CAPITULO 22

REMOTE PROCEDURE CALL

22.1. - Introducción

El modelo cliente-servidor sufre de una debilidad. Todo su paradigma de comunicaciones está basado en sentencias de input/output (send/receive)

En RPC (Birrel y Nelson 1984) los procesos pueden hacer llamados a procedimientos que no residen en la máquina donde están corriendo.

La llamada a una subrutina que se encuentra en otra máquina, no se ejecuta concurrentemente, ya que el proceso llamador queda bloqueado o suspendido hasta que termine la subrutina. La información se puede transportar de un lado a otro mediante los parámetros y puede regresar en el resultado del procedimiento. Así todo pasaje de mensajes e instrucciones de I/O son invisibles al programador.

Como sabemos en la llamada a un procedimiento local, se pueden utilizar pasajes de parámetros por valor o por referencia.

Cuando la llamada es remota tiene que verse lo más local posible. Para lograr esta transparencia cuando se compila un programa, en vez de adicionar código de una librería, se une un tipo de código especial llamado client-stub (resguardo del cliente) que se va a encargar de empaquetar los parámetros, hacer un send al server y hacer un receive bloqueándose en espera de la respuesta.

Cuando el mensaje llega al server, el kernel se lo pasa al server-stub (resguardo del servidor). Para que esto ocurra, el código del server-stub tuvo que ejecutar un receive y quedarse en espera de un requerimiento. Luego desempaqueta los parámetros y llama a la rutina del server que va a procesar el requerimiento (coloca los parámetros en el Stack). Una vez resuelto le pasa los datos al stub que los empaqueta, hace un send con el mensaje empaquetado y luego hace un receive bloqueándose para recibir un nuevo mensaje.

Al recibir este mensaje el cliente, se desbloquea el client-stub, desempaqueta el mensaje y lo coloca a disposición del cliente.

La razón fundamental para la existencia de estas rutinas stubs radica en que el compilador generará en base a especificaciones formales del servidor sendas rutinas stubs, la del cliente para que empaquete los parámetros y construya el mensaje y la del servidor para que los desempaquete apropiadamente. Esto facilita la vida de los programadores, reduce la posibilidad de error y provee de transparencia al sistema respecto de las diferentes representaciones internas.

22.2. - Etapas de un RPC

1) El procedimiento cliente llama al stub cliente de manera transparente. Usando Stack.
2) El stub cliente arma el mensaje y se lo envía al kernel.
3) El kernel realiza el send del mensaje al kernel de la máquina remota.
4) El kernel remoto le da el mensaje al stub del server
5) El stub del server desempaqueta los parámetros y se los pasa al server. Usan Stack.
6) El server propiamente dicho realiza su trabajo y retorna un resultado al stub.
7) El stub del server empaqueta el valor retornado y se lo manda al kernel.

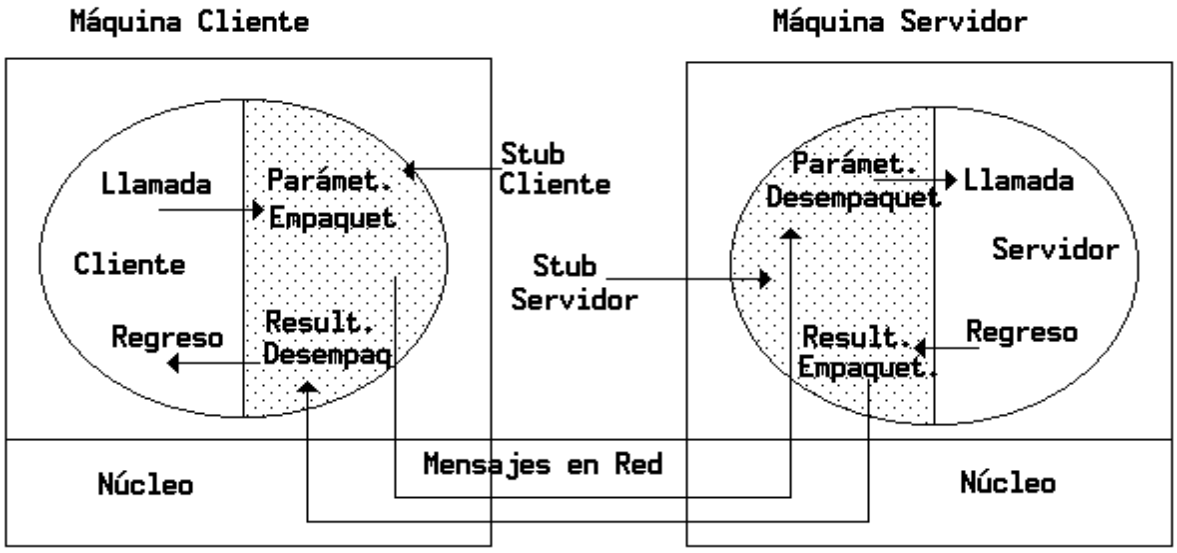

Fig. 22.1.

8) El kernel remoto envía el mensaje al kernel del cliente.
9) El kernel del cliente sube el mensaje al stub del cliente.
10) El stub cliente desempaqueta el resultado y se lo pasa al cliente.

El paso de los mensajes por los respectivos stub's es transparente tanto para el cliente como para el servidor.

Un mismo server puede proveer varias funciones, por lo tanto cada vez que llega un mensaje al stub server, debe chequearse cuál es el servicio que se requiere (que tiene que ser puesto como parte del mensaje por el stub cliente) y recién ahí realizar la llamada local.

22.3. - **Pasaje de Parámetros**

Recordemos que en forma estándar existen dos formas de pasar parámetros en una llamada a una subrutina. Uno de ellos es pasar parámetros por **valor** en cuyo caso el dato que se desea informar al procedimiento se copia directamente (usualmente se coloca en el stack) y la otra forma es por **referencia** en cuyo caso lo que se copia hacia el procedimiento es un puntero que indica la dirección en dónde se encuentra almacenada el dato en sí. Existe una tercera forma que se denomina **copia/restauración** en la cual el dato se copia como en el pasaje por valor pero el procedimiento invocado luego lo copia de regreso después de la llamada escribiendo sobre el valor original.

Veamos un ejemplo de cómo se haría este pasaje de parámetros. Supongamos que el cliente desea sumar dos números enteros (4 y 7). La llamada al procedimiento sum aparece en el cliente en donde el stub toma los dos parámetros y construye el mensaje que enviará al servidor indicando además que tipo de función solicita ya que el servidor podría proveer diferentes llamadas. Cuando el mensaje llega al servidor su stub analiza cuál es el procedimiento invocado y luego lleva a cabo la operación solicitada. Luego el stub del servidor toma el resultado de la operación lo empaqueta en un mensaje y éste se reenvía de regreso al cliente quien lo desempaqueta y se lo devuelve al proceso.

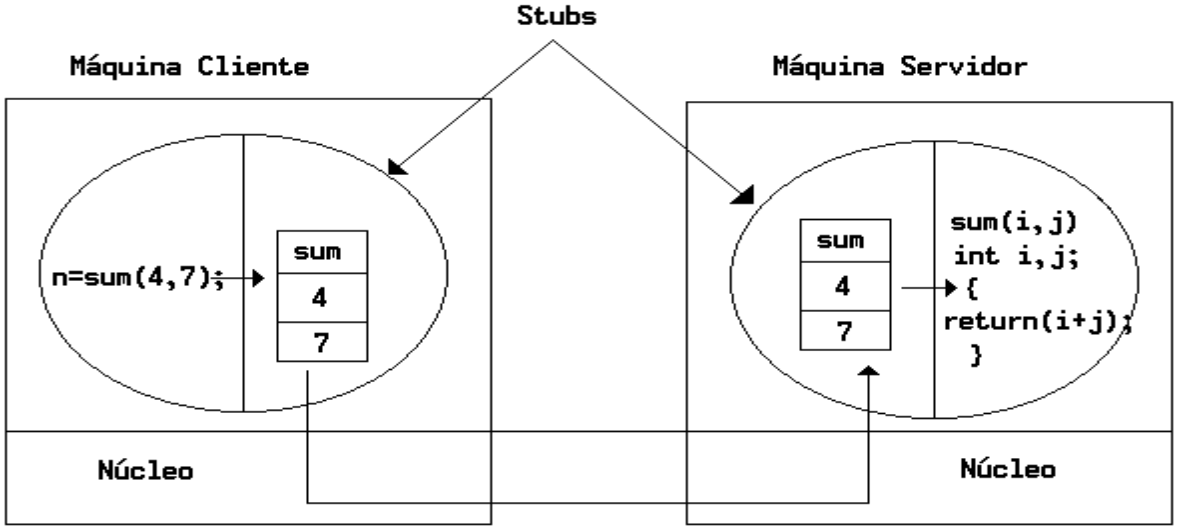

Fig. 22.2. - Cálculo remoto de sum(4,7).

Todo funciona bastante sencillamente en tanto el cliente y el servidor ejecuten en máquinas idénticas y los parámetros pasados sean de tipo escalar (enteros, caracteres, boolean). Sin embargo en un sistema distribuido de gran tamaño es común que se encuentren diferentes tipos de máquinas (Pcs, mainframes con diferente tipo de representación interna -ASCII, EBCDIC-); también se plantea un problema con los números de punto flotante y con la forma de numerar los bits en algunas máquinas (de derecha a izquierda *little endian* o de izquierda a derecha *big endian*).

Para solucionar este problema, se estableció una forma canónica para el pasaje de parámetros. Este método obliga a que el stub cliente o el servidor realicen conversiones de los mensajes recibidos o enviados siempre lo cual resulta ineficiente si ambos extremos de la comunicación proveen en forma nativa el mismo tipo de representación de datos.

Otra solución es indicar en el mensaje el tipo de máquina que lo está enviando, y no realizar ningún tipo de transformación previa. Si el receptor es el mismo tipo de máquina, toma los parámetros sin ningún inconveniente, de otra forma hace la conversión que sea necesaria.

Otro problema en el pasaje de parámetros es si se pasan punteros, ya que un puntero solo tiene sentido en el espacio de direccionamiento (address space) en donde se encuentra el proceso.
Una solución es prohibir el pasaje de punteros y parámetros por referencia.

Otra solución puede ser pasar los elementos del vector en su totalidad dentro del mensaje, así el stub del servidor podría llamar a la rutina apuntando a un buffer propio, donde guardó el vector que le pasaron como pará-

metro. Si el server escribe sobre esta parte de la memoria, el stub del servidor tiene que encargarse de devolverlo al cliente, así puede copiarlo modificado.

Luego los parámetros por referencia son reemplazados por un mecanismo de *copy/restore*.

Para optimizar más esta solución, bastaría con que el stub sepa si el buffer es un parámetro de Entrada (no necesita ser devuelto al cliente), Salida (no necesita ser enviado al servidor) o E/S.

Existe una forma de manejar el caso de apuntadores para estructuras de datos complejas. Aquí lo que se realiza es enviar el apuntador mediante la colocación en un registro y realizando un direccionamiento indirecto. La referencia inversa lo que realiza es enviar un mensaje del servidor al cliente para pedirle que le busque el dato deseado. Si bien este esquema es muy ineficiente hay ciertos sistemas que lo utilizan.

Pero aún existe un problema como ser en la función INC(i,i) [i=i+1], al stub del server le llegarían dos parámetros diferentes pero en realidad es el mismo, con lo cual si i' = 1 (como parámetro de Entrada) ... en el server ... i'' = 2 (como parámetro de Salida) [i'' = i' + 1], pero ambos hacen referencia al mismo lugar de memoria en el cliente, por lo tanto al restaurar los valores en la variable i del cliente puede quedar 1 o 2.

22.4. - **Generación de un Código RPC**

La generación de un código RPC se hace automáticamente. Se define un archivo donde se especifica el nombre y los tipos de parámetros de las funciones remotas al cliente, como el número de programa (o server). Esto se logra teniendo un compilador que lee las especificaciones y genera tanto el stub cliente como el stub del servidor. El código fue generado para ambos stubs a partir de la misma especificación con lo cual reduce las probabilidades de error haciendo transparente las dificultades del pasaje de términos.

22.5. - **Dynamic Binding (Conexión dinámica)**

Uno de los problemas existentes se basa en el direccionamiento al server, es decir la forma en que el cliente localiza al servidor.

Una forma es incluir la dirección en el código del cliente, pero esto no sería muy flexible a cambios. O sea que de existir cambios en la dirección del servidor se debería recompilar el código del cliente. Para solucionar esto se plantea el dynamic binding.

22.5.1. - **Modo de operar**

Lo primero que se debe hacer para poder implementar este tipo de binding, es que exista una especificación formal del servidor que incluya :

- el nombre del server,
- el número de versión, y
- una lista de procedimientos provistos por el server (tipo de parámetros que utiliza y si son de E, S o E/S).

El principal motivo de contar con esta especificación formal radica en que ésta es entrada de la rutina que generará los stubs del cliente y del servidor.

Cuando el Server comienza su ejecución, hace un call a una rutina propia de inicialización, fuera del loop principal, que se encarga de realizar un EXPORT de la interfase del server (manda un mensaje al programa BINDER (conector) indicándole que lo dé de alta y que conozca de su existencia). Para realizar la registración del server en el binder, se necesitan los siguientes datos: el nombre del server, su número de versión, un identificador único y un handle (asa - usado para localizarlo y que depende del sistema). También puede proporcionarse en esta registración información para la autenticación de mensajes (un servidor puede declarar que sus servicios sólo pueden brindarse a ciertos clientes).

Cuando el Cliente hace una llamada a un procedimiento remoto por primera vez, el stub cliente manda un mensaje al binder pidiendo que haga un import de la versión nn del server mm (Dynamic Binding puede ser gracias a un mecanismo rendez-vous y normalmente, un S.O. provee de un rendez-vous daemon sobre un port RPC fijo). El binder busca en su tabla de servers que hayan exportado una interfase que coincida con lo que el cliente está buscando. Si no existe entonces retorna un error, de lo contrario el binder retorna el handler y un identificador único (del proceso). El stub del cliente utiliza este handle como dirección a donde enviar el mensaje. El identificador único que se envía también en el mensaje, es usado por el kernel del server para direccionar automáticamente el mensaje al server correcto.

El número de versión es importante porque de esta forma el binder puede garantizar que los clientes que utilicen interfases obsoletas no puedan localizar al servidor impidiendo que realicen la llamada y obtengan resultados impredecibles.

22.5.2. - **Ventajas y Desventajas**

Este método de exportar e importar es muy flexible. Permite aplicar un método de autenticación a cada cliente que quiere hacer un bind con los servers.

Pero tiene el overhead extra que se genera para exportar e importar las interfases y además este binder en un sistema distribuido de gran tamaño puede convertirse en un cuello de botella.

22.6. - **SEMÁNTICA DE RPC EN PRESENCIA DE FALLAS**

La transparencia que proporciona RPC en cuanto a que las llamadas remotas parecen llamadas locales se ve afectada con el surgimiento de fallas, pues son difíciles de enmascarar. Existen 5 tipos de problemas que pueden ocurrir, a saber :

1) **El cliente no puede localizar al server.**
   Esto provoca que el programador deba que codificar qué hacer en caso de tener un error en el bind al server. (procedimiento de EXCEPCIÓN)
2) **Se pierde el mensaje de requerimiento del cliente al server.**
   Para estos casos el kernel pone un timer cuando se envía un mensaje y si no recibe un ACK dentro de un lapso, se retransmite el mensaje. Si realmente se perdió el mensaje, el server toma la retransmisión del requerimiento como la original.
3) **El mensaje de respuesta del server se pierde.**
   Se depende nuevamente del timer, aunque es más difícil de implementar ya que el kernel del cliente no sabe si sucede 2) o 3) o el server está lento. Existen operaciones que son idempotentes (tales que no causan daño alguno el repetirlas o sea que no cambian su valor) como por ejemplo leer una cadena de bytes de un archivo. El problema es con las otras operaciones (por ejemplo la transferencia electrónica de fondos), y una solución para esto, es que el kernel ponga un número de secuencia en el mensaje así se podrán identificar cuál es original y cual la copia y cotejar si se ha procesado el pedido. Otra solución es ponerle a cada mensaje un header que identifique si es el original o una copia. La idea es que una solicitud original siempre puede ejecutarse inmediatamente pero las solicitudes que son copias requieren de mayor cuidado.
4) **El server se cae luego de recibir el requerimiento.**
   Esto tiene un problema un poco mayor. Como ser:
   a) si el server ejecutó el pedido y se cayó antes de enviar la respuesta en cuyo caso el cliente debería aplicar alguna rutina de manejo de excepciones, y
   b) si el server se cayó justo antes de ejecutar el pedido, situación que el cliente debe abordar reiterando el pedido al servidor.
   El núcleo del cliente no puede saber cuál de las dos situaciones ha ocurrido. Hay cuatro formas de solución:
   1) *At Least Once* : (1 o más) Consiste en esperar a que el Server levante de nuevo, y mandar la operación otra vez. Esto es conveniente cuando se trata de operaciones idempotentes.
   2) *At Most Once :* (0 o 1) Se da por vencido al instante y reporta un error. Asegura que el RPC se hace a lo sumo una vez o puede no completarse nunca.
   3) *Exactly Once* : Es la deseable pero no hay forma de garantizarlo. Siempre va a existir un punto en donde se pierde todo rastro de lo hecho hasta el momento.
   4) *Don't Know??* : Si se cae un server, el cliente no promete nada, la operación se pudo ejecutar de 0-n veces. Fácil de implementar.
5) **El cliente se cae.**
   Esto implica que nadie va e estar esperando la respuesta del server quedando computaciones huérfanas (ORPHANS) que utilizan CPU, pueden lockear archivos, y utilizar recursos cuando realmente no se necesita. Además si el cliente se recupera rápidamente y se retransmitió el mensaje, puede recibir dos mensajes respuesta.
   a) Una solución puede ser que el stub cliente lleve un log de los mensajes que va enviando, así cuando bootea se chequea el log y se mata explícitamente aquellos ORPHANS que hubieran quedado (EXTERMINACION). Una desventaja de esto es que hay que escribir a disco cada vez. Y puede no funcionar, ya que un RPC puede a su vez hacer otro RPC y crear GRANDORPHANS (huérfanos de huérfanos), y demás descendientes que son imposibles de localizar.
   b) Otra solución se la conoce como REENCARNACIÓN que divide el tiempo en generaciones numeradas secuencialmente. O sea cuando un cliente rebootea, hace broadcast de un mensaje indicando a todas las máquinas que se inicia una nueva generación. Al recibir este mensaje, cada máquina se encarga de hacer un kill a todos aquellos procesos pertenecientes a una generación anterior. Si queda vivo algún ORPHAN, cuando llegue su reply inmediatamente se va a reconocer que es inválido ya que es de otra generación.
   c) Se basa en la anterior y se llama REENCARNACIÓN SUAVE. Cuando llega un broadcast indicando una nueva generación, la máquina ubica a todas las computaciones remotas y chequea si todavía sigue existiendo su dueño. Si no lo encuentra recién ahí toma la decisión de eliminarlo.
   d) Se la conoce como EXPIRACIÓN, se le da un lapso de tiempo a cada cómputo RPC para que realice su trabajo. Si este no puede terminar, tiene que pedir explícitamente que se le agrande el quantum. El problema aquí es la elección de un valor adecuado del valor del lapso de tiempo.

Todas estas soluciones no son recomendables en la práctica. La eliminación de un huérfano puede tener consecuencias imprevisibles. Por ejemplo, supongamos que un huérfano haya bloqueado uno o más registros de una base de datos, si se lo elimina súbitamente estos bloqueos pueden permanecer para siempre. Además un huérfano podría crear entradas en alguna ubicación remota de procesos que se ejecutarán en un futuro con lo cual la eliminación del padre no aseguraría la eliminación de todos sus rastros.

22.7. - **Aspectos de la implantación**

Muchas veces el éxito de un sistema distribuido descansa en su desempeño, éste a su vez depende de la velocidad de comunicación y ésta de la implantación. Veremos ahora algunos aspectos en lo cuáles las decisiones tomadas afectan la performance del sistema.

22.7.1. - **PROTOCOLOS RPC**

** La primera decisión es elegir entre un protocolo orientado a conexión o uno sin conexión:*

Conexión:

Ventajas: comunicación más fácil, el núcleo del cliente no debe preocuparse de si los mensajes se pierden o de si no hay reconocimiento.

Desventajas: En una LAN tiene pérdida de desempeño debido a que todo este software adicional estorba, además la ventaja de no perder los paquetes no tiene sentido ya que las LAN son confiables en esto.

Sin Conexión:

En general en sistemas dentro de un único edificio se utilizan protocolos sin conexión. Mientras que en redes grandes se utiliza uno orientado a conexión.

** Utilizar un protocolo estándar o alguno diseñado en forma específica para RPC, por ejemplo:*

* IP (o UDP, integrado a IP) tiene puntos a favor:

-- Ya está diseñado (ahorra un trabajo considerable)

-- Se dispone de muchas implementaciones

-- Estos paquetes se pueden enviar y recibir por casi todos los sistemas UNIX

-- Los paquetes IP o UDP se pueden transmitir en muchas de las redes existentes. En resumen IP y UDP son fáciles de utilizar, pero el lado malo es el desempeño, la cantidad de información que se agrega en el encabezado del mensaje incrementa mucho el overhead del sistema.

** Utilizar un protocolo especializado para RPC que, a diferencia de IP, o tiene que trabajar con paquetes que han estado brincando a través de la red durante unos cuantos minutos y luego aparecen de la nada en un momento inconveniente*.

Por supuesto debe ser inventado, implantado y probado lo que crea un trabajo adicional. Además el resto del mundo no se pone contento cada vez que nace un nuevo protocolo.

** Longitud del paquete y el mensaje:*

La realización de un RPC tiene un costo excesivo fijo de gran magnitud independiente de la cantidad de datos enviados, por lo que se esperaría que el protocolo y la red permitan transmisiones largas.

22.7.2. - **RECONOCIMIENTOS**

En casos que la RPC de gran tamaño tenga que dividirse en muchos paquetes es necesario determinar una estrategia de reconocimientos.

* *Protocolo Detenerse y esperar (stop and wait protocol)*: establece que el cliente envíe el paquete y espere un reconocimiento antes de enviar el segundo paquete.

* *Protocolo de Chorro (Blast Protocol)*: establece que el cliente mande todos los paquetes y luego espere el reconocimiento del mensaje completo. Cuando se pierde un paquete el servidor puede optar por abandonar todo, no hacer nada y esperar que el cliente haga un receso y vuelva a enviar todo el mensaje, o bien guardar lo que llegó bien y pedir que se retransmita el paquete perdido (Repetición selectiva, es buena en redes de área amplia).

Existe otra consideración más importante que es el **control de flujo**. Hay veces que el chip de interfase de red no permite recibir un número ilimitado de paquetes adyacentes debido a la capacidad del mismo. Cuando un paquete llega a un receptor que no lo puede recibir se produce un error de sobreejecución (overrun error) y el paquete se pierde (esto puede ocurrir en protocolos a chorro pero nunca en detenerse y esperar). Un emisor inteligente puede insertar un retraso entre los paquetes (esperando ocupado o iniciar un cronómetro y hacer algo mientras).

Si por otro lado la sobreejecución se debe a la capacidad finita del buffer en el chip de la red, entonces el emisor puede enviar n paquetes y después un espacio considerable, o definir un protocolo para que envíe un reconocimiento después de cada n paquetes.

Los protocolos especializados para RPC tienen un desempeño mucho mejor que los sistemas basados en IP o UDP, por un amplio margen.

Si se pierde el mensaje de reconocimiento en la práctica el servidor puede inicializar un cronómetro al enviar la respuesta y descartarla cuando llega el reconocimiento o se termina el tiempo. Además se puede interpretar una nueva solicitud del cliente como un signo de que llegó bien la respuesta.

22.7.3. - **RUTA CRITICA**

La serie de instrucciones que se ejecutan con cada RPC se llama ruta crítica la cual la podemos visualizar en la figura.

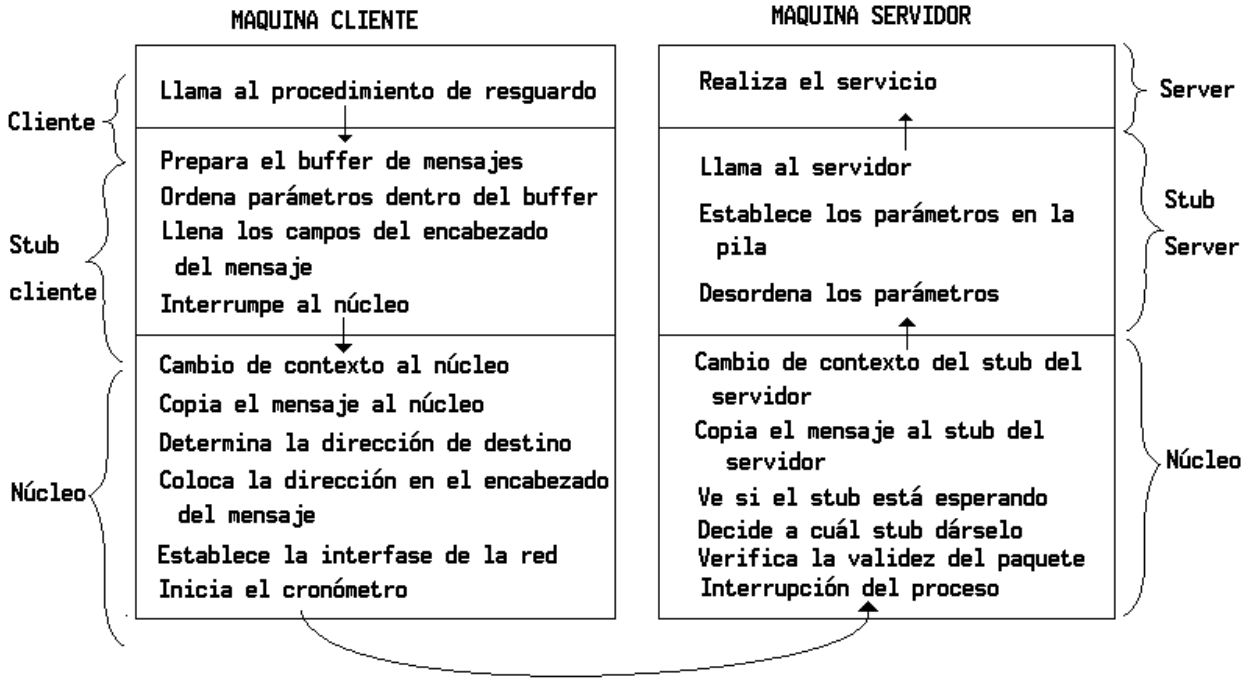

Fig. 22.3. - Ruta crítica del cliente al servidor.

Los diseñadores están interesados en determinar en qué porción de la ruta crítica se consume más tiempo. Luego de una serie de estudios y mediciones realizados los consejos que los expertos nos dan son los siguientes:

1) Evitar el uso de hardware extraño ya que si en dos etapas consecutivas de la ruta crítica intervienen componentes disímiles puede producirse una demora considerable.
2) Tampoco están contentos de haber basado su sistema en UDP debido al costo en tiempo de calcular la suma de verificación.
3) El uso de espera ocupada hubiera reducido el tiempo de cambio de contexto al espacio del usuario.

22.7.4. - **COPIADO**

El numero de veces que se debe copiar un mensaje varía de uno a ocho según el hard, soft y tipo de llamada. Hay varias técnicas para tratar de reducir esto. Una característica del hardware que es de gran ayuda es la **Dispersión-Asociación**.

Un chip de la red que realice la dispersión asociación se puede configurar de tal manera que organice un paquete mediante la concatenación de dos o más buffers de memoria. El hecho de poder asociar un paquete a partir de varias fuentes elimina el copiado. En general es más fácil eliminar el copiado en el emisor que en el receptor.

22.7.5. - **MANEJO DE CRONÓMETRO**

La cantidad de tiempo de máquina que se dedica al manejo de cronómetros no debe subestimarse. El establecimiento de un cronómetro requiere de la construcción de una estructura de datos que especifique el momento en que el cronómetro debe detenerse y la acción a realizar en caso de que eso suceda. Generalmente se usa una lista de estos procesos pendientes de que e cronómetro llegue a cero ordenada desde el de menor tiempo al mayor.

En la práctica muy pocos cronómetros ocupan todo su tiempo con lo cual el trabajo de introducir y eliminar un proceso de esta lista es un esfuerzo desperdiciado.

Además si el valor del cronómetro es muy pequeño habrá demasiadas retransmisiones y si es muy grande existirán demoras demasiado altas para la detección de la pérdida de un mensaje.

Esto sugiere que podría utilizarse una tabla de procesos, en donde una entrada contiene toda la información correspondiente a cada proceso del sistema. La activación de una RPC consta ahora de la suma de la longitud del tiempo de expiración a la hora actual y su almacenamiento en la tabla de procesos. Esto obliga a que el núcleo revise periódicamente (por ejemplo cada segundo) la tabla de procesos y si encuentra un valor distinto de cero (cero indicaría que no hay cronómetro activo para el proceso) que sea menor o igual que la hora actual entonces un cronómetro ha expirado. Los algoritmos que operan por medio de una tabla como ésta se denominan **algoritmos de barrido** (sweep algorithms).

22.7.6. - **ÁREA DE PROBLEMAS**

Hay varios problemas, algunos de ellos son:

- No se puede implantar el permiso para el acceso irrestricto de los procedimientos a las variables globales de forma remota y viceversa, a la vez que la prohibición de este acceso viola el principio de transparencia.
- Los lenguajes débilmente tipificados (por ejemplo C en donde se pueden escribir procedimientos que multipliquen dos vectores sin indicar su tamaño) traen problemas a los stubs en el ordenamiento de los parámetros.
- Transferir como parámetro un apuntador a una gráfica compleja ya que en un sistema con RPC el stub del cliente ni tiene forma de encontrar toda la gráfica.
- No siempre es posible deducir los tipos de los parámetros, ni siquiera a partir de una especificación formal del propio código. Por ejemplo la instrucción *printf* permite una cantidad arbitraria de parámetros de cualquier tipo.