipo:

```
Service=Printer AND
PrinterType=HP AND
OS!=WinNT
```

A lo cual el servidor le indicará los objetos que existen con dichas características.

Comparativa

CORBA vs DCOM

- CORBA es un estándar abierto y no propietario.
- CORBA proporciona soporte para diversos SO.
- CORBA es más completo y flexible.
- CORBA da una salida a los *legacy systems*

- DCOM esta integrado con la tecnología *MicroSoft*.
- DCOM ha tenido una fuerte penetración en el mercado.

Comparativa

CORBA vs RMI

- CORBA permite una mayor heterogeneidad en el desarrollo de aplicaciones (RMI sólo se puede desarrollar con Java).
- CORBA ademas de las funcionalidades de comunicación, proporciona servicios.
- RMI funciona sobre CORBA (IIOP).

- RMI es mucho más sencillo y cómodo de usar.
- RMI permite el paso de objetos por valor y por referencia.

Ejemplo

El ejemplo esta compuesto por cinco archivos:

- Definición IDL del objeto. (**`Cuenta.idl`**)
- Cabecera de la implementación del objeto. (**`Cuenta_impl.h++`**)
- Implementación del objeto. (**`Cuenta_impl.c++`**)
- Código servidor. (**`servidor.c++`**)
- Código cliente. (**`cliente.c++`**)

Comunicación en Sistemas Distribuidos

Grupos de comunicación

Hay tres tipos de grupos de comunicación:

- Uno a muchos
- Muchos a uno
- Muchos a muchos

Comunicación en Sistemas Distribuidos

Uno a muchos

Este esquema es conocido como *comunicación multicast*.

En este caso los procesos receptores de los mensajes constituyen un grupo, que a su vez pueden ser de dos tipos:

- Grupos cerrados
- Grupos abiertos

Comunicación en Sistemas Distribuidos

Grupos cerrados

Solo los miembros del grupo pueden enviar mensajes al grupo.

Un miembro externo solo puede enviar mensajes a un proceso individual y no al grupo.

Grupos abiertos

Cualquier proceso en el sistema puede enviar un mensaje al grupo como tal.

Comunicación en Sistemas Distribuidos

Un sistema de pasaje de mensajes con la facili-dad de grupo de comunicación provee la flexi-bilidad de crear y borrar grupos dinámicamente y permitir a un proceso agregarse o dejar un grupo.

Un mecanismo para realizar todo esto es un *servidor de grupos*.

Esta solución sufre de pobre confiabilidad y po-bre escalabilidad.

Comunicación en Sistemas Distribuidos

Muchos a uno

Emisores múltiples envían mensajes a un único receptor.

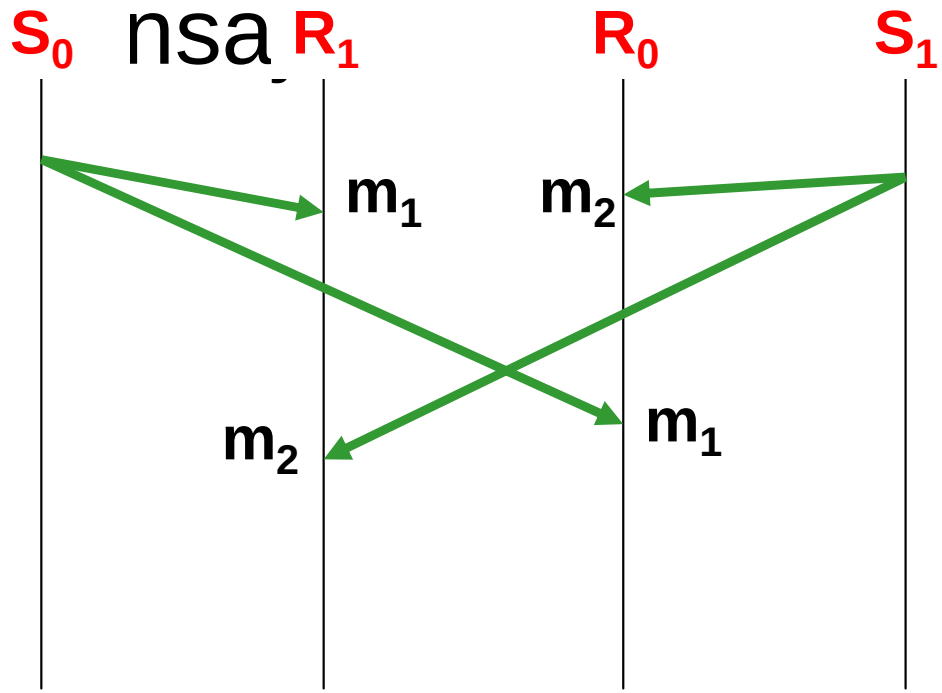
Hay un *no determinismo.*

Muchos a muchos

Múltiples emisores envían mensajes a múltiples receptores.

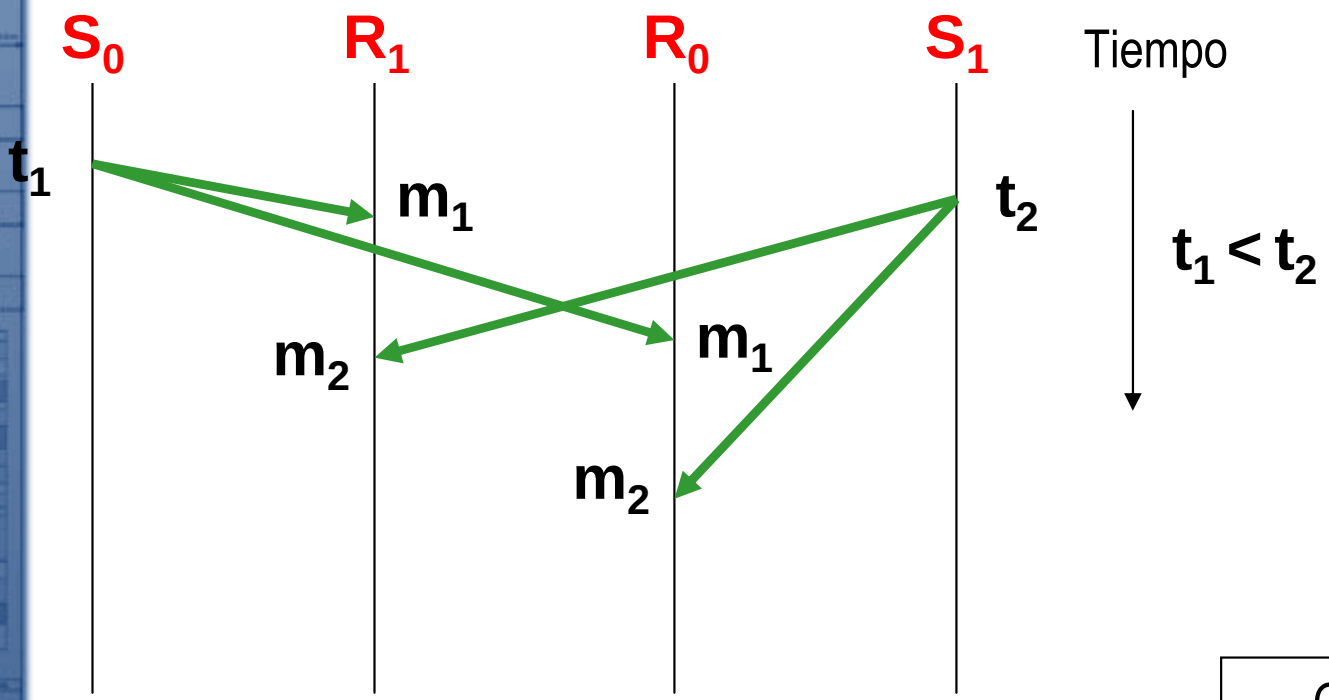
Comunicación en Sistemas Distribuidos

La cuestión mas importante en este esquema es el ordenamiento de los nsa



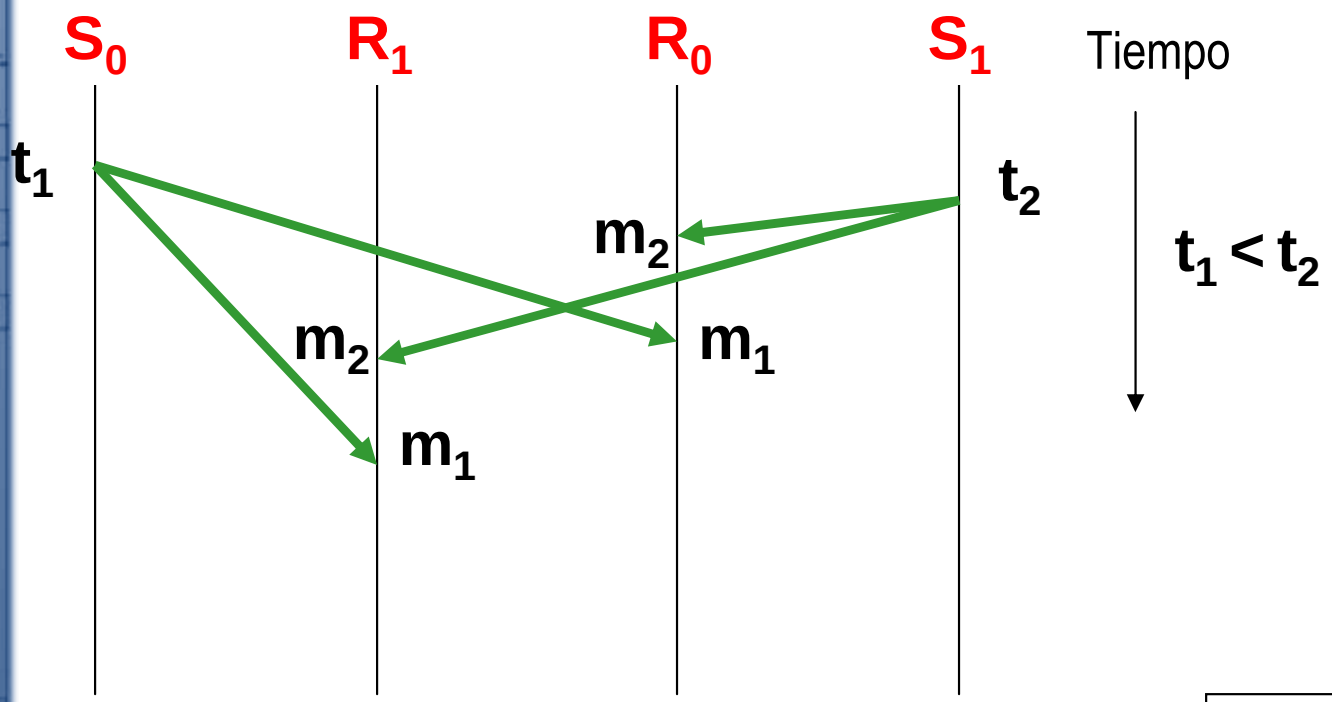
No hay restricción de orden

Comunicación en Sistemas Distribuidos



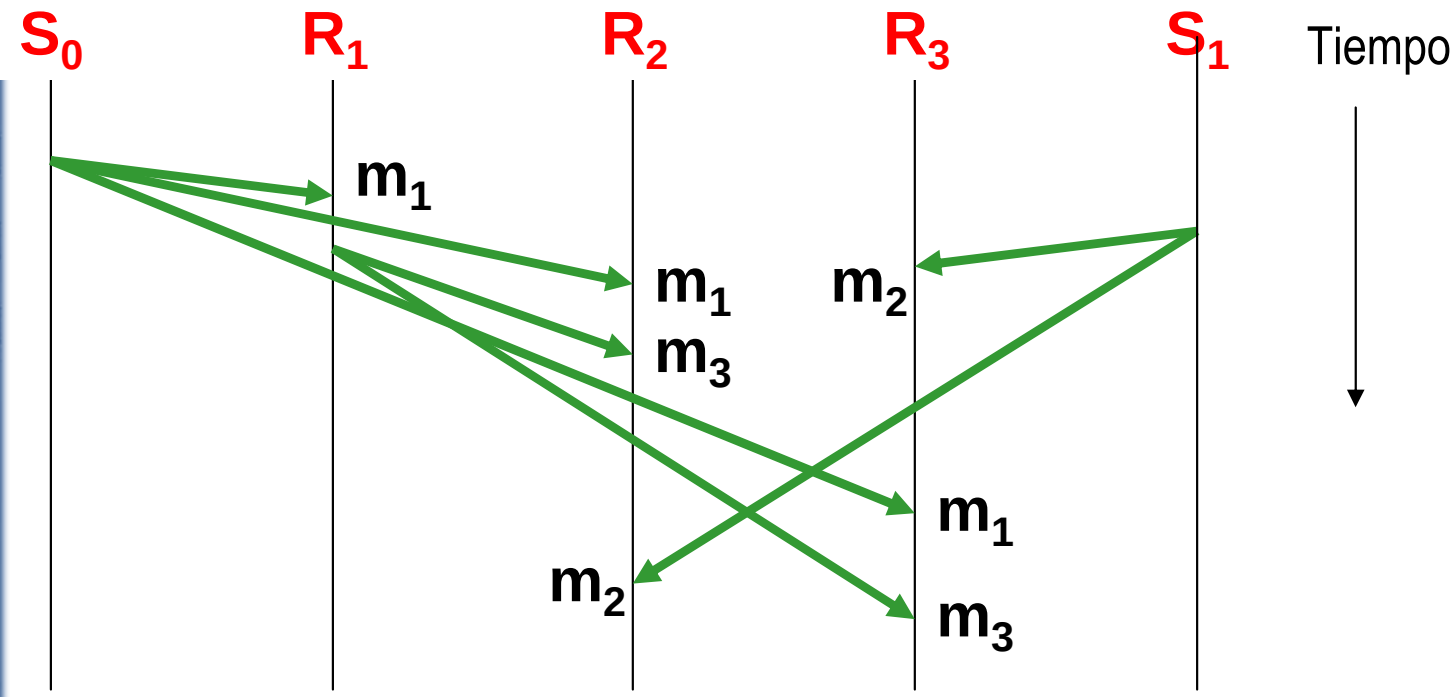
Ordenamiento absoluto

Comunicación en Sistemas Distribuidos



Ordenamiento consistente

Comunicación en Sistemas Distribuidos



Orden causal

Coming Next