Sistemas Distribuidos

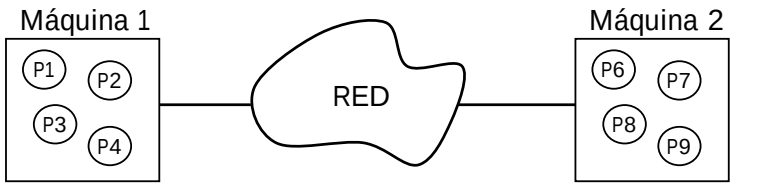
En este capítulo trataremos las cuestiones de comunicación a tener en cuenta en un sistema distribuido, no obstante lo haremos con un alto nivel de abstracción, considerando el modelo de comunicación entre los distintos componentes del sistema.

En primer lugar abordaremos los mecanismos para referenciar los componentes que se ofrecen un un sistema distribuido, esto es, la Denominación y Servicio de Nombres.

Seguidamente presentamos tres modelos concretos de comunicación apropiados para la comunicación entre los procesos que componen el sistema distribuido: las *llamadas a procedimientos remotos* (RPC's) de Sun, la *invocación a métodos remotos* (RMI) de Java y el modelo de CORBA.

La particularidad de estos mecanismos estriba en que mantiene a los procesos abstraídos del hecho de que forman un sistema distribuido, y que la comunicación entre ellos se realiza de igual manera a la comunicación con los procesos locales de cada máquina.

Los Modelos de Comunicación



Los procesos están separados → Lógicamente / Físicamente

Se requiere una línea de comunicaciones para

- Transferir Datos
- Sincronizarse

¿Cómo se comunican un grupo de procesos?	
	Una Pareja → Modelo CLIENTE-SERVIDOR
	A un Grupo → Modelo *MULTICAST*

Los componentes de un sistema distribuido no solamente están separados lógicamente, sino también físicamente, por lo que requieren líneas de comunicaciones para interaccionar.

Nosotros supondremos aquí que las aplicaciones y software básico de un sistema distribuido están construidos de tal forma que todos los componentes que requieren o proporcionan accesos a recursos están implementados como procesos.

Para que los procesos remotos implicados en un mismo trabajo puedan interaccionar, parece claro que se va a requerir una comunicación entre ellos para:

- Transferencia de datos
- Sincronización de operaciones o acciones.

Para la implementación de un sistema de paso de mensajes entre distintos ordenadores se requiere una red de comunicaciones con los consiguientes protocolos de comunicación para la transmisión de datos y señales de sincronización. Nosotros nos vamos a centrar únicamente en la semántica del alto nivel.

El mecanismo de comunicación que se va a utilizar para la comunicación entre procesos remotos va a ser el paso de mensajes, con la misma semántica que el correspondiente a los sistemas operativos centralizados. Es decir, que se va a disponer de primitivas de envío y recepción de mensajes, y que estas operaciones pueden ser síncronas o asíncronas (bloqueantes o no bloqueantes). Este mecanismo de comunicación por paso de mensajes recibe diversos nombres, tales como *canales*, s*ockets* o *puertos*.

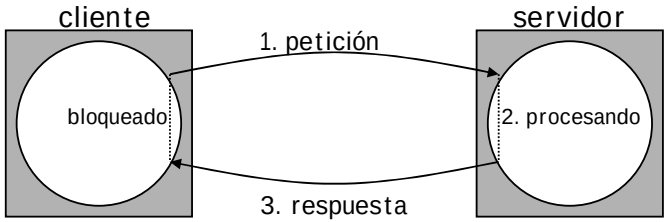
El rendimiento global de un sistema distribuido tiene una dependencia crítica de los mecanismos de los subsistemas de comunicaciones utilizados para la intercomunicación de procesos. Y no depende únicamente de la optimización de los niveles bajos de comunicaciones, sino de la implementación de la política o modelos de comunicaciones utilizados.

En los dos siguientes apartados vamos a presentar los dos modelos de comunicación más comúnmente utilizados en el diseño de sistemas distribuidos:

Modelo **cliente-servidor**, para comunicación entre parejas de procesos.

Modelo ***multicast***, para comunicación entre grupos de procesos cooperantes.

Modelo Cliente-Servidor



1. Cliente: Envío → bloqueado
2. Servidor: Recibe → procesa → contesta
3. Cliente: Recibe respuesta → continúa

RPC

Imprimir (Fichero,Impresora,ok)

Averiguar_Nodo_Impresor
Envía (Fichero, Nodo_Impresor)
Recibir (Respuesta, Nodo_Impresor)

¿Cómo se conocen los clientes y los servidores?	LOS SERVIDORES SON DINÁMICOS Los servidores deben registrarse con un nombre de servicio predefinido LOS SERVIDORES NO CONOCEN A LOS CLIENTES Los clientes deben incluir su id. en la petición

Clientes Y Servidores

¡NO son máquinas SON PROCESOS ambivalentes!

El **modelo cliente-servidor** está orientado a la provisión de servicios. Una conversación típica consiste en:

1. El proceso cliente transmite una petición al proceso servidor.
2. El proceso servidor ejecuta la petición solicitada.
3. El proceso servidor transmite la respuesta al cliente.

Este modelo implica la transmisión de dos mensajes y alguna sincronización entre el cliente y el servidor. Es decir, el proceso servidor debe estar pendiente de la llegada de peticiones del cliente, y a su recepción debe ejecutar el servicio solicitado y comunicar la respuesta. El proceso cliente por su parte, una vez enviada la petición en el paso 1, debe esperar bloqueado hasta la recepción de la respuesta después del paso 3.

Este modelo de comunicaciones puede implementarse directamente mediante el mecanismo de paso de mensajes (operaciones *enviar* y *recibir*) del sistema operativo, pero normalmente se utiliza una construcción del nivel del lenguaje que le abstrae al programador de las operaciones de envío-espera-recepción. Esta construcción, que veremos en detalle más adelante, en este capítulo, se conoce como **RPC** (*Remote Procedure Call*) o **Llamada a Procedimiento Remoto**, y esconde las operaciones de envío y recepción bajo el aspecto de la llamada convencional a una rutina o procedimiento.

Estamos viendo cómo se comunica un cliente con un servidor, pero hay una cuestión que aclarar: ¿cómo han llegado a conocerse el cliente y el servidor para iniciar una comunicación?

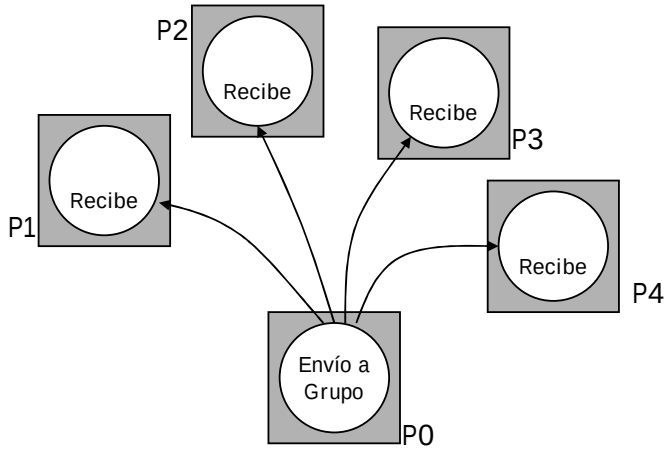
Los procesos clientes no pueden programarse, a priori, con los identificadores de comunicación de todos los servidores posibles de la red, por lo que se requiere algún mecanismo que permita un enlace dinámico. Este mecanismo normalmente consiste en que cuando un proceso servidor arranca, él mismo se registra en un servidor de nombres indicando su dirección y el nombre predefinido del servicio que proporciona. Cuando un cliente requiere un servicio, le pregunta a un servidor de nombres por la dirección de algún servidor que ofrezca tal servicio, obteniendo así su identificador de comunicación para enviarle la petición.

Por su parte, un proceso servidor, durante su vida, atiende a muchos procesos clientes distintos sin tener conocimiento previo de ellos, por lo que cada petición debe incluir el identificador de comunicación del proceso cliente que realiza la petición, para que así le pueda responder.

Debe tenerse en cuenta que un proceso dado no tiene por qué ser exclusivamente cliente o servidor. Un proceso se comporta como cliente o servidor en un intercambio concreto de información o servicio, pero un servidor puede solicitar servicios de otro servidor, convirtiéndose así en cliente, mientras que en un momento dado un cliente puede estar prestando un servicio a otro proceso, convirtiéndose a su vez en servidor.

Modelo *Multicast*

MULTICAST	Envío de múltiples copias de un mismo mensaje desde un proceso origen a múltiples procesos de destino.



APLICACIONES DE MULTICAST	☞ Búsqueda de un recurso ☞ Tolerancia a fallos ☞ Actualizaciones múltiples

La velocidad del envío multicast depende del algoritmo utilizado y del soporte hardware disponible. → Rendimiento General del Sistema

El modelo de comunicación ***multicast*** consiste en el envío de un mismo mensaje desde un origen a un grupo de nodos de destino. No se debe confundir con *broadcast* (o difusión), que es el envío de un mensaje de forma que pueda ser escuchado por todos los nodos de la red, y que en sistemas distribuidos se utiliza con menor frecuencia.

Puede haber varios motivos para enviar un mismo mensaje a un grupo de nodos:

Búsqueda de un objeto o recurso. Un cliente puede enviar un mensaje a un grupo de procesos servidores, y el que realmente tenga el objeto buscado será el único en contestar.

Tolerancia a fallos. Un cliente solicita un servicio muy importante mediante multicast. Uno o más servidores procesan la petición y contestan. El cliente toma la primera respuesta y deshecha las posteriores. De esta manera, se asegura una respuesta aunque falle un servidor.

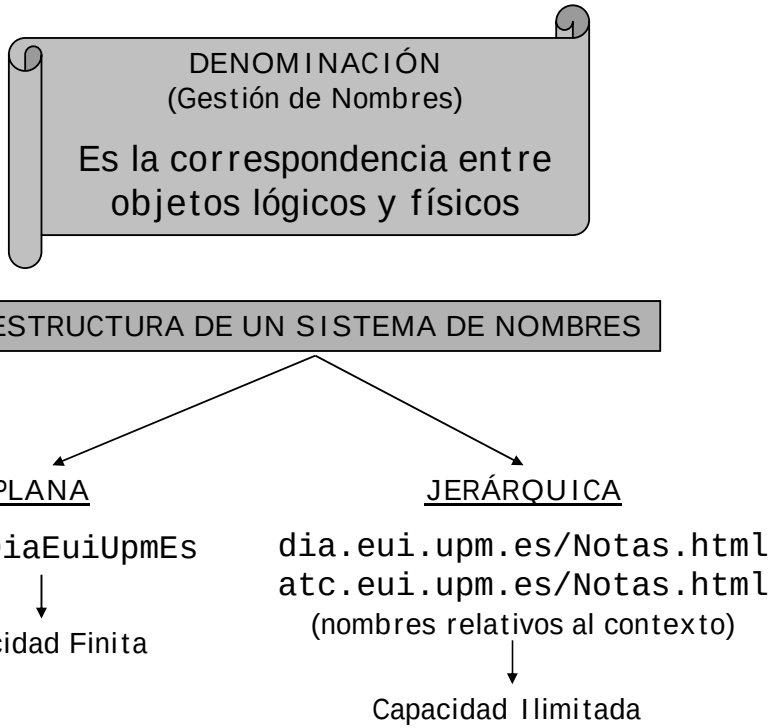
Actualizaciones. Cuando se quiere actualizar información común entre varios nodos –como tablas de encaminamiento o la hora– el proceso encargado del mantenimiento se lo envía a los demás mediante *multicast*.

Hay diversos algoritmos o métodos para el envío de mensajes en modo multicast, desde el mero envío secuencial de un mensaje tras otro desde el nodo origen, o la técnica de la inundación, hasta los basados en árboles de expansión, pero su estudio no está entre los objetivos de esta asignatura. No obstante sí debemos tener en cuenta que la elección del algoritmo de envío utilizado puede repercutir seriamente en el rendimiento general del sistema.

También puede ayudar a mejorar la velocidad de envío el disponer de cierto soporte hardware que, para el algoritmo que le convenga, posibilite el envío paralelo de las múltiples copias del mensaje.

El sistema de alto nivel que pueden utilizar los procesos para difundir mensajes a grupos puede estar basado también en primitivas RPC, aunque con algún mecanismo adicional en la primitiva de envío para poder indicar las direcciones a las que va dirigido el mensaje (por ej. una tabla con todas las direcciones o direcciones de grupo).

Denominación



Capacidad de Nombres ≠ Capacidad de Identificadores		
Internet	Capacidad de Identificadores:	Limitada
	Capacidad de Nombres:	Infinita

La **denominación**, o gestión de nombres, es la correspondencia entre objetos lógicos y físicos. Por ejemplo, un usuario trata con conjuntos de datos representados por nombres de ficheros, mientras que el sistema gestiona bloques físicos de datos almacenados en pistas de un disco. Normalmente el usuario se refiere a un fichero por un **nombre** textual, el cual posteriormente se traduce a un **identificador** numérico que acaba refiriéndose a bloques de un disco. Esta correspondencia entre los dos niveles proporciona a los usuarios una abstracción de cómo y dónde están realmente almacenados los datos.

En un sistema distribuido hay que añadir una nueva dimensión a la abstracción: ¿En qué lugar (máquina) de la red está el objeto referenciado?

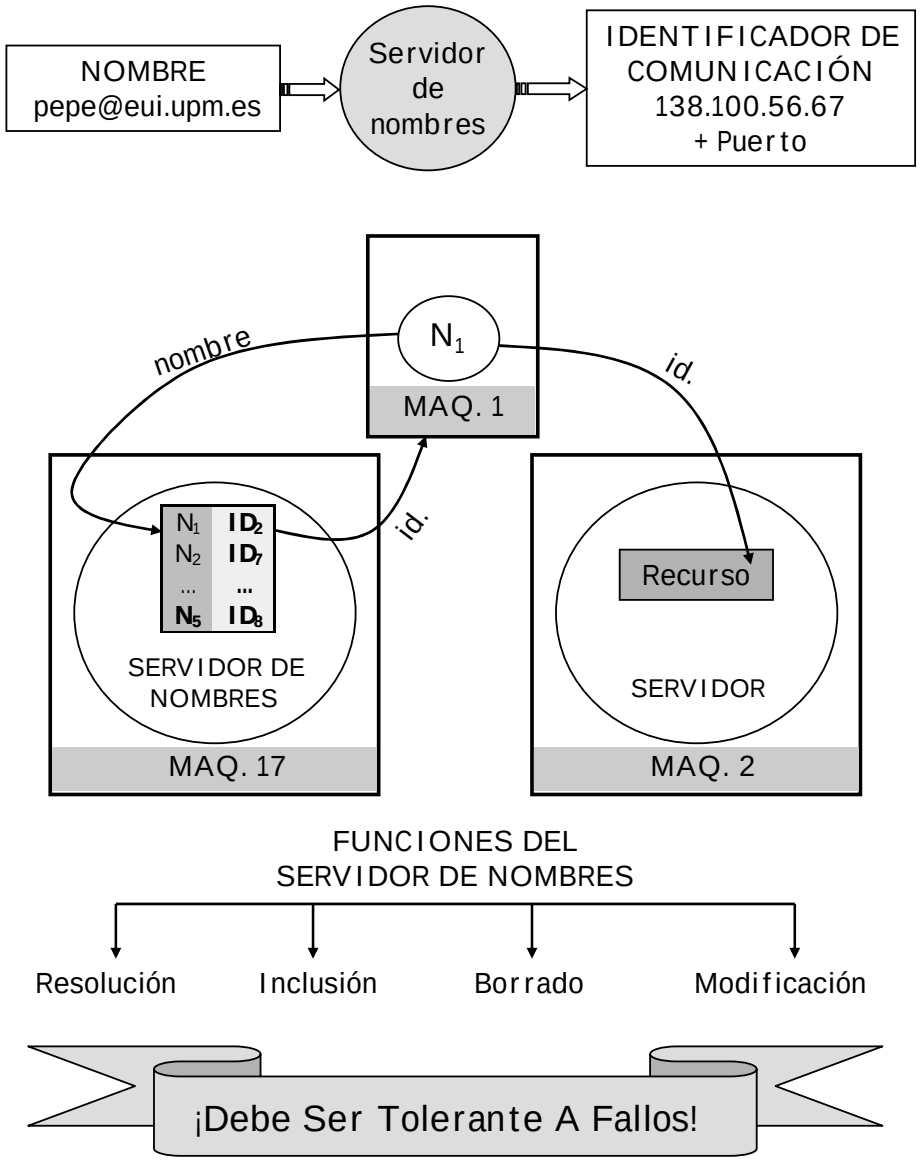
Capacidad y estructura del esquema de nombres . Veamos ahora cómo puede ser el espacio de nombres en cuanto a su capacidad y estructura.

El Espacio de Nombres puede tener una **capacidad** *Limitada* o *Infinita*. El actual espacio de las direcciones de Internet es un ejemplo de capacidad limitada (ej. 138.100.56.34).

En cuanto a su **estructura**, puede ser *Plana* o *Jerárquica*. La estructura plana de nombres está asociada a espacios de nombres de capacidad finita (a no ser que la longitud de los nombres sea ilimitada), mientras que en la estructura jerárquica las direcciones pueden crecer indefinidamente. Cuando se utiliza la estructura jerárquica, se dice que la resolución de nombres se realiza de “acuerdo al contexto”, es decir, traduciendo cada uno de los nombres anteriores al nombre final que indican la jerarquía por la que hay que pasar hasta llegar al objeto concreto. Ejemplo: `dia.eui.upm.es/notas.html` hace referencia a un fichero (`notas.html`) situado en la máquina `dia` de la `eui` de la `upm`. Para resolver tales nombres se va ascendiendo por esta jerarquía de nombres, de tal forma que en cada nivel se es capaz de resolver el nombre y obtener la dirección del siguiente nivel hasta llegar a la máquina de destino, y en ella obtener el objeto con el nombre indicado (`notas.html`).

No se debe confundir la capacidad de nombres con la capacidad de identificadores de dirección. Así, por ejemplo, aunque el actual sistema de direcciones de Internet es finito (en número de máquinas), el número de algunos recursos referenciables en la red es ilimitado, puesto que cada máquina puede contar con una estructura jerárquica de ficheros potencialmente ilimitada (salvo por la capacidad y limitaciones de tablas).

...Denominación



Podemos ver, entonces, la necesidad de la resolución o traducción de nombres por identificadores de dirección. Una posibilidad sería que cada programa o sistema operativo de un sistema distribuido se programara directamente con las direcciones de todos los objetos actuales y futuros de la red completa, pero no resultaría muy práctico, pues no es fácil conocer, a priori, todos los posibles objetos de una red, y sería imposible realizar cualquier cambio de dirección. Parece mucho más razonable ver que esta resolución de nombres no es más que un nuevo servicio que debe ofrecer el sistema a los clientes.

Este nuevo servicio de resolución de nombres se denomina **Servidor de Nombres** (en inglés también se le conoce como *binder*, ya que a la traducción de nombres se le denomina *binding*). Así, cuando un cliente necesite conocer la dirección de cualquier servidor, lo único que tiene que hacer es preguntárselo al servidor de nombres. Desde luego, el servidor de nombres residirá en alguna máquina de dirección bien conocida.

Funciones del Servidor de Nombres. Ya hemos visto que la función básica del servidor de nombres es la resolución o traducción de un nombre a un identificador, pero requiere otros servicios adicionales:

Resolución: La traducción del nombre por el identificador de comunicación.

Inclusión: Añadir una pareja nombre/identificador al servidor de nombres.

Borrado: Eliminar una entrada del servidor de nombres.

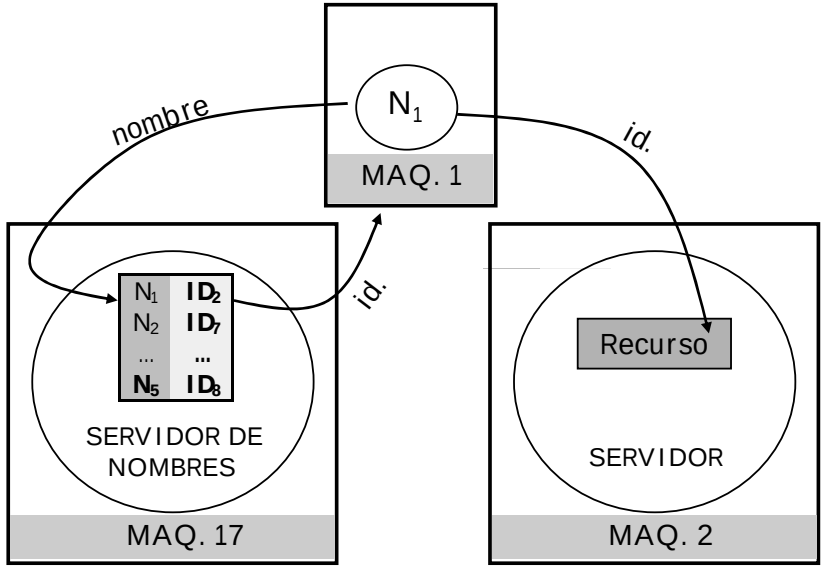
Modificación: Modificación del nombre/identificador de una entrada.

Ya hemos comentado que cuando un cliente requiere cualquier servicio del sistema distribuido, primero es necesario comunicarse con el servidor de nombres para obtener el identificador de un servidor del servicio requerido. ¡Y si el servidor de nombres falla o se cae? La respuesta es clara: Se pierden todos los accesos a los servicios del sistema.

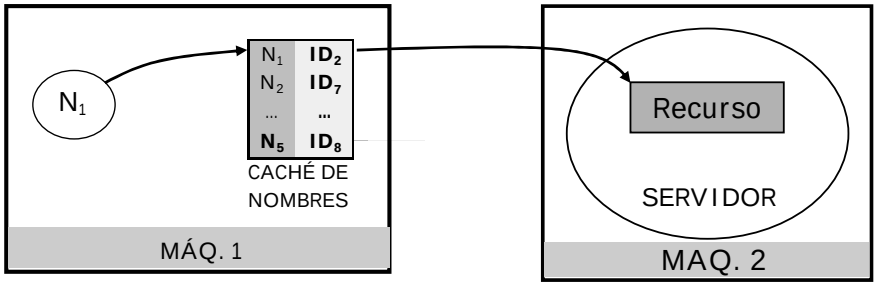
Dada la importancia del servidor de nombres, cuyo funcionamiento es vital para el resto del sistema, parece que se hace necesario que este servicio especial sea **tolerante a fallos**. Teniendo en cuenta, además, que va a ser un servicio muy requerido, pues todas las utilizaciones de cualquier servicio deben pasar primero por él, para facilitar la tolerancia a fallos y evitar el cuello de botella, suele ser normal que el servicio de nombres esté formado por servidores replicados que ofrezcan este servicio de nombres.

...Denominación

Para comunicarse un cliente con un servidor, ANTES debe comunicarse con un servidor de nombres



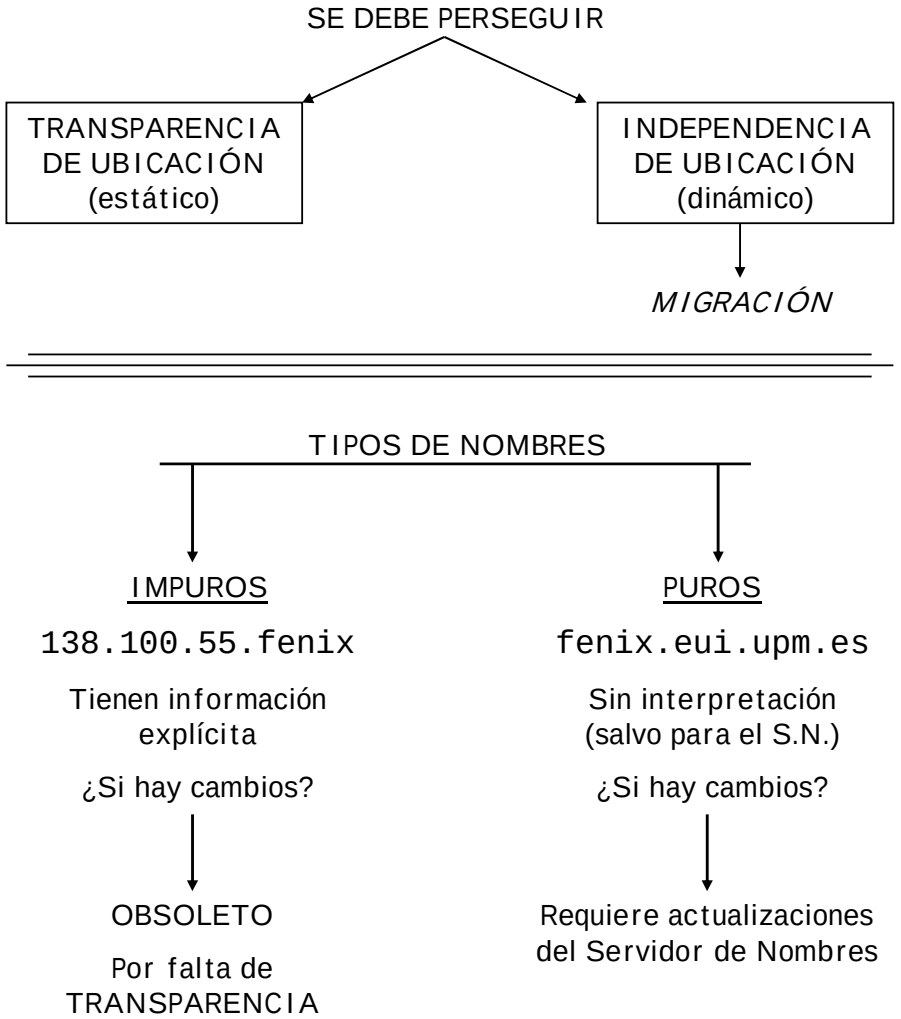
En accesos frecuentes a un objeto	Para evitar tiempos de resolución de nombres
	Utilizar CACHÉ LOCAL EN EL CLIENTE



Para acceder a un objeto remoto, el proceso cliente (que sólo conoce el nombre del recurso) debe conseguir el identificador de comunicación del recurso que solicita antes de comunicarse realmente con él. Para ello debe acudir primero a los servidores de nombres intermedios necesarios hasta conseguir dicho identificador de comunicación, con el consiguiente tiempo de demora debido a los tiempos de resolución o traducción de cada uno de los servidores de nombres requeridos.

Cuando se está accediendo a menudo a un objeto remoto, para evitar el tiempo de resolución de los nombres intermedios, el proceso cliente puede mantener una **tabla caché** con los identificadores de dirección de los objetos más recientemente referenciados, y utilizar directamente estos identificadores para acceder a los objetos.

...Denominación



A la hora de diseñar un sistema de nombres o de denominación, se deben perseguir estos dos objetivos:

- **Transparencia de ubicación**. El nombre de un objeto no debe revelar su ubicación física.
- **Independencia de ubicación**. El nombre del objeto no debe cambiar cuando cambie su ubicación física. Esto implica transparencia dinámica, ya que un nombre puede asociar el mismo objeto a lugares diferentes en momentos distintos.

Actualmente, la mayoría de los sistemas proporcionan simplemente la transparencia de ubicación, por lo que no ofrecen ***migración***, es decir, el cambio **automático** de ubicación de un objeto sin afectar a sus usuarios o clientes. *Chorus* y *Charlotte* son ejemplos de sistemas que permiten la migración.

Nombres Puros e Impuros. Los nombres **puros** son simplemente series de bits sin ninguna interpretación posible (salvo para el servidor de nombres). Otros nombres (los impuros) incluyen bits que indican directamente una dirección, permisos de acceso o cualquier otra información sobre el objeto.

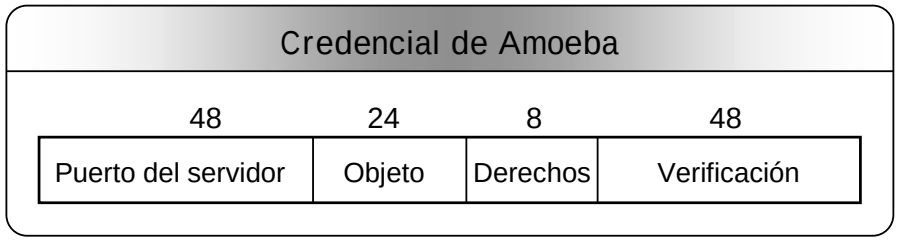
Los nombres **impuros** entran en conflicto con el principio de transparencia al que tanto hemos aludido. Obsérvese que con un nombre impuro, cualquier información implícita que lleve, puede quedarse obsoleta si el recurso al que se refiere cambia su dirección, permisos, etc. Con los nombres puros simplemente hay que preocuparse de mantener actualizadas las bases de datos de los servidores de nombres de cada contexto.

...Denominación

Si es fácil reproducir el id. de un proceso sin pedirlo al servidor de nombres → Puede haber accesos no autorizados

El Servidor de Nombres debe comprobar la identidad del cliente antes de proporcionar el id.

CREDENCIALES (*Capabilities*)



Control de acceso. Para evitar accesos no autorizados a los recursos del sistema, un primer paso consiste en hacer que el identificador de un recurso no se pueda obtener fácilmente a partir de su nombre si no es a través del servidor de nombres. Y, por supuesto, el servidor de nombres debe ocuparse de comprobar la identidad del cliente que solicita una resolución de nombres antes de darles el identificador solicitado. Los identificadores que se comportan así, se denominan **credenciales** *(capabilities)*. Por eso se dice que para poder acceder a un recurso o servicio, previamente debe obtenerse la credencial correspondiente.

Ejemplo: **Credencial de Amoeba**. Cuando un cliente requiere cierto servicio, en primer lugar se identifica y solicita la credencial correspondiente al servidor de nombres, el cual devuelve la credencial solicitada para el cliente identificado. Una vez se tiene la credencial, se obtiene de ella el *Puerto* del servidor que va a prestar el servicio requerido, con lo que ya se le puede enviar el mensaje con la petición del servicio y la credencial completa.

El campo *Objeto* lo utiliza el servidor para identificar el objeto específico con el que el cliente quiere realizar alguna operación. Para el caso de un archivo, este campo sería algo parecido a un i-nodo de Unix.

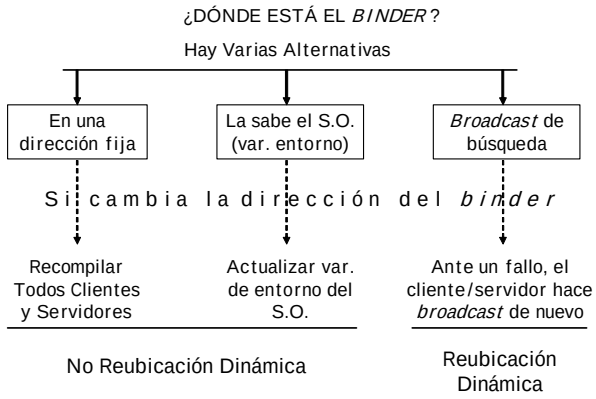
Los *Derechos* están compuestos por una serie de bits que indican las operaciones que le están permitidas al usuario para ese objeto (por ej. lectura, escritura, ...).

El campo *Verificación* se utiliza para dar validez a la credencial. La verificación la establece el servidor de nombres mediante un cierto algoritmo en función del resto de los campos de la credencial, y el servidor del objeto comprueba si la verificación que le llega efectivamente es la correspondiente a esa credencial. De ser así, y si cuenta con los derechos apropiados realiza la operación solicitada; en caso contrario, devolverá algún código de error al cliente.

De esta manera se evita que cualquier proceso de la red solicite indiscriminadamente cualquier operación, pues las credenciales solamente las pueden construir los servidores de nombres, y solamente mediante éstas puede solicitarse operaciones a los servidores.

Sistemas Distribuidos La Comunicación - 9

La Comunicación ...Denominación



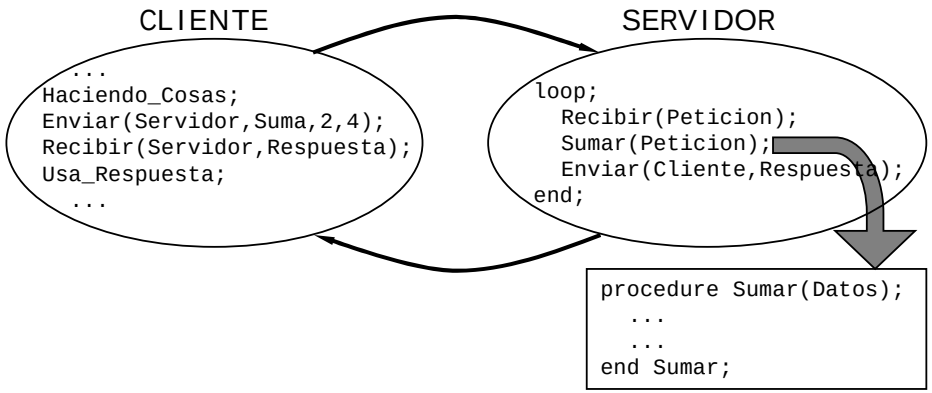
Sistemas Distribuidos La Comunicación - 10

¿Dónde está el *binder*? El *binder* nos proporciona la dirección del servidor solicitado, pero ¿cómo se lo pedimos al *binder*? ¿dónde está el *binder*?

Se tienen las siguientes alternativas:

- Ubicar siempre al *binder* en un ordenador de dirección bien conocida, e incluirla al compilar todos los clientes y servidores. Si se reubica el *binder*, se deben recompilar todos los clientes y servidores con la nueva dirección. Obviamente, no es posible la reubicación dinámica del *binder*.
- Que los sistemas operativos de los clientes y de los servidores sean responsables de proporcionar dinámicamente a sus procesos la dirección del *binder*. Esto puede hacerse mediante variables de entorno. Si se reubica el *binder*, debe actualizarse la dirección en la variable de entorno del sistema operativo, e informarse a los usuarios para que rearranquen los procesos. Con este enfoque se permite la reubicación ocasional del *binder* sin tener que recompilar todos los clientes y servidores; normalmente, solo obliga a pararlos y rearrancarlos de nuevo.
- Cuando un cliente o un servidor arranca, difunde un mensaje por toda la red para localizar al *binder*. Cuando un proceso *binder* reciba este mensaje, contesta al emisor con su dirección. Mediante este sistema, el *binder* puede ejecutarse siempre sobre cualquier ordenador y reubicarse dinámica y fácilmente.

Las RPC's



¡Pero entonces el cliente debe saber:

- si el servicio es local o remoto?
- la dirección del servidor que necesita?
- el formato de los datos en el servidor?

¿Dónde está la TRANSPARENCIA?

SOLUCIÓN

Remote
Procedure
Call

Aunque no vendría mal una descripción más detallada de estos protocolos de comunicación, debido a la limitación del tiempo disponible, dejaremos que se ocupen de esto las asignaturas de comunicaciones. Aquí nos centraremos en los mecanismos de alto nivel de comunicación entre procesos, suponiendo para ello que se dispone de un canal fiable de comunicaciones, y basándonos simplemente en el modelo cliente-servidor.

Ya hemos visto que la comunicación entre cliente y servidor está basada en el protocolo petición-respuesta. Es decir, el cliente invoca la petición del servicio enviando un mensaje al servidor; el servidor realiza el servicio requerido y le devuelve la respuesta al cliente en otro mensaje. El cliente siempre debe esperar el mensaje de respuesta del servidor antes de continuar su ejecución, incluso aunque no espere ningún resultado (pues podría haber habido algún error).

Estas operaciones se realizan mediante las conocidas primitivas de envío y recepción de mensajes. Por lo que podemos apreciar hasta ahora, para realizar un servicio local, normalmente se utilizan las tradicionales llamadas a procedimientos, mientras que si el servicio requerido es remoto, hay que acudir a las primitivas disponibles de envío y recepción de mensajes.

Se nos presentan varios problemas:

- El cliente debe saber si el servicio que necesita es local o remoto antes de solicitarlo.
- En caso de que el servicio se realice de forma remota, el cliente debe conocer la dirección del servidor.
- El cliente debe saber la disposición exacta de los parámetros en el mensaje que debe enviar al servidor, así como el formato del mensaje de respuesta.

Parece que este tipo de servicio adolece de la principal propiedad de los sistemas distribuidos: la transparencia.

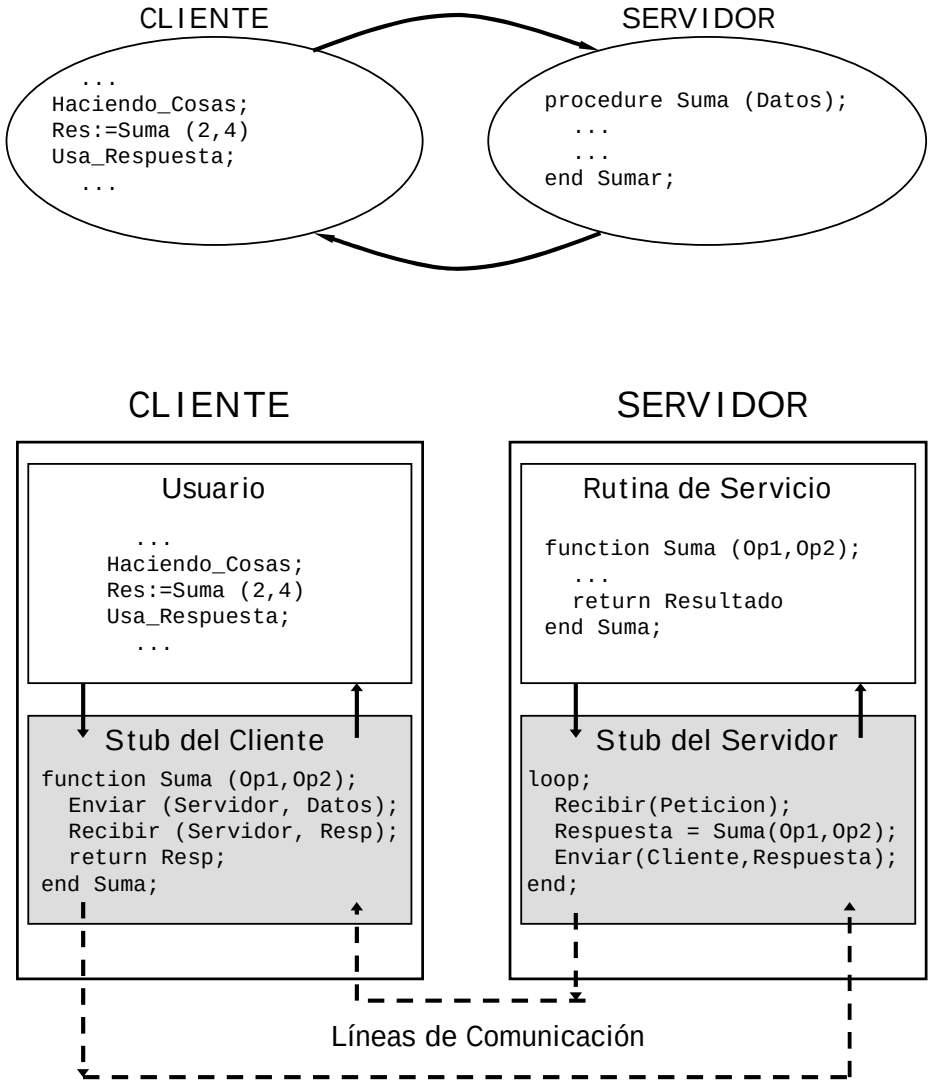
Esto hace pensar que sería bueno aislar o esconder el mecanismo de comunicación de los procesos de su implementación. Con esto se conseguiría:

- El programador podría concentrarse en escribir programas que solicitan servicios, independientemente de si la ejecución va atener lugar en un entorno centralizado o distribuido.
- Ni el el programador ni el cliente necesitan conocer la dirección de la máquina en la que reside el servidor en el momento de la petición.
- Tampoco necesita el cliente conocer el tipo de máquina o sistema operativo del servidor para ubicar convenientemente los parámetros en el mensaje.

Para conseguir esto, el cliente solamente debe realizar llamadas a procedimientos, de tal forma que dependiendo de cada caso, la ejecución del servicio solicitado tendrá lugar en la propia máquina o en una remota.

El mecanismo mediante el cual la ejecución de un procedimiento tiene lugar en una máquina remota se denomina **Llamada a Procedimiento Remoto** o **RPC** (*Remote Procedure Call*).

Estructura de las RPC



Una gran parte de las operaciones en los sistemas distribuidos son remotas, por lo que un proceso debe enviar un mensaje a otro proceso y esperar su respuesta.

Ya que los programadores están muy familiarizados con el concepto de llamada a procedimiento, estaría bien esconder los detalles de envío y recepción de mensajes detrás de la fachada de la elegante interfaz de un procedimiento.

Las llamadas a procedimientos remotos tienen la misma semántica que las llamadas a procedimientos ordinarios, es decir, al realizar la llamada, el llamante le pasa el control al procedimiento llamado, y lo recupera cuando el procedimiento llamado le devuelve el resultado. **Las RPC no son más que operaciones remotas disfrazadas de una interfaz procedural**.

El concepto de Llamada a Procedimiento Remoto lo presentaron por primera vez Birrell y Nelson en 1984 para solucionar los problemas del paso de datos entre máquinas y sistemas heterogéneos (que trataremos más adelante).

Los componentes del mecanismo de una RPC se muestran en la transparencia. La aplicación cliente llama a una subrutina que se ejecuta en otra aplicación: la rutina de servicio en el servidor. Si las aplicaciones cliente y servidor formaran parte del mismo proceso, el cliente llamaría directamente a la rutina que ejecuta el servicio; sin embargo, aquí el cliente tiene que llamar a otra subrutina que denominaremos ***stub* del cliente**. Esta subrutina tiene exactamente la misma interfaz que la rutina de servicio del servidor (la que realiza realmente el servicio requerido por el usuario), pero está implementada por código que solicita al servidor que ejecute la rutina del servicio, y que a continuación le devuelve al cliente los resultados que recibe del servidor.

Primero el *stub* del cliente copia los parámetros de la pila al mensaje de petición, y a continuación le pide al módulo de comunicación que envíe el mensaje de petición al servidor. El mensaje se recibe en el servidor por una rutina denominada ***stub* del servidor**. El *stub* del servidor toma los parámetros del mensaje de petición, los pone en la pila y termina por llamar a la verdadera rutina remota que realiza el servicio solicitado por el cliente. Cuando esta rutina finaliza y devuelve el resultado, se repite la misma operación, pero en sentido contrario: El *stub* del servidor pone los parámetros de resultado en un mensaje de respuesta y, mediante primitivas de comunicación, le envía el mensaje al *stub* del cliente. Éste saca el resultado del mensaje y se lo devuelve como parámetro de salida al cliente que lo llamó.

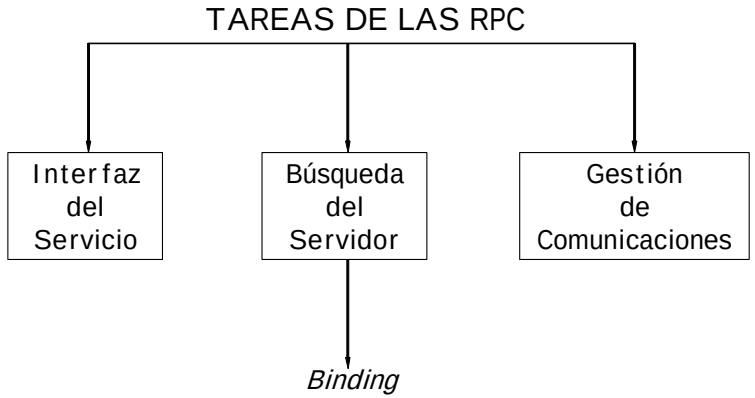
Si nos fijamos en el código del cliente, tiene exactamente el mismo aspecto que una llamada normal a procedimiento.

Como ya apuntamos en la transparencia anterior, la comunicación entre cliente y servidor no resulta tan sencilla como la simplificación que hemos hecho aquí. Veamos en las siguientes transparencias todas las tareas de las que debe ocuparse la RPC.

Cometido de las RPC

El Modelo de las RPC

Cometidos de las RPC



Sistemas Distribuidos La Comunicación - 13

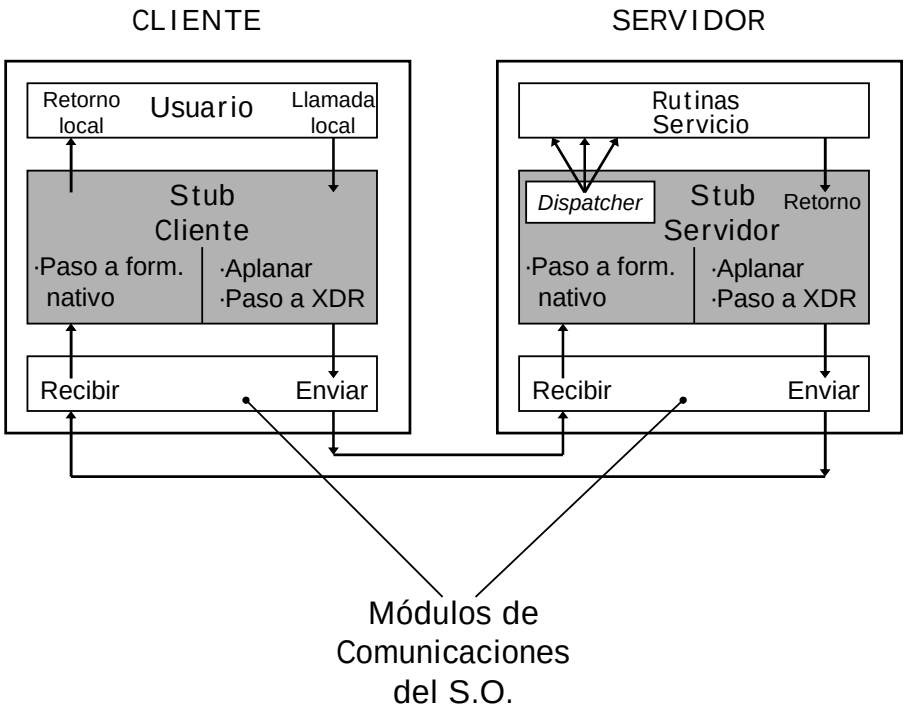
El software que soporta las llamadas a procedimientos remotos debe ocuparse de tres importantes tareas:

- **La Interfaz del Servicio**. Es decir, la integración del mecanismo de la RPC con los programas del cliente y del servidor, estando éstos escritos en lenguajes de programación convencionales. Esto incluye la serialización (*marshalling* y *unmarshalling*) de los parámetros que se pasan entre el cliente y el servidor, así como el establecimiento, en el servidor, de un mecanismo (*dispatcher*) que reparta las peticiones entre las rutinas de servicio apropiadas.
- **Búsqueda del Servidor**. Este proceso es el mecanismo que ya hemos visto, conocido como *binding*, que consiste en asignar el servidor apropiado más conveniente para el servicio que requiere el cliente.
- **Gestión de la Comunicación**. Esta tarea consiste simplemente en la transmisión y recepción de los mensajes de petición y de respuesta.

Puesto que la búsqueda del servidor ya la hemos tratado en las transparencias previas, en las siguientes pasaremos a ver con cierto detalle la Interfaz del Servicio. En cuanto a la gestión de la comunicación, ésta se basa directamente en servicios de comunicación ofrecidos por el sistema operativo, por lo que aquí nos centraremos en el tratamiento de errores, tanto de comunicación como de los fallos en el cliente o servidor.

Interfaz del Servicio

Las RPC de un servicio deben construirse a partir de la definición de su interfaz



La Interfaz del Servicio

La definición de la interfaz de un servicio es la base sobre la que se construye el resto del software que necesitan los programas del cliente y del servidor para permitir la llamada a procedimientos remotos.

En el sistema completo va a haber un proceso cliente y un proceso servidor, y cada uno consta de los siguientes elementos:

Proceso Cliente:
- Programa del usuario
- *Stub* del cliente

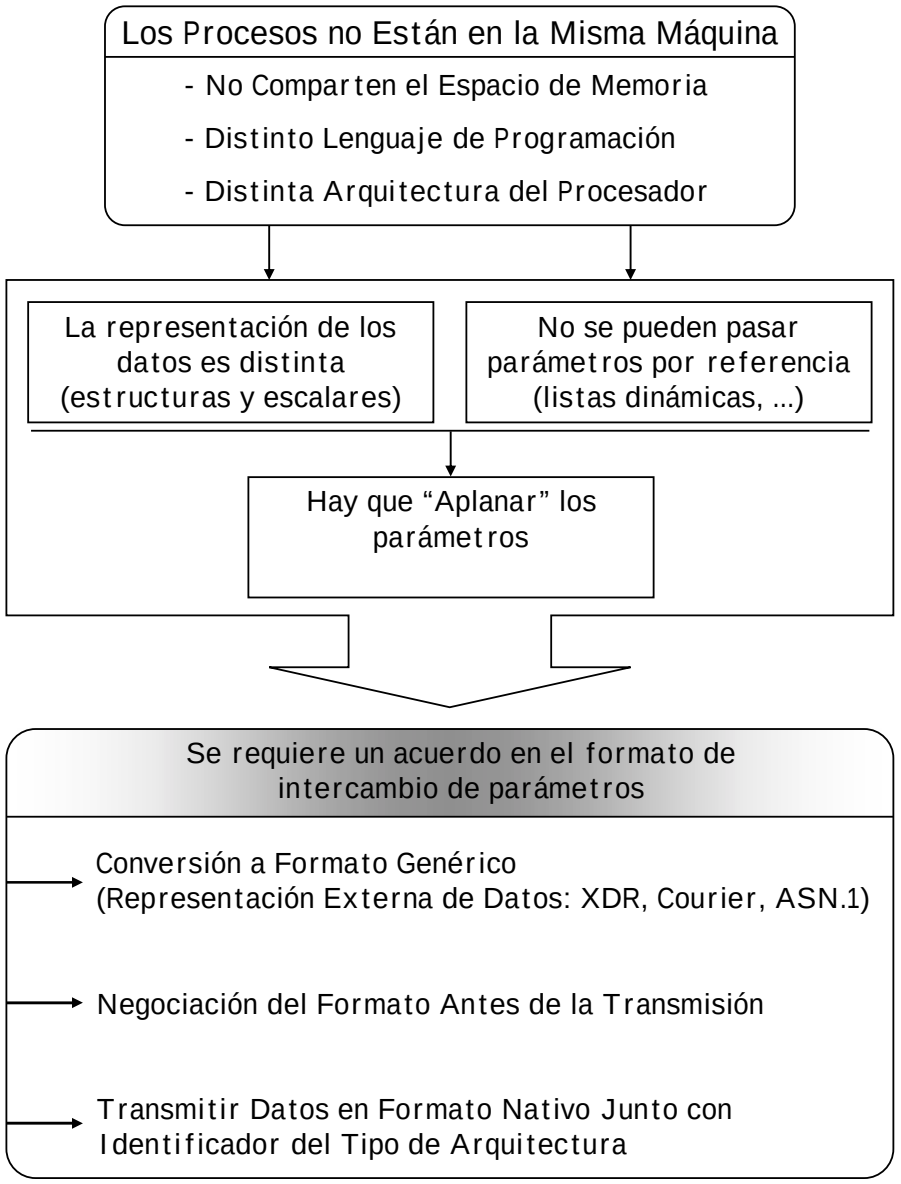
Proceso Servidor:
- *Stub* del servidor (incluye el repartidor de peticiones o *dispatcher*)
- Rutinas de servicio del servidor (las que realizan la operación solicitada).

Como se puede ver en el gráfico, ambos *stubs*, del cliente y del servidor, se apoyan en los servicios de comunicaciones que ofrece el sistema operativo, y uno de sus cometidos básicos es el paso de parámetros entre cliente y servidor. (En la siguiente transparencia trataremos el paso de parámetros)

Esta tarea encargada de la interfaz debe preocuparse de la construcción del programa cliente y del programa servidor. Para ello, debe generar el software apropiado y juntar todos los componentes para formar los dos programas.

El programa del usuario y las rutinas de servicio del servidor vienen dadas, así como el módulo de comunicación, que lo proporciona el sistema operativo. Falta por construir los *stubs* del cliente y del servidor. Veremos entonces, después del paso de parámetros, cómo se realiza la generación de *stubs*, también conocida como generación de interfaces.

El Paso de Parámetros



Ya hemos visto que una de las funciones del *stub* del cliente es coger los parámetros de entrada, ponerlos en un mensaje y enviarselos al *stub* del servidor. Sin embargo se presentan algunos problemas para hacer esto.

En primer lugar debe tenerse en cuenta que los datos en los programas suelen estar representados mediante estructuras de datos, mientras que en un mensaje la información debe ser una secuencia de bytes; por lo tanto, se debe establecer un mecanismo para "aplanar" o serializar los parámetros y pasarlos al mensaje. "Aplanar" significa tomar cualquier estructura de datos y reordenarla como una secuencia de bytes. Esto debe hacerse también con las estructuras de datos dinámicas (listas enlazadas), que deben pasarse a una serie de valores formando una secuencia de bytes.

Aunque el "aplanamiento" (*marshalling*, en inglés) de los parámetros podría ser una tarea fácil, no lo es. Por desgracia, los lenguajes utilizados para programar los distintos procesos de la red y los procesadores en los que se ejecutan, utilizan **diferentes representaciones de los datos** que manipulan (distinto tamaño y representación para los enteros –complemento a uno o complemento a dos–, distintas fronteras de alineamiento, distintos códigos de caracteres –ASCII o EBCDIC–, ...). Esto quiere decir que si un dato lo aplanamos en un proceso, lo ponemos en un mensaje y se lo enviamos a otro proceso de la red (en una máquina distinta), es muy posible que el formato en el que se reciben los datos no sea el que espera el receptor, por lo que el contenido del mensaje podría no entenderse o ser malinterpretado.

Un motivo de la interpretación errónea de datos por distintas arquitecturas es el debido a la **representación de los números en la memoria**. Las tiras de caracteres están compuestas por caracteres sucesivos, tal que cada uno de ellos ocupa un byte, y el carácter siguiente va a continuación en el siguiente byte de memoria. Pero para los números, por ejemplo los enteros, que suelen ocupar 32 bits (4 bytes), hay dos formas de ordenar los cuatro bytes:

Little-endian: El byte menos significativo del entero está en la dirección más baja. (Intel)
Big-endian: El byte menos significativo está en la dirección más alta. (Motorola, SPARC).

Para evitar este problema del formato de representación de los datos, **los procesos comunicantes deben ponerse previamente de acuerdo en el formato** en el que se van intercambiar los datos. Hay tres alternativas en el tipo de acuerdo a tomar:

- Antes de la transmisión, los datos se convierten a un formato genérico conocido por todos los procesos (representación externa de datos). En el receptor se convierten al formato local de su arquitectura. Algunos estándares de representación externa de datos son: XDR de Sun, Courier de Xerox y ASN 1.
- Para la comunicación entre dos ordenadores con arquitectura común, el paso anterior puede omitirse. Esto requiere que antes de la transmisión de los parámetros (al establecer la sesión de comunicación), los dos extremos negocien si se requiere pasar los datos a un formato genérico o no.
- Otra posibilidad consiste en transmitir los datos en su formato nativo (el de la arquitectura del transmisor) junto con un identificador del tipo de arquitectura subyacente. El receptor, consultando el tipo de arquitectura utilizado, decide si es necesario convertir los datos recibidos o no.

Un Ejemplo

Interfaz de las Rutinas de Servicio del Servidor y del Stub del Cliente

```
package Reloj is
  type Tiempos is record
    Hora:     t_Horas;
    Minutos:  t_Minutos;
    Segundos: t_Segundos;
    Dia:      t_Dias;
    Mes:      t_Meses;
    Año:      t_Años;
    Zona:     t_Zonas;
  end record;
  type Operaciones is (SET_TIME, GET_TIME);

  procedure Pedir_Hora (t: out Tiempos) return Status;
  procedure Poner_Hora (t: in  Tiempos) return Status;
end Reloj;
```

Stub del Cliente para Poner_Hora

```
package body Reloj is
  procedure Poner_Hora (t: in  Tiempos) return Status is
    Mensaje: string (1..32);
    M:  system.address;
    OK: boolean;

  begin
    M = Mensaje'address;
    M = Marshall (M, Mi_Dirección);
    M = Marshall (M, SET_TIME);
    M = Marshall (M, t);
    OK = Envia_Mensaje("Servidor_Reloj", Mensaje);
    if OK then
      Recibir (Mensaje);
      UnMarshall_Boolean (Mensaje, OK);
    end if;
    return (OK);
  end Poner_Hora;
  ...
  ...
end Reloj;
```


Aquí tenemos un ejemplo simplificado de una RPC para comunicarse con un Servidor de Reloj, que ofrece servicios para proporcionar y establecer una fecha y hora actual.

La interfaz del servidor consta de dos procedimientos y una estructura de datos, tal como se muestra en la parte superior de la transparencia. Como ya hemos dicho, la interfaz del *stub* del cliente debe ser idéntica a la ofrecida por las rutinas de servicio del servidor.

El cuerpo o implementación del *stub* del cliente contiene el código correspondiente a los dos procedimientos definidos en su interfaz. Veamos el aspecto del procedimiento `Poner_Hora`. En este procedimiento se declara un mensaje que lo rellena con un código de operación (`SET_TIME`) que identifica la operación que se va a solicitar al servidor, y con sus parámetros, es decir, el contenido de la estructura `t` de tipo `Tiempos`.

Una vez que el mensaje ya está formado, se envía mediante la primitiva `Envia_Mensaje` que ofrece el nivel de transporte o directamente el sistema operativo subyacente.

A continuación debe ponerse a esperar la respuesta del servidor, cosa que hace mediante la primitiva `Recibir`. Hasta que llega la respuesta, el proceso queda bloqueado o "en espera". Cuando llega el mensaje de respuesta, deben sacarse los datos de respuesta del mensaje y devolverlos como parámetros de salida del procedimiento `Poner_Hora`.

Obsérvese que mientras que el cliente permanece abstraído del verdadero mecanismo de comunicación utilizado, el programador del *stub* del cliente tiene todo el conocimiento de la semántica de comunicación necesaria, y por lo tanto sabe que debe enviar un mensaje con los datos necesarios a través de la red, y quedarse esperando un mensaje de respuesta; también conoce los datos que debe extraer del mensaje de respuesta para devolver los parámetros de salida indicados en su interfaz.

También deben tenerse en cuenta las rutinas `Marshall` y `UnMarshall`, cuyo cometido es adaptar los parámetros de entrada al formato de datos esperado por el servidor; y, al contrario, adaptar el resultado devuelto por el servidor al formato nativo del cliente.

En algunos casos, por comodidad, utilizaremos el término **serializar** o **serialización** para referirnos tanto a la serialización como a la deserialización de los datos.

...Un Ejemplo

```
procedure Stub_Servidor_Reloj is
  Mensaje: string (1..32);
  M:  system.address;
  OK: boolean;
  Operación: Operaciones;
  t: Tiempos;

begin
  loop
    Recibir (Mensaje);
    M = Mensaje'address;
    M = UnMarshall (M, Remitente);
    M = UnMarshall (M, Operación);
    case Operación is
      when SET_TIME =>
        M = UnMarshall (M, t);
        M = Mensaje'address;
        OK = Poner_Hora (t);
        Marshall (M, OK);
      when GET_TIME =>
        /* Codigo para deserializar
           datos y llamar a Pedir_Hora
        */
    end case;
    Envia_Mensaje (Remitente, Mensaje);
  end loop;
end Stub_Servidor_Reloj;
```

Stub del Servidor del Reloj

Pasemos ahora a ver el aspecto del *stub* del servidor. Como veremos, tiene una estructura muy distinta a la del cliente, pues el stub del servidor es un proceso que recibe peticiones de distinto tipo de todos los clientes.

El *stub* del servidor debe averiguar el tipo de petición que le envía el cliente, deserializar de los datos del mensaje (o *unmarshalling*) y pasárselos, como parámetros de entrada, a la rutina de servicio que ejecutará la operación solicitada por el cliente.

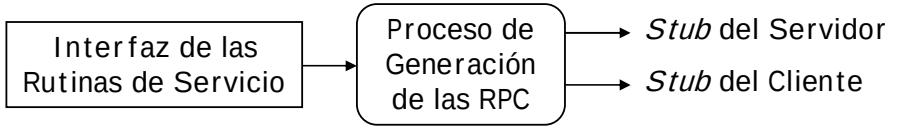
Como se puede apreciar en el código del ejemplo, el *stub* del servidor consta de un bucle infinito, y en cada iteración del bucle atiende una solicitud de un cliente. Así, nos encontramos con que en primer lugar llama a la primitiva `Recibir` para recoger un mensaje con la petición de un cliente. Una vez recibido un mensaje, averigua el tipo de operación solicitada (`Operación`), y en función de la operación requerida, extrae y deserializa los datos del mensaje, y los utiliza como parámetros de entrada en la llamada a la rutina de servicio correspondiente.

En el ejemplo se muestra en detalle el código correspondiente a una solicitud de la operación `SET_TIME`. Como vemos, extrae y deserializa los datos del mensaje mediante la función `UnMarshall`, para formar la estructura local de tipo `Tiempos`. A continuación llama a la rutina de servicio `Poner_Hora` pasándole como parámetro de entrada la estructura `t`.

La rutina `Poner_Hora` (que es local) tras su ejecución devuelve un código de resultado `OK`, indicando si la operación se pudo realizar o no. El *stub* del servidor entonces toma este código de resultado, lo serializa y lo pone en el mensaje de respuesta.

Lo último que le queda por hacer al *stub* del servidor es enviar el mensaje de respuesta al cliente (concretamente al *stub* del cliente). Hecho esto, se vuelve al principio del bucle donde se espera por un nuevo mensaje de algún cliente.

Generación de *Stubs*

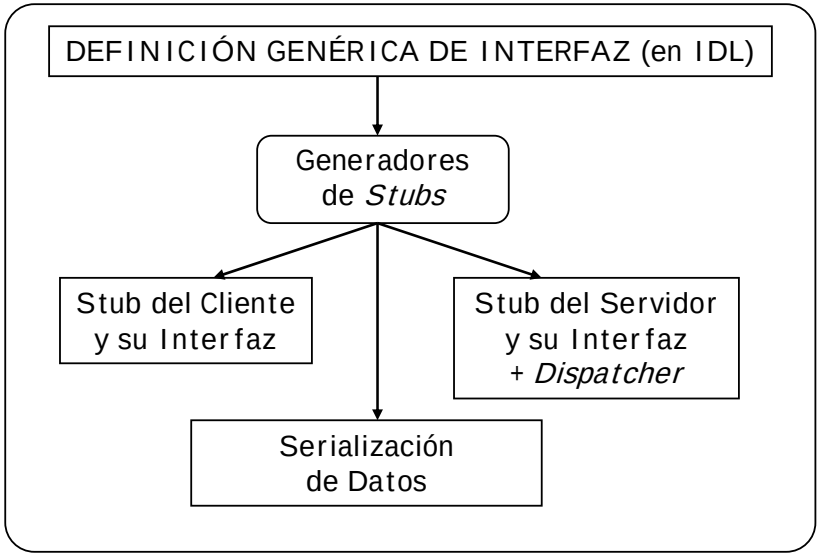


Las Interfaces de las Rutinas de Servicio y del Stub del Cliente ¡NO SON IGUALES!

- Componentes Heterogéneos del S.D.
- Diversos Leng. de Programación
- Múltiples Sistemas Operativos
- Procesadores Diversos

→ DISTINTA REPRESENTACIÓN DE LOS DATOS

Estandarización de la Interfaz Mediante un **Lenguaje de Definición de Interfaz** (IDL)
NCS (OSF), XDR (Sun), Courier (Xerox)



A partir de la interfaz de las rutinas de servicio (o del *stub* del cliente), puede construirse manualmente la implementación de los *stubs* del cliente y el servidor, no obstante, puesto que lo que hay que hacer es muy mecánico, parece razonable automatizar el proceso.

Sin embargo, se presenta un problema inesperado: ¡resulta que las interfaces de las rutinas de servicio del servidor y del stub del cliente no son exactamente iguales!

En el ejemplo de las transparencias anteriores estas interfaces eran exactamente iguales. Pero es que hasta ahora hemos obviado el hecho de que los procesos cliente y servidor están en máquinas distintas, y hemos supuesto que éstas son iguales o equivalentes. Pero la realidad es que en un sistema distribuido los componentes son heterogéneos, con lo que los procesos cliente y servidor se pueden haber programado en lenguajes diferentes, apoyados en sistemas operativos variados y ejecutándose sobre procesadores de distinta arquitectura. Por otra parte, nos encontramos con que el servidor debe ofrecer una interfaz genérica para todos sus clientes.

La única forma de solucionar esto es que el servidor no ofrezca la interfaz de sus servicios acorde a un lenguaje de programación específico, sino en un lenguaje genérico al que se le denomina **Lenguaje de Definición de Interfaces** (IDL).

Los lenguajes de definición de interfaces suelen estar derivados de otros lenguajes de programación, pero para construir los *stubs* añaden cierta información necesaria que no todos los lenguajes de programación proporcionan. Esta información adicional típicamente es: el sentido de los parámetros (entrada, salida, entrada/salida), discriminantes para registros variantes (o uniones) e información sobe la longitud de los vectores o matrices que se pasan como parámetros.

Ejemplos de IDLs son: NCS (de la OSF), XDR (de Sun), Courier (de Xerox), MiG (de Mach).

Ahora nos encontramos con que a partir de una definición de una interfaz genérica (realizada en un IDL) se deben generar las interfaces y los cuerpos o implementaciones apropiadas para los lenguajes del servidor y de cada cliente.

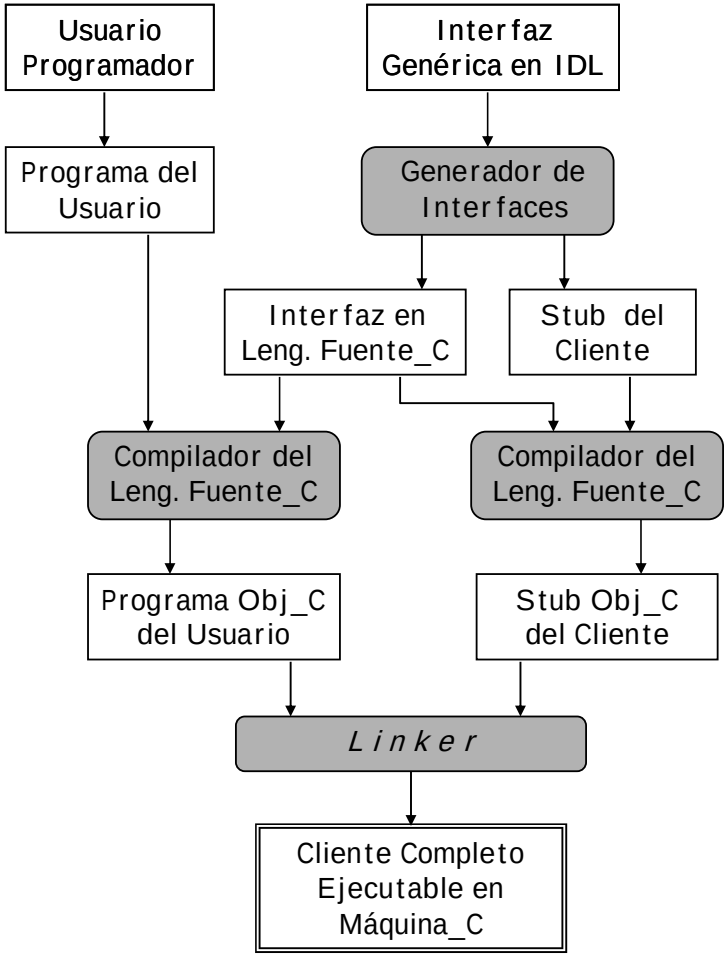
A partir de una definición genérica en IDL y un compilador de interfaces, se pueden realizar automáticamente las siguientes tareas:

1. Generación del *stub* del cliente (y su fichero de interfaz) para todos los perfiles (o firmas) de los procedimientos definidos en la interfaz. El *stub* se compilará y se montará con el programa del cliente.
2. Generación del *stub* del servidor con el repartidor (o *dispatcher*) para todas las operaciones ofrecidas por el servidor. El *stub* y el repartidor (incluido en el *stub*) se montarán con el programa del servidor.
3. Generación del fichero de definición de las operaciones ofrecidas, para el lenguaje correspondiente en el que están implementadas en el servidor. Los cuerpos correspondientes a estas definiciones las proporciona el programador de las operaciones del servidor.
4. Aprovechando la definición del perfil de los procedimientos de interfaz (que definen los tipos de los parámetros de entrada y salida), se generan las acciones apropiadas para la serialización correspondiente a cada procedimiento de interfaz en los *stubs* del cliente y del servidor.

La utilización de una definición común de la interfaz en el proceso de generación de los *stubs* del cliente y del servidor y el fichero de definición de los servicios del servidor, asegura que los tipos de los argumentos y de los resultados manejados por el cliente se ajustan a los definidos por el servidor.

...Generación de *Stubs*

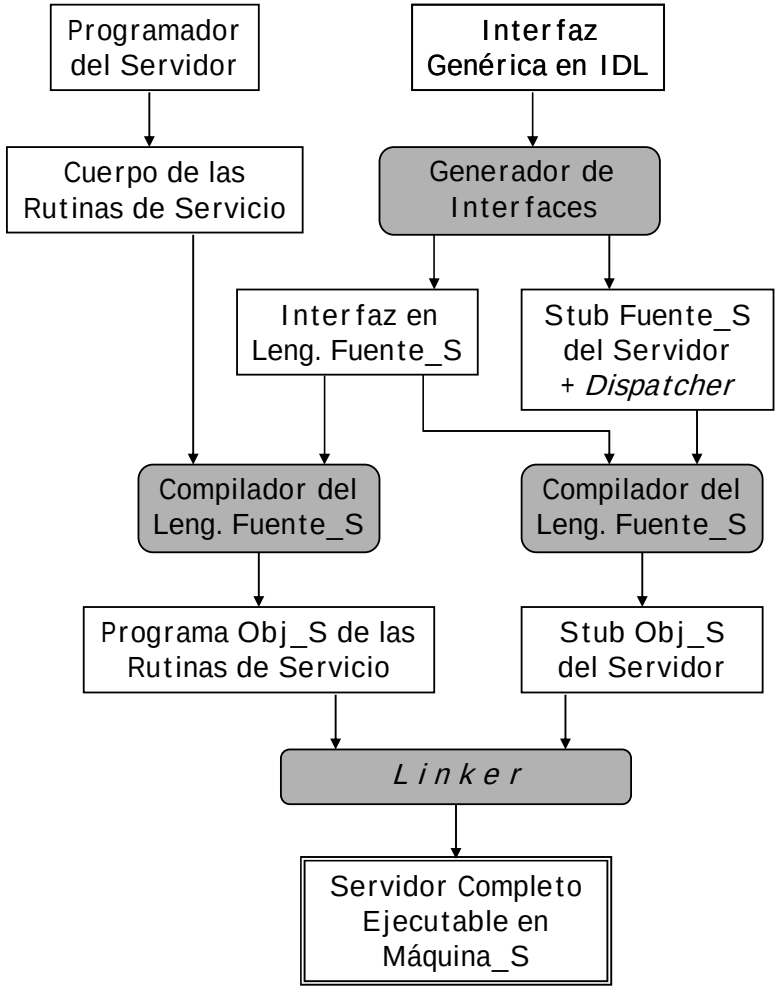
CLIENTE



Sistemas Distribuidos La Comunicación - 19

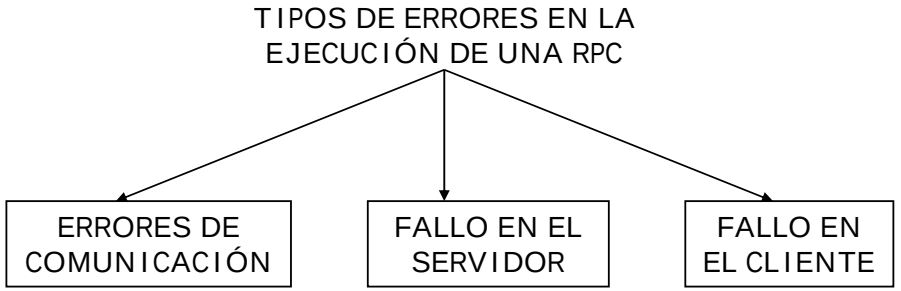
...Generación de *Stubs*

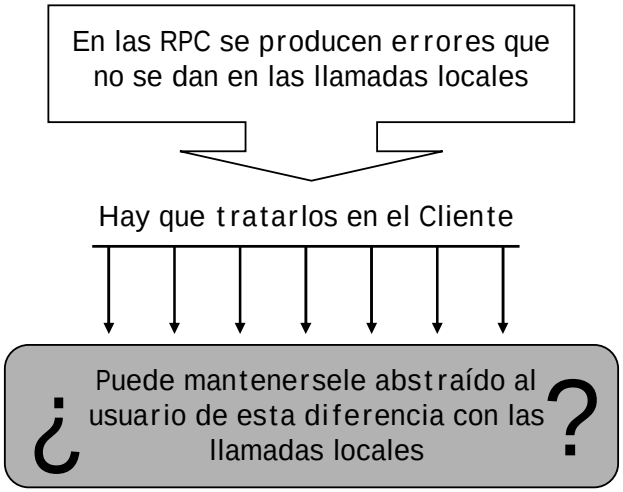
SERVIDOR



Sistemas Distribuidos La Comunicación - 20

Tratamiento de Errores





Tratamiento de Errores. El objetivo de las RPC es abstraer al usuario de la comunicación que implica una llamada a un procedimiento remoto, haciéndola parecer igual que las llamadas locales; y excepto por el hecho de que no se puede acceder a las variables globales, tal abstracción se consigue bastante bien.

Esto es así siempre que el cliente y el servidor funcionen correctamente. Pero los problemas surgen cuando aparecen los errores; entonces ya no es tan fácil enmascarar una llamada a procedimiento remoto bajo una llamada local. Veamos a continuación los tipos de errores que pueden surgir y qué se puede hacer con ellos.

En principio puede haber tres tipos principales de problemas:

Errores de Comunicación.
- No se puede localizar al servidor
- Pérdida de mensajes de petición
- Pérdida de mensajes de respuesta.

Fallo en el Servidor
- Error por parámetros incorrectos, operación no permitida, etc.
- Caída del Servidor

Fallo en el Cliente.
- Caída del cliente

En las transparencias siguientes veremos con cierto detalle cómo pueden tratarse cada una de estas situaciones.

Como veremos, debido a estos errores, en las RPC debe incluirse código adicional para tratarlos convenientemente, lo cual hace dudar sobre si las RPC deben ser totalmente transparentes al usuario o se le deben ofrecer a éste primitivas distintas para que sea consciente del tratamiento que le debe dar a cada situación de error.

Fallos de Comunicación

☞ **NO SE PUEDE LOCALIZAR AL SERVIDOR**

- Servidor caído ➜ No registrado
- Servidor caído después de registrarse

⇒ Servidor No Disponible

- Versiones incompatibles de cliente y servidor

El *binder* debe devolver un status de error
El servidor puede devolver un resultado

⇒ ¡No Confundir!

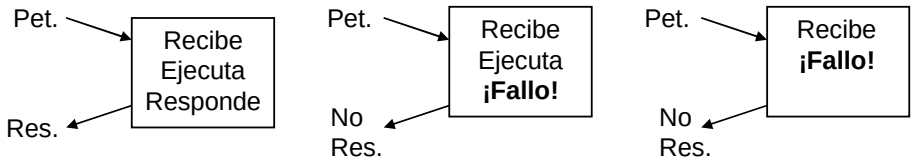
☞ **PÉRDIDA DE MENSAJES DE PETICIÓN**

- Si vence Temporización sin respuesta ➜ Repetición

Después de *n* reintentos ➜ Servidor caído o línea cortada

¡ No se puede localizar al servidor !

☞ **PÉRDIDA DE MENSAJES DE RESPUESTA**



¿Qué ha pasado?

Si no se obtiene respuesta ➜ ¡Repetir la petición?

Solo con operaciones idempotentes

Solución: Numero de secuencia en las peticiones

Tratamiento de Errores: Errores de Comunicación.

Vamos a considerar errores de comunicación a todos aquellos que, siendo ajenos al cliente y al servidor, impiden que una petición del cliente llegue al servidor, o que las respuestas de éste se le entreguen al cliente.

1) No se puede localizar al servidor

Cuando se le solicita al *binder* la dirección del servidor apropiado, puede ocurrir que el servidor esté caído y no se haya registrado, por lo que no está disponible. Otra posibilidad es que después de conseguir la dirección del servidor, éste se caiga y por lo tanto no se reciba ninguna respuesta a las peticiones. También puede suceder que la versión del cliente no contraste con la de los servidores disponibles. En cualquiera de estos casos, el *binder* devolverá un status que, a través del *stub*, le deberá llegar al programa del usuario.

Se deberá tener cuidado para no confundir un status de resultado (por ej. un -1) con un resultado de la operación (que también podría ser un -1).

2) Pérdida de mensajes de petición

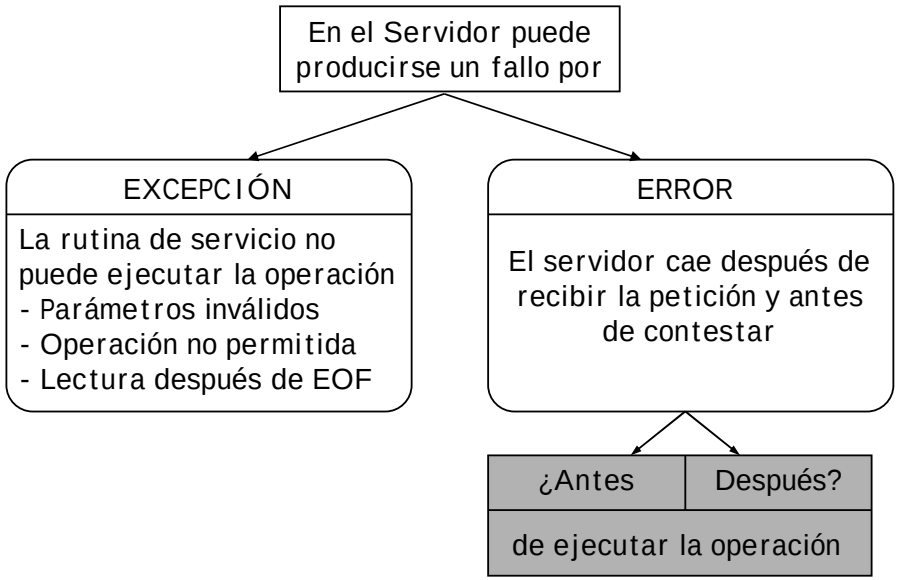
Para tratar la posible pérdida de las peticiones, la solución más fácil es que el módulo de comunicaciones ponga una temporización cuando envía el mensaje de petición. Si se vence la temporización sin recibir un *ACK* o el mensaje de respuesta, repite el envío de la petición. Los mensajes de petición pueden repetirse un cierto número de veces; después debe suponerse que o bien el servidor está caído o bien que la línea de comunicación está cortada, con lo que se vuelve al caso anterior: no se puede localizar al servidor.

3) Pérdida de mensajes de respuesta.

La solución obvia a este problema podría estar en apoyarse otra vez en las temporizaciones, de tal forma que si no se recibe la respuesta en un tiempo finito, simplemente se vuelve a enviar la petición. Pero este reenvío lo hace el cliente sin saber cuál es el motivo de no haber recibido la respuesta: ¿se perdió la petición?, ¿se perdió la respuesta? o ¿simplemente es que el servidor es lento?; y esto puede ser un problema.

Si se perdió la petición, no hay ningún problema, pero si se perdió la respuesta implica que las operaciones solicitadas van a repetirse. Algunas operaciones pueden repetirse varias veces sin ningún problema (por ejemplo, sumar dos matrices). Estas operaciones se dice que son **idempotentes**. Pero otras no pueden repetirse alegremente, por ejemplo, una orden de transferencia bancaria de un millón de euros. Para evitar la repetición de una misma operación, las peticiones pueden incluir un número de secuencia, de tal forma que en el servidor se compruebe que no se ejecuta dos veces la misma operación, y cuando le llega una petición repetida, no la ejecuta, sino que simplemente responde con el resultado.

Fallo en el Servidor



¡Solo se Pueden Repetir las Operaciones **Idempotentes**!

SEMÁNTICAS DE ENTREGA DE MENSAJES

	Reenviar mensaje	Filtrar duplicados	Re-ejecutar servicio / Retransmitir respuesta
QUIZÁS	No	No	No
AL MENOS UNA	Si	No	Re-ejecutar
COMO MUCHO UNA	Si	Si	Retransmitir respuesta

Tratamiento de Errores: Fallo en el Servidor

Dentro de este apartado vamos a considerar dos tipos de problemas:

- La rutina de servicio no puede ejecutar la operación solicitada.
- El servidor se cae después de recibir la petición y antes de responder con el resultado.

Puede haber varios **motivos para que la rutina de servicio no pueda ejecutar la operación solicitada**. Estos son algunos de ellos.

- Parámetros inválidos.
- Solicitud de una operación no existente o sin permiso para ella.
- Intentar leer de un fichero después de llegar al final.

Como se puede ver, todos estos motivos son responsabilidad del cliente. El servidor puede responder con un status de resultado indicando el motivo por el que no se ejecutó la operación solicitada.

El problema surge cuando el servidor se cae. Una vez que se le ha enviado una petición al servidor, éste puede caerse (si hay fallos) bien una vez que ha realizado la operación solicitada y antes de enviar la respuesta, bien antes de finalizar la operación. En cualquier caso, el *stub* del cliente, al no recibir la respuesta acabará detectando un fallo en la RPC. Y aquí se produce el mismo problema que con el error de comunicaciones ¿qué se debe hacer al no recibir la respuesta? Si la operación es idempotente, está claro, se vuelve a enviar la petición, pero si no lo es ¿qué se hace?

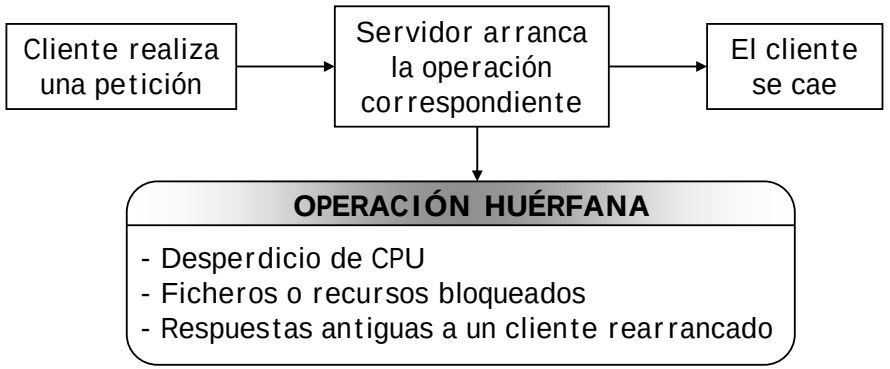
Para el tratamiento tanto de este tipo de errores como de los debidos a las comunicaciones, el módulo de comunicaciones puede construirse con distintas semánticas de entrega de mensajes.

Pero **¿qué es entregar un mensaje?** Cuando se envía un mensaje a otra estación, el mensaje lo recibe el módulo de comunicaciones mediante el *driver* correspondiente. Cuando este módulo comprueba la validez y orden del mensaje, se lo entrega al proceso de destino.

Veamos ya las distintas **semánticas de entrega de mensaje**:

- Semántica del **quizás**. No hay ninguna tolerancia a fallos. Se envía el mensaje de petición, y si no se recibe la respuesta en un tiempo determinado no se realiza ningún reenvío. Si el servicio solicitado no tiene que devolver un resultado, el cliente queda sin saber si se ejecutó o no.
- Semántica **al menos una**. Ante un *time-out*, se retransmite la petición. Esto asegura que la petición se ejecuta al menos una vez. Pero si hay fallos de comunicaciones en las respuestas, puede que se acabe ejecutando varias veces. Esta semántica es válida sólo cuando las operaciones son idempotentes.
- Semántica **como mucho una**. El módulo de comunicaciones del servidor establece un filtro mediante el cual no entrega dos veces la misma petición. Esto puede construirse mediante las peticiones con número de secuencia vistas anteriormente.

Fallo en el Cliente



SOLUCIONES

➳ EXTERMINACIÓN
- El stub hace un apunte antes de enviar la RPC
  Al rearrancar se pide al servidor que mate al huérfano
- Problemas: ·Sobrecarga de escritura en disco
  ·Si red partida ➜ inaccesibles

➳ REENCARNACIÓN
- Cada arranque del cliente es una "Época"
  Cada petición lleva su época
  En rearranque del cliente: Broadcast "nueva época"
  El servidor mata operaciones de épocas pasadas
- Problema: Si red partida ➜ inaccesibles
  El cliente rearrancado tira respuestas antiguas

➳ EXPIRACIÓN: Cada RPC lleva su porción de tiempo

Matar a un huérfano trae consecuencias imprevistas
- → Ficheros o B.D. bloqueadas
- → Peticiones huérfanas en colas remotas
- → ¿Y su descendencia?

↓

¡No Matar Huérfanos!

Tratamiento de Errores: Fallo en el Cliente

Cuando el cliente envía una petición al servidor, y el cliente se cae antes de recibir la respuesta, una operación se queda activa en un servidor sin que haya ningún proceso cliente esperando por el resultado. A una operación que se queda en esta circunstancia se le denomina **huérfana**.

Los huérfanos pueden provocar diversos problemas, pues como mínimo producen un desperdicio de CPU. También puede ocurrir que algunos ficheros o recursos se queden bloqueados en el servidor (solo si el servidor conserva estados entre peticiones). Incluso puede suceder que el cliente rearranque, envíe una nueva y distinta petición; inmediatamente recibiría la respuesta a la última petición que hizo antes de caerse, y posteriomente la respuesta a la nueva petición, lo cual le generaría cierta confusión.

Veamos algunas soluciones al problema de los huérfanos.

Solución 1: Exterminación. Antes de que el *stub* del cliente envíe una RPC, realiza un apunte en un fichero de registro en el disco. Si se cae el cliente, después de rearrancar comprueba el fichero de registro y se envía un mensaje al servidor para matar al huérfano. El problema de esta solución no es solamente la sobrecarga que supone el acceso al disco en cada RPC; el problema es que el huérfano puede haber creado sus propios huérfanos en otros servidores que aún en el caso de poder localizarlos a todos, puede que sea imposible acceder a ellos por no tener comunicación debido a fallos o cortes de la red.

Solución 2: Reencarnación. El cliente divide el tiempo en unidades secuenciales llamadas *épocas*, de tal forma que cada vez que arranca incrementa en uno el contador de épocas. Cada petición que realiza el cliente va acompañada de su época vigente. Cuando el cliente rearranca envía un mensaje de *broadcast* indicando una nueva época. Cuando un servidor recibe el mensaje, si tiene algún proceso ejecutando una petición de ese cliente, lo mata directamente. Esto no impide que algunos huérfanos sobrevivan (por ej, si la red está partida), pero si una respuesta de un huérfano le llega al cliente después de rearrancar, al comprobar la época comprobaría que está obsoleta y la desecharía.

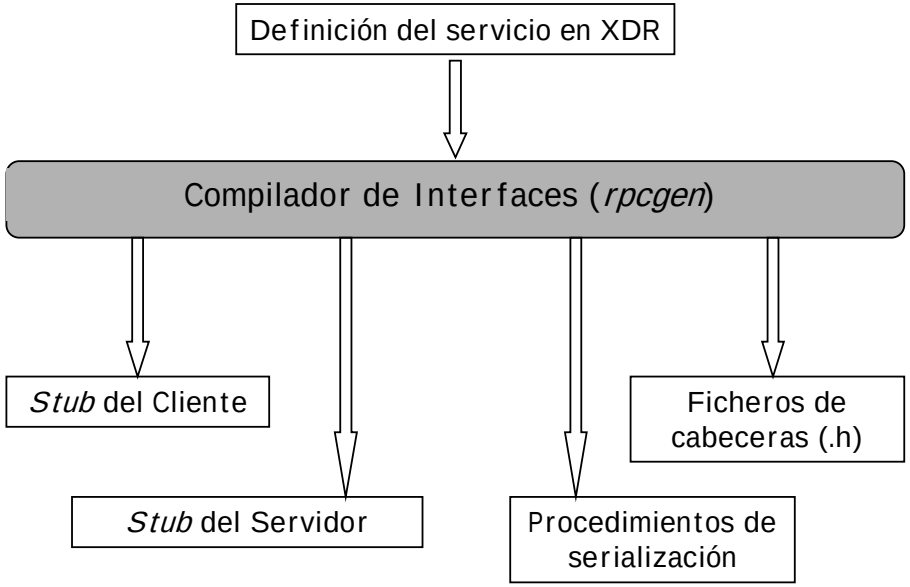
Solución 3: Expiración. A cada RPC se le asigna una porción de tiempo *T* para ser ejecutada. Si el servidor no realiza el trabajo en ese tiempo, debe pedirle otra porción de tiempo al cliente, y si no recibe esa porción del cliente, mata al proceso que se ocupa de esa petición. Si se cae el cliente, es seguro que después de un tiempo *T*, todos los huérfanos habrán muerto. El problema de esta solución estriba en elegir un tiempo *T* razonable que sea válido para todas las RPC (con requisitos de tiempo muy distintos).

En la práctica, **ninguna de estas soluciones se utiliza**, pues matar un huérfano puede producir consecuencias imprevistas. Por ejemplo, supongamos que un huérfano ha puesto un bloqueo a un fichero o a una base de datos; si el huérfano muere, los ficheros quedan bloqueados para siempre. También puede ocurrir que un huérfano haya realizado peticiones en colas remotas para arrancar procesos en un tiempo futuro, por lo que aunque se mate al huérfano, no se eliminan todas trazas que ha ido dejando.

Un Ejemplo: RPC de Sun

☞ Desarrollado en 1990 para soportar el modelo cliente-servidor de NFS

- Unix - NFS
- PC-NFS
- . . .



☞ No tiene *binder* de red, sino un *binder* local

☞ No consigue transparencia total al usuario → El usuario requiere utilidades de bajo nivel

Se han implementado diversos sistemas de RPC para Unix. Aquí vamos a comentar la RPC que Sun realizó en 1990 para soportar el modelo cliente-servidor sobre el que se construyó su sistema de ficheros en red NFS.

Aunque este mecanismo forma parte de los sistemas operativos de Sun, también puede encontrarse con otras instalaciones de NFS que se han desarrollado, por ejemplo para Windows (PC-NFS) y para diversas variedades de Unix.

El sistema de RPC de Sun proporciona un lenguaje de definición de interfaz llamado XDR (eXternal Data Representation) y un compilador de interfaces denominado *rpcgen*. A partir de una interfaz definida en XDR y con ayuda del *rpcgen*, se obtiene gran parte de los componentes del mecanismo completo de una RPC, es decir:

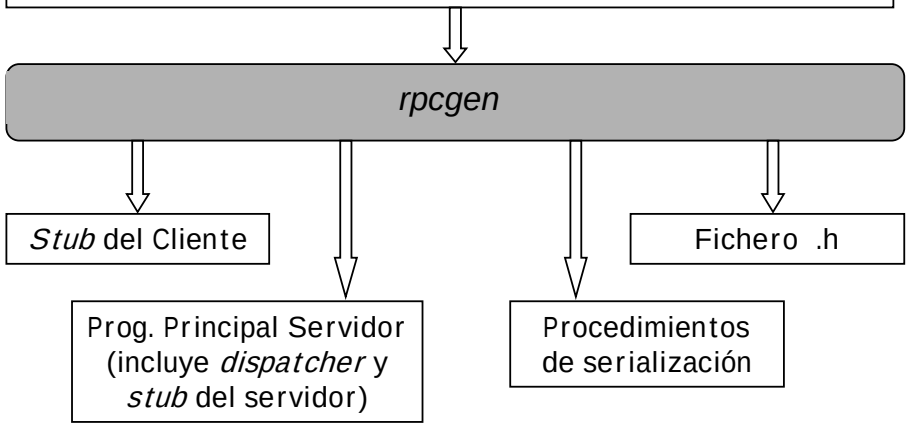
- El *stub* del cliente.
- El programa principal del servidor (que incluye el *stub* y el *dispatcher*).
- Los ficheros de definición o de cabeceras del *stub* del cliente y de las rutinas de servicio del servidor.
- Las rutinas de serialización de datos.

En las siguientes transparencias vamos a ir comentando cada uno de estos componentes, así como el servicio de *binding* ofrecido, al que se accede desde el *stub* del cliente. También veremos el aspecto que tiene el programa principal del cliente (aportado por el usuario).

A pesar de que la transparencia absoluta es algo deseable, no siempre es fácil conseguirla, como ocurre en este caso. Por eso, a veces, puede que el usuario tenga que acceder a ciertas utilidades de bajo nivel que ofrece este sistema y que comentaremos en último lugar.

Generación de Interfaz

```
/* Definicion, en XDR,  del servicio "Archivador"
   contenida en el archivo archivador.x
*/
const MAX = 1000;
typedef int Id_Fichero;
typedef int Fichero_PTR;
typedef int Longitud;
struct Datos {Longitud long_datos;
              char buffer[MAX];
              int error;
             };
struct Args_Escritura {Id_Fichero fichero;
                       Fichero_PTR posicion;
                       Datos dato;
                      };
struct Args_Lectura { Id_Fichero fichero;
                      Fichero_PTR posicion;
                      Longitud long_datos;
                     };
program ARCHIVADOR{
  version VERSION_ACTUAL{
    void ESCRIBIR (Args_Escritura)=1;
    Datos LEER (Args_Lectura)=2;
  }=2;
}=9999;
```



En un principio Sun definió el XDR como un lenguaje de representación externa de datos, aunque posteriormente lo amplió para convertirse en un lenguaje de definición de interfaz, por lo que puede utilizarse para especificar no solamente las **definiciones de las operaciones ofrecidas**, sino también **su identificador**, el **identificador o número del servicio** (en lugar de un nombre) y su **versión**.

Por terminología, diremos que un **proceso servidor** ofrece un **servicio** compuesto de varias **operaciones**.

Como se puede ver en el ejemplo, este lenguaje proporciona una notación (similar a C) para definir constantes, definiciones de tipos, tipos enumerados, uniones, etc., las cuales preceden a la definición del servicio (programa) y a la de sus operaciones ofrecidas.

La **definición de un programa** consta de la definición de sus operaciones junto con sus números de identificación asociados, la versión del programa y su número de identificación.

Una **definición de operación** está compuesta por el perfil y por su número de identificación. Este identificador es el que se incluye en el mensaje de petición para indicar la operación solicitada.

El perfil de una operación consta del tipo del resultado, del nombre del procedimiento, y del tipo del único parámetro (que es de entrada). Tanto el resultado como el parámetro de entrada pueden ser datos escalares o estructuras de datos.

Así pues, en nuestro ejemplo tenemos que:

Operación `ESCRIBIR` = 1

Operación `LEER` = 2

Versión del servicio = 2

Servicio `ARCHIVADOR` = 9999

Una definición de interfaz constituye la entrada al compilador de interfaces *rpcgen*, el cual genera como salida los siguientes componentes:

- *Stub* del cliente.
- Programa principal del servidor, que incluye el *dispatcher* y el *stub*.
- Rutinas de serialización (a formato XDR) que utilizarán tanto el *stub* del cliente como el del servidor.
- Ficheros de cabeceras (para lenguaje C) con las definiciones de tipos, constantes y procedimientos del *stub* del cliente y de las rutinas de servicio del servidor. El programador debe escribir los cuerpos o implementaciones correspondientes a estas especificaciones.

...Generación de Interfaz

```
/* archivador.h
 * Please do not edit this file.
 * It was generated using rpcgen.
 */
#include <rpc/types.h>

#define MAX 1000

typedef int Id_Fichero;
bool_t xdr_Id_Fichero();

typedef int Fichero_PTR;
bool_t xdr_Fichero_PTR();

typedef int Longitud;
bool_t xdr_Longitud();

struct Datos {
   Longitud long_datos;
   char buffer[MAX];
   int error;
};
typedef struct Datos Datos;
bool_t xdr_Datos();

struct Args_Escritura {
   Id_Fichero fichero;
   Fichero_PTR posicion;
   Datos dato;
};
typedef struct Args_Escritura Args_Escritura;
bool_t xdr_Args_Escritura();

struct Args_Lectura {
   Id_Fichero fichero;
   Fichero_PTR posicion;
   Longitud long_datos;
};
typedef struct Args_Lectura Args_Lectura;
bool_t xdr_Args_Lectura();

#define ARCHIVADOR ((u_long)9999)
#define VERSION_ACTUAL ((u_long)2)
#define ESCRIBIR ((u_long)1)
extern void *escribir_2();
#define LEER ((u_long)2)
extern Datos *leer_2();
```

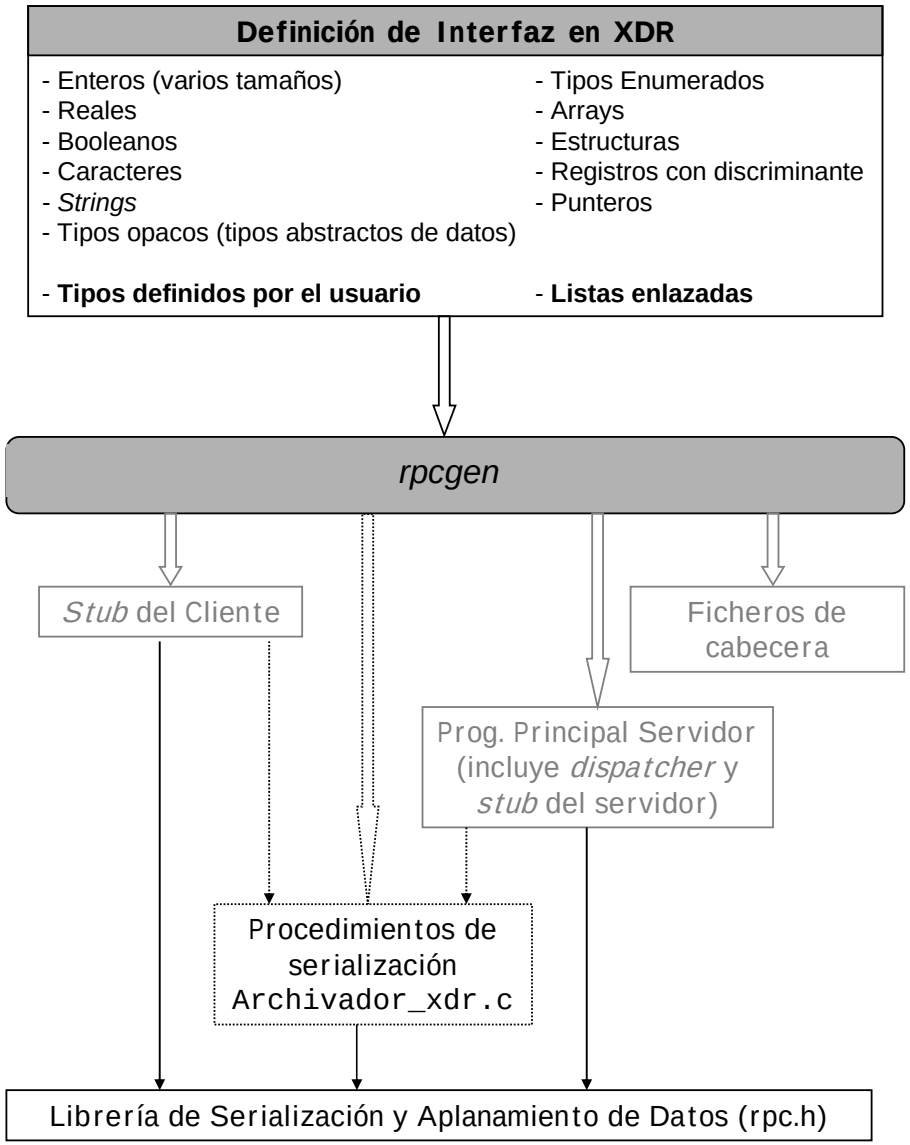

En esta transparencia tenemos el fichero de cabeceras generado por *rpcgen*. Toma como nombre el mismo nombre del fichero de definición en XDR pero con extensión `.h`.

Aquí se incluyen, entre otras cosas, las definiciones en C de los tipos definidos por el usuario en el fichero XDR. No merece la pena comentar aquí el significado en C de estas definiciones, y es preferible permanecer abstraído.

No obstante, merece la atención ver al final del fichero las definiciones de las constantes con los valores (indicados en el fichero XDR) del programa `ARCHIVADOR` y de su versión `VERSION_ACTUAL`.

También se proporcionan las definiciones de las dos operaciones declaradas. Por una parte sus códigos de operación, `ESCRIBIR` y `LEER`, respectivamente, y por otra el perfil de sus llamadas: `escribir_2` y `leer_2`. Como puede observarse, los nombres de los procedimientos asociados a las operaciones se forman con el nombre de la operación definida en XDR, en minúsculas, seguido de un subrayado y el número de la versión.

Marshalling



La RPC de Sun puede pasar cualquier tipo de estructuras de datos como argumentos o resultados, utilizando XDR como sistema de representación externa de datos.

El sistema dispone de las rutinas estándar que serializan automáticamente datos de los siguientes tipos:

- Enteros (varios tamaños)
- Reales
- Booleanos
- Caracteres
- *Strings*
- Tipos enumerados
- Tipos opacos (tipos abstractos de datos)
- Arrays
- Estructuras
- Registros con discriminante
- Punteros

Este sistema también permite pasar listas dinámicas como parámetros, siendo los *stubs* los que se encargan de aplanarlos y desaplanarlos automáticamente mediante rutinas de biblioteca previstas para ello.

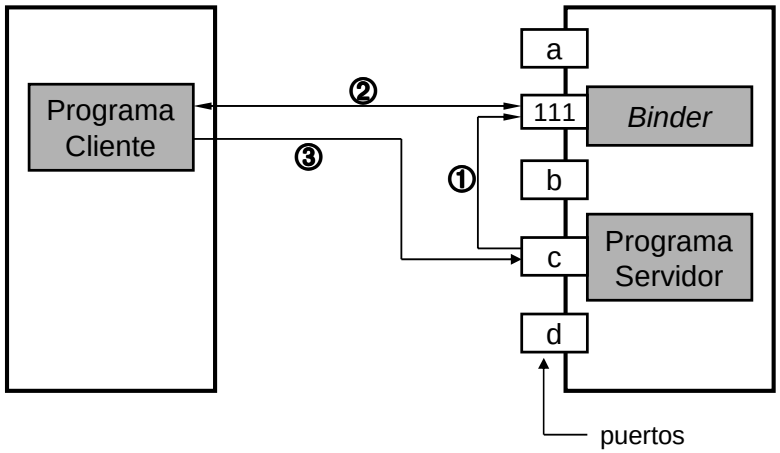
Si los parámetros son de tipos predefinidos en XDR, los *stubs* utilizan directamente rutinas de librería (definidas en `rpc.h`) para la serialización. No obstante, XDR permite la definición de tipos de usuario, por lo que los parámetros pueden ser de un tipo definido por el usuario. En este caso, *rpcgen* genera las rutinas adecuadas para serializar el parámetro, y las deja en un fichero con nombre `archivador_xdr.c`, siendo `archivador` el nombre sin extensión del fichero de definición de la interfaz en XDR.

Los *stubs* del cliente y servidor realizarán las llamadas oportunas a las rutinas de librería o a las generadas en `archivador_xdr.c` para serializar los datos del mensaje.

Binding

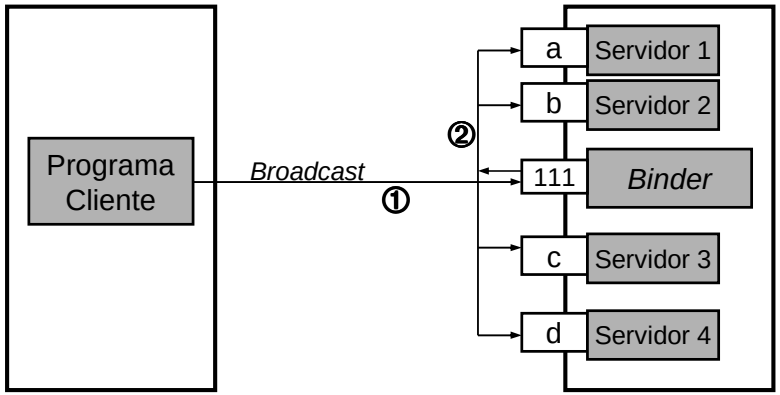
La RPC de Sun no ofrece un servicio de binding global a nivel de red, sino local para cada máquina.

Servicio ⟶ Puerto



Puerto del *binder* ⟶ Fijo y preestablecido

Puerto del servidor ⟶ Dinámico



Las RPC de Sun no ofrecen un servicio de *binding* a nivel de red, sino solamente local, es decir, en cada máquina hay un *binder* (Sun lo denomina *portmapper*) que indica el puerto por el que escucha cada uno de los servidores de esa máquina.

El *binder* de cada máquina es un servidor que atiende las peticiones por un puerto fijo predeterminado e igual para todas las máquinas (111), y se arranca automáticamente en el proceso de arranque del sistema. Cuando un servidor arranca, se registra en el *binder* de su ordenador, indicando el identificador (número) del servidor, la versión y el puerto por el que escucha.

Cuando un cliente efectúa una RPC, debe indicar el nombre o dirección de la máquina o *host* del servidor correspondiente. El stub del cliente entonces le envía al *binder* de esa máquina una petición con el identificador del servicio solicitado y la versión. Si el identificador y versión son correctos, el *binder* del *host* del servidor responde con el puerto al que se pueden enviar las peticiones del cliente.

Como se puede apreciar, los *binders* deben atender siempre las peticiones por un puerto fijo, mientras que los servidores pueden escuchar por cualquier puerto, puesto que es el *binder* el que realiza la correspondencia dinámica entre identificador de servidor y el puerto por el que escucha.

Cuando por algún motivo hay que enviar un mensaje a todos los servidores de un servicio, lo que se hace es enviar un mensaje por *broadcast* a un puerto concreto, esperando que todos los servidores del servicio en cuestión estén escuchando por ese puerto. Sin embargo, en nuestro caso, cuando para un servicio hay múltiples servidores en distintas máquinas, cada servidor puede utilizar un puerto distinto para recibir las peticiones, con lo que no sirve el sistema de envío múltiple y directo de mensajes visto hasta ahora.

En su lugar, lo que debe hacer el cliente es hacer un envío múltiple de una RPC especial, que llega a todos los *binders* locales de todas las máquinas (puesto que todos tienen un puerto fijo). Cada *binder* que recibe la petición, analiza el mensaje extrayendo el identificador del programa y la versión. Si estos son válidos (el servidor existe y la versión es correcta), redirige el mensaje al puerto correspondiente por el que atiende el servidor.

Una forma de localizar un servicio que no se sabe en qué máquina está, es haciendo un *broadcast* a todos los *binders* para que hagan una llamada al procedimiento 0 del servicio buscado. El procedimiento 0 de cada servicio está predefinido y se utiliza para hacer "*ping*" (preguntar si está vivo).

Programa del Usuario

```
/*
  Programa del usuario del servicio de archivos: cliente.c
*/
#include <stdio.h>
#include <rpc/rpc.h>
#include “Archivador.h”

main (int argc, char **argv) {
   CLIENT *id_cliente;
   char *Nombre_Servidor = “ServidorFavorito”;
   Args_Lectura a;
   Datos *Mis_Datos;

   id_cliente = clnt_create (Nombre_Servidor, ARCHIVADOR,
                             VERSION_ACTUAL, “TCP”);
   if (id_cliente == NULL){ /* sin conexión con el server */
      clnt_pcreateerror (Nombre_Servidor);
      exit (1);
   }
   a.fichero = 10;
   a.posición = 100;
   a.long = 1000;
   Mis_Datos = leer_2 (&a, id_cliente); /* RPC */
   if (Mis_Datos == NULL) { /* Error en la llamada */
      clnt_perror (Id_cliente, Nombre_Servidor);
      exit (1);
   }
   if (Mis_Datos.error == NULL) { /* Error en el server */
      fprintf (stderr, “Error en el servidor: %s\n”,
               Nombre_Servidor);
      exit (1);
   }
   ...
   ...
   clnt_destroy (id_cliente); /* cierra la comunicación */
}
```


Vamos a llamar “programa de usuario” a la parte del cliente proporcionada por el usuario. El programa de usuario constituye el programa principal del cliente.

Hemos dicho que, debido a la transparencia, un programa con llamadas locales debería ser exactamente igual a uno en el que las llamadas son remotas. Sin embargo, el sistema de RPC de Sun no ofrece una transparencia absoluta, y el usuario debe realizar ligeras modificaciones a las llamadas de su programa.

El motivo por el que no se consigue una transparencia total se debe, en gran medida, a que no se dispone de un servicio de *binding* global a nivel de red. Por esto, el usuario, al comienzo del programa, debe solicitar la dirección completa del servidor que ofrece el servicio a utilizar. Una vez que se le devuelve un descriptor de cliente, el usuario debe incluir el descriptor de cliente en todas las RPC dirigidas a ese servicio. Cuando ya no va a realizar más RPC, el usuario debe cancelar el descriptor de cliente que se le había concedido para trabajar con el servidor.

Como se puede ver en el ejemplo, un descriptor de cliente se obtiene mediante

```
clnt_create (MaquinaServidor,Id_Servicio,Versión,Protocolo);
```

que devuelve como resultado un descriptor de cliente. Siendo `MaquinaServidor` el nombre de una máquina conocida en la que sabemos que reside un servidor del servicio requerido. `Id_Servicio` es el número dado al programa en la definición de interfaz en XDR. En nuestro ejemplo, este valor se indica mediante el identificador `Archivador`. La versión a utilizar se indicará, igualmente, mediante `VERSION_ACTUAL`. El transporte de las RPC admite dos protocolos: TCP y UDP. En el último parámetro de la llamada a `clnt_create` debe indicarse el protocolo de transporte elegido.

Para cancelar la utilización de un servicio debe llamarse a

```
clnt_destroy (Id_cliente);
```

Obsérvese que el nombre de la RPC se forma con el nombre con el que se definió en la definición de la interfaz, convertido a minúsculas, y seguido del número de versión separado con un subrayado.

En cuanto a los errores, pueden producirse en el proceso de llamada (antes de enviar el mensaje) o durante su ejecución en el servidor. En el primer caso, el propio mecanismo de la RPC devuelve un cero como resultado de la llamada. En el segundo caso, debe ser el usuario el que debe estar de acuerdo con el programa servidor para comprobar si ha habido error o no mediante algún campo establecido al efecto en la estructura de datos devuelta por el servidor.

Stub del Cliente

```
/* archivador_clnt.c
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <rpc/rpc.h>
#include "archivador.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

void * escribir_2(argp, clnt)
    Args_Escritura *argp;
    CLIENT *clnt;
{
    static char res;

    bzero((char *)&res, sizeof(res));
    if (clnt_call(clnt, ESCRIBIR, xdr_Args_Escritura, argp,
                  xdr_void, &res, TIMEOUT) != RPC_SUCCESS)
{
        return (NULL);
    }
    return ((void *)&res);
}

Datos * leer_2(argp, clnt)
    Args_Lectura *argp;
    CLIENT *clnt;
{
    static Datos res;

    bzero((char *)&res, sizeof(res));
    if (clnt_call(clnt, LEER, xdr_Args_Lectura, argp,
                  xdr_Datos, &res, TIMEOUT) !=
RPC_SUCCESS){
        return (NULL);
    }
    return (&res);
}
```


El *stub* del cliente contiene tantas rutinas como operaciones tiene el servicio. En nuestro caso tenemos dos: `escribir_2` y `leer_2`.

Estos procedimientos no hacen mucho. En realidad su cometido consiste en pasar los parámetros al formato de codificación externa, aplanar estructuras y solicitar el envío del mensaje por la red.

Dentro de cada uno de estos dos procedimientos, la llamada a la función `clnt_call` es la que realiza estas labores descritas. Veamos cuales son los parámetros que se le pasan a esta función `clnt_call`:

- El descriptor de cliente
- Identificador de operación
- Rutina para serializar el argumento
- Argumento de entrada de la operación
- Rutina para deserializar el resultado
- Dirección de una variable en la que se recibirá el resultado de la RPC
- Tiempo total máximo de espera a la respuesta

Recuérdese que si se requieren varios parámetros de entrada en la llamada a la RPC, estos deben organizarse todos en una estructura, de tal forma que se pase un único parámetro. No obstante, en las versiones actuales de RPC sí se permite el paso de varios parámetros directamente, sin necesidad de meterlos en una estructura

La función `clnt_call` utiliza una semántica del tipo “al menos una”. El tiempo de espera entre reintentos tiene un valor por defecto que se puede modificar en el programa del usuario mediante `clnt_control`, después de llamar a `clnt_create`. Así, el número de reintentos es el tiempo total dividido entre el tiempo entre reintentos. Después de enviar un mensaje, espera la respuesta durante un tiempo, y si no llega, realiza varios reintentos. Si no se consigue ninguna respuesta, `clnt_call` devuelve un código de error. Si la RPC tiene éxito, `clnt_call` devuelve un cero. El resultado de la RPC se deposita en la dirección indicada en la llamada a `clnt_call` debidamente deserializada.

Programa del Servidor

```
/* archivador_svc.c
 * Please do not edit this file.
 * It was generated using rpcgen.
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include "archivador.h"

static void archivador_2();

main() /* Programa principal del servidor */
{
  register SVCXPRT *transp;
  (void) pmap_unset(ARCHIVADOR, VERSION_ACTUAL);

  transp = svcudp_create(RPC_ANYSOCK);
  if (transp == NULL) {
    fprintf(stderr, "cannot create udp service.");
    exit(1);
  }
  if (!svc_register(transp, ARCHIVADOR, VERSION_ACTUAL,
                    archivador_2, IPPROTO_UDP)) {
    fprintf(stderr, "unable to register (ARCHIVADOR,
              VERSION_ACTUAL, udp).");
    exit(1);
  }

  transp = svctcp_create(RPC_ANYSOCK, 0, 0);
  if (transp == NULL) {
    fprintf(stderr, "cannot create tcp service.");
    exit(1);
  }
  if (!svc_register(transp, ARCHIVADOR, VERSION_ACTUAL,
                    archivador_2, IPPROTO_TCP)) {
    fprintf(stderr, "unable to register (ARCHIVADOR,
              VERSION_ACTUAL, tcp).");
    exit(1);
  }

  svc_run();
  fprintf(stderr, ”Error: svc_run returned");
  exit(1);
  /* should never reach this point */
}
/* ... continua en la pagina siguiente */
```


El programa completo del servidor está compuesto por el programa principal (con el *stub* y el *dispatcher*), las rutinas para la serialización y las rutinas de servicio que debe escribir el programador del servidor.

El programa servidor que crea *rpcgen* esta compuesto por el programa principal (`main`) y el *dispatcher*, que recibe el nombre del servicio seguido de un subrayado y la versión (en nuestro ejemplo, `archivador_2`).

En esta primera parte del programa servidor veremos cómo se realiza el registro del servicio en el *binder*.

En primer lugar, se debe crear el *socket* con el que se van a atender las peticiones del cliente. Ya que la RPC de Sun soporta los dos protocolos de transporte TCP y UDP, se crea un *socket* para cada protocolo. Así, tenemos que hay una llamada a `svcudp_create` y otra a `svctcp_create`, de tal forma que cada una de ellas devuelve un descriptor de *socket*, con el que posteriormente se llama a la función de registro `svc_register`.

Una vez registrado el servidor, se debe pasar a recibir y servir peticiones. Esto se realiza llamando a la función `svc_run`, que consta de un bucle sin fin en el que se esperan peticiones del cliente. Cada vez que se recibe una petición del cliente le pasa la petición al *dispatcher* (`archivador_2`, en nuestro ejemplo).

Pasemos a la transparencia siguiente para ver nuestro *dispatcher*.

...Programa del Servidor

```
static void archivador_2(rqstp, transp)
  struct svc_req *rqstp;
  register SVCXPRT *transp;
{
  union { Args_Escritura escribir_2_arg;
          Args_Lectura leer_2_arg;
        } argument;
  char *result;
  bool_t (*xdr_argument)(), (*xdr_result)();
  char *(*local)();

switch (rqstp->rq_proc) {
  case NULLPROC:
    (void) svc_sendreply(transp, xdr_void, (char *)NULL);
    return;

  case ESCRIBIR:
    xdr_argument = xdr_Args_Escritura;
    xdr_result = xdr_void;
    local = (char *(*)()) escribir_2;
    break;

  case LEER:
    xdr_argument = xdr_Args_Lectura;
    xdr_result = xdr_Datos;
    local = (char *(*)()) leer_2;
    break;

  default:
    svcerr_noproc(transp);
    return;
  }
  bzero((char *)&argument, sizeof(argument));
  if (!svc_getargs(transp, xdr_argument, &argument)) {
    svcerr_decode(transp);
    return;
  }
  result = (*local)(&argument, rqstp);
  if (result != NULL && !svc_sendreply(transp, xdr_result, result)) {
    svcerr_systemerr(transp);
  }
  if (!svc_freeargs(transp, xdr_argument, &argument)) {
    fprintf(stderr, "unable to free arguments");
    exit(1);
  }
  return;
}
```


Aquí tenemos a nuestro *dispatcher*: `archivador_2`. Como vemos, recibe dos parámetros: `rqstp`, que contiene el código de la operación solicitada; y `transp`, que contiene los parámetros en formato externo XDR.

Básicamente consta de unas declaraciones locales, de una operación de selección `switch` y del envío de la respuesta al cliente. Este `switch` tiene tres opciones: dos correspondientes a las dos operaciones que ofrece servicio (`ESCRIBIR` y `LEER`), y una utilizada solamente para diagnósticos (`NULLPROC`) que simplemente devuelve un cero. Además de las tres opciones válidas dispone de la cláusula `default` para detectar un código de operación inválido.

En las opciones del `switch` correspondientes a las operaciones declaradas, primero se indica en `xdr_argument` la dirección de la rutina que sabe deserializar los parámetros de entrada, y en `xdr_result`, la que va a serializar el resultado de la llamada. A continuación establece en la variable `local` la dirección de la rutina de servicio correspondiente a la operación solicitada.

La última parte del *dispatcher*, a continuación del `switch`, inicializa a ceros tanto la variable que va a recibir los parámetros de entrada en formato nativo, como el resultado de la llamada. Mediante `svc_getargs`, se pasan los parámetros de entrada a formato nativo, depositándolos en la variable `argument`.

Seguidamente, la sentencia

```
result = (*local) (&argument, rqstp)
```

ejecuta la rutina de servicio.

El resultado se le devuelve al cliente mediante la función `svc_sendreply`, la cual también se encarga de serializar el resultado, en `transp`, mediante la rutina de serialización `xdr_result` que se le indica como parámetro.

Por último, se libera la memoria dinámica utilizada.

Al terminar el *dispatcher*, se vuelve a `svc_run` (desde donde se le llamó) a esperar otra petición.

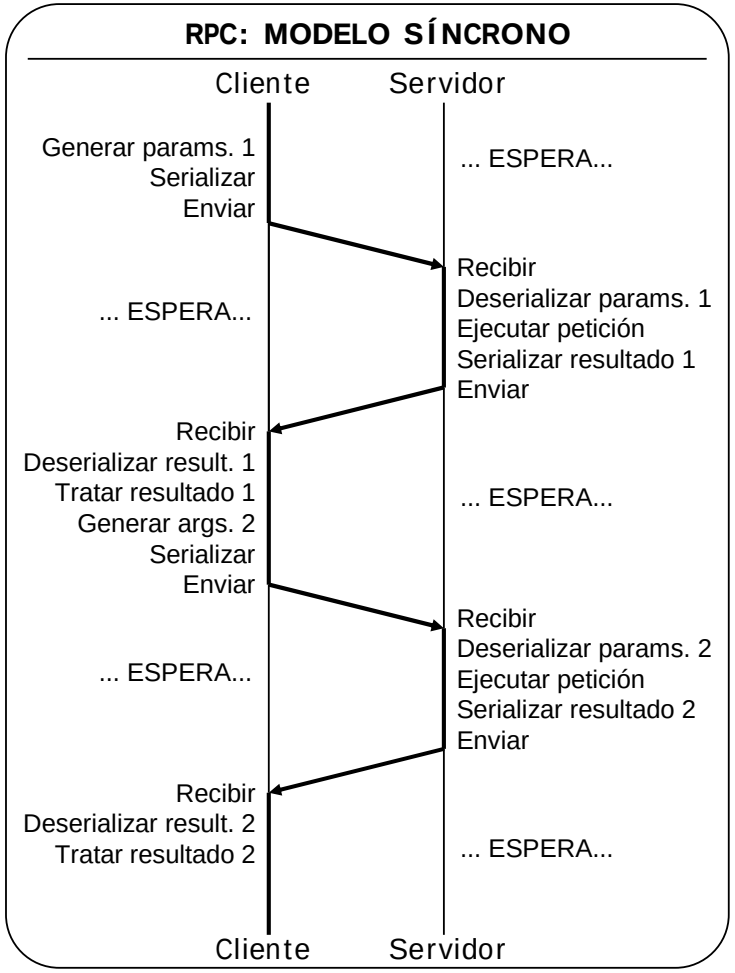
Utilidades de Bajo Nivel

- Herramientas de diagnóstico
  - Ping (procedimiento nulo)
  - Test de cod. op. inválidos
- Gestión de memoria dinámica en la serialización de los datos.
- Llamadas de *broadcast* a un servicio
- Llamadas en *batch* cuando no se espera respuesta.
- Rutinas *call-back* de cliente a servidor.
- Mecanismos de autenticación. (DES).

Las facilidades proporcionadas por las RPC de Sun descritas hasta ahora son suficientes para implementar la mayoría de los sistemas. No obstante en algunos casos pueden ser necesarias algunas utilidades que proporcionan un control adicional del programador sobre el sistema generado automáticamente. Aquí simplemente enumeraremos algunas de ellas.

- Llamadas remotas para comprobar si un servidor está activo o no. (procedimiento nulo).
- Llamadas remotas para comprobar códigos inválidos de operación.
- Gestión de memoria dinámica para una gestión más eficaz de la memoria en las rutinas de serialización.
- Llamadas de *broadcast* a todos los servidores de un servicio. `clnt_broadcast` implementa el mecanismo para *broadcast* comentado anteriormente en el apartado del *binding*.
- RPC asíncronas: Envío en modo *batch* de varias llamadas remotas que no requieren respuesta. Se envían mediante un buffer de salida, mejorando así la utilización de la red.
- Llamadas de tipo *call-back*. Esto permite indicarle al servidor ciertas rutinas remotas del cliente a las que debe llamar el servidor para completar la operación solicitada, convirtiéndose así, temporalmente, el cliente en servidor.
- Mecanismos de autenticación. Se utilizan pasar parámetros en las llamadas remotas que le permiten al servidor comprobar y autenticar la procedencia de la llamada y la veracidad de los parámetros. Utiliza el sistema DES (Data Encryption Standard).

RPC Asíncronas



Tras la llamada a una RPC, el cliente queda bloqueado hasta recibir el resultado. ⇒ ¡Pérdida de tiempo!

Las RPC que hemos tratado hasta ahora siguen el mismo modelo que las llamadas a procedimientos locales, es decir, son síncronas. Esto quiere decir que cuando el proceso cliente realiza la llamada, queda bloqueado hasta recibir la respuesta del servidor.

Esto puede crear cierta ineficiencia en el cliente, puesto que mientras está esperando la respuesta no puede hacer nada, mientras que en algunas situaciones, sería deseable que pudiese realizar otras tareas mientras espera la respuesta, esto es, un comportamiento asíncrono.

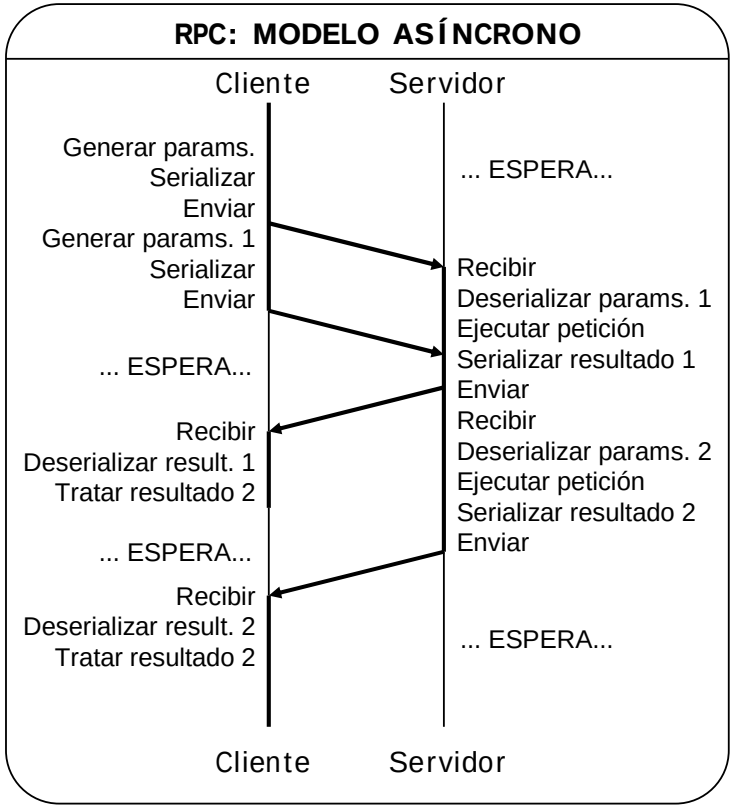
En el sistema distribuido de ventanas X-11, el cliente realiza muchas peticiones seguidas con poca información cada una y sin esperar su respuesta correspondiente, de tal forma que después de enviar una petición, se dedica a generar nuevos parámetros para realizar más peticiones, todo ello sin bloquearse en ningún momento. A medida que las respuestas le llegan al cliente, se van utilizando para representarlas en las ventanas convenientemente.

Si nos fijamos en los dos gráficos comparativos (el de esta transparencia y el de la siguiente), se puede ver que el modelo asíncrono es mucho más eficiente en el tiempo que el síncrono.

Gracias al modelo asíncrono, puede realizarse la misma petición a distintos servidores en paralelo, y luego quedarse esperando la respuesta del primero que llegue, o esperar más respuestas para hacer comparaciones.

Sistemas Distribuidos La Comunicación - 35

...RPC Asíncronas

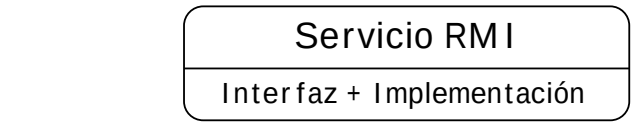


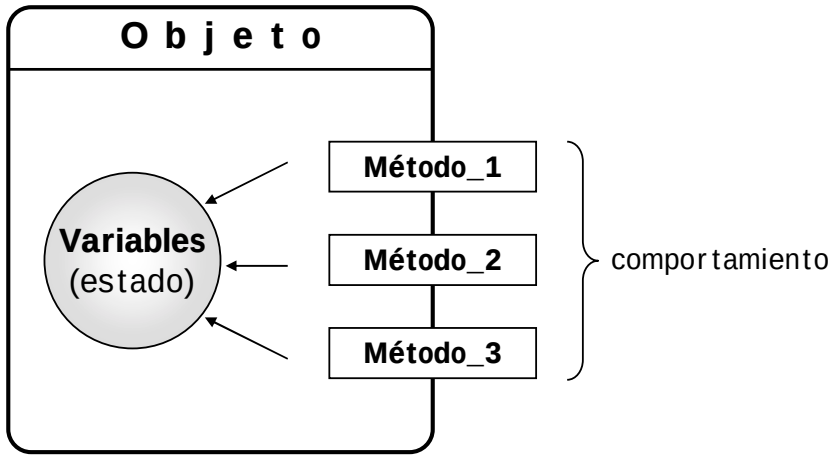
¡El mismo trabajo se realiza en menor tiempo!

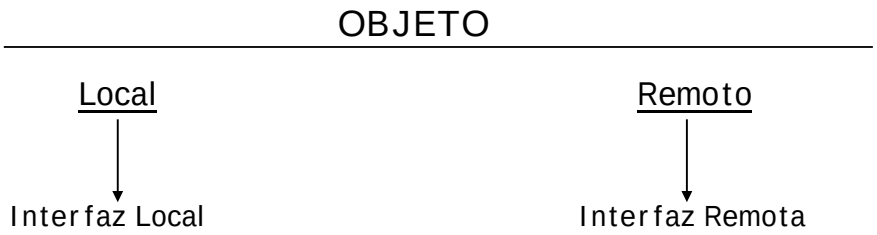
Esto puede hacerse cuando el cliente no necesita respuestas inmediatas para continuar su ejecución.

Sistemas Distribuidos La Comunicación - 36

RMI de Java







Llamadas a métodos remotos en Java: RMI (*Remote Method Invocation*).

La finalidad que persigue la llamada a método remoto es la misma que la perseguida en la llamada a procedimiento remoto (RPC): invocar de la manera más transparente posible un método o procedimiento de un servicio que reside en una máquina virtual distinta de la que reside el cliente. Téngase en cuenta que en la misma máquina física puede haber distintas máquinas virtuales de Java (JVM).

La diferencia entre estas dos tecnologías estriba básicamente en que mientras que las RPC se utilizan en diseños no orientados a objetos, RMI está soportado por el lenguaje orientado a objetos Java. Es decir, Java RMI es un *middleware* específico que permite a los clientes invocar a métodos de objetos como si estuviesen en la misma máquina virtual. RMI apareció en 1995 junto con la versión JDK 1.1 de Sun.

El modelo de objetos distribuidos de Java

El objetivo del modelo de objetos es la descripción de los conceptos y la terminología empleada en las llamadas a métodos remotos en el entorno Java.

Un servicio está formado por su interfaz, que define el conjunto de operaciones que va ofrecer dicho servicio, y por la implementación de dichas operaciones, soportada por los objetos del servicio.

Un objeto es una entidad identificable de manera única en todo el sistema, que cuenta con un estado y un comportamiento. El estado se mantiene mediante el uso de un conjunto de variables, mientras que su comportamiento se implementa mediante los métodos correspondientes a cada una de sus operaciones. Un método es una función asociada con un objeto, cuya ejecución generalmente modifica el estado del objeto.

Desde el exterior, el estado del objeto solo se puede cambiar a través de la invocación de ciertos métodos del objeto, conocidos como públicos, por eso se dice que un objeto es una entidad encapsulada.

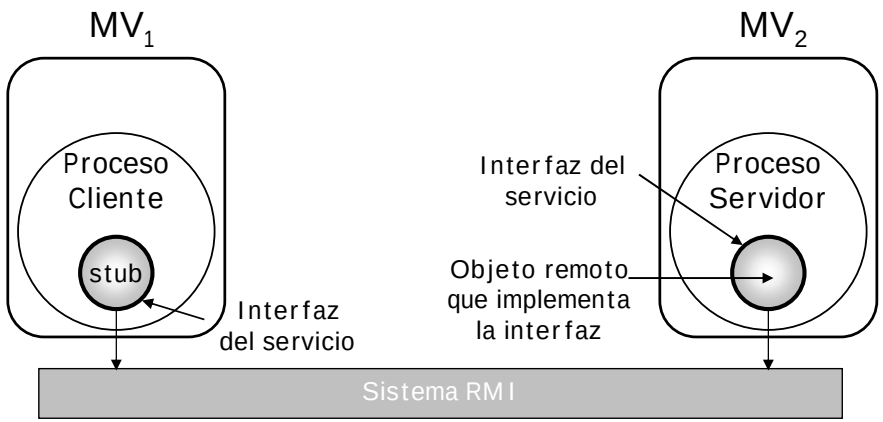
En Java existen dos tipos de objetos: los objetos locales y los objetos remotos.

Un objeto es local si sus métodos se invocan dentro de su máquina virtual. Es decir, por el proceso que creó dicho objeto o por los threads del proceso.

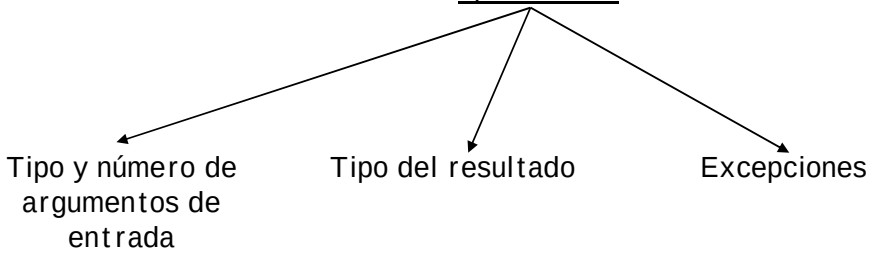
Un objeto es remoto si permite que sus métodos puedan invocarse por procesos que residen en otras máquinas virtuales.

Para que un objeto ofrezca públicamente métodos fuera de su máquina virtual es necesario que implemente una interfaz remota. Veamos a continuación las interfaces que ofrece Java.

Interfaz Remota



Interfaz: Declaración de las operaciones del servicio



Semántica de ejecución	
Sin fallos	Exactamente una
Con excepciones	Como mucho una

En Java existen dos tipos de interfaces: locales y remotas. Su objetivo es el mismo pues ambas describen servicios. Sin embargo, van orientadas a distinto tipo de clientes.

Si la **interfaz es local**, solamente es accesible por los procesos clientes dentro de la misma máquina virtual, pues no es visible fuera de esta máquina. Para permitir que clientes remotos accedan a un determinado servicio es necesario que la interfaz sea remota.

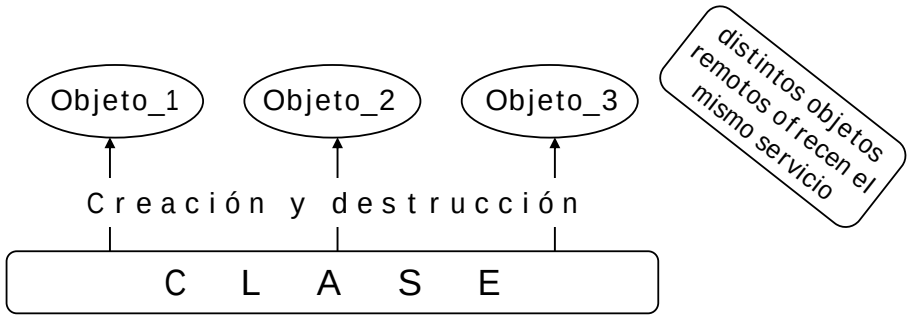
Una **interfaz remota** recoge la declaración en Java de las operaciones que conforman un servicio.

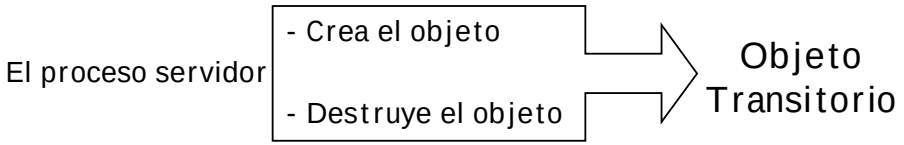
Por cada operación se especifica el tipo y el número de los argumentos de entrada y el tipo del resultado que devuelve, así como las excepciones que pueden producirse si hay errores de ejecución o de comunicación.

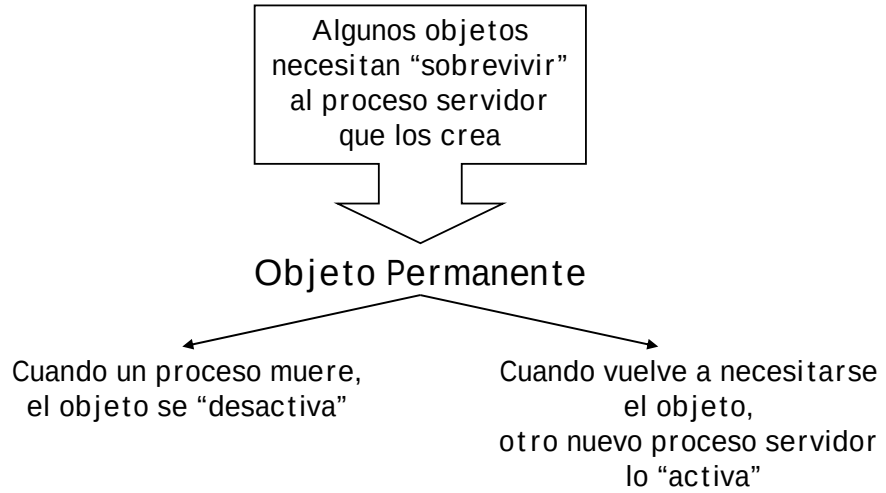
Los parámetros que se pasan en la invocación son solo de entrada y se pasan siempre por copia. Los parámetros o el resultado de la operación (si lo hay), pueden ser de tipos primitivos o de tipo "objeto", es decir tipos abstractos de datos creados por el usuario.

La semántica de ejecución de cada operación, en ausencia de fallos es "exactamente una", mientras que si se produce alguna excepción, la semántica es "como mucho una", ya que RMI se apoya siempre sobre TCP.

Implementación de un Objeto Remoto







Como ya se ha comentado, la interfaz de un servicio es la descripción del conjunto de métodos u operaciones asociadas con dicho servicio. Por tanto, para que se pueda ofrecer realmente el servicio, es necesario construir una clase que implemente cada uno de los métodos que se describen en dicha interfaz.

A los objetos de una clase que implementa un servicio remoto se les conoce como objetos remotos. En RMI puede haber más de un objeto que implemente simultáneamente el mismo servicio.

Un objeto remoto, por sí solo, no tiene vida propia por lo que es necesario que un proceso, conocido como *servidor*, lo cree y, posteriormente, lo destruya cuando ya no sea necesario.

Normalmente la vida de un objeto remoto se circunscribe a la vida del proceso servidor que lo creó. Este tipo de objetos se denominan **transitorios**, sin embargo, hay veces que interesa que el objeto remoto sobreviva cuando el proceso servidor muera. A éstos últimos se les conoce como objetos **permanentes**.

Un ejemplo de objetos permanentes son los objetos que soportan cuentas bancarias, ya que deben existir mientras exista la cuenta asociada. Sin embargo, como pueden existir miles de cuentas bancarias simultáneamente, sería imposible mantener simultáneamente miles de procesos servidores, uno por cada objeto que representa el estado de una cuenta.

En esta situación, interesa que el objeto remoto sobreviva al servidor que lo creó, de tal manera que el estado del objeto pueda salvarse en disco (desactivarlo) cuando el proceso muera y cargarse de nuevo (activarlo), mediante otro proceso servidor distinto, en el momento en que se vuelva a necesitar.

Por último, se debe indicar que un mismo servidor puede dar de alta y soportar simultáneamente varios objetos remotos que implementen, bien el mismo servicio, por ejemplo con diferentes calidades de implementación, o bien, diversos objetos que implementen servicios distintos.

Modelo de Ejecución

Proceso Servidor

```
Crear objeto del servicio
Registrar Servicio
Esperar peticiones
    Recibir petición
    Tomar parámetros petición
    Averiguar método apropiado
    Objeto.método_apropiado (parámetros)
    Construir mensaje respuesta
    Copiar al mensaje resultado del método
    Enviar mensaje de respuesta
Forever
```

Proceso Cliente

```
Objeto := Solicitar servicio a Servidor de Nombres
Resultado := Objeto.operación (parámetros)
```


Veamos ahora cómo se lleva a cabo la invocación de un método de un servicio remoto determinado.

Como ya se ha comentado, para ofrecer un servicio, es necesario que un proceso servidor cree el objeto remoto que implementa el servicio a ofrecer. Sin embargo, el hecho de crear un objeto no hace que éste sea visible al exterior automáticamente, por lo que es necesario que el servidor dé de alta el servicio asociado al objeto en el *servicio de nombres de Java*.

El servicio de nombres registra el nombre del servicio junto con una referencia de objeto. Esta referencia sirve como identificador de comunicación del objeto remoto. Una vez dado de alta el servicio, el objeto puede servir peticiones mientras el proceso servidor no decida darlo de baja en el servicio de nombres.

Supuesto que el cliente conoce el nombre del servicio que quiere usar, necesita saber si hay algún objeto que soporte dicho servicio, lo cual se consigue preguntándoselo al servicio de nombres.

Si hay algún objeto que implemente dicho servicio, el servicio de nombres le devolverá la referencia de dicho objeto. Con esta referencia, el cliente llama a las operaciones del servicio (los métodos del objeto) con la misma sintaxis que si llamara a un objeto local, esto es `referenciaDelObjeto.metodo`.

Los métodos son síncronos por lo que el proceso cliente se bloquea hasta que el método llamado devuelve el resultado o simplemente finaliza.

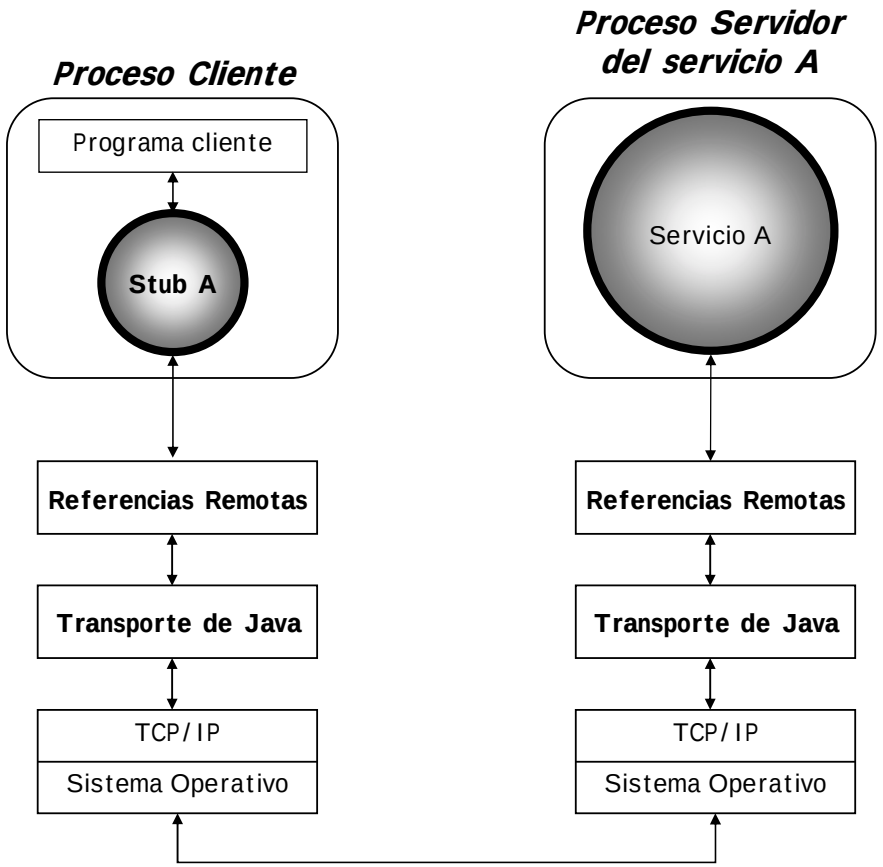
El servidor, por su parte, cuando recibe una petición sobre el servicio que ofrece, averigua qué método debe invocarse, toma los parámetros de entrada de la petición y llama al método correspondiente. A la finalización del método, construye un mensaje de respuesta donde copia el resultado que le devolvió el método (o las excepciones, si se producen errores).

Debe quedar claro que un objeto remoto solo reside en el proceso servidor, de tal manera que los clientes comparten el mismo objeto. Es decir, los clientes tienen solo la referencia al objeto remoto, estando el objeto y, por tanto, su estado únicamente en la máquina del servidor.

Como puede verse, este modelo de petición es el típico cliente/servidor en el que el *stub* o representante debe tener un conocimiento exacto de la ubicación y del tipo de servicio solicitado.

Seguidamente veremos la arquitectura que se necesita para que este modelo funcione correctamente.

Arquitectura de RMI



Capas de RMI

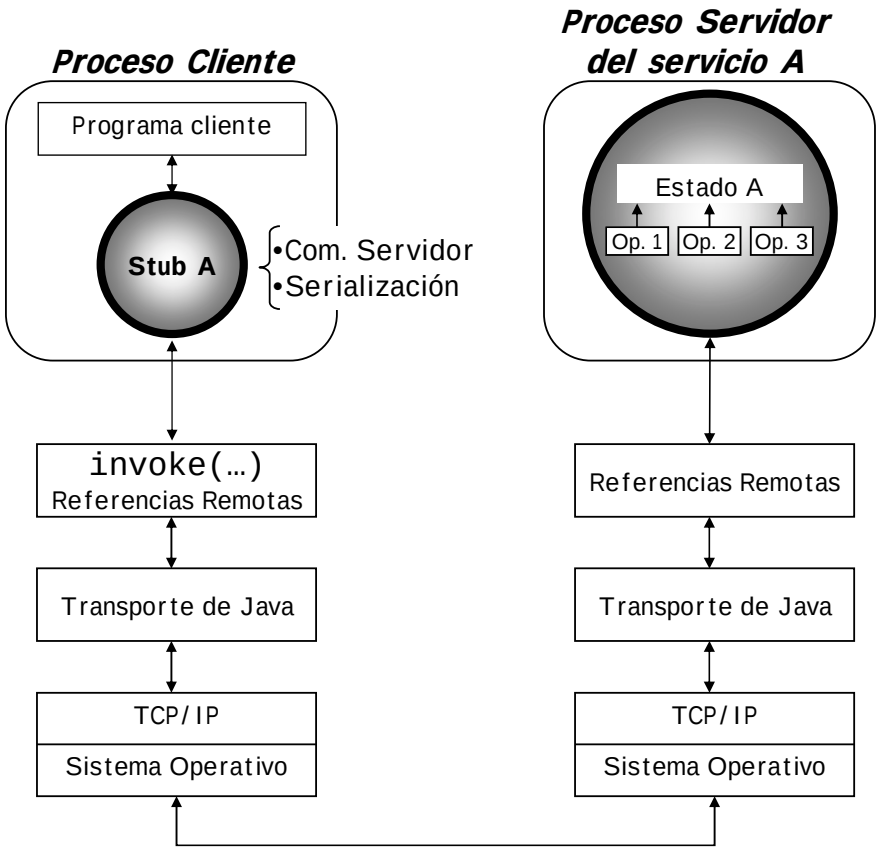
- ✓Stub
- ✓Referencias Remotas
- ✓Transporte de Java

La arquitectura de RMI que soporta Java 2 se compone de tres capas:

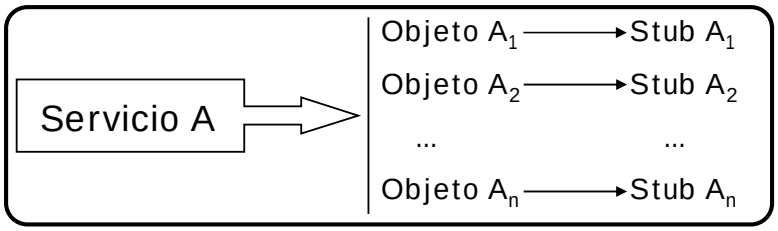
- capa de stub
- capa de referencias remotas
- capa de transporte.

Veámoslas a continuación una por una.

Capa del Stub



El *stub* se obtiene mediante
- → `rmic`
- → Carga dinámica



Capa de stub

Como se ha comentado antes, el cliente solo conoce la interfaz del servicio que quiere invocar, pero esa interfaz, por ser una mera descripción, no tiene utilidad si no está soportada por un objeto; por otra parte también hemos dicho que el objeto real que implementa el servicio solo reside en el servidor. Por tanto, el objeto que soporta la interfaz del cliente debe encargarse, de alguna manera, de hacer llegar las peticiones al objeto remoto y recoger las respuestas. Este objeto se conoce como representante del objeto remoto, o más comúnmente *stub* o *proxy*.

El *stub* se encarga de establecer la comunicación con el servidor, controlar dicha comunicación y realizar las operaciones de serialización y deserialización de parámetros y del resultado. Todas estas operaciones las hace apoyándose en los servicios que ofrecen las capas inferiores.

Cuando el cliente invoca un método del servicio, el *stub* recoge dicha llamada, extrae el nombre del método y los parámetros, los serializa y llama al método `invoke` con dicha información. El método `invoke` pertenece a la capa de referencias remotas, y su función es similar a la de `clnt_call` en RPC, ya que a cualquier método remoto de cualquier servicio se le llama a través de `invoke`.

La llamada a `invoke` es síncrona, por lo que el stub se queda esperando la respuesta. Cuando el *stub* recibe la respuesta la deserializa y la devuelve al cliente como respuesta del método que invocó.

El código del *stub* no tiene que escribirlo el programador sino que puede obtenerse por dos vías: mediante el uso del compilador de interfaces o mediante carga dinámica.

El compilador de interfaces de java (***rmic***) toma como entrada la implementación de una interfaz remota en Java y genera el *stub* del cliente en Java. Obviamente, la obtención del *stub* debe realizarse previamente a la ejecución del cliente.

La **carga dinámica** consiste en que el cliente, ya en ejecución, carga de la máquina servidora (más concretamente, del servicio de nombres de la máquina servidora) el *stub* del objeto remoto que desea utilizar.

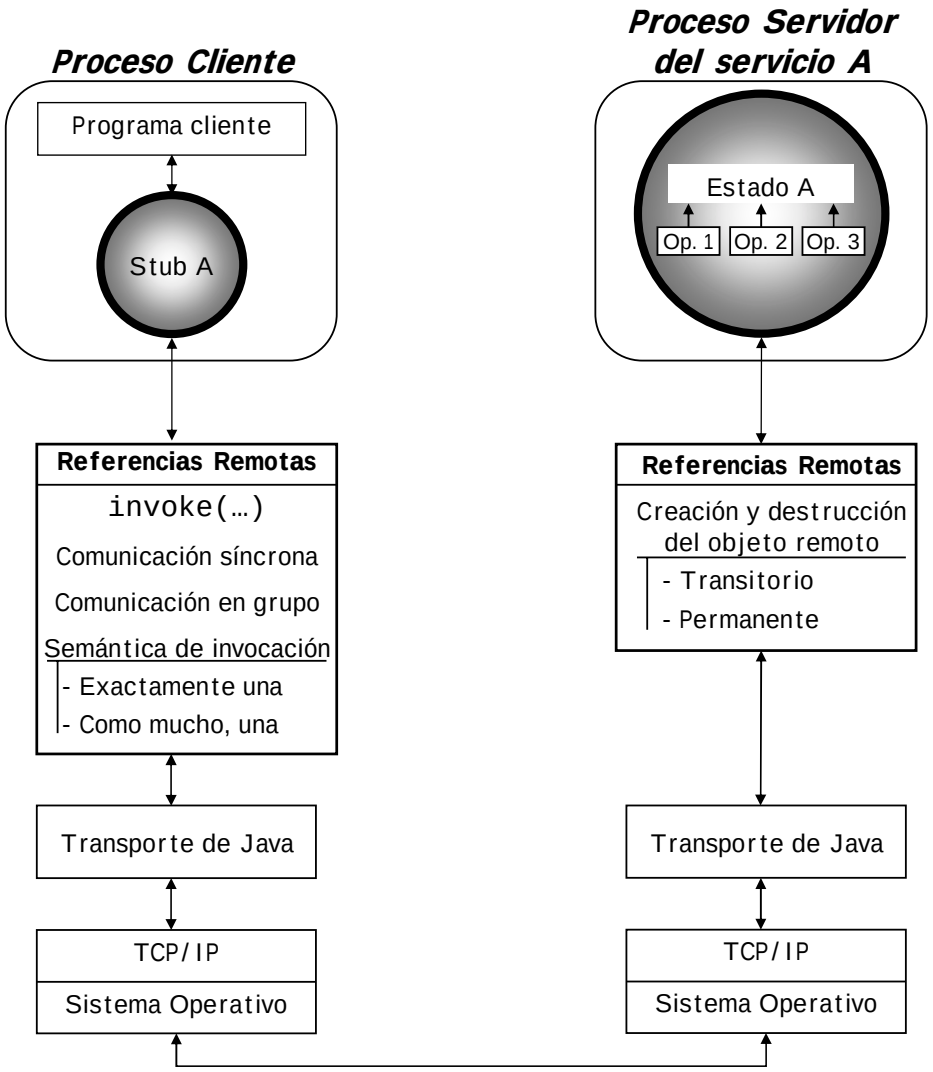
Hay que destacar que el cliente necesita un *stub* para cada objeto remoto que use. Es decir, si un cliente se conecta con dos objetos remotos que implementan el mismo servicio, necesita dos *stubs* distintos que accedan a su objeto correspondiente, puesto que esos dos objetos remotos posiblemente tienen estados distintos aunque implementen la misma interfaz, es decir, el mismo servicio.

Por ejemplo, con una misma interfaz bancaria, se pueden tener dos objetos remotos que representen el estado de dos cuentas de mismo banco, pertenecientes al mismo titular. En este caso el cliente necesitaría dos *stubs*.

Hasta la versión 1.1 de Java había que generar tanto el *stub* del cliente como el *stub* o esqueleto del servidor. Cuando llegaba una petición al *stub* servidor, se deserializaba y se llamaba al método adecuado del objeto remoto. Por último, se serializaba el resultado y se construía el mensaje de respuesta que se envía al cliente.

A partir de Java 2, ya no hace falta la generación del esqueleto del servidor, pues la localización del método a invocar se hace con una técnica de programación conocida como "reflexión". Esta técnica permite, en tiempo de ejecución, descubrir la naturaleza del objeto, por ejemplo a qué clase pertenece, las operaciones que soporta, etc. Nosotros nos vamos ha centrar en la versión más moderna, en la que no se requieren los esqueletos.

Referencias Remotas



Capa de referencias remotas

Esta capa ofrece todas las operaciones relacionadas con el ciclo de vida de un objeto remoto: creación, semántica de invocación de sus métodos y destrucción.

La creación y destrucción son operaciones que afectan al servidor, mientras que la semántica de invocación afecta a los clientes.

En cuanto al servidor, esta capa ofrece la creación de dos tipos de objetos remotos: los objetos transitorios, que viven mientras viva el servidor que los crea y los objetos persistentes que sobreviven al proceso servidor que los crea. Como resultado de la creación, esta capa le proporciona al servidor un **identificador de comunicación** o referencia remota del objeto creado, identificador que utiliza para registrar el objeto creado en el servicio de nombres.

Si el servidor crea un **objeto transitorio**, la referencia remota será transitoria y por tanto válida para el cliente siempre que esté vivo el objeto y el proceso que creó el objeto.

Si el servidor crea un **objeto permanente**, esta capa devuelve una referencia permanente que el cliente podrá utilizar siempre que el objeto exista, tanto si está activo (cargado en memoria) como si está inactivo (almacenado en disco). Cuando el proceso que creó el objeto finaliza, el objeto se almacena en disco, hasta que un cliente lo solicite, momento en que se vuelve a cargar en memoria.

Para que un cliente acceda a un objeto remoto necesita siempre la referencia a ese objeto. Esta referencia la obtiene del servicio de nombres, al preguntar por el servicio que soporta dicho objeto.

Con esta referencia el cliente puede utilizar dos tipos de semánticas de invocación a métodos remotos: la comunicación cliente/servidor y la comunicación en grupo.

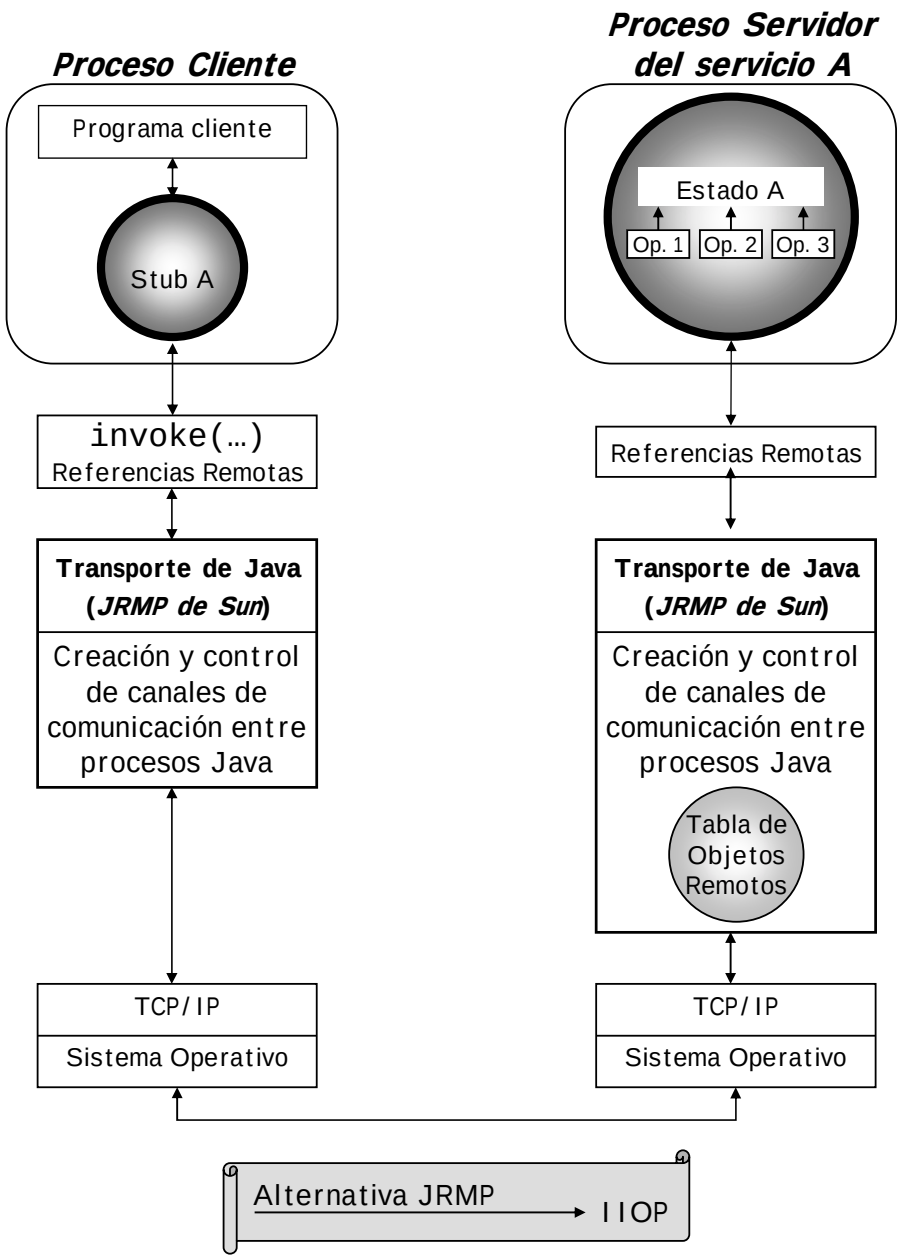
La **comunicación cliente/servidor** es la comunicación síncrona clásica, descrita anteriormente en el modelo de ejecución. La semántica de invocación es "exactamente una" si no hay errores y "como mucho una" en presencia de errores.

La **comunicación en grupo** permite que el stub del cliente realice peticiones a más de un objeto servidor simultáneamente, de tal manera que el cliente toma como válida la primera respuesta que reciba.

Previamente a cualquier invocación, el cliente debe crear el stub del servicio (con la sentencia `new objeto`). En ese momento el stub establece un canal de comunicación entre el cliente y el servidor por donde se enviarán los mensajes de petición originados por las operaciones `invoke`, así como las respuestas correspondientes. Es decir, se crea un canal por cada cliente de un objeto remoto. Si un proceso cliente es cliente de dos objetos remotos simultáneamente, tendrá abiertos dos canales de comunicación distintos, uno para cada objeto servidor.

Para crear y manipular un canal de comunicación, tanto el cliente como el servidor se apoyan en las funciones que ofrece la capa de transporte que se va a comentar a continuación.

Transporte de RMI



La **capa de transporte** de RMI permite el establecimiento y control de un canal de comunicación virtual orientado a la transmisión de flujo de bytes entre dos o más procesos. Es un protocolo propiedad de Sun, conocido como *Java Remote Method Protocol* (JRMP).

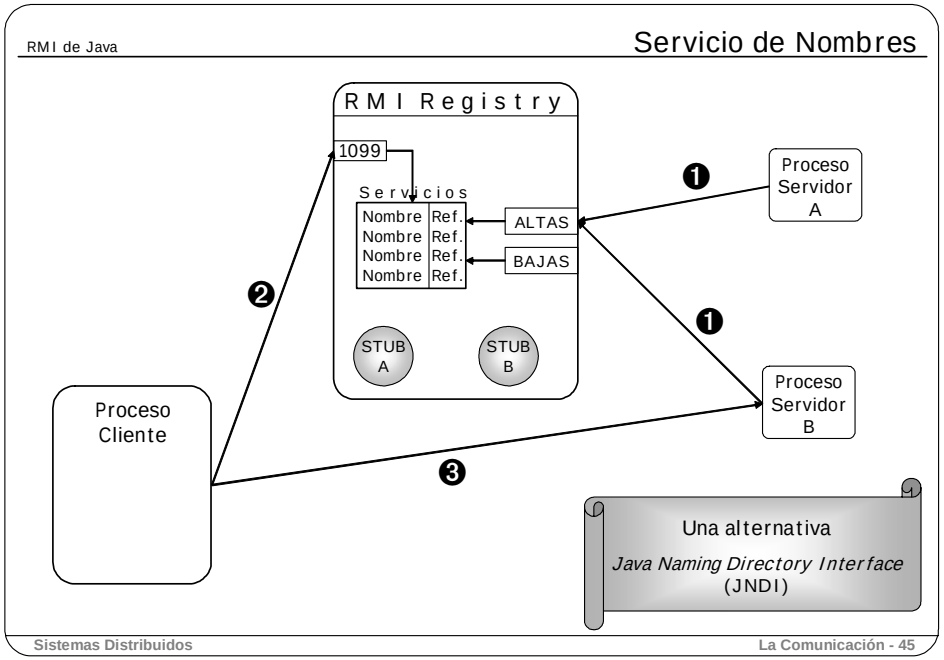
Este protocolo, por construcción, obliga a que el sistema operativo subyacente tenga una interfaz de transporte del tipo TCP/IP.

Esta capa de transporte de RMI crea una nueva conexión entre dos procesos Java siempre que residan en máquinas virtuales distintas, aunque residan en la misma máquina física.

Las funciones principales que ofrece está capa son, como ya se ha comentado, la creación de canales de comunicación, así como su monitorización para comprobar que las conexiones siguen vivas y funcionando correctamente. También se encarga de detectar la llegada de nuevos mensajes, ya sean de petición o respuesta y de mantener, en cada máquina virtual, una tabla de los objetos remotos que residen en ella.

El inconveniente de que el protocolo JRMP sea propiedad de Sun es que no es abierto, por lo que tiene ciertas dificultades para comunicarse con otros *middleware* como, por ejemplo, CORBA. Para solucionar este problema, Sun ofrece la posibilidad de usar un protocolo abierto como el IIOP (Internet InterORB Protocol) en lugar de JRMP.

Servicio de Nombres



RMI ofrece un servicio de nombres, conocido como ***RMI registry***, para la localización de servicios. Este servicio de nombres le ofrece al servidor operaciones típicas como el dar de alta un servicio o darlo de baja, mientras que al cliente le ofrece la localización de un servicio.

Cuando un servidor ha creado un objeto remoto lo hace visible al exterior dándolo de alta en el servicio de nombres.

El servicio de nombres registra pares (*nombre del servicio, referencia del objeto remoto*).

Una **referencia remota** es un identificador único formado por la dirección de Internet de la máquina donde reside el objeto remoto y un punto final de conexión que es el número de puerto TCP por donde escucha un *thread* que espera peticiones para ese objeto.

En la referencia remota también se puede almacenar el **código del *stub*** para el cliente del objeto remoto, por si un cliente necesita dicho *stub* en tiempo de ejecución y quiere cargarlo dinámicamente.

El servicio de nombres es un servicio que puede arrancarse en cualquier máquina y lo servicios pueden registrarse en cualquier máquina, no obligatoriamente en la misma máquina que el servidor del servicio. Aunque pueden existir varios servidores de nombres en un mismo sistema distribuido, lo normal es disponer de un servidor de nombres ubicado en un ordenador conocido por el resto de las máquinas que forman el sistema distribuido. También es conveniente que haya algún servidor más de reserva. El servidor de nombres, por defecto, escucha por el puerto 1099. En la clase `java.rmi.registry` está definida la constante **`REGISTRY_PORT`** con el valor 1099.

En la figura adjunta se muestra el esquema de utilización de servicios a través del servidor de nombres. En primer lugar el servidor debe registrar el servicio en el servidor de nombres. Una vez hecho esto, cualquier cliente puede solicitarle al servidor de nombres la referencia de un servicio y, una vez que la tiene, puede utilizar el servicio a través de la referencia remota.

Téngase en cuenta que proceso cliente, el proceso servidor y el servidor de nombres pueden estar en la misma máquina física (en distintas máquinas virtuales de Java) o, como es normal, residir en ordenadores distintos.

Si no se desea usar el servicio de *RMI registry*, también puede usarse el servicio de directorio *Java Naming Directory Interface* (JNDI), que permite conectarse con servidores de nombres heterogéneos.

Un Ejemplo

Definición del Servicio

Servicio: División de dos números

Objetos transitorios

Comunicación cliente/servidor

Stub generado por **`rmic`**

Protocolo de transporte de Java (JRMP)

Soporte de desarrollo: Java 2 SDK

Construcción del Servicio

1. Definir su interfaz
2. Implementar la clase que soportará la interfaz
3. Construir el proceso servidor
4. Crear el objeto remoto del servicio

Uso del Servicio

1. Generar el *stub* del servicio
2. Construir el proceso cliente

Vamos a describir ahora un ejemplo sencillo de llamada a método remoto mediante un servicio que ofrece la división de dos números.

Dicho servicio se va a crear con las siguientes características:

- Los objetos servidores serán transitorios, destruyéndose, por tanto, cuando finalice el proceso que los creó
- La semántica de comunicación es cliente/servidor (no es “en grupo”).
- El *stub* se generará con el compilador de interfaces `rmic`.
- Se usará el protocolo de transporte de Java JRMP
- El soporte para el desarrollo es Java 2 SDK de Sun

Los pasos que se van a seguir son los siguientes:

- Para construir el servicio:
  1. Primero se define su interfaz
  2. Después se implementa la clase que soportará dicha interfaz
  3. Por último se construye el proceso servidor que creará el objeto remoto de dicho servicio.
- Para usar el servicio, en el lado del cliente:
  1. Se generará el *stub* del servicio
  2. Se construirá el cliente que use el objeto remoto.

El Servicio y la Clase

Definición del Servicio

```
// Calculador.java
public interface Calculador
        extends java.rmi.Remote{
  public long div (long a, long b)
    throws java.rmi.RemoteException;
}
```

Construcción de la Clase

```
// CalculadorBasico.java
public class CalculadorBasico
        implements Calculador{
  public long div (long a, long b)
    throws java.rmi.RemoteException{
      return a/b;
  }
}
```

Definición del servicio

El servicio que ofrece la división de dos números podría ser el que se describe en la figura de la izquierda.

Las interfaces remotas deben heredar siempre la clase `Remote` y debe asociarse a cada firma de método una excepción `RemoteException`. Este tipo de excepción se producirá siempre que haya errores de comunicación entre cliente y servidor.

Construcción de la clase que implementa dicha interfaz.

Se observa que `CalculadorBasico` implementa la interfaz `Calculador`, con lo que los objetos que se creen instanciando `CalculadorBasico` serán remotos, pues `Calculador`, a su vez, hereda la clase `java.rmi.remote`.

Como veremos, el proceso servidor deberá crear objetos de la clase `CalculadorBasico` para ofrecer el servicio correspondiente.

Sistemas Distribuidos La Comunicación - 47

El Servidor

```
// ServidorCalculos.java
import java.io.*;
import java.rmi.UnicastRemoteObject;
import java.rmi.registry.*;
import java.rmi.server.*;
public class ServidorCalculos{
  public static void main(String args[]) {
  try {
    // 1. Se crea un objeto remoto
    Calculador calc = new CalculadorBasico();

    // 2. Se prepara el mecanismo para recibir peticiones
    Calculador refStub =
      (Calculador) UnicastRemoteObject.exportObject(calc);

    // 3. Se localiza el registro de nombres
    Registry r = LocateRegistry.getRegistry(“localhost”, 1099);

    // 4. Se da de alta el servicio Calculator
    r.rebind("rmi://localhost/ServicioCalculador", refStub);

    // 5. Se espera leer “EXIT” de la entrada estandar
    //    para finalizar el servicio
    BufferedReader rdr =
       new BufferedReader(new InputStreamReader(System.in));
    while (true) {
      System.out.println("Teclee EXIT para finalizar.");
      if ("EXIT".equals(rdr.readLine())); {
        break;
      }
    }

    //6. Se da de baja el servicio
    r.unbind("rmi://host/ServicioCalculador");

    //7. Se cierra la comunicacion
    UnicastRemoteObject.unexportObject(calc, true);
  }

  //8. Se capturan las excepciones
  catch (Exception e){
    e.printStackTrace();
  }
 }
}
```


En la figura se muestra el código del proceso servidor que va a crear un objeto del servicio y lo va hacer visible al exterior.

En el **paso 1**, se crea el objeto que soportará el servicio a partir de la clase `CalculatorBasico`.

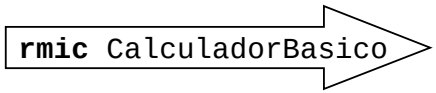
Además, este objeto debe ser capaz de recibir peticiones y enviar respuestas a los clientes, por lo que, en el **paso 2**, el método `exportObject` (de la clase `UnicastRemoteObject`) asocia el objeto remoto `calc` al mecanismo de comunicación de RMI para poder recibir peticiones de clientes y devolverles el resultado.

Además, la clase `UnicastRemoteObject` establece que el objeto remoto será transitorio, es decir, vivirá, como mucho, hasta que el servidor muera o lo destruya. Si se deseara que el objeto fuese permanente, esto es, que sobreviviera al servidor, entonces se debería utilizar la clase

`java.rmi.activation.Activatable`

Una vez establecido el soporte de comunicación, se va a hacer que el servicio sea visible al exterior. Para ello, en el **paso 3**, se localiza el identificador de comunicación del servicio de nombres (`rmiRegistry`) de la máquina local, para dar de alta en el servidor de nombres, en el **paso 4**, el servicio `ServicioCalculador` y el *stub* del objeto remoto que lo soporta: `refStub`.

El Cliente




```
// ClienteDelCalculador.java
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;
public class ClienteDelCalculador {
  public static void main(String[] args) {
    try {
      // 1. Se localiza el servicio
      Calculador calc =(Calculador)Naming.lookup(
           "rmi://host:1099/ServicioCalculador");
      // 2. Se invocan los métodos del servicio
      System.out.println(calc.div(4,2));
      System.out.println(calc.div(9,0));
    }
    //3. Se capturan las excepciones
    catch (Exception e){
      e.printStackTrace();
    }
  }
}
```


Construcción del cliente

Antes de comenzar la construcción del cliente, se debe someter al compilador de interfaces la clase `CalculadorBasico` que implementa la interfaz del servicio `Calculador`, con lo que se obtiene el fichero del *stub* `CalculadorBasico_stub.class`.

Una vez que se tiene el *stub* se construye un programa cliente, como se describe a continuación.

En el **paso 1** el cliente pregunta al servicio de nombres cuál es el identificador de comunicación del servicio `ServicioCalculator`. Si el registro de nombres lo encuentra devuelve el identificador de comunicación (disfrazado de objeto calculador) que sirve para invocar sus métodos remotos correspondientes, como se ve en el **paso 2**.

Obsérvese cómo en el nombre del servicio que se busca, se incluye un nombre de máquina *host* y un puerto. Si se omite el nombre de host, se toma, por defecto, la máquina local. Si se omite el puerto, se toma, por defecto el `REGISTRY_PORT`, es decir, el 1099.

En el **paso 3** se capturan las excepciones que se produzcan tanto por errores de transmisión como de ejecución del servidor, en cuyo caso se imprimen y se indica en qué métodos se han producido.

Otra Alternativa

Definición del Servicio

```
// Calculador.java
public interface Calculador
        extends java.rmi.Remote{
  public long div (long a, long b)
    throws java.rmi.RemoteException;
}
```

Construcción de la Clase

```
// CalculadorBasico.java
public class CalculadorBasico
  extends java.rmi.server.UnicastRemoteObject
    implements Calculador{
  public long div (long a, long b)
    throws java.rmi.RemoteException{
    return a/b;
  }
  public CalculatorBasico ()
    throws java.rmi.RemoteException{
    super();
}
```


Por lo que hemos visto hasta ahora, la utilización del mecanismo RMI es bastante similar a las RPC, aunque en la parte del cliente queda más sencilla y transparente. Sin embargo, la construcción del objeto remoto y el proceso servidor es bastante similar a la de las RPC, no obstante, esto puede simplificarse significativamente. Veamos la construcción alternativa que se presenta.

En esta alternativa, la definición del interfaz permanece inalterado, como parece razonable, pero para facilitar la creación del objeto remoto en el proceso servidor sea más transparente vamos a modificar ligeramente la construcción de la clase que implementa la interfaz. Para ello simplemente hay que heredar la clase `java.rmi.server.UnicastRemoteObject`. También debemos definir el método constructor de la clase, que recibe el mismo nombre que ésta: `CalculadorBasico`, y en él simplemente se llama al constructor de la clase que se hereda, es decir, de `java.rmi.server.UnicastRemoteObject`. Este último constructor llamará al su método `export`, consiguiendo el mismo efecto que cuando, en la versión anterior, en el proceso se llamaba a `UnicastRemoteObject.export` después de crear el objeto remoto.

El constructor de una clase que lleva el mismo nombre que ella, se ejecuta automáticamente al crear objetos mediante `new`.

...Otra Alternativa

El Servidor

```
// ServidorCalculos.java
import java.io.*;
import java.rmi.UnicastRemoteObject;
import java.rmi.Naming;
public class ServidorCalculos{
  public static void main(String args[]) {
  try {
    // 1 y 2. Se crea un objeto remoto
    // y queda listo para recibir peticiones
    Calculador calc = new CalculadorBasico();

    // 3 y 4. Se localiza el registro de nombres y
    // se da de alta el servicio de calculos
    Naming.rebind("rmi://host:1099/ServicioCalculador",
                  calc);

    // 5. Se espera leer "EXIT" de la entrada estandar
    //    para finalizar el servicio
    BufferedReader rdr =
       new BufferedReader(new InputStreamReader(System.in));
    while (true) {
      System.out.println("Teclee EXIT para finalizar.");
      if ("EXIT".equals(rdr.readLine())); {
        break;
      }
    }
    //6. Se da de baja el servicio
    Naming.unbind("rmi://host:1099/ServicioCalculador");

    //7. Se cierra la comunicacion
    UnicastRemoteObject.unexportObject(calc, true);
  }
  //8. Se capturan las excepciones
  catch (Exception e){
    e.printStackTrace();
  }
 }
}
```


Gracias a las modificaciones hechas en la implementación de la clase del objeto remoto, el proceso servidor solamente debe ocuparse de crear el objeto, sin necesidad de exportarlo (como antes lo hacía explícitamente), pues como ya hemos comentado, la creación de un objeto implica la ejecución del constructor asociado, el cual llama a su vez, mediante `super()`, a `UnicastRemoteObject.export`.

También se puede simplificar el registro del servicio en el servidor de nombres. En la versión anterior, primero había que localizar el servidor de nombre a utilizar (`Registry r = LocateRegistry.getRegistry`), y una vez que se tenía una referencia a él, se daba de alta el servicio:

(`r.rebind("rmi://host/ServicioCalculador", calc)`).

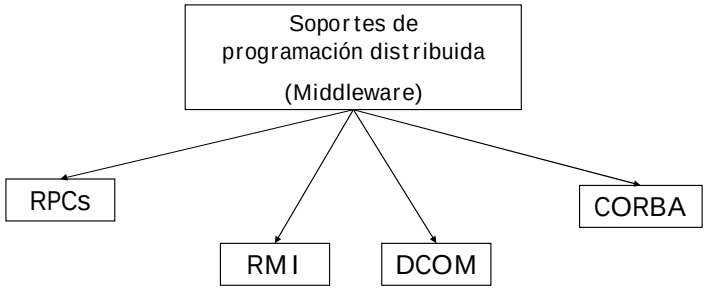
Esto puede abreviarse mediante `java.rmi.Naming.rebind`, lo cual da de alta el servicio localizando previamente el servidor de nombres indicado, abstrayendole al usuario de tal trabajo.

Para dar de baja el servicio, también utilizaremos la clase `Naming`. Mientras que la creación del mecanismo y de los recursos de comunicación se realiza de manera transparente, a la hora de devolverlos hay que hacerlo explícitamente, mediante `UnicastRemoteObject.unexportObject`, lo cual crea una asimetría con el mencionado mecanismo transparente para su creación.

¿Qué es CORBA?

CORBA

¿Qué es CORBA?



El más destacado es CORBA:

Common **O**bject **R**equest **B**roker Arquitecture

Sistemas Distribuidos La Comunicación - 52

¿Qué es CORBA? CORBA o **C**ommon **O**bject **R**equest **B**roker **A**rquitecture es modelo de soporte de programación distribuida al igual que lo es el modelo de llamada a procedimiento remoto.

Hay otros modelos como DCOM de Microsoft o java RMI para realizar llamadas a métodos remotos. El grado de soporte de la distribución es muy distinto en estos modelos siendo CORBA, hoy por hoy, el soporte más moderno de programación distribuida.

Se denomina *middleware* al conjunto de modelos que ofrecen un entorno de programación distribuida, sencillo, consistente e integrado que facilita las tareas de diseño, programación y gestión de aplicaciones distribuidas.

En esencia, el *middleware* es una capa de software que se coloca “entre medias” del usuario y del entorno distribuido, abstrayendo al usuario de la complejidad y heterogeneidad de las diferentes arquitecturas de las máquinas, protocolos de comunicación, sistemas operativos y lenguajes de programación.

Hay diferentes tipos de middleware, pero el más usual es el middleware orientado a objetos, que facilita que el cliente haga llamadas a métodos o procedimientos remotos de manera transparente. El modelo CORBA cae dentro de este tipo.

...¿Qué es CORBA?

MODELO para el desarrollo de sistemas distribuidos orientados a objetos

- Nace en 1991. Actualmente se va por la versión 3.

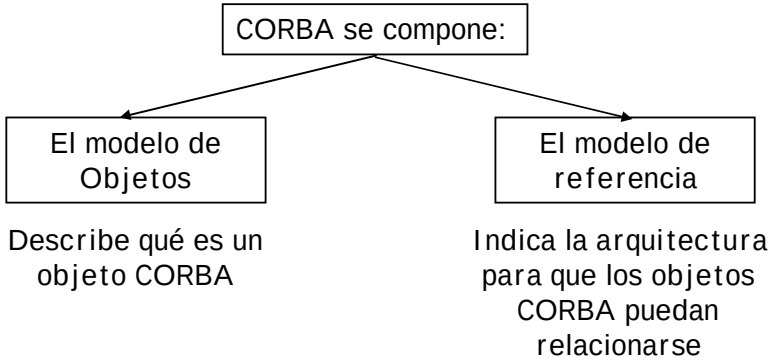
Desarrollado por

el Object Management Group (OMG)

para conseguir software:

- Orientado a objetos
- Portable
- Reusable
- Distribuido
- Que funcione en entornos heterogéneos

- Implementaciones gratis y de pago: Orbacus, ORBIX, Visibroker



¿Qué es CORBA?

En 1989 se forma el Object Management Group (OMG), con la finalidad de buscar soluciones a los problemas que surgen al desarrollar nuevos sistemas o de integrar aplicaciones centralizadas ya existentes en entornos distribuidos.

En 1991, el OMG recoge su propuesta en un modelo inicial denominado Object Management Architecture (OMA), en el que destaca el corazón de dicha arquitectura: la especificación de CORBA (Common Object Request Broker Arquitecture). Actualmente dicho grupo acaba de concluir la especificación de CORBA 3.0.

A partir de la especificación de CORBA, varios fabricantes han construido ya productos comerciales: ORBIX de IONA o el Visibroker de Borland. También, se han desarrollado numerosos productos de libre distribución: Oxbit, Mico, Tao, etc.

No todos los productos implementan todos los servicios del modelo y no todos son iguales en rendimiento, ya que CORBA dice “qué hacer” pero no “cómo hacerlo”.

Corba es un modelo para facilitar el desarrollo y la interoperabilidad de sistemas distribuidos orientados a objetos.

Vamos a describir dicho modelo en los apartados siguientes.

El modelo de gestión de objetos CORBA

Como ya se ha comentado, el objetivo del OMG, es promover el desarrollo de software distribuido, orientado a objetos, reutilizable, portable y capaz de interactuar con entornos heterogéneos, mediante el uso del modelo de *gestión de objetos distribuidos* CORBA.

CORBA es un modelo que describe un soporte de programación distribuida orientada a objetos. Es decir, este modelo **no cuenta cómo** hacer un soporte, **sino qué** debe hacer.

El modelo de *gestión de objetos distribuidos* CORBA se compone de:

- **El modelo de objetos** en donde se describe qué es un objeto CORBA
- **El modelo de referencia** en donde se explica qué arquitectura se propone para permitir que los objetos CORBA puedan relacionarse.

El Modelo de Objetos

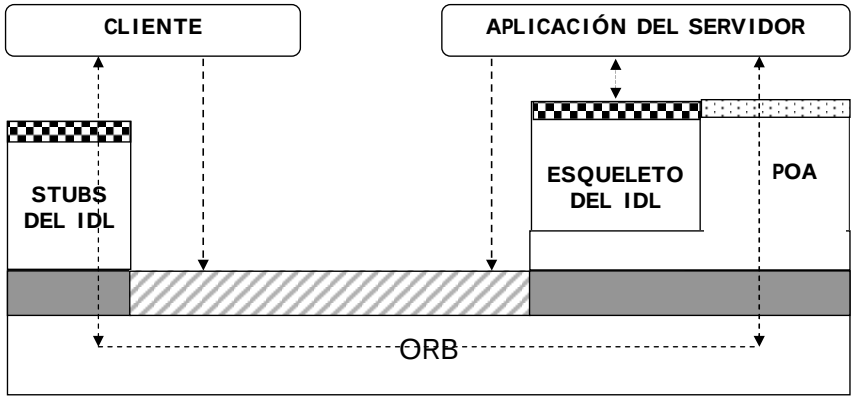
Un objeto CORBA:

- Es una entidad que ofrece un servicio
- Puede ser propio del sistema o creado por el programador
- Se define mediante el IDL
- Se identifica de manera única mediante su referencia

La comunicación con un objeto CORBA puede ser:

- Síncrona (exactamente una/como mucho una)
- Asíncrona (como mucho una)
- Sin respuesta o one-way (best-effort)

Cliente y servidor se relacionan siempre a través de interfaces:



Interfaz dependiente de la implementación del ORB
Interfaz idéntica en el lado del cliente y servidor
Interfaz idéntica en todas las implementaciones de ORB

CORBA es un modelo cliente/servidor orientado a objetos. Así, un servidor ofrece un servicio a través de un objeto CORBA.

Un objeto CORBA en un entidad software que recoge un servicio compuesto por una o más operaciones, visibles al exterior, que permiten modificar el estado de dicho objeto.

El lenguaje definición de interfaces, Interface Definition Language (IDL), permite la descripción de la interfaz que ofrece un objeto.

Una interfaz recoge el nombre del servicio y la firmas de cada procedimiento indicando el nombre de la operación, el conjunto de parámetros de entrada (in), de salida (out) o de entrada/salida (inout), con sus respectivos tipos y el tipo del resultado de la operación. También se pueden asociar excepciones de usuario o del sistema a cada una de las operaciones.

Un objeto CORBA se identifica de manera única en todo el sistema mediante su *referencia de objeto*. Una referencia de objeto es un identificador de comunicación que el cliente usa, pero no interpreta, para poder comunicarse con el servidor.

Por defecto la comunicación entre cliente y servidor es síncrona o bloqueante: Una vez que el cliente realiza una petición se bloquea en espera de la respuesta.

La semántica de entrega en la comunicación síncrona es **exactamente una** en ausencia de errores. En el caso de que se produzca un error se asegura la semántica **como mucho una.**

También se permite comunicación no bloqueante de dos tipos: asíncrona y en un solo sentido. Se da comunicación asíncrona si el servidor envía respuesta, o en un solo sentido, si el servidor procesa la petición pero no genera mensaje de respuesta.

En las operaciones **asíncronas** se asegura la semántica **como mucho una** y en las operaciones en las que el cliente no espera respuesta del servidor, es decir en un solo sentido, la semántica es *best-effort*.

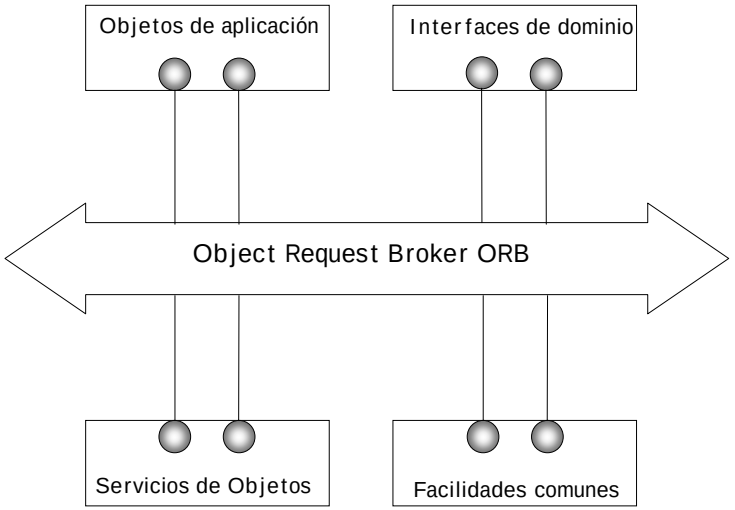
En las operaciones **asíncronas** se asegura la semántica **como mucho una** y en las operaciones en las que el cliente no espera respuesta del servidor, es decir en un solo sentido, la semántica es *best-effort*.

Resumiendo, en Corba cada servicio viene descrito por su interfaz, tanto si es un servicio que ofrece el propio sistema como si es un servicio escrito por programador. Así, la petición de un cliente obliga a que la interfaz del servicio sea idéntica tanto para el cliente cómo para el servidor. También interviene la interfaz del ORB para la transmisión de la petición y la respuesta entre cliente y servidor.

El ORB presenta dos tipos de interfaces: la interfaz pública que figura en el modelo de la OMG, y la privada que es la que generan los distintos fabricantes para hacer que se conecten los stubs con el ORB.

El Modelo de Referencia

La arquitectura del modelo Corba:



El ORB conecta objetos pertenecientes a:

- Servicios de objetos: facilidades de bajo nivel
  - Objetos de aplicación: aplicaciones del usuario
- Facilidades comunes: Facilidades de alto nivel
  - Interfaces de dominio: interfaces especificas

CORBA PERMITE CONSTRUIR SISTEMAS ABIERTOS HW Y SW

Si el modelo de objetos se centra en cómo describir el comportamiento o semántica que un objeto CORBA presenta a los clientes, el modelo de referencia o arquitectura de la gestión de objetos, muestra agrupaciones de objetos según su funcionalidad.

El ORB ofrece una interfaz pública que ofrece operaciones para envío y recepción con diferente calidad de servicio.

Habitualmente, un cliente o un servidor no trabajan directamente con el ORB sino con los servicios de objetos. Este grupo se considera parte del núcleo de CORBA, ya que ofrece servicios básicos o de bajo nivel al resto del sistema: por ejemplo, cómo localizar un objeto (servicio de nombres o de páginas amarillas (trading)), cómo mantener el orden (servicios de persistencia, transacciones y seguridad) o cómo generar eventos (servicios de eventos y de notificación).

Si los servicios de objetos son fundamentales en cualquier aplicación, hay otros que sólo son fundamentales en ciertos entornos. Por ejemplo, en un entorno médico es necesario un servicio de identificación personal, pero en entorno financiero, aparecerán otros servicios básicos que no tienen por qué incluir el de identificación. A este conjunto de objetos especializados se le denomina objetos de dominio y cubren áreas como telecomunicaciones, finanzas o comercio electrónico, entre muchas otras.

CORBA ofrece también, un conjunto de objetos que se denominan facilidades comunes. Una facilidad común es un servicio de alto nivel que es útil en cualquier entorno, como por ejemplo un servicio de correo electrónico o un servicio de impresión.

Todos los objetos antes descritos presentan una interfaz pública, diseñada por la OMG, para permitir la interoperabilidad entre diferentes sistemas.

Por último, las aplicaciones del usuario forman el conjunto de objetos de aplicación y cuyo funcionamiento se basa en el uso del resto de los componentes del modelo CORBA. Estos objetos son los únicos que no están estandarizados por la OMG, pero si ciertos objetos aparecen con frecuencia en diferentes aplicaciones pueden convertirse en candidatos para que la OMG les incluya en alguna de las categorías anteriores.

Finalmente, de está arquitectura podemos sacar la siguientes conclusiones:

Permite que los programadores no tengan que construir ciertas herramientas.

Si el mismo conjunto de servicios y facilidades están disponibles en cualquier entorno CORBA, entonces se pueden mover aplicaciones de un entorno a otro.

Por último, el cliente conoce cómo invocar operaciones de interfaces estándares sobre objetos en cualquier plataforma. Es decir, permite construir sistemas abiertos.

Especificación

Un Ejemplo: Servicio matemático.

La especificación del servicio en RPC de Sun:

```
/* matematico.x */

enum divstat{
  DIV_OK = 0,     /* NO ERROR */
  DIV_NOK = 1     /* ERROR */
};

/* Resultado de la división */
union resultado switch (divstat status) {
case DIV_OK:
        int cociente;
default:
        void;
};

struct enteros {
  int dividendo;
  int divisor;
};

program MAT_PROGRAM {
  version MAT_VERSION {
    resultado DIVIDIR(enteros)=1;
  }=1;
}= 0x03000000;
```

Los errores del servidor se recogen en variables

Se admite sólo:

- Un parámetro de entrada
- Un parámetro de salida

Se especifica:

- Nº de programa
- Nº de versión

La especificación en CORBA con idl:

```
// Matematico.idl

interface matematico {

  double dividir (in double  dividendo, in double divisor)

    raises(DivisionPorCero);

};
```

Se admite más de un parámetro de entrada y/o salida

Los errores del servidor se asocian a excepciones

Para ayudarnos a entender mejor lo que es CORBA, vamos a desarrollar un ejemplo sencillo en el entorno CORBA e irlo comparando con el clásico modelo de las RPC de Sun. Vamos a construir servicio matemático con una única operación que divide dos números enteros un.

Así, en la comparación con RPC comenzaremos a ver la especificación del servicio, construiremos un cliente que invoque un procedimiento remoto y, por último, construiremos un servidor que ofrezca el servicio.

En RPC de Sun, la especificación del servicio matemático se realiza con el lenguaje de especificación XDRL, y se recoge en el fichero `matematico.x`.

En el entorno CORBA, la especificación del servicio también se realiza en un lenguaje de especificación de interfaces, conocido como IDL. El fichero `Matematico.idl` muestra el servicio.

En el caso de las RPC, nos encontramos con que el diseñador debe asociar un número de programa y de versión al servicio matemático, así como numerar en orden ascendente las operaciones que ofrece la interfaz. (¿Qué ocurriría si dos interfaces distintos tuvieran el mismo número de programa y de versión?).

En CORBA no se indica un número de programa y de versión, sino simplemente el nombre del servicio.

En RPC no se indica explícitamente si los parámetros son de entrada y/o salida. En CORBA, sin embargo, se puede indicar si son de entrada (`in`), de salida (`out`) o de entrada/salida (`inout`).

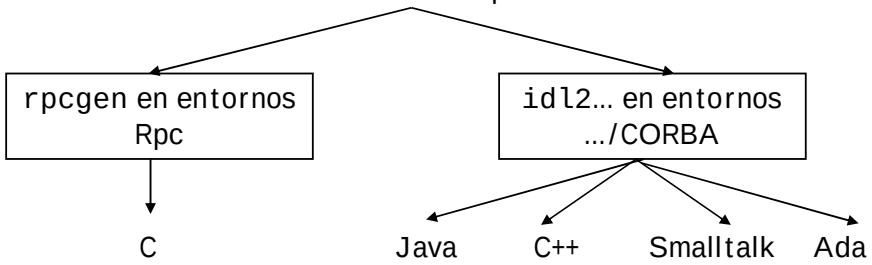
Al ejecutar la operación de división, el servidor puede devolver un resultado válido, o bien, un error de ejecución cuando se intenta dividir por cero.

Para poder recoger estas situaciones, en las RPC se construye un registro variante cuya variable `status` indicará con el valor `DIV_OK` si todo ha ido bien y, por tanto, `cociente` recoge el resultado, o bien, `DIV_NOK` para indicar que se ha producido un error de ejecución. Por tanto, el cliente deberá comprobar explícitamente este valor antes de recoger el cociente.

Para controlar situaciones de error en CORBA, se pueden declarar excepciones, como la que figura en el ejemplo: `DivisionPorCero` a la que se le podría asociar una determinada acción, por ejemplo, escribir un mensaje de error. También, se pueden producir excepciones predefinidas por CORBA como aquellas asociadas a errores de comunicación entre el cliente y servidor.

...Especificación

Los `interfaces` se someten al compilador de `interfaz`:



- En RPC el compilador `rpcgen` genera código C
- En CORBA hay múltiples compiladores `idl2c`, `idl2java`, ...
- En ambos entornos se generan *stubs* del cliente y servidor

Una vez definido el servicio, hay que someterlo al compilador de interfaces para generar los *stubs* del cliente así como el código de *dispacthing* del servidor.

En RPC, con el compilador `rpcgen matemático.x` se generan los ficheros de stubs en lenguaje C.

En CORBA existe también el concepto de compilación de interfaces, pero los lenguajes de programación a los que se traduce pueden ser múltiples: Java, Ada, C++, Smalltalk, etc.

Con el compilador `idl2java matematico.idl` se generan, al igual que RPC, los *stubs* del cliente y el código de *dispacthing* para el servidor en Java.

No vamos a enumerar la serie de ficheros que se crean porque depende del lenguaje de programación, por lo que solo iremos destacando aquellos más representativos.

El Cliente

Un Ejemplo en RPC y CORBA

El Cliente

Programación del cliente

1. Inicializa el soporte de transmisión
2. Localiza el servicio de nombres
3. Localiza el servicio matemático
4. Invoca el procedimiento de división
5. Analiza el resultado

Sistemas Distribuidos La Comunicación - 58

Programación de un cliente

Habitualmente, un cliente realiza las siguientes acciones:

1. Inicialización del soporte de transmisión.
2. Localización del servicio de nombres.
3. Localización del servicio matemático.
4. Invocación del método del servicio.
5. Utilización del resultado.

Además, dentro de estas acciones, hay que controlar que no se produzcan errores, ya sean de comunicación o de ejecución dentro del servidor.

Veamos a continuación cómo se realizan estas operaciones en los clientes de RPC y de CORBA respectivamente.

...El Cliente RPC

```
/* Cliente.c */
#include ...

main (int argc, char *argv[]){
  CLIENT                  * servidor;
  resultado      * resul;
  struct operandos {int dividendo; int divisor;};
  struct operandos oper;

  /* 1. Se inicializa el soporte de transmisión */   1
  /* 3. Se obtiene el id. Del servicio matemático */  3
  servidor =
        clnt_create("quijote",MAT_PROGRAM,MAT_VERSION,"udp");
  if (servidor == NULL) {{/* error de comunicacion */
    clnt_pcreateerror("quijote");
    exit(1);
  }
  /* Se obtienen los operandos */
  oper.dividendo = atoi(argv[1]);
  oper.divisor = atoi(argv[2]);

  /* 4. Se invoca la operación de división */   4
  resul = dividir_1(&oper, servidor);
  if (resul==NULL){/* error de comunicacion */
    printf("fallo en comunicación \n");
    exit(-1);
  }

  if (resul->status == DIV_NOK){/* Error de ejecución */
    printf("Intento de división n por cero \n");
    exit(-1);
  }

 /* 5. Se imprime el resultado */   5
  printf("%i/%i = %i \n ",oper.dividendo, oper.divisor,
         resul->resultado_u.cociente);
  exit(0);
}
```


En la descripción de cada uno de pasos de la programación del cliente vamos a ir viendo las diferencias entre las RPC y CORBA. En la figura de esta transparencia se muestra el código completo del cliente RPC, en la transparencia siguiente se muestra el de CORBA.

1. Inicialización del soporte de transmisión

Lo primero que se necesita para que cliente y servidor se comuniquen es un soporte de transmisión que permita el intercambio de mensajes.

En el caso de CORBA, el *Object Request Broker* (ORB) se encarga de realizar dicho intercambio, proporcionando un soporte de transmisión bidireccional y fiable que asegura, por defecto, que la semántica de entrega de mensajes es del tipo *como mucho una*. Además, se encarga de todas las tareas de *marshalling* y *unmarshalling*.

En el paso 1 “Inicialización del ORB” del cliente CORBA se inicializa este soporte virtual de transmisión abstrayéndose de los detalles de implementación de dicho soporte.

Con las RPC no existe esta transparencia, puesto que en este primer paso el cliente establece el protocolo de comunicación UDP. Además, la semántica de entrega de mensajes por defecto es al menos una.

2. Localización de servicio de nombres

En cuanto a la localización de un determinado servicio, la filosofía que se ha seguido sobre el servicio de nombres es completamente distinta en los dos entornos.

En RPC, el espacio de nombres no es global, sino local a cada una de las máquinas que constituyen el sistema, por esta razón es necesario que el cliente sepa en qué máquina reside el servicio (falta de transparencia de ubicación). Esto tiene el inconveniente de que puede haber en, distintas máquinas, dos servicios con el mismo nombre, que no tienen porqué dar el mismo servicio.

Además, el servicio de nombres de cada máquina, el *portmapper,* se ubica de manera estática, ya que se ejecuta en un puerto fijo (el 111) de cada máquina. Por esta razón la operación 2 (localización del servicio de nombres) no es necesaria en el modelo de RPC, pues no existe transparencia de ubicación de servicio.

Sin embargo, en CORBA aparece un espacio de nombres único que es soportado por un servicio de nombres que presenta reubicación dinámica. Así, en tiempo de ejecución (en la operación 2), el cliente pregunta al ORB dónde está el servicio de nombres, de tal manera que si el servicio de nombres cambiara de ubicación durante la ejecución del cliente, éste podría seguir funcionando si vuelve a preguntar de nuevo dónde reside este servicio.

El Cliente CORBA

```
// Client.java
import org.omg.CosNaming.*;

public class Client {
public static void main(String[] args) {
  try{
    // 1.Inicialización del ORB                                    (1)
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

    // 2.Se Obtiene el identificador del servicio de nombres       (2)
    org.omg.CORBA.Object
          rootObj = orb.resolve_initial_references("NameService");
    NamingContextExt
          root = NamingContextExtHelper.narrow(rootObj);

    // 3.Se Obtiene el identificador del servicio matemático       (3)
    org.omg.CORBA.Object
          mgrObj = root.resolve(root.to_name("MAT_PROGRAM"));
    Matematico  servidor = MatematicoHelper.narrow(mgrObj);

    // Se obtienen los operandos
    int dividendo = Integer.parseInt(args[0]);
    int divisor = Integer.parseInt(args[1]);

    // 4. Se invoca la operación de división                       (4)
    int cociente = servidor.dividir(dividendo,divisor);

    // 5. Se imprime el resultado                                  (5)
    System.out.println(dividendo + "/“ +divisor+" = "+ cociente);
  }

   // Se controlan las excepciones del servicio
   catch (MatematicoPackage.DivisionPorCero e) {
     e.printStackTrace();
   }

   // se controlan los errores de comunicación y del sistema
   catch (Exception e) {
     e.printStackTrace();
   }
}
```


3. Obtención del servicio matemático

En RPC, los pasos 1 y 3 (Obtención del identificador del servicio matemático), recogen dos operaciones:

- Creación de un *socket* de tipo UDP, por donde el cliente podrá enviar y recibir información del servidor.
- Obtención del identificador de comunicación del servicio matemático preguntándoselo al *portmapper* de la máquina “quijote” . Este identificador solo es válido mientras viva el cliente, de tal manera que si el cliente falla, al rearrancarlo, ese identificador ya no será válido, aunque el servidor siga vivo.

En CORBA, el identificador que se obtiene al invocar la operación `resolve` del servicio `MAT_PROGRAM` sobre el servicio de nombres (paso 3), es válido incluso aunque el cliente falle o finalice correctamente. Si el cliente guarda ese identificador puede usarlo en sucesivas ejecuciones.

4. Invocación del método del servicio

Con el identificador de servicio ya se está en disposición de llamar a la operación de división (paso 4 del ejemplo).

En RPC, esta operación `dividir_1` obliga, tanto al empaquetamiento de los parámetros de entrada en un registro, como a la inclusión del identificador de comunicación del servidor como parámetro. Es decir, no presenta transparencia de acceso, ya que una llamada local a un procedimiento no presenta estas restricciones de llamada.

En CORBA, la llamada a un procedimiento viene determinada por el lenguaje que se use; en el caso de Java, al estar orientado a objetos, la invocación sigue el modelo `objeto.metodo (parametros)` tanto si es una llamada local como si es una llamada remota.

Por tanto, vemos que `servidor.dividir (dividendo, divisor)` presenta transparencia de acceso en la llamada, puesto que ya sea el objeto remoto o local se invoca de la misma forma. En el lado del cliente, la llamada a este método provoca la ejecución del *stub* correspondiente, que se encarga de pasar al ORB la petición de envío al servidor. Al ser una llamada síncrona el *stub* espera la respuesta que le pasa el ORB y la devuelve al programa principal.

5. Resultado.

Por último, en este paso se muestra por la salida estándar el resultado de la división, si todo hay ido bien.

...Programa del Cliente

Gestión de errores de transmisión y de ejecución en el servidor
- En RPC se tratan explícitamente
- En CORBA se usan excepciones

En RPC se pregunta EN CADA LLAMADA remota si:

- ¿NULL? ¿Hay error de transmisión?
- ¿Status != NO_OK? ¿Hay error de ejecución?

En CORBA se establece UNA SOLA VEZ el control con:

- `Catch (Exception e)` ¿Hay error de transmisión? (Señalada por el ORB)
- `Catch (MatematicoPackage.DivisionPorCero e)` ¿Hay error de ejecución? (Excepción señalada por el servidor)

Pero ¿y si ha habido problemas?

Los problemas que pueden producirse son errores de ejecución en el servidor (por ejemplo división por cero) o errores en la comunicación. Vamos a ver como se tratan:

En RPC los errores deben tratarse explícitamente ya que el lenguaje C no soporta excepciones. En el código del ejemplo, las cajas sombreadas recogen el control de errores de transmisión para cada llamada remota.

La caja de línea discontinua encierra el código que controla el error de ejecución especificado en el servicio: la división por cero, preguntado por el status devuelto por el servicio.

En CORBA, los errores de ejecución se pueden controlar asociándolos a una excepción, supuesto que el lenguaje de programación soporte excepciones. Cuando se intenta dividir por cero, el servidor produce la excepción `DivisiónPorCero`, que el cliente trata en el código de la caja de línea discontinua, imprimiendo el tipo de error que se ha producido.

Los errores de transmisión también se asocian a excepciones, pero no es el servidor quién las produce sino el soporte de transmisión de CORBA, es decir el ORB. En el cliente, el tratamiento de estos errores se recoge en la caja sombreada, y solo hay que hacerlo una vez, independientemente del número de llamadas remotas que se hagan.

¿Qué es un objeto CORBA?

Un cliente en Java solicita operaciones sobre un objeto CORBA.

¿Cualquier objeto Java es un objeto CORBA?

¡¡ NO !!

Un **objeto CORBA** es un **OBJETO VIRTUAL**	Un objeto de un lenguaje de programación es real

¿Qué relación hay entre un objeto CORBA y un objeto real?

La idea es similar a la de memoria virtual:

En el modelo de memoria virtual:	En el modelo CORBA:
El proceso genera una dirección virtual	El cliente referencia un objeto CORBA
Gestor de memoria + MMU	ORB + POA
Acceso a una dirección física o real	Acceso a un objeto real (C++, Ada, C, Cobol, ...)

El servidor solo ofrece objetos CORBA

¡¡ TRANSPARENCIA DE ACCESO !!

En el ejemplo anterior hemos visto que en el entorno CORBA un cliente escrito en lenguaje Java invoca operaciones sobre un objeto CORBA.

Entonces ¿se puede decir que todo objeto Java es un objeto CORBA ?

La respuesta en que no y su justificación recoge la esencia del modelo CORBA:

La relación que existe entre un objeto CORBA y un objeto en cualquier lenguaje de programación es similar a la que existe entre una dirección virtual y una dirección física en un sistema de memoria virtual de un sistema operativo. En un ordenador convencional, un proceso genera direcciones virtuales, direcciones que no existen realmente en la memoria principal, pero que con la ayuda de software de gestión de memoria y de la MMU se traducen de manera transparente a direcciones reales de memoria principal.

En el modelo CORBA, **un objeto CORBA es un objeto virtual** es decir no tiene existencia por sí mismo si no hay detrás un objeto en un determinado lenguaje de programación que le soporte.

Es decir, los objetos CORBA que referencia un cliente son tan virtuales como lo son las direcciones que genera un proceso. Cuando se referencia un objeto CORBA habrá un conjunto de componentes del sistema que traducirán esta referencia a un objeto real implementado en un determinado lenguaje de programación, al igual que el sistema de gestión de memoria traduce las direcciones virtuales a direcciones reales o físicas.

En CORBA los elementos encargados de esta traducción son el bus virtual (ORB) y un elemento denominado **adaptador de objetos** o POA (*Portable Object Adapter*), del que hablaremos más adelante.

¿Qué ventaja se obtiene al hacer que un objeto CORBA sea virtual?

Esto permite que cliente y servidor puedan escribirse en diferentes lenguajes de programación: El cliente escrito en Java puede invocar operaciones sobre un objeto CORBA soportado por un objeto en C++ o Ada, o incluso por un conjunto de rutinas en C o Cobol. Es decir, le abstrae al cliente de la implementación del objeto, presentado transparencia de acceso, puesto que independientemente de si el objeto real está soportado en un lenguaje orientado a objetos o no, el acceso siempre es del tipo `objeto.metodo(parámetros).`

Por tanto, los servidores CORBA ofrecen siempre servicios representados por objetos CORBA.

De esto, también se desprende que un objeto CORBA y los objetos que les soportan (en algún lenguaje de programación concreto) tienen vidas independientes, es decir, tienen operaciones de creación y destrucción propias y pueden crearse y destruirse de manera independiente.

En el apartado siguiente veremos cómo se construye un servidor en RPC y CORBA y veremos que aquí es donde se presentan las mayores diferencias conceptuales.

El Servidor RPC

El Servidor

1. Inicializa el soporte de transmisión
2. Localiza el servidor de nombres
3. Da de alta del servidor matemático
4. Espera de peticiones
5. Invoca el procedimiento de división
6. Envía la respuesta

El Servidor RPC

```
/* matematico_svc.c */
main()
{...
  /* 1. Se crea un socket udp */ (1)
  transp = svcudp_create(RPC_ANYSOCK);
  /* 3. Se da de alta el servicio de nombres */ (3)
  svc_register(transp, MAT_PROGRAM, MAT_VERSION,
               mat_program_1, IPPROTO_UDP))
 /* 4. Se cede el control al dispatcher mat_program_1 */ (4)
  svc_run();
}
```

```
/* dispatcher mat_program_1 */
static void mat_program_1(rqstp, transp)
  ...
  /* Se analiza la petición */
  switch (rqstp->rqproc) {...
    case DIVIDIR:/* 5. Llamar al procedimiento de division */
      result = dividir_1(&argument, rqstp);  (5)
    break;
  }
  ...
  /* 6. Se envia la respuesta al cliente */ (6)
  svc_sendreply(transp, xdr_result, result);
  return;
}
```


Hay que recordar que, previamente a la construcción del servidor, hay que someter el fichero con la especificación del servicio al compilador de interfaces, para así obtener los ficheros con los perfiles de las operaciones y el *dispatcher.*

Un proceso servidor realiza habitualmente los siguientes pasos:

1. Inicializa el soporte de transmisión
2. Localiza el servidor de nombres
3. Da de alta el servicio en el servidor de nombres
4. Espera peticiones para ese servicio
5. Llama al procedimiento de división
6. Envía la respuesta al cliente

A continuación vamos a ver cómo se realiza cada uno de estos pasos en RPC y CORBA. En esta figura se muestra el servidor completo RPC y en la transparencia siguiente se ofrece el de CORBA.

1. Inicialización del soporte de transmisión

En el entorno RPC se crea un *socket* de tipo UDP para el envío y recepción, mientras que en el servidor CORBA simplemente se crea un bus virtual ORB.

2. Localización del servidor de nombres

En RPC, ya que no hay un servidor de nombres global, no hay que localizarlo, sino que directamente hay que dirigirse al *portmapper* de la máquina en la que reside el servicio deseado. El puerto por el que escucha el *portmapper* está prefijado y es el 111.

En CORBA, como el servidor de nombres es global y puede estar en cualquier máquina (y puede escuchar por cualquier puerto) hay que localizarlo, cosa que se hace solicitándoselo al ORB.

3. Alta del servicio en el servicio de nombres

En el entorno RPC se da de alta el servicio matemático “MAT_PROGRAM” en el *portmapper* o servicio de nombres local donde reside el proceso servidor.

En el entorno CORBA, el servidor da de alta, en el servidor de nombres global, el objeto CORBA `MAT_PROGRAM` que ofrecerá el servicio `Matematico`.

El código que realiza las tres operaciones anteriores en el entorno RPC (excepto el paso 2, que en RPC no tiene sentido) es generado automáticamente por el compilador de interfaz (`rpcgen`), recogiéndose en la parte `main` del esqueleto `matematico_svc.c`. Sin embargo, en CORBA estas tres operaciones se deben programar explícitamente.

El Servidor CORBA

Para ofrecer un servicio, el servidor

1. Crea un objeto real (servant)
2. Crea un objeto Corba
3. Los asocia
4. Da de alta el objeto CORBA

```
// Server.java
import org.omg.PortableServer.*; import org.omg.CosNaming.*;

public class Server {
   public static void main(String[] args) {
    try {
      // 1. Inicialización del ORB                                    (1)
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

      // Obtención del identificador del POA raiz
      POA rootPOA =
      POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
      // Activación del POA manager
      rootPOA.the_POAManager().activate();

      // Creación de una implementación del servicio (servant)
      MatematicoImpl managerServant = new MatematicoImpl();

      // Se crea el objeto CORBA y se asocia al servant
      org.omg.CORBA.Object CORBAObj=
                  rootPOA.servant_to_reference(managerServant);

      // 2. Localización del servicio de nombres                      (2)
      org.omg.CORBA.Object rootObj =
                  orb.resolve_initial_references("NameService");
      NamingContextExt root =
                  NamingContextExtHelper.narrow(rootObj);

      // 3. Alta del s. matematico en el servicio de nombres          (3)
      root.bind(root.to_name("MAT_PROGRAM"),CORBAObj);

      // 4. Espera de peticiones                                      (4)
      orb.run();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```


4. Espera de peticiones

La operación 4 cede control al *dispatcher* que se encargará de recoger la petición, llamar al procedimiento adecuado, recoger la respuesta y mandarla al cliente. Tanto en RPC como en CORBA, el *dispatcher* lo genera automáticamente el compilador de interfaces.

En el modelo RPC, el *dispatcher* es el procedimiento `mat_program_1` incluido en esqueleto `matematico_svc.c` y en CORBA el *dispatcher* también se incluye en el esqueleto que genera el compilador de interfaces (no vamos a comentar aquí dicho esqueleto para no complicar el ejemplo).

5. Invocación del procedimiento de división

Veamos cómo se invoca el procedimiento adecuado cuando llega una petición.

En los entornos RPC es simple: cuando le llega la petición de división al *dispatcher*, éste llama al procedimiento `dividir_1` que es el que realmente realiza la división.

En CORBA también es el *dispatcher* el encargado de llamar al procedimiento de división cuando se recibe una petición de tal operación. Sin embargo, para que esa petición llegue al *dispatcher* el servidor tiene que realizar una serie de acciones antes de ponerse a esperar a que lleguen peticiones. Veámoslas.

> Cuando hablábamos del cliente CORBA decíamos que en el exterior sólo veía objetos CORBA, es decir veía objetos virtuales.También comentábamos que tras un objeto virtual debe haber un objeto real en algún lenguaje concreto que lo soporte. Pero ¿quién y cómo se realiza esta asociación?
>
> El encargado de este trabajo es el servidor, que, en nuestro ejemplo en Java, debe crear el objeto que soporta el procedimiento que realmente divide. (En la jerga de CORBA, a este objeto se le denomina ***servant)***. Después creará un objeto CORBA y, por último, tendrá que asociar este objeto CORBA al objeto Java anterior. Después de esta asociación, el objeto CORBA está listo para darlo de alta en el servidor de nombres y hacerlo visible al exterior.

Todas estas tareas se enmarcan en un rectángulo sombreado en el código del servidor CORBA y no aparecen en el servidor de RPC porque son específicas de CORBA.

La asociación entre un objeto CORBA y el objeto real que lo implementa se guarda en un componente del sistema CORBA denominado **Adaptador Portable de Objetos** o POA (*Portable Object Adapter*).

Pero todavía nos queda contestar a la pregunta que teníamos pendiente al principio de este punto:¿Cómo invoca el dispatcher el procedimiento de división?

La función del *dispatcher* en CORBA es la misma que en RPC, analiza la petición y llama al procedimiento adecuado del objeto correspondiente.

Un Ejemplo en RPC y CORBA...Programa del Servidor

El procedimiento de servicio en RPC:

```
/* dividir.c */
#include ...#
resultado *dividir_1(enteros *p1, struct svc_req *p2){
  resultado *res;
  res = (resultado *)malloc(sizeof(struct resultado));
  if (p1->divisor == 0)
    res->status = DIV_NOK;
  else {
    res->status = DIV_OK;
    res->resultado_u.cociente=(p1->dividendo/p1->divisor);
  }
  return res;
}
```

La clase Java que implementa el servicio:

```
// matematicoImpl.java
 import org.omg.PortableServer.*;
 public class MatematicoImpl extends MatematicoPOA {
   public int dividir (int dividendo,int divisor)
        throws MatematicoPackage.DivisionPorCero {
     if (divisor == 0)
       {throw new MatematicoPackage.DivisionPorCero();}
     else
       {return (dividendo/divisor); }
  }
}
```

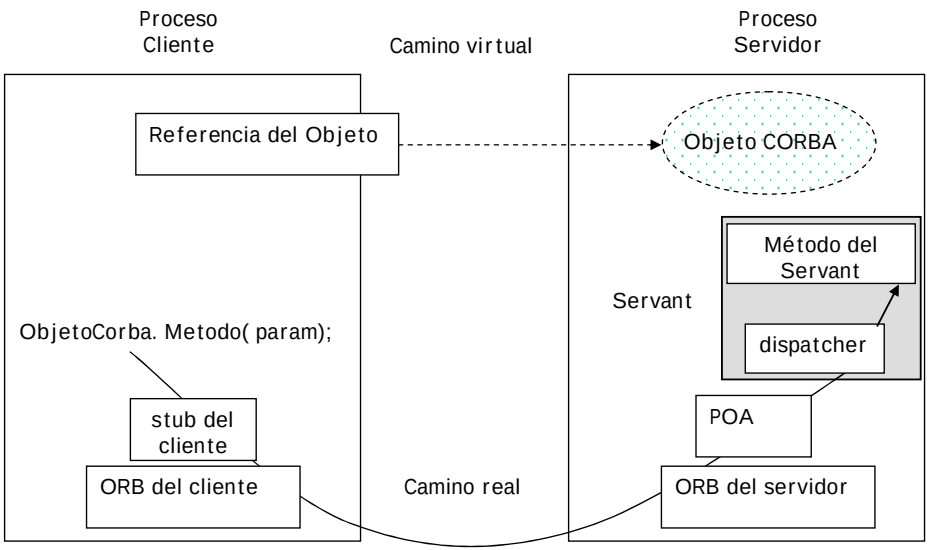

En esta transparencia se muestra el código, en lenguaje C, que implementa el procedimiento de división para el entorno RPC, así como la clase Java que contiene el procedimiento del servicio matemático que utilizaremos para el entorno de CORBA.

La clase `MatematicoPOA` que se hereda en la clase `MatematicoImpl` se genera automáticamente en la compilación de la interfaz del servicio `Matematico`.

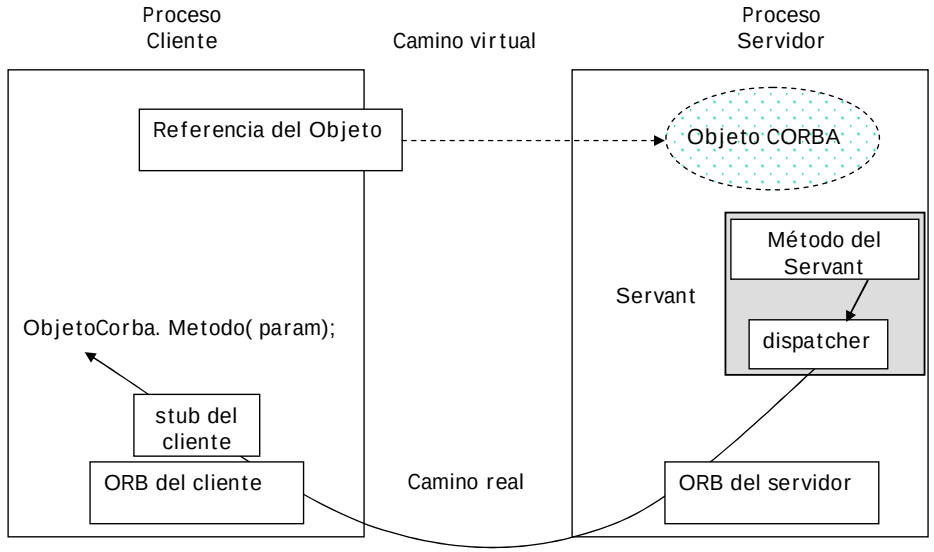
Ahora que tenemos todos los componentes que intervienen en la llamada a un objeto CORBA, veamos en la transparencia siguiente un esquema grafico de una llamada completa.

Un Ejemplo en RPC y CORBA...Programa del Servidor

Llamada a un método CORBA:



Envío de la respuesta:



Cuando le llega una petición al servidor, el *dispatcher* es el que se ocupa de llamar al procedimiento adecuado.

Cada objeto tiene su propio *dispatcher*, pues el reparto dentro del *dispacher* tendrá diferentes opciones en nombre y número, dependiendo de los nombres y número de métodos que tenga un objeto.

Pero **¿quién entrega las peticiones al *dispatcher*?**

En el entorno RPC, las peticiones se las entrega directamente el sistema operativo a través de un *socket*.

En CORBA, por cada servicio hay un POA, y de cada servicio puede haber varios objetos creados. Pues bien, cuando llega una petición para uno de los objetos de un servicio, es el POA de ese servicio el que sabe a cual de los objetos debe dirigirlo.

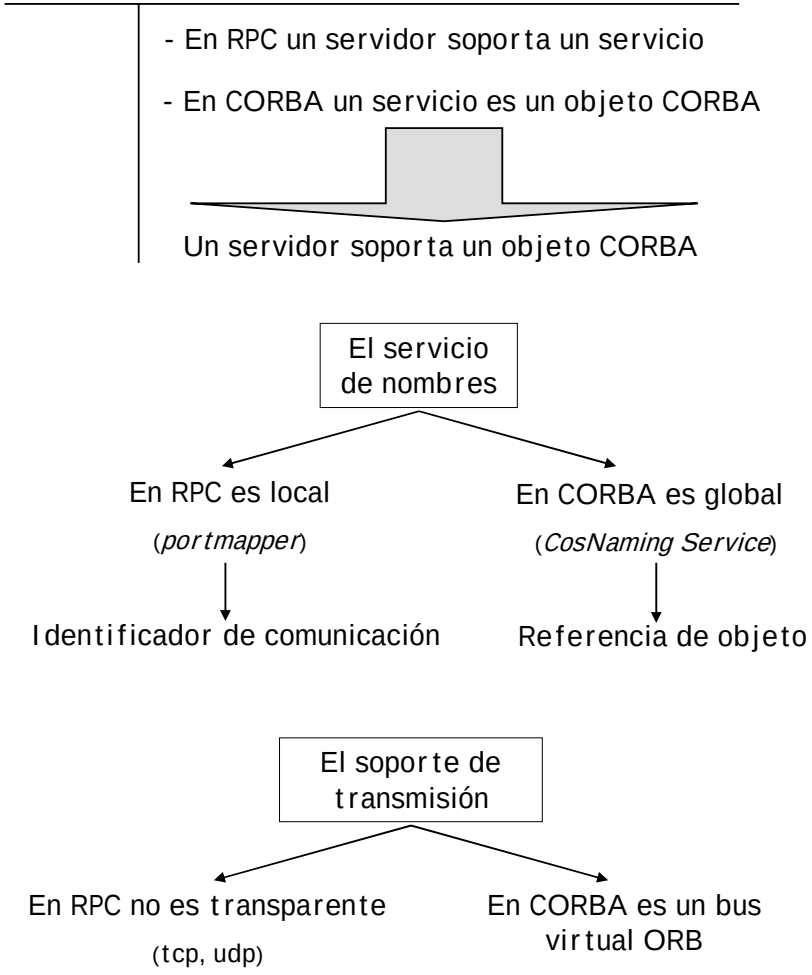
Y **¿quién le entrega el mensaje al POA?**

El bus ORB. Este bus entrega al POA una petición sobre un objeto CORBA con los datos en el formato local de esa máquina. A su vez, el bus ORB ha obtenido el mensaje del modulo de comunicación que haya por debajo, por ejemplo del sistema operativo local.

6. Envío de la respuesta

Una vez que el procedimiento adecuado (o rutina de servicio) ha terminado, le devuelve al *dispatcher* el resultado. A continuación, en el caso de RPC, el *dispatcher* serializa el resultado, lo convierte a un formato externo de representación y, por último, lo envía a través de un *socket*. En el caso de CORBA, el resultado en formato local se le pasa al ORB, el cual se encarga de serializarlo y enviarlo al cliente.

Resumiendo



- La llamada a una operación en RPC es "procedural"
- En CORBA se sigue el modelo `objeto.metodo(parámetros)` presentando transparencia de acceso y de ubicación

Resumiendo, en CORBA un cliente se comunica con un servidor que soporta un objeto CORBA que ofrece un determinado servicio.

Para poder comunicarse con el servidor, el cliente necesita un identificador de comunicación o referencia del objeto CORBA que soporta el servicio. Este identificador lo obtiene del servicio de nombres de CORBA, que es global, por lo que no puede haber dos objetos CORBA registrados con el mismo identificador.

El soporte de transmisión empleado es el bus virtual ORB, es decir, no es un componente hardware para la transmisión de mensajes, es un bus virtual que para conseguir la transmisión física puede apoyarse en las llamadas al sistema que ofrezca el módulo de comunicaciones del sistema operativo donde se ejecute o, incluso, el ORB puede apoyarse en la capa RPC que tenga esa máquina. Este bus virtual, por defecto, presenta la semántica *al menos una* y esconde todos los detalles de comunicación.

Se debe tener cuidado en la implementación del ORB, ya que si se realiza apoyándose en las RPC, la velocidad de transmisión se vería muy afectada, debido al número de capas de software que habría que atravesar hasta llegar al medio físico de transmisión y viceversa.

Por último, diremos que la forma de llamar a los métodos presenta transparencia de acceso y de ubicación ya que independientemente de que el método pertenezca a un objeto local o remoto se invoca de la misma manera:

```
Objeto.método(parámetros)
```