2

Comunicación de Procesos en Sistemas Distribuidos

Contenidos del tema

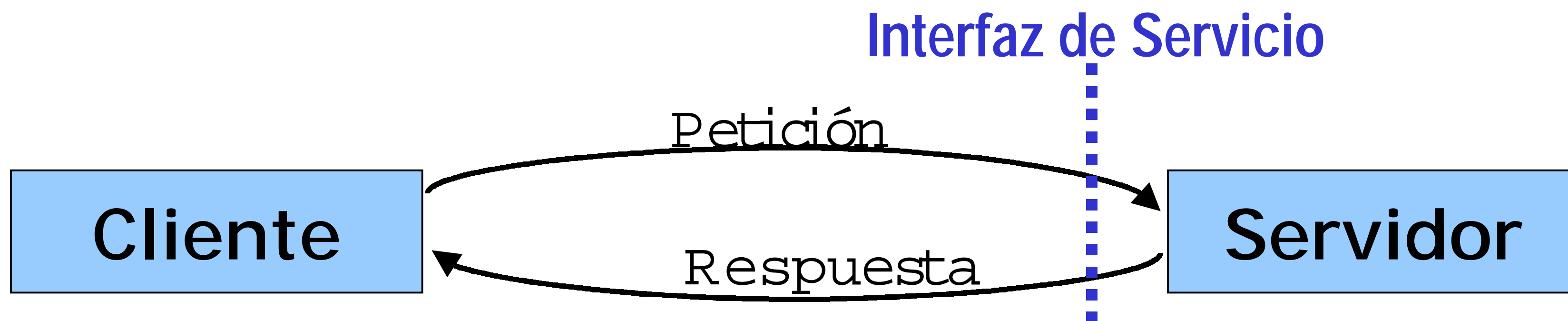
- Arquitectura de comunicaciones
- Características de la comunicación
- Comunicación en grupo
- Paso de mensajes
  - *Sockets*
- Llamadas a procedimientos remotos (RPC)
  - RPC de Sun
- Entornos distribuidos de objetos
  - Java RMI
  - CORBA
- Servicios Web: RPC basada en XML para la Web

Introducción

- Sistema de comunicación: Espina dorsal del SD.
- Modelos de comunicación entre procesos:
  - Memoria compartida: no factible, en principio (DSM), en SD
  - Paso de mensajes.
- Nivel de abstracción en comunicación con paso de mensajes:
  - Paso de mensajes puro.
  - Llamadas a procedimientos remotos.
  - Modelos de objetos distribuidos.
- Arquitectura de comunicaciones
  - Modelo cliente/servidor
    - con proxy o caché
    - múltiples capas
    - código móvil
  - Modelo *peer-to-peer*

Modelo cliente/servidor

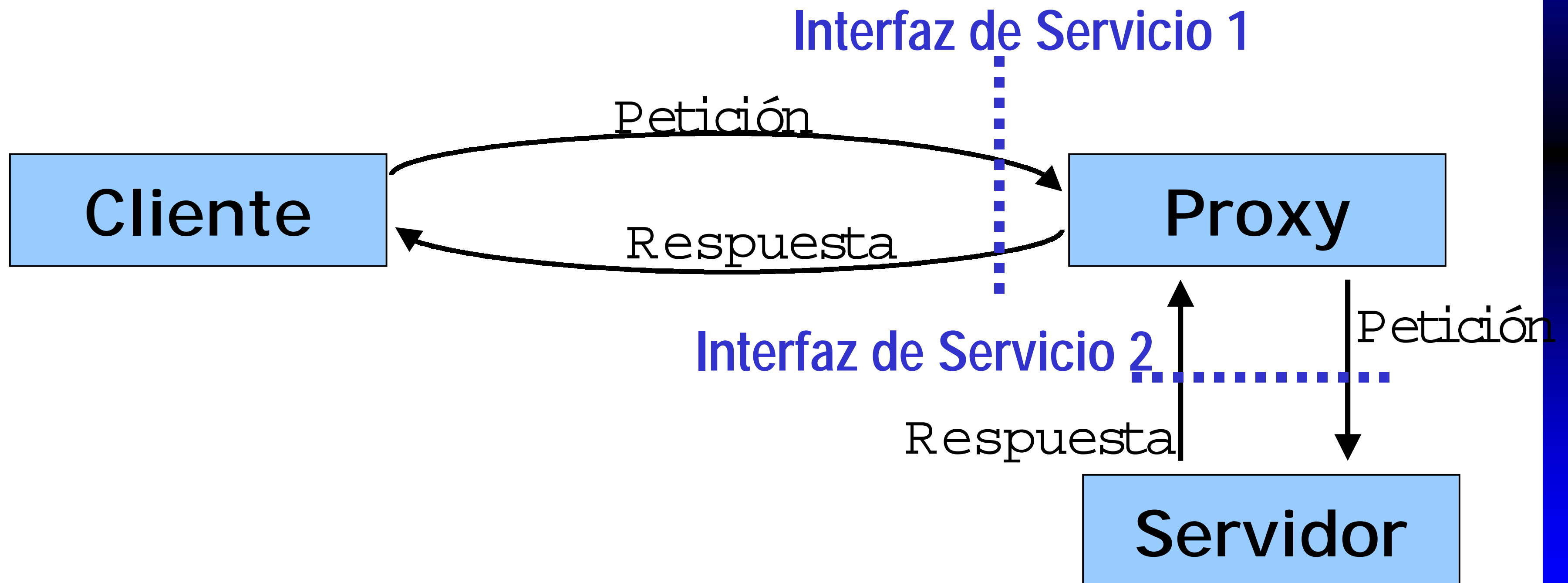
- Dos roles diferentes en la interacción
  - Cliente: Solicita servicio. Petición: Operación + Datos
  - Servidor: Proporciona servicio. Respuesta: Resultado



Fernando Pérez Costoya
José María Peña Sánchez

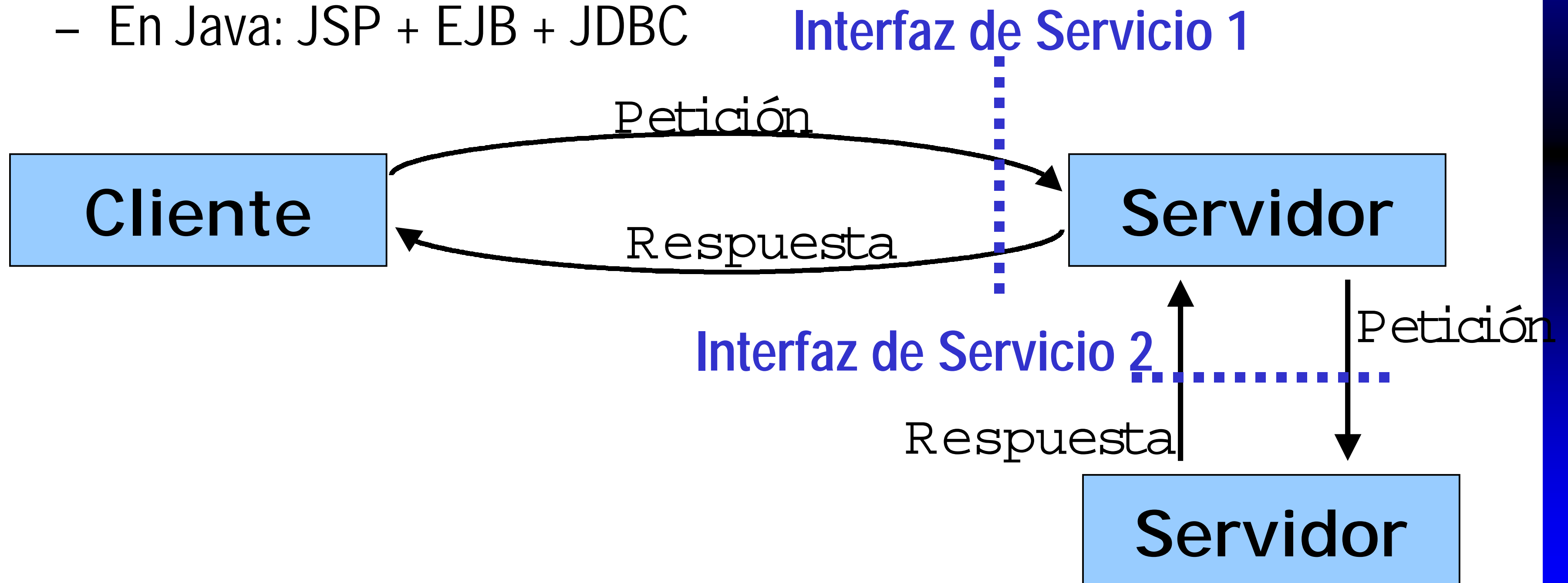
Modelo con proxy o caché

- Tres roles diferentes en la interacción
  - Cliente: Solicita servicio.
  - Servidor: Proporciona servicio.
  - Proxy: Intermediario (si tiene *memoria* se denomina caché)



Modelo multicapa

- Servidor puede ser cliente de otro servidor
- Típico en aplicaciones web:
  - Presentación + Lógica de negocio + Acceso a datos
  - En Microsoft: ASP + COM + ADO
  - En Java: JSP + EJB + JDBC



Código móvil

Modelo cliente/servidor alternativo:

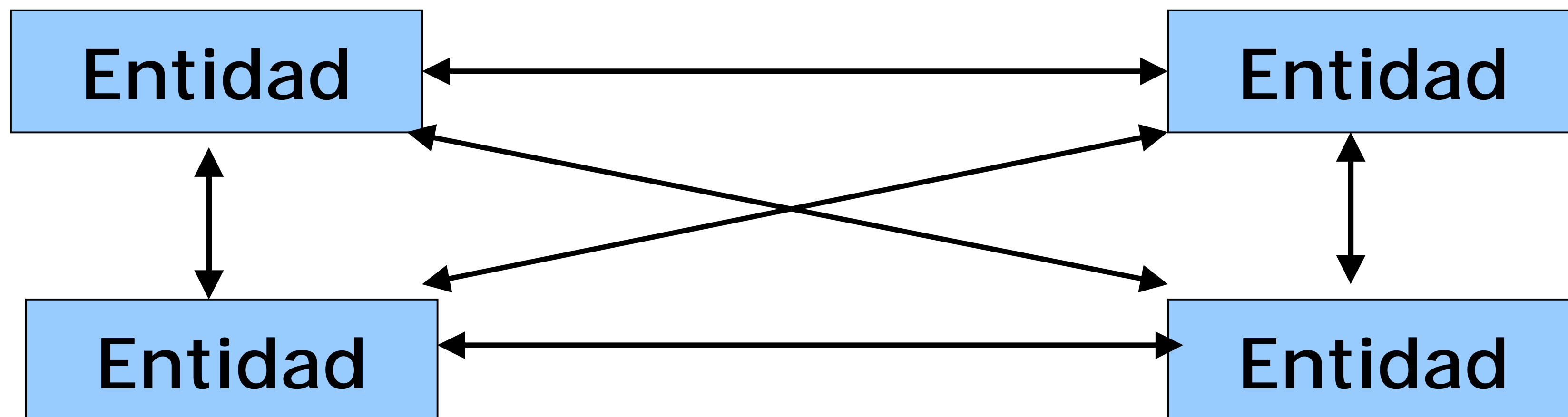
- "Viaja" el código en vez de los datos
- Requiere máquinas homogéneas o "Máquinas virtuales": (ej. *applet)*
- Problemas de seguridad

Agentes móviles

- Programa que viaja por el SD realizando una tarea
- En cada nodo procesa datos locales
- Se transfiere programa en vez de datos
- Ejemplo: *Data mining* en base de datos distribuida

Modelo *peer-to-peer*

- Un único rol: Entidad
- Protocolo de diálogo
  - Entidades se coordinan entre sí
  - Ejemplo: simulación paralela, al final de cada etapa las entidades se sincronizan e intercambian información



Características de la comunicación

Alternativas de diseño:

- Modo de operación bloqueante o no bloqueante
- Fiabilidad
- Comunicación síncrona vs. asíncrona
- Direccionamiento
- Comunicación punto a punto vs. comunicación en grupo.

Fernando Pérez Costoya
José María Peña Sánchez

Bloqueantes vs. no bloqueantes

Envío:

- Envío no bloqueante: El emisor almacena el dato en un buffer del núcleo (que se encarga de su transmisión) y reanuda su ejecución.
- Envío bloqueante: El emisor se bloquea hasta que ha sido enviado correctamente al destino.

Recepción:

- Recepción no bloqueante: Si hay un dato disponible el receptor lo lee, en otro caso indica que no había mensaje.
- Recepción bloqueante: Si no hay un dato disponible el receptor se bloquea.

Fiabilidad

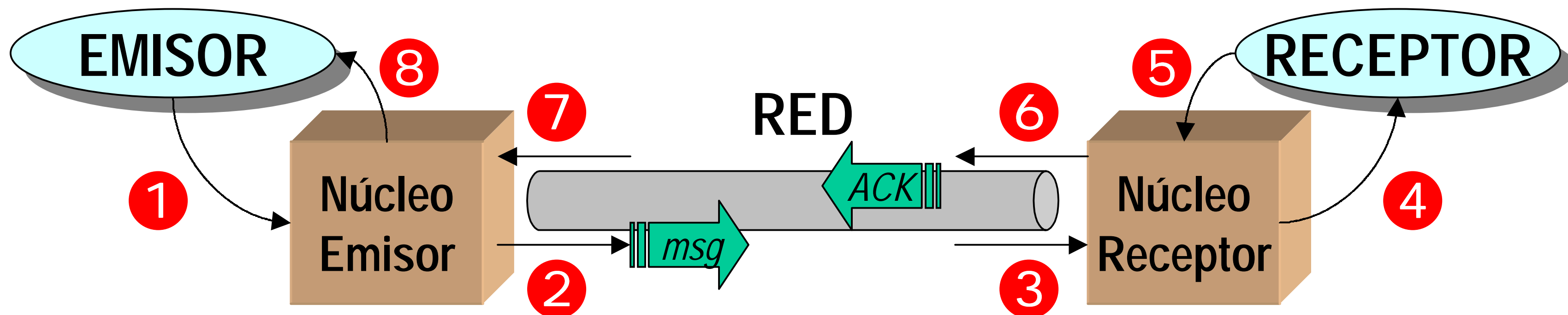
Aspectos relacionados con la fiabilidad de la comunicación:

- Garantía de que el mensaje ha sido recibido en nodo(s) destino(s)
- Mantenimiento del orden en la entrega de los mensajes
- Control de flujo para evitar "inundar" al nodo receptor
- Fragmentación de los mensajes para eliminar limitaciones en tamaño máximo de mensajes

Si el sistema de comunicación no garantiza algunos de estos aspectos, debe hacerlo la aplicación

Fernando Pérez Costoya
José María Peña Sánchez

Comunicación síncrona vs. asíncrona



Envío no bloqueante (comunicación asíncrona): [1:8] El emisor continúa al pasar el mensaje al núcleo.

Envío bloqueante no fiable (comunicación asíncrona): [1:2:7:8] El emisor espera a que el núcleo transmita por red el mensaje.

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

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

Petición-Respuesta (cliente/servidor) : [1:2:3:4:<servicio>:5:6:7:8]: Emisor espera a que el receptor procese la operación. Respuesta puede servir de ACK.

Fernando Pérez Costoya
José María Peña Sánchez

Direccionamiento

Cómo se especifica(n) receptor(es) de un mensaje:

- 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

Destino de mensaje es un grupo de procesos: multidifusión

Posibles usos en los sistemas distribuidos:

- Uso de datos replicados: actualizaciones múltiples.
- Uso de servicios replicados.
- Operaciones colectivas en cálculo paralelo.

Implementación depende de si red proporciona *multicast*:

- Si no, se implementa enviando N mensajes

Un proceso puede pertenecer a varios grupos

Existe una dirección de grupo

El grupo suele tener carácter dinámico

- Se pueden incorporar y retirar procesos del grupo
- Gestión de pertenencia debe coordinarse con la comunicación

Aspectos de diseño de com. en grupo

- Modelos de grupos:
  - Grupo abierto.
    - Proceso externo puede mandar mensaje al grupo
    - Suele usarse para datos o servicios replicados
  - Grupo cerrado.
    - Sólo procesos del grupo pueden mandar mensajes.
    - Suele usarse en procesamiento paralelo (modelo *peer-to-peer*)
- Atomicidad
  - O reciben todos los procesos el mensaje o ninguno
- Orden de recepción de los mensajes

Orden de recepción de los mensajes

Tres alternativas:

- **Orden FIFO**: Los mensajes de una misma fuente llegan a cada receptor en el orden que son enviados.
  - No hay ninguna garantía sobre mensajes de distintos emisores
- **Ordenación causal**: Si entre los mensajes enviados por dos emisores existe una posible relación “causa-efecto”, todos los procesos del grupo reciben primero el mensaje “causa” y después el mensaje “efecto”.
  - Si no hay relación, no se garantiza ningún orden de entrega
  - Concepto de “causalidad” se estudia en capítulo “Sincronización”
- **Ordenación total**: Todos los mensajes (de varias fuentes) enviados a un grupo son recibidos en el mismo orden por todos los elementos.

Paso de mensajes

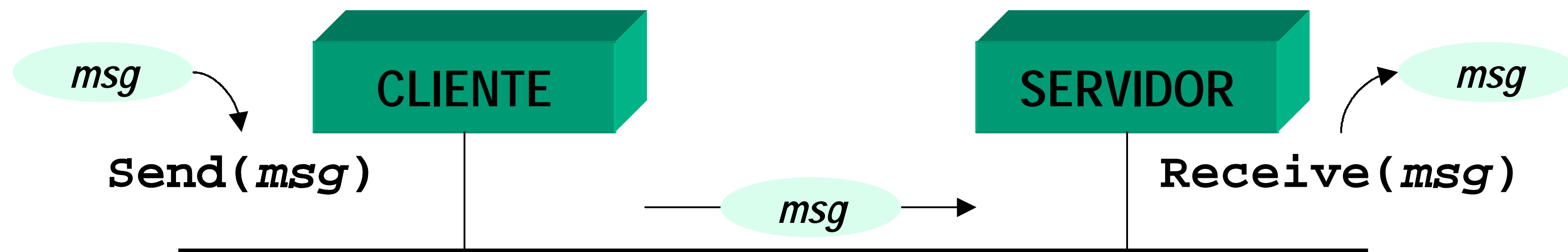
- Sockets

Paso de mensajes

- Primitivas de comunicación hipotéticas:
  - Envío: **`send(`*`destino,mensaje`*`)`**
  - Recepción: **`receive(`*`&origen,&mensaje`*`)`**
- Mecanismo puede estar orientado a conexión o no:
  - Con conexión: mayor fiabilidad
    - orden de mensajes, control de flujo, fragmentación automática, ...
  - Sin conexión: menor sobrecarga inicial
- Modo de comunicación típico:
  - **`Send`** no bloqueante y **`Receive`** bloqueante
  - Comunicación asíncrona
- Las aplicaciones definen el protocolo de comunicación:
  - En arquitectura cliente/servidor: Petición-respuesta

Cliente/servidor con paso de mensajes

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




```
Mensaje petic,resp;

petic.cod_op=OPER_1;
petic.param1=<...>;
...
petic.paramN=<...>;
send(servidor,petic);
receive(NULL,&resp);
...
```

```
Mensaje petic,resp;
receive(&cliente,&petic);
switch(petic.cod_op) {
  case OPER_1: Realiza
     petición con paráms.
     y genera resultados
     resp.resul=<...>;
  case OPER_2: ...
}
send(cliente,resp);
```


Fernando Pérez Costoya
José María Peña Sánchez

Mensajes de texto

Los mensajes pueden ser de texto.

Estructura del Mensaje:

- Cadenas de caracteres.
- Por ejemplo HTTP:

```
"GET //www.fi.upm.es HTTP/1.1"
```

Envío del Mensaje:

```
send(destino, "GET //www.fi.upm.es HTTP/1.1");
```

Mensajes binarios

Los mensajes pueden tener formato binario

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(destino, confirm);
```

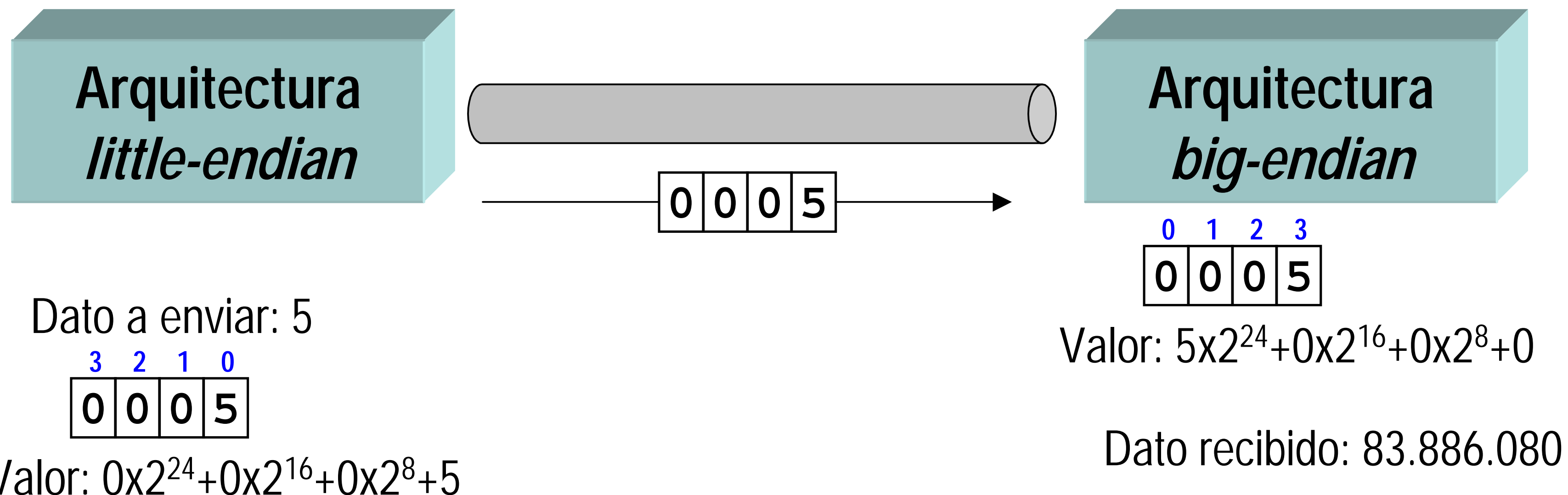

Fernando Pérez Costoya
José María Peña Sánchez

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 EBCDIC.



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.

Sockets

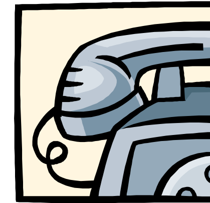
Sujetos a proceso de estandarización dentro de POSIX (POSIX 1003.1g).

Actualmente:

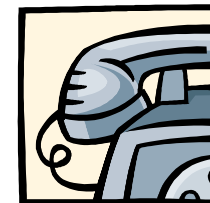
- Disponibles en casi todos los sistemas UNIX.
- En prácticamente todos los sistemas operativos:
  - WinSock: API de sockets de Windows.
- En Java como clase nativa.

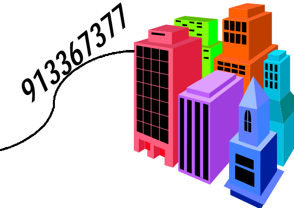
Conceptos básicos sobre *sockets*

- 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.

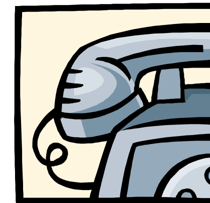


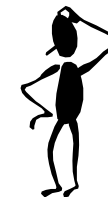
1.- Creación del socket





2.- Asignación de dirección





3.- Aceptación de conexión

Fernando Pérez Costoya
José María Peña Sánchez

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

Usuarios manejan direcciones en forma de texto:
- decimal-punto: 138.100.8.100
- dominio-punto: laurel.datsi.fi.upm.es

- 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*.

Fernando Pérez Costoya
José María Peña Sánchez

Creación de un socket

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

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

- Devuelve un descriptor de fichero (igual que un **`open`** de fichero).
- 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.

Fernando Pérez Costoya
José María Peña Sánchez

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 la su primera utilización (**`connect`** o **`sendto`**).

Fernando Pérez Costoya
José María Peña Sánchez

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.

Fernando Pérez Costoya
José María Peña Sánchez

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*.

Fernando Pérez Costoya
José María Peña Sánchez

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`**): Parámetor 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 nº de bytes enviados.

Recepción:

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

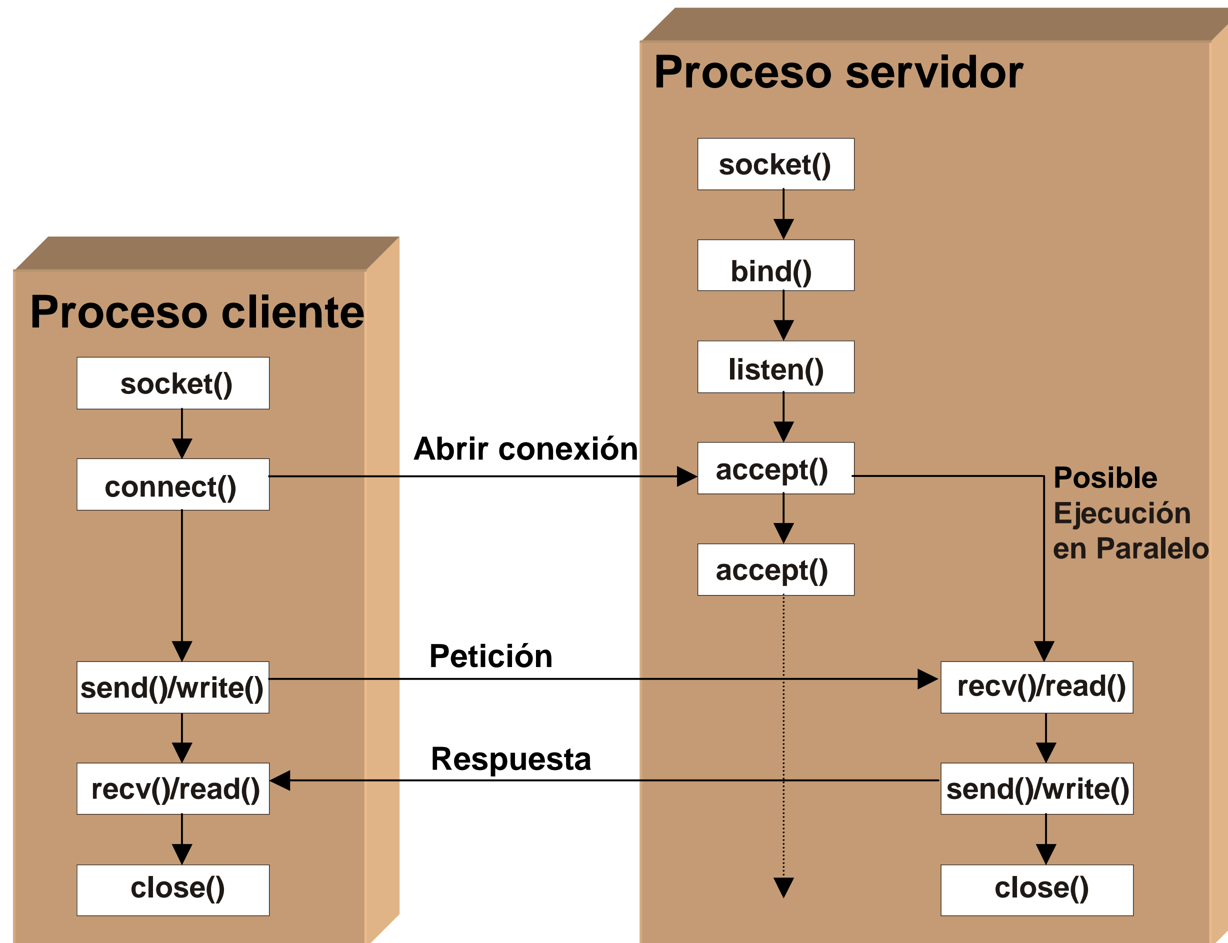
```
int recv(int s,char *mem,int tam,int flags)
```

- Devuelve el nº de bytes recibidos.

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

Fernando Pérez Costoya
José María Peña Sánchez

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:

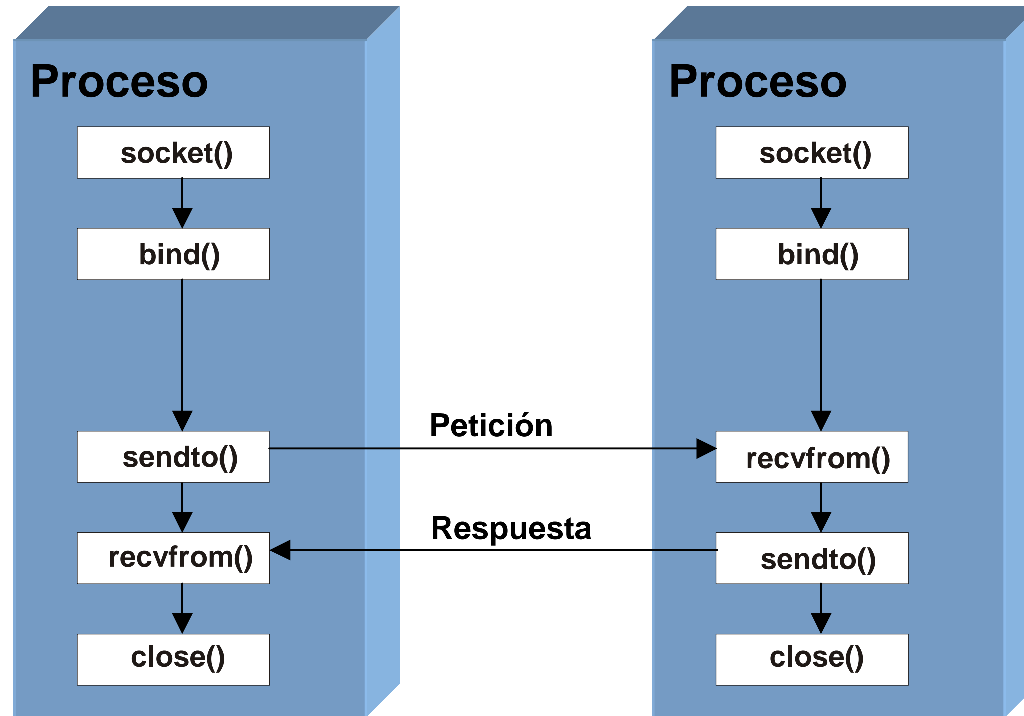
```
int recvfrom(int s,char *mem,int tam,
             int flags,struct sockaddr *dir,
             int *tam)
```

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

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

Fernando Pérez Costoya
José María Peña Sánchez

Escenario de uso de sockets datagrama



Fernando Pérez Costoya
José María Peña Sánchez

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

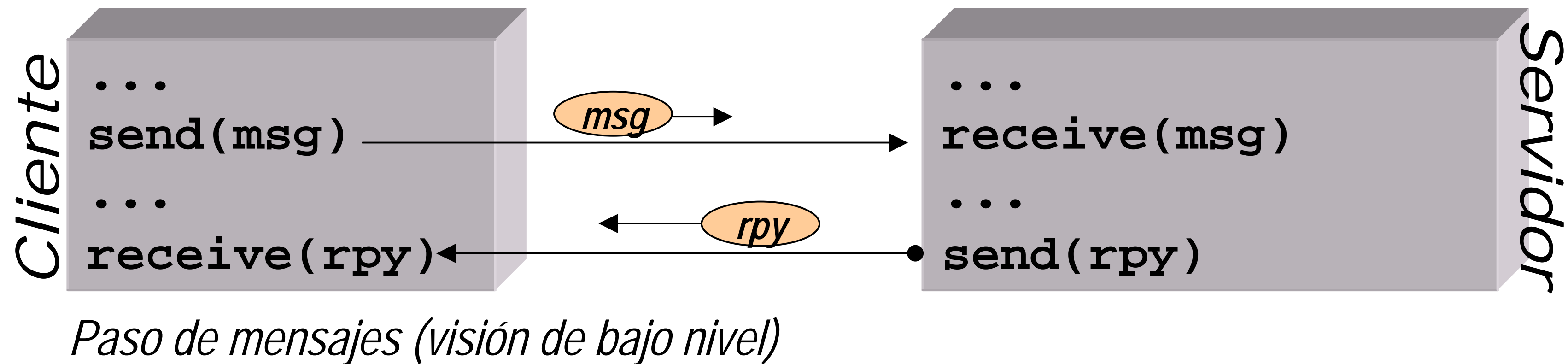
Fernando Pérez Costoya
José María Peña Sánchez

Llamadas a Procedimientos Remotos

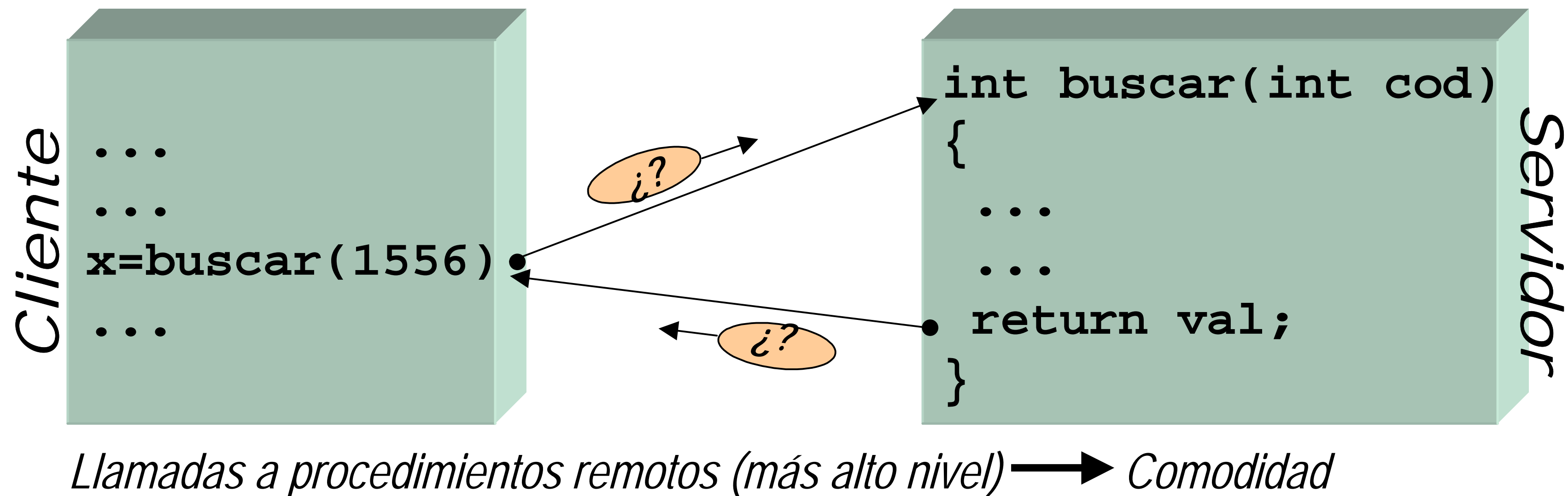
<RPC>

- RPC de Sun

Llamadas a Procedimientos Remotos



Paso de mensajes (visión de bajo nivel)



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

Fernando Pérez Costoya
José María Peña Sánchez

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).

Fernando Pérez Costoya
José María Peña Sánchez

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.

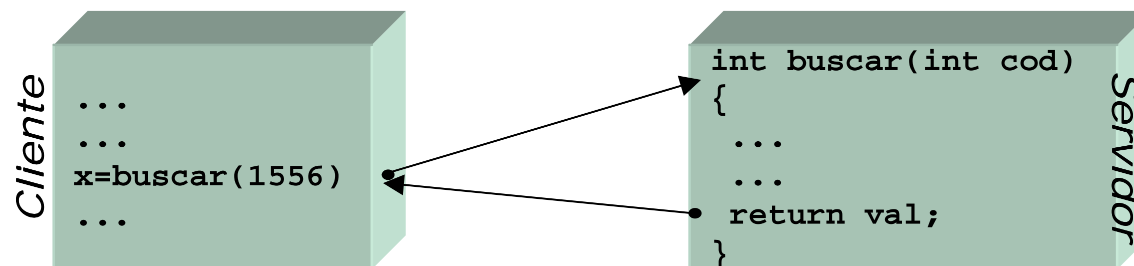
Servidor:

- 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.

Objetivo: acercar la semántica de las llamadas a procedimiento convencional a un entorno distribuido (transparencia).

Elementos Necesarios

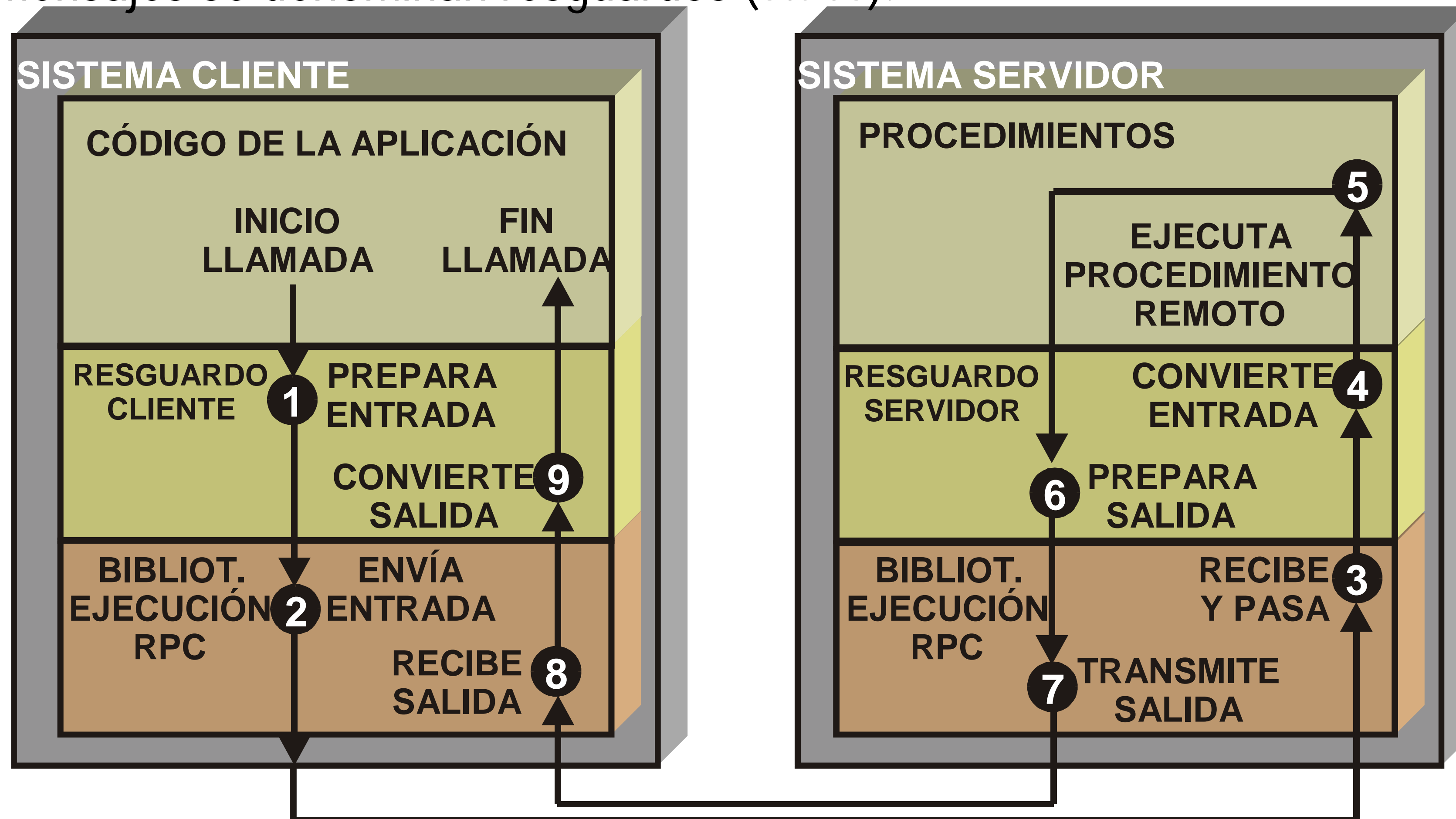
- Código cliente.
- Código del servidor.
- Formato de representación.
- Definición del interfaz.
- Localización del servidor.
- Semánticas de fallo.



Fernando Pérez Costoya
José María Peña Sánchez

Código Cliente/Código Servidor

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



Fernando Pérez Costoya
José María Peña Sánchez

Resguardos (*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 resguardos 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.
- Pasos de parámetros (entrada/salida):
  - Problemas con los punteros: Una dirección sólo tiene sentido en un espacio de direcciones.

Fernando Pérez Costoya
José María Peña Sánchez

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.

Fernando Pérez Costoya
José María Peña Sánchez

Enlazador Dinámico

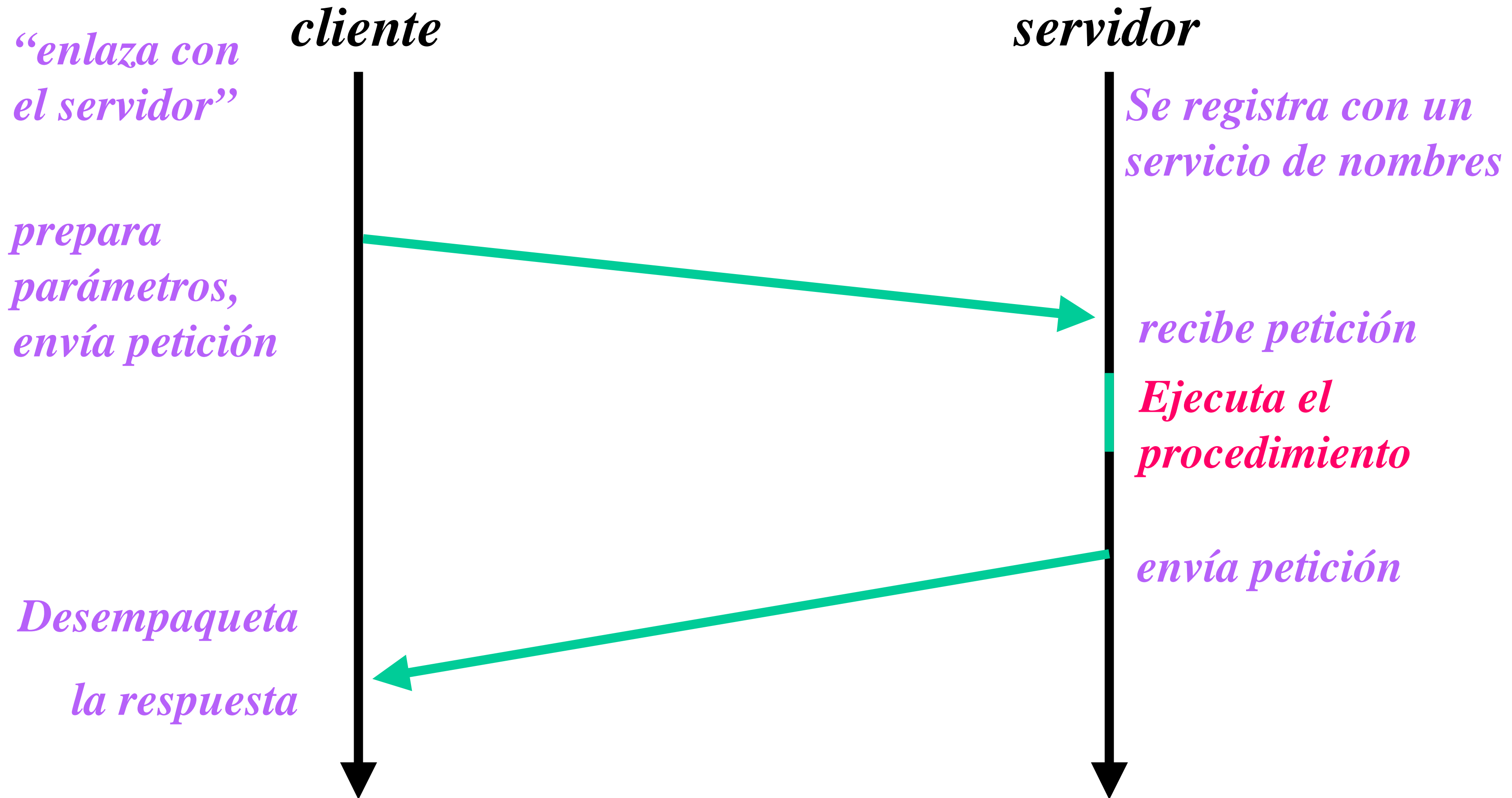
Enlazador dinámico (binder): Es el servicio que mantiene una tabla de traducciones entre nombres de servicio y direcciones. Incluye funciones para:

- 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

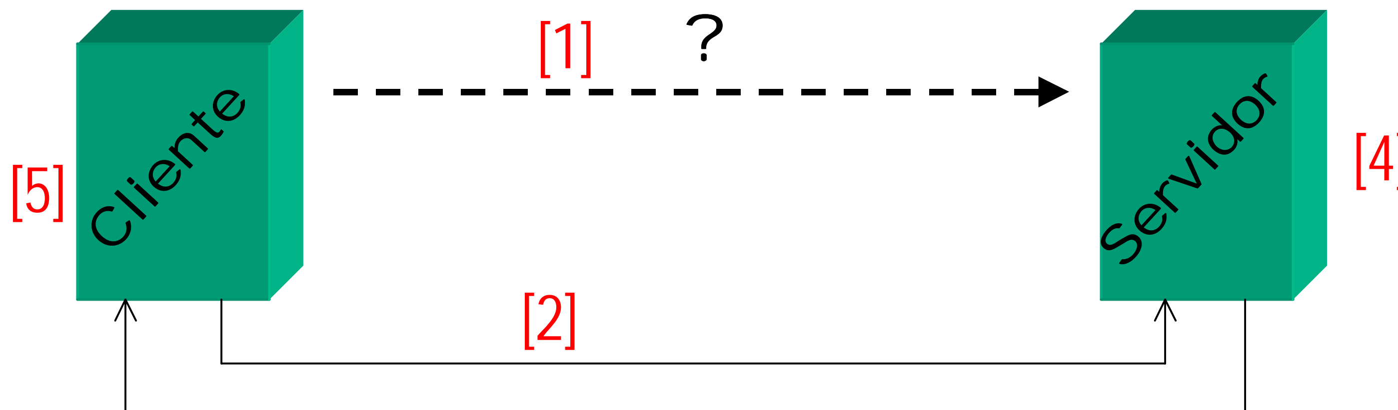


Fernando Pérez Costoya
José María Peña Sánchez

Semántica de 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 tenga 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.

Fernando Pérez Costoya
José María Peña Sánchez

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 nº de secuencia en el cliente
  - Un campo en el mensaje que indique si es una petición 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

Servidores con estado y sin estado

- Alternativa de diseño aplicable a cualquier servicio
- Influye en tolerancia a fallos
- Servidores con estado
  - Se mantiene información sobre los clientes
  - Mensajes de petición más cortos
  - Mejor rendimiento (se mantiene información en memoria)
  - Favorece estrategias predictivas:
    - El servidor puede analizar el patrón de accesos que realiza cada cliente
- Servidores sin estado
  - No se mantiene información sobre los clientes
  - Cada petición es autocontenida
  - Más tolerantes a fallos
  - Se reduce el nº de mensajes (p.ej. no son necesarios *open* y *close*)
  - No se gasta memoria en el servidor para almacenar el estado

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

Fernando Pérez Costoya
José María Peña Sánchez

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
- Coste 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 nº de programa y un nº de versión.
- Cada procedimiento específica un nombre y un nº 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*)
  - Fichero 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 (nº de programa, nº de versión y nº de puerto) en el *port mapper* local
  - El cliente envía una petición al *port mapper* del *host* donde ejecuta el servidor

Fernando Pérez Costoya
José María Peña Sánchez

Ejemplo de Fichero IDL (Sun RPC)

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

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

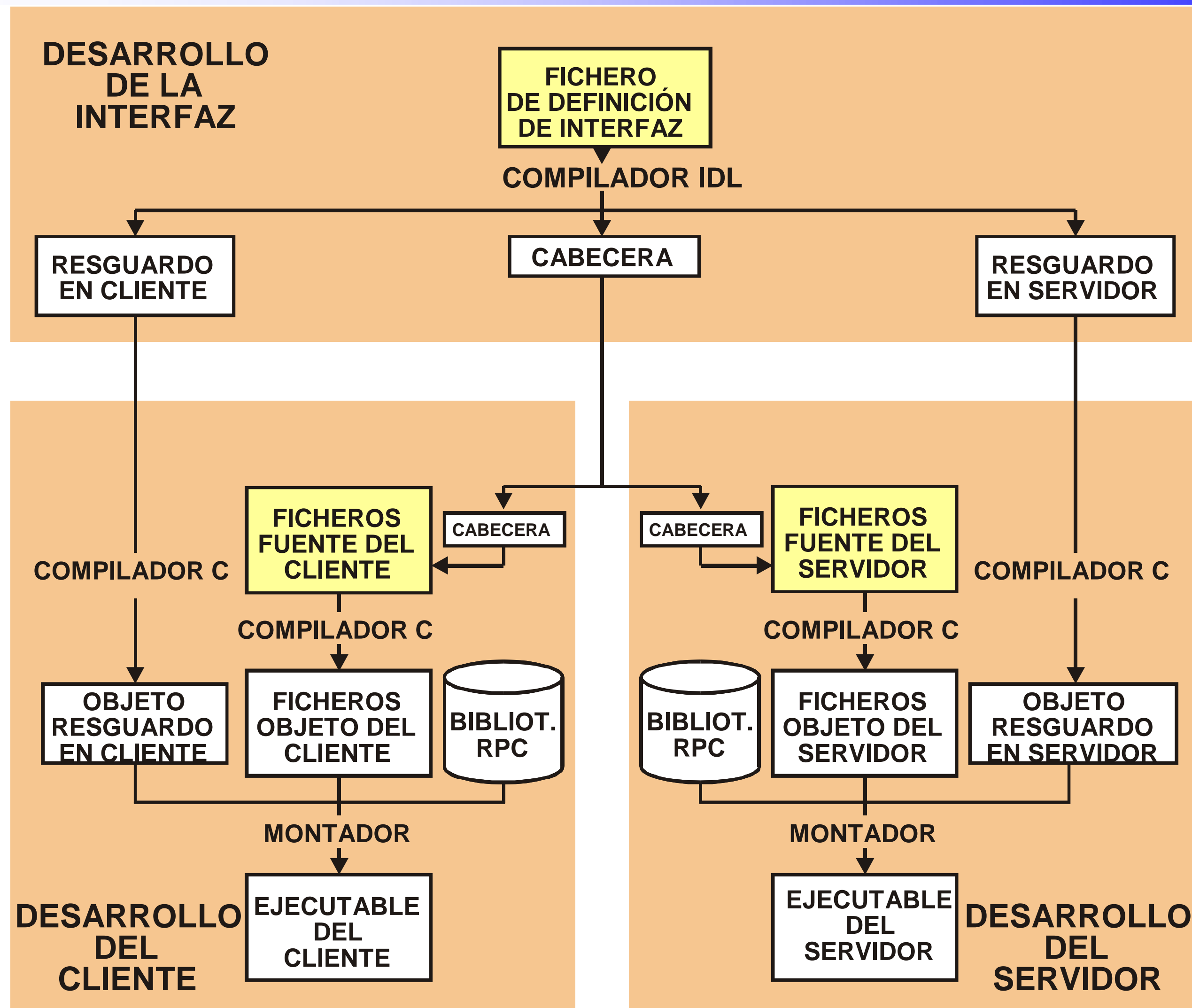

Fernando Pérez Costoya
José María Peña Sánchez

Programación con un Paquete de RPC

- El programador debe proporcionar:
  - La definición de la interfaz (fichero 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

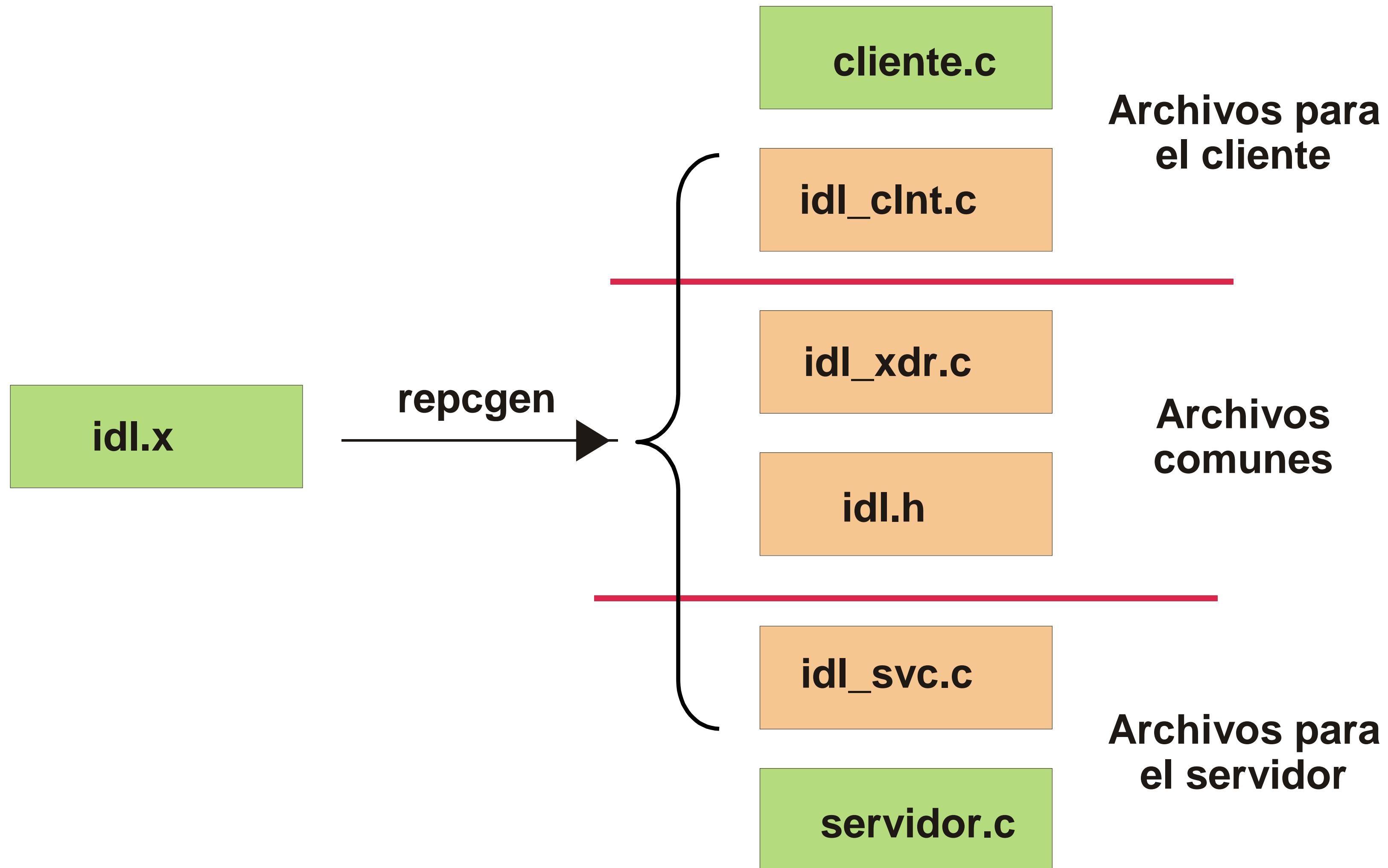
Fernando Pérez Costoya
José María Peña Sánchez

Programación con RPC



Fernando Pérez Costoya
José María Peña Sánchez

Esquema de una Aplicación



Fernando Pérez Costoya
José María Peña Sánchez