

# **Tema 2: Comunicación y Sincronización en Sistemas Distribuidos.**

## **2.1 Comunicación en Sistemas Distribuidos**

### **2.1.1. Protocolos en capas.**

### **2.1.2. El modelo Cliente-Servidor.**

#### **2.1.2.1. Clientes y Servidores.**

#### **2.1.2.2. Un ejemplo cliente-servidor.**

#### **2.1.2.3. Direccionamiento.**

#### **2.1.2.4. Primitivas de bloqueo vs. no bloqueo.**

#### **2.1.2.5. Primitivas almacenadas en buffer vs. no almacenadas.**

#### **2.1.2.6. Primitivas fiables vs. no fiables.**

#### **2.1.2.7. Implantación del modelo cliente-servidor.**

### **2.1.3. Llamadas a procedimientos remotos.**

#### **2.1.3.1. Operación básica de RPC.**

#### **2.1.3.2. Transferencia de parámetros.**

#### **2.1.3.3. Conexión dinámica.**

#### **2.1.3.4. Semántica de RPC en presencia de fallos.**

#### **2.1.3.5. Aspectos de implantación.**

#### **2.1.3.6. Áreas de problemas.**




### **2.1.4. Comunicación en grupo.**

#### **2.1.4.1. Introducción a la comunicación en grupo.**

#### **2.1.4.2. Aspectos de diseño.**

#### **2.1.4.3. Comunicación en grupo de ISIS.**

## **Bibliografía:**

-  Andrew S. Tanenbaum: "Sistemas Operativos Distribuidos"; tema 2, Prentice-Hall, 1996.
-  Coulouris, George: "Sistemas Distribuidos: conceptos y diseño"; tema 4 y 5, Addison-Wesley, 2001
-  William Stallings: "Sistemas Operativos"; 4ª Edición, tema 13, Prentice-Hall, 2001.

# Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

## 2.1. Comunicación en Sistemas Distribuidos.

- ✍ Diferencia más importante entre sistemas Distribuidos / sistemas Centralizados es: comunicación entre procesos. Los sistemas centralizados suponen de manera implícita la existencia de memoria compartida (problema productor-consumidor, semáforo).
- ✍ Normalmente, al conjunto de procesos u objetos que interactúan entre sí para implementar mecanismos de comunicación y de recursos compartidos para las aplicaciones distribuidas se le denomina **Middleware**.
- ✍ Proporciona abstracciones como: RPC, comunicación entre un grupo de procesos, notificación de eventos, replicación de datos compartidos y transmisión de datos multimedia en tiempo real.

## 2.1.1. Protocolos en capas.

- ✍ Toda comunicación en sistemas distribuidos se basa en **MENSAJES** (No memoria compartida).
- ✍ Es necesario adoptar una serie de acuerdos tanto de bajo nivel (transmisión de bits) como de alto nivel (formato e interpretación de la información)  
(ejemplo) Modelo OSI (interconexión de sistemas abiertos) de ISO (organización internacional de estándares).
- ✍ OSI : es un sistema abierto, Sistema preparado para comunicarse con cualquier otro sistema mediante reglas estándar que determinan el formato, contenido y significado de los mensajes enviados y recibidos (**protocolos**).
- ✍ OSI distingue entre protocolos:
  - ✍ orientados a conexión: antes de intercambiar datos, emisor y receptor establecen en forma explícita una conexión y un protocolo a utilizar.
  - ✍ sin conexión: el emisor transmite cuando está listo.
- ✍ OSI divide la comunicación en siete capas (físico, enlace, red, transporte, sesión, presentación y aplicación).
- ✍ Cada capa se encarga de un serie de funciones y proporciona un conjunto de operaciones que ofrece a la siguiente capa.

# Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

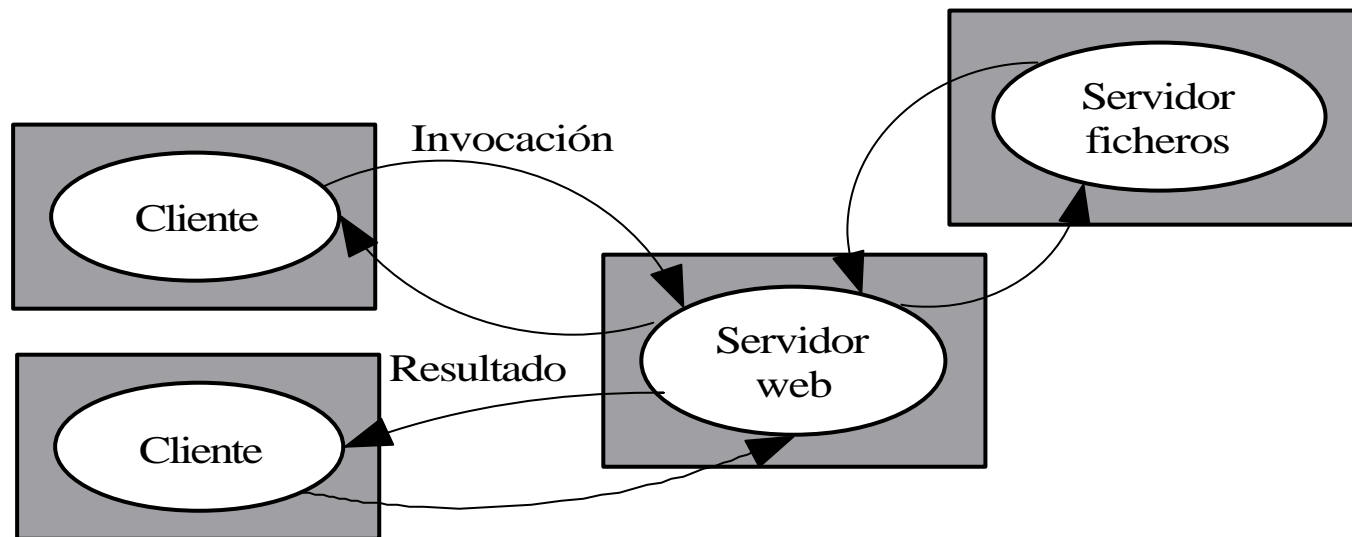
## 2.1.2. El modelo Cliente-Servidor.

- ✍ Problema: cada capa genera y añade/elimina un encabezado, supone un gasto excesivo para un sistema distribuido basado en LAN!!!!
- ✍ La mayoría de los sistemas distribuidos basados en LAN **NO** utilizan los protocolos con capas.
- ✍ OSI solo se encarga de transmitir bits e interpretar en un nivel superior pero no aporta nada con respecto a la estructura de un sistema distribuido.

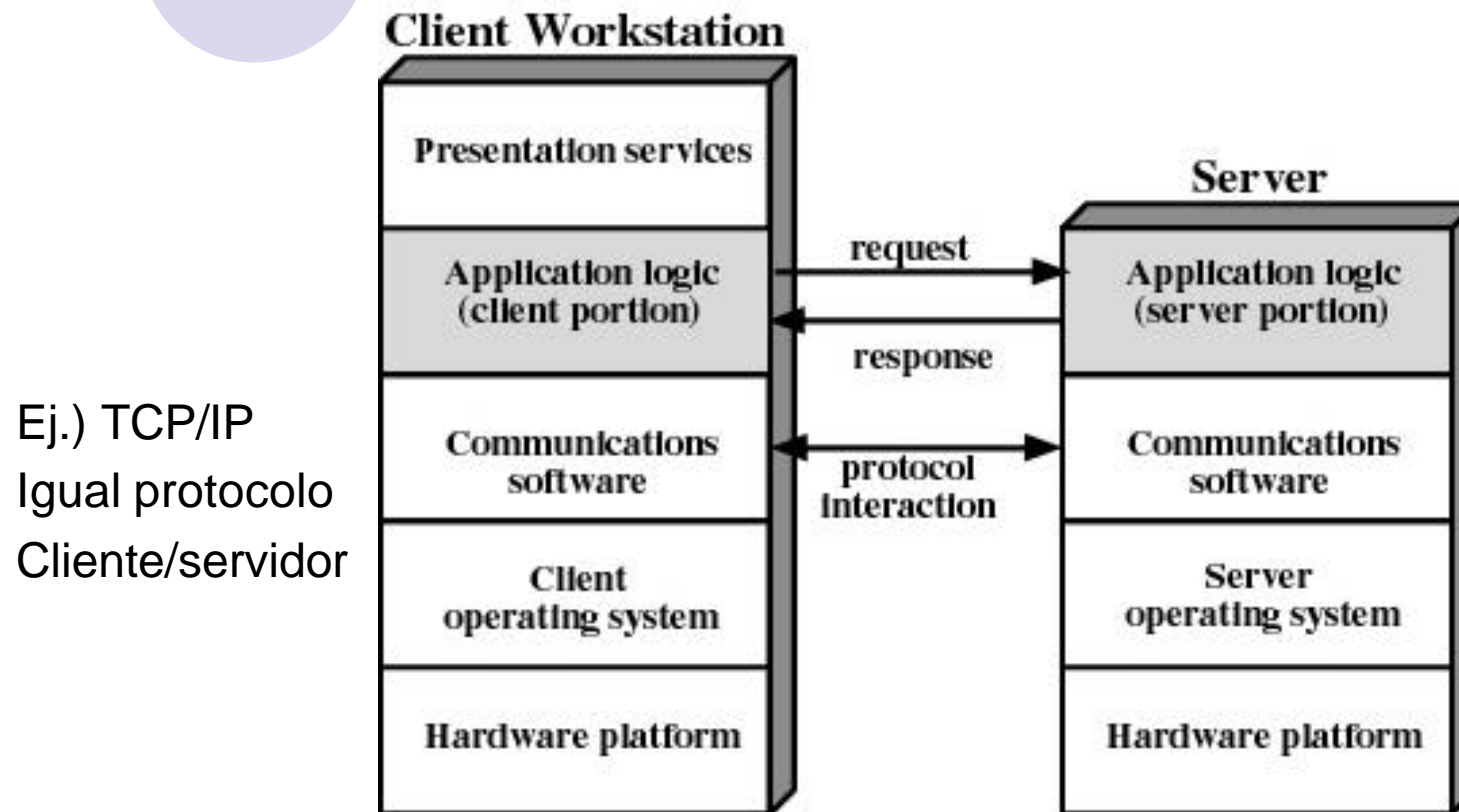
### 2.1.2.1. Clientes y Servidores.

- ✍ Modelo cliente-servidor: estructura el SO como un grupo de procesos en cooperación (servidores) que ofrecen servicio a los usuarios (clientes).
- ✍ Máquinas de clientes y servidores ejecutan el mismo micronúcleo.
- ✍ Para evitar gastos extras, el modelo cliente-servidor se basa en un protocolo solicitud/respuesta sencillo y sin conexión.
- ✍ El cliente envía un mensaje y obtiene un respuesta.
- ✍ Es eficiente, si todas las máquinas son iguales sólo se necesitan tres niveles de protocolos:
  - ✍ la capa física y de enlace se encargan de llevar paquetes entre servidor y cliente. Lo maneja el hardware (Ethernet o anillo).
  - ✍ No se realiza ruteo ni se establece conexión.
  - ✍ Equivalente a la capa de sesión es el protocolo solicitud/respuesta. Define el conjunto de solicitudes válidas y el conjunto de respuestas válidas.
- ✍ Los servicios de comunicación pueden ser reducidos a dos llamadas al sistema (envío y recepción de mensajes), *send(dest,&mptr)* *receive(addr,&mptr)*.

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.



## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

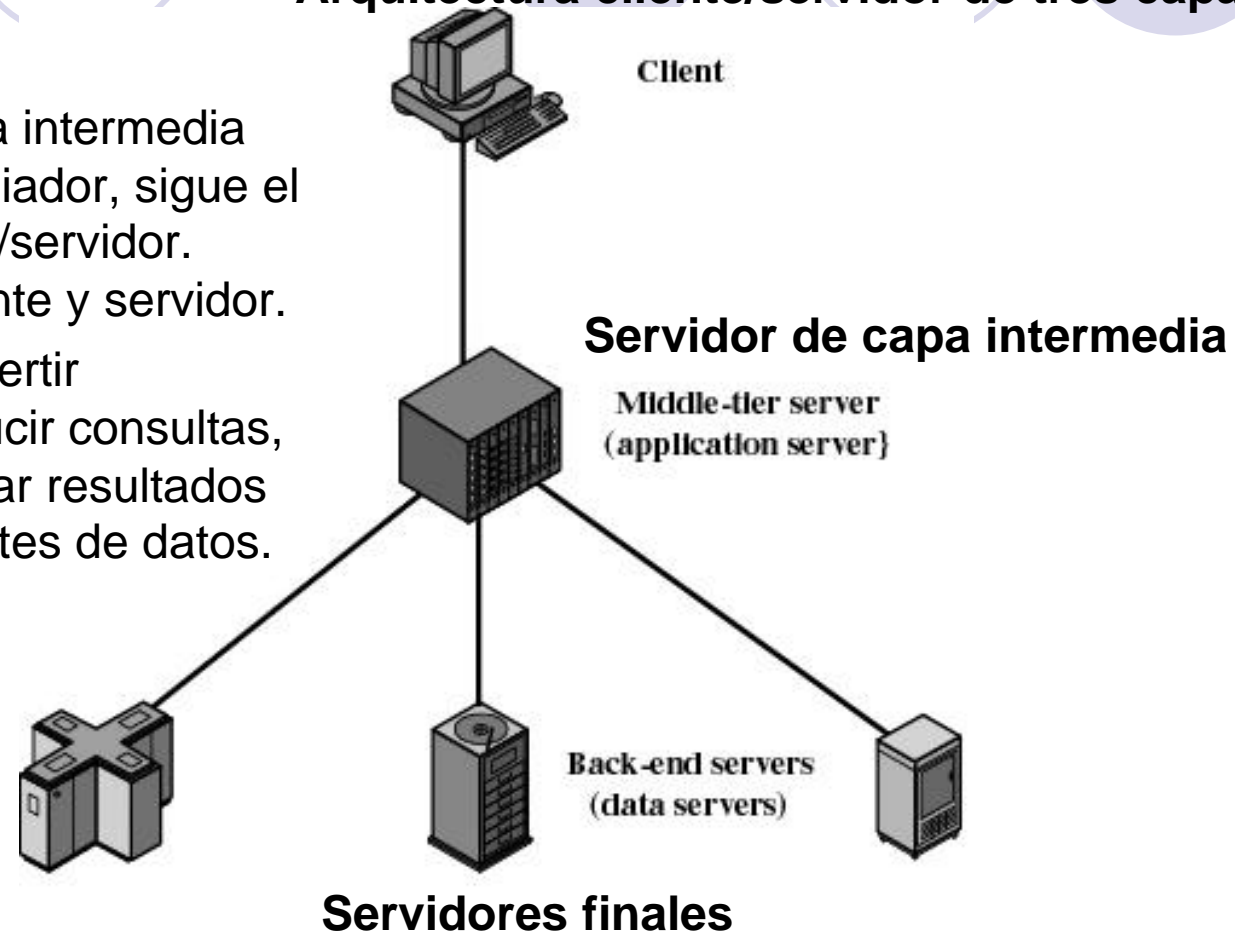


**Figure 13.2 Generic Client/Server Architecture**

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### Arquitectura cliente/servidor de tres capas

- ✍ Servidor de capa intermedia actúa como mediador, sigue el protocolo cliente/servidor. Actúa como cliente y servidor.
- ✍ Funciones: convertir protocolos, traducir consultas, mezclar e integrar resultados de distintas fuentes de datos.



**Figure 13.6 Three-tier Client/Server Architecture**

Ampliación de Sistemas Operativos.  
Sistemas Operativos Distribuidos

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

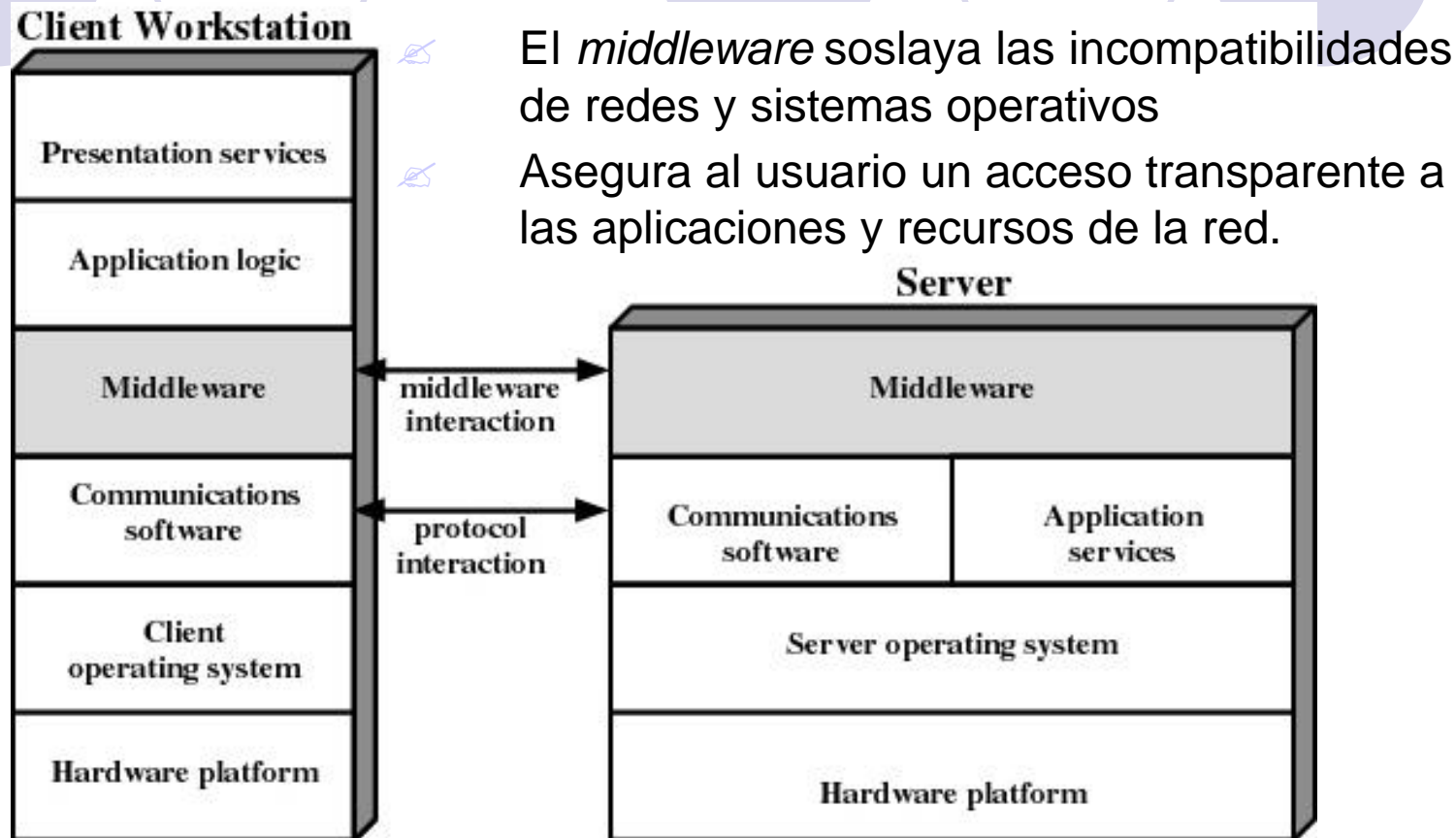


Figure 13.8 The Role of Middleware in Client/Server Architecture

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### 2.1.2.3. Direcccionamiento.

✍ Para que un cliente pueda enviar un mensaje a un servidor, debe conocer la dirección de éste.

ejemplo) la dirección se coloca dentro de *header.h* como una constante.

✍ Si en el envío de un mensaje sólo se indica la dirección de la máquina, nos obliga a ejecutar un proceso en cada máquina.

✍ ¿ Enviar mensajes a procesos? ¿Cómo identificar procesos?. Especificar el nombre de la máquina y el identificador del proceso. *maquine.process*.

✍ Variación: *maquine.local-id*. *local-id*: número entero de 16 o 32 bits. Cuando un proceso servidor se inicia comunica al núcleo que se desea escuchar *local-id* (comunicación de Berkeley).

✍ El direccionamiento *maquine.process* no es transparente, el usuario debe conocer la posición del servidor. (ejemplo) el servidor de ficheros se ejecuta en la máquina 234 que tiene algún problema. Otra máquina está disponible 145 pero los programas compilados con *header.h* tienen el número 234.

✍ Otra solución: Cada proceso una única dirección. Necesitamos un asignador centralizado de direcciones a los procesos. Pero, en grandes sistemas, debemos evitar sistema centralizados!!!.

✍ Otro método: asignar identificadores a los procesos de un espacio de direcciones grande. Probabilidad de que dos procesos tengan el mismo número es muy pequeña.

✍ Problema: ¿Cómo saber la máquina en la que está el proceso?.

✍ Solución: Utilizar un *paquete especial de localización*, este paquete de transmisión es recibido por todas las máquinas, todos los núcleos verifican si ese proceso les pertenece. La máquina que tiene ese proceso devuelve un mensaje con su dirección.

✍ Problema: Carga adicional en el sistema.

✍ Esta carga se puede evitar, destinando una máquina para asociación de nombres de servicios con nombres de máquinas. El **servidor de nombres** devuelve al cliente la dirección del servidor que pretende utilizar. Conocida la dirección, se solicita el servicio directamente.



## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

Método direccionamiento	Problema
Integrar <i>machine.number</i> al código del cliente	No es transparente
Dejar que los procesos elijan direcciones al azar y localizarlos mediante transmisiones	Carga adicional en el sistema
Utilizar servidores de nombres, buscarlos en tiempo de ejecución	Componente centralizado

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### 2.1.2.4. Primitivas de bloqueo vs. no bloqueo.

- ✍ Las primitivas de transferencia de mensajes pueden ser:
  - ✍ bloqueo (primitivas síncronas): *send* y *receive* ocasionan el bloqueo de los procesos que realizan la llamada. Hasta que el mensaje no es enviado o recibido no continúa la ejecución del proceso.
  - ✍ sin bloqueo (primitivas asíncronas): Después de *send* el proceso continúa su ejecución en paralelo con la transmisión del mensaje.
- ✍ Desventaja: el emisor no puede utilizar el buffer de mensajes hasta que el mensaje sea enviado y no sabe cuando puede terminar la transmisión

#### Soluciones:

- ✍ 1.- El núcleo copie el mensaje en el buffer interno del mismo. Tenemos una copia desperdiciada porque luego debemos copiar otra vez el mensaje en el buffer de transmisión hardware.
- ✍ 2.- Interrumpir al emisor para indicarle que el buffer está libre. Las interrupciones a nivel de usuario son difíciles de programar (método de encuesta o de interrupción).

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

Send con bloqueo	CPU inactiva durante la transmisión de los mensajes
Send sin bloqueo, con copia	Se desperdicia tiempo de CPU en copia adicional
Send sin bloqueo, con interrupción	dificulta la programación

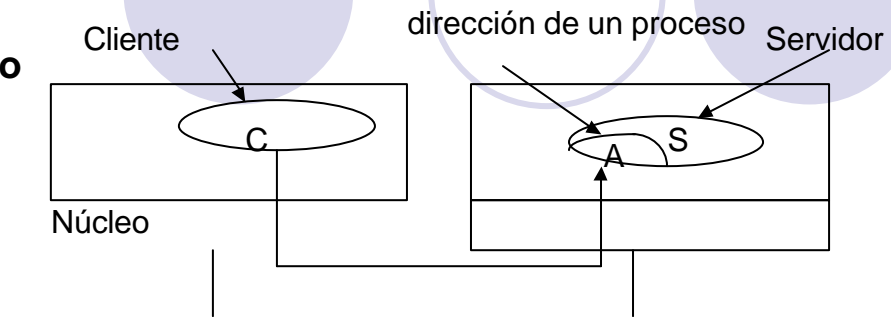
### ¿Cuál es la mejor?

- ✍ En condiciones normales, **Send con bloqueo**. No maximiza el paralelismo pero es fácil de comprender y de implementar. No requiere el manejo de buffer adicionales.
- ✍ Si es esencial el paralelismo de procesamiento y transmisión, la mejor opción es *send sin bloqueo con copia*.
- ✍ Receive sin bloqueo: indica al núcleo la localización del buffer y regresa de forma inmediata. Se prefiere la versión con bloqueo.
- ✍ Tiempos de espera??. send con bloqueo que no recibe respuesta, supone el bloqueo del emisor para siempre. Para evitarlo, se puede especificar un espacio de tiempo por el que se espera respuesta.

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### 2.1.2.5. Primitivas almacenadas en buffer vs. no almacenadas.

- ✍ primitivas no almacenadas. La dirección se refiere a un proceso específico *receive(addr,&m)*.
- ✍ este esquema funciona bien si el servidor llama a *receive* antes de que el cliente llame a *send*. *receive* indica al núcleo la dirección que va utilizar y la posición donde colocar el mensaje que está por llegar.
- ✍ si *send* se ejecuta antes ? descartar el mensaje y confiar en que el cliente vuelva a transmitir después de que el servidor ejecute *receive*. El cliente debe intentarlo antes de tener éxito!!!!.
- ✍ si varios clientes solicitan un mismo servicio, es probable que el servidor esté ocupado y no ejecute *receive*.
- ✍ se puede mantener durante un breve espacio de tiempo los mensajes pendientes por el núcleo del receptor. Si expira el tiempo, el mensaje se descarta. Necesitamos buffers!!!!.



- ✍ Debemos definir buzones. Un proceso interesado en recibir un mensaje, crea un buzón y especifica una dirección en la que buscar los paquetes de la red. Todos los mensajes que lleguen a la dirección se colocan en el buzón.
- ✍ *receive*: elimina un mensaje del buzón o se bloquea si no existen mensajes. Primitivas con almacenamiento en buffers.
- ✍ Los buzones son finitos!!!. Mantener el mensaje un tiempo, o descartarlo. Se reduce la probabilidad de problemas pero no se eliminan.
- ✍ Otra opción: No enviar el mensaje hasta que un proceso esté dispuesto a recibirlo. El cliente se bloquea hasta recibir un reconocimiento. Si el buzón está ocupado ? el emisor hace un respaldo y se bloquea hasta que podamos disponer del buzón.

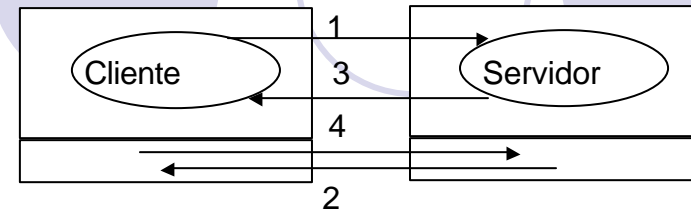
## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### 2.1.2.6. Primitivas fiables vs. no fiables.

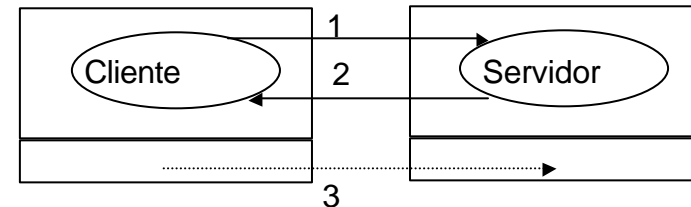
- ✍ Hemos supuesto que, cuando un cliente envía un mensaje, el servidor lo recibirá!!!. Los mensajes se pueden perder.

#### Tres enfoques:

1. send no confiable: El sistema no da garantía sobre la entrega de mensajes (se deja en manos de los usuarios).
  2. El núcleo de la máquina receptora tiene que enviar un reconocimiento al núcleo de la máquina emisora. Es entonces cuando el núcleo-emisor libera el proceso cliente.
  3. El cliente se bloquea después de enviar un mensaje. La respuesta del servidor funciona como reconocimiento. Si tarda el núcleo envía de nuevo la solicitud. No existe reconocimiento para la respuesta!!!.
- ✍ Esto tendrá repercusiones dependiendo del tipo de solicitud (ej. un gran cálculo por parte del servidor). No se debería descartar la respuesta hasta asegurarse de la recepción por parte del emisor. En algunos sistemas se reconoce la respuesta



1. Solicitud (del cliente al servidor)
2. Reconocimiento (de núcleo a núcleo)
3. Respuesta (del servidor al cliente)
4. Reconocimiento (de núcleo a núcleo)



1. Solicitud (del cliente al servidor)
  2. Respuesta (del servidor al cliente)
  3. Reconocimiento (de núcleo a núcleo)
- ✍ Combinación de los métodos 2 y 3: Utilizar un cronómetro en el servidor. Si la respuesta es rápida ( tres mensajes), si expira el tiempo, entonces se envía un reconocimiento separado (4 mensajes).

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### 2.1.2.7. Implantación del modelo cliente-servidor.

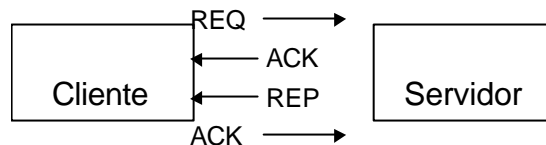
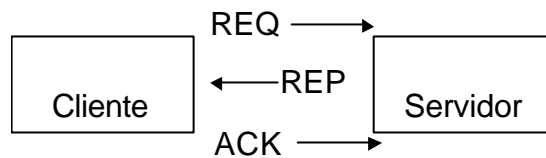
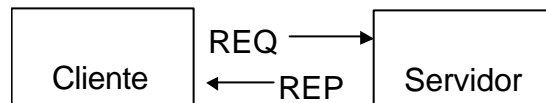
Direccionamiento	Integrar <i>machine.number</i> al código del cliente	Direcciones al azar para procesos y localizarlos mediante transmisiones	Utilizar servidores de nombres, buscarlos en tiempo de ejecución
Bloqueo	Send con bloqueo	Send sin bloqueo, con copia	Send sin bloqueo, con interrupción
Almacenamiento en buffers	No usar buffers, descartar los mensajes inesperados	Sin buffers, manteniendo temporalmente los mensajes inesperados	Buzones
Fiabilidad	No fiable	Solicitud-Reconocimiento-Respuesta-Reconocimiento	Solicitud-Respuesta-Reconocimiento

- ✍ Todas las redes tienen un tamaño virtual máximo de paquete (cientos de bytes).
- ✍ Los mensajes suelen perderse y llegar en un orden equivocado. Asignar un número a cada paquete según el orden de envío.
- ✍ Reconocimiento: ¿De cada paquete o del mensaje completo?. Valorar la tasa de pérdidas en la red: a) R. paquete: sobrecarga red; b) R. mensaje: retransmisión del mensaje completo.

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

Protocolo (suponemos que cada mensaje cabe en un paquete).

Código	Tipo de paquete	De	A	Descripción
<b>REQ</b>	Solicitud	Cliente	Servidor	El cliente desea servicio
<b>REP</b>	Respuesta	Servidor	Cliente	Respuesta del servidor al cliente
<b>ACK</b>	Reconocimiento	Cualquiera	El otro	Llegada del paquete
<b>AYA</b>	¿Estás vivo?	Cliente	Servidor	Verifica si el servidor está dispuesto
<b>IAA</b>	Estoy vivo	Servidor	Cliente	El servidor está dispuesto
<b>TA</b>	Reintentar	Servidor	Cliente	El servidor no tiene espacio
<b>AU</b>	Dirección desconocida	Servidor	Cliente	Ningún proceso utiliza esta dirección



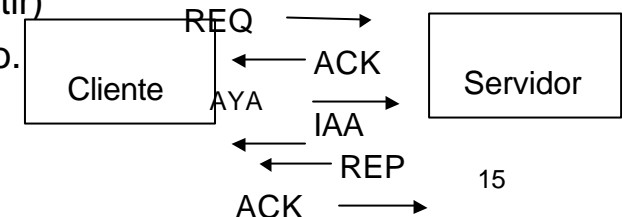
✍ AYA: El cliente recibe el reconocimiento del servidor pero no recibe la respuesta, ¿qué sucede? ¿la solicitud es complicada o el servidor está descompuesto?. Si la respuesta es IAA el cliente sigue esperando. Verifican el estado del servidor.

✍ TA y AU: el paquete REQ no es aceptado, ¿por qué?:

✍ el buzón está totalmente ocupado (repetir)

✍ la dirección ? a ningún buzón o proceso.

✍ el servidor no ha ejecutado *receive*.



## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### 2.1.3. Llamadas a procedimientos remotos.

- ✍ Modelo cliente-servidor: la comunicación la construye sobre la E/S.
- ✍ La E/S no es un concepto fundamental en los sistema centralizados. Si queremos que el cómputo distribuido se parezca al centralizado debemos evitar la utilización de primitivas *send/receive*.
- ✍ Alternativa (Birrell y Nelson 1984): permitir a los programas que llamen a procedimientos localizados en otras máquinas. La información entre procedimientos se transporta mediante parámetros: *Request procedure call*.
- ✍ **Problemas:** espacios de direcciones distintos, tipos de datos, las máquinas se pueden descomponer. Pese a ello RPC es una técnica de gran uso en los SO distribuidos.

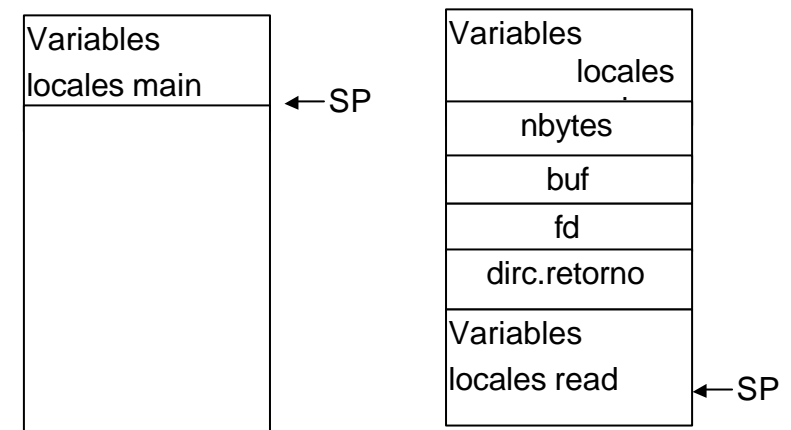
#### 2.1.3.1. Operación básica de RPC.

- ✍ Funcionamiento de una llamada convencional:

*count = read (fd, buf, nbytes);*

*int fd, nbytes; char buf[N?;*

- ✍ la pila antes/durante de la llamada a *read*
- ✍ los parámetros pueden pasarse por valor o por referencia.





## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

- ✍ Otra alternativa es utilizar: llamada copiar/restaurar. El valor se copia en la pila y la copia se vuelve a copiar después de la llamada.
- ✍ Queremos que RPC sea transparente: el procedimiento que hace la llamada no debe ser consciente de que el procedimiento llamado se ejecuta en una máquina distinta.
- ✍ ej.) un programa necesita leer datos de un fichero: procedimiento read
- 1. el enlazador extrae la rutina read de la biblioteca y lo inserta en el programa objeto.
- 2. read coloca los parámetros en registros y después hace una llamada al sistema READ al núcleo.

- ✍ ej. en RPC) READ se ejecuta en la máquina del servidor de ficheros
- 1. en la biblioteca se coloca una versión ? de read: **stub del cliente**. El procedimiento cliente llama al stub del cliente.
- 2. construye con los parámetros un mensaje y le pide al núcleo que lo envíe al servidor.
- 3. el núcleo envía el mensaje al núcleo remoto.
- 4. el mensaje llega al servidor y el núcleo lo transfiere a un **stub del servidor** que ya habrá ejecutado receive.
- 5. el **stub del servidor** desempaca los parámetros del mensaje y llama al procedimiento del servidor. El servidor no distingue si la llamada es local o remoto: los parámetros y la dirección de retorno están en la pila.

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

6. Cuando el servidor termina el trabajo, devuelve los resultados (envía los datos al buffer al que apunta el segundo parámetro, el buffer puede ser interno del stub del cliente).
7. el stub del servidor empaqueta el resultado en un mensaje y llama send con la dirección del cliente y luego a receive (bloquea).
8. el núcleo remoto envía el mensaje al núcleo del cliente.
9. el núcleo del cliente copia el mensaje en el stub del cliente.
10. el stub del cliente examina el mensaje, desempaca el resultado y se lo da al cliente.

- ✍ Cuando el proceso que hizo la llamada obtiene el control dispone de los datos sin saber quién realizó el trabajo.
- ✍ Todo el detalle de transferencia de mensajes se oculta en dos procedimientos de biblioteca (al igual que los detalles de interrupciones en las llamadas al sistema).

### 2.1.3.2. Transferencia de parámetros.

- ✍ La función del stub del cliente es tomar sus parámetros, empacarlos en un mensaje y enviarlos al stub del servidor:  
Paso de parámetros en RPC

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

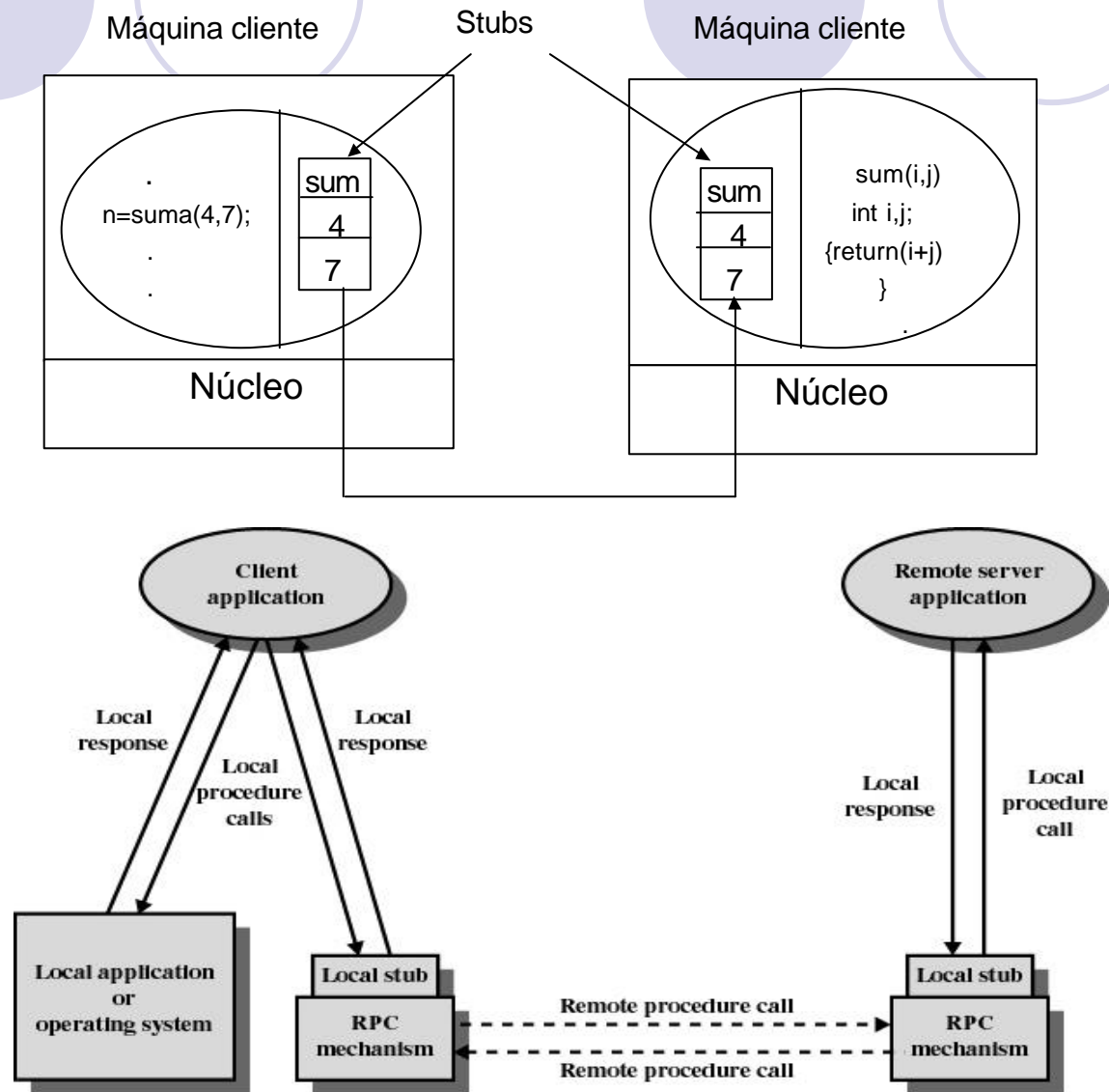


Figure 13.13 Remote Procedure Call Mechanism

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

✍ Si las máquinas cliente y servidor son ? y todos los parámetros y resultados de tipo escalar, este modelo funciona bien.

✍ Con frecuencia cada máquina tiene su propia representación de números y caracteres y tipos de datos.

ej) ASCII (1 byte por carácter)

UNICODE (2 bytes por carácter)  
complemento a uno ó a 2, numeración de los bytes (byte más significativo va 1º, Intel) o viceversa (SPARC).

✍ Podemos adoptar un acuerdo de representación y realizar las conversiones necesarias en cliente/servidor.

✍ Formato externo

✍ Formato en el emisor con indicación.

✍ Sin embargo, este método es ineficiente cuando el número de máquinas sean muy dispares y el número de conversiones excesivo.

✍ Pero, ¿dónde hacen su aparición los stubs?: en sistemas basados en RPC, dada una especificación del servidor y de las reglas de codificación podemos tener un compilador que genere los procedimientos: stub del cliente y del servidor.

✍ Pero, ¿cómo transferir los apuntadores?. Un apuntador sólo tiene sentido en el espacio de direcciones del proceso que lo utiliza.

### **Soluciones:**

✍ Realizar sólo pasos por valor, parece poco razonable.

✍ Sustituir el paso por referencia por copia/restauración. Podemos optimar este mecanismo si sabemos si el parámetro es de salida o de entrada.

✍ Seguimos sin poder utilizar apuntadores a una estructura de datos arbitraria. Se puede realizar un direccionamiento indirecto, es muy ineficiente pero muchos sistemas lo utilizan.

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### 2.1.3.3. Conexión dinámica.

¿Cómo localizar al servidor???

- ✍ Integrar dentro del código del cliente la dirección del servidor. Es muy rígido, si el servidor se desplaza, duplica o cambia la interfaz, habría que compilar numerosos programas.
- ✍ Por ello es mejor utilizar conexión dinámica.

Especificación formal del servidor

ej.) `?include <header.h>`

specification of file\_server, version 3.1:

```
long read(in char name[MAX_PATH?, out
char buff[BUF_SIZE?, in long bytes, in long
position);
```

```
long write(in char name[MAX_PATH?, out
char buff[BUF_SIZE?, in long bytes, in long
position);
```

```
int create(in char name[MAX_PATH?, in int
mode);
```

```
int delete(in char name[MAX_PATH?);
```

end;

- ✍ La especificación formal se utiliza por el generador automático de stubs, produce tanto el stub del cliente como el del servidor y los coloca en las bibliotecas respectivas.

- ✍ Cuando el servidor ejecuta su inicialización exporta la interfaz del servidor.

- ✍ El servidor envía un mensaje a un programa (*conector*) para darle a conocer su existencia (*registro del servidor*).






- ✍ El servidor proporciona al conector su nombre, número de versión, un único identificador y un handle (dirección Ethernet, una dirección IP, una dirección X.500, etc) para localizarlo.

- ✍ Cuando el cliente llama a alguno de los procedimientos remotos por primera vez, el stub del cliente envía un mensaje al conector donde solicita la importación de la versión de la interfaz del servidor.



- ✍ El conector verifica si existe el servidor adecuado y proporciona su handle e identificador único (existe más de un servidor) al stub del cliente.

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### Ventajas de exportación/importación de interfaces:

-  puede manejar varios servidores con la misma interfaz.
-  el conector puede difundir los clientes de manera aleatoria entre los servidores, carga más justa.
-  el conector puede hacer un registro de los servidores y cancelar el que no responda.
-  el conector puede ayudar en la autenticación.
-  el conector verifica que tanto servidor como cliente utilizan la misma versión de interfaz.

### Desventajas de exportación/importación de interfaces:

-  Costo de tiempo debido a la exportación/importación de interfaces. La duración de un proceso cliente es corta.
-  El conector se puede convertir en un cuello de botella. Podemos utilizar más conectores pero entonces incrementamos el número de mensajes para mantener sincronizados y actualizados los conectores.

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### 2.1.3.4. Semántica de RPC en presencia de fallos.

? Objetivo de RPC: Ocultar la comunicación.

- ✍ Problema: ¿Qué hacemos cuando aparecen errores?. No es fácil seguir tratando las llamadas remotas como si fueran locales.

#### Tipos de problemas:

##### El cliente no puede localizar al servidor

- ✍ el servidor está inactivo o se ha cambiado la versión de la interfaz.
- ✍ Podemos provocar una *excepción* para indicar este hecho.
- ✍ No todos los lenguajes manejan señales, la construcción de un manejador de excepciones viola la transparencia.

##### Pérdida de mensajes de solicitud

- ✍ El núcleo puede utilizar un cronómetro al enviar su solicitud. Si expira reenvía su solicitud.
- ✍ Si la pérdida de mensajes supera un número entonces tenemos el problema anterior “no se puede localizar el servidor”.

##### Pérdida de mensajes de respuesta

- ✍ Utilizar un cronómetro y volver a enviar la solicitud. El núcleo del cliente no está seguro de la razón por la que no hay respuesta: ¿Se perdió la solicitud? ¿Se perdió la respuesta? ¿El servidor es lento?.

ej) Hay solicitudes comprometidas:

transferencia de un millón de pesetas.

- ✍ Solución: 1) Asignar a cada solicitud un número secuencia o 2) incluir un bit adicional en el encabezado del mensaje.



## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### Fallos del servidor

- ✍ a) Llega una solicitud y el servidor la atiende pero falla antes de enviar la respuesta (excepción) o b) falla antes de resolverla (reenviar la solicitud).
- ✍ El cliente no puede determinar la naturaleza del problema y el tratamiento es distinto para a) y b).

#### Solución:

- ✍ esperar a que el servidor se reponga y realizar la operación de nuevo. Semántica al menos una (garantiza la RPC)
- ✍ darse por vencido e informar del fallo. Semántica a lo más una (RPC se realiza como mucho una vez)
- ✍ no garantizar nada.

### Fallos del cliente

- ✍ El cliente falla antes de obtener la respuesta del servidor (huérfanos).

#### Soluciones (Nelson 1981):

- ✍ Exterminación: Antes de enviar el mensaje, el stub del cliente crea en disco un registro indicando la intención. Después de arrancar, se verifica el contenido del registro y el huérfano se elimina de forma explícita.
  - ✍ !!! Gran gasto de escritura para una RPC. Un proceso puede solicitar otro servicio (huérfanos de huérfanos).
- ✍ Reencarnación: Dividir el tiempo en épocas numeradas secuencialmente. Cuando un cliente vuelve a arrancar, envía un mensaje a todas las máquinas declarando el inicio de una nueva época, se eliminan los cómputos remotos.
- ✍ Reencarnación sutil: Cuando el servidor recibe el mensaje de nueva época, determina si tiene cómputos remotos e intenta localizar al poseedor. Si no lo localiza, elimina el cómputo.
- ✍ Expiración: A cada RPC se le asigna un tiempo  $T$ . Si no lo puede terminar, pide otro quantum.
- ✍ Ningún método es recomendable. La eliminación de un huérfano puede tener consecuencias imprevisibles (un proceso puede bloquear una serie de ficheros).



## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### 2.1.3.5. Aspectos de implantación.

- ✍ El éxito de un sistema distribuido depende de la velocidad de comunicación. Y esta de la implementación y no de sus principios abstractos.

### Protocolos RPC

#### ¿Protocolo orientado a conexión o no?

- ✍ En un protocolo orientado a conexión la comunicación es más fácil, no debemos preocuparnos por pérdidas y reconocimientos (software de red).
- ✍ En una LAN es poco eficaz e inútil debido a su fiabilidad.

#### ¿Un protocolo estándar o diseñado para RPC?

- ✍ Al no existir estándares en RPC, el uso de un protocolo adaptado a RPC significa que cada cuál diseñe el suyo.
- ✍ Algunos sistemas distribuidos utilizan IP/UDP:

- ✍ El protocolo ya ha sido diseñado y se disponen de muchas implementaciones, supone ahorro de trabajo.

- ✍ Estos protocolos se pueden utilizar en UNIX y en diversas redes.

- ✍ Pero IP no se diseñó para RPC. Utiliza mucha información adicional carente de sentido en RPC.

- ✍ A largo plazo, se deberá desarrollar protocolos RPC. Una RPC implica un gran costo independientemente del tamaño de la transmisión. Es importante que permitan transmisiones largas.

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### Reconocimientos

¿Deben reconocerse los paquetes de manera individual o no?

- ✍ Protocolo detenerse y esperar (stop-and-wait protocol) vs protocolo de chorro (blast protocol).
- ✍ Facilidad a la hora de recuperar paquetes (mejor protocolo detenerse y esperar). Si utilizamos protocolo a chorro, debemos decidir cuándo y cómo subsanar la pérdida de un paquete (**repetición selectiva**).
- ✍ Uso del ancho de banda de la red (mejor repetición selectiva). En un LAN, los paquetes perdidos son raros y es mejor el método “detenerse y esperar” o el abandono total (protocolo a chorro).

### ¿Control de flujo?

- ✍ La mayoría de los chips no pueden aceptar dos paquetes adyacentes (overrun error) y el segundo paquete se pierde.
- ✍ Con el método detenerse y esperar este tipo de error es imposible. Para el caso de protocolo a chorro se puede introducir un cierto retardo entre paquetes.
- ✍ Si la sobreejecución se debe a la capacidad finita del buffer en el chip de la red, entonces el emisor puede enviar n paquetes y después esperar.
- ✍ La minimización del reconocimiento de paquetes y la obtención de un buen desempeño depende de las propiedades de sincronización del chip de red (el protocolo debe ajustarse al hardware que utilice).

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### Ruta crítica

✍ ruta crítica: serie de instrucciones que se ejecutan con cada RPC.

¿En qué parte de la ruta crítica se invierte la mayoría del tiempo?

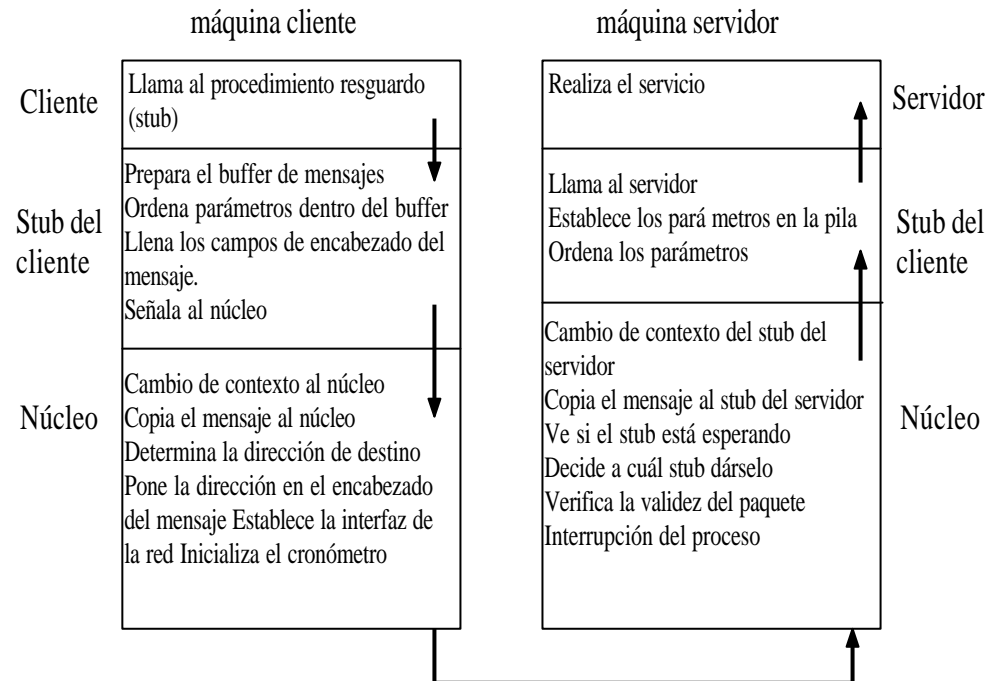
✍ *Empaquetamiento*: que lleva asociado la copia y la conversión de datos.

✍ *Copia de datos*: además serán copiados: 1) entre el usuario y el núcleo, 2) en cada nivel del protocolo (RPC/UDP/IP/Ethernet) y 3) entre la interfaz de red y los búferes del núcleo.

✍ *Inicialización de paquetes*: cabeceras y terminaciones (checksums).

✍ *Planificación de hilos y cambio de contexto*: 1) invocar a los stubs 2) planificar uno o más hilos en el servidor y 3) por cada operación send.

✍ *Espera por reconocimientos*



## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### Copiado

- ✍ El tiempo de ejecución de RPC se ve sancionado por el número de copias de un mensaje (entre uno a ocho, según el hardware, software y tipo de llamada).
- ✍ mejor caso: el chip de red utiliza DMA para transferir el mensaje del espacio del stub del cliente a la red (copia1). Cuando el núcleo del servidor recibe el mensaje, copia el paquete en el stub del servidor (copia2).
- ✍ pero caso: el núcleo copia el mensaje de stub del cliente en el buffer del núcleo para poderlo transferir (copia1). El núcleo copia el mensaje a un buffer hardware de la tarjeta de red (copia2). El mensaje se deposita en la red para ser transmitido (copia3). Cuando llega al servidor, el núcleo lo copia a un buffer del núcleo (copia 4). El mensaje debe ser copiado al stub de servidor (copia5). Además, si la llamada transfiere como parámetro un gran array, debe copiarse en la pila del cliente para la llamada al stub, de la pila al buffer de mensajes y del stub del servidor a la pila del servidor (tres más).

- ✍ ej) si para copiar 32 bits invertimos 500nseg ? 8 copias suponen 4microseg ? sea cual sea la vel. de la red, la tasa máx =1Mbyte/seg
- ✍ **dispersión-asociación** (scatter-gather): característica hardware que elimina el copiado innecesario. El chip de red organiza un paquete concatenando dos o más búferes de memoria (se pueden construir el encabezado desde el núcleo y los datos desde el stub del cliente).
- ✍ Es más fácil eliminar la copia en el emisor. Difícilmente una tarjeta de red puede determinar a qué servidor asignarlo.
- ✍ En los SO con memoria virtual se puede evitar el copiado al stub. Si el buffer del núcleo tiene un tamaño = a una página y el buffer del receptor del stub del servidor tiene el mismo tamaño, el núcleo puede modificar el mapa de memoria para asociar el buffer con el paquete en el espacio de direcciones del servidor y enviar el buffer del stub del servidor al núcleo.

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### ¿Todo este trabajo vale la pena??

- ✍ Sí, si el mapa de memoria se puede actualizar en menos tiempo.
- ✍ Requiere un control cuidadoso de los búferes. El proceso de usuario puede ver la cabecera del paquete, se viola la portabilidad.
- ✍ si los búferes se alinean de modo inverso, cabecera después de los datos, estos se pueden asociar al espacio de direcciones sin el encabezamiento. Pero requiere el empleo de dos páginas por buffer, un con la cabecera y otra con los datos.

### **Manejo del cronómetro**

- ✍ La mayoría de los protocolos inicializan un cronómetro cada vez que se envía un mensaje y se espera una respuesta. Si no llega se retransmite.

- ✍ El manejo de un cronómetro requiere la construcción de una estructura de datos que especifique el momento en que el cronómetro debe detenerse y la acción a realizar en caso de que suceda.
- ✍ Tendremos una lista de cronómetros ordenados. Si llega un reconocimiento antes de expirar el tiempo, se debe localizar la entrada y eliminarla de la lista.
- ✍ Muy pocos cronómetros expiran, la mayoría del trabajo se dedica a introducir y eliminar un cronómetro en la lista. El tiempo es aproximado.!!

### ¿No hay otra forma más eficiente de manejar los cronómetros?

- ✍ El tiempo de expiración se puede incluir en BCP, ahora la activación consiste en poner la suma del tiempo actual más el tiempo de expiración en ese campo y la desactivación ponerlo a 0. El núcleo debe de rastrear de forma periódica la tabla de procesos, para comparar el valor de cada cronómetro con el tiempo actual.

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### 2.1.3.6. Áreas de problemas.

- ✍ RPC mediante el modelo cliente-servidor se utiliza ampliamente en sist. distribuido pero no carece de problemas.
- ✍ Una RPC debe ser transparente: el programador no debe saber si los procedimientos de biblioteca son remotos o locales. Debe poder escribir procedimientos sin importar si se ejecutarán de manera local o remota. El introducir una RPC en un sistema que se ejecutaba en una CPU no debe ir acompañada de prohibición de uso de construcciones antes válidas ni exigir construcciones nuevas.

**NINGUN SIST. DISTRIBUIDO ACTUAL ES TRANSPARENTE!!**

✍ Variables globales: No es posible el uso de variables globales por un procedimiento remoto (se viola el principio de transparencia).

✍ lenguajes débilmente tipificados (C): es posible escribir un procedimiento sin especificar el tamaño de un vector ? es imposible que el stub del cliente ordene los parámetros (no puede determinar su tamaño)

✍ No podemos usar apuntadores.

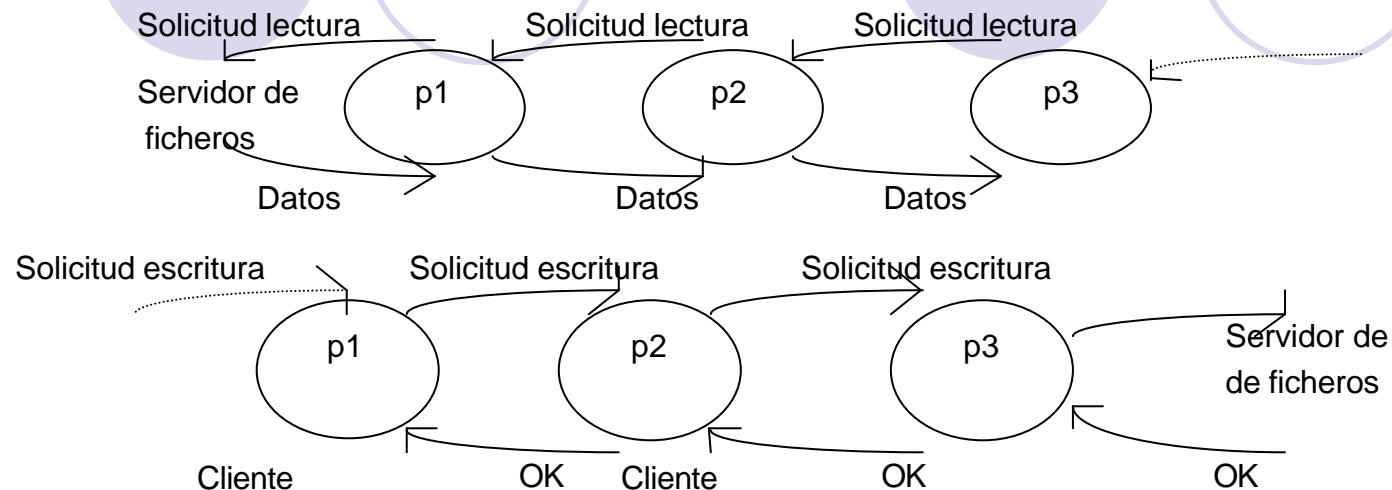
✍ Hay lenguajes que permiten un número arbitrario de parámetros (printf).

✍ Existen problemas con todos los pipes de la forma:  $p1 < f1 \mid p2 \mid p3 > f2$

*grep rat < f5 \mid sort > f6* ¿Quién es el cliente y quién el ser servidor?

✍ No funciona un pipe manejado por lectura ni por escritura!!!

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.



- ✍ El modelo cliente-servidor basado en RPC no se ajusta a este tipo de comunicación.
- ✍ Posible solución: pipes funciones como servidores duales o implementarlos como ficheros temporales que se leen o escriben en el servidor de ficheros, esto supone un gran costo.
- ✍ Un problema parecido presenta un shell que quiere obtener una entrada de usuario (el shell es un cliente del servidor del terminal pero cuando el usuario pulsa Ctrl-C, entonces el terminal se convierte en cliente y el shell en servidor).



## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### 2.1.4. Comunicación en grupo.

- ✍ RPC supone una comunicación entre dos partes: cliente y servidor!!!
- ✍ No es el mejor modelo para proporcionar tolerancia a fallos o mejorar la disponibilidad.

#### 2.1.4.1. Introducción a la comunicación en grupo.

- ✍ Grupo: un conjunto de procesos que actúan juntos en cierto sistema. Cuando un mensaje se envía a un grupo todos los miembros lo reciben. Comunicación de uno-muchos.
- ✍ Los grupos son dinámicos: se pueden crear grupos y destruir grupos. Un proceso se puede unir a un grupo o dejar otro. Un proceso puede ser miembro de varios grupos.
- ✍ Objetivo: permitir que los procesos trabajen con una única abstracción.

✍ Su implementación depende del hardware, de la red. En las que disponen del multitransmisión o multidifusión (una dirección especial) es directa. Las que disponen de transmisión simple, un paquete con cierta dirección se envía a todas las máquinas. Se puede implantar comunicación en grupo transmitiendo desde el emisor una paquete a cada miembro del grupo (n miembros, n paquetes).

✍ Los mensajes de multidifusión permiten:

- ✍ Tolerancia a fallos basada en servicios duplicados. Un grupo de servidores.
- ✍ Búsqueda de los servidores para localizar servicios disponibles, registrar sus interfaces o para buscar las interfaces de otros servicios.
- ✍ Mejores prestaciones basada en datos replicados. Cada vez que se producen cambios, se comunican mediante multidifusión.

✍ Propagación de las notificaciones de eventos. Notificar a los procesos del



## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### 2.1.4.2. Aspectos de diseño.

#### Grupos cerrados vs. grupos abiertos

- ✍ Grupos cerrados: sólo los miembros del grupo pueden enviar hacia el grupo como un todo. Se suele utilizar para el procesamiento paralelo (tienen un único objetivo y no interactúan con el mundo exterior).
- ✍ Grupos abiertos: cualquier proceso del sistema puede enviar a cualquier grupo. Sistemas que soportan servidores duplicados.

#### Grupos de compañeros vs. grupos jerárquicos

- ✍ Estructura interna del grupo:
  - ✍ Todas las decisiones se toman de forma colectiva.
  - ✍ Un proceso es el coordinador y todos los demás trabajadores.

- ✍ grupo de compañeros: es simétrico y no es vulnerable. La toma de decisiones es más difícil.
- ✍ grupo jerárquico: rápido, la pérdida del coordinador es una catástrofe.

#### Pertenencia al grupo

- ✍ Necesitamos un método para crear, eliminar, asociar, dejar grupos.
- ✍ Servidor de grupos: mantiene un BD de todos los grupos y de sus miembros. Es directo, eficiente y fácil pero centralizado!!!.
- ✍ Sistema distribuido: un proceso envía un mensaje a todos los miembros del grupo para anunciar su presencia y también para despedirse.
  - ✍ Si un miembro falla, los demás deben descubrirlo de forma experimental.
  - ✍ La salida/entrada al grupo debe sincronizarse con el envío de mensajes.

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### Direccionamiento al grupo

#### ¿Cómo dirigirse a un grupo?

- ✍ Proporcionar una dirección única al grupo (un sistema que soporte multitransmisión se puede asociar a una dirección de multitransmisión).
- ✍ Si soporta transmisión simple: Cada núcleo recibe el mensaje y obtiene la dirección del grupo, si ninguno de los procesos es miembro del grupo se descarta.
- ✍ En otro caso, el núcleo de la máquina emisora debe contar con una lista de todas las máquinas que tienen procesos que pertenecen al grupo.
- ✍ Segundo método: el emisor tenga los destinos de forma explícita. No es transparente. Cuando cambia la pertenencia del grupo, los procesos deben actualizar sus listas.
- ✍ Tercer método: Direccionamiento de predicados. Se envía un mensaje a todos los miembros que cumplen un predicado. Si es verdadero, se acepta. (ej. enviar un mensaje a todas las máquinas con al menos 4M de M.)

### Primitivas send y receive

- ✍ Debemos abandonar el protocolo solicitud/respuesta.
- ✍ Send/Receive indican como parámetro una dirección de grupo, la comunicación tiene un único sentido.

### Atomicidad

- ✍ El mensaje llega a todos los miembros del grupo o a ninguno de ellos.
- ✍ No es fácil, es posible la sobreejecución de un receptor.
- ✍ Se puede realizar un reconocimiento para garantizarlo, pero es posible que el emisor falle. En este caso no queda evidencia de un mensaje pendiente.
- ✍ Algoritmo (Joseph y Birman, 1989). El emisor envía un mensaje a todos los miembros y activa los cronómetros. En caso necesario, se envía retransmisión. Cuando un proceso recibe un mensaje, si es la primera vez, envía el mensaje a todos los del grupo y activa cronómetros. De esta manera se garantiza envío a todos.

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### Ordenación de mensajes

- ✍ Un sistema debe tener una semántica bien definida con respecto al orden de entrega de los mensajes.
- ✍ Cuando dos procesos envían sendos mensajes se deben enviar en el mismo orden de petición. Si no hay diferencia apreciable se elige uno de ellos.

### Grupos Solapados

- ✍ Es difícil la coordinación con respecto a otros grupos.

ejemplo) Tenemos dos grupos: 1) procesos A, B y C; 2) con B, C y D.

- ✍ Los procesos A y D deciden simultáneamente enviar un mensaje a sus grupos y el sistema realiza la ordenación. Se podría dar la situación:

- ✍ A envía a B, D envía a B, D envía a C, A envía a C ? B, C reciben los mensajes en un orden distinto

### Escalabilidad

- ✍ ¿Qué ocurre cuando existen decenas de miembros en un grupo? ¿cuando no se pueden mantener en una única LAN?
- ✍ El número de transmisiones puede crecer de forma exponencial.
- ✍ Resulta extremadamente complicado mantener el orden de transmisión.

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

### 2.1.4.3. Comunicación en grupo de ISIS.

- ✍ ISIS: es un conjunto de herramientas para construcción de aplicaciones distribuidas. No es un SO sino un conjunto de programas que se ejecutan sobre un SO existente.

Idea fundamental es **sincronía**.

- ✍ Sistema síncrono: los eventos ocurren estrictamente de forma secuencial y sin retardo, no existe solapamiento entre procesos. Imposibles de construir!!!
- ✍ Sistema vagamente síncrono: los eventos tardan un tiempo finito pero los eventos aparecen en el mismo orden.
- ✍ Sistema virtualmente síncrono: las condiciones de orden se relajan pero sin que afecte al sistema.
- ✍ Dos eventos no relacionados son concurrentes. Son libres de realizarse en cualquier orden.

- ✍ Dos eventos que estén relacionados de manera casual, deben ser recibidos en el orden en que se produjeron.

### Primitivas de comunicación en ISIS

- ✍ ABCAST: proporciona la comunicación vagamente síncrona y se utiliza para transmisión de datos a los miembros de un grupo.
- ✍ CBCAST: proporciona comunicación virtualmente síncrona y se utiliza para envío de datos.
- ✍ GBCAST: parecido a ABCAST pero se usa en pertenencia.

## Tema 2.1.- Comunicación en Sistemas Operativos Distribuidos.

ABCAST: protocolo de compromiso en dos fases:

- El emisor A asigna una marca de tiempo ( $n^o$  secuencial) al mensaje y lo envía a todos los miembros del grupo.
  - Cada miembro elige una marca de tiempo (mayor que cualquier otra que haya recibido o enviado) y la envía de regreso a A.
  - Cuando A recibe todas las marcas, elige la mayor y envía un mensaje a todos los miembros con esa marca.
  - Los mensajes se entregan a los programas de aplicación según el orden de las marcas de tiempo.
- Este protocolo funciona pero es complejo y caro ? ISIS inventa CBCAST.

CBCAST: garantiza la entrega de entrega ordenada de los mensajes relacionados de manera casual.

Si el grupo tiene  $n$  miembros, cada proceso mantiene un  $n$ -vector. El  $i$ -ésimo componente representa el  $n^o$  del último mensaje recibido del proceso  $i$ . Los vectores los mantiene el sistema.

- Cuando un proceso tiene que enviar un mensaje, incrementa su casilla y envía este vector como parte del mensaje.
- Cuando un proceso recibe un mensaje examina si existe un proceso que le haya enviado un mensaje que no ha recibido. Si es así espera a recibir éste, en otro caso lo entrega.

