TESINA DE LICENCIATURA

Título: Concurrencia en RPC: Análisis de Factibilidad y Relación con Protocolos de Comunicaciones
Autores: Franco Sebastián Montanaro
Director: Fernando G. Tinetti
Codirector: -
Asesor profesional: -
Carrera: Licenciatura en Sistemas

Resumen

En el presente trabajo se busca investigar, analizar, experimentar y mejorar el funcionamiento de ONC RPC (Open Network Computing Remote Procedure Call), específicamente enfocado hacia la incorporación de procesamiento concurrente en el servidor. Se busca automatizar esta incorporación aprovechando la propia derivación automática de código de herramientas disponibles en el contexto de RPC, es decir generando código automático además del (o modificando al) que genera el propio generador de código o compilador de RPC.
Específicamente relacionado con el procesamiento concurrente el objetivo es analizar y experimentar tanto a nivel de procesos como de hilos de ejecución (*threads*), es decir, intentar obtener un servidor fiable con multiproceso y multithreading. Específicamente relacionado con los protocolos de comunicación, se analizarán las alternativas específicas de los protocolos de transporte TCP y UDP que, al ser esencialmente diferentes, pueden generar problemas completamente disjuntos en cuanto a la factibilidad y eventual implementación de concurrencia del servidor RPC.

Palabras Claves

RPC
Sun RPC
Concurrencia
Multiproceso
Multithreading
TCP
UDP
rpcgen
Portmapper

Trabajos Realizados

Se analizó el problema de la concurrencia de los servidores de RPC. Se generaron herramientas automatizadas para incluir concurrencia en servidores de RPC con procesos y con threads. En ellas, se automatiza la generación de código necesario para mantener las características de Sun RPC y posibilitar concurrencia en el lado del servidor. Esta generación de código es tan transparente para el programador como la herramienta rpcgen original misma.

Conclusiones

Se obtuvo como resultado la certeza de que, en base a los archivos generados por el compilador rpcgen y aplicando a éstos las modificaciones necesarias, se puede brindar concurrencia del lado del servidor utilizando procesos y threads; tanto para el protocolo UDP como TCP.

Trabajos Futuros

En base a la generación automática de código de rpcgen se puede avanzar en al menos dos direcciones. Por un lado, se puede trabajar sobre la generación de archivos de especificación a partir de código no distribuido. Por otro lado, se podría avanzar en la semántica de control de errores que implementa rpcgen y que no necesariamente es la mejor para todos los casos.

Fecha de la presentación: Octubre 2014

Índice General

Índice de Figuras

Resumen

ONC RPC (Open Network Computing Remote Procedure Call) es un protocolo de llamada a procedimiento remoto inicialmente desarrollado por el grupo ONC de Sun Microsystems como parte del proyecto de su sistema de archivos de Red NFS (Network File System), algunas veces denominado Sun ONC o Sun RPC. RPC es una poderosa técnica para la construcción de aplicaciones distribuidas basadas en el esquema cliente/servidor que permite que los programas llamen a subrutinas que se ejecutan en un sistema remoto; donde el programa que realiza la llamada envía un mensaje incluyendo los parámetros del procedimiento al proceso servidor y espera un mensaje de respuesta que contendrá los resultados de la llamada.
RPC trabaja sobre los protocolos TCP y UDP y la mayoría de las implementaciones hacen uso de un generador de código llamado rpcgen y del estándar XDR para la codificación de datos.

Aunque la implementación original de Sun Microsystems para el sistema operativo Solaris contemplaba el procesamiento concurrente en los servidores RPC [1], las implementaciones actuales de Linux no lo proveen. Más aún, la documentación de rpcgen al respecto en Linux es escasa.

Sin lugar a dudas, superar los problemas de procesamiento secuencial, es decir, la imposición de procesar de a una solicitud por vez, es más que necesario en los servidores en el contexto del modelo de procesamiento cliente/servidor. Sin detallar extensivamente las desventajas de los servidores secuenciales, podemos citar la pérdida de tiempo, el mal uso de los recursos y la capacidad actual de los sistemas de cómputo de procesamiento paralelo debido a la disponibilidad de múltiples núcleos (CPUs) en el procesador.

Partiendo de la facilidad que brinda rpcgen al programador, ya que este sólo tiene que ocuparse de las implementaciones de las funciones remotas y la lógica del cliente, se analizará y detallará en la presente tesina los pasos necesarios para extender las funcionalidades actuales y permitir procesamiento concurrente del lado servidor mediante el uso de procesos y de threads; mostrando de esta manera la evolución de las diferentes formas de procesamiento en el tiempo.

1. Introducción

En este capítulo se presentan los conceptos principales de los sistemas distribuidos, incluyendo las definiciones que han elaborado los autores de mayor referencia en el tema.
Se describen también el modelo de arquitectura cliente/servidor y la comunicación de procesos para asentar los conceptos esenciales en los que se basa RPC.
Luego se brinda una introducción a RPC detallando las consideraciones necesarias para su diseño e implementación, para poder explicar el funcionamiento de una invocación remota básica.
Finalmente y en base a lo comentado, se detallan los objetivos propuestos en esta tesina.

1.1 Sistemas distribuidos

Según se define en [3], un sistema distribuido es aquel en el que los componentes localizados en computadoras, conectados a través de una red, comunican y coordinan sus acciones únicamente mediante el intercambio de mensajes. Esta definición lleva a que dichos sistemas presenten las siguientes características:

- Concurrencia de los componentes.
- Carencia de un reloj global.
- Fallos independientes de los componentes.

Compartir recursos, tanto de hardware como de software, es uno de los motivos principales para construir sistemas distribuidos.

Los desafíos que surgen en la construcción de sistemas distribuidos son la heterogeneidad de sus componentes; su carácter abierto, que permite que se puedan añadir o reemplazar componentes, la seguridad y la escalabilidad; el tratamiento a fallos, la concurrencia de sus componentes y la transparencia:

- Heterogeneidad: los sistemas se construyen sobre diferentes redes, sistemas operativos, hardware y lenguajes de programación. Técnicas como protocolos de comunicación y sistemas middleware son utilizados para solventar estas diferencias.
- Extensibilidad: es la característica que determina si el sistema puede extenderse y volver a implementarse en diversos aspectos. Esta se determina por el grado en el cual se pueden añadir nuevos servicios de compartición de recursos.
- Seguridad: entre los recursos de información que se ofrecen y se mantienen en los sistemas distribuidos, muchos tienen un alto valor intrínseco para los

usuarios; por lo que su seguridad es de considerable importancia. La seguridad de los recursos se compone a su vez de:

- Confidencialidad: protección contra el descubrimiento por individuos no autorizados.
- Integridad: protección contra la alteración o corrupción.
- Disponibilidad: protección contra interferencia con los procedimientos de acceso a los recursos.

- Escalabilidad: se dice que un sistema es escalable si conserva su efectividad cuando ocurre un incremento significativo en el número de recursos y el número de usuarios.
- Tratamiento de fallos: los sistemas computacionales a veces fallan. Cuando esto ocurre, los programas pueden producir resultados incorrectos o detenerse antes de haber completado la operación que realizaban. Los fallos en un sistema distribuido son parciales, es decir, algunos componentes fallan mientras que otros continúan funcionando; por lo que su tratamiento conlleva una considerable complejidad.
- Concurrencia: tanto los servicios como las aplicaciones proporcionan recursos que pueden compartirse entre los clientes en un sistema distribuido. Existe por lo tanto la posibilidad de que varios clientes intenten acceder a un recurso compartido al mismo tiempo; por las que dichas operaciones deben sincronizarse de forma de mantener la consistencia y evitar escenarios no deseados.
- Transparencia: se define como la ocultación de la separación de los componentes en un sistema distribuido, de forma que se perciba el sistema como un todo más que como una colección de componentes independientes.

1.2 Arquitectura distribuida cliente/servidor

Muchas de las aplicaciones de procesamiento de datos actuales utilizan una arquitectura distribuida en la cual un usuario debe acceder a datos o servicios a través de la red. El modelo cliente/servidor es una arquitectura de software donde un proceso cliente requiere y consume servicios de un proceso servidor. Este concepto fue utilizado inicialmente en la década del 80 y fue tomando relevancia en el corto plazo. Presenta una infraestructura versátil, modular y basada en mensajes la cual tiene por objetivo mejorar la usabilidad, flexibilidad e interoperabilidad de los sistemas en relación a los modelos centralizados [4].

Un cliente es un proceso (programa en ejecución) que envía un mensaje a otro proceso denominado servidor, solicitándole realizar una tarea (servicio). El servidor, que se ejecuta en otra máquina de la red, en una secuencia clásica ejecuta entonces la tarea requerida y genera algún tipo de resultado que retorna al cliente, quedando a la espera de una nueva petición.

Esta interacción entre cliente y servidor es conocida como invocación remota.
Hay que señalar que por defecto los términos cliente y servidor se refieren a procesos y no a las máquinas en la que estos son ejecutados, aunque en la práctica dichos términos se refieren también a las propias computadoras.
Un mismo proceso puede ser tanto un cliente como un servidor, puesto que los servidores a veces invocan operaciones en otros servidores. Los términos cliente y servidor se aplican a los roles desempeñados en una única solicitud. Ambos son distintos en muchos aspectos, ya que los clientes son activos y los servidores pasivos; y los servidores se están ejecutando continuamente mientras que los clientes sólo lo hacen el tiempo que duran las aplicaciones de las que forman parte.

1.3 Comunicación de procesos: sockets

Un socket es un canal de comunicación entre dos procesos. Los protocolos de transporte UDP y TCP utilizan la abstracción de sockets (conectores), para la comunicación entre procesos, la cual consiste en la transmisión de un mensaje entre un conector asociado a un proceso origen y otro a un proceso destino, como puede verse en la Figura 1.1 [3].
Esta abstracción permite manejar de una forma simple la comunicación entre procesos, aunque estos se encuentren en sistemas diferentes, sin necesidad de conocer el funcionamiento de los protocolos de comunicación subyacentes.
En el interior de un proceso, un socket se identifica por un descriptor de la misma naturaleza que los que identifican los archivos.

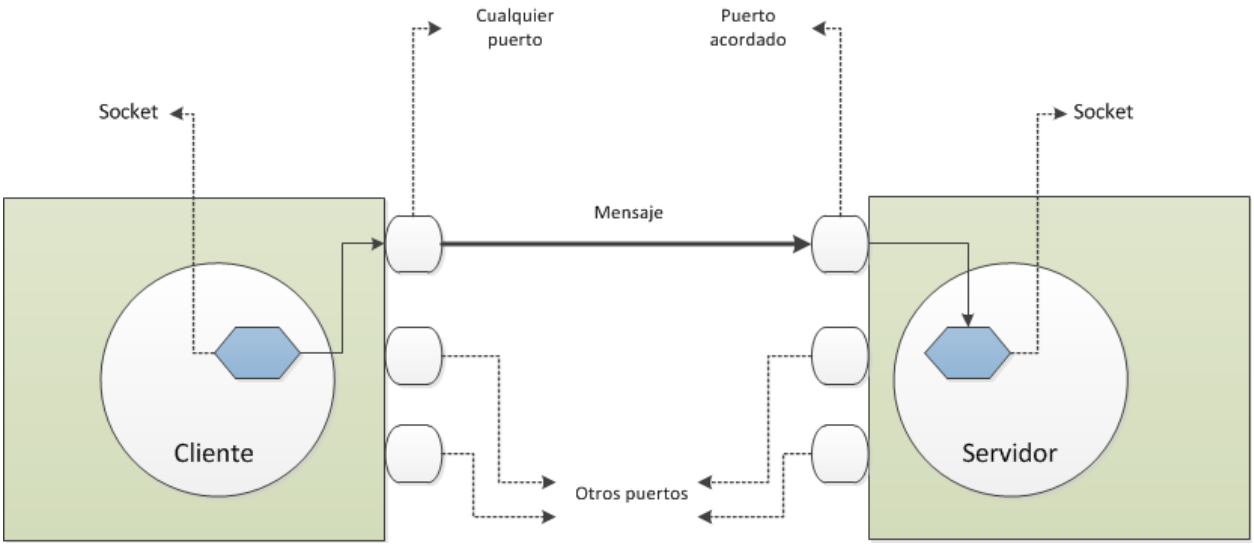


Figura 1.1. Comunicación entre sockets

En [3], se caracteriza a un socket por tres atributos:

1. Dominio: nos dice dónde se encuentran los procesos que se van a comunicar. Aunque cabe aclarar que existen otros valores de dominio, los de mayor relevancia son:

    - AF_UNIX: utilizado cuando los procesos están en el mismo sistema.
    - AF_INET: utilizado cuando los procesos están en distintos sistemas y éstos se hallan unidos mediante una red TCP/IP.

2. Protocolo: especifica qué protocolo se va a usar, llegado el caso en que el mecanismo de transporte permita más de uno.
3. Tipo: los protocolos de internet proveen 2 niveles diferentes de servicio:

    - Sockets Stream: son los más utilizados. Hacen uso del protocolo TCP, el cual nos provee un flujo de datos bidireccional, secuenciado, sin duplicación y libre de errores.
    - Sockets Datagram: hacen uso del protocolo UDP, el cual nos provee un flujo de datos bidireccional, pero los paquetes pueden llegar fuera de secuencia, pueden no llegar o contener errores. Por lo tanto el proceso que recibe los datos debe comprobar la secuencia, eliminar duplicados y asegurar la integridad. Se llaman también 'sockets sin conexión', porque no hay que mantener una conexión activa, como en el caso de sockets stream.

Los sockets presentan un conjunto de primitivas, los cuales nombraremos sin profundizar para poder entender el funcionamiento general. Las mismas son.

- socket: crea un socket.
- bind: asocia la dirección local (IP + número de puerto) al socket.
- listen: pone un socket en modo espera no bloqueante.
- connect: establece la conexión con una máquina remota, exigiendo la dirección del proceso (IP + número de puerto).
- accept: indica que se ha recibido y aceptado una conexión de un cliente (tomada de la cola de espera) y genera un nuevo socket para que el servidor intercambie información con el cliente.
- send, sendto: envío de datos.
- recv, recvfrom: recepción de datos.
- close: finaliza una conexión.

La relación entre estas primitivas puede verse en los esquemas de funcionamiento en la

comunicación cliente/servidor a nivel de procesos, representados en las Figuras 1.2 y 1.3 [3].

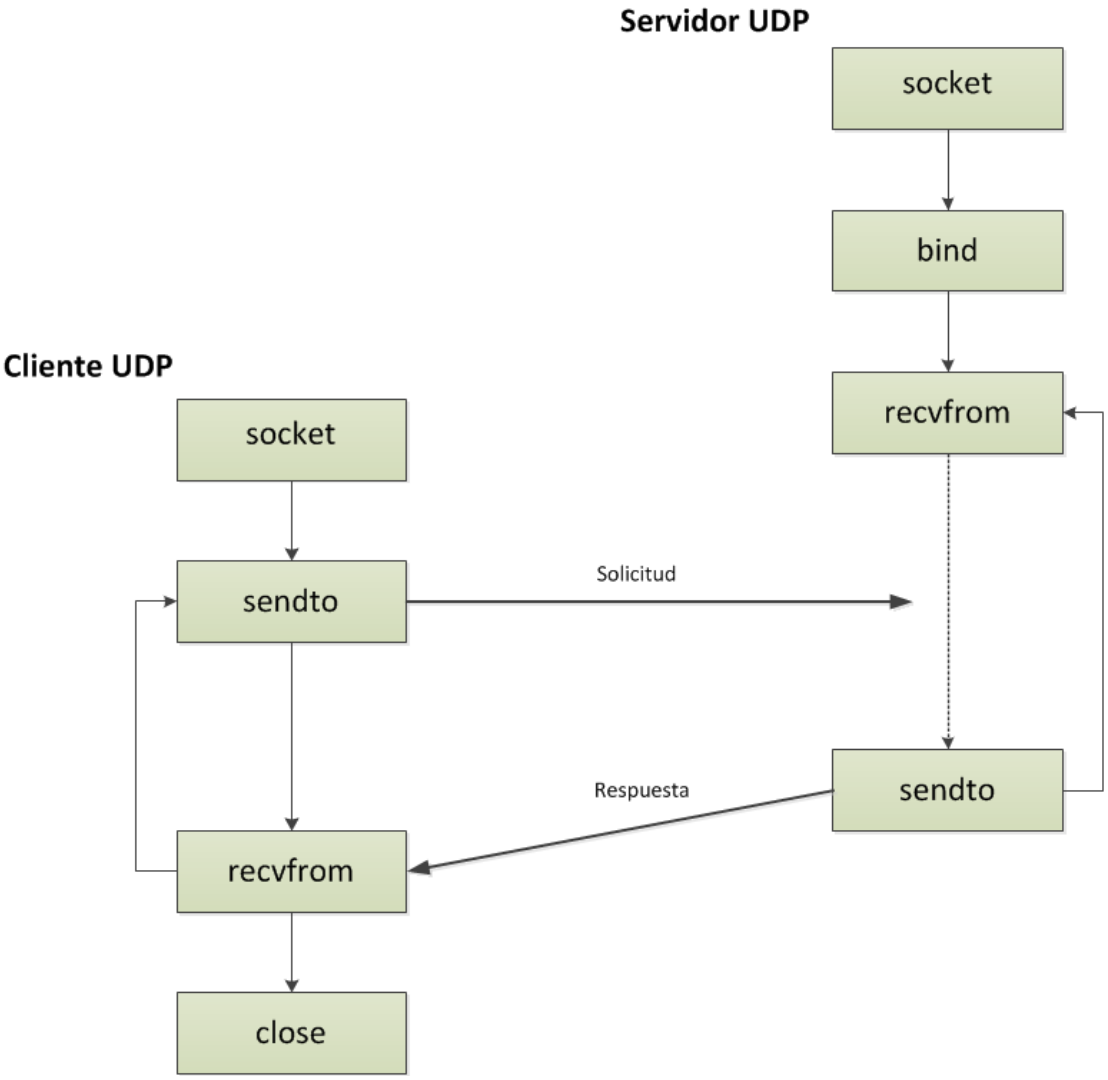


Figura 1.2. Esquema de comunicación UDP

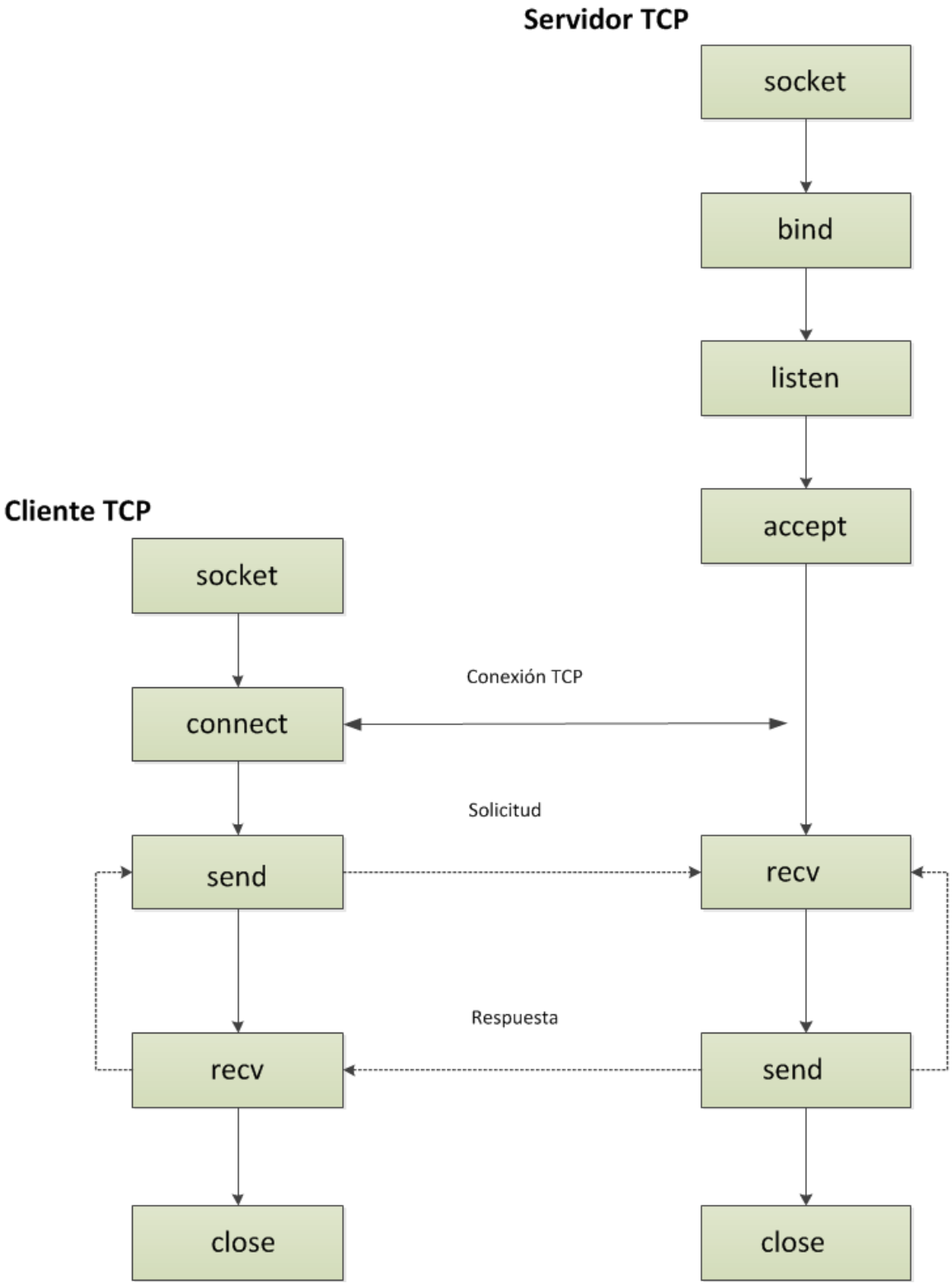


Figura 1.3. Esquema de comunicación TCP

1.4 Llamadas a procedimientos remotos

Muchos sistemas distribuidos se han basado en el intercambio explícito de mensajes entre procesos. Sin embargo, los procedimientos *send* y *recv* no ocultan la comunicación en absoluto, lo cual es importante para lograr transparencia en estos tipos de sistemas.
La manera de dar solución a esta problemática fue sugerida por Birren y Nelson en [17], permitiendo a los programas invocar procedimientos localizados en otra máquina y dando lugar al método actualmente conocido como Remote Procedure Call (RPC).
RPC es entonces, un mecanismo para permitir la comunicación entre un proceso cliente y uno servidor, generalmente localizados en máquinas diferentes. Cuando un proceso en una máquina A llama a un procedimiento en una máquina B, el proceso llamador en A se suspende, tomando lugar la ejecución del procedimiento en B. La información desde el proceso llamador al llamado es transportada en los parámetros y retorna como resultado del procedimiento, es decir, no existe un pasaje de mensajes visible para el programador [4].
Este método implica: un procedimiento, una rutina para la conversión de parámetros (*stub*) y una entidad de transporte para la conexión de red en ambos extremos de la comunicación, es decir, en el cliente y el servidor. Cuando un proceso cliente intenta comunicarse con uno servidor, hace una llamada a su *stub* asociado en su espacio de direcciones. Luego, es este quien empaqueta los parámetros en un mensaje e invoca a la entidad de transporte para enviar dicho mensaje al proceso servidor. La entidad de transporte del servidor deriva el mensaje a su *stub*, quien obtiene los parámetros asociados y realiza la llamada al procedimiento solicitado. Estas etapas se llevan a cabo en el orden inverso al finalizar la ejecución del procedimiento, retornando el control al cliente y finalizando el ciclo de la comunicación.
En este caso, el hecho de que el procedimiento está siendo invocado remotamente es transparente para el cliente.

Aunque el concepto de RPC descripto es simple, Y. Liu indica en [14] que para su diseño e implementación se deben tener en cuenta consideraciones propias de la comunicación remota que no existen o son fácilmente manejables cuando la llamada se realiza localmente:

- Performance: las llamadas a procedimientos remotos presentan una mayor sobrecarga de procesamiento en relación a las llamadas locales, por lo que su rendimiento tiende a ser menor. Esto se debe, a grandes rasgos, al tiempo y espacio extra necesario para establecer la comunicación, la transferencia por la red y el aplanamiento de datos.
- Binding: antes de que un programa pueda llamar a un procedimiento, es necesario establecer una relación (enlace) entre los mismos. En el caso de llamadas locales, esto se lleva a cabo de manera estática en tiempo de compilación. Cuando hablamos de llamadas remotas, no es tan simple ya que los procesos residen en máquinas diferentes. De esta manera, enlazar un procedimiento remoto implica localizar la máquina en la cual reside y el proceso particular que lo implementa.

- Semántica de llamada: una diferencia fundamental entre procedimientos locales y remotos radica en la cantidad de veces que cada uno puede llegar a ser ejecutado. Cuando hablamos de llamadas locales, la respuesta es fácil de obtener: si el procedimiento llamado retorna, quien realizó la llamada sabe que el mismo fue ejecutado exactamente una vez y, si la máquina se cae durante la ejecución, tanto el proceso llamador como el llamado se ven afectados y nada puede hacerse; por lo que no hay necesidad de preocuparse por la cantidad de veces que el procedimiento se ejecutó. Sin embargo, este no es el caso con invocaciones remotas, donde ciertos escenarios pueden darse y el número de ejecuciones puede llegar a desconocerse. Por ejemplo, el mensaje de solicitud (o el de respuesta) puede perderse y el servidor puede caer. Debido a esto, si el cliente no recibe respuesta luego de un tiempo establecido, no tiene la certeza de si el procedimiento se ejecutó parcialmente, completamente o ni siquiera comenzó. Es más: si el cliente decide volver a transmitir su solicitud, el procedimiento puede llegar a ejecutarse más de una vez.
  En base a esto, pueden diferenciarse 3 semánticas de llamada:

  1. Al menos una: el procedimiento es ejecutado una o más veces. Esta es considerada una semántica débil, ya que el cliente no puede estar seguro de la cantidad exacta de ejecuciones debido a las fallas mencionadas y las retransmisiones posibles. Esta forma puede lograrse fácilmente permitiendo al cliente retransmitir su solicitud hasta obtener una respuesta. Sin embargo, para que sea de utilidad, el procedimiento remoto debe ser idempotente, es decir, el resultado de múltiples ejecuciones es idéntico a una sola.
  2. A lo sumo una: es la utilizada por la mayoría de los sistemas RPC. En ella, el procedimiento puede ser ejecutado una vez o ninguna. Si el cliente recibe respuesta normalmente, puede estar seguro de que su llamada fue ejecutada una vez. Sin embargo, si recibe algún tipo de error, no puede saber si el mismo fue ejecutado completamente, parcialmente o ni siquiera comenzó.
  3. Exactamente una: considerada una semántica fuerte, es la menos implementada por los sistemas RPC. En ella, como indica su nombre, el procedimiento se ejecuta una sola vez a pesar de las posibles fallas. Esto es difícil de lograr debido a la posibilidad de la caída del servidor, lo cual requiere mecanismos de recuperación ante fallos para distinguir entre llamadas previas y posteriores a la caída y mantener los datos persistentes.

- Sincronización: existen dos formas en las cuales un procedimiento remoto puede llevarse a cabo: sincrónicamente o asincrónicamente con respecto al llamador. Con la forma sincrónica (o bloqueante), el llamador se bloquea hasta que recibe una respuesta del servidor; mientras que de la forma asincrónica (no bloqueante), tanto el llamador como el llamado pueden ejecutarse en paralelo.

- Confiabilidad: aspectos como la pérdida, corrupción y duplicación de mensajes, así como la caída de alguno de los nodos de la comunicación son la causa de que la confiabilidad de los sistemas RPC sea una de las tareas de mayor dificultad para los diseñadores. Esto originó el uso de diversas técnicas para fallas en la comunicación, tales como números de secuencia, sumas de comprobación y tiempos de espera. De hecho, algunos sistemas RPC dependen de un protocolo de transporte confiable o utilizan ciertas técnicas en capa de aplicación para funcionar confiablemente.
- Manejo de excepciones: Un mecanismo para el manejo de excepciones permite que un cliente lleve a cabo ciertas acciones de recuperación ante errores. En general, un manejador de excepciones funciona de la siguiente forma: cuando se detecta un error durante una llamada remota, en lugar del resultado del propio procedimiento, un código de error (posiblemente con algunos parámetros) es recibido por el cliente. En base a dicho código, el cliente es capaz de conocer la naturaleza del error y llamar a una rutina de reparación o terminar con su ejecución si no puede resolverlo.
- Administración del servidor: los procesos servidores son diferentes en términos de la forma en que son creados, su tiempo de vida, su población cliente y el compartimiento del espacio de direcciones.
  Existen al menos tres estrategias para organizar dichos procesos:

    1. Los procesos son creados estáticamente, con existencia indefinida, con servicio para múltiples clientes y sin compartir memoria entre sí.
    2. Los procesos son creados bajo demanda por un proceso administrador, el cual se encuentra en una dirección conocida a priori en el servidor. Normalmente, cada uno de estos procesan un cliente particular, terminan al finalizar la solicitud eliminando la sesión de dicho cliente y no comparten memoria con otros procesos.
    3. Un proceso distribuidor es creado estáticamente junto con un conjunto de procesos trabajadores. El proceso distribuidor es responsable de recibir cada solicitud y derivarla a un proceso trabajador, el cual procesará la misma y comunicará el resultado al cliente; volviendo a estar disponible para una nueva solicitud. Generalmente estos procesos comparten memoria.

- Protocolo de transporte: los componentes de RPC se encuentran por encima de la capa de transporte. Esta es la responsable de la transmisión de solicitudes y respuestas entre clientes y servidores; y la podemos clasificar en base a si es orientada a conexión o no. Los protocolos orientados a conexión son confiables, ya que garantizan que cada paquete llegue exactamente una vez, en el orden correcto y sin corromper. El ejemplo clásico para este tipo es TCP (Transmission

Control Protocol). Por otro lado, los protocolos no orientados a conexión son no confiables, en el sentido de que no brindan ninguna de las garantías mencionadas anteriormente. El ejemplo clásico para este tipo es UDP (User Data Protocol).

- Representación de los datos: existen diferentes arquitecturas de computadoras con diferentes representaciones internas de los tipos de datos. Estos pueden llegar a diferir, por ejemplo, en el orden de los bytes para ciertos tipos (big-endian, little-endian), el tamaño (16 bits, 32 bits) y la representación de los bits (ASCII, EBCDIC, etc.). Como el cliente y el servidor pueden correr en máquinas distintas con diferentes arquitecturas, las representaciones de los parámetros y los resultados podrían llegar a no ser iguales en ambos. De esta manera, se necesita una traducción de los datos enviados en cada comunicación para que los mismos puedan ser interpretados correctamente.

1.5 Operación RPC básica

Para entender cómo trabaja RPC, es importante primero interpretar cómo funciona una llamada a un procedimiento convencional. Consideremos una llamada en C como

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

donde *fd* es un entero que referencia a un archivo, *buf* es un arreglo de caracteres dentro del cual serán ubicados los bytes leídos y *nbytes* es un entero que indica la cantidad de bytes a leer. Si la llamada es realizada desde el programa principal, la pila antes de la misma será como se muestra en la Figura 1.4 (a). Para realizar la llamada, el llamador coloca los parámetros en la pila dejándola en el estado que se muestra en la Figura 1.4 (b). Luego de que *read* haya terminado de ejecutarse, coloca el valor de retorno en un registro, elimina la dirección de retorno y transfiere el control al llamador. Éste, obtiene entonces los parámetros de la pila, retornando la misma al estado original que presentaba anteriormente a la llamada.

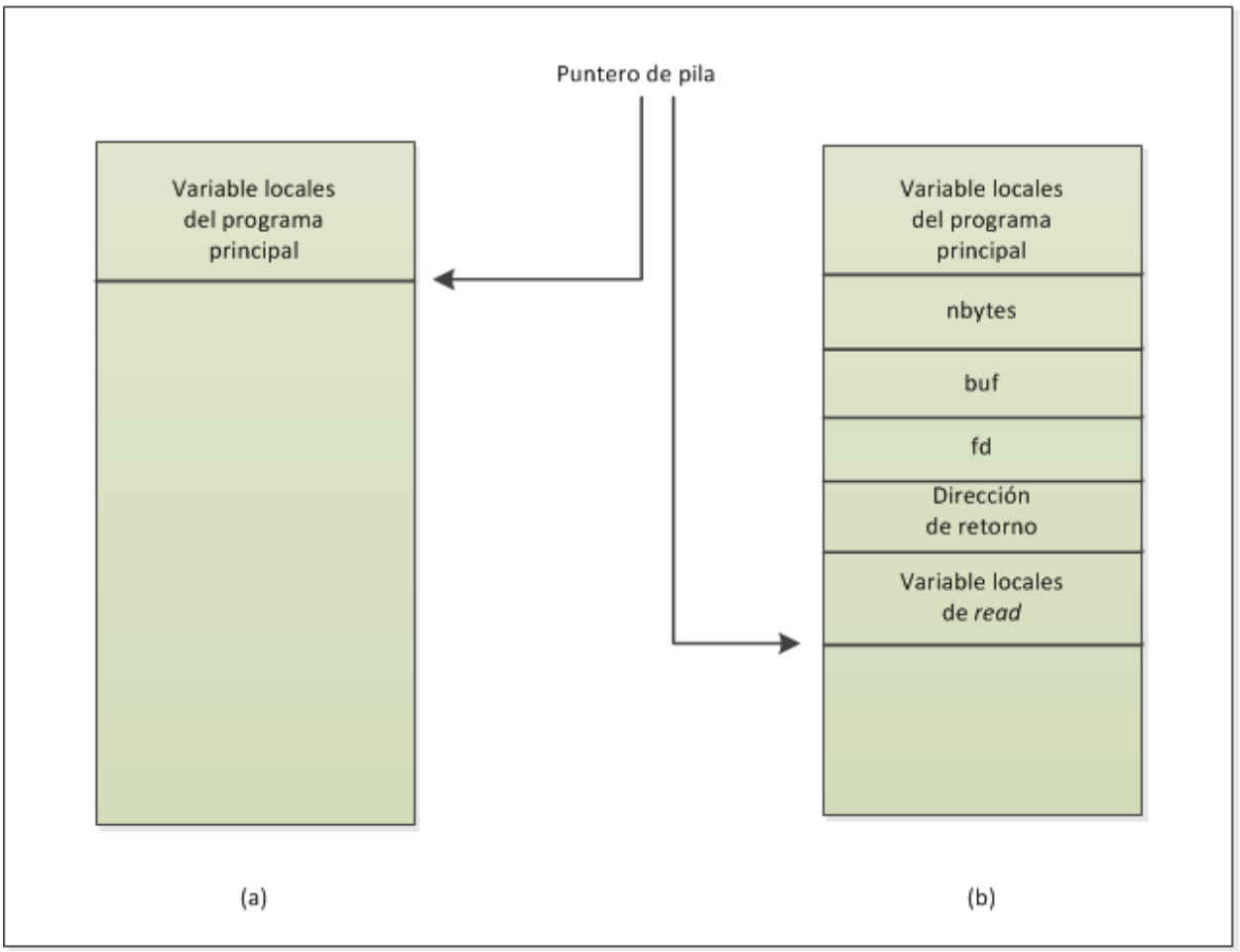


Figura 1.4. Estados de la pila

La idea detrás de RPC es hacer que una llamada remota se parezca lo más posible a una local. En otras palabras, se quiere lograr la mayor transparencia posible (el procedimiento que llama no debería ser consciente de que lo que invoca se está ejecutando en otra máquina).

Supongamos que un programa necesita leer datos de un archivo, por lo que el programador realizará una llamada a *read* en el código para obtener lo que necesita. En un sistema tradicional (con un único procesador), la rutina para *read* es extraída desde su librería correspondiente e insertada dentro del programa objeto. Esta es un procedimiento corto, el cual es generalmente implementado a través de una llamada al sistema equivalente, es decir, un *read* a nivel del sistema operativo. En otras palabras, el procedimiento *read* inicial es como una interfase entre el código del usuario y el sistema operativo local.

A pesar de que el procedimiento realiza una llamada al sistema, éste es llamado de una forma usual colocando los parámetros en la pila, como se muestra en la Figura 1.4 (b).

RPC consigue su transparencia de una manera análoga. Cuando *read* es un procedimiento remoto, la rutina para dicha funcionalidad es extraída desde una librería exclusiva de RPC y se conoce como *stub* (en el cliente). Al igual que nuestra función *read* original, esta última es

llamada usando la secuencia (b). También al igual que la original, realiza una llamada al sistema operativo local. Pero, a diferencia de la original, no le pide datos al sistema operativo. En su lugar, lo que hace es empaquetar los parámetros en un mensaje y solicitar que el mismo sea enviado al servidor como se muestra en la Figura 1.5. Posterior a la llamada a *send*, el stub cliente llama a *recv*, bloqueándose a sí mismo hasta obtener una respuesta.

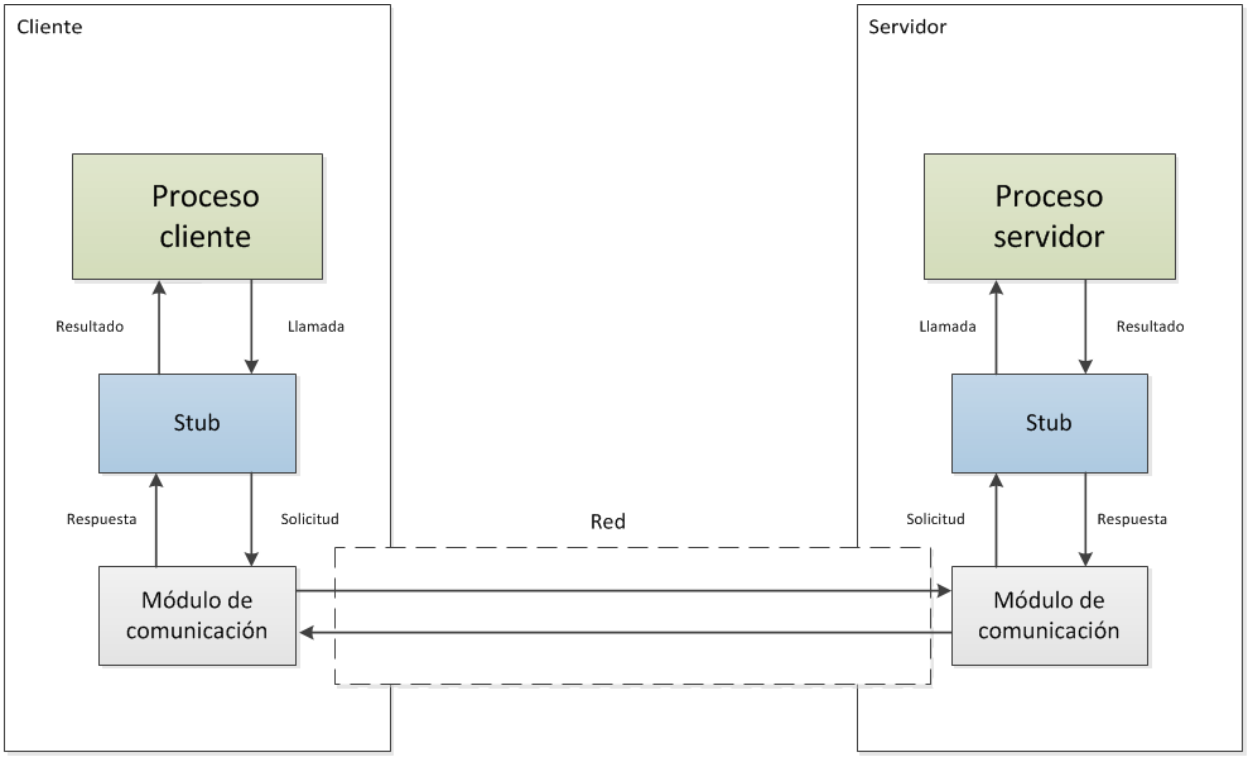


Figura 1.5. Llamada RPC básica

Cuando el mensaje llega al servidor, su sistema operativo se lo pasa al *stub*. Este *stub* servidor es el equivalente al que comentamos anteriormente en el cliente: una porción de código que transforma las solicitudes provenientes de la red en llamadas a procedimientos locales. Generalmente, dicho *stub* habrá llamado a *recv* y estará bloqueado a la espera de solicitudes. El *stub* entonces desempaqueta los parámetros en el mensaje e invoca al procedimiento solicitado de la forma normal**.**

Desde el punto de vista del servidor, es como si estuviera siendo llamado directamente por el cliente: los parámetros y la dirección de retorno están en la pila y nada parece anormal. El servidor lleva a cabo entonces la operación y devuelve su resultado a su llamada de forma usual. Por ejemplo, en el caso de *read*, el proceso servidor llenará el buffer al que apunta el segundo parámetro con los datos (siendo este buffer interno al *stub* del servidor).

Cuando luego de completar la llamada dicho *stub* retoma el control, empaqueta el resultado en

un mensaje e invoca a *send* para enviarlo al cliente. Luego de eso generalmente realiza una llamada a *recv* nuevamente para quedar a la espera de la siguiente solicitud.
Cuando la respuesta llega al cliente, su sistema operativo notifica que esta va dirigida al proceso cliente (en realidad a su *stub*, aunque el sistema operativo no puede ver la diferencia). El *stub* entonces desempaqueta el mensaje, obtiene su resultado, copia el mismo en el buffer del cliente y finaliza, desbloqueando al proceso llamador. Cuando este último retoma el control, todo lo que sabe es que los datos solicitados están disponibles. Desconoce el hecho de que el trabajo fue realizado de manera remota y no por su sistema operativo local. Es este mismo desconocimiento el sentido de todo el esquema: desde el punto de vista del cliente, los servicios remotos son accedidos realizando llamadas normales (locales) y no a través de invocaciones a *send* y *recv.* Todos los detalles del pasaje de mensajes están ocultos por los procedimientos de la librería de RPC, de igual manera en que las llamadas al sistema están ocultas por las librerías tradicionales [4].

1.6 Pasaje de parámetros por referencia

Recordemos que una dirección de memoria (o puntero) tiene significado sólo en el espacio de direcciones del proceso en el que se utiliza y, por definición de sistema distribuido, los procesos clientes y servidor no comparten el mismo. Surge entonces preguntarnos ¿cómo son pasadas las referencias?.
Como primera aproximación a una solución podríamos prohibir este tipo de pasaje; hecho sin validez en la práctica debido a la importancia del mismo. En realidad, no es necesario. En el ejemplo anterior sobre la función *read*, el *stub* del cliente sabe que el segundo parámetro (buffer) apunta a un arreglo de caracteres. Supongamos por un momento que además conoce su longitud. Con esto en mente, una estrategia particular comienza a tomar importancia: copiar el arreglo y enviarlo al servidor, indicando su tamaño. El *stub* del servidor puede entonces llamar al procedimiento con un puntero a este arreglo, incluso aunque el mismo tengo una dirección de memoria diferente a la del cliente. Esto implica que los cambios en la estructura sean realizadas en el área de memoria del servidor; cuyo *stub*, al retomar el control, copiará y enviará la misma al cliente. En resumen, estamos diciendo que el pasaje por referencia fue reemplazado por el de valor-resultado.
Se puede además optimizar lo anterior para ganar eficiencia: si el *stub* del servidor sabe si el buffer es un parámetro de entrada o salida, una de las copias puede eliminarse. Es decir, si el arreglo es un parámetro de entrada, no necesita ser copiado en la respuesta al cliente; de la misma forma en que si es un parámetro de salida no necesita ser copiado en la solicitud [4].

1.7 Objetivos de la tesina

Se busca investigar, analizar, experimentar y en el caso en que sea posible, mejorar el funcionamiento de ONC RPC, todo esto específicamente enfocado hacia la incorporación de

procesamiento concurrente en el servidor. En los casos en que sea posible, se buscará automatizar esta incorporación aprovechando la propia derivación automática de código de herramientas disponibles en el contexto de RPC, es decir generando código automático al propio generador de código o compilador de RPC.

Específicamente relacionado con el procesamiento concurrente el objetivo es analizar y experimentar tanto a nivel de procesos como de hilos de ejecución (threads), es decir, intentar obtener un servidor fiable con multiproceso y multithreading. Específicamente relacionado con los protocolos de comunicación, se deben analizar las alternativas específicas de los protocolos de transporte TCP y UDP que, al ser esencialmente diferentes pueden generar problemas completamente disjuntos en cuanto a la factibilidad y eventual implementación de concurrencia del servidor RPC.

En todos los casos en que sea factible la incorporación automática de procesamiento concurrente, se buscará mantener la semántica original de la especificación y del código RPC generado. Relacionado con esto, el código agregado se diferenciará e identificará claramente respecto del código generado por el compilador de RPC.

2. Sun RPC

En este capítulo se especifica más detalladamente cómo funciona la implementación RPC de Sun en relación a los conceptos detallados en el Capítulo 1. De esta forma, se analizan aspectos como la identificación de los procedimientos, sincronización, semántica de invocación, tiempos de espera y manejo de duplicaciones.
Se introducen también los conceptos de Portmapper y XDR, utilizados por Sun RPC como respuesta a las problemáticas de la ubicación de procedimientos remotos y la codificación de datos, respectivamente.

2.1 Modelo

Sun RPC respeta el modelo de funcionamiento de llamadas descripto inicialmente, en el que un hilo de ejecución lógico permite la comunicación entre un proceso cliente (o 'llamador') y un proceso servidor (o 'llamado'):

1. El proceso cliente realiza una llamada a un procedimiento y se bloquea hasta recibir una respuesta.
2. Cuando el cliente recibe dicha respuesta, obtiene el resultado de su llamada.
3. El cliente continúa con su ejecución normal.

En el lado del servidor, se debe tener un proceso esperando el arribo de una solicitud. Cuando esto sucede, ejecuta lo necesario y genera una respuesta que enviará al cliente para luego volver a quedar a la espera.

Cada procedimiento RPC es definido unívocamente mediante un número de programa y uno de procedimiento. El primero especifica un grupo de procedimiento relacionados, cada uno de los cuales tiene un número de procedimiento diferente. A su vez, cada programa tiene además un número de versión; de modo tal de que cuando se realiza un cambio al servicio remoto (por ejemplo, se agrega u optimiza un procedimiento), no es necesario generar un nuevo número de programa.
Entonces, cuando se quiere hacer uso de un servicio remoto debemos conocer estos valores (número de programa, procedimiento y versión) para identificarlo [22].

2.2 Portmapper

Todo proceso cliente debe conocer la ubicación del servicio (procedimiento) remoto que quiere

utilizar. Esto, a niveles técnicos, es conocer la dirección de la máquina donde reside el proceso servidor y el número de puerto asociado del mismo; tarea a la que se denomina *binding*.
Existen diferentes enfoques para llevarla a cabo. Estáticamente se podría pensar en embeber en el código del cliente la ubicación del servicio que necesita consumir, lo que implicaría tener que volver a compilar el código del cliente cada vez que dicha dirección cambie. Un enfoque más dinámico puede ser utilizar una base de datos con la información necesaria de manera que la misma pueda ser consultada sin necesidad de recompilar. Entre estos dos extremos existen diferentes variaciones en cuanto a la manera en que se brinda un servicio de *binding*.
Este debate de diseño tiene su fundamento ya que la red no nos proporciona dicha funcionalidad: su comunicación se limita a la transferencia de mensajes entre procesos que residen en máquinas físicas (PCs), tarea para lo cual hacen uso de las direcciones IP y los números de puertos. Un puerto es un canal de comunicación lógico entre procesos a través de los cuales estos son capaces de recibir y enviar mensajes desde y hacia la red.
Cómo un proceso es capaz de esperar hasta la llegada de un mensaje hacia un puerto determinado varía de un sistema operativo a otro; pero todos brindan la posibilidad de suspender un proceso hasta la llegada de un mensaje que lo reactive.
Según se afirma en [22], Sun RPC realiza *binding* dinámicamente a través de un servicio de red llamado Portmapper.
Este programa mapea un programa RPC (identificado por su número de programa y su número de versión) a un número de puerto específico. El mismo se ejecuta en un puerto reservado al cual cada cliente puede comunicarse para obtener la ubicación de su servicio remoto. Es obligación entonces para el servidor que brinda el servicio RPC estar ejecutando el servicio de Portmapper. El Portmapper brinda soporte para TCP y UDP y corre en el puerto 111 para ambos.

La interacción entonces entre un programa RPC y el Portmapper ocurre de la siguiente manera:

1. Luego de que el administrador del sistema en el servidor inicia el Portmapper, este queda escuchando por solicitudes TCP o UDP en el puerto 111.
2. Cuando se inicia un programa RPC en el servidor, este se registra a sí mismo al Portmapper, quien mantiene una tabla de asociación de puertos a programas. La función *main* del proceso servidor (la cual es parte del *stub* como veremos más adelante), hace un llamado a *svc_create.* Esta función determina los protocolos de red soportados por el servidor, crea un socket para cada uno y enlaza estos a un puerto aleatorio. Luego de esto el proceso servidor se comunica con el Portmapper solicitándole que relacione su número de programa y versión con el número de puerto asignado (para cada protocolo).
3. Para acceder a un servicio específico, el cliente RPC envía un mensaje al Portmapper en el servidor especificando el número de programa y versión requeridos (cabe destacar que el cliente necesita conocer la IP de la PC en la que el servicio está corriendo). Para esto hace un llamado a *clnt_create* cuyos argumentos son la IP o nombre del servidor, el número de programa, el número de versión y el protocolo utilizado. Una solicitud RPC es enviada al Portmapper del servidor, generalmente utilizando UDP como protocolo.

4. El Portmapper examina su tabla de asociaciones en busca del servicio especificado. Si el mismo se encuentra registrado, retorna al cliente su número de puerto asociado.
5. El cliente RPC almacena y utiliza el número de puerto recibido en las llamadas posteriores.

Esta secuencia puede verse en la Figura 2.1.

Sun RPC brinda un conjunto de rutinas a través de las cuales es posible interactuar con el Portmapper. En los sistemas UNIX se puede consultar la tabla de asociación a través del comando *rpcinfo*.

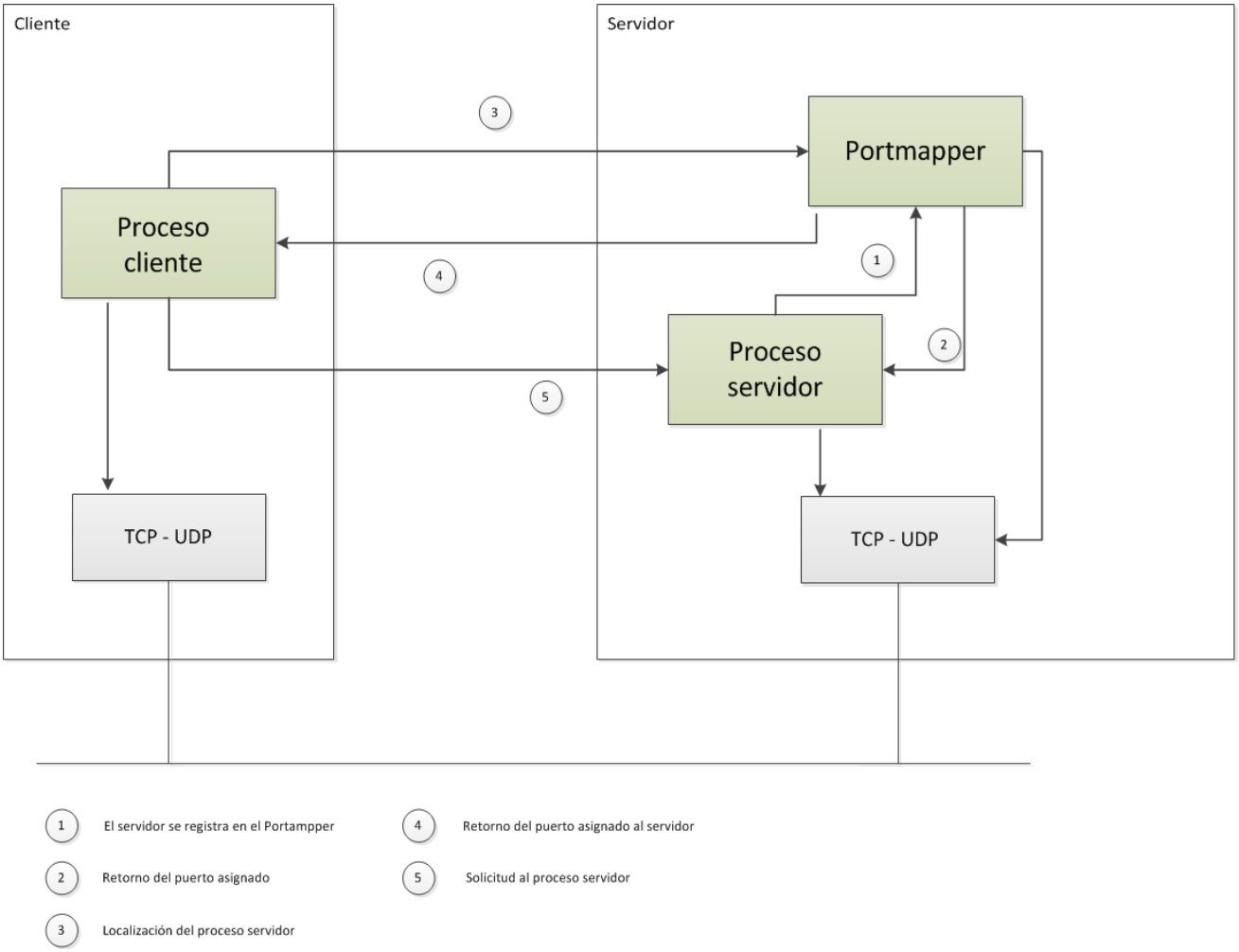


Figura 2.1. Interacción entre ONC RPC y Portmapper

2.3 Relación con el protocolo de transporte

Sun RPC permite trabajar con TCP y UDP. Es independiente del protocolo de transporte utilizado ya que no necesita ningún tipo de información sobre cómo se intercambian los mensajes entre los procesos para funcionar. Es por esto que la confiabilidad de sus aplicaciones queda ligada a la que implementa la capa de transporte. Estas deben tener en cuenta dicho protocolo ya que la elección de uno u otro puede indirectamente inferir en su funcionamiento (por ejemplo, en el tamaño de los mensajes que puede enviar) [18]. Es claro que si el protocolo elegido es TCP, al ser confiable, la mayoría del trabajo está hecho. Para con UDP, en cambio, al ser no confiable la aplicación debe implementar políticas de tiempos de espera, retransmisiones y detección de duplicaciones ya que el transporte no lo brinda.

Otro aspecto de interés en la relación con el protocolo elegido es que en base a éste podemos inferir la semántica de las llamadas en algunos escenarios. Por ejemplo:

- Si estamos trabajando con UDP, la aplicación retransmite solicitudes debido a finalizaciones en los tiempos de espera y recibe una respuesta, se puede deducir que el procedimiento fue ejecutado al menos una vez.
- Si estamos trabajando con TCP y se recibe una respuesta, se puede deducir que el procedimiento fue ejecutado exactamente una vez.

Cabe aclarar que para TCP tiene sentido también el uso de tiempos de espera para los casos en los que el servidor cae.

La selección del protocolo dependerá del tipo de aplicación con la que estemos trabajando.
UDP, al ser no confiable, es una buena elección para aplicaciones con las siguientes características:

1. Los procedimientos son idempotentes, es decir, los mismos pueden ser ejecutados más de una vez y producen los mismos resultados sin efectos secundarios. Por ejemplo, leer un archivo es idempotente y no así la creación de uno.
2. El tamaño de los argumentos y los resultados es menor a 8KB, que es el máximo permitido de los paquetes UDP.
3. El servidor debe procesar una gran cantidad de solicitudes. Si este es el caso, utilizar UDP será ventajoso en cuanto a tiempo y espacio debido a que no necesita mantener ningún tipo de información de control y estados (que sí utiliza TCP para lograr confiabilidad).

Contrariamente, algunas de las siguientes características en las aplicaciones nos inducen a elegir TCP:

1. La aplicación necesita un transporte subyacente confiable.
2. Los procedimientos no son idempotentes.

3. El tamaño, ya sea de los argumentos o del resultado, excede los 8KB.

2.4 XDR

Sun RPC puede operar adecuadamente con diversos tipos de datos y estructuras independientemente de las diferentes representaciones de los mismos que pueden encontrarse sobre las diversas plataformas. Esto es posible mediante la utilización de un estándar para la descripción y codificación de datos conocido como XDR (eXternal Data Representation).
XDR es un parte esencial de Sun RPC en el sentido de que los datos entre estos sistemas se transmiten utilizando este estándar. Funciona correctamente a través de diferentes lenguajes, sistemas operativos y arquitectura de computadoras.

XDR utiliza un lenguaje para describir formatos de datos. Es importante resaltar que XDR no es un lenguaje de programación, sino una especificación que incluye un lenguaje de descripción de datos el cual es extendido por Sun RPC para la definición de procedimientos remotos.
La tarea de convertir desde una representación particular al formato XDR se conoce como serialización; mientras que el proceso inverso se denomina deserialización.

El enfoque utilizado por XDR para estandarizar la representación de los datos es el canónico, es decir, adhiere a reglas predefinidas y establecidas. Cualquier programa ejecutándose en una máquina puede usar XDR para crear datos portables mediante la traducción de su representación local al estándar XDR; de la misma forma que inversamente un programa receptor puede traducir dichos datos a su representación interna [22].
XDR proporciona soporte para los tipos de datos listados a continuación:

- Enteros
- Enumerativos
- Booleanos
- Punto flotante
- Opaque
- Arreglos
- String
- Estructuras
- Uniones discriminadas
- Void
- Constantes
- Definición de tipos
- Datos opcionales (uniones particulares)

2.4.1 Relación del lenguaje RPC con XDR

Un IDL (Interface Definition Language) es un lenguaje utilizado para describir la interfaz de un componente de software. Éste no es de ejecución sino declarativo, es decir, dice lo que existe pero no cómo existe. Sun RPC emplea como IDL una ampliación del lenguaje XDR que permite la definición de procedimientos. Este lenguaje es conocido como RPCL (RPC Language) y es idéntico al lenguaje XDR, excepto en que agrega definición para el programa y su versión.
Al ser los tipos de datos de XDR descriptos en un lenguaje formal, también deben serlo los procedimientos que operan con dichos tipos. RPCL es una extensión de XDR utilizada para este propósito: especifica los tipos de datos utilizados por RPC y genera las rutinas XDR necesarias para estandarizar su representación [19]. Para esto se utiliza un compilador llamado rpcgen que genera el código C asociado, del que hablaremos en el Capítulo 3.

2.4.2 Ejemplificación XDR

Debido a la similitud entre ambas sintaxis, suele confundirse el código escrito en XDR con el lenguaje C. En el Apéndice A se describe la relación entre ambos lenguajes, con ejemplos y descripciones sobre cómo rpcgen convierte las definiciones XDR en lenguaje C [22].
Más allá de la unificación en la representación de datos, otro punto importante en XDR es su sencillez: con éste se pueden especificar y enviar a través de una red estructuras de datos complejas que implicarían un esfuerzo mayor al desarrollador si tuviese que ocuparse por sí mismo. Por ejemplo:

- Estructuras sin límites fijos, como los strings, se definen como:

```
string text<>;
```

A diferencia de C, en XDR existe un tipo *string*, ya que de esta manera no presentan la ambigüedad que tienen en C: los programadores generalmente, al declarar char *, quieren indicar que se trata de una cadena de caracteres finalizada en nulo, pero esta puede también representar un puntero a un simple caracter o a un arreglo de caracteres.

- Una estructura dinámica como una lista de enteros se especifica de manera intuitiva:

```
struct list {
   int value;
   list *next;
     };
```

Luego, dicha lista podrá ser utilizada como parámetro o resultado de un procedimiento remoto de la forma:

int print(list)

XDR se encargará de aplicar las reglas necesarias para que estos datos, a pesar de que usan punteros que por definición solo tienen sentido en el sistema local, lleguen a la máquina remota como en su representación original.

2.5 Conceptos de interés

2.5.1 Tiempos de espera

Sun RPC utiliza una estrategia de tiempos de espera (*timeout*) y retransmisiones necesarios para controlar fallas en la comunicación remota [19].
Existen 2 tipos de *timeout*:

1. El *timeout* total, que es la cantidad de tiempo que un cliente espera por una respuesta del servidor. Este valor es utilizado tanto en TCP como UDP y su valor por defecto es de 25 segundos. Si un cliente no recibe respuesta luego de ese plazo significa que se cumple alguna de las siguientes condiciones:

- El servidor no se encuentra activo.
- El sistema remoto falló.
- El destino es inalcanzable.

2. El *timeout* de retransmisión, que es utilizado solamente por UDP y como su nombre indica es la cantidad de tiempo entre retransmisiones. Su valor por defecto es de 5 segundos.

No existe necesidad de *timeout* de retransmisión con TCP ya que éste es confiable. Supongamos que el servidor nunca recibe la solicitud de un cliente TCP, éste agota su timeout total y vuelve a solicitar. Cuando el servidor recibe la solicitud, acusa su recepción al cliente. Llegado el caso en que este acuse se pierda, el cliente volverá a solicitar pero el servidor será capaz de reconocer dicho escenario; descartando el paquete y volviendo a informar el recibo anterior. Con TCP entonces la confiabilidad (tiempos, retransmisiones, manejo de duplicados e información de recepción) es provista por la capa de transporte y no es necesario que Sun RPC se encargue. Como mencionamos anteriormente, esto no sucede con UDP.

Algunas veces es necesario modificar el valor por defecto, por ejemplo si el servidor es lento o la operación necesaria se sabe va a exceder dicho límite. Esto puede hacerse utilizando la función *clnt_control:.*

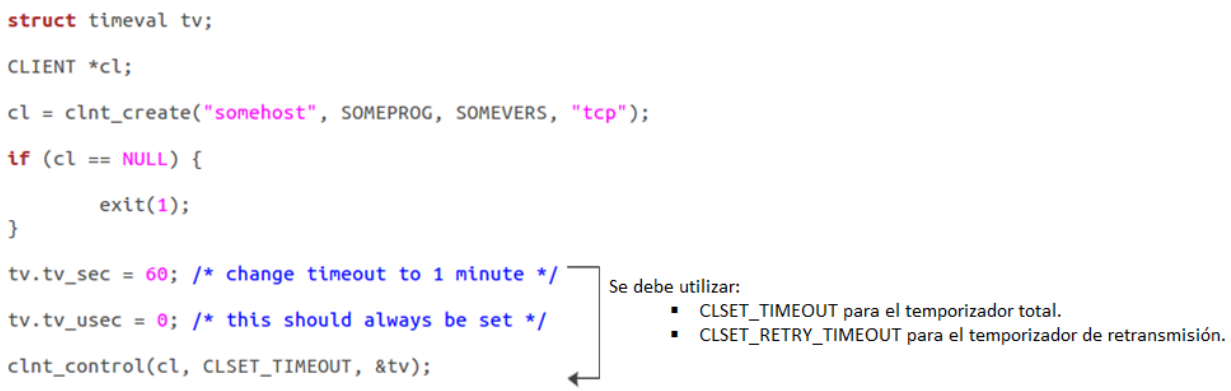


Figura 2.2. Modificación de timeout

2.5.2 ID de transacción

Otro punto de interés y del que haremos uso a futuro es el ID de transacción (XID), que como refiere su nombre es un número utilizado para identificar llamadas remotas. Cuando un cliente realiza una llamada, Sun RPC adhiere un entero de 32 bits al mensaje a modo de identificación. La respuesta del servidor, en caso de que la haya, deberá tener el mismo XID que su solicitud asociada para que el cliente pueda procesarla [19].
Este dato no se modifica cuando se retransmiten solicitudes, por lo que podemos entender cuál es su propósito:

1. El cliente verifica que el XID de la respuesta coincida con el que fue enviado en la solicitud, ignorando esta en el caso contrario. Si TCP está siendo utilizado, el cliente no debería recibir una respuesta con un XID diferente al enviado; pero con UDP, y la posibilidad de retransmisiones y pérdidas en la red, recibir una respuesta con un XID incorrecto es definitivamente una posibilidad válida.
2. El XID es un indicador válido para detectar solicitudes duplicadas y se podría utilizar en el servidor para evitar las mismas. Tal es el caso de la versión para Solaris, que hace uso de una caché de respuestas. Esto es, cada vez que responde una solicitud almacena y asocia su resultado a su XID. Cuando una nueva ingresa, si su XID se encuentra registrado en la caché quiere decir que se trata de una duplicación; por lo que en lugar de ejecutar nuevamente, vuelve a enviar el resultado al cliente [25].

Lamentablemente, en las versiones de Sun RPC para Linux no se cuenta con esta opción.

3. El programa rpcgen

En este capítulo se presenta y analiza el funcionamiento del compilador rpcgen. Se estudia la estructura de cada uno de los archivos que se relacionan con éste para tratar de entender en detalle el circuito de la comunicación remota utilizando RPC.
Finalmente, se ejemplifica la secuencia de pasos necesarios para convertir un programa de ejecución local en uno que hace uso de un procedimiento remoto.

3.1 Introducción

Los detalles en la programación de aplicaciones que utilizan llamadas a procedimientos remotos pueden ser abrumadores e incluso hasta desmotivadores: imaginemos tener que escribir las rutinas XDR necesarias para convertir nuestros parámetros y resultados en un formato de red estándar y viceversa.
Afortunadamente, existe rpcgen. Éste es un compilador que facilita las tareas de los programadores de aplicaciones RPC realizando el trabajo engorroso y permitiendo focalizarse solamente en el funcionamiento propio de su aplicación.
El programa rpcgen acepta la definición de la interfaz de un procedimiento remoto escrito en un lenguaje particular de RPC (RPCL) similar a C. En base a este archivo de definición genera diferentes archivos con código C, los cuales podemos clasificar en: un representante en el cliente y otro en el servidor que brindan transparencia en la comunicación conocidos como *stubs*; un archivo para la serialización de datos a XDR y un último de cabecera (.h) para definiciones.
El *stub* trabaja conjuntamente con la librería de RPC a nivel interno y efectivamente oculta la red subyacente a los procesos. Estos archivos generados pueden ser compilados normalmente, sin necesidad de pasos intermedios: el programador escribe un procedimiento servidor particular y enlaza éste al *stub* servidor generado, obteniendo un programa ejecutable. Para hacer uso de éste, el mismo programador (u otro) necesita escribir un simple programa que haga llamadas locales a las funciones del *stub* cliente y enlazarlos para generar un cliente ejecutable.

El compilador brinda diversas opciones para generar y suprimir archivos puntales (por ejemplo, aquellos que contienen los procedimientos funcionales). Existen varios parámetros que facilitan y condicionan el desarrollo. Estos varían en las diferentes versiones del compilador en base al sistema operativo. Para Linux, algunos de consideración son:

- -l: genera el stub del cliente.
- -m: genera el stub del servidor.
- -a: unión de las opciones -l y -m.
- -M: genera código seguro para ambientes de multithreading (MT - Safe).
- -N: permite el pasaje de múltiples parámetros.

Como todos los compiladores, rpcgen reduce el tiempo de desarrollo que de otro modo se perdería codificando y probando las rutinas de bajo nivel; procesamiento que conlleva un costo mínimo de eficiencia y flexibilidad. Sin embargo, varios compiladores permiten a los programadores recurrir a 'soluciones de emergencia' para mezclar código de alto y bajo nivel si su aplicación requiere alguna forma de optimización y rpcgen no es la excepción: rutinas particulares pueden ser asociadas a la salida del compilador sin ningún inconveniente; por lo que se podría empezar un desarrollo tomando los archivos de rpcgen como punto de partida y reescribir lo que sea necesario [1] [22].

3.2 Análisis del stub cliente y servidor

Como mencionamos, ambos archivos son generados por el compilador a partir de la descripción de la interfaz del procedimiento remoto, por lo que sólo dependen de ésta.
El stub cliente puede interpretarse como un representante del servidor ya que es quien recibe la llamada del proceso cliente. Brinda transparencia actuando como nexo hacia el procedimiento remoto. Dicho de otra manera, el stub se utiliza de manera tal que el cliente realiza una llamada local a su procedimiento como si fuera el servidor real.
Entre las tareas de las que se encarga el stub cliente encontramos:

1. Localiza al servidor que implementa el procedimiento remoto usando un servicio de *binding*.
2. Empaqueta los parámetros de entrada (aplanado) en un formato común para el cliente y servidor.
3. Envía el mensaje resultante al servidor.
4. Espera la recepción del mensaje de respuesta.
5. Extrae el resultado (desaplanado) y lo devuelve al cliente que hizo la llamada.

El stub servidor, de manera inversa, puede interpretarse como un representante del cliente ya que es quien realiza la llamada al servidor. Conoce la interfaz ofrecida por el servicio remoto y realiza llamadas locales a este como si fuera el cliente real. Es el responsable entonces de la invocación real del procedimiento remoto.
Entre las tareas de las que se encarga el stub servidor encontramos:

1. Registra el procedimiento en el servicio de *binding*.
2. Ejecuta un bucle de espera de mensajes.
3. Recibe las peticiones.
4. Desempaqueta los mensajes (desaplanado).
5. Determina qué método concreto debe ser invocado.
6. Invoca al procedimiento con los argumentos recibidos y recibe el resultado de la respuesta.

7. Empaqueta el valor obtenido (aplanado) y envía el mensaje al stub del cliente.

3.3 Convirtiendo procedimientos locales en remotos

A partir de una aplicación escrita para ejecutarse de una forma local clásica, veremos cómo modificar su funcionamiento para que haga consumo de un servicio remoto usando rpcgen. El siguiente es un simple ejemplo de un programa escrito en C que imprime un mensaje por consola, sin más fines que los pedagógicos [26].

3.3.1 Modelo local

En el siguiente archivo se define una función main que recibe como parámetro una cadena de caracteres, la cual intenta almacenar en un archivo local llamado 'Register.txt' e informa el resultado de la operación. Llegado el caso en que el archivo referido no exista, éste será creado.

```
/* printmsg.c: */
/* Print a message on the console */
/* Return a boolean indicating whether the message was actually printed */

#include <stdio.h>

int printmessage(char *msg) {

        FILE *f;
        f = fopen("Register.txt", "a");

        if (f == NULL) {
                return (0);
        }

        fprintf(f, "%s\n", msg);
        fclose(f);
        return(1);
}

int main (int argc, char *argv[]) {

        char *message;

        if (argc != 2) {
                fprintf(stderr, "usage: %s <message>\n", argv[0]);
                exit(1);
        }

        message = argv[1];
        if (!printmessage(message)) {
                fprintf(stderr, "%s: couldn't print your message\n", argv[0]);
                exit(1);
        }
        printf("Message Delivered!\n");
        exit(0);
}
```

Figura 3.1. Archivo modelo local

3.3.2 Modelo remoto

Tomando como base la funcionalidad del ejemplo anterior (almacenar una cadena de caracteres en un archivo), replicaremos la misma pero esta vez la rutina encargada de realizar la escritura será invocada remotamente. Al ejecutarse en otra máquina, se deduce que el archivo que se genera o edita se encuentra en el sistema de archivos del servidor. Los pasos a seguir para realizar la conversión de local a remoto son los siguientes:

3.3.2.1 Especificación de la interfaz

Una vez definidos los parámetros necesarios para el procedimiento que quiere hacerse remoto, utilizaremos el lenguaje de RPC para definir su especificación. Para esta tarea debemos crear un archivo con extensión .x (necesaria para que el compilador pueda procesarlo); al cual llamaremos archivo de especificación. En éste se definen el nombre, la versión y las funciones del programa; asociándoles un identificador numérico:

- Desde 0x20000001 en adelante para el programa (aunque generalmente hay uno solo).
- Desde 1 en adelante para la versión (usualmente hay una sola).
- Desde 1 en adelante para funciones (pueden haber varias).

En el archivo de especificación de la Figura 3.2 se define la primera versión del programa 0x20000001, cuyo único procedimiento recibe un string y retorna un entero. Existe un procedimiento nulo (con identificador 0) implícito generado por rpcgen automáticamente al compilar que se utiliza para verificar que el servidor se encuentre disponible.

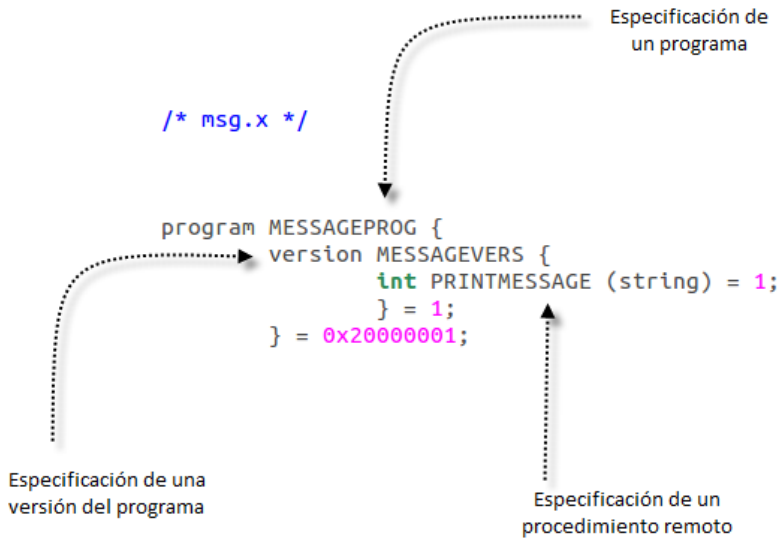


Figura 3.2. Archivo de especificación

3.3.2.2 Generación de archivos

Al compilar el archivo de especificación anterior con rpcgen y haciendo uso de la opción –a (para que genere todos los archivos), se obtienen:

- msg_clnt.c: es el stub del cliente, cuyo nombre se forma añadiendo el prefijo '_clnt' al nombre del archivo de especificación.
- msg_svc.c: es el stub del servidor, cuyo nombre se forma añadiendo el prefijo '_svc' al nombre del archivo de especificación.
- msg.h: es un archivo de cabecera con definiciones compartidas que debe ser incluido en el cliente y en el servidor. Su nombre se forma cambiándole la extensión al archivo de especificación por .h.
- msg_client.c: es la implementación del cliente. En éste se define una función *main* y una local que invoca al procedimiento remoto. Su nombre se forma añadiendo el prefijo '_client' al nombre del archivo de especificación.
- msg_server.c: es la implementación del servidor. En éste se define la función remota vacía que luego deberá ser implementada por el programador. Su nombre se forma añadiendo el prefijo '_server' al nombre del archivo de

especificación.

- Makefile.msg: es un archivo utilizado para compilar los anteriores. Su nombre es siempre 'Makefile' y tiene como extensión al nombre del archivo de especificación.

Por defecto los archivos generados por el compilador son los 2 stubs, el de cabecera y uno que define rutinas para XDR (que no siempre es necesario y del que hablaremos más adelante). Al añadirle el parámetro -a, rpcgen crea también los referentes a las implementaciones del cliente y servidor, junto con el Makefile.

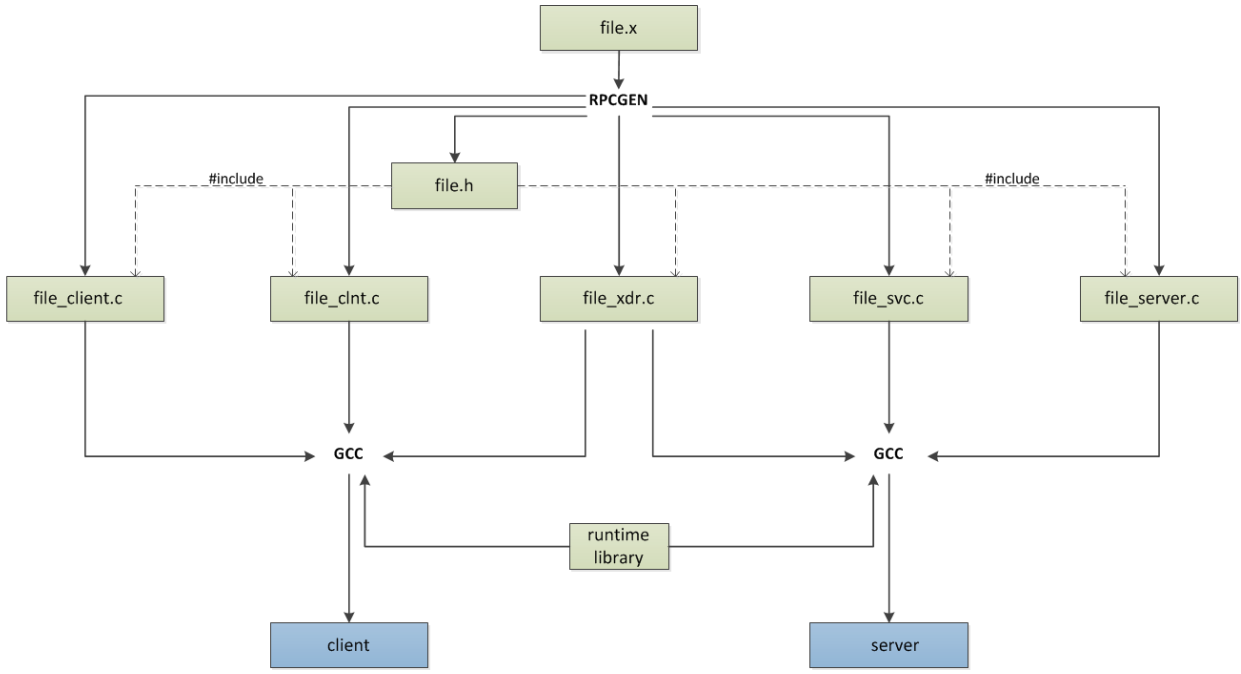


Figura 3.3. Archivos en base a rpcgen

3.3.2.3 Implementación del servidor (msg_server.c)

Este archivo tendrá tantas funciones como se hayan declarado en el archivo de especificación (.x). Todas las funciones recibirán como parámetros un puntero al tipo declarado y una estructura con datos de la solicitud. Su definición puede verse en la Figura 3.4.
Todas las funciones tendrán una variable local estática del mismo tipo al que apunta el valor de retorno. Se necesita que la variable local sea estática para que su valor no sea borrado antes de que el stub servidor llegue a enviar la respuesta al cliente.

El nombre de las funciones se genera con:

- El nombre declarado.
- El identificador numérico.
- El sufijo '_svc' (asociado a 'service').

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "msg.h"

int *
printmessage_1_svc(char **argp, struct svc_req *rqstp)
{
        static int  result;

        /*
         * insert server code here
         */

        return &result;
}
```

Figura 3.4. Implementación del servidor

La idea para con este archivo es entonces agregar código de manera tal de que se utilicen los parámetros y generar un resultado, en caso de ser necesario, para retornar al cliente.
Para nuestro ejemplo, deberíamos intentar guardar lo recibido en el archivo y regresar el resultado de la operación, como puede verse en la Figura 3.5.

```
int *
printmessage_1_svc(char **argp, struct svc_req *rqstp)
{
        static int  result;
        FILE *f;

        f = fopen("Register.txt", "a");
        if (f != NULL) {
                fprintf(f, "%s\n",  *argp );
                result = 1;
                fclose(f);
        }
        else
                result = 0;

        return &result;
}
```

Figura 3.5. Implementación del servidor con funcionalidad

3.3.2.4 Implementación del cliente (msg_client.c)

Desglosaremos el mismo para facilitar su comprensión, indicando en primer lugar su estructura para luego especificar cada una de sus funciones.
En cuanto a su estructura, podemos dividirlo en:

1. Una función main.
2. Una función encargada de invocar al procedimiento remoto, por lo que desde ahora en adelante nos referiremos a ella como función de invocación.

```c
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "msg.h"


void
messageprog_1(char *host)
{

        /* Some code here */

}


int
main (int argc, char *argv[])
{

        /* Some code here */

}
```

Figura 3.6. Estructura del archivo cliente

La función main se encarga de:

1. Controlar de que haya un parámetro de línea de comandos que referencie al servidor (con su IP o nombre) y llamar a una función local con éste.

Figura 3.7. Función main del archivo cliente

La función de invocación se encarga de:

1. Preparativos previos necesarios para las llamadas remotas independientemente de lo que esté especificado en el archivo .x. Aquí se establece la conexión con el servidor haciendo uso del nombre (o IP) ingresado, los números de programa y versión a invocar y el protocolo de transporte (UDP por defecto). Se hace uso de una estructura específica de RPC (CLIENT), conocida como manejador del cliente, que funciona como nexo entre el procedimiento del cliente y su stub; en cuya definición no vale la pena profundizar.
2. Una llamada para cada función definida en el archivo .x, donde se obtiene el resultado y se valida ante fallos.
3. Eliminación de los recursos utilizados. Aquí se finaliza la conexión establecida y es independiente de los que esté especificado en el archivo .x.

El nombre de la función de invocación se genera con:

- El nombre del programa.
- El identificador numérico de la versión del programa.


```
void messageprog_1(char *host)
{
        CLIENT *clnt;
        int  *result_1;
        char * printmessage_1_arg;

#ifndef DEBUG
        clnt = clnt_create (host, MESSAGEPROG, MESSAGEVERS, "udp");
        if (clnt == NULL) {
                clnt_pcreateerror (host);
                exit (1);
        }
#endif  /* DEBUG */
        result_1 = printmessage_1(&printmessage_1_arg, clnt);
        if (result_1 == (int *) NULL) {
                clnt_perror (clnt, "call failed");
        }
#ifndef DEBUG
        clnt_destroy (clnt);
#endif   /* DEBUG */
}
```

Figura 3.8. Función local del archivo cliente

La idea para con este archivo es entonces agregar código de manera tal de que se utilicen los parámetros y el valor de retorno. Para el ejemplo, debería recibirse y enviarse el valor a escribir en el archivo remoto.

3.3.2.5 Stub del cliente (msg_clnt.c)

Presenta un funcionamiento estándar que generalmente no debe ser modificado por el programador y tendrá tantas funciones como se hayan declarado en el archivo de especificación. Estas funciones son las invocadas desde el cliente, funcionando como nexo entre el procedimiento local y el remoto; y todas utilizarán una rutina de la librería RPC llamada *clnt_call* para la comunicación, haciendo uso del manejador, el identificador del procedimiento que se quiere invocar y variables para el argumento y resultado junto con sus rutinas de serialización.

El nombre de las funciones se genera con:

- El nombre declarado.
- El identificador numérico.

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

#include <memory.h> /* for memset */
#include "msg.h"

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

int *
printmessage_1(char **argp, CLIENT *clnt)
{
        static int clnt_res;

        memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call (clnt, PRINTMESSAGE,
                (xdrproc_t) xdr_wrapstring, (caddr_t) argp,
                (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return (&clnt_res);
}
```

Figura 3.9. Stub del cliente

3.3.2.6 Stub del servidor (msg_svc.c)

Desglosaremos el mismo para facilitar su comprensión, indicando en primer lugar su estructura para luego especificar cada una de sus funciones.
En cuanto a su estructura, podemos dividirlo en:

1. Una función main.
2. Una función encargada de ejecutar el procedimiento de cada solicitud; por lo que desde ahora en adelante nos referiremos a ella como función de procesamiento.

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

#include "msg.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

static void
messageprog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
        /* Some code here */

}

int
main (int argc, char **argv)
{
        /* Some code here */

}
```

Figura 3.10. Estructura del stub del servidor

La función main se encarga de:

1. Crear un socket UDP y asociarlo al proceso servidor en el Portmapper.
2. Crear un socket TCP y asociarlo al proceso servidor en el Portmapper.
3. Llamar a *svc_run*, función que queda a la espera de la llegada de solicitudes.

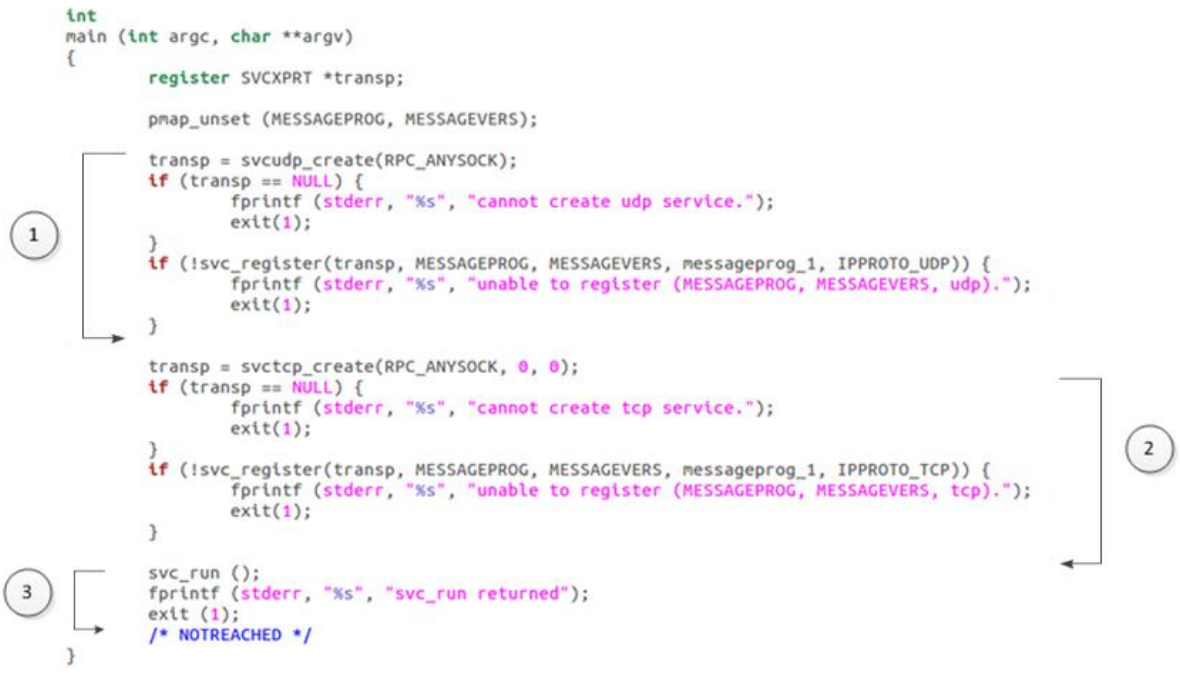


Figura 3.11. Función main del stub del servidor

Utiliza rutinas específicas de libc para la creación de sockets y una estructura denominada manejador del servicio de transporte (SVCXPRT) para mantener una referencia a los mismos.

La función de procesamiento, por otro lado, se encarga de:

1. Definir cuál es el procedimiento solicitado.
2. Extraer los parámetros e invocarlo
3. Retornar el resultado.
4. Liberar recursos.

Recibe un manejador del servicio de transporte a través del cual podrá retornar el resultado al cliente, obtener los parámetros y liberar recursos, entre otros; y una estructura que representa la solicitud en sí propia que indica cual es el procedimiento solicitado (aunque en nuestro ejemplo definimos uno solo, pueden existir varios). Podemos ver que dicha estructura es pasada al procedimiento que finalmente se ejecuta. Esto es así ya que en ella se encuentran datos que pueden ser necesarios de conocer, como por ejemplo para autenticar al cliente.

El nombre de la función de procesamiento se genera con:

- El nombre del programa.
- El identificador numérico de la versión del programa.

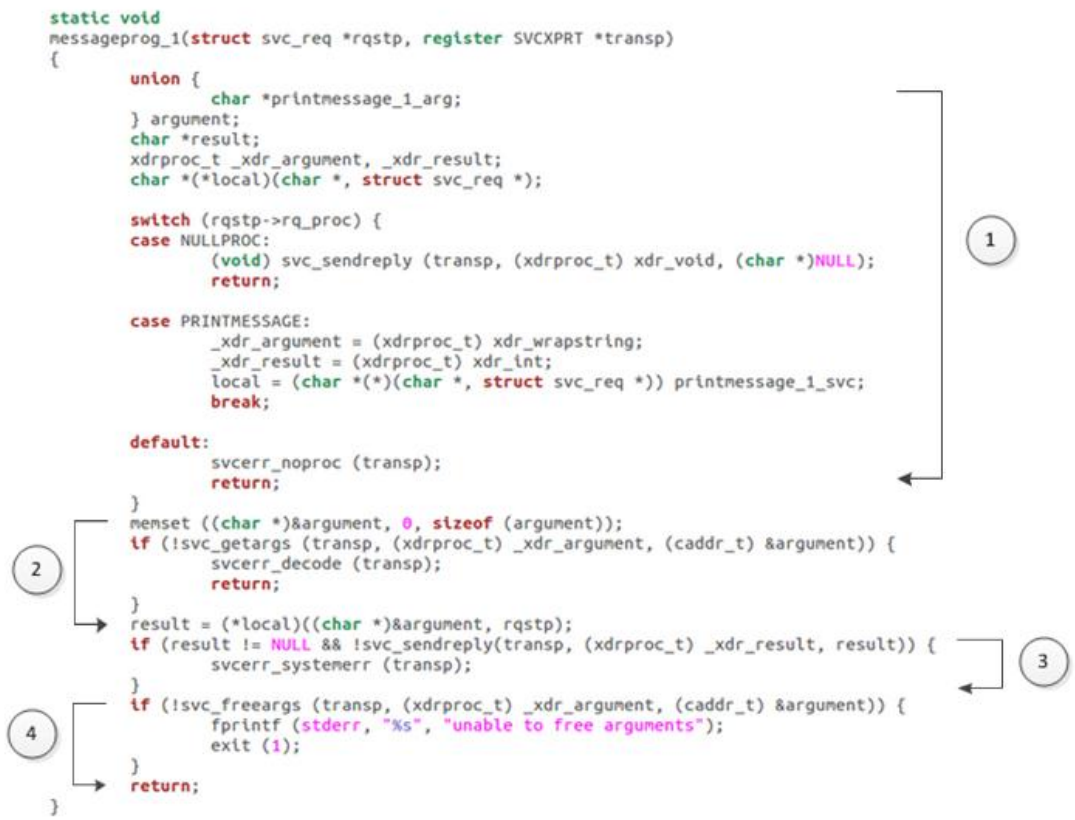


Figura 3.12. Función de procesamiento del stub del servidor

Como vimos, en este archivo se utilizan estructuras de datos cuya definición puede verse a continuación:

- Manejador del servicio de transporte (SVCXPRT).

```
/* Server side transport handle */

struct SVCXPRT {
  int xp_sock;
  u_short xp_port;               /* associated port number */
  const struct xp_ops {
    bool_t      (*xp_recv) (SVCXPRT *__xprt, struct rpc_msg *__msg);
                               /* receive incoming requests */
    enum xprt_stat (*xp_stat) (SVCXPRT *__xprt);
                               /* get transport status */
    bool_t      (*xp_getargs) (SVCXPRT *__xprt, xdrproc_t __xdr_args,
                               caddr_t args_ptr);
                               /* get arguments */
    bool_t      (*xp_reply) (SVCXPRT *__xprt, struct rpc_msg *__msg);
                               /* send reply */
    bool_t      (*xp_freeargs) (SVCXPRT *__xprt, xdrproc_t __xdr_args,
                                caddr_t args_ptr);
                               /* free mem allocated for args */
    void        (*xp_destroy) (SVCXPRT *__xprt);
                               /* destroy this struct */
  } *xp_ops;
  int           xp_addrlen;      /* length of remote address */
  struct sockaddr_in xp_raddr;   /* remote address */
  struct opaque_auth xp_verf;    /* raw response verifier */
  caddr_t               xp_p1;   /* private */
  caddr_t               xp_p2;   /* private */
  char          xp_pad [256];    /* padding, internal use */
};
```

Figura 3.13. Manejador del servicio de transporte (SVCXPRT)

- Representante de la solicitud (svc_req).

```
/* Service request */
struct svc_req {
  rpcprog_t rq_prog;            /* service program number */
  rpcvers_t rq_vers;            /* service protocol version */
  rpcproc_t rq_proc;            /* the desired procedure */
  struct opaque_auth rq_cred;   /* raw creds from the wire */
  caddr_t rq_clntcred;          /* read only cooked cred */
  SVCXPRT *rq_xprt;             /* associated transport */
};
```

Figura 3.14. Estructura svc_req

3.3.2.7 Compilación y ejecución

Generamos los ejecutables cliente y servidor mediante el comando *make* y el archivo Makefile creado en el paso anterior. De esta forma se obtienen *msg_server* y *msg_client*.

Ejecutamos cada archivo en su respectiva terminal teniendo en cuenta que:

- El servidor debe iniciar primero para evitar fallas de conexión.
- Debemos indicar el nombre del servidor al cliente.

Para este ejemplo, el tipo de dato utilizado es simple y existen las rutinas de conversión necesarias para serializar. Esto no siempre es así ya que a veces es necesario definir nuestros propios tipos de datos. Para esos casos, cuando rpcgen detecta una definición en el archivo inicial, genera un nuevo archivo (cuyo nombre se forma añadiendo el prefijo '_xdr' al nombre del archivo de especificación) que contiene las rutinas de conversión para los mismos.
Para cada tipo de dato presente en el archivo .x, rpcgen asume que la librería de XDR contiene una rutina con el nombre de ese tipo con el prefijo xdr_ (por ejemplo, xdr_int). Si el archivo de especificación define al dato, rpcgen genera la rutina asociada. Si el programa utiliza tipos de datos que no son los contemplados por XDR (es decir para los que existen rutinas) y tampoco se definen en el .x, el programador es quien debe especificar la rutina de conversión.

Como comentario final debemos decir que tradicionalmente una llamada a procedimiento remoto acepta un único parámetro y devuelve un único resultado. Para poder pasar diferentes valores a un procedimiento remoto es necesario agrupar éstos en una estructura y pasar esta como parámetro. Si se quieren recoger varios valores de retorno, también es necesario agruparlos en una estructura y devolver la misma.
Existe una solución a este detalle provista por la opción -N, un parámetro al compilador que posibilita múltiples argumentos en las llamadas.

4. Concurrencia en Sun RPC y rpcgen

En este capítulo se plantea la falta de concurrencia en el lado del servidor de las implementaciones de Sun RPC para Linux. Se analizan además las características principales de procesos y threads, con el fin de poder comprender las ventajas y desventajas de aplicar concurrencia con uno u otro.
Finalmente, se presenta la implementación de Sun RPC para el sistema operativo Solaris como caso de estudio y punto de partida para alcanzar el desarrollo de un servidor RPC concurrente en Linux.

4.1 Procesamiento secuencial

La concurrencia en Sun RPC implementada en Linux es nula. Aunque sea difícil pensar un modelo cliente/servidor con procesamiento secuencial, este es el caso de las versiones para Linux. En un servidor de estas características, su funcionamiento general puede pensarse como:

```
while (true) {
        recibir solicitud
        realizar acción
        enviar respuesta
}
```

Dicho funcionamiento podemos verlo gráficamente en la Figura 4.1, donde se toma el ejemplo del capítulo anterior:

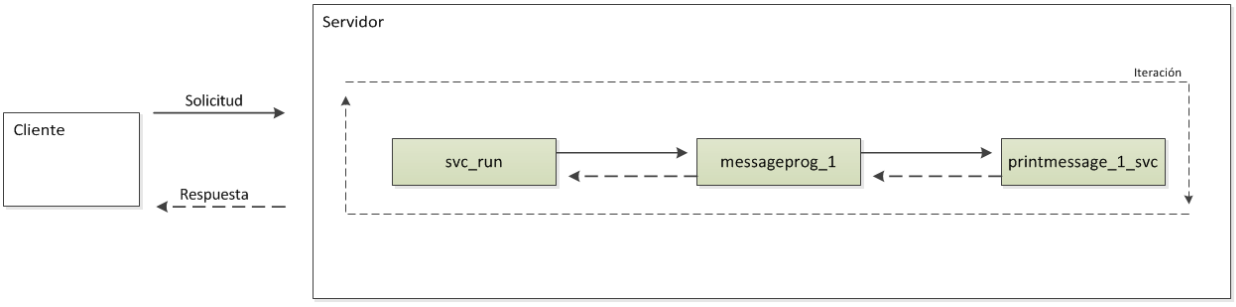


Figura 4.1. Procesamiento secuencial RPC

La ventaja de este enfoque es su simplicidad. El programador no debe preocuparse por el funcionamiento del servidor ya que es el modelo estándar elegido por Sun.
El problema fundamental de este modelo es que los clientes son encolados por orden de llegada y sus solicitudes no son atendidas hasta no finalizar las predecesoras. Pensemos por ejemplo en un cliente A que llama a un procedimiento remoto cuyo tiempo de procesamiento es de 1 minuto. Instantes después, un segundo cliente B solicita al mismo servidor un procedimiento que sólo necesita 1 segundo para finalizar. A pesar de que la diferencia de tiempo es excesivamente considerable, el cliente B deberá esperar el minuto necesario hasta que A finalice, con los inconvenientes que esto conlleva: B retrasa su procesamiento considerablemente; su timeout puede finalizar si es menor a 1 minuto, problema que tenderá a repetirse en estos escenarios ya que la configuración del timeout es exclusivo de cada cliente y éstos no pueden saber a priori el nivel de congestión del servidor. A esto sumémosle que, si el cliente corre sobre UDP y nuestro servidor no filtra duplicaciones, ante cada finalización de timeout B retransmitirá (pensando que la solicitud anterior no llegó) y cada una será encolada por el servidor. Pasado un tiempo, estas duplicaciones serán procesadas y si nuestro procedimiento no es idempotente el resultado final será muy diferente al esperado.

Otra desventaja propia de no aplicar concurrencia es el tiempo de procesamiento que puede llegar a desperdiciarse. Esto es: si el procedimiento encargado de la solicitud de A realiza frecuentes operaciones de entrada/salida o queda a la espera de algún evento para continuar, el procesador permanecerá ocioso y a la espera de poder continuar; cuando en realidad podría estar trabajando con otra solicitud sin problemas.

Planteada la falta de concurrencia en el servidor procederemos a explicar las diferentes soluciones analizadas para incorporar este tipo de procesamiento. Comenzaremos tratando la concurrencia con procesos y luego con threads; donde la elección de una u otra por parte del programador dependerá de las características de la aplicación que se esté intentando modelar.

4.2 Procesos y Threads

Para entender el rol de los threads en los sistemas distribuidos, es importante primero entender qué es un proceso y como se relacionan éstos con los threads. Un proceso puede definirse como un programa en ejecución. Formalmente un proceso es una unidad de actividad que se caracteriza por la ejecución de una secuencia de instrucciones, un estado actual y un conjunto de recursos del sistema asociados.
Los procesos son creados, gestionados y eliminados por el sistema operativo y están formados por:

- Las instrucciones de un programa destinadas a ser ejecutadas por el microprocesador.
- Su estado de ejecución en un momento dado, es decir, los valores de los

registros de la CPU para dicho programa.
- Su memoria de trabajo.
- Información extra que permite al sistema operativo su planificación.

El sistema operativo es el encargado de asegurar la independencia de los procesos. Esto implica que un proceso no pueda afectar el correcto funcionamiento de otro. En otras palabras, asegurar la transparencia entre varios procesos que comparten recursos.
Esta transparencia en la concurrencia no se obtiene a un bajo precio. Por ejemplo, cada vez que un proceso es creado el sistema operativo debe crear un espacio de direcciones independiente, inicializar segmentos de memoria y copiar el programa asociado, entre otras operaciones con un alto costo de procesamiento. Además, la conmutación entre procesos (dejar de ejecutar un proceso para comenzar a ejecutar otro) implica una serie de operaciones costosas; como lo son guardar el contexto del proceso (valores de registros, contador de programa, puntero de la pila, etc.) y modificar la unidad de gestión de memoria. Por otro lado, si el sistema operativo permite gestionar más procesos de los que la memoria principal puede abarcar, deberá virtualizar utilizando almacenamiento físico, incrementando el tiempo de respuesta considerablemente [27].

Un hilo (desde ahora en adelante thread) es la unidad de procesamiento más pequeña que puede ser planificada por un sistema operativo. Dentro de un proceso puede haber varios threads de ejecución, por lo que éste podría estar haciendo varias tareas a la vez. Estos threads comparten una serie de recursos tales como el espacio de memoria, los archivos abiertos y permisos; técnica que permite simplificar el diseño de una aplicación que debe llevar a cabo diferentes funcionalidades simultáneamente.
El hecho de que los threads de un mismo proceso comparten recursos hace que cualquiera de ellos pueda modificarlos. Cada thread tiene su propio contador de programa, pila de ejecución y estado de CPU.
Cabe aclarar que al tener una pila propia sus variables locales también lo son, siendo las globales aquellas que se comparten y con las que su uso debe ser cuidadoso. En la Figura 4.2 puede verse una representación de los diferentes espacios de memoria al usar threads.
El proceso sigue en ejecución mientras al menos uno de sus threads siga activo. Cuando el proceso finaliza, quiere decir que todos sus threads de ejecución también han terminado. Consecuentemente, el proceso deja de existir y todos sus recursos son liberados.

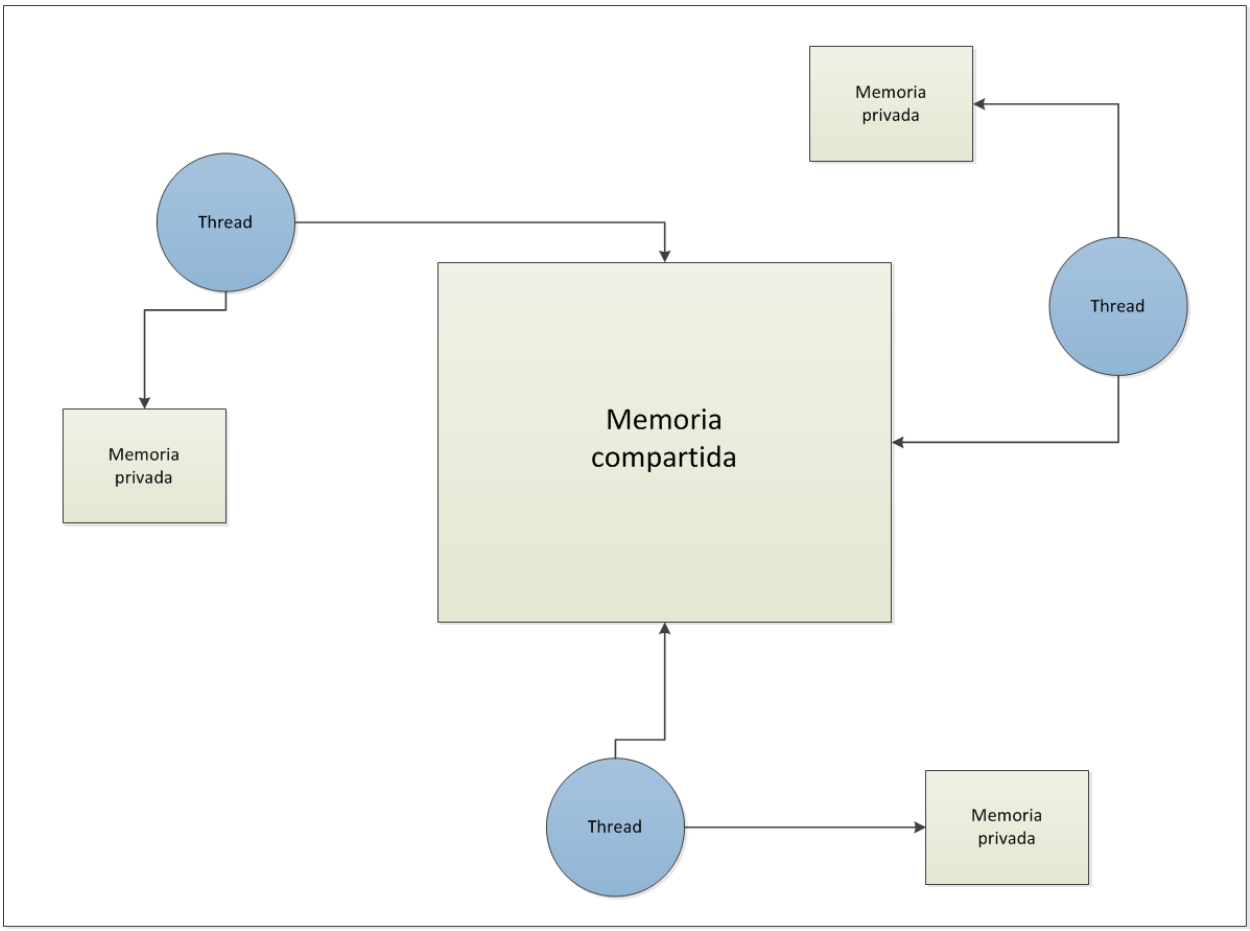


Figura 4.2. Memoria con multithreading

Pensando un sistema con threads en lugar de procesos, se puede mantener la concurrencia sin necesidad de degradar el rendimiento del sistema.
Los threads se distinguen de los procesos en que estos últimos son independientes e interactúan sólo a través de mecanismos de comunicación dados por el sistema. Los threads comparten recursos en forma directa coexistiendo con otros, por lo que su independencia debe considerarse con otro enfoque y requiere mayor esfuerzo intelectual del programador. Por ejemplo, para proteger variables compartidas es normal el uso de técnicas de exclusión mutua como los semáforos; aunque se debe tener cuidado en su uso para evitar resultados no esperados, como interbloqueos.
Lo comentado anteriormente hace que sea más rápido cambiar de un thread a otro dentro del mismo proceso, que cambiar de un proceso a otro, ya que los threads comparten datos y espacio de direcciones mientras que los procesos no. Al cambiar de un proceso a otro el sistema operativo (mediante el *dispatcher*) genera lo que se conoce como *overhead*, que es tiempo desperdiciado por el procesador para realizar un cambio de contexto; en este caso pasar del estado de ejecución al estado de espera y colocar el nuevo proceso en ejecución. El

tiempo perdido en cambiar de un thread a otro, siempre que pertenezcan al mismo proceso, es casi despreciable [27].

Resumiendo, el uso de threads mejora el rendimiento del sistema debido a:

- Se tarda mucho menos tiempo en crear un thread en un proceso existente que en crear un proceso.
- Se tarda mucho menos en terminar un thread que un proceso.
- Se tarda mucho menos tiempo en cambiar entre 2 threads de un mismo proceso que de un proceso a otro.
- Los threads aumentan la eficiencia de la comunicación entre programas en ejecución. En la mayoría de los sistemas, en la comunicación entre procesos debe intervenir el núcleo para ofrecer protección de los recursos y realizar la comunicación misma. En cambio, entre threads pueden comunicarse entre sí sin necesidad de la intervención del núcleo. Por lo tanto, si hay una aplicación que debe implementarse como un conjunto de unidades de ejecución relacionadas, es más eficiente hacerlo con una colección de threads que con una colección de procesos separados.

Para aquellos programas que pueden sacar ventaja aplicando concurrencia, la decisión entre usar procesos o threads puede ser difícil. Algunas consideraciones a tener en cuenta son:

- Un thread puede perjudicar otros threads del mismo proceso ya que comparten el espacio de memoria virtual. Un proceso, en cambio, no puede hacer esto debido a que cada uno tiene una copia del espacio de memoria.
- Copiar memoria para un nuevo proceso agrega una sobrecarga adicional de trabajo mayor a la creación de un thread. Sin embargo, esta copia es llevada a cabo sólo cuando se quiere modificar, por lo que la sobrecarga será mínima si el proceso hijo sólo se encarga de leer.
- Los threads deberían ser usados por programas que necesitan paralelismo de grano fino. Por ejemplo, cuando un problema puede ser dividido en múltiples tareas similares. Los procesos deberían ser usados por programas que requieren un paralelismo grueso.
- Compartir datos entre threads es trivial. Para con procesos se deben utilizar mecanismos de comunicación interprocesos, lo que hace más engorrosa la comunicación aunque también menos propensa a errores de concurrencia.

4.3 Propuesta de Sun RPC e implementación en rpcgen

Según [1], la implementación de Sun RPC para Solaris brinda procesamiento para múltiples threads (en adelante MT, en referencia a *Multithreading*) a partir de la versión 2.4. En ella, incorpora funcionalidades que permiten a los desarrolladores crear servidores MT de 2 modos

posibles: Automático o Usuario.
Con el modo automático, el servidor crea automáticamente un nuevo thread ante cada solicitud. Dicho thread procesa la misma, envía una respuesta y finaliza. En el modo usuario, es el desarrollador quien decide cómo crear y administrar threads para procesar concurrentemente las solicitudes. El modo automático es mucho más fácil que el usuario, pero este último ofrece más flexibilidad al desarrollador.

Existen 2 funciones para brindar MT en el servidor: rpc_control() y svc_done().
rpc_control() es utilizada para setear el modo en que se quiere que el servidor funcione (usuario o automático); mientras que svc_done() se utiliza luego de cada procesamiento para que el servidor pueda liberar recursos si estamos en modo usuario.

El manejador del servidor (SVCXPRT) que comentamos en capítulos anteriores, es una estructura utilizada para obtener los parámetros y enviar el resultado de cada solicitud. De esta forma, no puede ser compartida libremente entre threads. Sin embargo, cuando un servidor Solaris está operando en cualquier modo MT genera una copia de la misma, haciendo posible operar concurrentemente.

En el modo automático, la librería de RPC hace todo el trabajo. El programador solo tiene que habilitar dicho modo invocando a rpc_control antes de llamar a svc_run.
En el modo usuario, la librería de RPC no interviene en la creación y administración de threads. Lo que sí se hace es generar una copia de estructuras de datos, como SVCXPRT, para que el programador aplique la lógica necesaria.
Existe un parámetro propio de Solaris para rpcgen (-A) que genera lo necesario para manejo de theads en modo automático.

En Linux, no tenemos las funcionalidades que brinda Solaris y es por eso que, como dijimos inicialmente, no permite concurrencia. Además, las funciones de Sun RPC del lado del servidor están definidas como no seguras para ambientes MT. Es por esto que se propone analizar la factibilidad de incorporar procesamiento concurrente del lado del servidor para Linux, utilizando procesos o threads y dejando al programador la elección de uno u otro.
La idea inicial es, conociendo los archivos generados por rpcgen (sobre todo los stubs), incorporar la lógica necesaria para interceptar las solicitudes, delegar su procesamiento y poder de estar forma ser capaces de recibir otras.
De encontrarse posible, se intentará automatizar dicha funcionalidad para tratar de incorporar al rpcgen de Linux el manejo automático para MT que permite la versión de Solaris.

5. Implementación de la propuesta

En este capítulo se analiza la posibilidad de incorporar concurrencia en un servidor RPC para la plataforma Linux, en base a los conceptos estudiados anteriormente.
Se comienza tratando de agregar concurrencia utilizando procesos a través de la funcionalidad *fork* para luego intentar aplicar el mismo esquema teórico con threads a través del uso de *pthreads*.
A medida que se avanza en el desarrollo de una posible solución teórica, se plantean las problemáticas encontradas y se intenta dar respuesta a cada una de ellas.

5.1 Concurrencia con procesos

En base al funcionamiento de Sun RPC comentado en capítulos anteriores se puede pensar en modificar el código estándar para permitir generar un proceso por cada solicitud; de forma tal de proporcionar concurrencia con procesos.

El funcionamiento genérico sería entonces:

```
while (true) {
        recibir solicitud
        crear proceso que la procese
}
```

El interrogante que surge entonces es ¿dónde creamos el proceso? Es lógico pensar que esto debe hacerse en el stub del servidor, que es el encargado de recibir e invocar al procedimiento. Dentro del mismo, el lugar donde se realiza esta acción es la rutina de procesamiento cuyo funcionamiento, recordemos, consistía en:

1. Identificar y seleccionar el procedimiento que invoca el cliente inicializando los filtros XDR adecuados.
2. Extraer el argumento del procedimiento.
3. Invocar al procedimiento con el parámetro recibido.
4. Enviar el resultado al cliente.
5. Retornar a *svc_run*.

En base a lo anterior, podemos pensar que en cualquiera de los pasos anteriores al punto 3 podemos crear un proceso hijo para delegar el procesamiento. Esta comparación en el funcionamiento puede verse en la Figura 5.1.

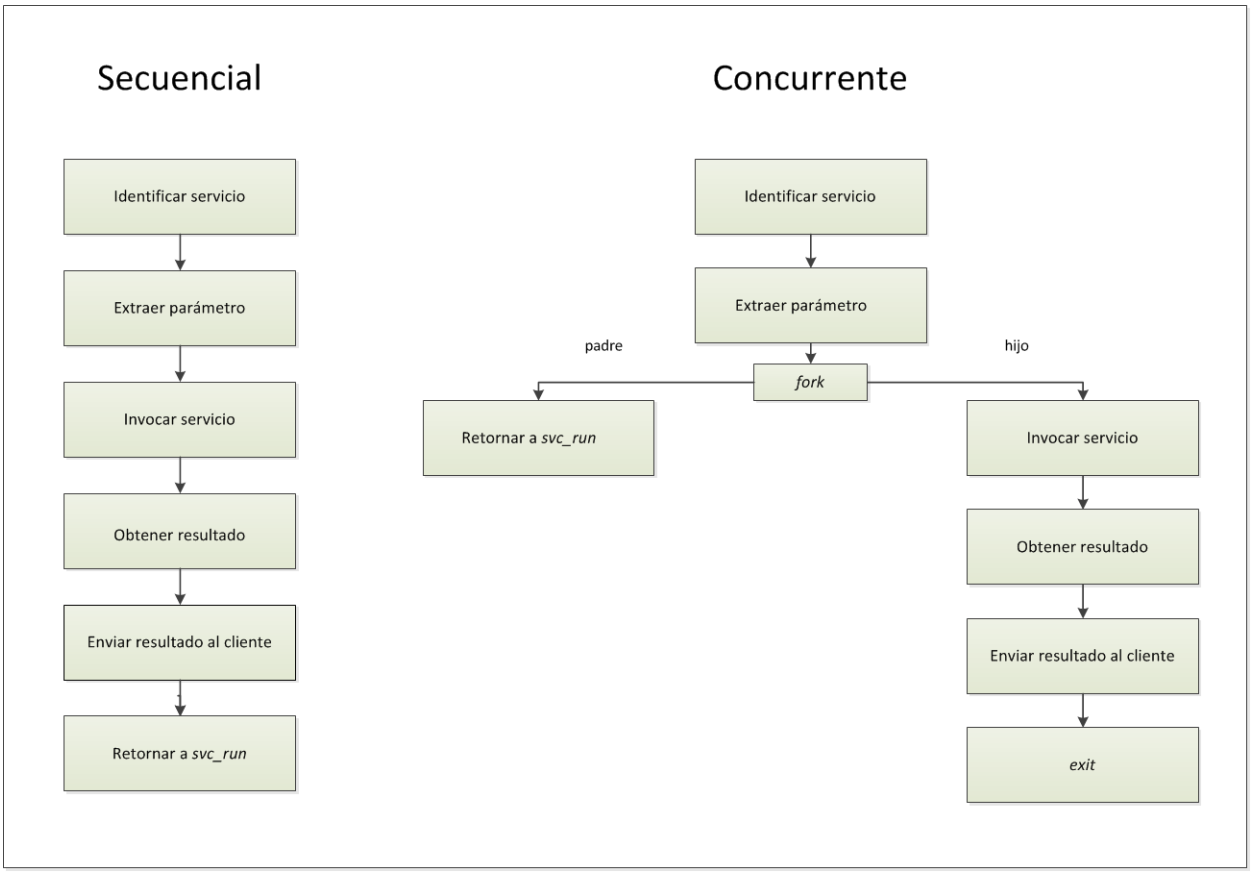


Figura 5.1. Comparación entre RPC secuencial y concurrente con procesos

5.1.1 Creación de procesos con fork()

fork es una llamada al sistema característica en los sistemas Unix que permite crear procesos. El proceso creado se denomina hijo en relación al proceso que realiza la llamada, conocido como padre. Cuando un proceso padre invoca a *fork*, crea un proceso y continúa con el flujo normal de procesamiento. En el proceso hijo, en cambio, se comienza a ejecutar desde el punto en que *fork* fue invocado.
La distinción al momento de la creación entre padre e hijo se hace evaluando el valor de retorno de la función:

- Si retorna un valor negativo, indica fallo.
- Si retorna un valor >= 0, indica éxito. Puede decirse que *fork* retorna el ID del proceso creado. Esto es verdadero cuando el proceso receptor es el padre, ya que el proceso hijo recibirá 0. De esta manera, luego de llamar a *fork* una simple

comparación puede decirnos qué proceso estamos ejecutando.

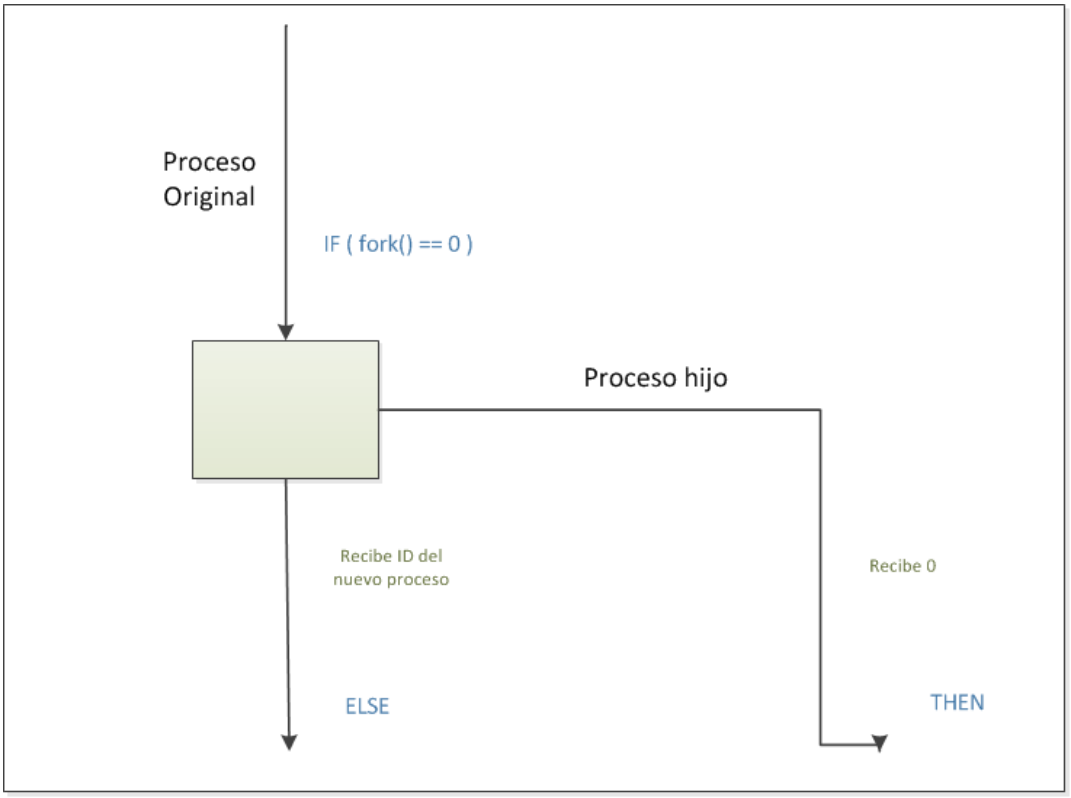


Figura 5.2. Flujo de ejecución de fork

fork duplica el espacio de memoria del proceso padre, por lo que cada uno contará con las mismas variables (en diferentes espacios de direcciones). También se mantienen las referencias a recursos, como por ejemplo archivos: si el padre tenía un archivo abierto antes de la creación, el hijo también contará con dicho acceso de forma independiente, es decir, su referencia será válida aunque el proceso padre decida cerrar el archivo [27].

Como en parte se ha comentado, esta operación es costosa. Algunos sistemas operativos, como es el caso de Linux, intentan reducir este costo a través de una semántica de *copy-on-write* (COW). Con ésta, cuando un proceso crea una copia de sí mismo, las páginas cargadas en memoria que puedan ser modificadas por dicho proceso o su copia se marcan como *copy-on-write*. Cuando un proceso modifica la memoria, el núcleo del sistema operativo interviene en la operación y crea una copia de forma que los cambios en la memoria ocupada por un proceso no sean visibles por el otro. En resumen, sólo se realiza la copia si alguno de los procesos necesita modificar la memoria [23].
Esta técnica se basa en el hecho de que generalmente los procesos hijos ejecutan un conjunto de acciones reducidas y finalizan sin modificar todas las estructuras del padre; por lo que éstas no son necesarias de replicar.

Por último, debemos comentar que un proceso puede esperar por el cambio de estado de alguno de sus hijos y obtener información sobre éste a través de llamadas al sistema. Un cambio de estado puede considerarse como:

- El hijo finalizó.
- El hijo se detuvo por una señal.
- El hijo se reanudó por una señal.

En el caso de que el hijo finalice, esto le permite al sistema liberar recursos asociados (de no ocurrir esto último el hijo permanecerá en estado *zombie*).

Entre dichas llamadas encontramos *wait* y *waitpid*.

- pid_t wait(int ***status*);**
- pid_t waitpid(pid_t *pid*, int ***status*, int *options*);**

La diferencia entre una y otra es que la primera queda a la espera de cualquier proceso hijo mientras que en la segunda referenciamos a uno en particular a través de su ID [23].

5.1.2 Aplicación de concurrencia con procesos en ejemplo inicial

Tomaremos a modo de ejemplo al servidor de la sección 3.3.2 que escribía un mensaje en un archivo, cuyo stub se encuentra especificado en la Figura 3.10, que se incluye aquí por comodidad:

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

#include "msg.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

static void
messageprog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
        /* Some code here */

}

int
main (int argc, char **argv)
{
        /* Some code here */

}
```

Recordemos que la función *main* registraba al servidor y quedaba a la espera de solicitudes vía *svc_run*, por lo que no existe necesidad de modificarla para agregar concurrencia.
Pensado en interceptar la solicitud para crear un nuevo proceso, procederemos a:

1. Crear una función, que denominaremos *wrapper,* utilizando la firma de la que utiliza el stub. Es decir:

   static void messageprog_1 (struct svc_req *rqstp, register SVCXPRT *transp)

   Esta es la que invoca el servidor ante cada solicitud. Agregamos en ella entonces la lógica de creación del proceso que invocará al procedimiento remoto, como puede verse en la Figura 5.3.

```c
static void
messageprog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{

        pid_t forkpid;
        forkpid = fork();
        if (forkpid == -1) {
                fprintf(stderr,"%s","Error creating new process");
                exit(1);
        }

        if (forkpid == 0) {

                /* This code will be executed only for child process */
                execute(rqstp, transp);
                exit(0);
        }

        return;
}
```

Figura 5.3. Creación de proceso en wrapper

2. Crear la función encargada puntualmente de procesar la solicitud a la que llamaremos *execute*. Allí, análogamente a lo visto en Figura 3.12, se decidirá qué procedimiento es requerido, se extraerán los parámetros y se invocará al procedimiento; para luego retornar un resultado, liberar recursos y finalizar. Esta secuencia se representa en la Figura 5.4.

```c
static void execute(struct svc_req *rqstp, register SVCXPRT *transp)
{
        // Definir cuál fue el procedimiento solicitado

        // Extraer los parámetros e invocarlo

        // Retornar el resultado

        // Liberar recursos
}
```

Figura 5.4 Función execute

De esta manera, cuando el *wrapper* finaliza nuestro servidor queda compuesto conceptualmente por un proceso a la espera de una nueva solicitud y otro que procesará la anterior.

La solución anterior nos brinda concurrencia con procesos aunque no tiene en cuenta los procesos *zombies* que se generan. Procesos *zombies* se denominan aquellos que finalizaron su ejecución pero siguen manteniendo una entrada en la tabla de procesos, lo que permite al proceso que lo ha creado leer el estado de su salida. Esto se debe a que dicho proceso hijo no recibió una señal por parte de su padre informando que su vida útil ha terminado [23]. Técnicamente, el padre nunca ejecuta el *wait* necesario para liberar los recursos del hijo. Dichos procesos *zombies* no consumen RAM ni procesador, aunque no por esto deben dejar de considerarse ya que pueden disminuir el rendimiento del sistema.
Una solución posible sería evitarlos vía código invocando a *wait* en el padre. También pueden evitarse eliminando al proceso padre, en cuyo caso el hijo en lugar de *zombie* se denominará *huérfano* y pasará a depender del proceso *init*, quien continuamente está a la espera de cambios de estado en sus procesos hijos y podría liberar recursos.
En la solución planteada, si hiciéramos uso de *wait* en el proceso generador, nuestra concurrencia no tendría sentido ya que el padre estaría continuamente esperando que su hijo termine e imposibilitado de aceptar otra solicitud; aún peor, la opción de eliminar al padre dejaría al servidor fuera de servicio.

Podríamos pensar en algún proceso que temporalmente localice y elimine *zombies*, solución que no es posible ya que el uso de *wait* no puede aplicarse debido a que este proceso temporal no es el padre original; y llamadas al kernel tampoco pueden utilizarse porque no se puede 'matar algo que está muerto'.

Ante esta situación, la solución encontrada fue hacer uso de un tercer proceso. Sea A un proceso padre, B un proceso hijo de A y C un proceso hijo de B, podemos pensar en que:

- A aceptará una solicitud y la delegará a B, quedando a la espera de éste.
- B delegará la solicitud a C y finalizará su ejecución.
- A puede ahora utilizar *wait* para liberar los recursos de B y quedar a la espera de otra solicitud.
- C ejecutará la función solicitada y finalizará. En este momento su padre (B) ha dejado de existir, convirtiendo a C en un proceso *huérfano*. Pero, como comentamos inicialmente, el proceso *init* se encargará de liberar sus recursos.

A nivel de código, debemos modificar la función wrapper anterior tal como se muestra en la Figura 5.5.

```
static void
display_prg_1(struct svc_req *rqstp, register SVCXPRT *transp)
{

        pid_t forkpid;
        forkpid = fork();
        if (forkpid == -1)
                fprintf(stderr,"%s","Error creating the first process");
                exit(1);

        if (forkpid == 0) {

                /* This code will be executed only for the first child process */
                if ((forkpid = fork()) == -1) {
                        fprintf(stderr,"%s","Error creating the second process");
                        exit(1);
                }
                else {
                        if (forkpid == 0) {

                                /* This code will be executed only for the second child process */
                                execute(rqstp, transp);
                                exit(0);
                        }
                        exit(0);
                }

        }

        /* This code will be executed only for father process */
        if (waitpid(forkpid, NULL, 0) != forkpid)
                fprintf(stderr,"%s","Error waiting for child process");


        return;
}
```

Figura 5.5. Creación de un tercer proceso en wrapper

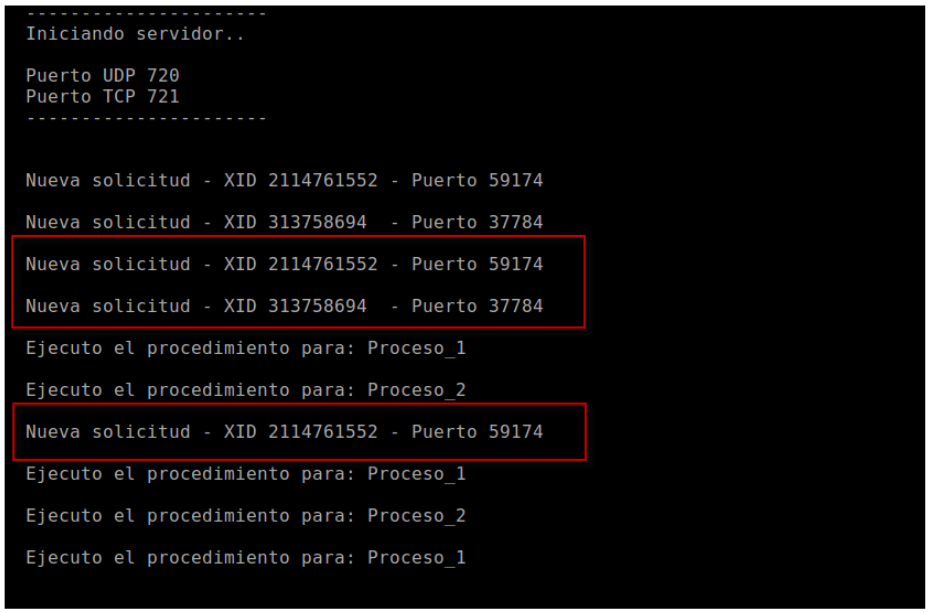
Se puede objetar que las prestaciones del servidor disminuirán debido a la nueva duplicación de recursos agregada. Aunque esto tiene su grado de verdad, el tiempo extra utilizado es preferible al incremento de *zombies* proporcional a las solicitudes que consecuentemente terminarán afectando al servidor de manera crítica.

5.1.3 Solicitudes duplicadas

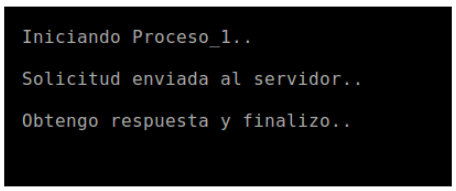
Durante las pruebas de validación a la solución anterior, se detectó que si se utiliza UDP como protocolo de transporte se procesan concurrentemente solicitudes duplicadas. Esto se debe a que como UDP no brinda confiabilidad, es decir, los datos pueden llegar o no, Sun RPC retransmite solicitudes cada un intervalo de tiempo si no recibe respuesta para asegurarse de que su solicitud llegue realmente. Estas retransmisiones se generan cada 5 segundos por defecto y debemos diferenciarlas del timeout global, con valor inicial de 25 segundos, el cual semánticamente controla que el servidor se encuentre funcional y no la llegada de una solicitud particular [22].
Con este escenario, si un procedimiento remoto demora una cantidad de tiempo mayor que el *timeout* de retransmisión del cliente, éste duplicará la solicitud y si el servidor no filtra las mismas, como es el caso de Sun RPC, podemos generar situaciones no deseadas si la

operación no es idempotente.
Este comportamiento podemos verlo en la Figura 5.6. Allí:

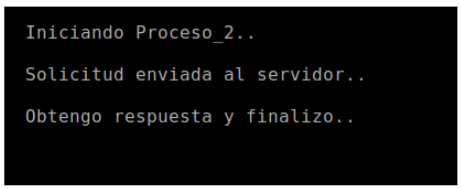
- El servidor recibe solicitud de 2 clientes.
- Cada cliente envía como parámetro la cadena '*Proceso_número*', donde *número* se corresponde con el número de cliente.
- El servidor al iniciar imprime los puertos asociados a TCP y UDP.
- El servidor ante cada solicitud imprime el puerto y XID del cliente antes de lanzar el proceso.
- El procesamiento de cada solicitud fue limitado a visualizar un mensaje de llegada, para focalizar en las duplicaciones. Se simula procesamiento mayor a 5 segundos para que los clientes retransmitan.
- Se resaltan las solicitudes duplicadas.



Servidor



Cliente 1



Cliente 2

Figura 5.6. Ejemplo de duplicación UDP

Como punto a favor de Sun RPC cabe mencionar que dicho temporizador es configurable; pudiendo decirle a nuestro cliente que retransmita cada intervalos de *n* segundos haciendo uso de la función *clnt_control* como vimos anteriomente. Más allá de esto, es difícil que un cliente

pueda adaptar sus tiempos para retransmitir al funcionamiento del servidor, ya que este último en un momento determinado puede estar manejando una cantidad considerable de solicitudes y en otro estar libre; por lo que sus prestaciones no puede estimarse. Además, en el escenario ideal en que esto pueda hacerse, siempre debemos contemplar fallas para situaciones particulares, como el tráfico de la red.
Es por esto que se intentó erradicar el problema de fondo y generar un servidor con la capacidad de darse cuenta cuando recibe una duplicación y no repetir procesamiento.

Para esta tarea haremos uso del XID comentado en el Capítulo 2. Teniendo en cuenta que el cliente no modifica el XID en duplicaciones, el servidor puede descartar aquellas solicitudes que tengan un XID ya recibido. Dicho dato podemos encontrarlo, ocultamente, en la estructura *svcudp_data* que referencia el puntero xp_p2 del manejador del servidor (SVCXPRT) visto anteriormente. La definición de dicha estructura puede verse en la Figura 5.7.

```
struct svcudp_data {
        unsigned int su_iosz;
        unsigned long su_xid;
        XDR su_xdrs;
        char su_verfbody[MAX_AUTH_BYTES];
        char * su_cache;
};
```

Figura 5.7. Estructura svcudp_data

Se puede objetar que 2 clientes distintos puedan elegir un mismo XID para su respectiva solicitud, situación que, aunque improbable como veremos, no deja de ser una posibilidad. Es por esto que además de tener en cuenta al XID, nuestra solución hará uso del número de puerto del cliente que, como vimos, es elegido aleatoriamente por el sistema operativo. Con esta combinación, la probabilidad de que distintos clientes en un mismo intervalo de tiempo coincidan en solicitar un procedimiento con idénticos XID y número de puerto disminuye considerablemente; sobre todo si pensamos en cómo se genera un XID [28], tal como se muestra en la Figura 5.8:

```
struct timeval now;

gettimeofday(&now);

call_msg.rm_xid = getpid() ^  now.tv_sec ^ now.tv_usec;

// donde ^ es OR exclusivo bit a bit.
```

Figura 5.8. Generación de XID

Independientemente de la estructura que se piense en utilizar para mantener el registro de los XID recibidos, debemos tener en cuenta que esta crecerá proporcionalmente a las solicitudes

entrantes y que, de ser considerables, las prestaciones del servidor pueden reducirse si no se lleva a cabo un mantenimiento de manera tal de que no se mantenga información que el servidor no necesitará (por ejemplo, solicitudes que ya fueron respondidas). En respuesta a esto, surge el concepto de marca de tiempo: para cada solicitud entrante se registra el tiempo de llegada al servidor y éste eliminará los registros que considere sin utilidad llevando una actualización periódica de la estructura y evitando crecimientos indeseables.

Además, y más allá de que un servidor generalmente no está ligado a un único cliente, al generar código con rpcgen podemos coordinar inicialmente este tiempo de actualización en base al *timeout* del cliente inicial. Esto es: si nuestro cliente se sabe que finalizará por *timeout* en *m* segundos, del lado del servidor tenemos la certeza de que el XID utilizado por éste en una solicitud tendrá sentido de que esté registrado como máximo hasta *m* segundos; ya que luego de este intervalo el cliente de no recibir respuesta finalizará por *timeout* global y, en caso de que vuelva a solicitar, el XID utilizado será diferente al inicial.

La solución elegida es registrar las solicitudes en el *wrapper* del servidor en una estructura de lista; donde cada nodo estará formado por la terna XID - Puerto - Marca de tiempo. Dicha estructura se crea al iniciar el servidor y ante una nueva solicitud, su funcionamiento consiste en obtener el XID y el número de puerto de la misma y buscar una coincidencia registrada. De no encontrarse, quiere decir que es la primera solicitud de ese cliente, por lo que se crea un nuevo nodo para esta, se almacena en la lista y se continúa con el procesamiento normal. En el caso en que exista una coincidencia, el *wrapper* finalizará sin haber delegado el procesamiento al proceso, evitando de esta forma ejecutar algo que ya se hizo o está siendo ejecutado.

El motivo de registrar el tiempo de llegada, es que mientras se itera en la estructura en búsqueda de coincidencia, se verifica que el tiempo de permanencia de cada nodo no sea mayor al permitido (que por defecto será el valor del *timeout* global). Si el tiempo es mayor, el nodo es eliminado ya que podemos estar seguros de que ese cliente recibió nuestra respuesta o finalizó por tiempo de espera. El funcionamiento del buscador puede verse en la Figura 5.9. Esta tarea implica un tiempo extra que se decide invertir para garantizar la estabilidad del servidor.

Función *l_find*

Recibe:
1. Lista de solicitudes
2. XID de la solicitud actual
3. Puerto de la solicitud actual

// Para cada nodo de la lista

// Comparo XID y puerto del nodo con los de la solicitud actual.
De ser iguales, registro que existe.

// Comparo la hora de llegada del nodo con la hora actual.
Si la diferencia entre ambas es mayor al timeout global configurado, elimino el nodo.

// Retorno el resultado de la búsqueda

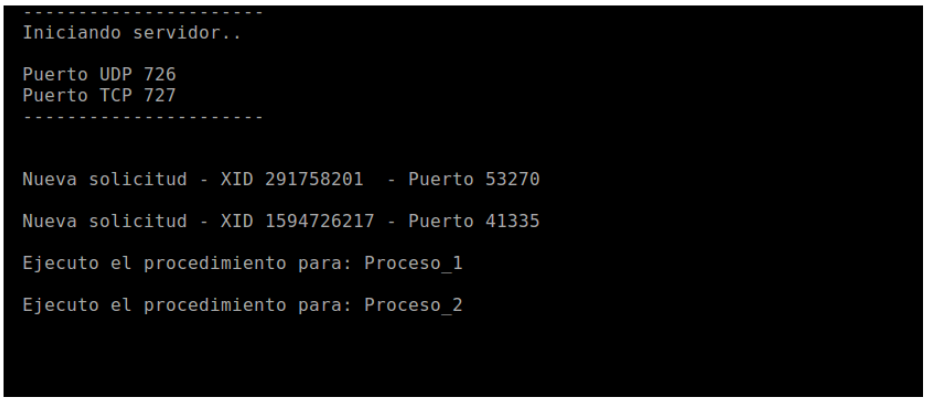
Figura 5.9. Funcionamiento del buscador de solicitudes

Técnicamente, el código utilizado ahora en el wrapper puede verse en la Figura 5.10.

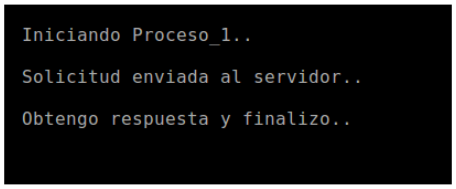
```
static void
display_prg_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
        unsigned long xid = ((svcudp_data *) transp->xp_p2)->su_xid;
        unsigned short int port = transp->xp_raddr.sin_port;
        time_t timestamp = time(NULL);

        if(!l_find(&l,xid,port)) {
                l_add(&l,xid,timestamp,port);
                pid_t forkpid;
                forkpid = fork();
                if (forkpid == -1) {
                        fprintf(stderr, "%s", "Error creating the first process");
                        exit(1);
                }
                if (forkpid == 0) {
                        if ((forkpid = fork()) == -1) {
                                fprintf(stderr, "%s", "Error creating the second process");
                                exit(1);
                        }
                        else {
                                if (forkpid == 0) {
                                        execute(rqstp, transp);
                                        exit(0);
                                }
                                exit(0);
                        }
                }
                if (waitpid(forkpid, NULL, 0) != forkpid)
                        fprintf(stderr, "%s", "Error waitingfor child process");

                return;
        }
}
```

Figura 5.10. Wrapper final para multi-procesos

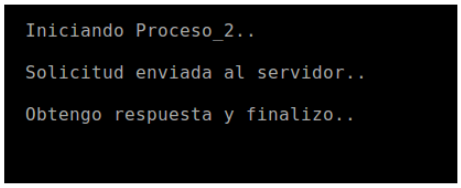
Aplicando la solución comentada, podemos ver en la Figura 5.11 que el servidor filtra duplicaciones y solo ejecuta una vez el procedimiento solicitado por cada cliente.



Servidor



Cliente 1



Cliente 2

Figura 5.11. Ejemplificación con filtro de duplicaciones

Este comportamiento se encuentra automatizado en rpcgenmp (Capítulo 6 - Extensiones a rpcgen).

5.2 Concurrencia con threads

Definida la solución para brindar concurrencia con procesos, podemos pensar en reemplazar a éstos con threads y darle al programador la posibilidad de elección en base a las características de su aplicación.

Antes de describir la solución propuesta, comentaremos brevemente cómo utilizar threads en C de forma tal de asentar los conocimientos básicos que usaremos en dicha solución. Los siguientes conceptos tienen su fundamento en [11] y [23].

5.2.1 Creación y manipulación de threads

POSIX Threads, generalmente conocido como pthreads, es un estándar para la manipulación de threads de ejecución con implementaciones disponibles para diferentes sistemas operativos, como GNU/Linux, FreeBSD, OpenBSD, Solaris y Windows.
Pthreads define un conjunto de tipos de datos, funciones y constantes para el lenguaje de programación C implementados a través de un archivo de cabecera llamado *pthread.h* y una librería que brinda funciones para la administración de threads y sincronización de concurrencia.

Para crear programas que hagan uso de la biblioteca pthreads necesitamos instalarla en nuestro sistema y enlazar a ella nuestro programa al momento de compilar. Por ejemplo, si usamos gcc como compilador:

```
gcc programa_pthreads.c -o programa_pthreads -lpthread
```

Cada thread es identificado a través de un ID, para lo que se utiliza el tipo de dato pthread_t.
Una vez creado, el thread ejecuta una función que se le especifica al momento de su creación.
En Linux, estos aceptan y devuelven un parámetro simple de tipo void *, de manera tal de poder utilizar cualquier tipo de dato realizando los casteos correspondientes.

La función para crear un nuevo thread es pthread_create:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*),
                   void *arg);
```

Cuyos parámetros son:

1. Un puntero a una variable de tipo pthread_t, en donde el ID del thread es almacenado.
2. Un puntero a un objeto de atributos a través del cual se controlan detalles de cómo el thread interactuará con el resto del programa.
3. Un puntero a la función que el thread ejecutará.
4. El argumento del thread, que es lo que se le pasará a la función del tercer parámetro cuando el thread comience a ejecutarse.

Si la función falla, no se crea el thread y lo referenciado por thread es indefinido. En caso de éxito, se retorna 0. De otra manera, un número de error es devuelto.

Una llamada a *pthread_create* retorna inmediatamente y el thread original continúa su ejecución. Mientras tanto, el nuevo thread empieza a ejecutar su función. Linux maneja ambos de manera asincrónica, por lo que el orden de ejecución de las instrucciones de cada thread no puede conocerse.

En circunstancias normales, un thread puede terminar de 2 maneras:

- Retornando comúnmente de la función.
- A través de *pthread_exit*, función que recibe un puntero al valor de retorno del thread:

void pthread_exit(void **value_ptr*);

Análogamente a los procesos, los threads tienen la posibilidad de esperar la finalización de otros. Esto se realiza desde la función *pthread_join*, cuyos argumentos son el ID del thread a esperar y un puntero a una variable donde se almacenará el valor de retorno de éste.

int pthread_join(pthread_t *thread*, void ***value_ptr*);

Existen diversos atributos configurables para un thread, de los que vale la pena mencionar el estado de relación para con quien lo ejecutó. Un thread puede ser acoplable (*joinable*, por defecto) o independiente (*detach*). Al igual que con un proceso, el sistema no libera recursos automáticamente al momento de finalización de un thread acoplable. Esto se realiza exclusivamente cuando otro thread llama a *pthread_join* para obtener su valor de retorno. Para un thread independiente, en cambio, se liberan sus recursos automáticamente cuando finaliza; motivo por el cual otro thread no puede sincronizar con éste y obtener su valor de retorno. Para indicar este estado se utiliza la función:

int pthread_attr_setdetachstate(pthread_attr_t **attr*, int *detachstate*);

donde el primer argumento es un puntero a un objeto de atributo y el segundo debe tomar uno de los siguientes valores:

- PTHREAD_CREATE_DETACHED
- PTHREAD_CREATE_JOINABLE

5.2.2 Aproximación inicial

Conceptualmente, como para con procesos, se intentará aplicar MT haciendo uso de en una función *wrapper* que ahora deberá crear threads vía *pthread_create* en lugar de procesos con *fork*. Para mayor claridad, en este momento no tendremos en cuenta el filtro para duplicaciones.

Como el *wrapper* de procesos recibe 2 parámetros y un thread solamente acepta un void*, se necesita definir una estructura auxiliar y enviar una referencia a esta estructura, realizando los casteos necesarios en ambos extremos, la cual puede verse en la estructura de la Figura 5.12.
El *wrapper* entonces solicitará memoria y asignará las referencias. Esto es necesario para

evitar solapamiento de datos, ya que si se utilizan los punteros iniciales en las siguientes solicitudes los datos de nuestro thread serán reemplazados por los correspondientes a las siguientes solicitudes.

Luego se lanza un thread marcado como *detached*, ya que no necesitamos sincronizar con ningún otro thread para obtener algún resultado, y será éste quien se encargue de responder al cliente. La función *execute* pasada al thread deberá entonces desempaquetar lo recibido y continuar con el funcionamiento por defecto de rpcgen visto anteriormente.

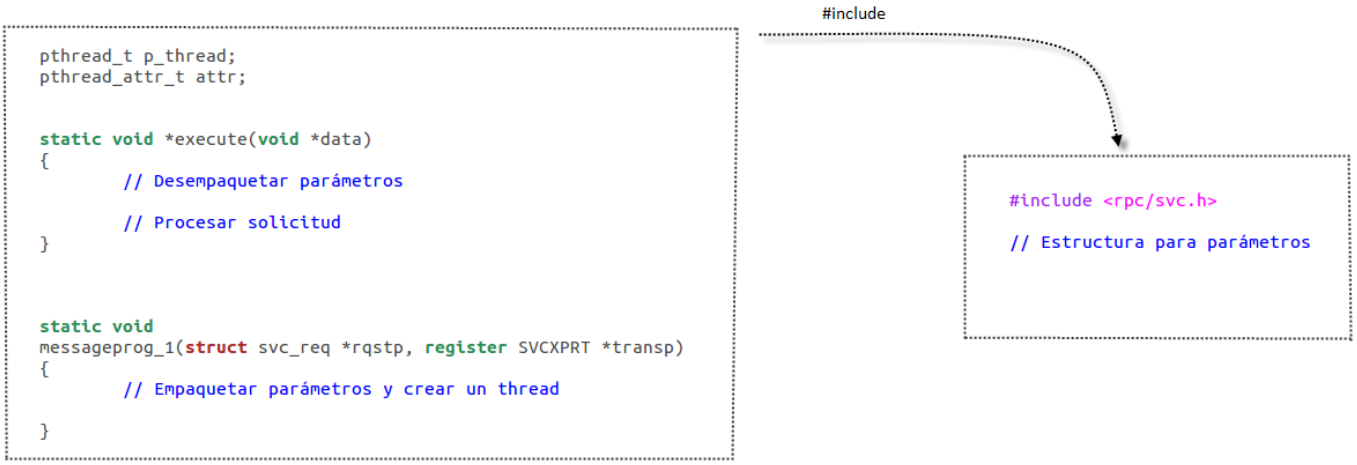


Figura 5.12. Aproximación inicial para multithreading

A pesar de que se respeta la secuencia lógica que funcionó con procesos, no es el caso para threads debido a detalles de funcionamiento que se comprobaron recién durante las pruebas de validación:

1. Decodificación de parámetros con TCP
   Cuando utilizamos TCP como protocolo de transporte, nuestro servidor MT no puede decodificar los parámetros del procedimiento. Esto se debe a que *svc_getargs,* la función usada para dicha operación, no es segura para ambientes MT.
   Se nos presenta el mismo problema si intentamos liberar la memoria utilizada a través de *svc_freeargs* [24].
   Es necesario mencionar que la documentación sobre las funciones anteriores es muy escasa y no se puede considerar completamente confiable. Lo que sí es seguro es que falla.
2. Solicitudes entrecruzadas con UDP
   Cuando utilizamos UDP como protocolo de transporte, aunque puede verse la concurrencia ejecutada correctamente en el servidor, en el retorno de los

resultados las respuestas se entrecruzan; es decir, los clientes reciben respuestas de solicitudes diferentes a las suyas. Además, algunas respuestas nunca llegan al cliente ocasionando la finalización de *timeout*.

A continuación detallaremos las propuestas de soluciones encontradas a estos problemas.

5.3 MT con TCP: Solución a la decodificación de parámetros

Para verificar/replicar el problema encontrado y despejar dudas de que ocurra exclusivamente para una distribución, se ejecutaron los casos prácticos en las siguientes distribuciones de Linux:

1. Mint - versión 3.2.0.2
2. Debian - versión 2.6.32
3. Ubuntu - versión 3.0.0
4. Lihuen - versión 2.6.32
5. Mandriva - versión 2.6.38
6. Fedora - versión 2.6.27
7. CentOS - versión 2.6.32

Y en todas se obtiene el mismo resultado: las funciones del lado del servidor, entre ellas *svc_getargs*, no son seguras para ambientes MT. El problema se puede ver en la siguiente Figura 5.13, donde el cliente instantáneamente luego de invocar al servidor recibe el mensaje de error.

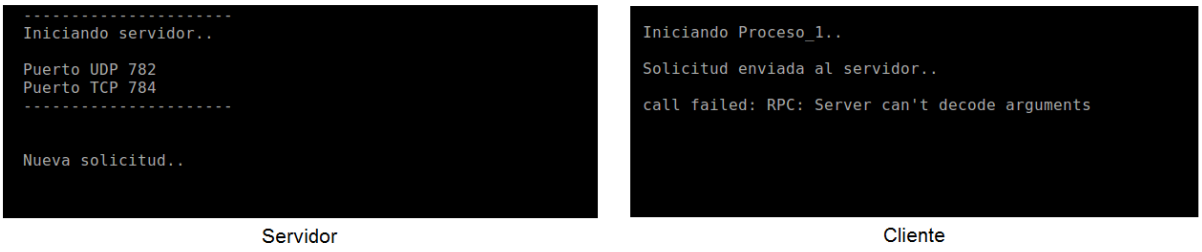


Figura 5.13. Error en la decodificación de parámetros TCP

Podemos pensar entonces en obtener los parámetros del procedimiento en el proceso, empaquetar estos en una estructura y lanzar el thread con la misma.
Otro punto a tener en cuenta es que se necesita asignar memoria para dicha estructura ante cada solicitud. Esto debe ser así porque el proceso tiene un área de memoria estática asignada (pila). Cuando recibe una solicitud y se llama al *wrapper*, se almacenan en la pila las referencias a las estructuras que necesita el servidor para ejecutarse. Una vez que lanza el thread, el *wrapper* finaliza y el programa regresa a *svc_run*; lo que ocasiona que los datos

anteriormente apilados sean borrados de la misma (técnicamente, dejan de referenciarse). Ahora, ante una nueva solicitud, la secuencia se repite: se generan las estructuras necesarias para el servidor, se apilan las referencias y se llama al *wrapper*. Es en este punto donde la asignación de memoria empieza a tomar importancia: cuando asignamos estamos usando memoria libre de uso común dinámica (*heap*); es decir, el sistema operativo busca memoria libre para asignar, la cual no puede utilizarse hasta que no sea liberada explícitamente (o el proceso finalice y el sistema recupere mediante un colector). Si no hiciéramos esta tarea, y enviáramos referencia a la pila, las estructuras de la nueva solicitud van a sobreescribir las de su predecesora. Esta puede continuar procesándose normalmente ya que sus punteros no se enteran de este cambio. Pero en este escenario lo hará con referencias hacia datos que no se corresponden con los originales.
Este es uno de los problemas fundamentales de la programación con threads: comparten la memoria del proceso, por lo que existen casos en los que un thread puede modificar datos utilizados por otro si no se contempla la situación.

La idea comentada anteriormente puede verse en la Figura 5.14, donde las responsabilidades del *wrapper* abarcan las funcionalidades hasta el desglose de parámetros inclusive, vistas anteriormente:

1. Decide cuál es el procedimiento que necesita ser invocado.
2. Desglosa los parámetros recibidos para el mismo.
3. Completa una estructura con los datos que necesita el thread para ejecutarse.
4. Crea y lanza el thread.


```
static void messageprog_1(struct svc_req *rqstp, register SVCXPRT *transp) {

        xdrproc_t _xdr_argument, _xdr_result;
        argum * arg = (argum *) malloc(sizeof(argum));
        resul * res = (resul *) malloc(sizeof(resul));
        localFunc local;

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

        case PRINTMESSAGE:
                _xdr_argument = (xdrproc_t) xdr_wrapstring;
                _xdr_result = (xdrproc_t) xdr_int;
                local = (bool_t (*) (char *, void *,  struct svc_req *))printmessage_1_svc;
                break;

        default:
                svcerr_noproc (transp);
                return;
        }
        memset ((char *)arg, 0, sizeof (argum));

        if (!svc_getargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) arg)) {
                svcerr_decode (transp);
                return;
        }

        thr_data *data_ptr =(thr_data *)malloc(sizeof   (struct data_str));
        data_ptr->_xdr_result = _xdr_result;
        data_ptr->transp = transp;
        data_ptr->_xdr_argument = _xdr_argument;
        data_ptr->rqstp = rqstp;
        data_ptr->func = local;
        data_ptr->arg = arg;
        data_ptr->res = res;

        pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
        pthread_create(&p_thread,&attr,execute_function,(void *)data_ptr);
}
```

Figura 5.14. Wrapper TCP para decodificación de parámetros

La función *execute* que puede verse en la Figura 5.15, entonces:

1. Desglosará lo recibido.
2. Invocará al procedimiento, enviará la respuesta al cliente y liberará recursos.

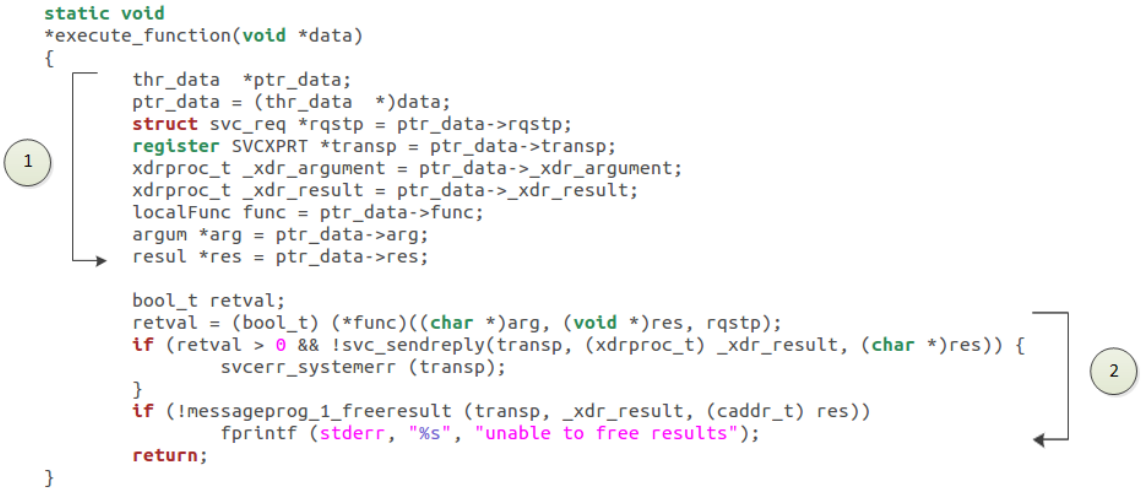


Figura 5.15. Nuevo execute para decodificación TCP

Se puede notar que en esta versión de *execute* no se utiliza *svc_freeargs* debido a lo comentado anteriormente: no es segura para MT, por lo que el servidor cae si hacemos uso de ella. Internamente, *svc_freeargs* hace uso de una funcionalidad XDR denominada xdr_free [22]. Esta, si se utiliza rpcgen con la opción -M para indicar que se genere código seguro para MT, se invoca luego del regreso de los resultados. En nuestro caso, adherimos dicha llamada (messageprog_1_freeresult) en *execute* para no perder funcionalidad.

Por último, debemos comentar que se utiliza una estructura definida para permitir la comunicación entre el *wrapper* y el thread denominada *thr_data* y se definen como tipo de dato a aquellos que representan los parámetros de entrada y salida del procedimiento, brindando legibilidad y simplicidad. Estas definiciones pueden verse en la Figura 5.16.

```c
#ifndef GLOBAL_TIME_OUT
#define GLOBAL_TIME_OUT 25
#endif


#include <rpc/xdr.h>
#include <rpc/auth.h>


union argument {
        // Parámetro al procedimiento remoto
};
typedef union argument argum;


union result {
        // Resultado del procedimiento remoto
};
typedef union result resul;



typedef bool_t (*localFunc)(char *, void *, struct svc_req *);

struct data_str
{
        // Esta estructura contiene:
        //      - Rutina XDR para el parámetro
        //      - Rutina XDR para el resultado
        //      - Manejador del servicio (SVCXPRT)
        //      - Representante de la solicitud (svc_req)
        //      - Función a ejecutar
        //      - Parámetro al procedimiento remoto
        //      - Resultado del procedimiento remoto

};
typedef struct data_str thr_data;
```

Figura 5.16. Archivo header para estructuras TCP

Con esta corrección se permite MT con TCP sin fallas en el desglose de parámetros y en la liberación de memoria, como puede verificarse en la Figura 5.17.

Este comportamiento se encuentra automatizado en rpcgenmt (ver Capítulo 6).

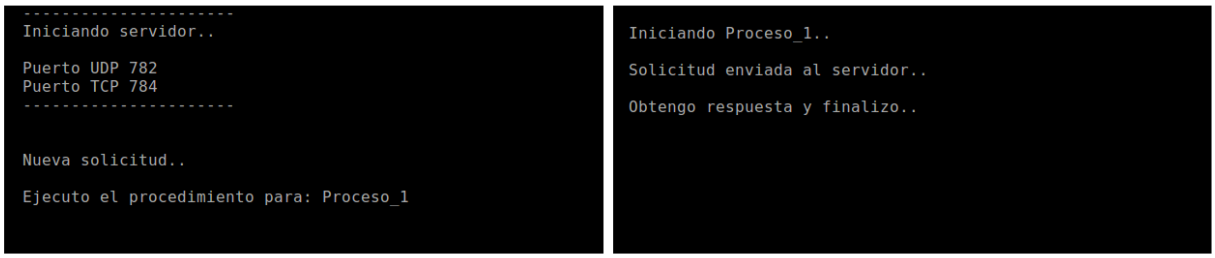


Figura 5.17. Ejemplificación de la solución a la decodificación TCP

5.4 MT con UDP: retornos cruzados

Caso 1

Cuando realizamos MT con UDP en las pruebas de validación, se detectó que los clientes reciben respuestas que no se condicen con su solicitud inicial. Mostraremos dicho escenario a través de un ejemplo donde:

- El servidor recibe solicitud de 3 clientes.
- Cada cliente (C1, C2 y C3) envía como parámetro (P1, P2 y P3) la cadena '*Cliente_número*', donde *número* se corresponde con el número de cliente.
- El servidor al iniciar imprime los puertos asociados a TCP y UDP.
- El servidor ante cada solicitud no duplicada imprime el puerto y XID del cliente antes de lanzar el thread.
- El procedimiento del servidor adhiere la hora de llegada al parámetro recibido, imprime y envía este resultado al cliente. Simula además un tiempo de procesamiento vía *sleep.*
- El cliente imprime el resultado obtenido.

Cabe aclarar que al no tener problemas sobre la decodificación de parámetros ni en la liberación de memoria con UDP, el código a analizar presenta la lógica inicial donde el wrapper englosa parámetros y lanza el thread (Figura 5.12).

Inicialmente, intentando reasignar las referencias antes de lanzar el thread, se obtuvo como resultado lo graficado en la Figura 5.18.

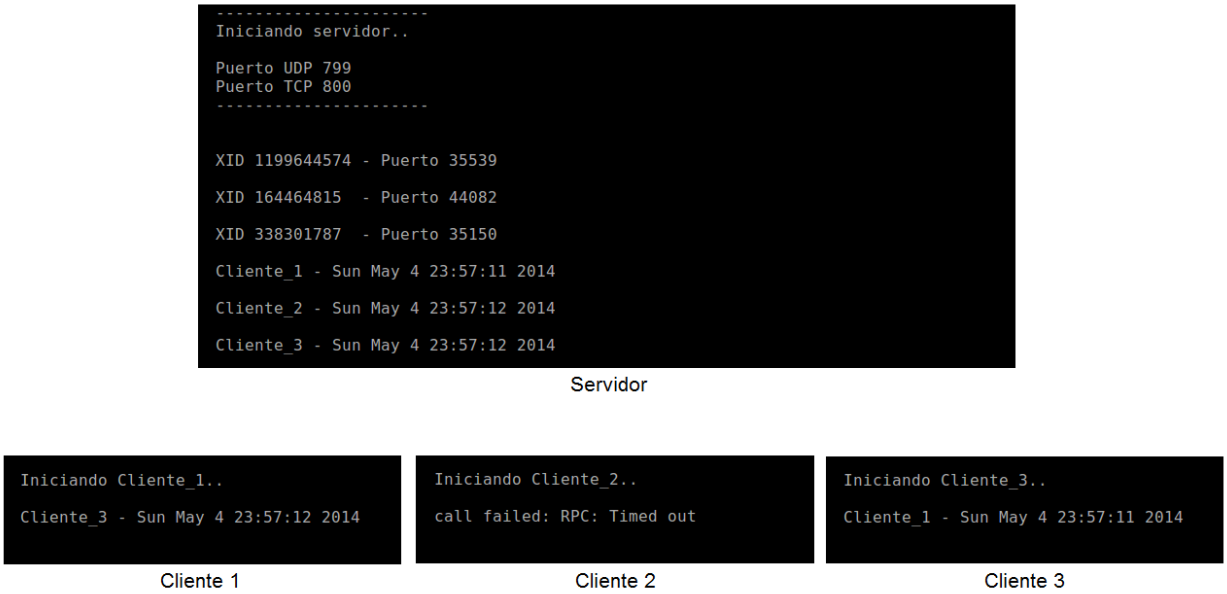


Figura 5.18. Retornos cruzados UDP

Como puede verse, existe entrecruzamiento de resultados entre C1 y C3; mientras que C2 finaliza por timeout. Se intentó entonces monitorear el tráfico de la red con la herramienta Wireshark para indagar cuáles eran los mensajes que realmente se enviaron y observamos que:

- Líneas 1 a 8: asociación de puertos iniciales.
- Línea 9: C1 envía su solicitud.
- Línea 12: C2 envía su solicitud.
- Línea 15: C3 envía su solicitud.
- Línea 16: el servidor responde a C3 con P1 y XID3.
- Línea 17: el servidor responde a C3 con P2 y XID3.
- Línea 19: C1 vuelve a solicitar.
- Línea 20: el servidor responde a C1 con P3 y XID1.
- Líneas 21 a 24: C2 vuelve a solicitar.

Se detectó que el número de puerto y el XID que utiliza el servidor al momento de responder es siempre el último recibido (en este caso los de C3). Las variables de puerto y XID parecen estar sobrescribiéndose ante cada nueva solicitud; premisa que la línea 19 nos ayuda a reforzar: cuando C1 vuelve a solicitar, el puerto y XID de la siguiente respuesta se actualizan.
Se cambió el tiempo de simulación de procesamiento en el proceso servidor para forzar a que los 3 procedimientos finalicen antes de que algún cliente vuelva a solicitar y se ve que todas las respuestas van a C3. En consecuencia, el servidor enviará la respuesta de cada procedimiento al último cliente que solicitó.

Con esta situación, si n clientes acceden en un intervalo de tiempo no muy distante, sólo el último de ellos recibirá respuesta; y ésta no necesariamente (y con alta probabilidad) será la que le corresponda. En otras palabras, el stub usa el XID y número de puerto del último cliente para responder a cada solicitud; por lo que los n resultados de las n solicitudes regresarán a dicho cliente. Éste al recibir la primera de ellas continuará con su procesamiento y no se enterará del resto de las respuestas que le fueron dirigidas. Como es muy probable que el procesamiento de alguna de las n-1 solicitudes finalice antes que la número n, el último cliente generalmente recibirá una de estas respuestas y no la suya en particular.

Caso 2

Para evitar el escenario anterior, intentamos reemplazar la asignación de referencias del *wrapper* por copia de memoria, como se muestra en la Figura 5.19.

Reemplazar:

```
// Si no es una duplicación
        // Imprimo puerto y XID
        data_str *data_ptr=(struct data_str*)malloc(sizeof   (struct data_str));
        data_ptr->rqstp = rqstp;
        data_ptr->transp = transp;
        // Lanzo el thread
```

por:

```
// Si no es una duplicación
        // Imprimo puerto y XID

        data_str *data_ptr=(struct data_str*)malloc(sizeof   (struct data_str));
        data_ptr->rqstp = (struct svc_req *) malloc(sizeof(struct svc_req));
        memcpy(data_ptr->rqstp,rqstp,sizeof(struct svc_req));
        data_ptr->transp = (SVCXPRT *) malloc(sizeof(SVCXPRT));
        memcpy(data_ptr->transp, transp, sizeof(SVCXPRT));
        // Lanzo el thread
```

Figura 5.19. Duplicación de memoria en wrapper UDP

y la situación es la misma: las respuestas van al último cliente que solicita.
Se pensó entonces en imprimir los valores del puerto y XID de cada solicitud para tratar de entender en qué momento cambiaban y se descubrió una situación en principio inesperada: al imprimir los valores antes de llamar a la *svc_sendreply*, función que envía el resultado al

cliente, se ve que efectivamente se repite el valor del XID pero no así el del número de puerto. Esto es: en el monitoreo de tráfico de red se ve que ambos valores corresponden al último cliente, pero en el servidor, antes de enviar la respuesta, el número de puerto es el correcto.
En la Figura 5.20 se muestra lo comentado (se quitan las impresiones del procedimiento servidor para mejor legibilidad).

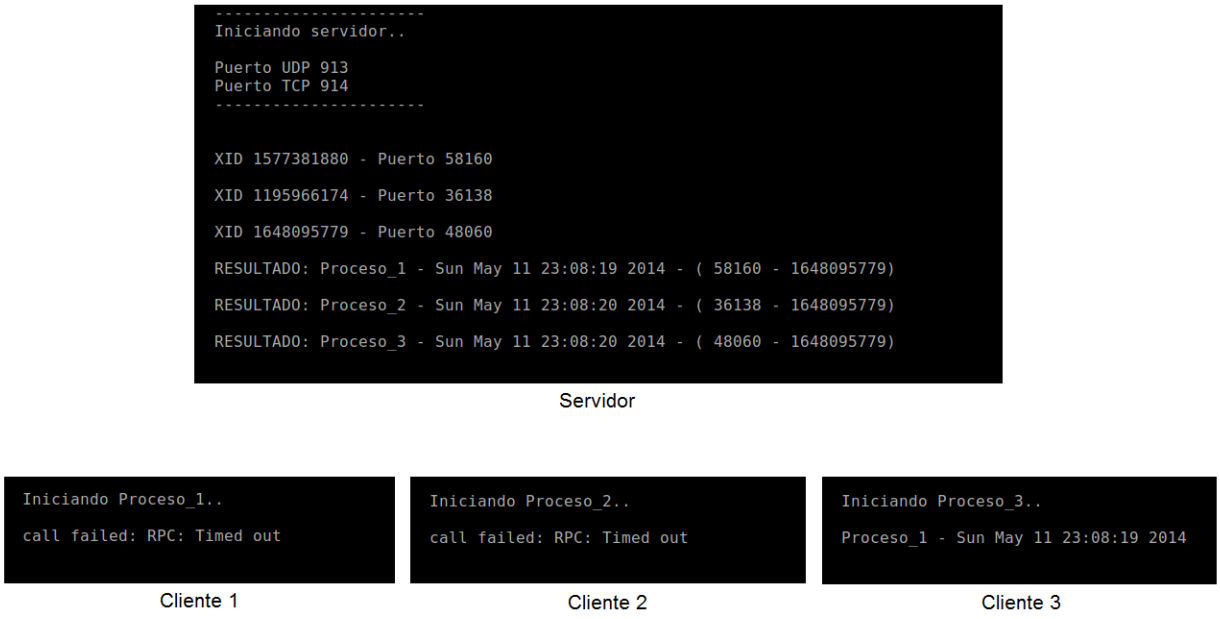


Figura 5.20. Monitoreo de XID y puertos UDP

Debido a la escasa documentación existente sobre Sun RPC, no se pudo conocer exactamente el motivo de esta situación; llegando a pensarse que cuando se utiliza UDP, al ser no orientado a conexión y no tener el servidor que mantener información sobre cada cliente, se utiliza un área de memoria reservada donde se almacena el número de puerto de la última solicitud. Es decir, cuando queremos volver al cliente la función *svc_sendreply* utiliza datos internos del stub que no son los que nosotros podemos gestionar.

Caso 3

Más allá de que no se pueda solucionar el hecho de regresar siempre a un mismo puerto, se intentó encontrar el motivo por el cual el XID no se actualiza ya que vemos que efectivamente es el mismo siempre.

La respuesta a esto es que al realizar la copia de memoria, el puntero que referencia a la estructura que contiene el XID de la solicitud continúa referenciando el mismo espacio de memoria luego de ser copiado. Esto es: el XID se encuentra en la estructura *svcudp_data* referenciada por *xp_p2* del manejador, como vimos en la Figura 5.7. Si a su vez realizamos una copia de este espacio, el problema debería resolverse. Esto, a nivel código, puede verse en la Figura 5.21.

```
// Si no es una duplicación

        // Imprimo puerto y XID

        data_str *data_ptr=(struct data_str*)malloc(sizeof   (struct data_str));

        data_ptr->rqstp = (struct svc_req *) malloc(sizeof(struct svc_req));
        memcpy(data_ptr->rqstp,rqstp,sizeof(struct svc_req));

        data_ptr->transp = (SVCXPRT *) malloc(sizeof(SVCXPRT));
        memcpy(data_ptr->transp, transp, sizeof(SVCXPRT));

        data_ptr->transp->xp_p2 = malloc(sizeof(struct svcudp_data));
        memcpy(data_ptr->transp->xp_p2,transp->xp_p2,sizeof(struct svcudp_data));

        // Lanzo el thread
```

Figura 5.21. Copia de svcudp_data para XID UDP

Efectivamente, como se suponía y pudo confirmarse mediante el tráfico capturado con Wireshark, el XID de cada cliente se mantiene. Ahora, aunque siempre se responda al mismo cliente, éste será capaz de diferenciar los paquetes recibidos y sólo entenderá como suyo el correcto; desechando aquellas respuestas que no se condigan con el XID esperado. En detalle:

- Líneas 1 a 8: asociación de puertos iniciales.
- Línea 9: C1 envía su solicitud.
- Línea 12: C2 envía su solicitud.
- Línea 15: C3 envía su solicitud.
- Líneas 16 a 18: los clientes vuelven a solicitar.
- Línea 19: el servidor responde a C3 con P1 y XID1. El cliente no acepta ya que no es el XID esperado.
- Línea 21: el servidor responde a C3 con P2 y XID2. El cliente no acepta ya que no es el XID esperado.
- Línea 23: el servidor le responde a C3 con P3 y XID3, por lo que la respuesta es recibida.

En la Figura 5.22, podemos ver que C3 muestra correctamente su resultado (a diferencia del primer caso, donde mostraba la respuesta de C1):

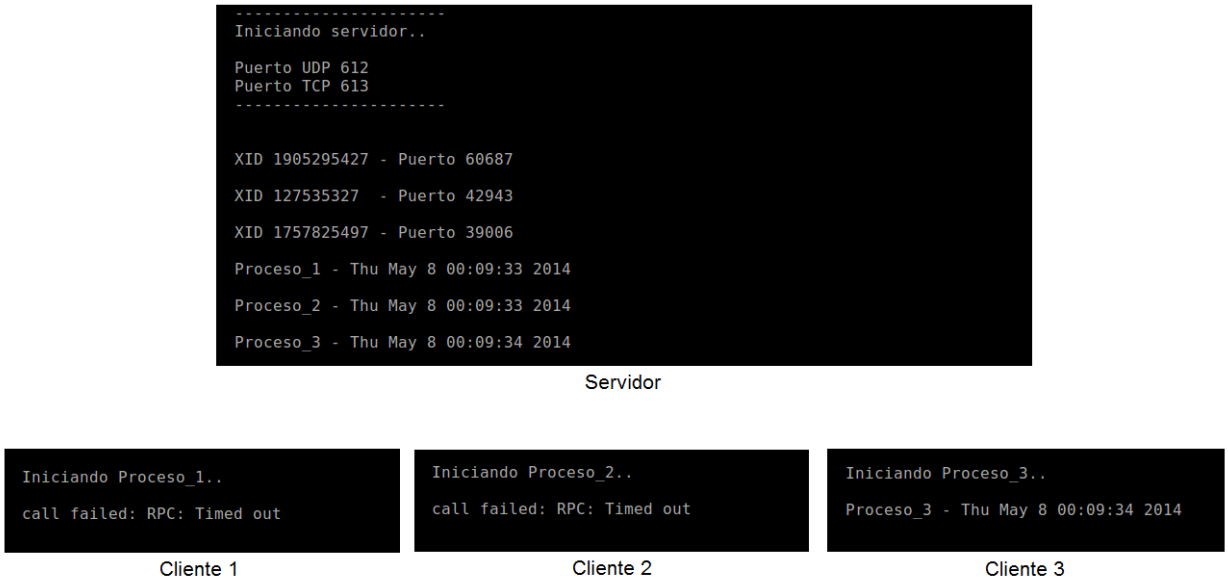


Figura 5.22. Ejemplificación de XID correcto

En resumen, cada thread en el servidor tiene correctamente asociado el XID del cliente pero el número de puerto sigue siendo el último recibido; hecho que no podemos modificar dado que no se conoce su implementación.
Se pensó en desarrollar nuestra propia *svc_sendreply* para que, reemplazando a la original, tenga en cuenta al verdadero puerto que se necesita pero notamos que dicha función termina usando invocaciones internas de las que tampoco se registra documentación; por lo que volvíamos al problema inicial.
El cliente, entonces, puede recibir más de una respuesta para su solicitud pero tomará como válida sólo aquella que se corresponda con el XID generado.
Es necesario aclarar algunas consideraciones: se puede decir que los clientes que no reciben respuesta finalizarán por *timeout* global; y quizá, dependiendo de la lógica implementada, volverán a solicitar, esta vez con un XID distinto del anterior, operaciones que pudieron haberse ejecutado correctamente; cayendo nuevamente en la necesidad de que éstas sean idempotentes. Esto tiene su grado de verdad, pero también es verdadero de que si se quieren ejecutar operaciones no idempotentes de las cuales se necesita recibir un valor de resultado, UDP no es la solución ideal, siendo TCP la selección adecuada para dicho propósito.
Lo que estamos tratando de solucionar es la posibilidad de brindar concurrencia en el servidor con threads utilizando un transporte no confiable, resultado que se logra: dejando el resultado de la operación de lado, el servidor permite MT (sin problemas de decodificación o liberación de memoria). Se evita que un cliente reciba un resultado que no le corresponde, lo que es una mejora a la situación inicial.
Este comportamiento se encuentra automatizado en rpcgenmt (ver Capítulo 6).

6. Extensiones a RPCGEN

En el capítulo anterior se explicaron las alternativas y los problemas y soluciones para agregar MT a los servidores cuando se utiliza RPC. En este capítulo explicaremos cómo se incorpora este funcionamiento de manera automática, aprovechando que:

- rpcgen genera código a partir del IDL que conocemos.
- Conocemos qué código genera rpcgen.
- Hemos desarrollado un conjunto de alternativas de incorporación de MT en un servidor con RPC que podemos replicar de manera adaptada al IDL del programador y a la generación de código de rpcgen.

6.1 Servidor multi-procesos

El funcionamiento comentado anteriormente en la Sección 5.1 se encuentra automatizado en rpcgenmp. El mismo se basa en el rpcgen original y realiza de manera automática las correcciones necesarias para que el servidor delegue cada solicitud en un nuevo proceso. Se generan todos los archivos que intervienen en el sistema de RPC definido a partir de la especificación dada por el usuario en el IDL y el código generado por el propio rpcgen.
Se aplica la lógica de proceso auxiliar para evitar *zombies* y, si el protocolo elegido es UDP, se agregarán, además, las estructuras y la lógica necesarias para filtrar duplicaciones. Para esto se hace uso de 2 archivos auxiliares, distintos de los generados por rpcgen: *svc_types.h* para la definición y *svc_types.c* para la implementación. Se edita también el archivo Makefile agregando la referencia necesaria para que genere correctamente los programas, incluyendo el procesamiento concurrente. Se agrega como parámetro de generación del código fuente que el programador indique explícitamente el protocolo de transporte a utilizar (TCP o UDP).

Partiendo de los archivos que rpcgen genera, en la Figura 6.1 se detallan cuáles de ellos son modificados por nuestra versión y cuáles son generados por ésta. Tales modificaciones son transparentes al programador y como puede verse, lo que se modifica es solamente el funcionamiento del servidor (con la pequeña excepción de que cuando el protocolo elegido es TCP, el cliente debe modificarse para que realice este tipo de llamadas).

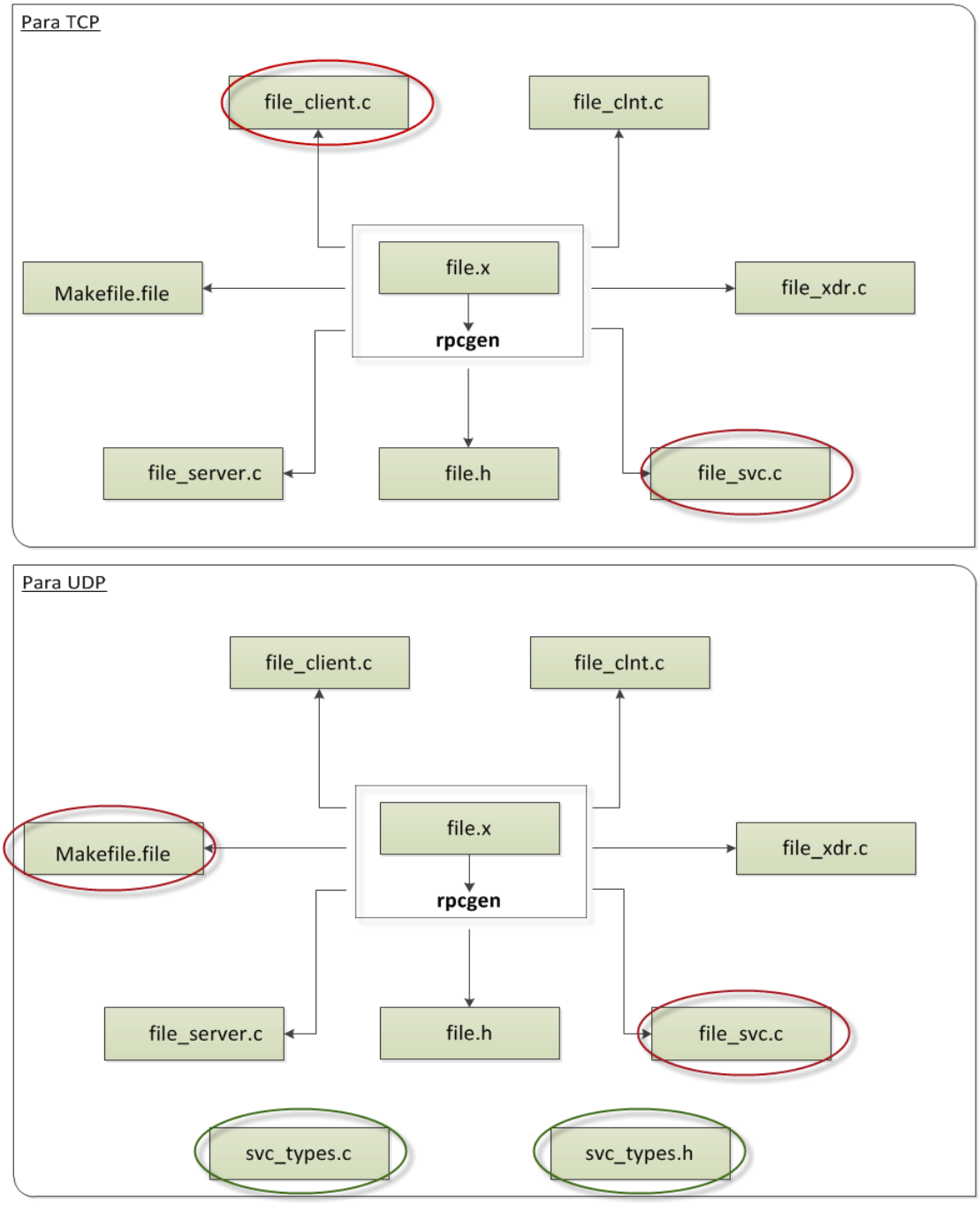


Figura 6.1. Archivos que intervienen en rpcgenmp

Para mostrar el funcionamiento de un servidor multiprocesos utilizando nuestra versión de rpgcen, mostraremos un ejemplo simple en el cual:

- Los clientes reciben una cadena a modo de identificador (Proceso_1, Proceso_2 y Proceso_3) y una secuencia de enteros. Con estos últimos, generarán una lista de enteros que enviarán al servidor junto con su identificador.
- El servidor ante cada solicitud creará un proceso, imprimirá su ID y delegará en éste el procesamiento.
- Cada proceso sumará los enteros de cada lista y enviará la suma final al cliente. Mientras realiza esta operación, irá imprimiendo la suma parcial junto al identificador del cliente para poder mostrar la concurrencia.

Como primer paso, definimos el archivo de especificación que puede verse en la Figura 6.2.
En él se define el procedimiento *sum* que recibe una estructura conformada por un string y una lista de enteros y que retorna un entero.

```
/* lisproceso.x */

typedef string texto<>;

struct nodo {
        int x;
        nodo *sig;
};

struct parametro {
        texto nombre;
        nodo* lista;

};

program LISTA_PROCESO {
    version LISTA_VERSION {
     int sum(parametro) = 1;
   } = 1;
} = 0x20000001;
```

Figura 6.2. Archivo de especificación multi-proceso

Luego compilamos utilizando nuestra versión de rpcgen llamada rpcgenmp. Con esto se generarán los archivos detallados anteriormente.
Agregamos la lógica comentada para el procesamiento de la lista junto con algunos mensajes

de monitoreo para una mejor comprensión. Una vez hecho esto, generamos los ejecutables del cliente y del servidor mediante el archivo makefile. Ambos pasos pueden verse en la Figura 6.3.

Figura 6.3. Generación y compilación con rpcgenmp

Finalmente, ejecutamos el servidor y los clientes. Como puede verse en la Figura 6.4, el servidor ejecuta concurrentemente las solicitudes recibidas. En la Figura 6.5 se grafica la funcionalidad de cada cliente: recibir los parámetros, formar la estructura, invocar al procedimiento remoto e imprimir el resultado obtenido.
Por último, en la Figura 6.6 se muestra una captura de los procesos activos durante la ejecución de este ejemplo; pudiendo verificarse que existen 3 procesos (uno por cada cliente) cuyos ID se corresponden con los creados por el servidor.

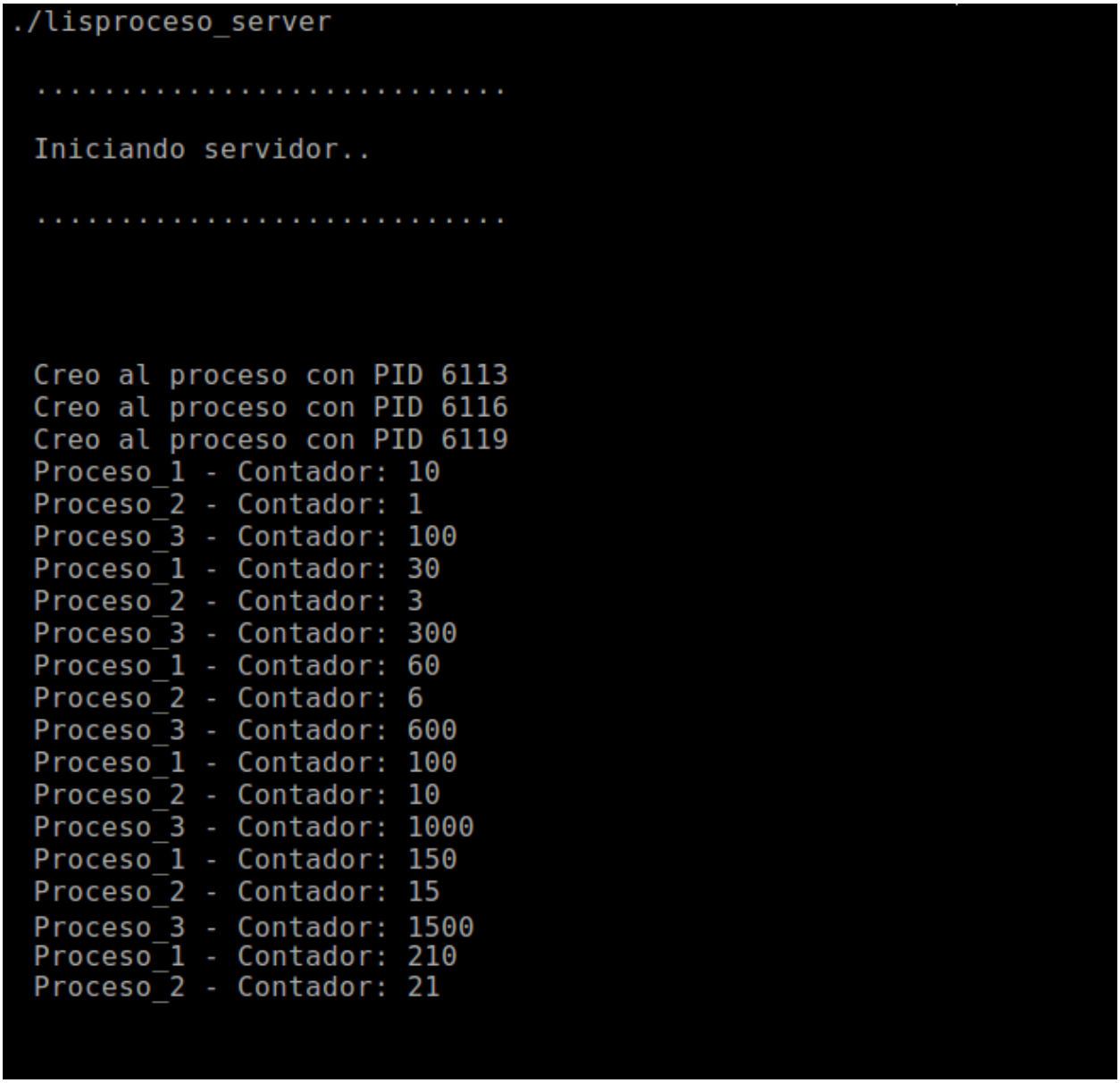


Figura 6.4. Ejecución del servidor multi-proceso

```
./lisproceso_client localhost Proceso_1 10 20 30 40 50 60
 Iniciando Proceso_1..
 Lista ingresada: 10 20 30 40 50 60
 Suma de la lista: 210
```

Cliente 1

```
./lisproceso_client localhost Proceso_2 1 2 3 4 5 6
 Iniciando Proceso_2..
 Lista ingresada: 1 2 3 4 5 6
 Suma de la lista: 21
```

Cliente 2

```
./lisproceso_client localhost Proceso_3 100 200 300 400 500
 Iniciando Proceso_3..
 Lista ingresada: 100 200 300 400 500
 Suma de la lista: 1500
```

Cliente 3

Figura 6.5. Ejecución de los clientes multi-proceso

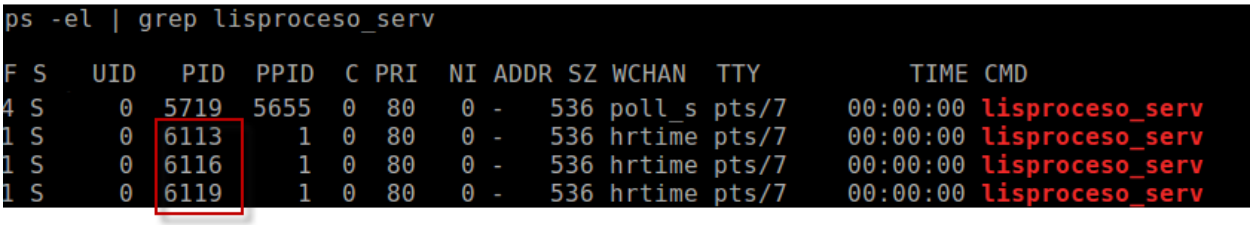
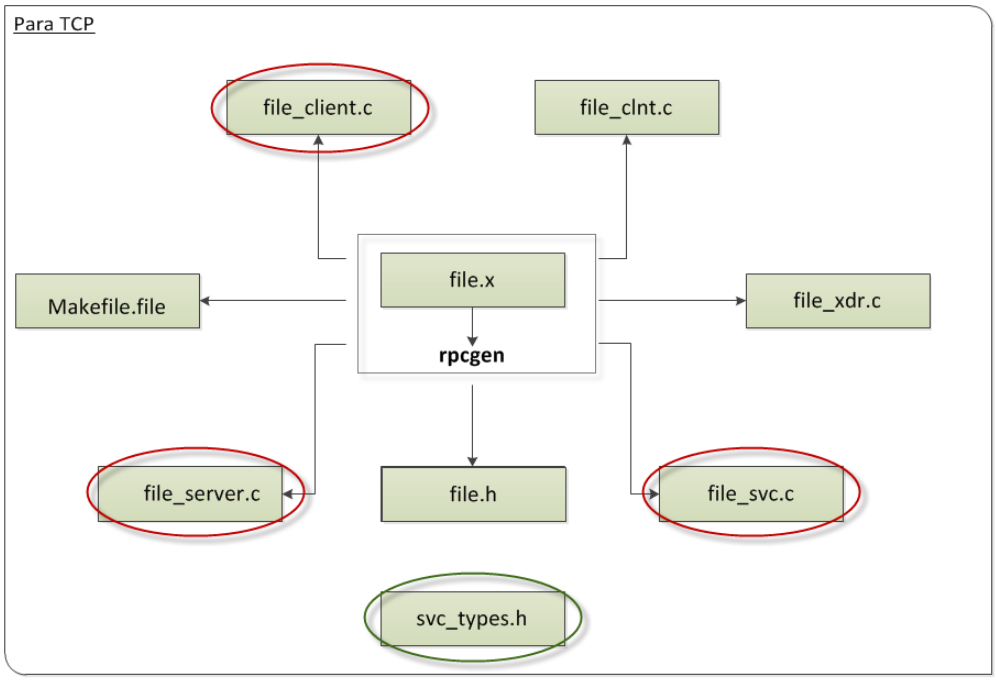
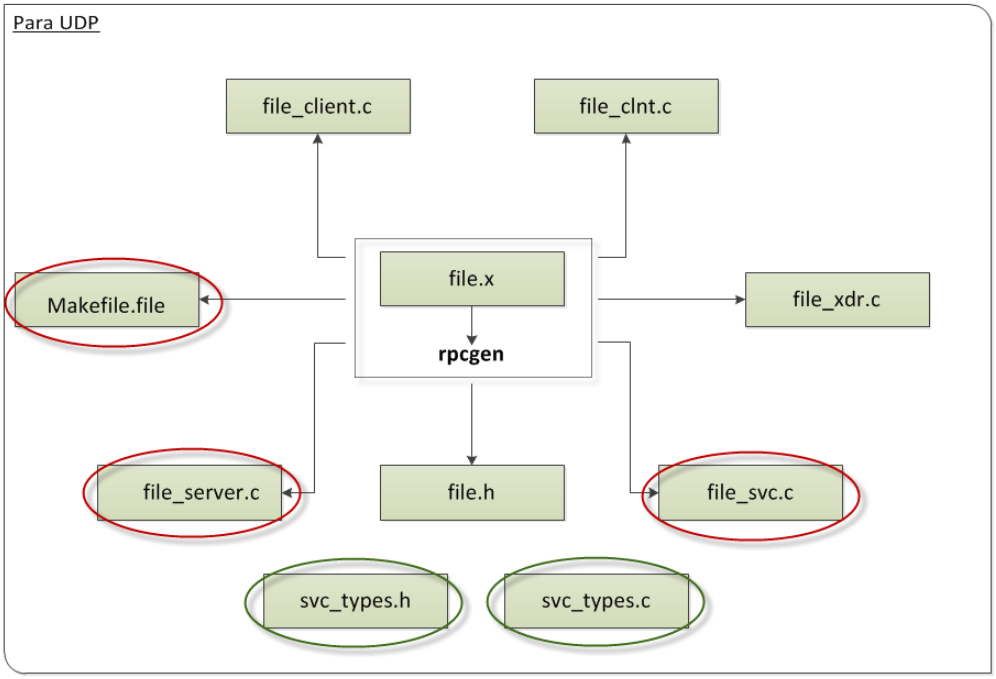


Figura 6.6. Monitoreo para multi-proceso

6.2 Servidor multithreading

El funcionamiento comentado anteriormente en la Sección 5.2 se encuentra automatizado en rpcgenmt. Al igual que en el caso anterior para procesos, se basa en el rpcgen original y realiza de manera automática las correcciones necesarias para que el servidor delegue cada solicitud en este caso en un nuevo thread. Se generan todos los archivos que intervienen en el sistema de RPC definido a partir de la especificación dada por el usuario en el IDL y el código generado por el propio rpcgen.
También se incorpora la filtración de duplicaciones vista para procesos cuando se utiliza UDP como protocolo de transporte.
Para ambos protocolos de transporte se define la estructura que permitirá la comunicación entre el wrapper y el thread. Debido a que, como vimos, TCP necesita una lógica diferente para dicho pasaje, se definen los tipos necesarios para parámetros de entrada y salida. Esto se logra, de igual forma que para con procesos, utilizando 2 archivos auxiliares ajenos a los generados por rpcgen (*svc_types.h* y *svc_types.c) y* editando el Makefile.
En este caso específico, a diferencia del agregado de concurrencia con procesos, se hace uso de la opción -M para generar código seguro. Además de un correcto manejo de las variables, con esta opción se agrega una rutina de liberación de código en el archivo de implementación del servidor (_server.c) que invoca a xdr_free.
Se dejó esta funcionalidad si el protocolo elegido es TCP, para compensar su problema en la liberación de memoria. Para UDP, al no haber problemas, se mantiene la estructura del funcionamiento de rpcgen en el archivo de implementación pero la invocación a xdr_free es eliminada, para evitar errores de liberación múltiple; es decir la posible liberación múltiple del mismo espacio de memoria (recordemos que para UDP svc_freeargs se mantiene).
Se agrega como parámetro de generación del código fuente que el programador indique explícitamente el protocolo de transporte a utilizar (TCP o UDP).
Partiendo de los archivos que rpcgen genera, en la Figura 6.7 se detallan cuáles de ellos son modificados por nuestra versión y cuáles son creados por ésta. Tales modificaciones son transparentes al programador y como puede verse, lo que se modifica es solamente el funcionamiento del servidor (nuevamente, con la pequeña excepción de que cuando el protocolo elegido es TCP el cliente debe modificarse para que realice este tipo de llamadas).





Donde:

Son archivos generados por rpcgen y modificados por nosotros.

Son archivos generados por nosotros.

Figura 6.7. Archivos que intervienen en rpcgenmt

Aplicando el ejemplo visto para multi-proceso a esta versión con threads, podemos ver que el funcionamiento es idéntico. El archivo de especificación puede verse en la Figura 6.8, donde en relación al del ejemplo anterior solamente se cambia el nombre del programa y del archivo.

```
/* listhread.x */

typedef string texto<>;

struct nodo {
        int x;
        nodo *sig;
};

struct parametro {
        texto nombre;
        nodo* lista;

};

program LISTA_THREAD {
    version LISTA_VERSION {
     int sum(parametro) = 1;
   } = 1;
} = 0x20000001;
```

Figura 6.8. Archivo de especificación MT

La generación y compilación de los archivos no se modifica y se grafica en la Figura 6.9.

```
./rpcgenmt listhread.x -t
make -f Makefile.listhread
```

Figura 6.9. Generación y compilación con rpcgenmt

Ahora, el servidor delegará cada solicitud a un nuevo thread. Para una mejor comprensión, modificamos el servidor para que imprima el ID de cada nuevo thread generado.

En la Figura 6.10 podemos ver la concurrencia en el procesamiento del servidor y en la Figura 6.11 la invocación de cada cliente.
Finalmente, en la Figura 6.12 se muestra una captura de los threads activos durante la ejecución de este ejemplo; pudiendo verificarse que existen 3 (uno por cada cliente) cuyos ID se corresponden con los creados por el servidor.

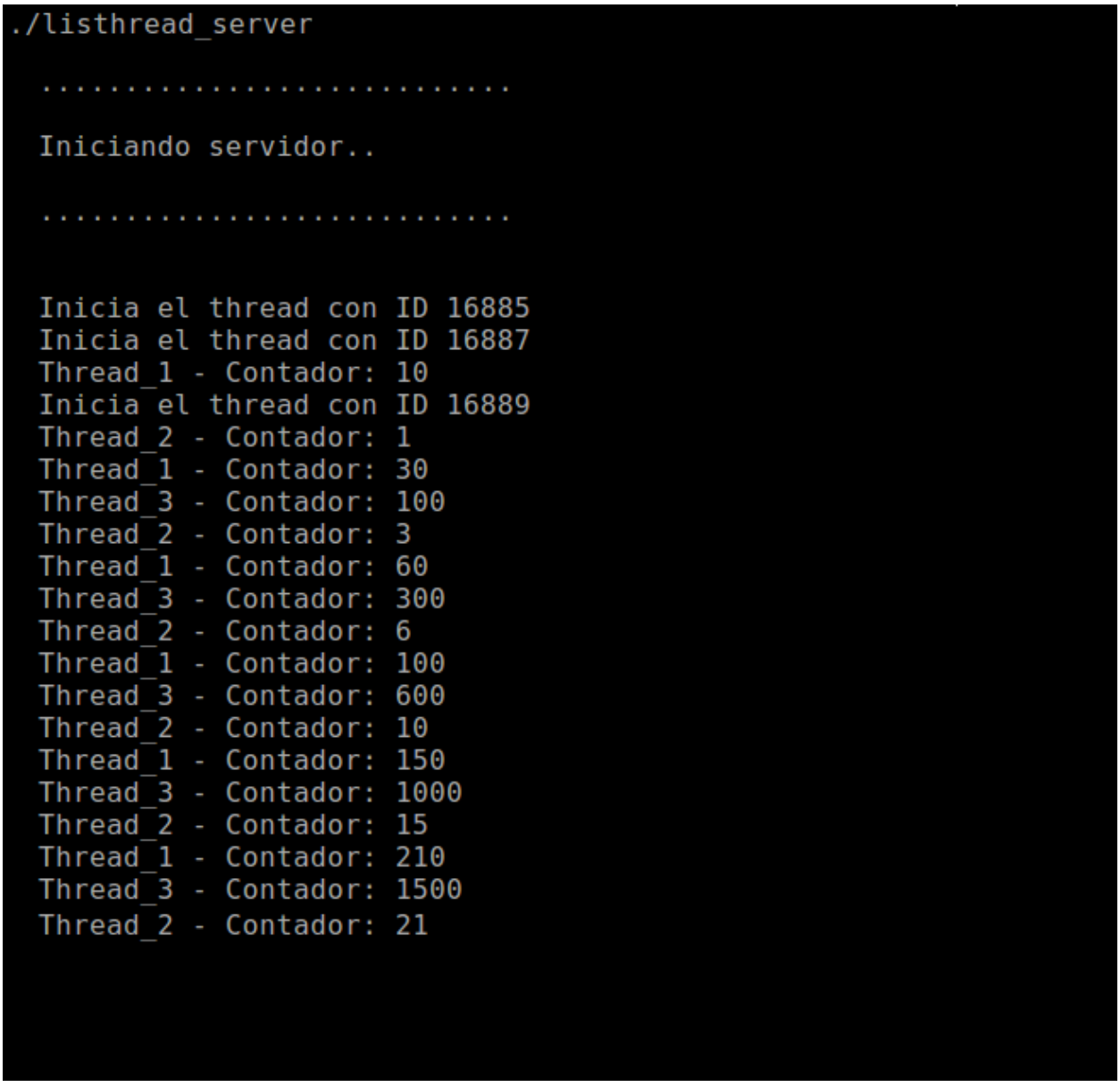


Figura 6.10. Ejecución del servidor MT

```
./listhread_client localhost Thread_1 10 20 30 40 50 60
 Iniciando Thread_1..
 Lista ingresada: 10 20 30 40 50 60
 Suma de la lista: 210
```

Cliente 1

```
./listhread_client localhost Thread_2 1 2 3 4 5 6
  Iniciando Thread_2..
  Lista ingresada: 1 2 3 4 5 6
  Suma de la lista: 21
```

Cliente 2

```
./listhread_client localhost Thread_3 100 200 300 400 500
 Iniciando Thread_3..
 Lista ingresada: 100 200 300 400 500
 Suma de la lista: 1500
```

Cliente 3

Figura 6.11. Ejecución de los clientes MT

```
ps -fLe | grep listhread_server
UID          PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
root     16882 16545 16882  0    1 21:41 pts/2    00:00:00 sudo ./listhread_server
root     16883 16882 16883  0    4 21:41 pts/2    00:00:00 ./listhread_server
root     16883 16882 16885  0    4 21:41 pts/2    00:00:00 ./listhread_server
root     16883 16882 16887  0    4 21:41 pts/2    00:00:00 ./listhread_server
root     16883 16882 16889  0    4 21:41 pts/2    00:00:00 ./listhread_server
```

Figura 6.12. Monitoreo para MT

7. Conclusiones

Durante este trabajo estudiamos los conceptos fundamentales de la programación distribuida para tratar de entender el funcionamiento del mecanismo RPC.
De esta forma, analizamos la secuencia necesaria para reemplazar invocaciones a funciones locales por llamadas a procedimientos remotos, focalizándonos en la implementación RPC de Sun para la plataforma Linux. En base a ésta se fueron detallando las problemáticas que la programación distribuida presenta y las técnicas encontradas para su solución, lo que nos llevó a estudiar la relación de RPC con los protocolos de transporte, la forma en que los procedimientos son localizados en la red, la serialización de datos para interactuar con sistemas de diferentes características y los mecanismos de tiempos de espera y retransmisión.
Luego se investigó el compilador rpcgen en Linux, a partir del cual un programador puede desarrollar una aplicación distribuida especificando solamente las funcionalidades a utilizar y agregando la lógica de negocio necesaria. A pesar de ser una herramienta de gran ayuda, debido al trabajo de soporte y bajo nivel que realiza, señalamos su imposibilidad para generar un servidor con procesamiento concurrente.
A partir de allí, intentamos encontrar una solución teórica para aplicar concurrencia haciendo uso de procesos y de threads, tomando como base los archivos que el compilador genera e intentando encontrar patrones de comportamiento en dicho proceso sobre los cuales añadir lógica. Durante dicho desarrollo, fueron surgiendo inconvenientes como la duplicación UDP, las funcionalidades que no podían usarse con TCP y las fallas de las respuestas UDP.
Teniendo como contrapunto la escasa documentación existente sobre la versión de Sun RPC para Linux, dimos solución a tales problemas para poder mantener un servidor concurrente y estable para ambos protocolos. Para tales fines, utilizamos estructuras auxiliares en el servidor para filtrar duplicaciones y permitir comunicación interna; y se modificó la manera en que los procedimientos remotos son finalmente invocados, dependiendo del protocolo de transporte que se esté utilizando, para evitar los escenarios en donde se generaban conflictos.
Finalmente, se plasmó lo analizado en nuestras propias versiones de rpcgen (una para procesos y otra para threads). En ellas, se automatiza la generación de código necesario para mantener las características de Sun RPC y posibilitar concurrencia en el lado del servidor. Esta generación de código es tan transparente para el programador como la herramienta rpcgen original misma.
El programador ahora puede usar una extensión de rpcgen para Linux que brinda concurrencia mediante procesos y threads, donde la decisión entre una u otra dependerá de las características de la aplicación que se quiera implementar.

Referencias

[1] Oracle, ONC+ Developer's Guide, December, 2002 http://docs.oracle.com/cd/E18752_01/html/816-1435/index.html
[2] R. Thurlow, RFC 5531, RPC: Remote Procedure Call Protocol Specification Version 2, May 2009, http://tools.ietf.org/html/rfc5531
[3] G. Coulouris, J. Dollimore, T. Kindberg, Distributed Systems: Concepts and Design, 4th Ed., Addison Wesley, 2005
[4] A. S. Tanenbaum, M. van Steen, Distributed Systems: Principles and Paradigms, 2nd Ed., Prentice Hall, 2007
[5] M. L. Liu, Distributed Computing: Principles and Applications, Addison-Wesley, 2004
[6] G. Andrews, Foundations of Multithreaded, Parallel, and Distributed Programming, Addison-Wesley , 1999
[7] J. F. Kurose, K. W. Ross, Computer Networking: A Top-Down Approach, 6/E, ISBN-10: 0132856204, Addison-Wesley, 2013
[8] W. R. Stevens, S. A. Rago, Advanced Programming in the UNIX Environment, 3rd Ed., Addison-Wesley Professional, 2013, ISBN-10: 0321637739.
[9] W. R. Stevens, B. Fenner, A. M. Rudoff, Unix Network Programming, Volume 1: The Sockets Networking API, 3rd Ed., Addison-Wesley Professional, 2003, ISBN-10: 0131411551.
[10] W. W. Gay, Linux Socket Programming by Example, April 2000, ISBN-10: 0789722410
[11] D. R. Butenhof, Programming with POSIX Threads, Addison-Wesley Professional, 1997, ISBN-10: 0201633922
[12] D. E. Comer, D. L. Stevens, Internetworking With TCP/IP, Vol III: Client-Server Programming And Applications, Linux/Posix Sockets Version, Addison-Wesley, 2000, ISBN-10: 0130320714
[13] R. Gardner, S. D'Angelo, M. Sears, Years Later: Still Improving the Correctness of an NFS Server, 2009, https://www.kernel.org/doc/ols/2009/ols2009-pages-95-100.pdf
[14] Y. Liu, A survey Of Issues in Remote Procedure Call, 2006, http://sydney.edu.au/engineering/it/research/tr/tr422.pdf
[15] B. Liskov, L. Shrira, J. Wroclawski, Efficient At-Most-Once Messages Based on Synchronized Clocks, http://groups.csail.mit.edu/ana/Publications/PubPDFs/Efficient%20At-Most-Once%20Messages%20Based%20on%20Synchronized%20Clocks.pdf
[16] ELEN 602: Computer Communications and Networks, Fall semester-2004, Remote Procedure Calls (RPC), http://www.ece.tamu.edu/~reddy/ee602_00/rpc_04.pdf
[17] A. Birrell, B. Nelson, Implementing Remote Procedure Call, http://www.cs.princeton.edu/courses/archive/fall03/cs518/papers/rpc.pdf
[18] The Open Group, Remote Procedure Calls: Protocol Specification, 1998, http://pubs.opengroup.org/onlinepubs/9629799/chap4.htm
[19] R. Srinivasan, RPC: Remote Procedure Call Protocol Specification, Version 2, 1995, http://www.ietf.org/rfc/rfc1831.txt
[20] R. Haden, Remote Procedure Call, http://www.rhyshaden.com/rpc.htm
[21] D. Purdue, Transparency and Performance Issues in Sun RPC, http://classic.auug.org.au/publications/auugn/vol16no1/sunPaper.html

[22] Hewlett-Packard Company, HP TCP/IP Services for OpenVMS - ONC RPC Programming, 2005, http://h71000.www7.hp.com/doc/82final/6528/ba548-90003.pdf
[23] W. R. Stevens, Unix Network Programming, Volume 2: Interprocess Communications, 2nd Ed., Prentice Hall PTR, 1999, ISBN 0-13-081081-9.
[24] rpc_svc(3) – Library routines for ONC server remote procedure calls, http://www.tru64unix.compaq.com/docs/base_doc/DOCUMENTATION/V51B_HTML/MAN/MAN3/2467____.HTM
[25] B. Callaghan, B. Pawlowski, P.Staubach, NFS Version 3 Protocol Specification, 1995, https://www.ietf.org/rfc/rfc1813.txt
[26] RPC Programming Guide, http://eia.udg.edu/~teo/sd/documents/rpc/rpcgen%20programming%20guide.htm
[27] A. S. Tanenbaum, Sistemas Operativos Modernos, 3rd Ed., Pearson Educación,, 2009, ISBN: 978-607-442-046-3
[28] D. Keromytis, Randomness Used Inside the Kernel, 1999, https://www.usenix.org/legacy/publications/library/proceedings/usenix99/full_papers/deraadt/deraadt_html/node18.html

Apéndice A. XDR

1. Definiciones

Un archivo RPCL consiste en una serie de definiciones:

```
definition-list:
    definition ";"
    definition ";" definition-list
```

Donde una definición puede ser de alguno de los siguiente tipos:

```
definition:
    enum-definition
    typedef-definition
    const-definition
    declaration-definition
    struct-definition
    union-definition
    program-definition
```

2. Enumeraciones

Las enumeraciones XDR tienen la misma sintaxis que las de C:

```
enum-definition:
    "enum" enum-ident "{"
    enum-value-list
    "}"
enum-value-list:
    enum-value
    enum-value "," enum-value-list
enum-value:
    enum-value-ident
    enum-value-ident "=" value
```

El siguiente ejemplo define una enumeración con 3 valores:

```
enum colortype {
```

```
    RED = 0,
    GREEN = 1,
    BLUE = 2
};
```

Este código al ser compilado produce el siguiente resultado (en C):

```
enum colortype {
    RED = 0,
    GREEN = 1,
    BLUE = 2
};
typedef enum colortype colortype;
```

3. Definiciones de tipo

Las definiciones de tipos XDR tienen la misma sintaxis que las de C:

```
typedef-definition:
    "typedef" declaration
```

El siguiente ejemplo define una cadena de caracteres con una longitud máxima de 255:

```
typedef string fname_type<255>;
```

Este código al ser compilado produce el siguiente resultado:

```
typedef char *fname_type;
```

4. Constantes

Las constantes XDR pueden utilizarse donde sea que se requiera una constante entera (por ejemplo en la especificación del tamaño de un arreglo). Su definición es:

```
const-definition:
    "const" const-ident "=" integer
```

El siguiente ejemplo define una constante DOCE igual a 12:

```
const DOCE = 12;
```

Este código al ser compilado produce el siguiente resultado:

```
#define DOCE 12
```

5. Declaraciones

XDR proporciona 4 tipos de declaraciones:

declaration:
simple-declaration
fixed-array-declaration
variable-array-declaration
pointer-declaration

Para cada una, su sintaxis seguida de un ejemplo:

A. Declaración simple

simple-declaration
type-ident variable-ident

Por ejemplo:

colortype color en XDR
colortype color en C

B. Declaración de arreglos de longitud fija

fixed-array-declaration:
type-ident variable-ident “[” value “]”

Por ejemplo:

colortype palette[8] en XDR
colortype palette[8] en C

C. Declaración de arreglos de longitud variable

Estos no tienen una sintaxis explícita en C, siendo la de XDR:

variable-array-declaration:
type-ident variable-ident “<" value “>”
type-ident variable-ident “<" “>”

El tamaño máxima puede ser o no detallado. En caso de ser omitido se está indicando que el arreglo puede ser de cualquier tamaño. Por ejemplo:

```
int heights<12>; /* a lo sumo 12 ítems */
int widths<>;    /* cualquier número de ítems */
```

Al no tener sintaxis definida en C, rpcgen genera estructuras para su representación. Por ejemplo, para el caso de *heights*:

```
struct {
        u_int heights_len;   /* número de ítems */
        int *heights_val;    /* puntero al arreglo */
} heights;
```

Donde el componente *_len* almacena la cantidad de ítems en el arreglo y *_val* un puntero al mismo. La primera parte del nombre de estos componentes es el de la variable declarada para XDR.

D. Declaración de un puntero

Estos son idénticos a los de C. No se pueden enviar punteros a través de la red (recordar que los espacios de direcciones son diferentes, por lo que no tiene sentido) pero se pueden usar punteros XDR para enviar y recibir tipos de datos recursivos, como listas o árboles. En XDR este tipo recibe el nombre de *optional-data,* cuya sintaxis:

```
optional-data:
        type-ident "*" variable-ident
```

Por ejemplo (es igual tanto para XDR como para C):

```
        listitem *next;
```

6. Estructuras

Las estructuras XDR se declaran de forma muy similar a C:

```
struct-definition:
    "struct" struct-ident "{"
    declaration-list
    "}"
declaration-list:
    declaration ";"
    declaration ";" declaration-list
```

El siguiente ejemplo define una estructura con 2 componentes:

```
struct coord {
    int x;
    int y;
};
```

Este código al ser compilado produce el siguiente resultado:

```
struct coord {
     int x;
     int y;
};
typedef struct coord coord;
```

Como puede verse, la salida es idéntica a la entrada excepto en que se la agrega la definición del final para habilitar la utilización de *coord* en lugar de *struct coord* en las declaraciones.

7. Uniones

Las uniones XDR son discriminadas y más análogas a los registros variantes de Pascal en vez de a las uniones de C:

```
union-definition:
     "union" union-ident "switch" "(" simple-declaration ")" "{"
       case-list
     "}"

case-list:
     "case" value ":" declaration ";"
     "case" value ":" declaration ";" case-list
     "default" ":" declaration ";"
```

El siguiente es un ejemplo de un tipo que puede ser retornado como resultado de una lectura de datos:

```
union read_result switch (int errno) {
          case 0
                    opaque data[1024];
          default:
                    void;
};
```

Este código al ser compilado produce el siguiente resultado:

```
struct read_result {
    int errno;
    union {
```

```
        char data[1024];
    } read_result_u;
};
typedef struct read_result read_result;
```

Notar que el nombre de la unión de la estructura resultante es idéntico el nombre de la declarada en XDR, excepto por el sufijo *_u*.

8. Programas

Los programas RPC se declaran utilizando la siguiente sintaxis:

```
program-definition:
     "program" program-ident "{"
          version-list
     "}" "=" value

version-list:
     version ";"
     version ";" version-list

version:
     "version" version-ident "{"
          procedure-list
     "}" "=" value

procedure-list:
     procedure ";"
     procedure ";" procedure-list

procedure:
     type-ident procedure-ident "(" type-ident ")" "=" value
```

El siguiente ejemplo especifica un programa simple de consulta y asignación de horario:

```
program TIMEPROG {
    version TIMEVERS {
        unsigned int TIMEGET (void) = 1;
        void TIMESET (unsigned) = 2;
    } = 1;
} = 0x20000001;
```

En el archivo de cabecera resultante, al compilar se generan las siguientes definiciones:

```
#define TIMEPROG 0x20000001
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

9. Casos especiales

Existen ciertas excepciones en las reglas sintácticas definidas:

A. Booleans

C no tiene incorporado un tipo de dato boolean. Sin embargo, la librería RPC tiene uno conocido como *bool_t*. rpcgen compila entonces las declaraciones de tipo *bool* encontradas en XDR en *bool_t* en el archivo de cabecera resultante.

B. Strings

C no tiene incorporado un tipo de dato string. En su lugar utiliza un puntero hacia una cadena de caracteres finalizada con un valor nulo (char *). En XDR se declaran strings haciendo uso de la palabra clave *string.* rpcgen compila cada ocurrencia de *string* en *char *.*

C. Opaque

RPC y XDR utilizan el tipo de datos opaque para describir datos sin tipo (es decir, una secuencia de bytes). Se declaran como arreglos de longitud fija o variable, cuya traducción al compilar es:

- Longitud fija en XDR

Declaración: `opaque diskblock[512];`

Después de compilar: `char diskblock[512];`

- Longitud variable en XDR

Declaración: `opaque filedata<1024>;`

Después de compilar:

```
struct {
        u_int filedata_len;
        char *filedata_val;
} filedata;
```

D. Void

Cuando se declara un void la variable no se nombra. Estas declaraciones pueden encontrarse solamente en uniones y definiciones de programa (como argumento o resultado de un procedimiento remoto).