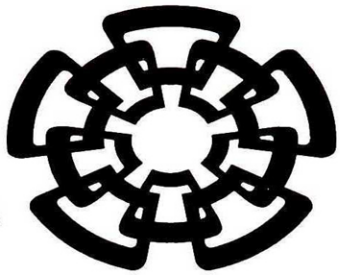


Centro de Investigación y de Estudios Avanzados del I.P.N.

Departamento de Ingeniería Eléctrica

Sección de Computación

COMPILADOR DEL PROTOCOLO RPC PARA LA PROGRAMACIÓN DE APLICACIONES DISTRIBUIDAS

Tesis que presenta la Lic. Graciela Judith Esparza Azcoitia para obtener el grado de MAESTRO EN CIENCIAS dentro de la especialidad de INGENIERÍA ELECTRICA con opción en COMPUTACIÓN

Trabajo dirigido por el M. en C. Raúl García Ruiz

México, D.F., Agosto de 1997

Becario de CONACYT

AGRADECIMIENTOS

A mis padres, a mis hermanos y a Miguel porque hemos logrado estar unidos, a pesar de las distancias.

A mis compañeros de trabajo y amigos que siempre me han apoyado.

Al personal y a los profesores de las secciones de Computación, Comunicaciones así como la Secretaría de Planeación porque me brindaron su ayuda y colaboraron en mi formación académica.

Al CINVESTAV y al Consejo Nacional de Ciencia y Tecnología (Conacyt) por el apoyo económico brindado.

CONTENIDO

CAPITULO

1

INTRODUCCION

1.

1. INTRODUCCIÓN

1.1. Objetivo de la tesis

En la actualidad, las redes de computadoras son un medio de comunicación elemental que nos permiten intercambiar información y compartir recursos, como discos, cintas, archivos, bases de datos, módems, impresoras, plotters y otros dispositivos. Las redes permiten a los usuarios estructurar tales recursos de forma que se adapten a su organización, promoviendo una mejor distribución y la descentralización de los mismos.

En este contexto, es importante que los usuarios de los servicios tengan acceso a los recursos distribuidos sin solicitar de manera explícita la transacción requerida para usarlos, en otras palabras, tener acceso a los servicios remotos de forma transparente. Los programas de aplicación que soportan esta capacidad son conocidos como *aplicaciones o sistemas distribuidos* y su objetivo es proporcionar un medio ambiente que oculte la localización geográfica de las computadoras y servicios, de tal manera que parezcan locales.

La programación de aplicaciones distribuidas puede resultar muy compleja ya que involucra el conocimiento de diferentes redes, familias de protocolos, interfaces, estándares para la representación de datos, etc. Debido a esto, los programadores han tenido la necesidad de una plataforma[i] de fácil uso para escribir aplicaciones distribuidas. Una de estas plataformas es "Open Network Computing" (ONC) desarrollada por SUN MicroSystems. Esta plataforma consta de rutinas para llamadas a procedimientos remotos (Remote Procedure Call -RPC-) y rutinas para la representación de los datos (eXternal Data Representation -XDR-).

Por otra parte, la plataforma ideal para las aplicaciones distribuidas no debe asociarse con un Sistema Operativo o una arquitectura específica, por lo que debe ser aceptada por varios fabricantes, constituyéndose en un estándar de industria como es el caso de ONC de SUN.

Con base en estos antecedentes se planteó como el objetivo fundamental de esta tesis el desarrollo de una herramienta que facilitara la programación de aplicaciones distribuidas, que funcionara bajo el entorno del sistema Operativo MS-DOS y a la vez fuera compatible con la plataforma ONC para establecer sesiones con las estaciones de trabajo de SUN o cualquier otra estación similar.

En las secciones 1.2 y 1.3 de este capítulo, se citan los antecedentes y conceptos que enmarcan el desarrollo de las aplicaciones distribuidas y su entorno general, así como los motivos que impulsaron a la elaboración de esta tesis, para entonces explicar el problema a nivel general y las consideraciones que se tomaron para el desarrollo de una herramienta de este estilo. En la sección 1.4 se explican los resultados obtenidos,

---

[i] Una *plataforma para aplicaciones distribuidas* la podemos definir como un conjunto de rutinas que proporcionan funciones específicas para el desarrollo de aplicaciones que involucran el intercambio de información entre dos o más entidades distribuidas

los módulos de programación que comprende esta tesis y las herramientas que se utilizaron para el desarrollo de la misma.

En el capítulo 2 se explican el modelo RPC, productos comerciales que lo emplean y el modelo RPC de SUN. En el capítulo 3 se especifican las bibliotecas RPC y XDR elaboradas para la creación de aplicaciones distribuidas, posteriormente se definen la generación de stubs, encabezados y rutinas XDR. En el capítulo 4 se menciona la interfaz empleada para la generación de las rutinas RPC y XDR. En el capítulo 5 se explican los módulos que constituyen el compilador CRAD, así como parte del código para los programas generador de compiladores y generador de análisis léxico empleados. En el capítulo 6 se presenta un ejemplo sencillo del empleo de este compilador. El apéndice A explica el modelo XDR, el apéndice B el modelo cliente - servidor y el apéndice C el modelo RPC.

1.2. Antecedentes

En la década de los setenta proliferaron las redes de computadoras. La red ARPA (Advanced Research Projects Agency) fue la primera y enlazaba a importantes centros de cómputo de los Estados Unidos con fines de investigación y desarrollo en tecnología de conmutación de paquetes. Después aparecieron redes privadas, de servicio público, unas a nivel internacional y nacional, otras instaladas en una área local.

Debido a la gran necesidad de comunicación entre las computadoras así como la de desarrollar sistemas más versátiles, eficientes y funcionales, aparecieron los sistemas de multiprocesamiento y de procesamiento distribuido. Por lo tanto se requirió de intercomunicación entre procesos, ejecución de comandos remotos, acceso a bases de datos remotas, comunicación a distancia, intercambio de mensajes y gráficas, etc. Así, el desarrollo de sistemas de comunicación de datos se ha convertido en una área de investigación y de desarrollo prioritaria.

Para el desarrollo de un sistema de comunicación de datos se requieren reglas para establecer un diálogo e intercambiar información entre los equipos de cómputo, a esto se le ha denominado protocolo de comunicación. Debido a la complejidad de la comunicación entre los sistemas de procesamiento de datos, no es posible ejecutar todas las tareas en un solo protocolo. Por lo que es usual que se apliquen simultáneamente varios protocolos de comunicación de datos.

El movimiento de la información entre las computadoras heterogéneas es una tarea importante. Cerca de los años 80 la Organización Internacional para la Estandarización (International Organization for Standarization -ISO-) reconoció que existía la necesidad de usar un modelo para la comunicación entre sistemas (computadoras). En 1984 se dio a conocer el modelo OSI (Open System Interconnection).

El modelo de referencia OSI (de interconexión de sistemas abiertos) ha permitido estructurar y jerarquizar adecuadamente las funciones de comunicación. El problema de mover información entre computadoras se divide entre 7 capas. Cada capa ofrece sus servicios a la capa inmediata superior y usa los servicios de la capa inmediata inferior. Dentro de cada capa N, una o varias entidades desarrollan las funciones asignadas a este nivel y se comunican con entidades similares de otro sistema por medio de un protocolo de capa N. El resultado final es una familia de protocolos estructurados verticalmente.

Para 1975 ARPA tenía tantos usuarios que dejó de ser de una red experimental y la responsabilidad para su operación fue transferida a la Agencia de Comunicaciones de Defensa de los EE.UU. Entonces DARPA (Defense Advanced Research Project Agency)

propuso una red de conmutación de paquetes para establecer comunicación entre las diversas instituciones en los Estados Unidos. DARPA y otras organizaciones gubernamentales entendieron el gran potencial que esta tecnología podría proporcionar. Con la meta de conectar sistemas heterogéneos, y ante la ausencia (en ese tiempo) de un modelo de comunicación y una familia de protocolos que resolviera dicho problema, DARPA, junto con la Universidad de Stanford, crearon un conjunto de protocolos de comunicación cuyo resultado fue la familia de protocolos TCP/IP de Internet, de los cuales el protocolo de control de Transmisión (Transmission Control Protocol: TCP) y el protocolo Internet (Internet Protocol: IP) son los miembros mas conocidos. [Stal91].

La familia de protocolos TCP/IP se utiliza para comunicación a través de cualquier conjunto de redes interconectadas o internet. Esta familia de protocolos también incluye protocolos para correo electrónico, emulación de terminales y transferencia de archivos, entre otros.

Mostramos en la figura 1-1 algunos de los protocolos internet más importantes y su relación con el modelo OSI. [cisco93].

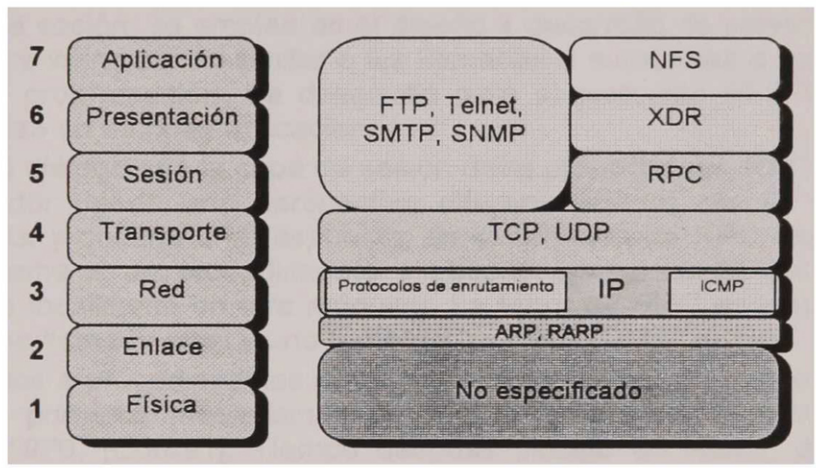


Figura 1-1. Conjunto de protocolos Internet y Modelo de Referencia OSI

Adicionalmente a estos protocolos, en la figura 1-2 mostramos los correspondientes de NetWare, creados por Novell Inc. e introducidos cerca de los 80. Este conjunto de protocolos fue derivado de Xerox Network Systems (XNS) y orientado a LAN.

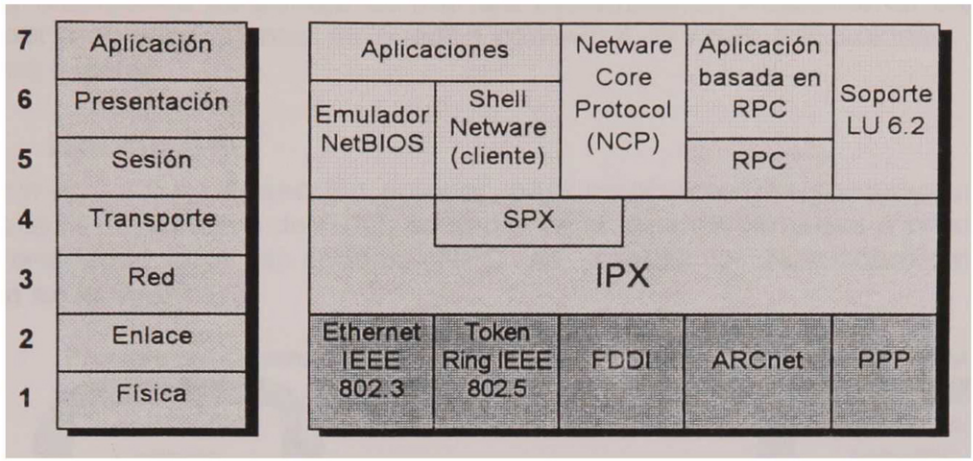


Figura 1-2. Netware y el Modelo de Referencia OSI

En los dos casos se presenta el modelo RPC (Remote Procedure Call). RPC es un modelo de la capa de sesión, se emplea en el diseño y desarrollo de servicios de red, el cual, como su nombre lo indica, es similar a las llamadas a subrutinas o procedimientos en los lenguajes de programación. Se desarrolló para el protocolo NFS (Network File System), pero se utiliza en muchas aplicaciones.

Para establecer el diálogo que la capa de sesión debe proporcionar, RPC desarrolla el modelo cliente-servidor desde una perspectiva diferente. Si un cliente transmite un mensaje a un servidor y obtiene una respuesta, en el contexto de RPC esto equivale a un programa que llama a un procedimiento y obtiene un resultado. En tal caso, el procedimiento puede localizarse en otra máquina. La tarea de RPC es ayudar a ocultar la diferencia entre una llamada local y una remota.

La idea de llamadas a procedimientos remotos se ha considerado desde hace mucho tiempo. Uno de los primeros mecanismos de RPC lo desarrolló Jim White, cuando trabajó en SRI en 1970. [Corb91]. Tiempo después trabajó en Xerox, donde realizó varios proyectos de investigación relacionados con redes. En los siguientes años se desarrollaron varios mecanismos de RPC en Xerox, pero solamente uno, *curier* , fue lanzado como producto en 1981. La implantación de *curier* la realizó Alan Frieier. Bob Lyon de la compañía Sun Microsystems, desarrolló en 1985 la versión de RPC de SUN y el "Network File System" (NFS). NFS usa el mecanismo RPC para proporcionar acceso transparente de los archivos en un medio ambiente distribuido. NFS inició la industria de estándares de facto para los sistemas de archivos distribuidos. La flexibilidad de la implantación de Lyon se ha confirmado por el número de plataformas RPC que se han portado.

Debido a que muchos modelos de referencia emplean RPC y que es una alternativa para facilitar la programación de aplicaciones distribuidas, surgió la inquietud de investigar el funcionamiento de este modelo, así como de desarrollar un software para las computadoras personales. Basados en la plataforma ONC de SUN y en el compilador de RPC para las estaciones de trabajo, se diseñó una plataforma y un compilador para las computadoras personales conservando compatibilidad entre ellos.

Al iniciar este trabajo ya existían algunas implantaciones y cierta información referente al tema. Pero el deseo de involucrarse y de conocer a fondo este tema para aportar un software en esta institución, despertó un gran interés que llevó a la conclusión de que es

una herramienta de gran ayuda al permitir establecer sesiones entre computadoras personales y estaciones de trabajo de manera transparente. Actualmente existe mucha literatura y por medio de Internet se pueden conseguir muchas aplicaciones, incluyendo los programas fuentes.

1.3. Modelo RPC

Como se mencionó en la sección anterior, para establecer una comunicación cliente-servidor desde la perspectiva de RPC, se implanta la idea de llamadas a procedimientos remotos. Para desarrollar el modelo RPC se analizó el funcionamiento que se esquematiza en la figura 1-3.

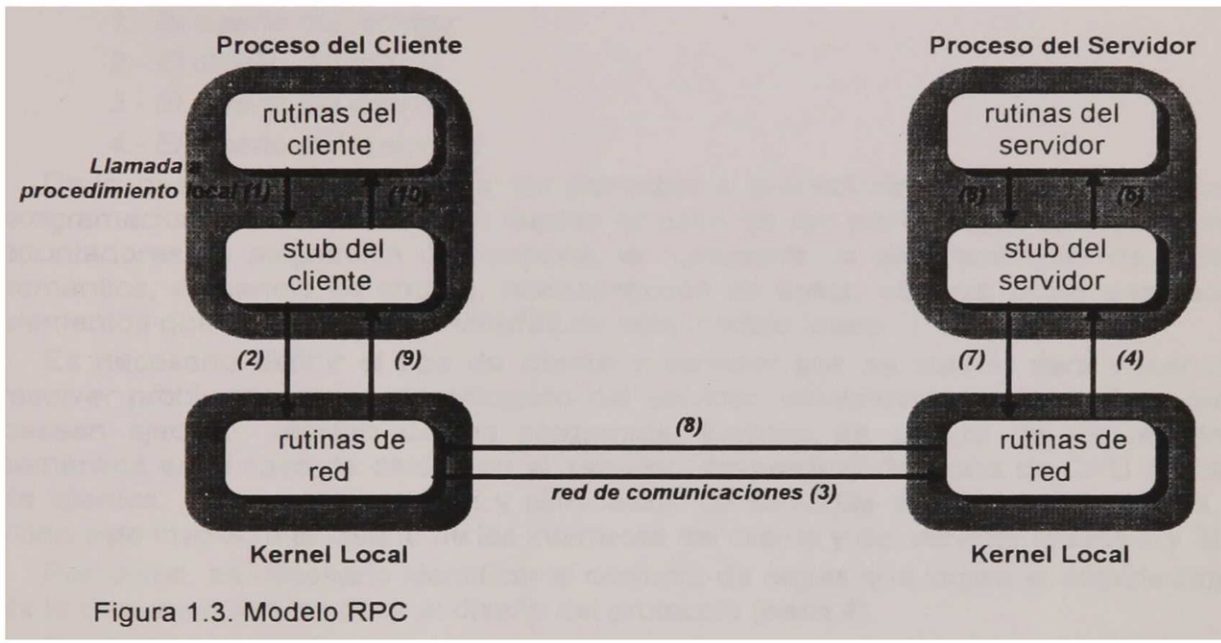


Figura 1.3. Modelo RPC

RPC se basa en el modelo cliente-servidor, en donde el cliente solicita un servicio y el servidor lo atiende. Para representar esto, RPC utiliza un procedimiento conocido como "stub", que se encarga de ocultar los detalles de comunicación a través de la red además de representar el papel del procedimiento solicitado. El funcionamiento general de RPC se describe a continuación:

Como primer paso, el programa cliente solicita que se ejecute un procedimiento **(paso 1),** el stub recibe tal solicitud, recolecta los parámetros y los empaqueta en un mensaje. A continuación, el mensaje se envía a la capa de transporte para su transmisión ***(paso 2).***

El transporte se encarga de manejar ese mensaje enviándolo a través de la red hacia el servidor ***(paso 3)***.

Cuando el mensaje llega al servidor solicitado, la capa de transporte envía el mensaje o solicitud al stub del servidor ***(paso 4)***, que se encargará de desencapsular los parámetros y determinar el procedimiento que se ejecutará y hará la llamada a dicho procedimiento ***(paso 5)***.

Las rutinas del servidor atienden la solicitud, se ejecuta la rutina y se regresa el resultado al stub del servidor ***(paso 6)***.

Al recibir los resultados el stub del servidor, los encapsula y genera un mensaje que se envía a la capa de transporte, para que los transmita ***(paso 7)***.

La capa de transporte transmite los resultados hacia el cliente ***(paso 8)***.

La capa de transporte de la máquina cliente recibe el mensaje y lo transmite al stub del cliente ***(paso 9)***, éste desencapsula los resultados y los envía al programa cliente que hizo la llamada ***(paso 10).***

Con esta secuencia de pasos el cliente no se da cuenta de que la llamada al procedimiento es remota.

Para la implantación de este modelo, nos enfrentamos a diversas consideraciones como:

1.- El diseño del interfaz

2.- El diseño del cliente

3.- El diseño del servidor

4.- El diseño del protocolo

Dado que RPC es análogo a las llamadas a procedimientos en los lenguajes de programación, se debe tomar en cuenta el paso de los parámetros, el manejo de los apuntadores, la asignación de memoria, el transporte, la generación de los stubs, la semántica, el manejo de errores, representación de datos, etc., así como una serie de elementos que figuran como la interfaz de este modelo (paso 1).

Es necesario definir el tipo de cliente y servidor que se quiere, para saber cómo resolver problemas como identificación del servidor, identificación de la rutinas que se desean ejecutar, registro de los programas, tiempos de espera de los resultados, semántica en el caso de caídas en el servidor, desperdicio de ciclos de CPU en caídas de clientes, bloqueo de archivos y eliminación de llamadas en caídas de clientes, etc. Todo esto involucra el diseño de las interfaces del cliente y del servidor (pasos 2 y 3).

Por último, es necesario identificar el conjunto de reglas que logran el establecimiento de la comunicación, es decir el diseño del protocolo (paso 4).

Como se puede observar, hacer una llamada a un procedimiento remoto implica la transformación de esta llamada en llamadas al sistema, conversión de datos, comunicación entre redes. Resumiendo las consideraciones de diseño deben tomar en cuenta:

- *Paso de parámetros*
- *Enlace (binding)*
- *Protocolo de Transporte*
- *Manejo de excepciones*
- *Semántica de llamadas*
- *Representación de datos*
- *Ejecución*
- *Seguridad*

Dado que la representación de los datos depende de la arquitectura de la computadora[ii], para que exista comunicación entre entidades heterogéneas, debemos considerar un estándar para la representación de datos externos. El estándar propuesto

---

[ii] Algunas computadoras almacenan el byte menos significativo de un entero en la dirección de memoria más baja, otras almacenan el byte más significativo en las direcciones más bajas y otras no almacenan en bytes contiguos de memoria

por SUN es "eXternal Data Representation" (XDR). Este estándar ayuda a simplificar la comunicación entre un cliente y un servidor empleando un mecanismo uniforme de acceso remoto.

Los datos que se envían entre el cliente y el servidor, deben traducirse a XDR. Las rutinas que ejecutan esta tarea se les conoce como filtros.

Para diseñar una aplicación distribuida, los programadores deben escribir los "stubs" y los filtros. Para facilitar aún más la tarea de la programación distribuida, es necesario un programa generador que automatice mucho del código asociado con RPC y XDR. En este caso hablamos de un compilador del protocolo RPC que genere de manera automática esos filtros y stubs. De esta forma el empleo de RPC para los programadores es aún más transparente.

El problema que se planteó para la tesis, fue la realización de un compilador que facilite la programación distribuida y que cumpla con el modelo RPC . Para el diseño del compilador RPC y de las bibliotecas asociadas, nos basamos en la plataforma ONC de SUN.

1.4. Resultados

En este trabajo de tesis se desarrolló un compilador RPC para computadoras personales basado en el modelo y los estándares de SUN. Se consideró la plataforma ONC de SUN por razones de mercado, ya que muchas compañías han adoptado las especificaciones de SUN en sus sistema.

El compilador desarrollado se le llamó ***CRAD*** (***C***ompilador ***R***PC para ***A***plicaciones ***D***istribuidas).

CRAD requiere de una especificación en el lenguaje RPC. Este lenguaje se basa en el lenguaje rpcl, que maneja el compilador *rpcgen* de SUN. En este lenguaje se indica al compilador las rutinas que se desean ejecutar de manera remota, así como los tipos de datos que estas rutinas envían y regresan. La sintaxis de los tipos de datos y estructuras de este lenguaje es similar a los empleados en el lenguaje C. El código que genera el compilador CRAD es en lenguaje C. Para que un usuario pueda programar aplicaciones distribuidas, sólo tiene que realizar pequeños cambios en sus rutinas y escribir una especificación en el lenguaje RPC, indicando las rutinas y los tipos de datos. La especificación RPC se compila con CRAD, que genera el stub del cliente, el stub del Servidor, y si es necesario los filtros XDR. De esta manera el usuario podrá compilar sus rutinas y las generadas por CRAD para generar los programas cliente y servidor.

Para las personas que deseen desarrollar sus propios stubs de clientes y servidores, pueden emplear directamente las rutinas de las bibliotecas RPC y XDR que se desarrollaron.

En este proyecto de tesis se empleó el compilador C de Microsoft v 6.0 y un Interfaz para la programación de Aplicaciones ("Aplication Programming Interface" [API]) 2.0 de Pathway Wollongon.

CAPITULO

2

Llamadas a procedimientos remotos

2

2. LLAMADAS A PROCEDIMIENTOS REMOTOS

En este capítulo se explican el modelo RPC y los puntos que se consideraron para la implantación de este modelo. Se hace una comparación de los productos comerciales de SUN, Xerox y Apollo que emplean el modelo RPC para programar aplicaciones distribuidas y se muestra un ejemplo de como aplicarlos. El análisis de cada uno de estos producto fue importante para determinar sus ventajas, además de la contribución importante que cada uno tiene. El caso de SUN, se analizará con mayor detalle.

2.1. Modelos Cliente - Servidor

Debido a que el modelo RPC se basa en el modelo cliente - servidor es importante conocer los distintos tipos de clientes y servidores que se pueden diseñar. Para esta tesis se investigó cómo se podría implantar este modelo y las características, ventajas y desventajas de los diversos tipos de clientes y servidores. En el apéndice B se explica el modelo cliente - servidor.

Para el desarrollo del software cliente - servidor fue importante considerar los estándares para establecer comunicación entre las computadoras. Para esto se emplearon los protocolos de transporte de la familia de protocolos TCP/IP. Estos protocolos permiten que el software cliente - servidor pueda elegir 2 tipos de interacciones: una orientada a conexión (protocolo de transporte TCP) y la otra sin conexión (protocolo de transporte UDP).

Una manera de crear una comunicación punto a punto empleando los protocolos de transporte de TCP es a través de una *interfaz de sockets.* Un socket es un mecanismo que proporciona puntos terminales o extremos de comunicación entre procesos independientes. En los protocolos TCP/IP un punto extremo de comunicación consiste de una dirección IP y un número de puerto TCP ó UDP [comer93] [Woll90].

2.2. Estructura General del Modelo RPC

En una llamada a procedimiento remoto (RPC) un proceso local invoca a otro proceso situado en un sistema remoto. La figura 2.1 muestra los pasos que sigue una llamada RPC.

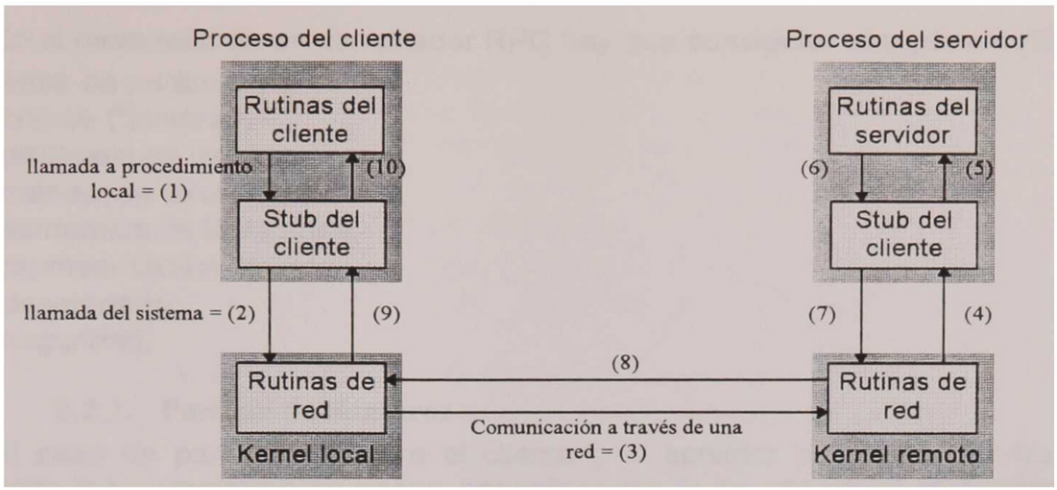


Figura 2-1 Modelo RPC

La secuencia presentada en la figura es la siguiente:

1. El cliente llama a un procedimiento local, denominado *stub del cliente,* quien se encarga de empaquetar los argumentos de la llamada en uno o más mensajes cuyo destino es el servidor.
2. El stub del cliente envía los mensajes al sistema remoto a través de la red, para lo cual hace llamadas a ciertas funciones de comunicación dentro de un kernel local.
3. Los mensajes se transfieren al sistema remoto haciendo uso de un protocolo orientado a conexión o sin conexión.
4. En el sistema remoto, el *stub del servidor* se encuentra esperando peticiones del cliente para desempaquetarlas y convertirlas a otro formato.
5. El stub del servidor realiza una llamada a un procedimiento local para invocar a la función que el cliente solicita. El procedimiento local se llama con los parámetros que el stub del cliente envió en los mensajes.
6. Cuando se ejecuta el procedimiento invocado en el servidor, éste regresa un resultado al stub del servidor.
7. El stub del servidor convierte estos valores y si es necesario los empaqueta en uno o más mensajes para enviarlos al stub del cliente a través de la red.
8. Los mensajes regresan al stub del cliente.
9. El stub del cliente lee los mensajes que provienen de las rutinas de comunicación en el kernel local.
10. Después de convertir los valores regresados, el stub del cliente entrega el resultado a la función original, dando la apariencia de una llamada a una función local.

El concepto de llamadas a procedimientos remotos oculta, por medio de los stubs, todo el código involucrado en la comunicación entre el cliente y el servidor para facilitar la programación de aplicaciones distribuidas.

RPC también se considera parte de la capa de presentación del modelo de referencia OSI ya que incluye especificaciones para el intercambio de argumentos y resultados entre el cliente y el servidor en un formato estándar. Esto se traduce en software que se puede portar a diferentes sistemas por lo que las aplicaciones no tienen que preocuparse por el ordenamiento de bytes entre otras tareas.

El objetivo de RPC es hacer transparente el desarrollo de aplicaciones distribuidas permitiendo la llamada a procedimientos remotos como si se tratara de llamadas a procedimientos locales.

En el desarrollo de un compilador RPC hay que considerar lo siguiente.[Stevens90]:

- paso de parámetros,
- enlace ("binding")
- protocolo de transporte,
- manejo de excepciones
- semántica de la llamada,
- representación de datos,
- desempeño,
- seguridad.

2.2.1. Paso de parámetros

El paso de parámetros entre el cliente y el servidor podría no ser transparente. Cuando los parámetros se pasan por referencia (y no por valor) el servidor no tiene forma de acceso a las localidades de memoria referidas por el cliente.

Una solución típica es permitir únicamente el paso de argumentos por valor. Para cada procedimiento remoto, se definen específicamente los argumentos de entrada y los valores de retorno.

2.2.2. Enlace

Se refiere a la facilidad del cliente para contactar al sistema remoto apropiado donde se ejecutará el procedimiento remoto. Puede haber dos variantes:

a) encontrar una computadora remota para un servidor deseado

b) encontrar el servidor correcto en una computadora determinada.

Para localizar una computadora remota que cuente con un servidor determinado se puede usar una base de datos centralizada que registre a los sistemas y los servidores disponibles para atender clientes remotos. Esto se puede realizar bajo un esquema de intercambio de mensajes entre los servidores y la autoridad central. De esta forma, los clientes deben contactar a esta autoridad para localizar un servicio determinado.

Otra técnica es preguntar al cliente a cuál computadora desea conectarse y contar con un "superservidor" que conozca las direcciones de los servidores disponibles en tal sistema.

2.2.3. Protocolo de transporte

La mayoría de los productos RPC soportan uno o dos protocolos de transporte diferentes, por ejemplo:

RPC de Sun	TCP (orientado a conexión), UDP (sin conexión)
Courier de Xerox	SPP (con conexión)
RPC de Apollo	UDP, DDS (protocolo propietario de Apollo, sin conexión)

Cuando se usa un protocolo sin conexión el stub del cliente debe resolver la pérdida de paquetes. El protocolo orientado a conexión proporciona una comunicación confiable pero con mayor carga de información de control en cada paquete.

2.2.4. Manejo de excepciones

Cuando ocurre algún problema en la comunicación a través de la red, en los stubs del cliente o del servidor o en los procedimientos del servidor, deben tomarse las precauciones apropiadas para evitar que la falla en un extremo afecte el otro extremo.

Por ejemplo, puede darse el caso que el cliente tenga la facultad de detener un proceso que tome demasiado tiempo al ejecutarse en el servidor, o el proceso del cliente termine después de haber invocado un procedimiento remoto pero antes de obtener el resultado (el servidor necesita saber que el cliente ha desaparecido).

2.2.5. Semántica de las llamadas

Se usa para clasificar a los procedimientos remotos de acuerdo al número de veces que se ejecutan ante la presencia de fallas en el servidor o en el sistema de comunicación.

Hay tres formas de semántica RPC:

1. *Exactamente una* significa que el procedimiento remoto se ejecutó una vez.
2. *Máximo una* significa que el procedimiento remoto fue ejecutado una vez o que no lo logró. Si se regresa un valor normal al solicitante, sabemos que el procedimiento remoto se ejecutó una vez; pero si regresa un error, no sabemos con certeza si se completó la ejecución del procedimiento remoto. En otras palabras, si falla el servidor, el stub del cliente renunciará y devolverá un código de error. La retransmisión no se intenta. En este caso el cliente sabe que la operación se efectuó una sola vez o ninguna, pero no más.
3. *Al menos una* significa que el procedimiento remoto al menos se ejecutó una vez, pero posiblemente más. Esto es típico en procedimientos en donde el cliente realiza varias solicitudes hasta que recibe una respuesta válida; pero si el cliente tiene que enviar su solicitud más de una vez para recibir respuesta, existe la posibilidad de que el procedimiento remoto se haya ejecutado más de una vez.

La semántica se complica por las posibles caídas en los servidores , los clientes y el servidor de comunicación. En los sistemas con un solo procesador, no ocurre esta situación, porque una caída que provoca falla al servidor, también provoca falla al cliente.

2.2.6. Representación de los datos

Si los sistemas que soportan el cliente y el servidor tienen una arquitectura y diferente formas de representar a los datos; es necesario realizar una conversión de datos para que éstos se entiendan. Por ejemplo, en algunas plataformas y sistemas se han planteado algunas convenciones como:

- El conjunto de protocolos TCP/IP usa el ordenamiento de octeto (byte) "big-endian" para los campos de 32 y 16 bits en los encabezados del protocolo. Este incluye campos como el identificador (ID) de red y de computadora y el número de puerto UDP de 16 bits.
- El conjunto de protocolos XNS también usa el ordenamiento big-endian para todos los campos de 16 y 32 bits.
- El Protocolo Trivial de Transferencia de Archivos (TFTP) usa el orden big-endian para los campos de 16 bits (número de bloque y código de error) y usa ASCII para los datos (nombre de campo, modo y descripción de errores).

2.2.7. Desempeño y Seguridad

Usualmente la disminución del desempeño al usar RPC en comparación al de una llamada local está dada por un factor mayor a 10. Sin embargo, hay razones importantes para distribuir las aplicaciones entre sistemas locales y remotos. Para un

procedimiento que es llamado con dos argumentos de 16 bits y que regresa dos valores de 16 bits, la diferencia entre una llamada local y una remota puede ser de un factor de 100. [Stevens90].

2.3. *Representación de los datos en productos RPC comerciales*

A continuación se describe de manera general la representación de los datos para los productos RPC de SUN, Courier de Xerox y RPC de Apollo. En los tres casos se usa una representación de tipo implícito, esto es, únicamente se transmite por la red el valor de la variable, no el tipo.

1. **RPC de Sun**. La representación de datos que usa Sun se denomina XDR. Impone un ordenamiento de byte big-endian y el tamaño mínimo de cada campo es de 32 bits. Por lo que, si un cliente en un sistema pasa un entero de 16 bits a un servidor que corre en un sistema similar usando XDR, el cliente debe convertir el valor de 16 bits en uno de 32 para que el servidor lo regrese a 16 bits. En el apéndice A se describe la representación de los diferentes tipos de datos usando XDR

2. **Courier de Xerox**. Este protocolo RPC también define un estándar que deben usar el cliente y el servidor. Se trata de un ordenamiento big-endian con un tamaño mínimo de campo de 16 bits. Los caracteres de los datos se codifican en el conjunto de caracteres de 16 bits NS de Xerox. Se usa ASCII de 8 bits para caracteres normales, con opción de cambio a otros conjuntos de caracteres como el "Greek" (usado cuando se envía texto matemático a ciertas impresoras).

3. **RPC Apollo**. En este caso, en lugar de imponer un estándar único para la representación de los datos, el NDR de Apollo soporta múltiples formatos. El transmisor puede usar su propio formato (si es uno de los formatos permitidos), por lo que el receptor tiene que convertirlo a su formato (si se trata de un formato diferente). Esta técnica tiene la ventaja de que dos sistemas con la misma arquitectura no necesitan realizar conversión de datos para comunicarse.

La figura 2.2 Se comparan los tipos de datos y formatos soportados por productos RPC.

Tipos de Datos	*RPC de Sun*	*Courier Xerox*	*NDR Apollo*
lógico 8 bits			*
lógico 16 bits		*	
lógico 32 bits	*		
entero con signo 8 bits			*
entero sin signo 8 bits			*
entero con signo 16 bits		*	*
entero sin signo 16 bits		*	*
entero con signo 32 bits	*	*	*
entero sin signo 32 bits	*	*	*
entero con signo 64 bits	*		*
entero sin signo 64 bits	*		*
orden de bytes	big-endian	big-endian	big-endian o little-endian

Tipos de Datos	RPC de Sun	Courier Xerox	NDR Apollo
formato de entero con signo	complemento a 2	complemento a 2	complemento a 2
punto flotante de 32 bits	*		*
punto flotante de 64 bits	*		*
formato punto flotante		IEEE	IEEE, VAX, IBM o Cray
tipo de caracter	ASCII	NS de 16 bits	ASCII o EBCDIC
enumeración	*	*	*
estructura (registro)	*	*	*
arreglo unidimensional fijo	*	*	*
arreglo unidimiensional variable			*
arreglo multidimensional fijo			*
arreglo multidimensional variable	*		*
union	*	*	*
datos opaque de longitud fija	*	*	*
datos opaque de longitud variable			

Figura 2-2 Tipos de datos soportados por productos RPC.

2.4. Llamadas a procedimientos remotos en productos comerciales

2.4.1. SUN RPC

Los RFC 1057 (1988) y RFC 1831 (1995) proporcionan la especificación de la versión 2 del protocolo RPC usado por Sun. Los RFC 1014 (1987) y RFC 1831 (1995) proporcionan una descripción de XDR. Las versiones de 1995 describen los protocolos RPC y XDR como estándar para la comunidad Internet.

El protocolo RPC de Sun consta de las siguientes partes [Stevens90]:

- *rpcgen*, un compilador que toma la definición de un interfaz de procedimiento remoto y genera los stubs de cliente y del servidor.
- El estándar *XDR* (**R**epresentación de **D**atos e**X**ternos).
- Una biblioteca a tiempo de ejecución que atiende detalles de comunicación.

En la figura 2.3 se muestra un ejemplo de los archivos necesarios para implementar una llamada a un procedimiento remoto. Como punto de partida es indispensable escribir *los procedimientos del servidor*, que se ejecutan de manera remota por el cliente, *el archivo de especificación RPC* y la *función principal* (main) del cliente. Los procedimientos del servidor y la función principal del cliente se escriben en lenguaje C,

mientras que la especificación se escribe en el lenguaje RPC. En el apéndice C se describe el protocolo de mensajes de RPC.

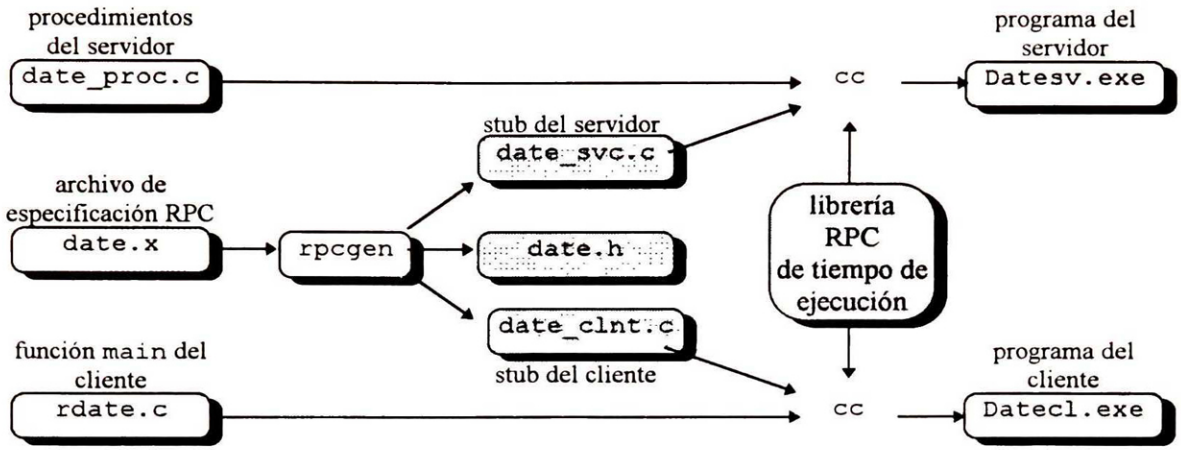


Figura 2-3 Archivos involucrados al generar un programa RPC de Sun.

El compilador *rpcgen* toma el archivo de especificación RPC mostrado en el listado 2.1 y genera dos archivos en lenguaje C (el stub del cliente y el stub del servidor) junto con un archivo de encabezado incluido en ambos. El stub del cliente se compila con la función main del cliente y se enlazan con funciones de la librería RPC para generar el programa ejecutable en la computadora del cliente.

De manera similar se genera el programa del servidor compilando el stub del servidor (que contiene la función main para el servidor) junto con las funciones del servidor y enlazando las funciones de la librería RPC.

Si el compilador RPC detecta estructuras dentro del archivo de especificación RPC, genera los filtros XDR correspondientes. (Ver apéndice A para la descripción de XDR).

2.4.1.1. Programas y Procedimientos de RPC

Un mensaje de solicitud (o de llamada) RPC, tiene 3 campos enteros que indican:

- Un número de programa remoto
- Un número de versión del programa remoto
- Un número de procedimiento remoto

Por medio de estos números se identifica de manera única a un procedimiento remoto. Los números de programas administrados por SUN, son enteros de 32 bits, en grupos de 536870912 (2000 0000 en hexadecimal) asignados como sigue:

```
0x00000000 - 0x1fffffff definido por Sun
0x20000000 - 0x3fffffff definido por el usuario
0x40000000 - 0x5fffffff transitorio
0x60000000 - 0x7fffffff reservado
0x80000000 - 0x9fffffff reservado
0xa0000000 - 0xbfffffff reservado
0xc0000000 - 0xdfffffff reservado
0xe0000000 - 0xffffffff reservado
```

Es necesario identificar la versión del protocolo RPC que se maneja (en este caso versión2).

```
/*
 * date.x - Especificación del servicio remoto de fecha y hora.
   define 2 procedimientos:
            bin_date_1() regresa la fecha y hora binarias (sin
                         argumentos)
            str_date_1() toma el tiempo binario y regresa una
                         cadena de caracteres que pueda entenderse para
                         las personas.
 */

program DATE_PROG {
   version DATE_VERS {
       long        BIN_DATE(void) = 1; /*número de procedimiento = 1*/
       string      STR_DATE(long) = 2; /* número de procedimiento = 2*/
   } = 1;                  /*Número de versión del programa remoto = 1*/
} = 0x31200000;           /*Número de programa remoto */
```

listado 2.1 Especificación del servicio remoto para el ejemplo de la figura 2.1

2.4.1.2. Consideraciones en el diseño

Paso de parámetros

Se permite un solo argumento y un solo resultado. Lo argumentos o resultados múltiples deben incluirse en una estructura.

Binding, conexión o enlace

El servidor cuenta con un servidor de nombres llamado "portmapper" que administra una Base de Datos, en donde se registran los servicios (o rutinas), sus protocolos y los puertos asociados a los servicios. El cliente se comunica con el portmapper en un puerto conocido y le envía una petición indicando el programa, versión y procedimiento deseado. El cliente debe enviar el protocolo de transporte sobre el cual se ejecuta el programa deseado. Si el programa está registrado, el portmapper regresa el puerto en el que se encuentra el servicio que se solicita. El cliente ejecuta el procedimiento remoto a través de ese puerto. En la figura 2.4 se esquematiza este procedimiento.

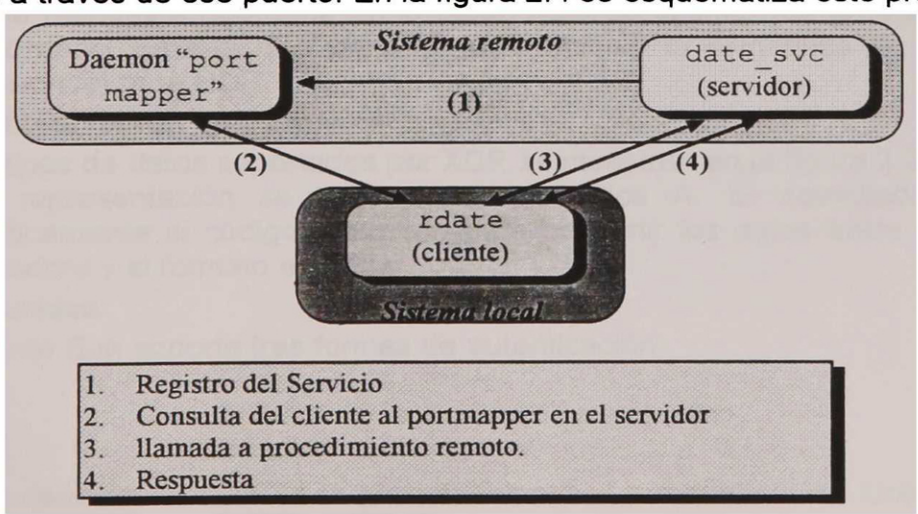


Figura 2-4 Pasos para establecer una llamada remota

Protocolo de transporte

RPC de Sun soporta los protocolos UDP y TCP. Cuando se usa TCP, debe existir forma de delimitar los registros en el flujo de bytes. RPC de Sun usa un protocolo de marcado de registros definido en RFC 1057. Usa un entero de 32 bits al comienzo de cada registro para especificar el número de bytes en el registro.

Cuando se usa UDP el número total de argumentos no debe generar un paquete que exceda de 8192 bytes de longitud. De manera similar, el tamaño total de los valores de retorno debe ser menor a 8192 bytes.

Manejo de excepciones

Cuando se usa UDP como protocolo de transporte, las peticiones RPC se retransmiten automáticamente, si es necesario. Después de un número determinado de intentos y si no hubo respuesta, se detiene la transmisión y se regresa un error al solicitante. Cuando se usa TCP, también se envía un error si el servidor termina la conexión. No hay forma de que el cliente envíe una interrupción al servidor.

Semántica de las llamadas

En el protocolo RPC de Sun, cada petición del cliente tiene un número único de transacción ID (un entero de 32 bits llamado *xid*). No se permite que el servidor examine este valor en otra forma que no sea una comparación. Tanto las funciones del cliente UDP como TCP inician este ID a un valor aleatorio cuando se crea el identificador ("handle") del cliente. Este valor cambia cada vez que se realiza una nueva petición RPC. Las funciones TCP y UDP del servidor regresan el valor ID enviado por el cliente. Adicionalmente, las funciones TCP y UDP del cliente realizan una prueba para comparar el ID en la transacción antes de regresar un resultado RPC. Esto asegura que la respuesta corresponde a la solicitud del cliente.

Las funciones UDP del servidor tienen una opción para recordar las peticiones que ha recibido del cliente. Para esto el servidor mantiene una tabla de sus respuestas con un índice de acuerdo a su ID de transacción, número de programa, número de versión, número de procedimiento y dirección UDP del cliente. Cada vez que se recibe una petición, el servidor UDP consulta la tabla para verificar si la petición está duplicada. En este caso no se llama de nuevo al procedimiento remoto, sino que se envía al cliente la respuesta guardada en la tabla. Aquí se asume que la respuesta previa se perdió o dañó. Esta técnica pretende proporcionar la semántica de máximo-una-vez (at-most-once) para UDP.

Representación de datos

Los tipos de datos soportados por XDR se muestran en la figura 2.2 y la descripción de su representación se hace en el apéndice A. El compilador RPC genera automáticamente el código requerido para convertir los datos entre el formato de la computadora y el formato estándar.

Seguridad

RPC de Sun soporta tres formas de autenticación

- Nula.
- Unix.
- DES:

La autenticación nula es la predeterminada. La autenticación Unix agrega a cada petición RPC los siguientes campos: una marca de tiempo, el nombre de la computadora local, el ID de usuario efectivo del cliente, el ID del grupo de cliente

efectivo y una lista de los demás ID a las que pertenece el cliente. El servidor puede examinar esos campos para determinar si atiende o no la petición del cliente.

La autenticación DES se puede encontrar en el RFC 1057.

2.4.2. COURIER DE XEROX

Courier es un lenguaje de especificación y también un protocolo que se usa principalmente bajo Unix para comunicar otros sistemas Xerox (servidores de archivos Xerox, servidores de impresión Xerox, entre otros). La siguiente descripción se basa en el Courier bajo 4.3BSD (cuyo código fuente se puede obtener en el BSD Networking Release). [Stevens90].

En Courier se define un protocolo llamado de Transferencia de Datos en Bloque (Bulk Data Transfer -BDT-) que se usa en los casos en los que la aplicación va a transferir una gran cantidad de datos, así como un argumento o un resultado. La implantación Unix de Courier soporta transferencia en bloque.

Para Obtener el software del cliente y el servidor se deben escribir 3 archivos:

- *La especificación RPC en lenguaje Courier.*
- *El programa del cliente en lenguaje C.*
- *Las funciones del servidor en lenguaje C.*

El compilador de Courier, `xnscourier`, se usa para transformar el archivo de especificación en cinco archivos, como se muestra en el ejemplo de la figura 2.5. El compilador requiere que el archivo de especificación posea extensión .cr. La función `Date1_support.c` define varias funciones de soporte usadas por el cliente y el servidor.

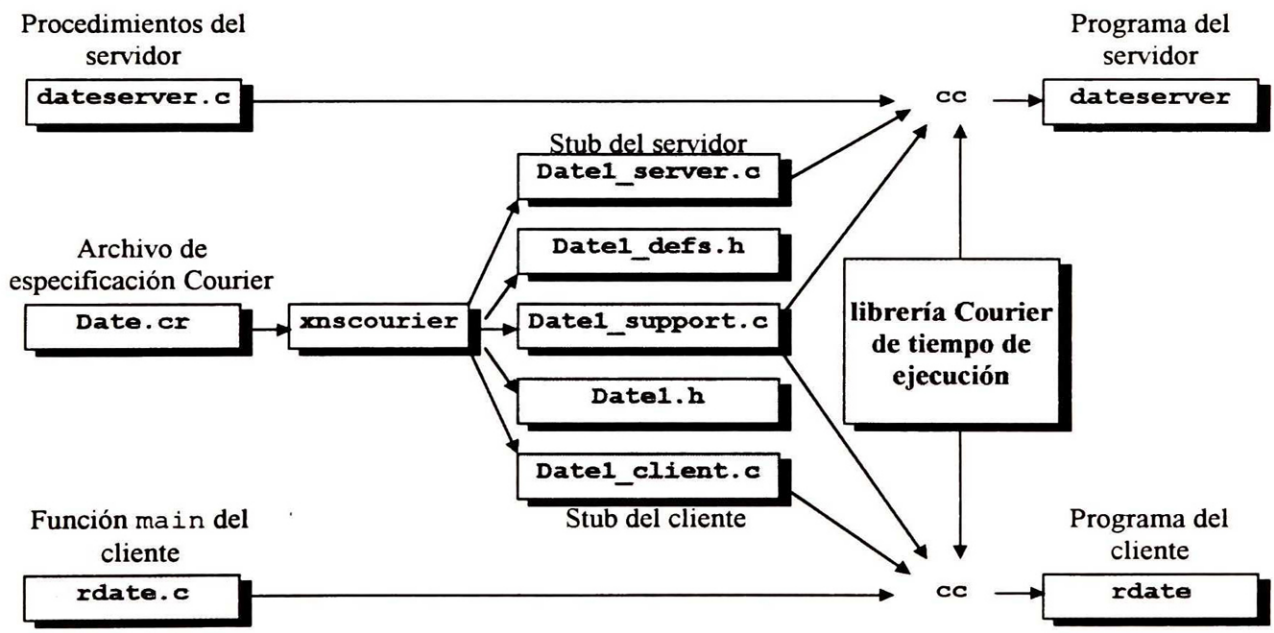


Figura 2-5 Archivos involucrados en la generación de un programa Courier

El archivo de especificación para el ejemplo de la figura 2.5 se muestra en el listado 2.2:

```
Date : PROGRAM 876 VERSION 1 =

BEGIN
      -- Define los procedimientos remotos.
      BinDate : PROCEDURE []
                  RETURNS [bindate: LONG INTEGER]
                  = 0;
      StrDate : PROCEDURE [bindate: LONG INTEGER]
                  RETURNS [strdate: STRING]
                  =1;
END.
```

listado 2.2 Archivo de especificación

Las palabras clave de Courier deben estar en mayúsculas. La elección del número de programa (876) en el ejemplo es arbitraria. El compilador Courier agrega un prefijo (`Date1` en el ejemplo) a todos los archivos que genera. Este prefijo se forma con el nombre del programa y el número de versión.

El código del archivo de encabezados `Date1.h` generado por el compilador se muestra en el listado 2.3:

```
/** Definiciones para la versión 1 número 876 de Date. **/
#ifndef _Date1
#define _Date1
#include          <xnscourier/courier.h>
#include          <xnscourier/courierconnection.h>

typedef struct {
            longInteger bindate;
} BinDateResults;
extern BinDateResults   BinDate();
typedef struct {
            string strdate;
} StrDateResults;
extern StrDateResults   StrDate();
#endif _Date
```

listado 2.3 Archivo de especificación de courier

Se observa que Courier define el resultado de cada función como una estructura. Esto se debe a que el procedimiento remoto puede regresar cualquier número de valores. El lenguaje C no soporta regreso de varios valores de una función, por lo que todos los procedimientos remotos en Courier regresan una estructura que contiene un elemento por cada valor de retorno. Courier también permite el paso de cualquier número de argumentos a un procedimiento remoto.

El programa del cliente para el ejemplo se muestra en el listado 2.4:

```
/* rdate.c - programa remoto para servicio de fecha remoto.*/

#include    <stdio.h>
#include    "Date1_defs.h"     /*archivo generado por el compilador*/
#include    <sys/types.h>
#include    <netns/ns.h>       /*definición de la estructura ns_addr*/

main(argc, argv)
int   argc;
char  *argv[];
{
      CourierConnection        *conn;       /*handle RPC*/
      char                     *server;
      struct ns_addr           ns_addr(),   /*rutina de librería BSD*/
                               servaddr;
      BinDateResults           binresult;   /*resultado de BinDate()*/
      StrDateResults           strresult;   /*resultado de StrDate()*/

      if (argc != 2) {
            fprintf(stderr,"uso: %s dirección del host\n", argv[0]);
            exit(1);
}
server = argv[1];
servaddr = ns_addr(argv[1]);    /*convierte a una estructura ns_addr*/
/*
 * Crea el "handle" del cliente.
 */
if ((conn = CourierOpen(&servaddr)) == NULL) {
      fprintf (stderr, "No es posible lograr conexión a %s\n", server);
      exit(2);
}
/*
 * Primera llamada al procedimiento remoto BinDate().
 */
binresult = BinDate(conn, NULL);
printf("hora en host %s = %ld\n", server, binresult.bindate);
/*
 * Llamado al procedimiento remoto StrDate().
 */
strresult = StrDate(conn, NULL, binresult.bindate);
printf("hora en el host %s = %s", server, strresult.strdate);

CourierClose(conn);      /*termina la conexión*/
exit(0);
}
```

listado 2.4 Cliente courier

El archivo `Date1_defs.h`, que se incluye en el programa del cliente, lo genera el compilador Courier. Este archivo incluye al archivo de encabezados `Date1.h`.

El cliente abre una conexión con la función `CourierOpen`, llama a los dos procedimientos remotos y cierra la conexión. Nótese que en la llamada a la función que establece la conexión se especifica únicamente el nombre del sistema remoto y no el procedimiento remoto que va a llamarse.

Los primeros dos argumentos de ambos procedimientos remotos son los mismos (el handle RPC -conn- y un apuntador a NULL). El segundo argumento se usa cuando hay una transferencia en bloque, por lo que en el ejemplo está apuntando a NULL. El tercer argumento en el segundo procedimiento es el resultado de la llamada al primer procedimiento remoto.

Cada procedimiento remoto regresa una estructura cuyo tipo es el nombre de procedimiento agregado en `Results`. Los `typedef` de C para `BinDateResults` y `StrDateResults` se encuentran en el archivo `Date1.h`.

Un procedimiento remoto Courier puede regresar valores de tres maneras:

- retorno normal,
- rechazo del sistema remoto para ejecutar el procedimiento,
- aborto generado por el sistema remoto.

En el caso del ejemplo, el cliente realiza un retorno normal. El rechazo y el aborto pueden manejarse en el programa, si no es así, el programa del cliente termina.

En el lado del servidor, sólo se tienen que codificar los procedimientos del servidor. La librería a tiempo de ejecución (`run-time`) proporciona una función `main` que se invoca cuando llega una petición RPC para algunos de los procedimientos soportados.

```
/* dateserver.c - procedimientos remotos llamados por el stub del
 * servidor.
 */
#include    "Date1_defs.h"    /*generado por el compilador*/
/* Regresa la hora y la fecha en binario.*/
BinDateResults
BinDate(conn, bdtptr)
CourierConnection       *conn;
char                    *bdtptr;
{
      BinDateResults    result;
      long              time();          /*función Unix*/

      result.bindate = time((long *) 0);
      return(result);
}
/* Convierte la hora binaria a cadena de caracteres.*/
StrDateResults
StrDate(conn, bdtptr, bintime)
CourierConnection       *conn;
char                    *bdtptr;
long                    bintime;
{
      StrDateResults    result;
      char              *ctime();        /*función Unix*/

      result.strdate = ctime(&bintime);  /*convierte a tiempo local*/
      return(result);
}
```

listado 2.5 Servidor courier

Los procedimientos del servidor regresan una estructura. No es necesario guardar de forma estática la estructura ya que los procedimientos regresan la estructura completa, no su dirección.

Para ejecutar un procedimiento remoto en cualquier computadora con Sistema Operativo Unix que soporte RPC Courier, se debe ejecutar un demonio courier (llamado `xnscourierd`) que espere las peticiones de conexión del cliente en un puerto determinado. Una vez que se ha compilado y enlazado el programa del servidor, se debe avisar al demonio Courier que el servidor está disponible. Para lograr lo anterior, hay que agregar al archivo `/etc/Courierservices`, la especificación del número de programa, número de versión y trayectoria (localización) del archivo ejecutable para el programa del servidor. Normalmente el demonio `xnscourierd` se ejecuta cuando se inicia el sistema.

Los pasos involucrados en la ejecución del programa cliente se explican a continuación y se ilustran en la figura 2.6.

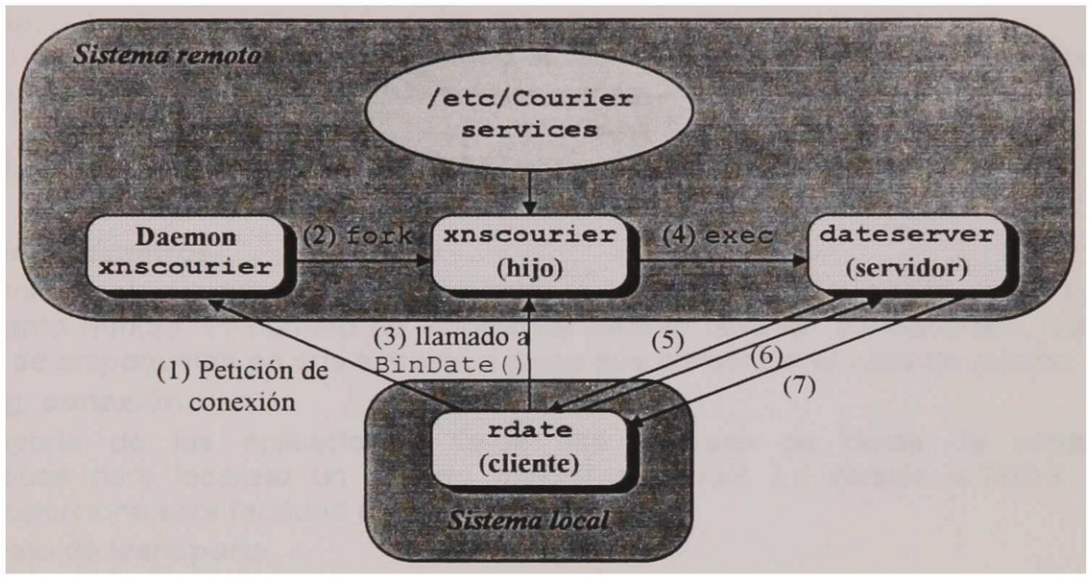


Figura 2-6 Pasos involucrados en la ejecución del programa cliente

1. El cliente llama a `CourierOpen` para solicitar el establecimiento de una conexión a la computadora especificada. La conexión se realiza usando el puerto 5 en la computadora del servidor, en la cual el proceso `xnscourierd` espera las peticiones del cliente.
2. El demonio crea (`fork`) un proceso hijo para manejar la conexión. Este proceso lee los primeros bytes de la conexión para verificar que se trata de un cliente Courier. Entonces el proceso hijo espera la primer petición RPC del cliente, que especifica el programa, versión y procedimiento que se llamará.
3. Cuando un cliente llama a la función `BinDate`, el stub del cliente escribe el número de programa, versión y procedimiento deseados.
4. El proceso hijo lee la información de la conexión y consulta la información en el archivo `/etc/Courierservices` para determinar el programa del servidor que se ejecutará. El proceso hijo ejecuta el programa apropiado (`exec`), pasando el valor del número de procedimiento al programa del servidor como un argumento en la línea de comando. De ésta manera el stub del servidor determina cuál de los dos procedimientos se esta llamando.
5. El programa `dateserver` inicia y la función `main` del stub del servidor procesa los argumentos de la línea de comando para determinar el procedimiento remoto

que se llama. Entonces llama a la función `BinDate`. Cuando regresa el resultado, el stub del servidor lo convierte al formato estándar Courier y lo envía al cliente.

6. El cliente llama a la función `StrDate`. El stub del cliente envía los números de programa, versión y procedimiento usando la conexión activa para que el stub del servidor los lea. El stub del servidor determina que el programa y la versión no han cambiado, por lo que llama a la función `StrDate`. Antes de ese llamado, el stub del servidor lee los argumentos de la conexión y los convierte del formato Courier al requerido por la función. Si el stub del servidor recibe una petición de un programa o versión diferente, tiene que ejecutar el programa apropiado para manejar la nueva petición. Si el programa y la versión no cambian, el sistema remoto puede usar un sólo proceso para manejar las peticiones del cliente.
7. Cuando la función del servidor regresa un valor al stub, éste lo convierte y lo envía al cliente.

Cuando se obtiene el resultado solicitado el cliente llama a `CourierClose` para terminar la conexión, provocando la finalización normal del proceso del servidor.

2.4.2.1. Consideraciones en el diseño

Paso de parámetros

Se permite cualquier número de argumentos, que se transfieren individualmente al procedimiento remoto. El número de valores de retorno también es indefinido, dado que éstos se empaquetan en una estructura única que constituye el valor de retorno.

Binding, conexión

La mayoría de las aplicaciones Xerox usa la base de datos de objetos Clearinghouse para localizar un servicio particular de red. La versión 4.3BDS de Courier proporciona esta facilidad de forma limitada.

Protocolo de transporte

Courier usa el Protocolo de Paquetes en secuencia XNS (SPP) para la conexión entre el cliente y el servidor. Es un protocolo orientado a conexión, por lo que no hay límite en la cantidad de datos que pueden intercambiarse entre cliente y servidor.

Manejo de excepciones

La detección de una falla en el servidor se traduce como una caída de la conexión. Algunas versiones de Courier permiten al cliente la recuperación de ciertas condiciones de excepción.

Semántica de las llamadas

Courier se apoya en SPP para proporcionar una conexión confiable entre cliente y servidor. Si el procedimiento remoto se ejecutó una sola vez, se realiza un retorno normal. El retorno de algún error implica que el procedimiento se ejecutó hasta una vez

Representación de datos

La figura 3.2 muestra los tipos de datos que soporta Courier. El compilador `xnscourier` genera automáticamente el código C requerido para convertir los argumentos y resultados entre el formato de la computadora y el formato estándar.

Seguridad

La validación del acceso de usuarios a los servicios de red se realiza, en la mayoría de las aplicaciones Xerox, usando el protocolo de Autenticación de Xerox. La

explicación protocolo Xerox se describe en [XEROX86]. La autenticación es similar al sistema Kerberos [Stevens90].

La versión 4.3BSD de Courier soporta de forma limitada la autenticación simple y no soporta autenticación rigurosa.

2.4.3. RPC DE APOLLO

La arquitectura para llamadas a procedimientos remotos de Apollo se llama Arquitectura de Computación de Red (NCA). En esta arquitectura se basa el producto de Apollo denominado Sistema de Computación de Red (NCS), el cual está constituido por las partes siguientes:

- NIDL, el Lenguaje de Definición de Interfaz de Red, a partir del cual un compilador genera los stubs del cliente y servidor.
- NDR, la Representación de Datos de Red, que define los formatos estándar usados para pasar los tipos de datos soportados.
- Una librería de tiempo de ejecución.

En los siguientes párrafos se abordarán las características que dan transparencia a la ejecución de las aplicaciones remotas.

2.4.3.1. Consideraciones en el diseño

Paso de parámetros

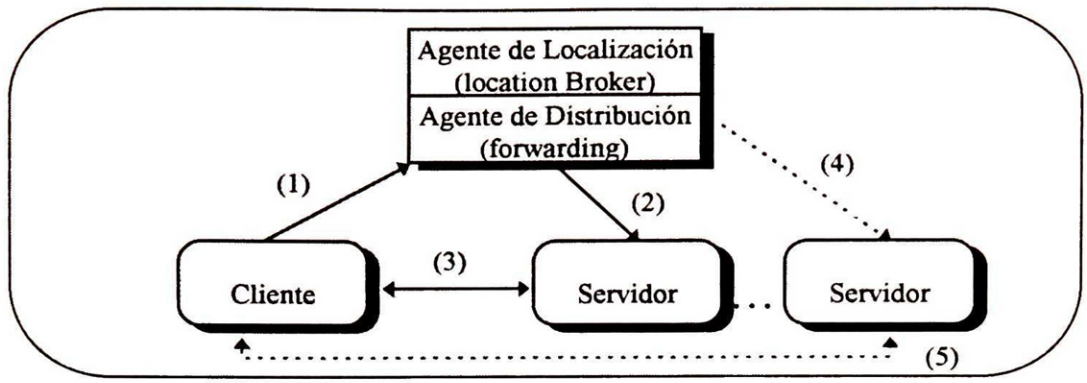
RPC/NCA permite cualquier número de argumentos y cualquier número de valores de retorno. Al momento de escribir la función RPC prototipo se debe especificar si cada parámetro es de entrada (pasa del cliente al servidor), de salida (pasa del servidor al cliente) o ambos.

Binding

En el sistema de Apollo se proporciona un agente de distribución ("forwarding") con funciones similares al "portmapper" descrito en RPC de Sun. Recordemos que para contactar al servidor, el cliente de RPC de Sun tiene que intercambiar un mensaje con el "portmapper" para determinar la dirección del servidor. En la propuesta de Apollo el cliente envía su primer petición al agente para que éste lo dirija al servidor. El servidor envía su respuesta directamente al cliente (incluyendo en ella su dirección), por lo que el cliente puede usar la dirección del servidor para peticiones subsecuentes.

El agente de distribución forma parte de un agente de conexión, o enlace (binding) mayor llamado el Agente de Localización (Location Broker). Este agente responde a una solicitud de información del cliente acerca del nombre de sistemas, direcciones de servidores que proporcionan servicios particulares. El cliente puede contactar al agente de localización enviando un mensaje de difusión en la red, que incluya el tipo de servicio buscado. Todos los servidores que soporten el servicio responderán, sin embargo el cliente normalmente usará el servidor cuya respuesta haya sido la primera.

El agente de localización responde con la dirección completa del socket, no sólo los números de puerto en el protocolo (en RPC de Sun el "portmapper" requiere la especificación del protocolo de transporte, TCP o UDP y regresa el número de puerto apropiado para el protocolo). Estos pasos se representan en la figura 2.7.



1. El cliente envía petición al Agente de distribución
2. El agente contacta y envía la petición al servidor
3. El servidor envía la respuesta y su dirección al cliente y el cliente usa esta dirección para llamadas subsecuentes
4. El agente puede distribuir a varios servidores la petición
5. El cliente recibe las respuestas y aceptará al primer servidor que responda.

Figura 2-7. Pasos Involucrados para hacer una llamada remota en Apollo

Protocolo de transporte

Las llamadas RPC de Apollo están diseñadas para correr sobre un protocolo de transporte sin conexión. Se usa UDP o DDS (un protocolo propietario de Apollo).

El sistema de computación remota de Apollo fragmenta y ensambla mensajes largos de red , que no pueden acomodarse en un paquete de red (UDP tiene un límite superior de casi 64 Kbytes mientras que Xerox de 546 bytes).

El manejador (handle) del cliente que identifica un servidor particular contiene la dirección completa del socket del servidor, lo que implica que el cliente puede tener independencia del protocolo, ya que la dirección completa del socket es parte del "handle" de cada llamada. El cliente no necesita saber el protocolo de transporte que se está usando hasta que logra relacionar su "handle" al servidor específico. Aún cuando el "handle" está asociado a un servidor particular, el cliente considera al primero como una estructura `opaque`, por lo que no necesita saber sobre que protocolo de transporte se está trabajando.

Manejo de excepciones

En RPC de NCA es posible proporcionar al cliente reportes de excepciones (fallas particulares) del servidor. También se permite que el cliente envíe señales al servidor que lo está atendiendo.

Semántica de las llamadas

El desarrollo de Apollo provee la semántica de llamadas "at-most-once" (hasta una). Para lograrlo, el servidor mantienen un registro de la petición anterior para cada cliente con el que se ha comunicado. De esta forma si se detecta una retransmisión de una petición anterior, el servidor envía una respuesta registrada sin ejecutar de nuevo el procedimiento remoto. El servidor retransmite el paquete de respuesta si ha recibido un reconocimiento del cliente indicando que éste ha recibido la respuesta. Si el cliente envía una nueva petición, el reconocimiento se asume implícito, de otra forma el cliente

envía el reconocimiento explícito. El servidor puede remover algunas respuestas de su memoria si no hay solicitudes repetidas después de un tiempo.

El cliente tiene la facultad de preguntar el estado actual del servidor, por lo que puede decidir si retransmite su petición previa o espera el resultado del servidor. El servidor puede contestar a la petición de estado con un paquete que indique que se encuentra trabajando en la solicitud o que ésta nunca fue recibida.

En el sistema de Apollo se permite al programador indicar si su procedimiento remoto es "idempotente", caso en el cual el servidor no guarda las respuestas previas y llama de nuevo al procedimiento remoto si recibe una petición duplicada. En este caso el cliente no envía reconocimientos de recepción de respuestas del servidor.

Representación de datos

La figura 2.2 muestra los tipos de datos soportados por NDR.

CAPITULO

3

Bibliotecas RPC, XDR, Stubs y encabezados

3

3. BIBLIOTECAS RPC, XDR, STUBS Y ENCABEZADOS

Para la programación de aplicaciones distribuidas el compilador CRAD emplea las bibliotecas RPC y XDR para facilitar la programación. Para obtener un conocimiento más claro de las funciones que genera el compilador, se explicarán en este capítulo las funciones que se programaron para conformar las bibliotecas mencionadas. Estas funciones se emplean en los stubs y en las rutinas XDR. Además se explicará la generación de stubs, encabezados y rutinas XDR.

3.1 Biblioteca de funciones XDR generadas para el manejo de los datos

En el apéndice A se especifica el protocolo XDR, que deben seguir las computadoras para la representación e intercambio de estructuras de los datos. Para el empleo de dicho protocolo se generó la biblioteca de funciones XDR, la cual se emplea en las rutinas RPC.

XDR define la representación Big-endian ó el primer bit más significativo (MSB), y la representación en punto flotante de IEEE.

Para emplear las rutinas XDR es necesario incluir el archivo de encabezados de XDR (`xdr.h`) y la llamada a dicha librería (`xdr.lib`).

Para la explicación de esta biblioteca se clasificarán las rutinas XDR en 3 categorías: [Bloo92]

- Rutinas de creación.
- Rutinas para el manejo de streams XDR (Estos streams son Buffers o áreas de memoria que se manejan dentro y fuera de los filtros)
- Rutinas de conversión simple y complejas. Estas rutinas actúan como filtros para codificar y decodificar los datos y liberar espacio en memoria. Todos los filtros tienen la sinopsis que se especifica en la siguiente línea.

```
bool_t xdrproc (XDR *xdrs, <tipo> *argresp)
```

En este ejemplo se identifican dos argumentos, una variable apuntador `xdrs` del tipo XDR. En la estructura (XDR), se manejan y almacenan la conversión de los datos de entrada y salida al formato XDR. Los campos de esta estructura se representa en la figura 3.1.

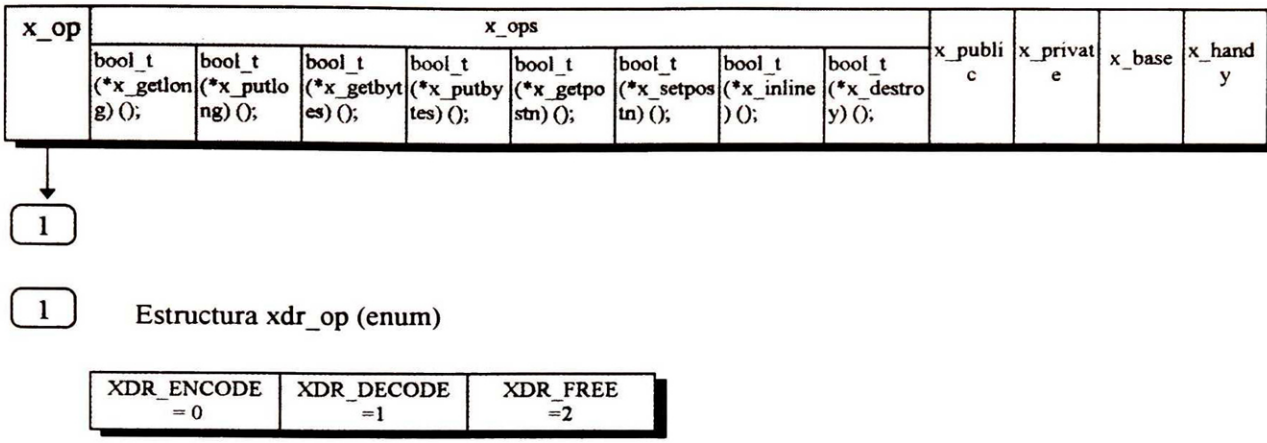


Figura 3-1 Estructura XDR.

Las funciones que se presentan dentro de la estructura x_ops, indican:

- `x_getlong`: Rutina que obtiene un tipo long de la cadena que tiene esta estructura
- `X_putlong`. Coloca un tipo long en la cadena
- `x_getbytes`. Obtiene algunos bytes de la cadena.
- `x_putbytes`. Coloca algunos bytes a la cadena.
- `x_getpostn`. Regresa la posición de x bytes.
- `x_setpostn`. Se coloca en alguna posición indicada de la cadena
- `x_inline`. Regresa el apuntador del buffer o cadena de datos
- `x_destroy`. Libera la memoria de la cadena y destruye la estructura

Los variables restantes indican:

- `x_public`: Datos del usuario
- `x_private`: apuntador a datos privados
- `x_base`. Datos privados usados para Información de posición.
- `x_handy`. Información privada extra.

Las rutinas de la biblioteca XDR permiten el manejo y la asignación de memoria de una cadena de caracteres ó buffers de datos XDR donde se codifican y decodifican los datos. Dentro de la estructura XDR se asigna la cadena de caracteres mencionada. Como la estructura XDR pasa como parámetro en cada una de las rutinas xdr, dichas rutinas hacen uso de la cadena de caracteres incluida en la estructura, para codificar sus datos e insertarlos en la cadena de caracteres. Una vez iniciada la cadena de caracteres, se debe asociar las rutinas para el manejo de dicha cadena. Para lo cual se emplean la declaración de funciones dentro de la estructura XDR.

3.1.1 Rutinas XDR para la creación de la cadena de caracteres

Para la codificación de los tipos al protocolo XDR, emplea la rutina `xdrmem_create` que inicia una cadena de caracteres dentro de la estructura principal XDR. Esta rutina asigna memoria para dicha cadena. Se declara como:

```
void xdrmem_create (XDR *xdrs, char *addr, u_int size, enum xdr_op op)
```

3.1.2 Rutinas XDR para el manejo de la cadena de caracteres

Para manejar la cadena de caracteres dentro de la estructura de datos XDR, se inician varias rutinas que se asignan a las funciones declaradas dentro de la estructura XDR.

xdr_destroy () : Destruye una cadena de caracteres o área de memoria, liberando recursos. Invoca la rutina `destroy()` asociada con la estructura XDR mencionada. Se declara como:

```
void xdr_destroy(XDR *xdrs)
```

xdr_getpos () : Regresa la posición actual de la cadena de caracteres. Por medio de esta rutina se permite insertar en la posición adecuada la cadena que se codifica o se decodifica. Esta rutina llama a la rutina `get_position`. Se declara como:

```
u_int xdr_getpos( XDR *xdrs)
```

`xdr_inline` (): Regresa una apuntador a la pieza contigua de la cadena de caracteres. Se declara como:

```
Long * xdr_inline (XDR *xdrs, int len)
```

`xdrrec_readbytes`(): Lee un número de bytes de la cadena de caracteres

`xdrrec_skiprecord()` : Salta al inicio del siguiente registro.

`xdr_setpos()` : Se coloca en la cadena de caracteres.

3.1.3 Rutinas XDR para el manejo de conversión de tipos simples (filtros)

Estas rutinas convierten tipos de datos construidos en el lenguaje C dentro de la representación XDR. Los filtros tienen el propósito de codificar los datos dentro del formato XDR, decodificarlos de nuevo ó liberar la memoria asociada con la estructura XDR.

Cada filtro simple regresa un tipo denominado `bool_t` indicando TRUE si la codificación, decodificación ó liberación de memoria de la rutina tuvo éxito, en otro caso regresa FALSE. Las fallas pueden deberse a datos ilegales, un error en la cadena de caracteres, tales como insuficiente espacio en disco, etc.

La estructura XDR, tiene un campo de operación (xdr_op) donde se indica el tipo de operación que se desea realizar ya sea XDR_DECODE, XDR_ENCODE o XDR_FREE.

Las rutinas que manejan los tipos básicos o filtros básicos se explican en las siguientes líneas.

xdr_bool (): Convierte un tipo bool, actualmente un entero en el lenguaje C. Su representación XDR tiene los valores 0 ó 1.

```
bool_t xdr_bool (xdr *xdrs, bool_t *bp)
```

xdr_char (): hace la conversión entre un char del lenguaje C y la forma XDR externa asociada. Debido a que xdr_char ocupa 4 bytes, cuando se trabaja con arreglos o strings de caracteres, es mejor emplear las rutinas xdr_bytes (), xdr_opaque() ó xdr_string().

```
bool_t xdr_char(xdr * xdrs, char *cp)
```

xdr_short (): Hace la conversión entre un entero short del lenguaje C a su asociada forma XDR.

```
bool_t xdr_short (XDR *xdrs, short *sp)
```

xdr_u_short (): Hace la conversión entre un entero corto (short) sin signo del lenguaje C al formato XDR.

```
bool_t xdr_short (XDR *xdrs, unsigned short *usp)
```

xdr_int (): Hace la conversión de un entero del lenguaje C al formato XDR.

```
bool_t xdr_int(XDR *xdrs, int *ip)
```

xdr_u_int (): Hace la conversión de un entero sin signo del lenguaje C al formato XDR.

```
bool_t xdr_u_int (XDR *xdrs, unsigned *up)
```

xdr_long (): Hace la conversión de enteros largos (long) del lenguaje C al formato XDR

```
bool_t xdr_long(XDR *xdrs, long *lp)
```

xdr_u_long() : Hace la conversión de enteros largos sin signo del lenguaje C al formato XDR.

```
bool_t xdr_u_long (XDR *xdrs, unsigned long *ulp)
```

xdr_float (): Hace la conversión de los tipos float del lenguaje C al formato XDR

```
bool_t xdr_float (XDR *xdrs, float *fp)
```

xdr_double (): Hace la conversión de los tipos double del lenguaje C al formato XDR.

```
bool_t xdr_double (XDR *xdrs, double *dp)
```

xdr_void (): Siempre regresa TRUE. Es utilizado junto con las rutinas que no tienen parámetros de entrada o de salida.

```
bool_t xdr_void ()
```

xdr_enum (): Hace la conversión de tipos enum del lenguaje C al formato XDR.

```
bool_t xdr_enum (XDR *xdrs, enum_t *ep)
```

3.1.4 Rutinas XDR para el manejo de conversión de tipos complejos (filtros)

Existe un grupo de rutinas de la biblioteca XDR encargadas de trasladar tipos de datos más complejos dentro de la representación XDR. Estas rutinas manejan vectores, arreglos, uniones, `strings` y apuntadores o referencias. Estas rutinas también regresan el tipo `bool_t` (TRUE o FALSE) y maneja los tipos de operación mencionados en el punto 3.1.3, para identificar la operación a realizar.

`xdr_array ()`: Arreglo de longitud variable con un tamaño de elementos arbitrario. Es la rutina de conversión para un arreglo de longitud variable, apuntado por `*arrp`, conteniendo los elementos especificados en `*sizep`, cada `elsize` bytes. El tamaño `sizep` debe establecerse para la codificación e iniciarse cuando se decodifica. Si el arreglo es mayor que `maxsize`, la rutina regresa una falla (FALSE). Debe alojarse memoria para el arreglo antes de la decodificación dentro de *arrp. El procedimiento XDR `xdrproc_t elproc`, es el filtro usado para trasladar o convertir cada elemento

#include <xdr.h>

```
bool_t xdr_array (XDR *xdrs, char ** arrp, u_int
*size, u_int maxsize, u_int elsize, xdrproc_t elproc)
```

`xdr_bytes ()`: convierte un arreglo de bytes de longitud variable dentro y fuera del formato de la estructura XDR. Esta rutina trata al arreglo como bytes opaque (ver apéndice A). Este arreglo inicia en la variable *`arrp`. Se reserva memoria en la estructura XDR para la decodificación si *`arrp` es NULL. Se libera esta memoria con la rutina `xdr_free`. La variable tipo `sizep` indica el número de bytes que debe iniciarse para la codificación *y decodificación. La rutina falla si hay mas bytes que la variable* `maxsize`.

```
bool_t xdr_bytes(XDR *xdrs, char **arrp, u_int
*sizep, u_int maxsize)
```

`xdr_opaque ()`: Datos de longitud fija (no interpretados). Convierte datos `opaque` de longitud fija. La variable `cnt` indica el número de bytes a ser codificados o decodificados.

```
bool_t xdr_opaque (XDR *xdrs, char *cp, u_int
cnt)
```

`xdr_pointer ()`: Convierte datos indirectos como `xdr_reference()`, pero ésta rutina puede manejar apuntadores NULL. Trabaja con árboles binarios y listas ligadas. La variable `objsize` indica el tamaño de **`objpp`. La variable `objproc` es el filtro XDR para los datos que contendrá la estructura.

```
bool_t xdr_pointer (XDR *xdrs, char **objpp,
u_int objsize, xdrproc_t objproc)
```

`xdr_reference ()`: recursivamente codifica/decodifica los apuntadores de los datos de tamaño `size`. Inicia con la *`pp` y usa el filtro XDR (`proc ()`) para los datos. Esta rutina no maneja apuntadores NULL.

```
bool_t xdr_reference (XDR *xdrs, char **pp, u_int
size, xdrproc_t proc)
```

xdr_string (): Codifica arreglos de caracteres terminados en NULL. Convierte la cadena *strp terminada en NULL al formato XDR y lenguaje C. La rutina Falla si se encuentra más caracteres que los mencionados en maxsize y regresa FALSE. Si *strp es NULL cuando se decodifican los datos, xdr_string() aloja el espacio apropiado. Con la rutina xdr_free libera la memoria. Strp no puede se NULL, debe apuntar a alguna dirección de memoria válida.

```
bool_t xdr_string (XDR *xdrs, char **strp, u_int
maxsize)
```

xdr_union (): Convierte una unión discriminante. Después de convertir el tipo discriminante enum_t como dscmp, se convierte la unión localizada en la variable unp. La variable choise es un apuntador a un arreglo de estructuras terminado en NULL conteniendo un par de valores [value, proc]. Si el discriminante es el mismo cono cualquier de los valores, el procedimiento proc se usa para trasladar los contenidos de la unión. Si ningún valor del discriminante coincide, entonces el procedimiento defaultarm se utiliza para la conversión.

```
bool_t xdr_union (XDR *xdrs, int *dscmp, char *unp,
struct xdr_discrimi *choise, bool_t (*defaultarm) ())
```

xdr_vector (): Convierte arreglos de longitud fija con un tamaño arbitrario de elementos. Convierte arreglos del tamaño indicado en la variable size_length, iniciando en la variable arrp. Cada elemento del arreglo tiene el tamaño en bytes indicado en elsize, usando el filtro elproc para convertir a cada uno.

```
bool_t xdr_vector (XDR *xdrs, char *arrp, u_int size,
u_int elsize, xdrproc_t elproc)
```

xdr_wrapstring (): Convierte arreglos de caracteres terminados en NULL de longitud variable. Convierte un string del lenguaje C terminado en NULL localizado en la variable *strp. Esto es equivalente a: xdr_string (xdrs, strp, MAX), donde MAX es el entero sin signo posible más grande. Si *strp es NULL en la decodificación, se reserva memoria y se libera con xdr_free ().

```
bool_t xdr_wrapstring (XDR *xdrs, char **strp)
```

3.2 Biblioteca de funciones RPC

3.2.1 Autenticación del cliente

En RPC, el cliente debe crear una estructura tipo CLIENT para solicitar servicios al servidor. Antes de hacer una petición, el servidor debe realizar la autenticación del cliente: AUTH_NONE, AUTH_UNIX, o AUTH_DES. AUTH_NONE no requiere verificación antes de que las peticiones sean atendidas; AUTH_UNIX requiere verificación estándar de usuario / clave (password); AUTH_DES usa encripción y verificación.

auth_destroy(): Destruye la información de autenticación.

```
void auth_destroy(AUTH *auth)
```

authnone_create( ): Crea información de autenticación por default, no se pasa información útil con cada RPC.

```
AUTH * authnone_create( )
```

authunix_create(): Construye información de autenticación a partir de credenciales UNIX.

```
AUTH *authunix_create(char *host, int iud, int
gid, int grouplen, int *gidlistp)
```

authunix_create_default( ): Llama a authunix_create( ) llenando los parámetros estándar de autenticación para ese usuario.

```
AUTH *authunix_create_default ( )
```

3.2.2 Llamada desde el lado cliente

callrpc( ), clnt_broadcast( ), y clnt_call( ) son funciones del lado cliente usadas para enviar la solicitud de una RPC. Estas funciones permanecen bloqueadas hasta obtener una respuesta. Se regresa la estructura de estado clnt_stat y si es diferente de cero, denota un problema que puede ser explicado usando las rutinas de error.

clntcall( ) se usa después de que la estructura cliente ha sido creada y se ha realizado la autenticación del cliente.

clnt_call(): Hace una llamada RPC usando el "handle" del cliente. Codifica los argumentos requeridos y responde usando los procedimientos XDR especificados.

```
enum clnt_stat clnt_call( CLIENT *clnt, u_long
procnum, xdrproc_t inproc, char *in, xdrproc_t outproc,
char *out, struct timeval timeout)
```

callrpc(): Realiza una RPC a un programa, versión y procedimiento específicos en el host requerido. La información de entrada y salida pasa por las rutinas de

conversión especificadas tanto para la codificación como para la decodificación XDR. `callrpc()` intenta la petición 5 veces (cada 5 seg.) y regresa un error de temporización si transcurren 25 seg. Regresa 0 si hay éxito ó un entero de estado en caso de falla. Los RPC realizadas de esta forma usan transporte UDP/IP sin control de autenticación.

```
int callrpc( char *host, u_long prognum, u_long
versnum, u_long procnum, xdrproc_t inproc, char *in,
xdrproc_t outproc, char *out)
```

clnt_broadcast( ): Difunde una petición de RPC a todas las redes conectadas localmente usando autenticación AUTH_UNIX. Cada vez que se recibe una respuesta, el procedimiento `eachresult( )` maneja los resultados. Los paquetes de difusión tienen tamaño limitado por la unidad de transferencia de enlace de datos. (Para Ethernet son 1400 bytes).

```
enum clnt_stat clnt_broadcast( u_long prognum,
u_long versnum, u_long procnum, xdrproc_t inproc, char
*in, xdrproc_t outproc, char *out)
```

clnt_freeres( ): Manejo de errores y reporte al cliente. Libera datos almacenados por el sistema RPC/XDR a partir de la decodificación de una respuesta en el cliente. `Out` es la dirección de los resultados, `outproc` es la rutina XDR usada. Regresa TRUE en caso de éxito o en caso contrario FALSE.

```
bool_t clnt_freeres(CLIENT *clnt, xdrproc_t
outproc, char *out)
```

clnt_geterr( ): Manejo de errores y reporte al cliente. Lleva la estructura errónea fuera del manejador del cliente (handle) a una localidad de memoria reservada.

```
void get_err(CLIENT *clnt, struct rpc_err errp)
```

clnt_perrno( ): Manejo de errores y reporte al cliente. Procesa el estado del cliente que regresan `callrpc( )` y `clnt_broadcast( )`. Envía el resultado a `stderr`.

```
void clnt_perrno(enum clnt_stat stat)
```

clnt_perror( ): Manejo de errores y reporte al cliente. Es como `clnt_perrno( )` pero se usa con `clnt_call( )`.

```
void clnt_perror(CLIENT *clnt, char *str)
```

clnt_sperrno( ): Manejo de errores y reporte al cliente. Es como `clnt_perrno( )` pero regresa un apuntador al mensaje en lugar de `stderr`.

```
char *clnt_sperrno(enum clnt_stat stat)
```

clnt_sperror( ): Manejo de errores y reporte al cliente. Regresa un apuntador a un mensaje estático, de otra forma funciona como `clnt_sperrno( )`.

```
char * clnt_sperror(CLIENT *clnt, char *str)
```

3.2.3 Administración del "handle" del cliente

Es necesario que por cada cliente se mantenga una única estructura CLIENT por cada conexión cliente/servidor abierta. Muchas rutinas RPC toman un "handle" CLIENT como un argumento de la asociación cliente / servidor. El "handle" normalmente es un apuntador indirecto doble a una estructura CLIENTE.

Un objeto cliente (referido por su "handle") contiene información de la conexión cliente/servidor .

clnt_control( ): Cambia y recupera información del cliente objeto. Cambia o recupera las características de CLIENT, incluyendo: temporización (timeout), temporización para reintento UDP, descriptor del socket (ambos extremos) y su estado. Trabaja con UDP y TCP. Regresa TRUE si hay éxito, en caso contrario FALSE.

```
Bool_t clnt_control(CLIENT *clnt, int request,
char *info)
```

clnt_create( ): Creación genérica de un cliente. Crea la estructura cliente para el host , programa y versiones específicas. El protocolo de transporte puede ser TCP o UDP. En UDP, los mensajes RPC son de hasta 8K bytes, entonces para mayor cantidad de datos se debe usar TCP. Si hay éxito, se regresa CLIENT. Si falla, es decir, si el portmap del servidor solicitado no coincide en el número de programa, regresa NULL. Un error en la versión no se toma como error, pero se muestra más tarde con una `clnt_call()`.

```
CLIENT *clnt_create(char *host, u_long prognum,
u_long versnum, char *protocol)
```

clnt_destroy( ): Libera recursos asociados con un "handle" de cliente. Libera memoria asociada con estructuras de datos privadas para CLIENT. Si las rutinas RPC se usaron para abrir el socket, entonces éste se cierra.

```
void clnt_destroy (CLIENT *clnt)
```

clnt_pcreateerror( ): Reporta a `stderr` errores de creación de cliente. Traduce errores por creación de CLIENT en algo útil hacia `stderr`.

```
void clnt_pcreateerror(char *str)
```

clnttcp_create( ): Crea un handle de cliente TCP para el programa y versión especificados en la computadora direccionada (dir. IP). El tamaño de los buffers TCP de recepción y envío puede especificarse, o dejarse en cero para usar valores por default. Si hay éxito regresa el handle CLIENT, de otra forma NULL.

```
CLIENT * clnttcp_create(struct sockaddr_in *addr,
u_long prognum, u_long versnum, int *sockp, u_int
sendsz, u_int recvsz)
```

clntudp_create( ): Crea un handle de cliente UDP. Similar a `clnttcp_create( )`. Para el transporte UDP se especifican el tiempo de espera para reintentos y

el tiempo de vencimiento del temporizador. El tiempo total permitido para completar una RPC se especifica en `clnt_call( )`. `clntudp_create()` toma tamaños de los buffers de default. En caso de éxito regresa el handle CLIENT, si falla regresa NULL.

```
CLIENT * clntudp_create(struct sockaddr_in *addr,
u_long prognum, u_long versnum, struct timeval wait,
int *sockp)
```

rpc_createrr( ): Es una variable global que reporta errores de creación del cliente. Es usada por `clnt_pcreateerror( )`.

```
struct rpc_createerr rpc_createerr;
```

3.2.4 Registro de un servidor en el Portmap

Después de que el servidor inicia actividades, debe registrarse así mismo en el portmap. Se usa un conjunto de procedimientos para lograr comunicación con el servicio portmap.

registerrpc( ): Registro de un procedimiento UDP con el portmap local. Cuando llega una petición válida, los procedimientos decodifican los argumentos de entrada XDR y codifican la salida en el formato XDR. Regresa 0 si hay éxito; -1 si falla.

```
int registerrpc(u_long prognum, u_long versnum,
u_long procnum, char *(*procname) ( ), xdrproc_t
inproc, xdrroc_t outproc)
```

svc_register( ): Registra un servicio/rutina de un protocolo especificado, directamente en el portmap local.

```
bool_t svc_register(SVCXPTR *xptr, u_long
prognum, u_long versnum, void (*dispatch) ( ), u_long
protocol)
```

svc_unregister( ): Borra registros del portmap local para el programa y versión requerido.

```
void svc_unregister(u_long prognum, u_long
versnum)
```

xprt_unregister( ): Retira registro de "handles" del servicio de transporte con RPC.

```
void xptr_unregister(SVCXPRT *xptr)
```

3.2.5 Administración del handle del servicio de transporte SVCXPTR

Debido a que CLIENT requiere estructuras para asociar información con el cliente, los servidores necesitan estructuras similares. La estructura de datos SVCXPRT o handle del servicio de transporte describe las comunicaciones del servidor.

svc_destroy( ): Destruye la estructura del servicio de transporte y libera recursos asociados con el handle del servicio de transporte SVCXPRT.

```
void svc_destroy(SVCXPRT *xprt)
```

svctcp_create( ): Crea y regresa un servicio de transporte TCP en un socket particular. Como su equivalente del cliente `clnttcp_create( )`, usa buffers de E/S con tamaños específicos o default. Regresa NULL si falla.

```
SVCXPRT *svctcp_create(int sock, u_int sendsz,
 u_int recvsz)
```

svcudp_create( ): Crea y regresa un apuntador a un servicio de transporte UDP. El socket se enlaza con un puerto UDP local. Regresa NULL si falla.

```
SVCXPRT * svcudp_create(int sock)
```

3.2.6 Manejo y reporte de errores en el lado servidor

Se usan para reconocer los errores del servidor. Estas rutinas son usadas por la función de despacho del lado servidor para declarar problemas de transacción del cliente, como errores de codificación XDR, autenticación, procedimiento, programa y versión, así como problemas de ejecución en general.

svcerr_decode( ): Se usa si el servicio no puede decodificar los parámetros pedidos, por ejemplo, si falla `svc_getargs( )`.

```
void svcerr_decode(SVCXPRT *xprt)
```

svcerr_no proc( ): Indica que el servicio no tiene el número de procedimiento requerido o no está disponible.

```
void svcerr_noproc(SVCXPRT *xprt)
```

svcerr_no prog( ): Indica un número de programa que no está disponible o registrado.

```
void svcerr_noprog(SVCXPRT *xprt)
```

svcerr_progvers( ): Indica un error en la versión del programa.

```
void svcerr_progvers(SVCXPRT *xprt)
```

svcerr_systemerr( ): Ha ocurrido un error indeterminado no involucrado con el protocolo, por ejemplo, memoria insuficiente.

```
void svcerr_systemerr(SVCXPRT *xprt)
```

3.2.7 Servidor E/S y utilerías

Se usan para simplificar la interfaz servidor/socket. Típicamente el servidor se encuentra verificando peticiones en un ciclo. Cuando detecta alguna, proporciona servicios usando estas funciones.

svc_freeargs( ): Libera memoria reservada XDR/RPC reservada por svc_getargs( ). Regresa TRUE si hay éxito, de otra forma FALSE.

```
bool_t svc_freeargs(SVCXPRT *xprt, xdrproc_t inproc, char *in)
```

svc_getargs(): Decodifica los argumentos XDR de la petición que llega.

```
bool_t svc_getargs(SVCXPRT *xprt, xdrprox_t inproc, char *in)
```

svc_getcaller(): Da y obtiene la dirección de red del cliente SVCXPRT.

```
struct sockaddr_in * svc_getcaller(SVCXPRT *xprt)
```

svc_run(): Se usa para atender indefinidamente peticiones de servicio al momento en el que llegan. Esta función regresa sólo en caso de ocurrir un error del que no se puede recuperar. Llama a una función select( ), con un tiempo límite infinito, produciendo una svc_getreqset( ) cuando select( ) regresa un descriptor de archivo de lectura elegible. Esto significa una petición entrante.

```
void svc_run( )
```

svc_sendreply( ): La usan las rutinas de despacho de servicio para responder al cliente , pasando los datos especificados por el codificador XDR especificado. Regresa TRUE si hay éxito, FALSE en otro caso.

```
bool_t svc_sendreply(SVCXPRT *xprt, xdrproc_t outproc, char *out)
```

3.2.8 Acceso Directo XDR

Estas rutinas de librería pueden ser usadas para implantar llamados a procedimientos remotos de bajo nivel. Las rutinas describen los mensajes RPC (información, respuesta, peticiones, etc.) en el formato neutral XDR. Estas funciones pueden ser usadas para implantar transacciones tipo RPC de ONC para llamadas a procedimientos remotos. Típicamente regresan bool_t TRUE si hay éxito, o FALSE en caso de falla.

xdr_accepted_reply(): Codifica una respuesta a RPC en la forma XDR.

```
bool_t xdr_accepted_reply(XDR *xdrs, struct accepted_reply *arp)
```

xdr_callmsg(): Crea una representación XDR de un mensaje de llamada RPC.

```
bool_t xdr_callmsg(XDR *xdrs, struct rpc_msg
*cmsgp)
```

xdr_rejected_reply(): Registra el mensaje de rechazo en formato XDR, debido a errores en la versión o de autenticación.

```
bool_t xdr_rejected_reply(XDR *xdrs, struct
rejected_reply *rrp)
```

xdr_replymsg( ): Construye un mensaje de respuesta RPC (aceptado, rechazado o NULL) en formato XDR.

```
bool_t xdr_replymsg(XDR *xdrs, struct rpc_msg
rmsgp)
```

3.3 Biblioteca de funciones generadas para el manejo del portmap

Los servicios de red del portmap mantienen el intercambio entre los servicios del programa y sus direcciones universales de puertos TCP/IP. El portmap asocia los números de puertos del protocolo TCP/IP con los números de programas RPC, proporcionando acceso a los procedimientos remotos. El servidor RPC primero registra sus programas con el servidor portmap, incluyendo el número de puerto y el número de versión del programa. El cliente consulta al servidor portmap quien le regresa el puerto para contactar al servidor apropiado y ejecutar un procedimiento específico.

Para realizar las funciones del portmap, se generó una biblioteca de funciones para administrar la base de datos que maneja el portmap y registrar los datos de los servicios (nombre de programas, versiones, procedimientos, protocolos, etc.)

pmap_getmaps () : Obtiene un mapa de las direcciones de programas de la computadora remota. Regresa una lista completa de direcciones de programas que tiene el portmap con dirección IP almacenada en `*addr`. Regresa un NULL si el servicio portmap no puede contactarse.

```
struct pmaplist *map_getmaps (struct sockaddr_in
*addr)
```

pmap_getport (): Obtiene el número de puerto de un servicio remoto. Se especifica el número de programa RPC (`prognum`), número de versión (`versnum`) y el protocolo de transporte ya sea IPPROTO_UDP ó IPPROTO_TCP (`protocol`). La función regresa la dirección del socket en la variable `addr`, la cual debe estar iniciada con el espacio de memoria requerido. La función regresa Cero si no existe el mapa o el pormap remoto no puede contactarse. Si no puede contactarse al portmap, se emplea la rutina RPC `rpc_createerr` regresando un estado de error. Si no coinciden las versiones, el portmap regresa el número de puerto con la indicación de que las versiones son diferentes

```
u_short pmap_getport (struct sockaddr_in *addr,
u_long prognum, u_long versnum, u_long protocol)
```

pmap_rmtcall (): Solicita al portmap un servicio remoto. Solicita al portmap de la dirección IP especificada en `*addr`, haga una llamada RPC a un procedimiento en esa computadora. Si el procedimiento tiene éxito regresa en la variable `*portp` el número de puerto del programa. Si la solicitud del procedimiento remoto no está registrado se envía un indicador.

```
Enum clnt_stat pmap_rmtcall (struct sockaddr_in
*addr, u_long prognum, u_long versnum, u_long procnum,
char *in, char *out, xdrproc_t inproc, xdrproc_t
outproc, struct timeval timeout, u_long *portp)
```

pmap_set (): Registra un servicio con el portmap local. Se emplea para registrar los servicios locales del pormap tales como Numero de programa, número de versión y el protocolo de transporte `[prognum, versnum, protocol]` asociado con el número de puerto `[port]`. El valor del protocolo puede ser IPPROTO_UDP IPPROTO_TCP. Esta rutina regresa TRUE si puede registrar el servicio y FALSE en caso contrario. El servidor usa esta rutina para registrarse a sí mismo con el portmap local (cuando llama a la rutina `svc_register ()`). La rutina `pmap_set ()` y `pmap_unset ()` no pueden emplearse (llamarse) a través de una computadora remota. Sólo se emplean para el registro local.

```
bool_t pmap_set (u_long prognum, u_long versnum,
int protocol, u_short port)
```

pmap_unset (): Borra un servicio registrado en el portmap. Le indica al portmap que borre los datos contenidos en [`prognum, versnum,*`]. Regresa TRUE si la rutina tuvo éxito, FALSE en otro caso.

```
bool_t pmap_unset (u_long prognum, u_long
versnum)
```

xdr_pmap (): Rutina usada para crear los parámetros de bajo nivel XDR para las llamadas al portmap. Regresa TRUE si la rutina tiene éxito y FALSE en caso contrario.

```
bool_t xdr_pmap (XDR *xdrs, struct pmap *regp)
```

*xdr_pmaplist ():*Rutina usada para crear una lista de servicios del portmap codificando dicha lista al estándar XDR. Regresa TRUE si tiene éxito y FALSE en otro caso. (la emplea `pmap_getmaps()`).

3.4 Generación de stubs cliente-servidor, encabezados y rutinas xdr

Para la creación de un programa que utiliza RPC es necesario la especificación de dicho programa en el lenguaje RPC (En el apéndice C se explica el protocolo RPC y la definición del lenguaje). Esta especificación debe contener el programa que se desea ejecutar de manera remota y las estructura de datos que utiliza. La estructura que debe seguir una especificación en el lenguaje RPC se muestra en la figura 3.2

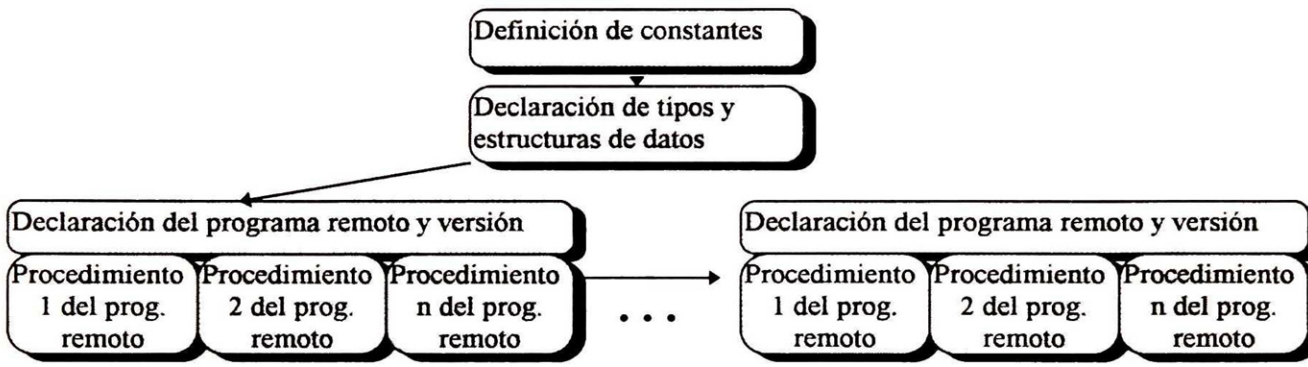


Figura 3-2. Estructura que sigue una especificación en lenguaje RPC.

En el listado 3.1 se presenta un ejemplo de una especificación en lenguaje RPC con los módulos mencionados en la figura 3.2. En el listado se explica cada una de las declaraciones. En al apéndice A y C se explican con mayor detalle la declaración de estructuras y tipos que pueden definirse en el lenguaje RPC.

3.4.1 Encabezados

Al traducir el programa de especificación RPC al lenguaje de programación C, se necesita conocer los tipos de datos y la función de cada una de las rutinas RPC para convertir dicha especificación.

Dada la especificación RPC, se requiere que los tipos declarados tengan una equivalencia en el lenguaje C. Algunas equivalencias pueden consultarse en el apéndice C.

El objetivo de crear un archivo de encabezados, es indicarle al programa cliente las declaraciones de las estructuras y las constantes necesarias para la ejecución remota. Este archivo debe incluir el archivo de encabezados de tipos de la biblioteca RPC (types). Las constantes se traducen con el comando #define. Si tomamos la constante de la especificación del listado 3.1, se traduciría al archivo de encabezados como:

```
#define MAXNOM          ((u_long) 255)
```

```
/* Esto representa un comentario en le lenguaje RPC */
const MAXNOM= 255; /* Declaración de constantes */

/* declaración de tipos y estructuras de datos*/
typedef string nombre_p <MAXNOM>;          /*declaración de un nuevo tipo
                                           (nombretipo). Todos los tipos
                                           definidos fuera de una estructura
                                           deben definirse con la instrucción
                                           typedef */

struct datos {                             /* declaración de una estructura */
        nombre_p nombre;
        string dirección <MAXNOM>;
        int clave;
}

/* declaración de programas, versiones y procedimientos*/
   Para la declaración de un programa se emplea el comando program y el comando
version para identificar el nombre del programa remoto y la versión. Después de estas
declaraciones se colocan los procedimientos que pertenecen a ese programa y versión.

Program EMPLEADOS{                         /* declaración del programa */
        version VEREMP {                   /* Declaración de la versión del programa*/
                int inserta_nombre (datos) = 1; /* Procedimientos del programa.
                                           El procedimiento inserta_nombre
                                           Regresa un entero y envía como
                                           parámetro la estructura datos.
                                           Este procedimiento está identificado
                                           con el número 1 */
                int clave_emp (nombre_p) =2; /* procedimiento 2 del programa. El
                                           procedimiento clave_emp regresa un
                                           entero y envía como parámetros el
                                           tipo nombre_p */
        } = 1;                             /* Indica en número de la versión */
} = 0x2000;                                /*indica en número del programa*/
```

Listado 3.1 Especificación en el lenguaje RPC.

La conversión al lenguaje C de los tipos de datos en RPC definidos con `typedef`, dependerá del tipo que se trate. Si se trata de un tipo simple como `int`, `char`, `long`, etc., su traducción es casi similar, mientras que para tipos complejos como `string`, `opaque`, etc. y estructuras definidas por el usuario varía su manejo. Si tomamos la especificación del listado 3.1, los tipos y las estructuras se traducen al lenguaje C como:

```
typedef char *nombre_p;
typedef struct datos {
    nombre_p nombre;
    char *dirección
    int clave;
} datos;
```

Cuando en la especificación se declaran estructuras y nuevos tipos, es necesario la creación de las rutinas XDR que realizarán la conversión de datos para el intercambio de información y compatibilidad entre computadoras. En el archivo de encabezados se declaran esas rutinas XDR, las cuales se construyen con la cadena "`bool_t xdr_`" seguida del nombre de la estructura. Esto indica que se declara una función XDR, donde se regresa el tipo `bool_t` que es un tipo booleano cuyo valor es TRUE ó FALSE dependiendo del éxito de la rutina. Las nuevas rutinas XDR declaradas, siempre llevan dos parámetros, uno indica la estructura XDR y el otro un apuntador al tipo que se está convirtiendo (el apuntador depende del tipo que se trate). Si consideramos la especificación del listado 3.1, la declaración de las rutinas XDR quedarían como:

```
bool_t xdr_nombre_p (XDR *xdrs, nombre_p *objp);
bool_t xdr_datos ( XDR *xdrs, datos *objp);
```

La definición del programa y su versión se declaran en el archivo de encabezados en el lenguaje C como constantes con el comando `#define` seguido del valor asignado en el archivo de especificación. Los nombres de procedimientos también deben declararse con el comando `#define` seguido del número de programa, para utilizarlo como referencia cuando se hagan las llamadas a procedimientos. Por ejemplo si tomamos la especificación del listado 3.1, el archivo de encabezados para estos casos quedaría como:

```
#define EMPLEADOS           ((u_long) 0x2000)
#define VEREMP              ((u_long) 1)
#define inserta_nombre      ((u_long) 1)
#define cve_emp             ((u_long) 2)
```

Definidas estas nuevas constantes, se requiere que en el archivo de encabezados se definan los nombres de los procedimientos que se emplearán para hacer las llamadas remotas. Estos procedimientos contendrán las rutinas de comunicación que permitirán las llamadas remotas como si fueran locales. Estos nombres de procedimientos se generan a partir de los procedimiento declarados en la especificación seguido de un guión bajo (_) y del número de versión del programa. Estos procedimientos se

declaran como externos con un apuntador al tipo de retorno, el nombre con la estructura mencionada y como argumentos un apuntador a la estructura de datos declarada para ese fin y un apuntador a la estructura cliente CLIENT. Si consideramos la especificación del listado 3.1 la declaración de los procedimiento sería:

```
extern int *inserta_nombre_1 (datos *argp, CLIENT *clnt);
extern int *clave_emp_1 ( nombre_p *argp, CLIENT *clnt);
```

3.4.2 Rutinas XDR

Por cada uno de lo nuevos tipos y estructuras que se definan, se debe generar una rutina XDR para convertir estos nuevos tipos al estándar XDR. En el apéndice A se explica el protocolo XDR.

Para la generación de las nuevas rutinas XDR, se emplea la biblioteca XDR que contiene rutinas `xdr` básicas para datos simples y complejos. Cada rutina ó filtro XDR regresa un tipo booleano `bool_t` que tiene los valores de TRUE o FALSE si la rutina tiene éxito o no. El nombre de la nueva rutina XDR se forma de la cadena "`bool_t xdr_`" y el nombre de la estructura. Los argumentos de la rutina son: la estructura XDR y un apuntador a la estructura que se está manejando. La declaración de las rutinas XDR para los tipos de datos de la especificación del listado 3.1 se vería como:

```
bool_t xdr_nombre_p (XDR *xdrs, nombre_p *objp);
bool_t xdr_datos ( XDR *xdrs, datos *objp);
```

Si la rutina XDR que se va a generar es una estructura, se codifica cada elemento de la estructura con las rutinas básicas de la biblioteca XDR. Por ejemplo la declaración `typedef string nombre_p <MAXNOM>` (en el lenguaje RPC) ó `typedef char *nombre_p` (que es su equivalencia en el lenguaje C) genera la rutina XDR del listado 3.2:

```
bool_t   xdr_nombre_p ( XDR *xdrs, nombre_p *objp){
          if (!xdr_string(xdrs, objp, MAXNOM)) {
                return (FALSE);
          }
          return (TRUE);
}
```

Listado 3.2 Rutina XDR de la estructura nombre_p

la rutina básica `xdr_string` se emplea porque la estructura que se está convirtiendo (`nombre_p`) es del tipo `string`.

De la misma forma, si tenemos la siguiente declaración de estructura en el lenguaje RPC:

```
struct datos {
        nombre_p nombre;
        string dirección <MAXNOM>;
        int clave;
}
```

ó su equivalencia en el lenguaje C

```
typedef struct datos {
    nombre_p nombre;
    char *dirección
    int clave;
} datos;
```

la rutina XDR generada se muestra en el listado 3.3:

```
bool_t   xdr_datos( XDR *xdrs, datos *objp){
          if (!xdr_nombre_p(xdrs, &objp->nombre)) {
                return (FALSE);
          }
          if (!xdr_string(xdrs, &objp->dirección, MAXNOM)) {
                return (FALSE);
          }
          if (!xdr_int(xdrs, &objp->clave)) {
                return (FALSE);
          }
    return (TRUE);
}
```

Listado 3.3. Rutina XDR generada de la estructura datos

El compilador CRAD genera automáticamente estas rutinas XDR. Con la práctica el programador puede implementar sus propias rutinas XDR, usando la biblioteca XDR.

3.4.3 CLIENTE

Para realizar un programa que emplee el modelo RPC es necesario contemplar dos módulos principales: el cliente y el servidor. El cliente emplea rutinas de comunicación para establecer sesiones remotas con el servidor. Para enviar la información a través de las rutinas de comunicación, se necesita codificar los datos del cliente en el formato estándar XDR. Para esto, se generan las rutinas XDR necesarias. El cliente hace uso del servidor portmap para solicitar el puerto donde se localiza el servicio (el programa remoto) requerido.

3.4.3.1 *Código principal del Cliente (main)*

Para la programación de un cliente se puede emplear el compilador CRAD para que genere a partir de la definición del programa RPC, de manera automática, el stub del cliente, los encabezados y las rutinas XDR. Otra forma de generar el cliente es emplear directamente las bibliotecas XDR y RPC y programar con ellas un cliente.

Cualquiera que sea la elección del programador, se requiere la modificación del código principal del cliente donde debe incluir algunas rutinas para realizar las llamadas remotas así como la declaración de algunas variables para estas rutinas.

Un cliente requiere:

- Conocer el procedimiento que desea ejecutar de manera remota.
- Identificar la máquina donde solicitará el servicio (obtener su dirección IP)
- Determinar un tiempo de espera para recibir la respuesta del servidor
- Crear una estructura donde codificará los datos del procedimiento que serán enviados.
- Realizar la llamada remota y decodificar los resultados.

Si el programador desea escribir sus propias rutinas, entonces debe declarar en el código del programa principal del cliente algunas variables (como `int sock, struct sockaddr_in server_addr,`) crear una estructura cliente (`register CLIENT *client`) donde residirá información importante sobre el tipo de servicio que se está solicitando, el programa remoto, el puerto creado para el cliente, etc. Además deberá iniciar un tiempo de espera por una respuesta y declarar las variables que requieran algunas funciones que desee emplear.

Una forma de simplificar las tareas mencionadas es que en el código principal del cliente se emplee la rutina `clnt_create` para iniciar la estructura cliente con los datos del servidor solicitado, el programa remoto, la versión y el protocolo que desea. De cualquier manera es necesario la declaración de la estructura cliente para emplear esta rutina.

La rutina `clnt_create` establece una comunicación con el servidor portmap en el puerto 111 por medio del protocolo UDP. El portmap maneja una base de datos donde se registra sus servicios. La rutina solicita que le regrese el puerto donde tiene registrado el programa, la versión y el protocolo enviados como parámetros. Si ocurre un error se emplea la rutina `clnt_pcreateerror` para imprimir la causa de la falla. Si no hubo error, la rutina `clnt_create` regresa un apuntador a la estructura CLIENT, donde se almacena información importante para la comunicación con el puerto indicado (tal es el caso de la dirección del puerto y un socket asociado). Posteriormente el cliente puede establecer una sesión remota a través del puerto recibido y solicitar la ejecución de cualquiera de los procedimientos remotos que pertenecen al programa (por ejemplo `inserta_nombre_1, clave_emp_1` que son los procedimientos generados de la especificación RPC).

La secuencia de pasos se esquematiza en la figura 3.3

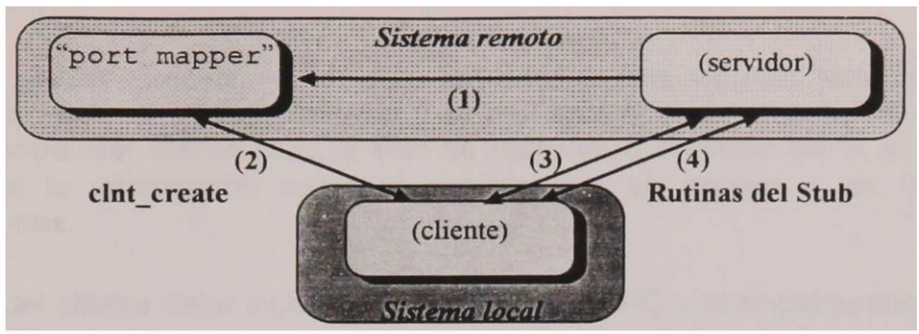


Figura 3-3 ***Pasos para una llamada remota: (1) El servidor se registra en el portmapper cuando inicia y declara todos los servicios que ofrece. (2) Cuando el cliente solicita un programa primero pregunta al portmapper para verificar su existencia. Si existe el portmapper le envía el puerto en donde se localiza el servicio. (3) El cliente establece una sesión con el servidor en el puerto recibido y hace las llamadas a las rutina que necesita***

`clnt_create` usa las rutinas XDR para enviar los datos del servicio solicitado. En la figura 3.4 se muestra una estructura general del servidor empleando la rutina `clnt_create`. En las siguientes líneas se presenta un ejemplo del código:

```
cl=clnt_create (servidor, EMPLEADOR, VEREMP, "tcp")
if (cl==NULL) {
    clnt_pcreateerror(servidor);
    exit(1);
}
/* llamar a la rutina remota */
```

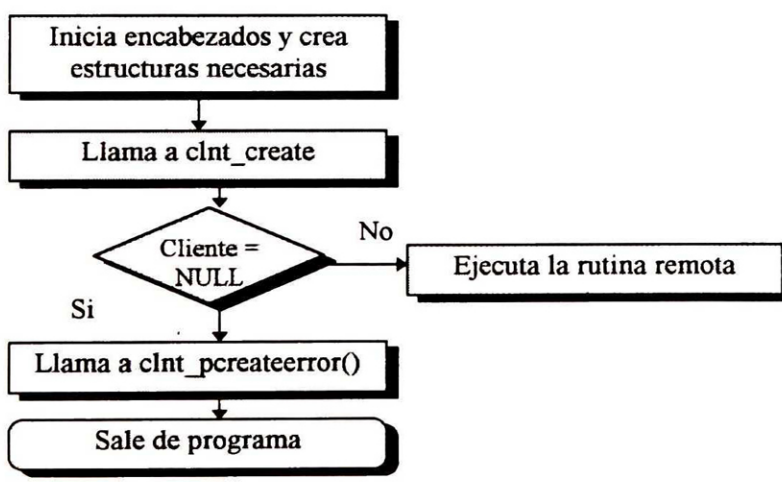


Figura 3-4. Estructura general de un cliente con la rutina clnt_create ()

3.4.3.2 *Stub del cliente*

El stub es el procedimiento que se encarga de ocultar todo el proceso de comunicación con el servidor remoto y la realización de las llamadas remotas. En el código principal del cliente (`main`) sólo se realizan la creación de la estructura cliente conteniendo la información sobre el servicio que se solicita y las llamadas a los procedimientos.

El stub del cliente debe incluir los encabezados RPC y el encabezado generado con la declaración de las estructuras y procedimientos. Debe declarar una variable de temporización para recibir la respuesta por parte del servidor. En el caso de que el tiempo se termine y el cliente no reciba respuesta, se detecta un error (que puede indicar alguna perdida de conexión). Esta variable se declara con la estructura timeval de la siguiente forma:

```
static struct timeval TIMEOUT = {25, 0}
```

Para la declaración de los procedimientos remotos, se emplean dos parámetros, uno que indica el argumento del procedimiento y otro la estructura cliente CLIENT. Se debe declarar e iniciar una variable donde se reciba la respuesta o resultado de la llamada al procedimiento remoto. Para la llamada al procedimiento remoto se emplea la rutina `clnt_call` , la cual requiere como argumentos:

- Un apuntador a la estructura cliente creada
- El nombre del procedimiento remoto
- La rutina xdr que se empleará para codificar/decodificar el argumento del procedimiento
- la variable donde se almacenan los argumentos,
- la rutina xdr que codificará/decodificará el resultado de la llamada
- La variable donde se almacenará el resultado
- El tiempo de espera del cliente.

Si la rutina no tiene éxito (es decir, no regresa RPC_SUCESS) entonces regresa NULL a la función que la llamó. En caso contrario regresa la variable donde se localiza la respuesta a la llamada.

El nombre de los procedimiento remotos de la especificación del listado 3.1 se declara como:

```
extern int *inserta_nombre_1 (datos *argp, CLIENT *clnt);
extern int *clave_emp_1 ( nombre_p *argp, CLIENT *clnt);
```

La declaración del primer procedimiento se presenta en el listado 3.4:

```
int  *inserta_nombre_1 (datos *argp, CLIENT *clnt){
    static int res;
    bzero ((char *) &res, sizeof(res));
    if (clnt_call (clnt, inserta_nombre, xdr_datos,argp, xdr_int, &res,
       TIMEOUT) != RPC_SUCCESS){
          return (NULL);
    }
    return (&res);
}
```

Listado 3.4 Declaración de procedimiento en el stub del cliente

La rutina `bzero` inicia la variable de respuesta. Para la llamada `clnt_call` los datos enviados son:

- La estructura cliente *clnt*
- El nombre del procedimiento remoto *inserta_nombre*
- La rutina xdr que se empleará para codificar/decodificar el argumento del procedimiento *xdr_datos*
- la variable donde se almacenan los argumentos *argp*
- la rutina xdr que codificará/decodificará el resultado de la llamada *xdr_int*
- La variable donde se almacenará el resultado *res*
- El tiempo de espera del cliente *TIMEOUT*

De igual manera se declara el segundo procedimiento (listado 3.5):

```
extern int *clave_emp_1 ( nombre_p *argp, CLIENT *clnt);
```

```
int  *clave_emp_1 (nombre_p *argp, CLIENT *clnt){
    static int res;
    bzero ((char *) &res, sizeof(res));
    if (clnt_call (clnt, clave_emp, xdr_nombre_p,argp, xdr_int, &res,
       TIMEOUT) != RPC_SUCCESS){
          return (NULL);
    }
    return (&res);
}
```

Listado 3.5. Declaración de procedimiento en el stub cliente

3.4.4 SERVIDOR

El código del servidor, está formado por 2 módulos

- El stub del servidor
    - Iniciación del portmap
    - Iniciación de los servicios
    - Ejecución indefinida
    - Programa despachador
- Rutinas locales del servidor.

El stub del servidor se encargará de iniciar el portmap, registrar sus servicios en una base de datos, recibir y atender las peticiones remotas. El portmap en el caso del CRAD se inicia cuando el servidor se ejecuta. En el caso de SUN el portmap es un proceso que se inicia desde que la computadora se enciende y conserva la información de los servicios. *El despachador* en el stub se encarga de ejecutar las rutinas locales (procedimientos declarados en la especificación RPC de un programa) , codifica y decodifica cuando sea necesario, envía el resultado al cliente y libera el espacio empleado con los argumentos. Por cada programa declarado en la especificación RPC, se genera un despachador. El nombre del despachador tiene la siguiente estructura "nombre del programa _versión del programa"

Las rutinas locales del servidor, las realiza el programador y las ejecuta el stub.

3.4.4.1 *Stub del Servidor*

El stub del servidor debe incluir los encabezados `<stdio.h>`, `<rpc.h>`, <encabezado generado >. Este último es el encabezado donde declaramos las estructuras y constantes de los programas remotos que se van a ejecutar. Se debe declarar el programa despachador e iniciar el programa principal. Para la especificación del programa RPC del listado 3.1 estas declaraciones se insertarían como:

```
#include <stdio.h>
#include <rpc.h>
#include <pr.h>
static void empleados_1();
```

Las rutinas que emplea el stub del servidor son (consultar la librería RPC para mayor información):

`potmap_reg()` encargada de iniciar el proceso del servidor, registrándose a sí mismo en la base de datos en el puerto 111 con todos sus procedimientos.

`portmap_unset ()`. Elimina en la base de datos del portmap un registro previo del programa y la versión que se está instalando.

`svcudp_create()`. Inicia la estructura del servidor creando un puerto para el protocolo de transporte UDP. Cuando crea el puerto se emplea la rutina `svc_register()` para registrar el servicio (programa, versión, rutina de despacho y

protocolo) junto con el puerto del protocolo UDP creado (estructura SVCXPRT). Lo mismo se hace para el protocolo de transporte TCP (empleando la rutina **svctcp_create ()**). Registrados los servicios inicia el proceso **svc_run()** que se ejecuta de manera indefinida hasta que un error del sistema lo interrumpa. Este proceso consulta las solicitudes de llamadas remotas que llegan a los puertos de los programas registrados (servicios), establece nuevos puertos (sockets) para atender a varias solicitudes cuando la conexión establecida sea TCP (esclavos) y ejecuta el despachador que le corresponde. En la figura3.5 se presenta un diagrama de los pasos mencionados.

Tomando la especificación del listado 3.1 (mostrada en las líneas donde declaramos lo siguiente:

Las siguientes líneas forman parte de la especificación del listado 3.1. De estas líneas se genera el programa principal del stub del servidor que se presenta en el listado 3.6.

```
Program EMPLEADOS{
          version VEREMP {
                int  inserta_nombre (datos) = 1;
                int clave_emp (nombre_p) =2;
          } = 1;
} = 0x2000;
```

La estructura SVCXPRT del listado 3.6, declara una estructura que maneja el tipo de servicio de transporte solicitado. En este caso se emplea para los dos protocolos (UDP, TCP). La constante RPC_ANYSOCK indica que el puerto sobre el cual se montará el servicio se determinará automáticamente (un puerto disponible válido). Si no se desea de esta forma, en el lugar de esta constante se indica un número de puerto válido.

El despachador tiene declarados los argumentos que emplean cada procedimiento, los resultados que generan y el nombre de la rutina local que debe ejecutarse. El despachador emplea la rutina **svc_getargs()** para obtener los argumentos de la petición recibida, esta rutina emplea las rutinas XDR para decodificar los datos. Si no es posible obtener los argumentos se emplea la rutina **svcerr_decode ()** para indicar el error.

Si los argumentos se obtienen, entonces el despachador ejecuta el procedimiento solicitado. Envía el resultado al cliente por medio de la rutina **svc_sendreply ()**. Esta rutina emplea nuevamente las rutinas XDR para codificar los resultados. Si existe un error en la transmisión se emplea la rutina **svcerr_systemerr ()** para indicarlo. Si no existe error se liberan los argumentos por medio de la rutina **svc_freeargs ()** y se regresa a la rutina svc_run ()

En el caso de que el cliente solicite un procedimiento que no está registrado, se emplea la rutina **svcerr_noproc ()** indicando el error.

```
main () {
   register SVCXPRT *transp;
   portmap_reg ();
   (void) pmap_unset (EMPLEADOS, VEREMP);

   transp = svcudp_create (RPC_ANYSOCK);
   if (transp == NULL) {
            fprintf(stderr, "no se puede crear el servicio UDP");
            exit(1);
   }
   if (!svc_register (transp, EMPLEADOS,VEREMP,empleados_1,
       IPPROTO_UDP)){
            fprintf(stderr, "no se puede registrar el servicio
                  [EMPLEADOS, VEREMP, udp]");
            exit(1);
   }
   transp = svctcp_create (RPC_ANYSOCK, 0, 0);
   if (transp == NULL) {
            fprintf(stderr, "no se puede crear el servicio TCP");
            exit(1);
   }
   if (!svc_register (transp, EMPLEADOS,VEREMP,empleados_1,
       IPPROTO_TCP)){
            fprintf(stderr, "no se puede registrar el servicio
                  [EMPLEADOS, VEREMP, tcp]");
            exit(1);
   }
   svc_run ()
   fprintf(stderr, "El programa svc_run regresó");
   exit(1);
}
```

Listado 3.6 Programa principal del stub del servidor

El código para el despachador se presenta en el listado 3.7

```
  static void empleados_1 (struct svc_req *rqstp, register SVCXPRT
*transp){
        union {
            datos inserta_nombre_1_arg;
            nombre_p clave_emp_2_arg;
        }
        char *result;
        bool_t (*xdr_argument)(), (*xdr_result)();
        char *(*local)();
        switch (rqstp->rq_proc) {
              case NULLPROC:
                  (void) svc_sendreply (transp, xdr_void, (char *)NULL);
                  return;
              case inserta_nombre:
                    xdr_argument=xdr_datos;
                    xdr_result =xdr_int;
                    local = (char *(*)()) inserta_nombre_1;
                    break;
              case clave_emp
                    xdr_argument=xdr_nombre_p;
                    xdr_result =xdr_int;
                    local = (char *(*)()) clave_emp_1;
                    break;
              default:
                    svcerr_noproc(transp);
                    return;
              }
        bzero ((char*) &argument, sizeof (argumento));
        if (!svc_getargs (transp, xdr_argument, &argument)){
              svcerr_decode (transp);
              return;
        }
        result= (*local)(&argument, rqstp);
        if (result!=NULL && !svc_sendreply (transp, xdr_resul,
      result)){
              svcerr_systemerr (transp);
        }
        if ("svc_freeargs (transp, xdr_argument, &argument)) {
              fprintf (stderr, " No se pueden liberar los argumentos");
            exit (1);
        }
        return;
  }
```

Listado 3.7. Código para el despachador del stub del servidor

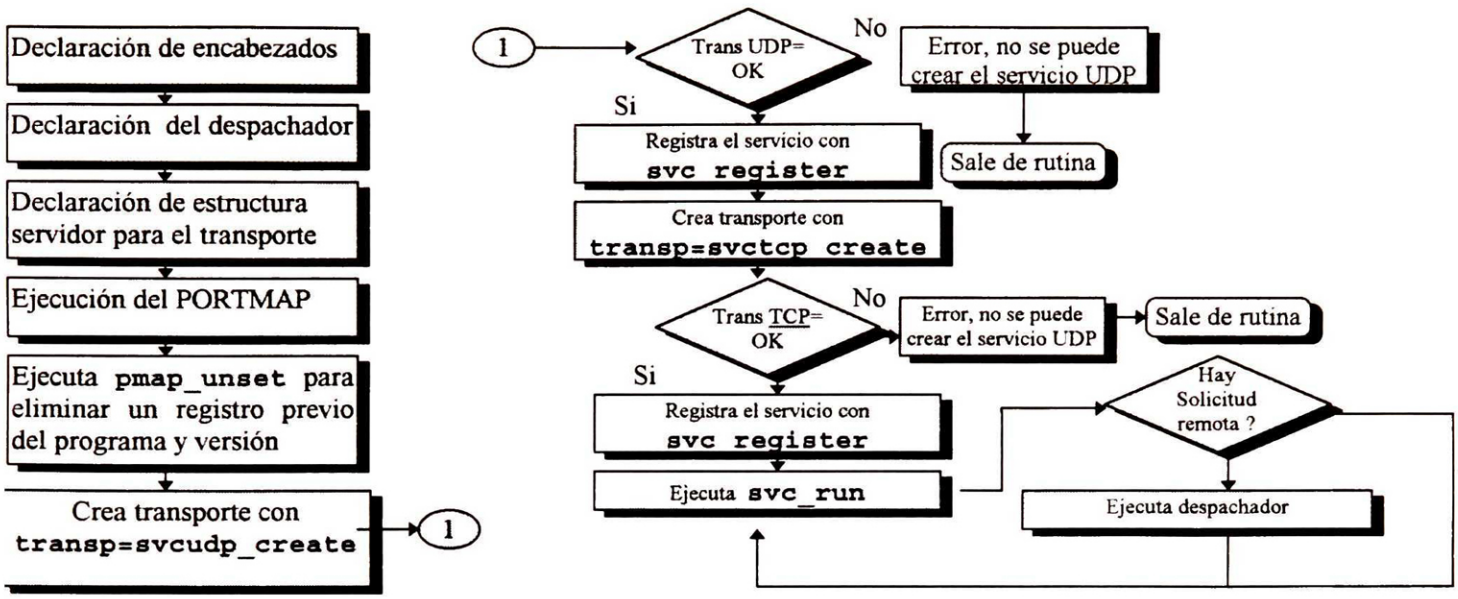


Figura 3-5 Pasos ejecutados por el stub del servidor

3.4.4.2 Rutinas del servidor (procedimientos remotos)

Las rutinas del servidor las realiza el programador. Estas rutinas son llamadas por el stub del servidor a través de una llamada local, dependiendo de la rutina que el cliente haya solicitado. Es este caso no se mostrará el código de estas rutinas. En el capítulo 6 se presenta un ejemplo completo.

4

4. INTERFAZ DE SOCKETS E INTERFAZ DE COMUNICACIÓN CRAD

En este capítulo se hablará de la interfaz de sockets empleada para la realización de las rutinas de comunicación, el tipo de cliente y servidor diseñado así como los algoritmos de las funciones más importantes.

Para mayor referencia de los tipos de clientes y servidores se puede consultar el apéndice B.

4.1 Uso de la interfaz de sockets para clientes y servidores

La interfaz de socket es un mecanismo que permite a los programas de aplicación conectarse con el software de los protocolos de comunicaciones. Esta interfaz es conocida como API (Applications Program Interface). El API empleado para el desarrollo de este trabajo fue PathWay API versión 1.1 compatible con Microsoft C versión 5.1.

Esta interfaz nos proporciona las herramientas necesarias para desarrollar software de aplicación basado en TCP/IP.

Los sockets se clasifican por el servicio deseado: socket de cadena de caracteres o "streams", que emplea el protocolo TCP y socket de datagramas que emplea el protocolo UDP En la figura 4.1 se ilustran la secuencia de llamadas al API del software de comunicaciones que realiza un cliente y un servidor usando TCP.

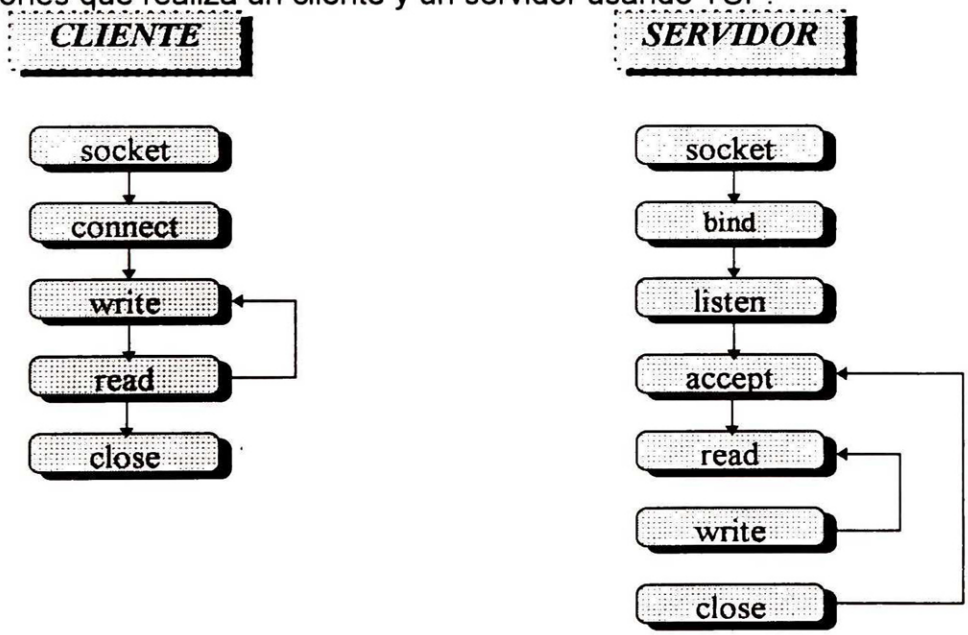


Figura 4-1 Ejemplo de la secuencia de llamadas vía interfaz de sockets hechas por el cliente y el servidor usando TCP. El servidor corre indefinidamente , esperando por nuevas conexiones en un puerto definido. Este acepta cada nueva conexión, procesa la solicitud del cliente y cierra la conexión.

El cliente crea un **socket** y llama a **connect** para conectarse al puerto apropiado en el servidor. Una vez establecida la conexión puede enviar y recibir información (**write y read**) hacia/desde el servidor. Cuando termina de utilizar la conexión, el cliente llama a **close.**

En el lado del servidor se usa **bind** para especificar un puerto local que usará el servidor, se llama a **listen** para iniciar el tamaño de la cola de conexiones y entonces comienza un ciclo. Dentro de este ciclo, el servidor llama a **accept** para esperar la siguiente solicitud de conexión. Una vez establecida la conexión se usa **read y write** para intercambiar información con el cliente y finalmente se usa **close** para terminar la conexión. El servidor regresa a la llamada **accept** donde espera la siguiente conexión. [Woll91].

Para manejar la interfaz de sockets todo programa (cliente o servidor) llama a la rutina `socket` para crear un socket y obtener su identificador. Los argumentos que se utilizan para llamar a esta rutina son: la familia de protocolos que utilizará, el tipo de comunicación requerido y la identificación del protocolo. Todos los protocolos TCP/IP son parte de la familia Internet, especificados con una constante simbólica PF_INET. En la siguiente línea se presenta una declaración de socket:

```
s=socket (AF_INET, SOCK_DGRAM,0)
```

La estructura socket permite que cada familia de protocolos defina una o más representaciones de direcciones. Todos los protocolos de TCP/IP usan la familia de direcciones Internet representada por la constante AF_INET, la cual especifica que una dirección de un punto extremo de comunicación contiene una dirección IP y un número de puerto de protocolo. Cuando una aplicación especifica un extremo de comunicación con una dirección IP y un número de puerto, utiliza una estructura definida llamada `sockaddr_in`.

En resumen las llamadas al API de sockets permiten a la aplicación especificar una dirección de punto extremo local (llamando a `bind`) para forzar a un socket dentro del modo pasivo a que sea usado por un servidor (llamando a `listen`) o para forzar al socket en modo activo para que lo use un cliente (llamando a `connect`). Los servidores pueden hacer mas llamadas para obtener nuevas solicitudes de conexiones (`accept`) y el cliente y el servidor pueden enviar y recibir (llamando a `read o write`) información. Finalmente cliente y servidor pueden liberar el socket una vez que ha finalizado su uso (llamando a `close`).

4.1.1 Biblioteca de Sockets

Creación de un Socket:

`socket` (): Creación de un socket. La variable `type` indica el tipo de servicio. Puede ser SOCK_STREAM para sockets streams y SOCK_DGRAM para sockets datagram. La variable protocolo se inicia en 0 para los dos tipos.

```
S = socket (af, type, protocol);
```

Inicia un socket en modo no bloqueado:

`fcntl` (): Inicialización de un socket en modo no bloqueado. Un socket puede usarse en dos modos: bloqueado y no bloqueado. Cuando se crea un socket, por default se crea en modo bloqueado, indica que una llamada hecha en el socket no termina hasta que un evento apropiado ocurra o se presente un error. Por ejemplo la llamada `recv()` espera hasta que los datos lleguen. En un modo no bloqueado la llamada regresa inmediatamente, si ocurrió o no un evento. El programador indica el paso que seguirá. Requiere como argumentos el socket, el comando `F_SETFL` y la indicación de bloqueo `FNDELAY`.

```
fcntl (s, F_SETFL, FNDELAY)
```

Asignar un nombre al socket:

`bind` (): Permite dar nombre al socket. Cuando se crea al socket, se crea sin nombre, es decir, no tiene asociación con una dirección local o remota. La función `bind` permite tal asociación. Requiere como argumentos el descriptor de socket (s), una dirección local a la cual el socket se va a enlazar (`addr`) y la longitud (en bytes) de la dirección (`addrlen`).

```
bind (s, addr, addrlen)
```

La variable `addr` es una estructura del tipo `sockaddr_in`:

```
struct sockaddr_in {
        short                   sin_family;
        u_short             sin_port;
        struct in_addr          sin_addr;
        char                sin_sin_zero[8];
}
```

Ejemplo:

```
struct sockaddr_in      sin;
sin.sin_family= AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons (MYPORT);
bind (s, (struct sockaddr_in *)&sin, sizeof (sin));
```

TCP/IP especifica una representación estándar para enteros binarios usados en los encabezados del protocolo. La representación conocida como "network byte order", representa enteros con el primer byte mas significativo. Las rutinas de sockets incluyen varias funciones que convierten de "network byte order" a el orden del host local. Los programas siempre deben llamar a las rutinas de conversión, haciendo de esta manera un código fuente portable.

Las funciones `htons` y `ntohs` convierten un entero corto del orden del byte nativo del host a network byte order y viceversa. Similarmente `htonl` y `ntohl` convierten enteros largos desde el orden del byte nativo del host a network byte order y viceversa. [commer93].

Conexión stream

Una conexión stream sigue las funciones mostradas en la figura 4.1

Cliente :

Inicia una conexión:

`connect` (). Conecta al cliente. Esta función necesita el descriptor del socket, una estructura para especificar la dirección destino y el número de puerto (`name` del tipo `sockaddr`) y la longitud en bytes de la estructura que contiene la dirección destino (`namelen`)

```
connect (s, name, namelen)
```

Servidor

El servidor debe ejecutar dos pasos (después de que la creación del socket y el enlace (`bind`) han terminado) para responder a la acción del cliente: listen () y `accept` ().

`Listen` (): Esta llamada es exclusivamente para los sockets stream. Ejecuta dos tareas importantes: Permite al servidor prepararse para conexiones entrantes e informa al kernel cuando múltiples solicitudes simultáneas llegan a un socket y deben ser puestas en una cola. La llamada acepta como argumentos, el socket y un número que especifica la longitud de la cola de solicitudes.

```
Listen (s, backlog)
```

`accept`(): Después de que un socket ha sido iniciado con listen (), el servidor debe permitir aceptar conexiones. La llamada a esta función se bloquea hasta que una solicitud de conexión llegue. `accept` requiere el descriptor del socket que espera, la dirección y la longitud de la dirección. El sistema crea un nuevo socket y un nuevo descriptor para que atienda la solicitud del cliente:

```
newsock = accept (s, addr, addrlen)
```

Transferencia de datos

Después de la creación y el establecimiento de una conexión, El socket puede ser usado para transmitir datos. Pueden emplearse `send()`, `sendto()`, `recv()` y `recfrom()` que son análogas a `write()` y `read()` de UNIX.

`Send`() : Los argumentos que requiere esta llamada son: descritor de socket, apuntador al mensaje que será enviado, longitud del mensaje y las banderas, que en este momento no son soportadas (deben iniciarse con 0).

```
Send (s, buf, len, flags)
```

`recv` . Recibe datos desde un socket conectado.

```
recv (s, buf, len, flags)
```

Fin de conexión

`sock_close` (): Descarta un socket. Recibe como parámetro el descriptor

```
sock_close(s)
```

Conexión datagrama

Un socket de datagramas usa el servicio UDP. A diferencia del socket stream que usa TCP. La comunicación entre sockets de datagramas no requiere del establecimiento de conexiones. Estos sockets son conocido como sockets sin conexión. Cada mensaje que se envía entre sockets de datagramas incluye la dirección destino.

La creación de un socket, el enlace (`bind()`), son similares a los de una conexión stream. La única diferencia es que para el envío y recepción de datos se emplean las llamadas `sendto` () y `recvfrom`().

En la tabla 4.1 se presenta una lista de todas las llamadas del API empleado.

4.2 *Descripción del modelo cliente-servidor generado por CRAD*

Las rutinas de comunicación que utilizan los stubs que genera CRAD y que se encargan de generar los punto extremos de comunicación, establecer sesiones, enviar y recibir información, ejecutar el servidor para esperar solicitudes, etc., permiten implantar un modelo especial de cliente y servidor que cumple con las características de manejar un sólo proceso tanto para el cliente como para el servidor (multiprotocolo y multiservicio).

Este software está diseñado para las computadoras personales que manejan un solo proceso. No se desarrolló un software adicional para la simulación de concurrencia. Sin embargo solo se necesitarían algunas rutinas adicionales para que funcionara en un ambiente multiprocesos.

El servidor generado por CRAD es multiprotocolo porque tiene la capacidad de registrar servicios basados en comunicación orientada a conexión (TCP) y sin conexión (UDP). Siguiendo la lógica de los programas se podrían anexar protocolos de transporte.

El servidor es multiservicio porque soporta la ejecución de diversos programas, es decir, puede ofrecer varios servicios registrados en distintos puertos y atender a cada uno por medio de prioridades manejadas por colas.

El servidor tiene a su vez un "proceso" de atención a cada servicio dependiendo del protocolo de transporte indicado en cada solicitud.

Como se observa en la figura 4.3 (tomada del apéndice B), para diseñar un servidor orientado a conexión (TCP) que maneje concurrencia (no es el caso de CRAD), se utiliza el modelo maestro - esclavo. El maestro se encarga de recibir las solicitudes de conexión y de generar procesos esclavos para que atiendan tales solicitudes.

Nombre de rutina	Descripción
accept	Acepta una conexión en un socket
bind	Enlaza un nombre a un socket
connect	inicia una conexión en un socket
endhostent	Cierra el archivo de base de datos del host
endnetent	Cierra el archivo de base de datos de networks
endprotoent	Cierra el archivo de base de datos de protocolos
endservent	Cierra el archivo de base de datos de services
fcntl	Inicia un socket en modo no bloqueado
gethostbyaddr	Obtiene de la base de datos host la identificación del servidor
gethostbyname	Obtiene de la base de datos host la identificación del servidor
gethostent	Obtiene de la base de datos host la identificación del servidor
gethostid	Obtiene un identificador unicao
gethostname	Obtiene el nombre del host actual
getnetbyaddr	Obtiene una entrada del archivo network
getnetbyname	Obtiene una entrada del archivo network
getnetent	Obtiene una entrada del archivo network
getprotobyname	Obtiene una entrada del archivo protocol
getprotobynumber	Obtiene una entrada del archivo protocol
getprotoent	Obtiene una entrada del archivo protocol
getservbyname	Obtiene una entrada del archivo service
getservbyport	Obtiene una entrada del archivo service
getservent	Obtiene una entrada del archivo service
getsockname	obtiene información acerca de un socket
listen	inicia un socket en modo pasivo
lselect	Multiplexaje de entrada/salida síncrono
recv	Recibe un mensaje
recvfrom	Recibe un mensaje
rresvport	Regresa un puerto UDP/TCP reservado
select	Multiplexaje de entrada/salida síncrono
send	Envía un mensaje desde un socket
sendto	Envía un mensaje desde un socket
sethostent	Abre y reorganiza el archivo de host
sethostname	Inicia el nombre del host actual
setnetent	Abre y reorganiza el archivo networks
setprotoent	Abre y reorganiza el archivo protocol
setservent	Abre y reorganiza el archivo services
socket	Crea un socket
sock_close	cierra un socket
socket_init	Incia la biblioteca de sockets

Tabla 4-1: Listado de las llamadas soportadas por el API de pathway.

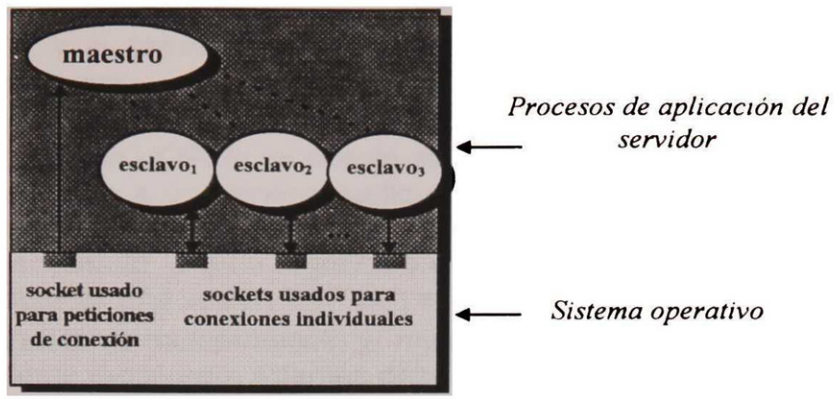


Figura 4-2 Servidor concurrente orientado a conexión

En la Figura 4.4 se ilustra el diseño de un servidor con un sólo proceso, manteniendo el esquema maestro-esclavo:

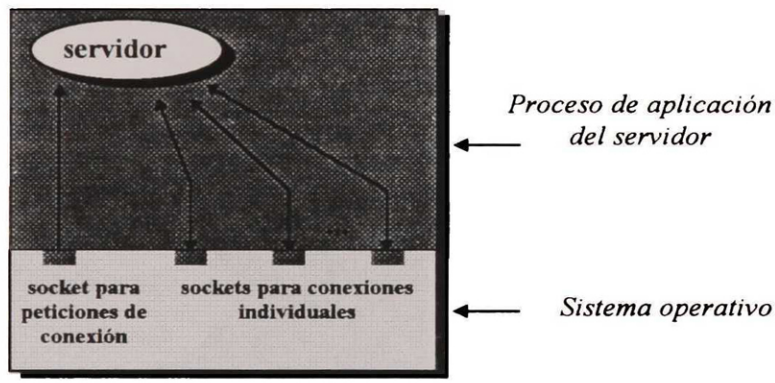


Figura 4-3 Servidor de un solo proceso orientado a conexión

En la figura 4.3 también está presente un socket sobre el cuál llegarán las solicitudes de conexión, de la misma manera el servidor crea sockets esclavos para atender las solicitudes. La creación de los sockets esclavos solo se realiza cuando el protocolo de transporte es TCP, en el caso de UDP no es necesario la generación de nuevos sockets para atender las solicitudes. Estas se manejan a través de una cola de atención y se responden sobre el mismo socket asignado para el servicio. El diseño del servidor multiprotocolo con un solo proceso se presenta en la figura 4.5

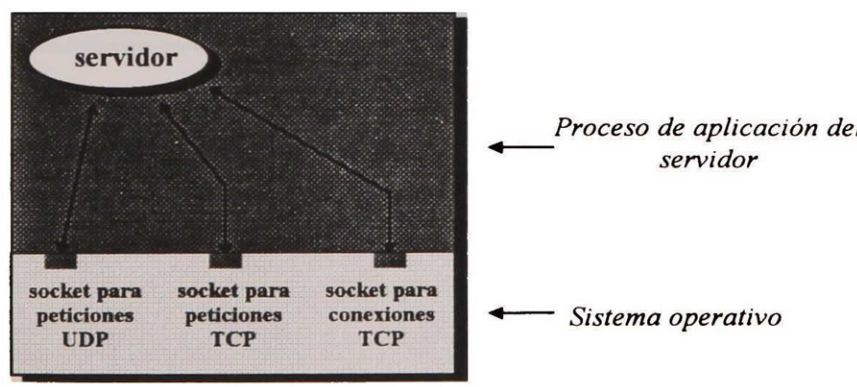


Figura 4-4 Servidor de un solo proceso multiprotocolo

En la figura 4.4. se identifican el socket dedicado a las peticiones UDP, el socket para las peticiones TCP y el socket para el establecimiento de conexiones TCP (para los esclavos que se generen).

En el diseño propuesto en esta tesis, el portmap está registrado en el puerto 111 bajo el protocolo de transporte UDP. Si un cliente solicita al portmap que le indique el número de puerto TCP asociado a un servicio particular , tiene que emplear el protocolo UDP para establecer contacto en el puerto 111 con el portmap. El portmap por su parte le regresa el puerto solicitado (si está registrado el servicio para tal protocolo). Entonces el cliente puede establecer una sesión TCP con el puerto del protocolo especificado.

El diseño propuesto es multiprotocolo y multiservicio, en donde existe un socket por cada servicio que este registrado. Además debe generarse el socket para cada protocolo de transporte soportado. Por ejemplo, un socket UDP para el servicio X, y un socket TCP para el mismo servicio, aunque se puede definir un solo protocolo para un servicio. Cuando se trata de un protocolo orientado a conexión el socket maestro sólo acepta la conexión dependiendo de su capacidad en la cola de conexiones aceptadas y genera un socket esclavo para que se atienda la solicitud y se envíe al cliente la respuesta. En la figura 4.6 se esquematiza de manera general, el modelo del servidor diseñado en esta tesis.

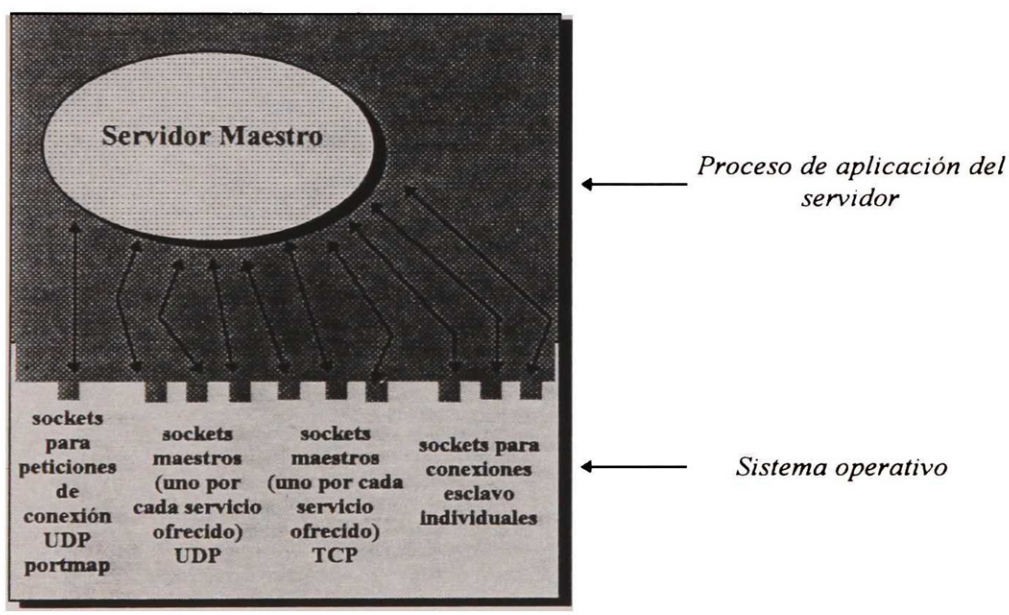


Figura 4-5 Modelo del servidor diseñado en esta tesis

4.3 Interfaz de comunicación CRAD

Las rutinas RPC para el cliente y el servidor y las rutinas XDR, emplean la interfaz de sockets para establecer una conexión, generar el servidor, atender las solicitudes en el puerto, enviar y recibir información entre otras aplicaciones. En esta sección se explicará el empleo de dicha interfaz y algunas de las rutinas más importantes diseñadas para el compilador CRAD.

4.3.1 INTERFAZ DEL CLIENTE

Para la programación del cliente que efectúa una llamada remota, se requiere la modificación del programa principal del cliente, el stub del cliente, las rutinas XDR y los encabezados. En la figura 4.6 se presenta un esquema general de las llamadas que realiza un cliente.

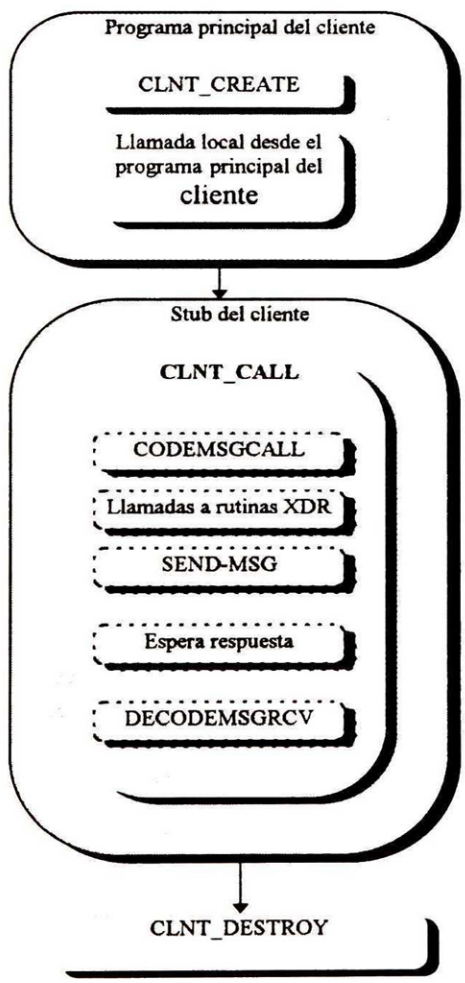


Figura 4-6 Rutinas involucradas en la creación de un cliente.

A continuación se explican algunas de las rutinas más importantes que se emplean para la construcción del cliente.

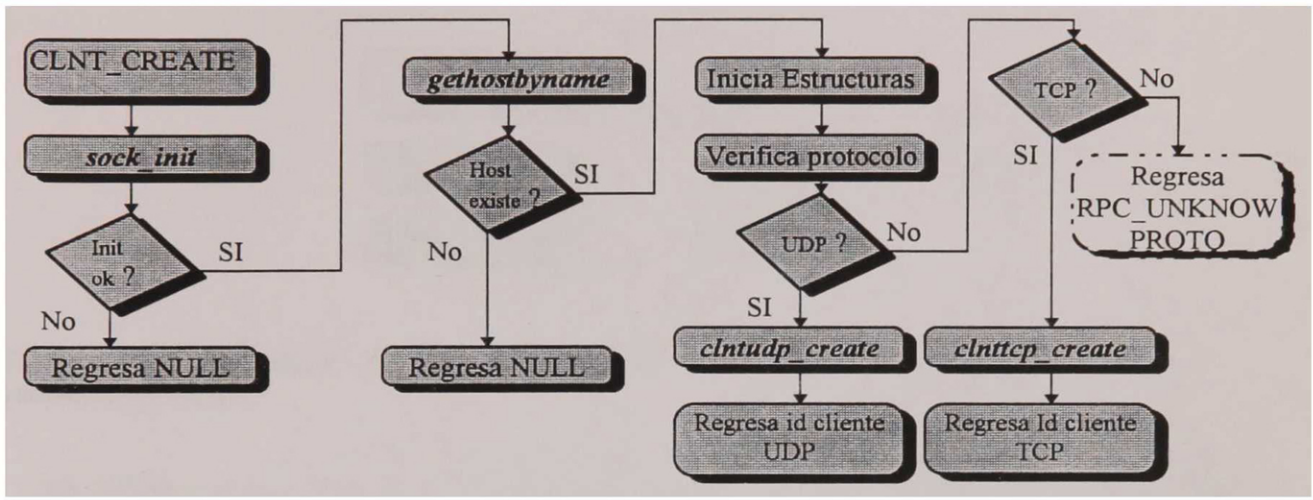


Figura 4-7 Diagrama del programa clnt_create encargado de contactar al servidor y generar un socket dependiendo del protocolo solicitado.

La rutina `clnt_create` mostrada en la figura 4.7, llama a `clntudp_create` o a `clnttcp_create` dependiendo del protocolo sobre el que se establece la conexión. En la figura 4.8 se presenta `clntudp_create()`. `clnttcp_create()` es similar a esta rutina.

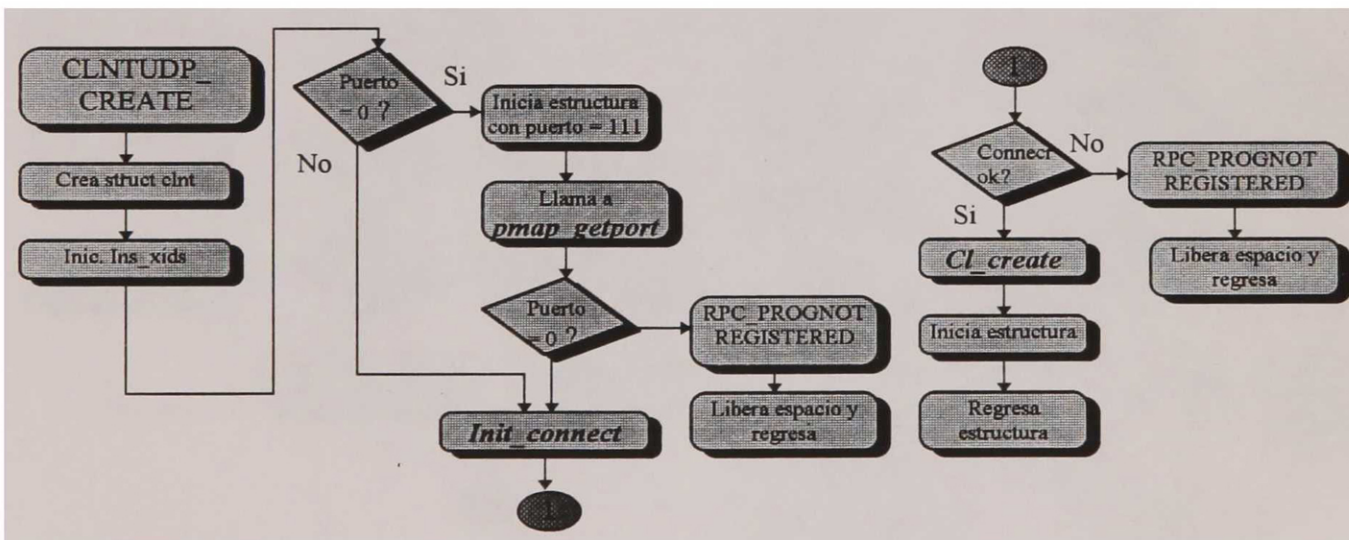


Figura 4-8 Diagrama de la rutina clntudp_create quien crea un socket UDP y obtiene el puerto donde debe establecer contacto.

La rutina `pmap_getport` mostrada en la figura 4.9, se encarga de llenar las estructuras necesarias para hacer la llamada remota al puerto del portmap. Para ello emplea la rutina `pmap_rmtcall`, que regresa RPC_SUCCESS si la llamada tuvo éxito y regresa el puerto a la rutina que lo solicitó.

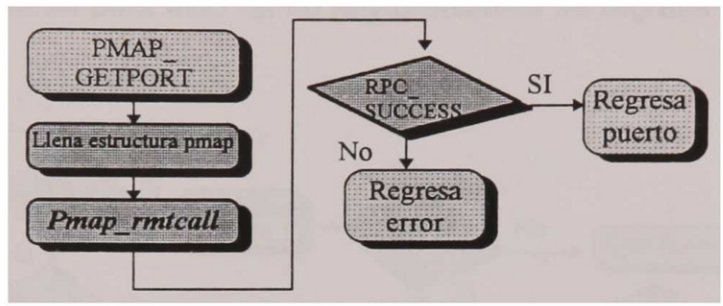


Figura 4-9 Diagrama de la rutina pmap_getport que efectúa una llamada remota para solicitar el puerto de un servicio

La rutina `pmap_rmtcall` mostrada en la figura 4.10, se encarga de llamar a la rutina que codifica el mensaje en el formato del protocolo RPC, anexando los parámetros y convirtiendo todo al formato XDR. Al mensaje se le agrega un encabezado dependiendo del protocolo de transporte. Se envía el mensaje y se espera a que llegue la respuesta. Cuando llega la respuesta, se desempaqueta, se decodifica y se verifica que sea la respuesta a la llamada. Se regresa RPC_SUCCESS para indicar que no hubo ningún error.

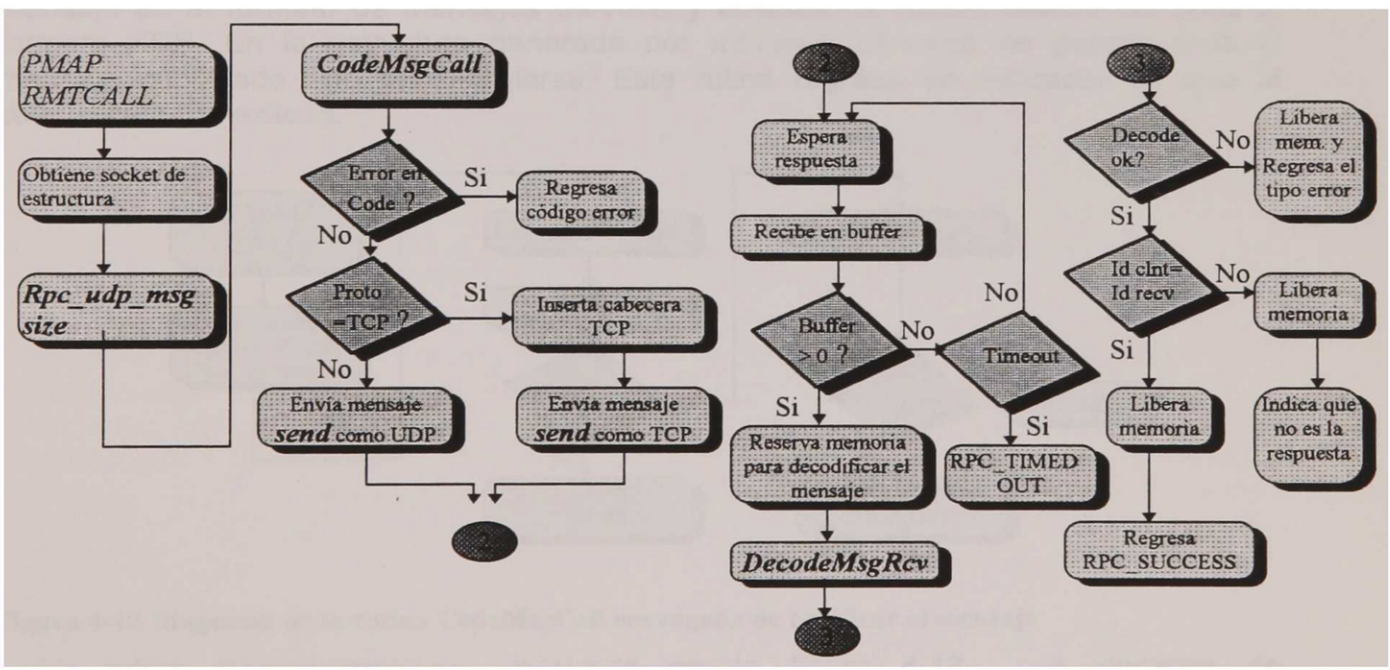


Figura 4-10 Diagrama de la rutina pmap_rmtcall encargada de hacer una llamada remota

La rutina `init_connect` mostrada en la figura 4.11, crea el socket para el protocolo especificado, utiliza la función `fcntl` para iniciar al socket no bloqueado y emplea `connect` para permitirle al cliente especificar una dirección remota para un socket previamente creado. Si el socket usa TCP, `connect` usa 3 paquetes para establecer una conexión, si el socket usa UDP, `connect` especifica el extremo remoto pero no transfiere ningún datagrama para esto. Si no hay problemas se regresa el descriptor del socket.

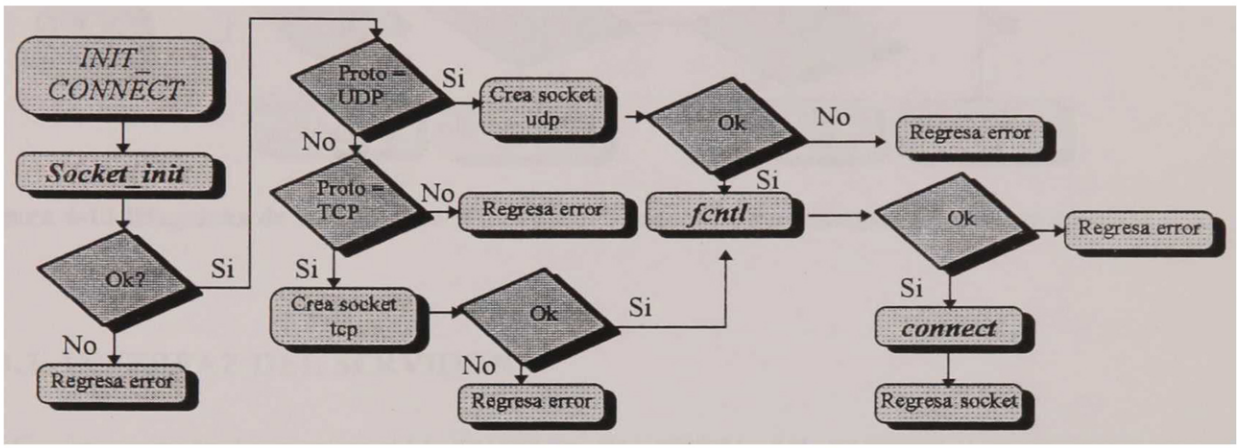


Figura 4-11 Diagrama de la rutina init_connect para establecer direcciones remotas al socket

La rutina `CodeMsgCall`, mostrada en la figura 4.12, se encarga de codificar el mensaje en el formato de mensajes de RPC y codificar al mismo tiempo los tipos al formato XDR. En la estructura generada por `xdrmem_create` se guarda todo el mensaje codificado listo para enviarse. Esta rutina regresa un indicador de que la codificación fue exitosa.

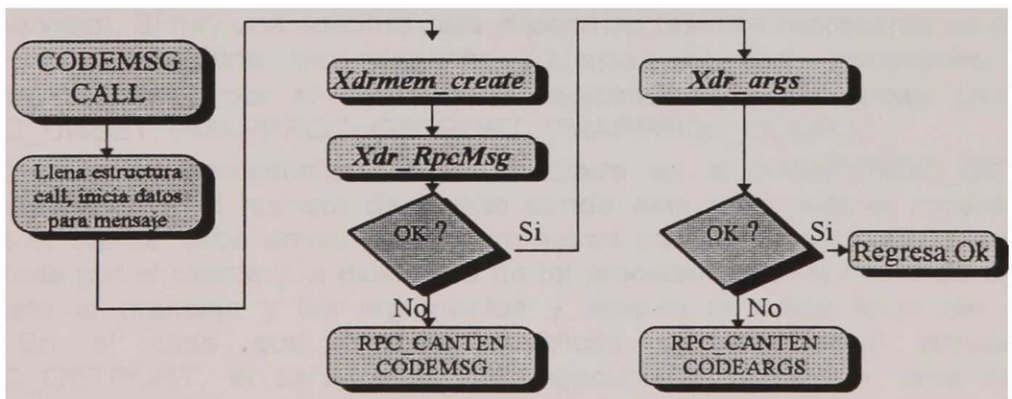


Figura 4-12 Diagrama de la rutina CodeMsgCall encargada de codificar el mensaje

La rutina `DecodeMsgRcv`, mostrada en la figura 4.13, se encarga de desempaquetar el mensaje recibido convirtiendo las rutinas del formato XDR al formato normal, así como desempaquetar el resultado. Si no se presenta ningún error, entonces se regresa RPC_SUCCESS.

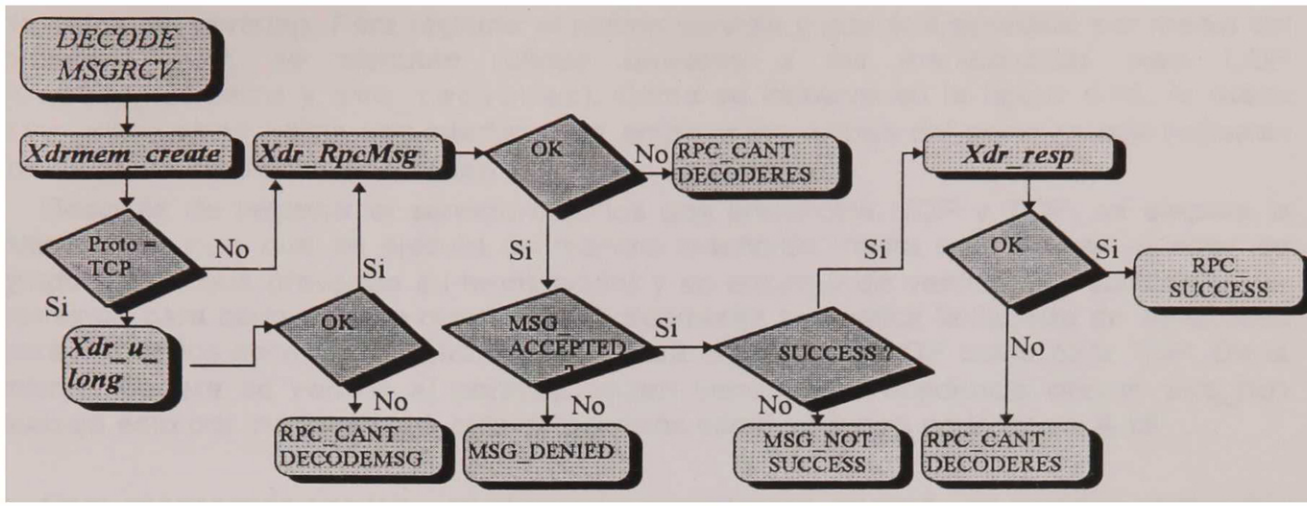


Figura 4-13 Diagrama de la rutina DecodeMsgRcv encargada de desempaquetar los datos

4.3.2 INTERFAZ DEL SERVIDOR

Es importante la explicación detallada del diseño del portmap y la relación con el servidor. En la figura 4.14 se ilustran los detalles.

En esta figura se representa de manera general los módulos que comprende el portmap y el servidor. Antes de iniciar el servidor se ejecuta el portmap como si fuera un proceso independiente. El portmap se enlaza al puerto 111, al mismo tiempo se registra en la tabla de puertos como cualquier otro servicio ofreciendo sus rutinas al cliente. El registro sólo se hace al iniciar el servidor. Existe una interfaz que se encarga de verificar las solicitudes en las colas de atención de cada uno de los sockets (es decir las colas para cada servicio). Si hay una solicitud para el portmap (esto se representa en el cuadro punteado) existe la rutina de despacho (`dispatch`) que comprueba que el procedimiento solicitado por el cliente esté registrado en el portmap (puede ser PMAPPROC_UNSET, PMAPPROC_GETPORT, PMAPPROC_DUMP).

Por lo general el procedimiento más solicitado es el PMAPPROC_GETPORT, encargado de regresar el numero de puerto donde está registrado el programa y la versión que el cliente debe enviar como parámetros para esa rutina. Para verificar la rutina solicitada por el cliente y la existencia de tal procedimiento, la rutina de despacho desempaqueta el mensaje y los argumentos y ejecuta la rutina local del portmap solicitada. En el caso que el cliente solicite se ejecute el procedimiento PMAPPROC_GETPORT, el servidor portmap ejecuta la rutina que lleva el mismo nombre: `pmapproc_getport`. Al finalizar la rutina, empaqueta el resultado y lo envía al cliente

Por el lado del servidor, existe una rutina que inicia la interfaz de sockets para el manejo básico de éstos. El siguiente módulo llamado `svcudp_create` crea un socket del protocolo UDP para que puedan acceder al servicio a través de dicho protocolo y lo enlaza a un puerto disponible. El módulo `svc_register` registra los datos del servicio, es decir registra el nombre del programa, la versión, el puerto y el protocolo en la base

de datos del portmap. Para registrar el mismo servicio y que sea accedido por medio del protocolo TCP, se ejecutan rutinas similares a las mencionadas para UDP (svctcp_create y svc_register). Como se observa en la figura 4.14, la rutina svc_register utiliza una interfaz para emplear las rutinas del portmap que registran los datos mencionados (o insertan en la base de datos).

Después de registrar el servicio bajo los dos protocolos UDP y TCP, se emplea la rutina svc_run que se ejecuta de manera indefinida (hasta que ocurra un error de grado mayor que provoque su terminación) y se encarga de verificar las solicitudes de conexión para cada servicio registrado. Cíclicamente se verifica la llegada de solicitudes para todos los servicios registrados tanto para el protocolo UDP como para TCP. De la misma manera se verifica el portmap, quien tiene una precedencia mayor. svc_run maneja esto por medio de una cola de servicios como se ilustra en la figura 4-15

Para anexar más servicios en el tipo de servidor que se creó , se pueden seguir dos procedimientos: Agregar más rutinas sobre el mismo identificador de programa o generar dos programas, cada uno con sus respectivas rutinas y se agrega a la cola de la figura 4.15 en el protocolo de transporte deseado.

A continuación se esquematizan algunas de las rutinas principales que maneja el servidor.

La rutina svcudp_create, mostrada en la figura 4.16, obtiene un identificador del transporte para el protocolo UDP, creando un socket UDP y enlazándolo a un puerto libre. Después de crear este socket, emplea la rutina xprt_create para llenar la estructura con algunos valores útiles que se emplean en otras rutinas .

La rutina svctcp_create, mostrada en la figura 4.17, es similar a svcudp_create, pero se emplea una instrucción adicional identificada como listen que hace que el socket este pasivo (es decir, listo para aceptar solicitudes entrantes). Listen también inicia el número de solicitudes de conexión que el software del protocolo puede tener en la cola asociada a un socket dado mientras el servidor maneja otra solicitud.

Una vez que se tiene el identificador del transporte, el siguiente paso es registrar el servicio por medio de la rutina svc_register, mostrada en la figura 4.18. Esta rutina asocia el número de programa, la versión y el protocolo con el procedimiento de despacho. Este procedimiento es invocado por la biblioteca RPC cuando se recibe una solicitud para ese número de programa y versión. Para controlar el manejo de las rutinas de despacho y los identificadores de transporte se ejecuta la rutina insert_dbxprt y para dar a conocer el servicio a través del portmap se utiliza la rutina pmap_set.

La rutina insert_bdxprt, mostrada en la figura 4.19, crea el nodo con la información a almacenar y lo inserta en una lista circular. Se maneja una serie de rutinas por cada servicio y una lista que maneja los sockets aceptados y rechazados para el control de la atención de múltiples llamadas al mismo servicio.

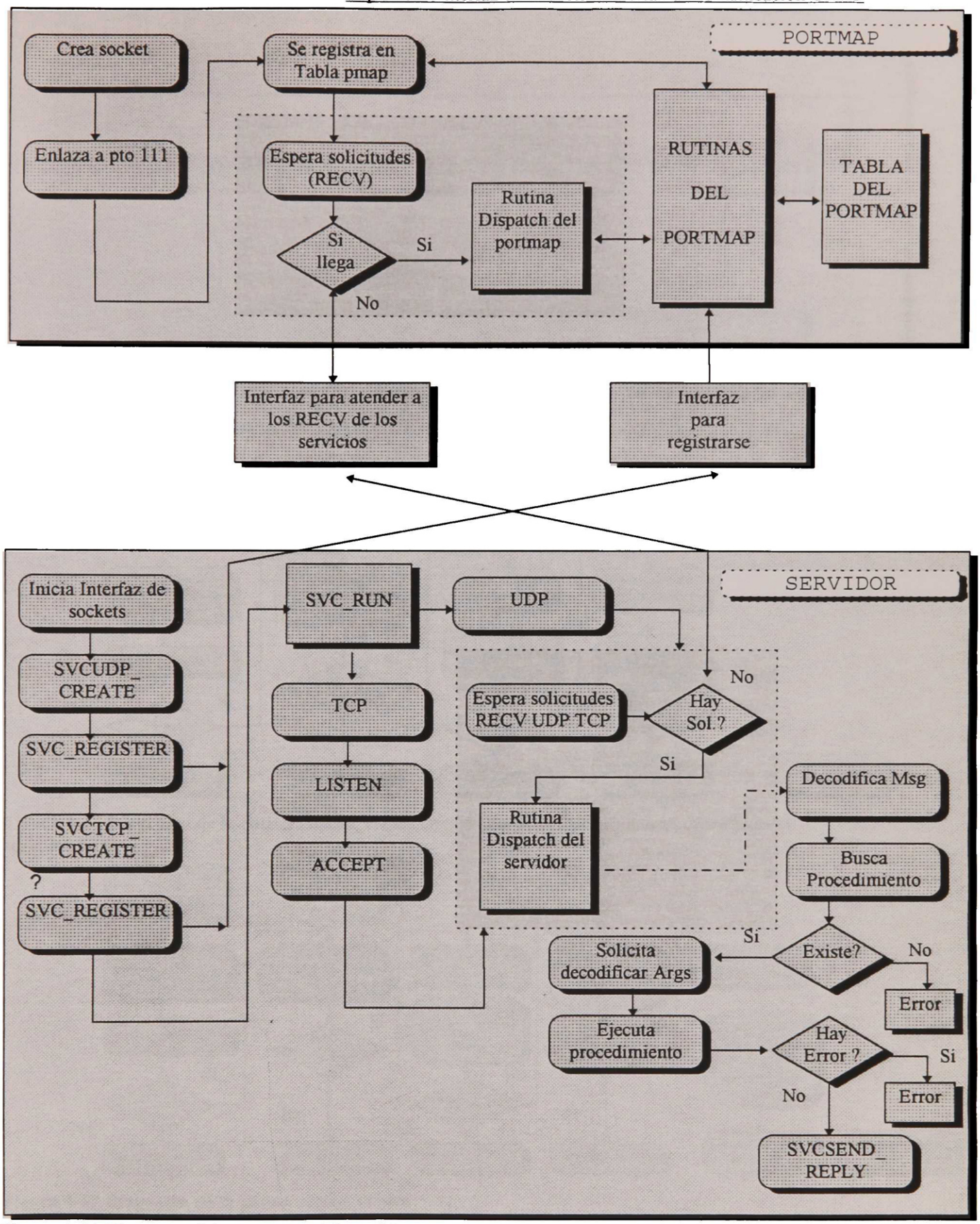


Figura 4-14 Diagrama del portmap y del servidor

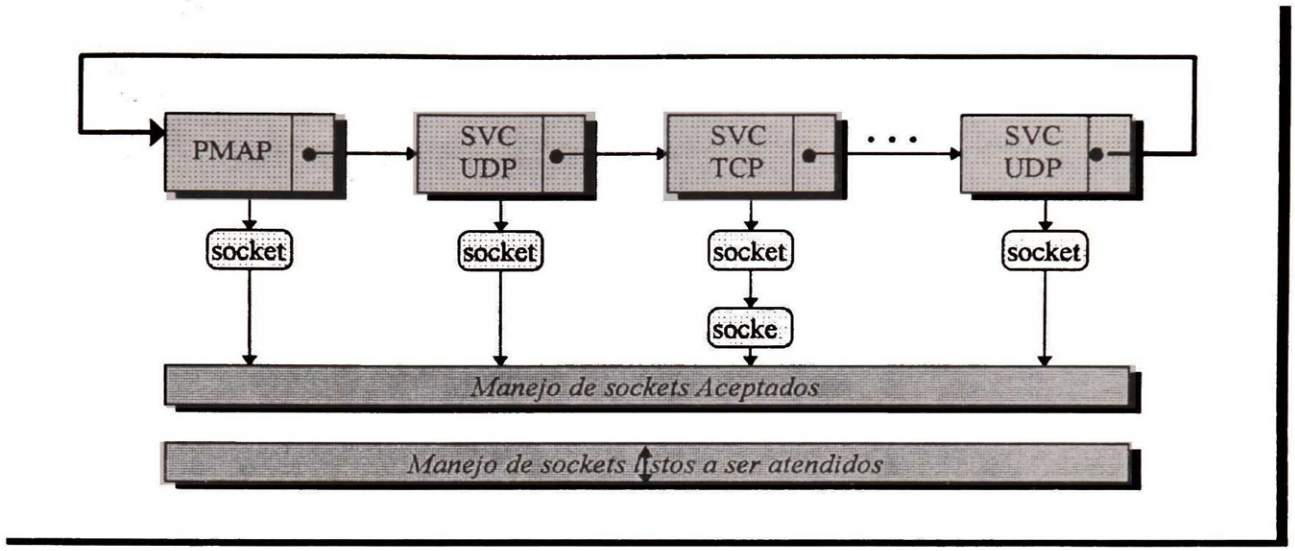


Figura 4-15 Diagrama del manejo las listas circulares para el control de los servicios

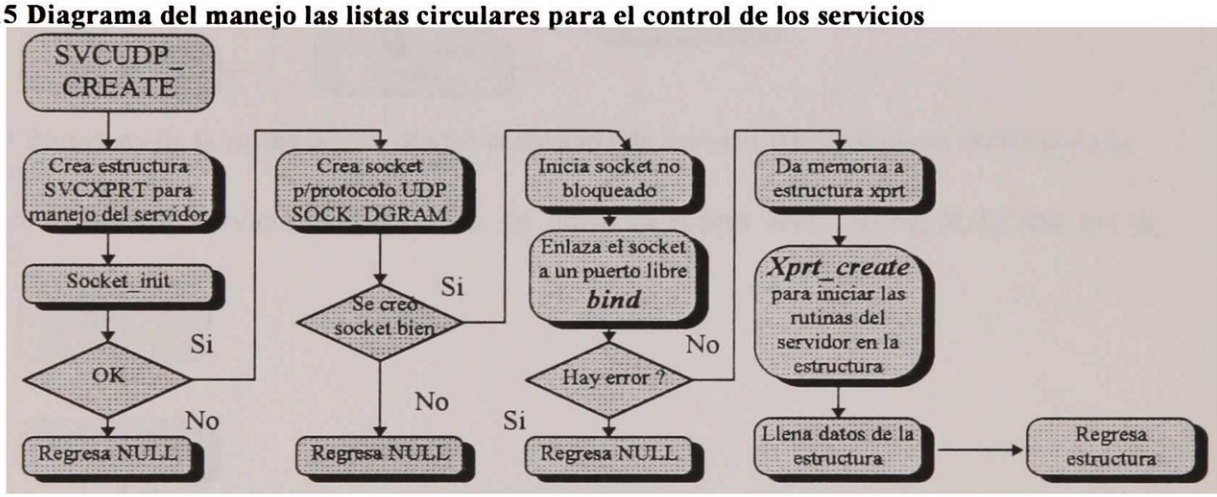


Figura 4-16 Diagrama de la rutina svcudp_create encargada de obtener una estructura para el servidor UDP

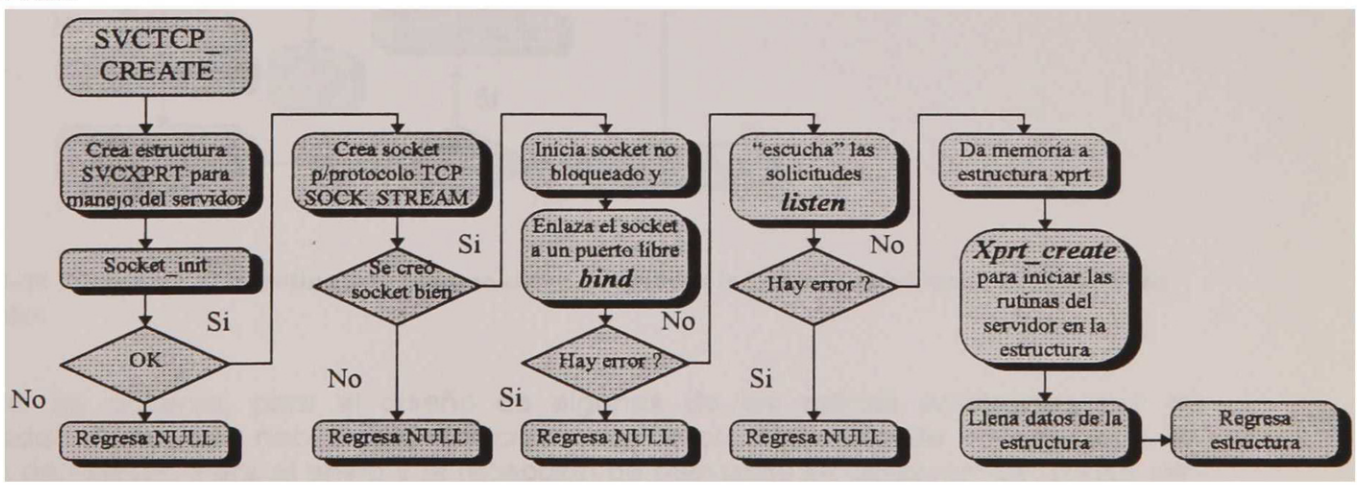


Figura 4-17 Diagrama de la rutina svctcp_create

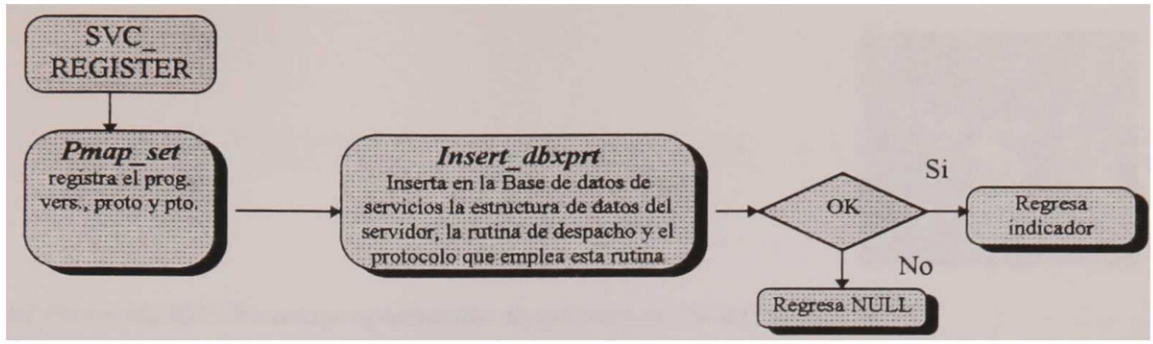


Figura 4-18 Diagrama de la rutina svc_register encargada de registrar el servicio a la base de datos del portmap e insertar a la base de datos de servicios

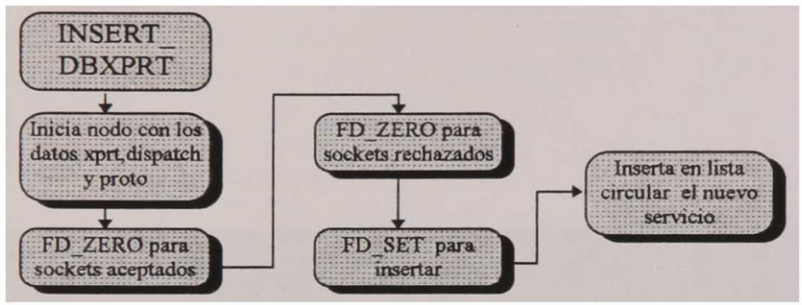


Figura 4-19 Diagrama de la rutina insert_dbxprt encargada de insertar las estructuras servidor en la lista circular

Para controlar los servicios registrados se tiene la rutina SVC_RUN, ilustrada en la figura 4.20.

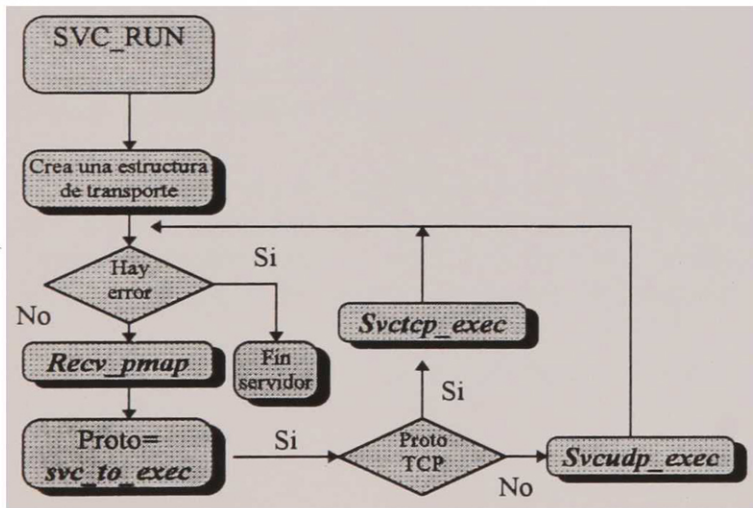


Figura 4-20 Diagrama de la rutina svc_run que corre de manera indefinida atendiendo a los servicios registrados

Como se observa, para el diseño de algunas de las rutinas empleadas por el compilador CRAD, fue necesario involucrase con la programación de sockets para el manejo de TCP/IP. Para el envío y la recepción de mensajes se utilizaron las rutinas xdr que se enviaron en el formato RPC. Ambos se explican con mayor detalle en el capítulo 3 y en los apéndice A y C.

CAPITULO

5

Compilador del Protocolo RPC Para la programación de aplicaciones distribuidas

Compilador CRAD

5

5. COMPILADOR CRAD

Para facilitar la programación de aplicaciones distribuidas se diseñó el programa CRAD (Compilador RPC para Aplicaciones Distribuidas) que se encarga de generar los filtros XDR y los stubs del cliente y del servidor quienes a su vez hacen llamadas a las rutinas de comunicación (interfaz) a través de la red. En este capítulo se describirá cómo se diseñó este compilador, los módulos que comprende, el software que se utilizó para la programación, las estructuras que se emplearon para los módulos del compilador, así como la explicación del código que genera.

5.1. Fases del Compilador

Un compilador, lee un programa escrito en un lenguaje (lenguaje fuente) y lo traduce a un programa equivalente en otro lenguaje (lenguaje objeto). Como parte de este proceso el compilador debe informar al usuario de posible errores en el programa fuente. Existe diversos lenguajes fuentes, así como lenguajes objeto. Un lenguaje objeto puede ser otro lenguaje de programación o un lenguaje de máquina.

El programa CRAD, lee programas fuentes en el lenguaje de programación RPC, que es una mezcla de la especificación XDR, y unas reglas adicionales para la definición de funciones remotas. Como programa objeto, CRAD genera programas en lenguaje C: uno para las funciones de cliente, otro para el servidor y si es necesario, también genera unos filtros para la conversión de tipos al formato XDR.

En los compiladores hay dos partes importantes [Ullman90]: la parte de **análisis** y la de **síntesis**. La primera se encarga de dividir el programa fuente en componentes y crear una representación intermedia del programa fuente. La segunda, se encarga de construir el programa objeto, a partir de la representación intermedia.

Un compilador opera en fases, cada una de las cuales transforma al programa fuente de una representación a otra. En la figura 5.1 se muestra una representación general de las fases de un compilador.

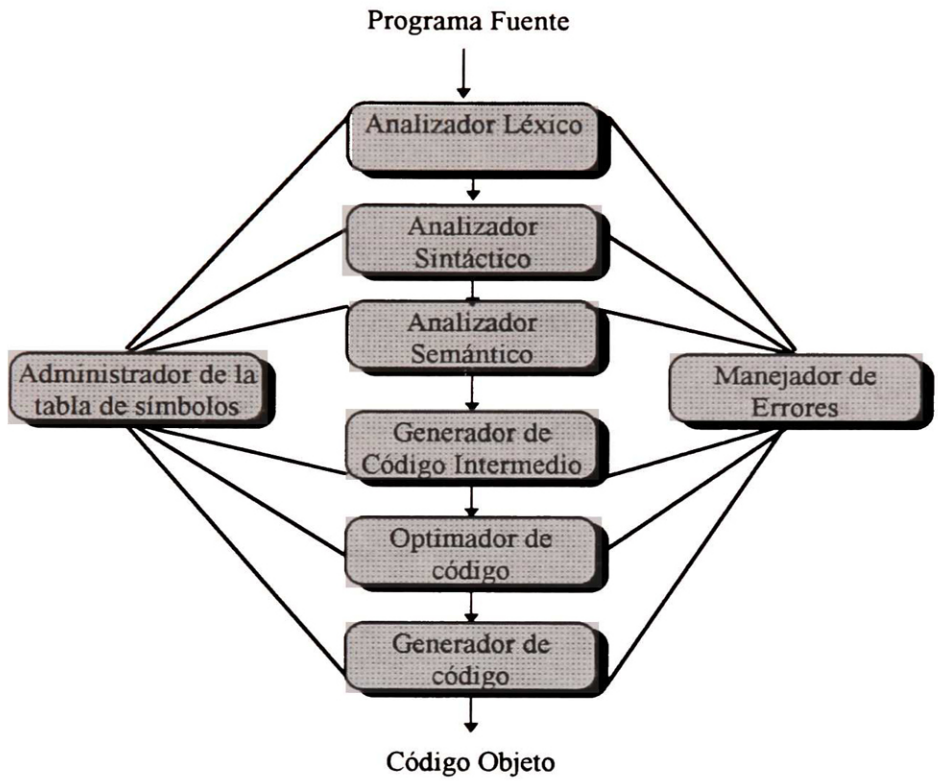


Figura 5-1 Fases de un Compilador

Durante el análisis lineal conocido también como **análisis léxico** o exploración, se analiza la cadena de caracteres del programa fuente (de izquierda a derecha) y se agrupa en componentes léxicos, que son una secuencia de caracteres que tienen un significado colectivo, eliminando los espacios en blanco, durante el análisis.

Durante el análisis Jerárquico o **análisis sintáctico**, los caracteres o los componentes léxicos se agrupan para cumplir una serie de reglas. Es decir, se agrupan en frases gramaticales que el compilador emplea para sintetizar la salida. Por lo general estas frases gramaticales se representan mediante un árbol de análisis sintáctico. La estructura jerárquica de un programa normalmente se expresa utilizando reglas "recursivas". Las gramáticas independientes del contexto son una formalización de reglas "recursivas" que se pueden usar para guiar el análisis sintáctico.

En el **Análisis Semántico,** es donde, después de verificar que se encuentran bien escritas las líneas del programa fuente de acuerdo a las reglas sintácticas, es necesario que se determine el sentido de esas líneas. Es decir, aunque el programa fuente esté bien escrito, es preciso que se analice si es correcto lo que se esta solicitando. Por ejemplo, esta fase debe verificar: tipos no declarados o duplicados, operadores y operandos permitidos, variables repetidas , falta de algún apuntador, etc.

De manera general el compilador registra todos los identificadores del programa fuente y los almacena junto con información especial en una **tabla de símbolos** (información relacionada con memoria asignada, su tipo, su ámbito, nombres de procedimientos, tipo de sus argumentos y método de pasar cada argumento, el tipo que

devuelve, etc.) . La tabla de Símbolos es una estructura de datos que contiene un registro por cada identificador, con los campos para los atributos de éste.

Cuando el programa de análisis léxico detecta un identificador en el programa fuente, se introduce en la tabla de símbolos; sin embargo los atributos de un identificador no se pueden determinar durante el análisis léxico.

El módulo para la **detección de errores** puede emplearse en cualquiera de las fases. Después de los análisis sintáctico y semántico, algunos compiladores generan una **representación intermedia** del programa fuente. Es una especie de lenguaje ensamblador que facilita la traducción al programa objeto. La fase "***optimización" del código***, intenta mejorar el código intermedio, de manera que resulte un código de máquina más rápido de ejecutar. Para finalizar, **se genera el código** objeto o código de máquina "relocalizable" o código ensamblador.

Las módulos de CRAD para implementar las fases descritas anteriormente son:

- Programa de análisis léxico
- Programa de análisis Sintáctico
- Programa de análisis Semántico
- Generador de errores
- Generador de la Tabla de símbolos
- Generación de Código Objeto:
    - Generador de encabezados
    - Generador del Stub del cliente
    - Generador del Stub del Servidor
    - Generador de Filtros XDR

En la figura 5.2 se muestra la arquitectura de CRAD.

En la actualidad, existen herramientas de desarrollo de software que permiten diseñar compiladores. Algunas más específicas, permiten la elaboración de ciertas fases. Estas herramientas se identifican como compiladores de compiladores o generadores de compiladores. Los programas de análisis léxicos para todos los lenguajes, esencialmente son iguales, excepto que cambian algunas palabras clave y signos particulares. Es decir, existen herramientas que generan *programas de análisis léxicos* a partir de una especificación basada en expresiones regulares [Ullman90]. Lo que producen estos programas, es un *autómata finito.* También existen los *generadores de análisis sintácticos*, cuya entrada es una *gramática independiente del contexto* entre otros.

En el diseño de CRAD, se aprovecharon unas herramientas de software que generan programas para análisis léxicos y sintácticos. Estas herramientas son Lex y Yacc [LEX] [YACC]. Lex es un lenguaje de patrón-acción y los patrones se especifican por medio de expresiones regulares. Un compilador de LEX puede generar un "reconocedor" de expresiones regulares mediante un autómata finito eficiente. Yacc que significa "otro compilador de compiladores más" (Yet Another Compiler-Compiler), genera un programa de análisis sintáctico.

Arquitectura de CRAD

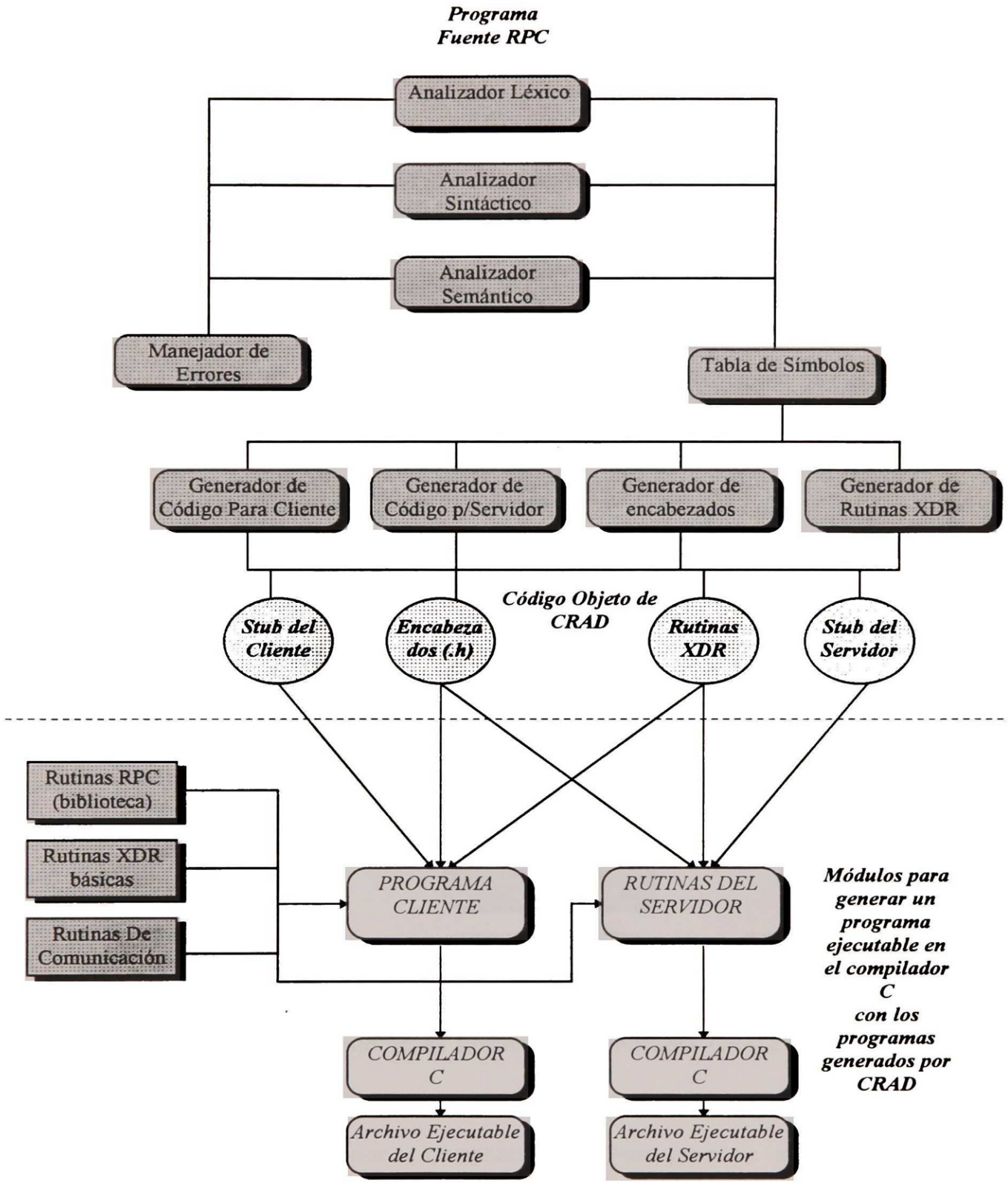


Figura 5-2 Arquitectura del Compilador CRAD

Para el empleo de éstas herramientas, fue necesario la investigación detallada del funcionamiento los compiladores LEX y YACC, el reconocimiento del software que generan para la manipulación, en algunos casos, de éste código.

5.2. *Análisis Léxico*

La función del programa de análisis léxico, consiste en leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el análisis sintáctico para hacer su evaluación. Esto se esquematiza en la figura 5.3:

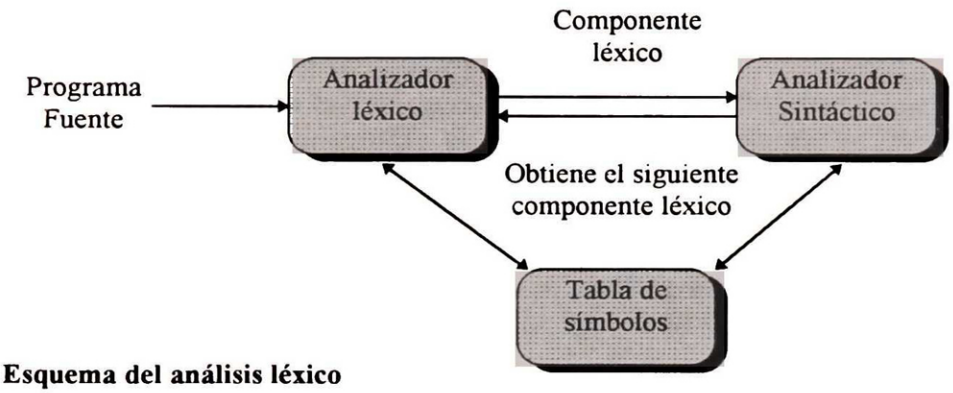


Figura 5-3 Esquema del análisis léxico

Para la generación del programa de análisis léxico se usó el compilador LEX. El procedimiento se ilustra en la figura 5.4:

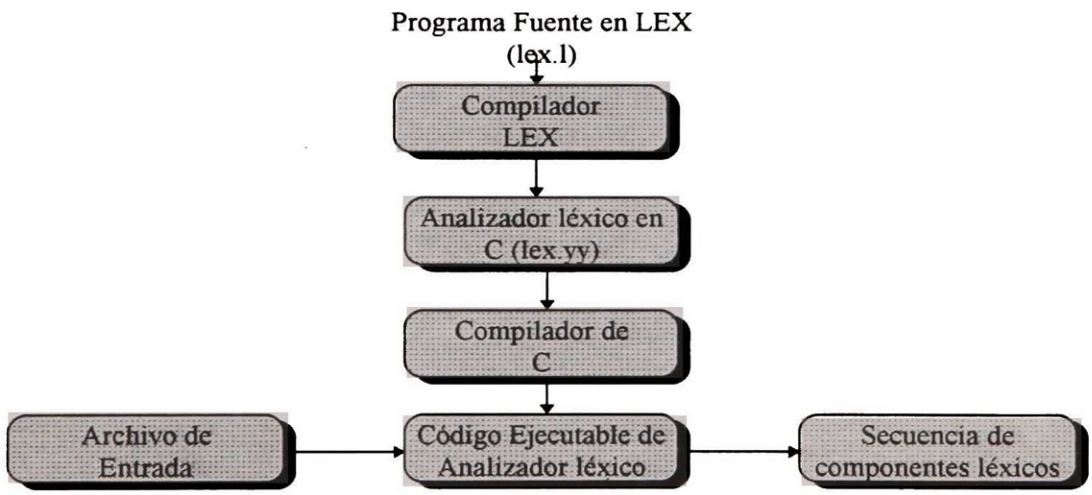


Figura 5-4 Secuencia de pasos para generar un programa que ejecute un análisis léxico

Como se observa, LEX tiene funciones similares al compilador CRAD. En LEX por lo general:

- Se prepara una especificación para el programa que hace el análisis léxico en el lenguaje LEX.
- Esta especificación se pasa por el compilador LEX para producir un programa en lenguaje C. Este programa es una representación tabular de un diagrama de transiciones que se construye a partir de las expresiones regulares de lex.l junto con una rutina estándar que utiliza la tabla para reconocer lexemas.

- El código en C puede compilarse y producir un código objeto , que es el programa encargado de hacer el análisis léxico.

Un programa en LEX tiene tres partes: declaraciones, reglas de traducción y procedimientos auxiliares.

1 **Declaraciones:** En esta sección se declaran variables, constantes y definiciones regulares que se emplean como componentes de las expresiones regulares que aparecen en las reglas de traducción (2).

2 **Reglas de traducción:** Estas reglas son proposiciones de la forma:

```
p1          {acción 1}
p2          {acción 2}
...
pn          {acción n}
```

donde pi es una expresión regular y cada acción es un fragmento de programa que describe la acción que el programa de análisis tomará cuando pi concuerde con un "lexema". Por lo general en LEX las acciones se escriben en C.

3 **Procedimientos auxiliares** En esta sección se escriben los procedimientos auxiliares que requieran las acciones (reglas de traducción)

LEX se coordina con el programa que hace el análisis sintáctico, ya que al terminar de reconocer una expresión regular, ejecuta la acción, que generalmente da el control al análisis sintáctico.

En el listado 5.1 se muestra la especificación en LEX que se diseñó para el programa de análisis léxico del compilador CRAD.

```
%{
/*_____________________________________
LEXRPC.L -- Análisis léxico de RPC
_____________________________________*/
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include "rpcsint.h"
#include "errorlib.h"

extern YYSTYPE yylval;
char id_numero[20];
%}
/*_____________________________________
        REGLA
_____________________________________*/
delim [ \t]
nl    [\n]
eb    {delim}+
letra [A-Za-z]
digito [0-9]
alfanum [A-Za-z_0-9]
id        {letra}{alfanum}*
numero {digito}+
hex       [0][x|X]({digito}|[a-fA-F])*
otro      .
```

```
%x      COMMENT
%%

/*_____________________________________
        GRAMATICA LEXICA
_____________________________________*/
{eb}            {/* no hay accion */}
{nl}            {++yylineno;}
enum            {return(ENUM);}
typedef         {return(TYPEDEF);}
const           {return(CONST);}
struct          {return(STRUCT);}
union           {return(UNION);}
case            {return(CASE);}
switch          {return(SWITCH);}
default         {return(DEFAULT);}
bool            {return(BOOL);}
string          {return(STRING);}
opaque          {return(OPAQUE);}
void            {return(VOID);}
program         {return(PROGRAM);}
version         {return(VERSION);}
int             {return(INT);}
hyper           {return(HYPER);}
unsigned        {return(UNSIGNED);}
float           {return(FLOAT);}
double          {return(DOUBLE);}

{hex}           {strcpy(yylval.buf,yytext);
                 return(HEX);
                 }
{id}            {strcpy(yylval.buf,yytext);
                 return(ID);
                }
{numero}        { strcpy(yylval.buf,yytext);
                  return(NUMERO);
                }

{digito}+\.{digito}*((e|E)("+"|"-"){digito}+)?      {return(FLOAT);}
\.{digito}+((e|E)("+"|"-"){digito}+)?               {return(FLOAT);}

"/*"            {BEGIN(COMMENT);}
<COMMENT>"*/"       BEGIN(0);
<COMMENT>[^*\n]+    ;
<COMMENT>"\n"   {++yylineno;}
<COMMENT>"*"        ;

{otro}          return yytext[0];
%%
```

Listado 5.1 Especificación en LEX del analizador léxico de CRAD.

En la sección de declaraciones , aparece la definición de ciertas constantes, utilizadas por las reglas de traducción. Las declaraciones se encierran entre llaves especiales %{ y %}. Todo lo que aparece entre estas llaves se copia directamente al programa de análisis léxico y no se consideran parte de las definiciones regulares. Después siguen las definiciones regulares y las reglas de traducción que están separadas por el símbolo %%.

5.3. Análisis Sintáctico

El generador de programas para análisis sintácticos YACC, emplea la técnica denominada LALR (LookAhead-LR, análisis sintáctico LR con símbolos de anticipación) [Ullman90]. Su funcionamiento se ilustra en la figura 5.5.

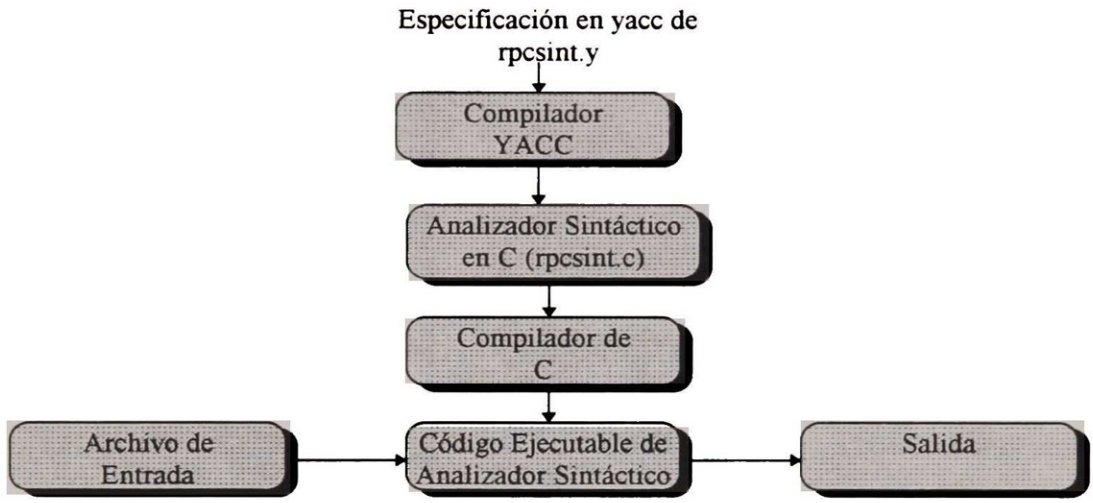


Figura 5-5 Pasos para generar un programa de análisis sintáctico

Un programa fuente en YACC tiene tres partes: declaraciones, reglas de traducción y rutinas de apoyo en lenguaje C. Estas partes están separadas por el símbolo %% como se muestra a continuación:

declaraciones	**%{** *Inicia declaraciones Ordinarias* Declaraciones Ordenarias **%}** *Termina declaraciones ordinarias* *% componentes léxicos de las gramaticales*
%%	*Indica separación de sección*
reglas de traducción	*cada regla consta de una producción de la gramática y la acción semántica asociada (secuencia de proposiciones en C). Ver el listado 5.2 para mayor detalle.*
%%	*Indica separación de sección*
rutinas en C de apoyo	*En ésta sección se ubican las rutinas en el Lenguaje C que realizan funciones para el análisis sintáctico,*

semántico y la generación de las tablas de símbolos y los mensajes de error.

En el listado 5.2 se presenta la especificación del programa que realiza el análisis sintáctico del compilador CRAD. Sólo se especifican algunas de las rutinas de apoyo del lenguaje C elaboradas para éste analizador.

```
/*_____________________________________
        Especificación del Analizador
Sintáctico de CRAD
_____________________________________*/
        /*Parte de declaraciones :
   En esta parte hay dos secciones opcionales . En
la primera son declaraciones ordinarias en C,
delimitadas por %{ y %}. Aqui se colocan las
declaraciones de todas las temporales usadas por las
reglas de traducción o los procedimientos de la
segundas y terceras secciones. También se declaran
los componentes léxicos de las gramáticas. */
%{
#include<stdio.h>
#include<string.h>
#include <conio.h>
#include <stdlib.h>
#include "errorlib.h"
#include "rpcgral.h"
#include "strseman.h"
#include "rpcglob.h"
%}
%union{
     int i;
     long l;
     float r;
     char buf[80];
 }
%{extern indicador;%}
/* Las siguiente declaraciones son los componentes
léxicos de las gramáticas. */
/* PALABRAS RESERVADAS */
%token          PROGRAM      VERSION
                PROCEDURE    ID
        TYPEDEF
%token           ENUM        STRUCT
                 UNION       CONST
```

```
%token          SWITCH       CASE
                DEFAULT
/* TIPOS DEFINIDOS */
%token          OPAQUE       STRING
                BOOL         VOID   INT
%token          UNSIGNEDINT  HYPER
                UNSIGNEDHYPER
                FLOAT
%token          DOUBLE       UNSIGNED
/* CONSTANTES Y OTROS */
%token          NUMERO       CONSTANT
                FLOAT        HEX
%token          COMENTA      FIND
                NFIND  DEFINICION
%token  LISTA   DECLARACION
        NUEVA_LISTA
%token  FIJO    VARIABLE
        NODEFINIDO   APUNTADOR
%token  ESTRUCTURA   TIPONODEF
        SIMPLEENDTLISTA
%token  STRUCT_LISTA
        PROG2DEF
/* TOKENS DECLARADOS PARA EL
ANALISIS SEMANTICO Y LA GENERACION
DE COD. */
%token  STRUCTSIMPLE         STRUCTFIJO
        STRUCTVARIABLE
%token  STRUCTNODEF
        STRUCTAPUNT
        ENUMSIMPLE
%token  ENUMFIJO
        ENUMVARIABLE
        ENUMNODEF
%token  ENUMAPUNT
        UNIONSIMPLE          UNIONFIJO
%token  UNIONVARIABLE
        UNIONNODEF   UNIONAPUNT
%token  CORCHA       CORCHC
```

```
%start          lista_def
/*Nos indica con qué regla de producción iniciará el analizador*/
%%
                /*REGLAS DE TRADUCCION (GRAMATICA)
```

En esta parte después del primer par %% se colocan las reglas de traducción. Cada regla consta de una producción de la gramática y la acción semántica asociada. Un conjunto de producciones en YACC se escriben como:

```
<lado izquierdo>  :   <alt 1>    { acción semántica 1 }
                  |   <alt 2>    { acción semántica 2 }
                  .   .          .
                  |   <alt n>    { acción semántica n }
```

En una producción en YACC un caracter simple entrecomillado 'c' se considera como el símbolo terminal c, y las cadenas sin comillas de letras y dígitos no declarados como componentes léxicos se consideran no terminales. Los lados derechos alternativos se pueden separar con una barra vertical, y un símbolo de punto y coma sigue a cada lado izquierdo con sus alternativas y sus acciones semánticas. El primer lado izquierdo se considera como el símbolo inicial.

Una acción semántica en YACC es una secuencia de proposiciones en C. En una acción semántica, el símbolo $$ se refiere al valor del atributo asociado con el no terminal del lado izquierdo, mientras que $i se refiere al valor asociado con el i-ésimo símbolo gramatical (terminal o no terminal) del lado derecho. */

```
lista_def:      definicion ';'
                |       lista_def definicion ';'
                |       error
                ;
definicion      :       type_def
                |       constant_def
                |       program_def
                |       error
                ;
type_def        :TYPEDEF declaration                    {Inserta_Numero_Linea();}
                |ENUM  enum_ident                       enum_body
        {Indica_Fin_Estructura();}
                |STRUCT struct_ident    struct_body             {Indica_Fin_Estructura();}
                |       UNION  union_ident   union_body          {Indica_Fin_Estructura();}
                ;
enum_ident      :       ID
                ;
enum_body       :       '{' enum_value_list '}'
                ;
enum_value_list :       enum_value
                |       enum_value_list ',' enum_value
                ;
enum_value      :       variable_ident '=' value
                ;
struct_ident    :       ID
                ;
struct_body     :       '{' declaration_list '}'
                ;
declaration_list:       declaration ';'
                |       declaration_list declaration ';'
                ;
union_ident     :       ID
                ;
union_body      :       SWITCH '(' simple_dec ')' '{' case_list '}'
                |       SWITCH '(' simple_dec ')' '{' case_list default_def '}'
                |       SWITCH error
                ;
case_list:      :       CASE val_case ':' case_dec ';'
```

```
                |   case_list CASE val_case ':' case_dec ';'
                |   CASE error
                ;
case_dec        :   declaration
                |   VOID
                ;
val_case        :   ID
                |   NUMERO
                ;
default_def     :   DEFAULT ':' case_dec ';'
                |   DEFAULT error
                ;
constant_def    :   CONST const_ident '=' NUMERO
                |   CONST error
                ;
const_ident     :   ID
                ;
value           :   NUMERO
                |   ID
                ;
declaration     :   declaration_def
                |   declaration_other
                ;
declaration_other:  other_type variable_ident '[' value ']'        {Inserta_Informacion(FIJO);}
                |   other_type variable_ident '<' value '>'    {Inserta_Informacion(VARIABLE);}
                |   other_type variable_ident '<' '>'          {Inserta_Informacion(NODEFINIDO);}
                ;
other_type      :   OPAQUE {strcpy(tipo,"opaque");}
                |   STRING {strcpy(tipo,"string");}
                ;
declaration_def :   simple_dec
                |   fixed_array_dec
                |   variable_array_dec
                |   pointer_dec
                ;
simple_dec      :   type_ident variable_ident                   {Inserta_Informacion(SIMPLE);}
                ;
fixed_array_dec :   type_ident variable_ident '[' value ']'     {Inserta_Informacion(FIJO);}
                ;
variable_array_dec: type_ident variable_ident '<' value '>' {Inserta_Informacion(VARIABLE);}
                |   type_ident variable_ident '<' '>'    {Inserta_Informacion(NODEFINIDO);}
                ;
pointer_dec     :   type_ident pointer  variable_ident
                ;
pointer         :   '*'
                ;
program_def     :   PROGRAM program_ident '{' version_list '}' '=' value_program
                |   PROGRAM error
                ;
program_ident   :   ID      {ap_programa=Inserta_Nombre_Programa(yylval.buf);}
                ;
version_list    :   version ';'
                |   version_list  version ';'
```

```
                ;
version         :       VERSION version_ident '{' procedure_list '}' '=' NUMERO
                |       VERSION error
                ;
version_ident   :       ID
                ;
procedure_list  :       procedure ';'
                |       procedure_list procedure ';'
                |       procedure error
                |       procedure_list procedure error
                ;
procedure       :       type_proced procedure_ident '(' type_proced ')' '=' NUMERO
                |       type_ident error
                ;
procedure_ident :       ID
                ;
type_proced     :       type_ident
                |       other_type
                |       VOID            {strcpy(tipo,"void");}
                ;
type_ident      :       INT                     {strcpy(tipo,"int");}
                |       UNSIGNED INT            {strcpy(tipo,"unsigned int");}
                |       HYPER                   {strcpy(tipo,"hyper");}
                |       UNSIGNED HYPER          {strcpy(tipo,"unsigned hyper");}
                |       FLOAT                   {strcpy(tipo,"float");}
                |       DOUBLE                  {strcpy(tipo,"double");}
                |       BOOL                    {strcpy(tipo,"bool");}
                |       typedef_name
                ;
typedef_name    :       ID      {strcpy(tipo,yylval.buf);}
                ;
variable_ident  :       ID      {Inserta_Identificador(yylval.buf);}
                ;
value_program   :       HEX     {strcpy(ap_programa->numero_programa,yylval.buf);}
                |       NUMERO  {strcpy(ap_programa->numero_programa,yylval.buf);}
                ;
%%
```

/*RUTINAS EN C DE APOYO

Esta es la tercera parte de una especificación. Aqui se puede declarar el analizador léxico (yylex()) y los procedimientos de recuperación de errores, así como todas las rutinas necesarias para el análisis semántico. Se presentan sólo unas de las rutinas del compilador CRAD. */

```
/*-------------------------------------------------------------------------------------------------------------
        BUSCA_INSERTA:
-------------------------------------------------------------------------------------------------------------*/
Tab_Tpos * Busca_Inserta(char *nombre){
        Tab_Tpos *ap_tabla_tipos;

        if((ap_tabla_tipos=Busca_Tipo(nombre))!=NULL){
                if(ap_tabla_tipos->info ==TIPONODEF){
                ap_tabla_tipos->info=ESTRUCTURA;
                        return(ap_tabla_tipos);}
                else {
                        Tope=Inserta_Error(Tope,yylineno,nombre,T2DEC,'\0');
```

```
                        /*Tipo doblemente declarado"*/
                        return(NULL);}
        }
        else{
                ap_tabla_tipos=Inserta_Tipo(nombre);
                return(ap_tabla_tipos);
        }
}
/*-----------------------------------------------------------------------------------------------------------
        ANALISIS_SINTACTICO
-----------------------------------------------------------------------------------------------------------*/

void Análisis_Sintáctico(){
        yyparse();
}
```

Listado 5.2 Especificación en YACC para el análisis sintáctico.

5.4. Análisis Semántico , Tablas de símbolos y Manejo de Errores

Parte del análisis semántico del compilador se genera durante el análisis sintáctico. Si existen errores sintácticos, el programa que efectúa el análisis semántico no se activa en su totalidad. Solo almacena información relevante para posteriormente analizar la semántica y si detecta algún error semántico durante la verificación de la sintaxis, lo almacena en una estructura especial de errores. Al mismo tiempo que se ejecuta el análisis sintáctico, se genera la tabla de símbolos, que se emplea para los siguientes análisis y para la generación del código. Cuando no existen errores sintácticos, el compilador hace una verificación sobre los datos recopilados en la tabla de símbolos y determina si existen errores semánticos adicionales a los que se detectan durante el análisis sintácticos.

La manera en que funciona el manejo de errores, en CRAD, es la forma de recuperación de errores de YACC, usando producciones de error, junto con las rutinas de manejo de errores para el análisis semántico. Se determinaron los no terminales que tienen recuperación de error y se anexó una gramática de error a las producciones (con la palabra clave **error**). Se maneja una pila de símbolos que al detectar un error se vacía hasta encontrar uno con el que pueda proseguir el análisis sintáctico normal.

De este compilador se generan las siguientes estructuras:

- Una tabla para los tipos normales con apuntadores a listas de tipos cuando se trata de estructuras.
- Una tabla para las Constantes
- Listas para almacenar el nombre del programa, los procedimientos y las versiones que se ejecutarán de manera remota.

La estructura de las tablas y listas se representan en la figura 5.6

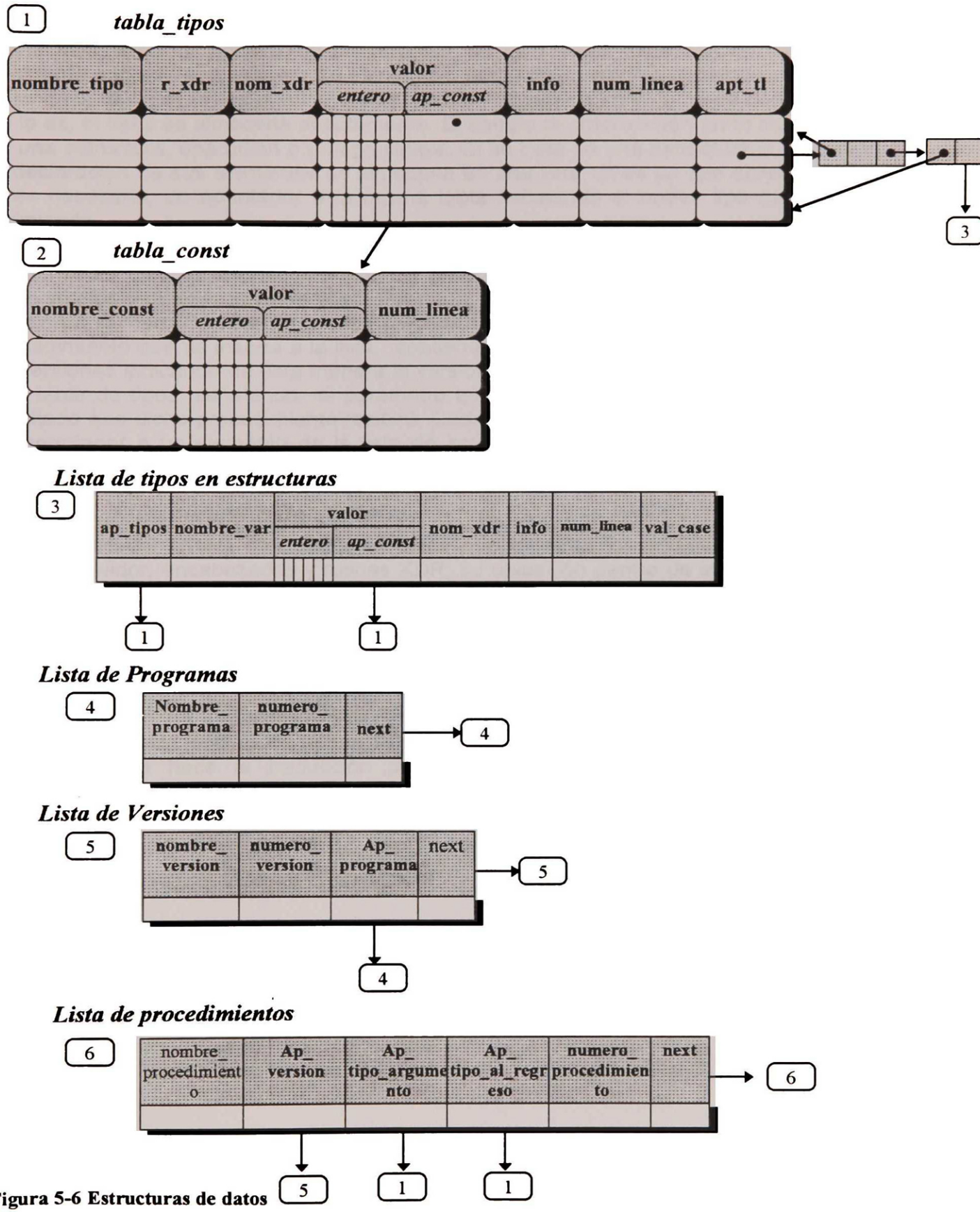


Figura 5-6 Estructuras de datos

Los tipos se almacenan en el campo *Nombre_tipo*. Cada uno tiene asociado una rutina XDR. Algunos tipos como los arreglos, en su sintaxis necesitan un valor o una constante que defina el número de elementos, para ello se emplea el campo *valor.* En el caso de la declaración de una constante, ésta se almacena en la tabla de constantes, si no lo es, el valor se almacena directamente. El campo de información (*info*) indica si tipo es una estructura, una unión o un tipo simple. Si se trata de una estructura o una unión, la declaración de sus elementos se almacena en una lista. Si es un tipo simple emplea, si es necesario, un apuntador a la misma tabla señalando el nuevo tipo que se está definiendo.

Las listas para la identificación de los programas, versiones y procedimientos son listas ligadas y enlazadas entre ellas. Por cada número de programa que se genera, se asocia un número de versión con los procedimientos respectivos. De igual forma cada procedimiento que se inserta a la lista, contiene un apuntador a un elemento en la lista de versiones indicando de ésta manera la versión que le corresponde y dos apuntadores a la tabla de tipos, señalando el parámetro que el procedimiento enviará así como el resultado que dicho procedimiento recibirá. Cada elemento de la lista de versiones tiene un apuntador a un elemento de la lista de programas, indicando el programa al cual pertenece dicha versión.

5.5. Generación de código

El compilador CRAD contiene 4 módulos que generan los códigos objeto para el cliente, servidor, encabezados y rutinas XDR; su ubicación dentro de la arquitectura del compilador se muestra en la figura 5.2.

5.5.1. Generador de encabezados

En la figura 5.7 se muestra el algoritmo que genera los encabezados del compilador diseñado. En este código se hacen llamadas al generador de rutinas XDR al mismo tiempo que se hace la traducción para la declaración de tipos del archivo de encabezado.

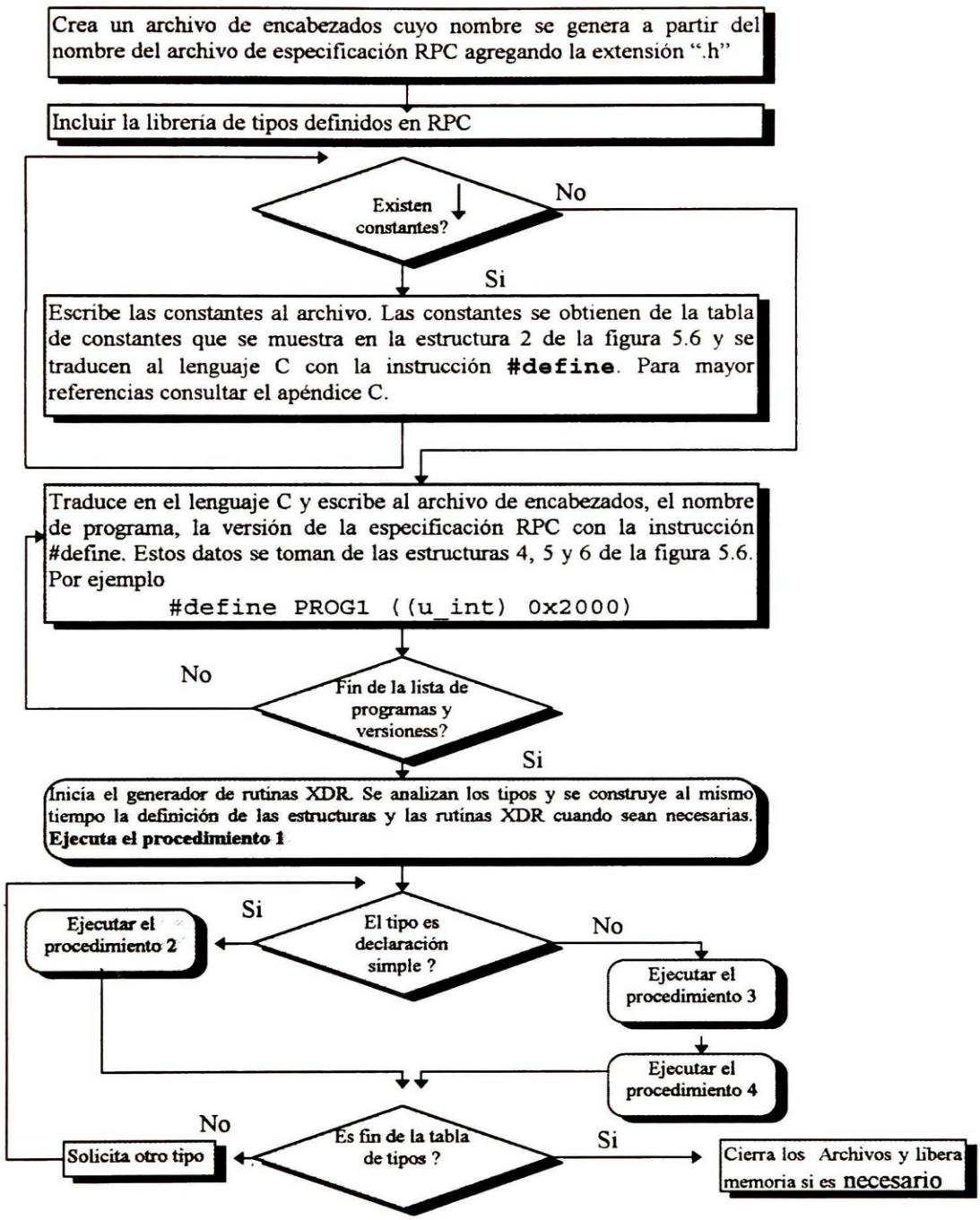


Figura 5-7. Algoritmo del generador de encabezados.

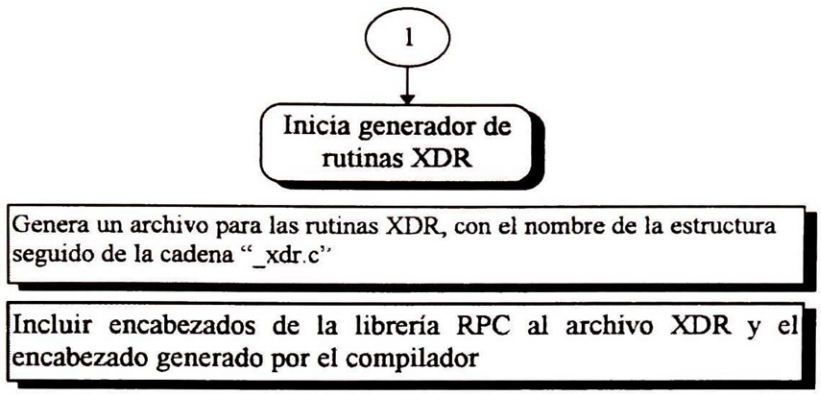


Figura 5-8 Procedimiento 1

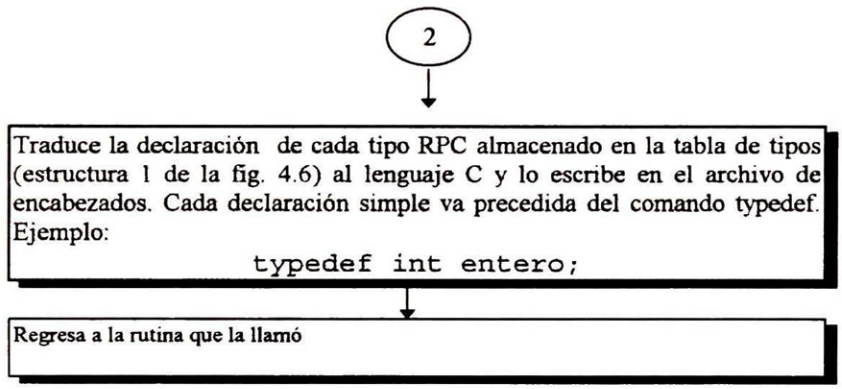


Figura 5-9. Procedimiento 2

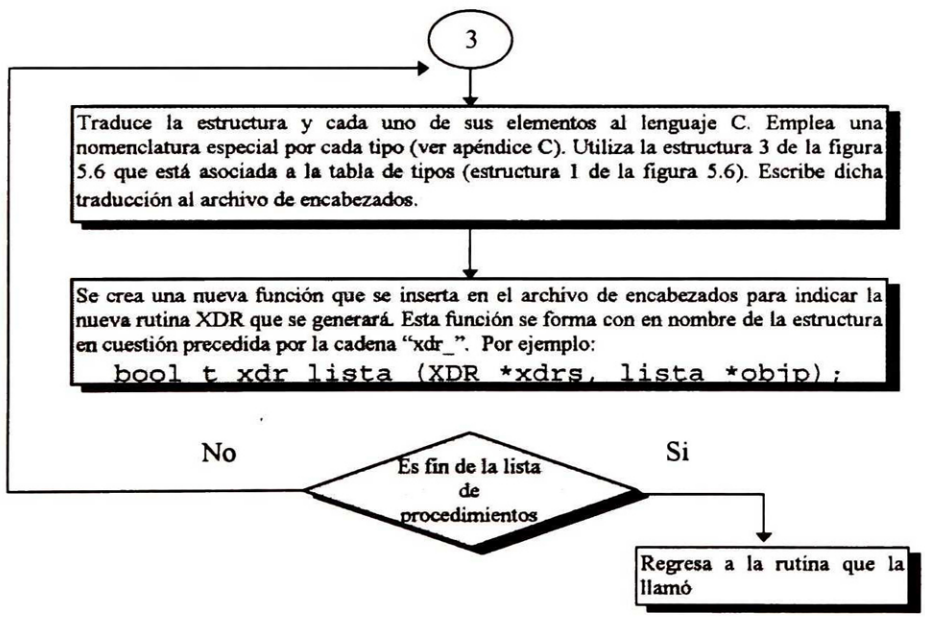


Figura 5-10. Procedimiento 3

5.5.2. Generador de rutinas XDR

En la figura 5.11 se presenta el algoritmo sobre el generador de rutinas XDR desarrollado en este compilador. El código de encabezados genera el archivo para las rutinas XDR. Este se genera con el nombre de la estructura en cuestión seguida se la cadena "_xdr.c". En el código para las rutinas XDR se utiliza dicho archivo previamente generado.

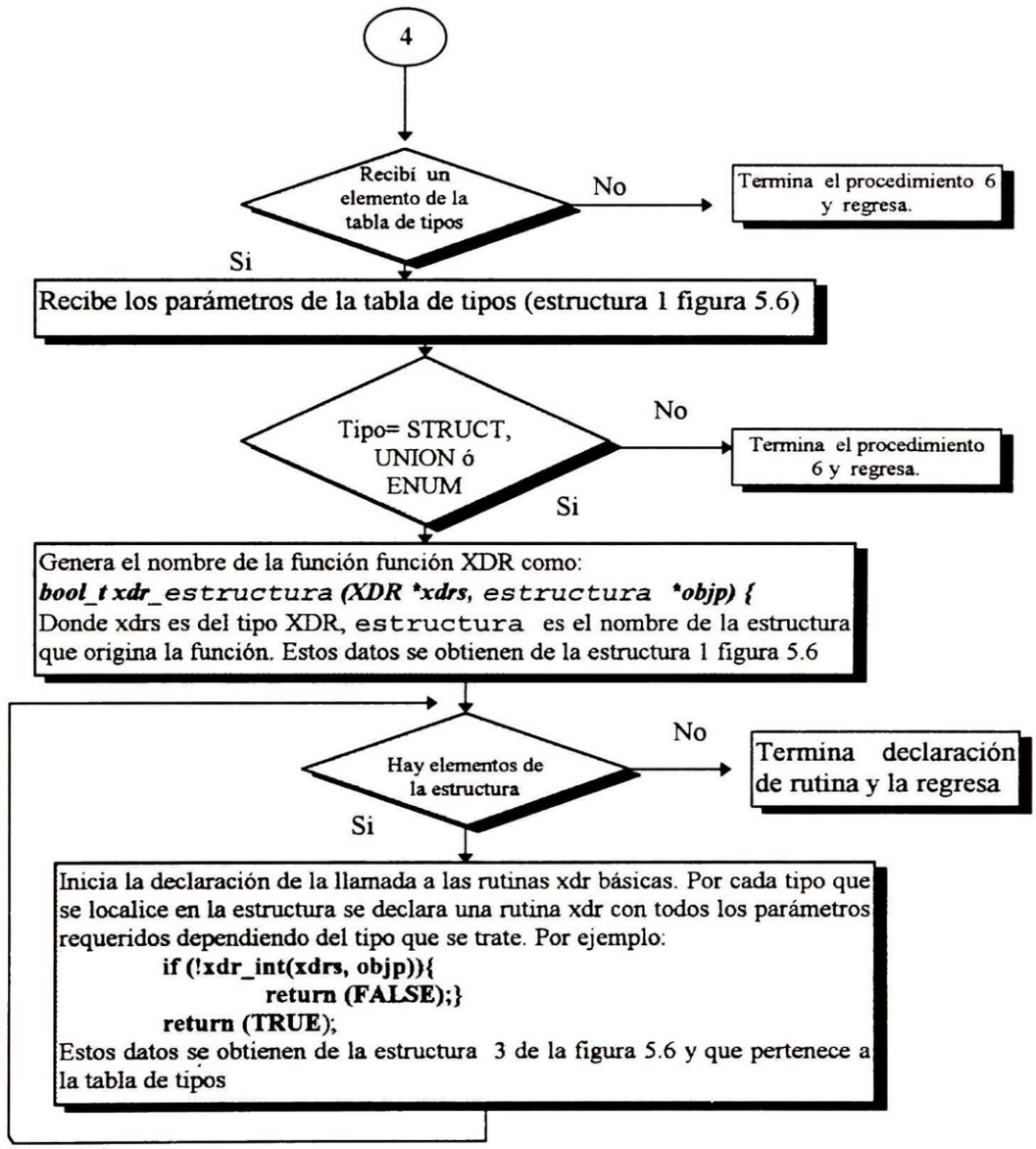


Figura 5-11. Algoritmo del generador de rutinas XDR (procedimiento 4).

5.5.3. Generador de rutinas del cliente (stub del cliente)

En la figura 5.12 se presenta el algoritmo para la generación del stub del cliente. Este algoritmo emplea la lista de procedimientos que se generó a partir de los análisis del compilador y se representa como la estructura 6 en la figura 5.6. En el stub del cliente se declaran todos los procedimientos recibirán las llamadas locales y se encargarán de realizar las llamadas remotas. Existe parte del código que es similar en cada una de las funciones que se declaran. Para ello se emplean estructuras conteniendo la información mencionada la cual se escribe directamente al archivo en el lenguaje C. Es el caso de la declaración de includes y algunos parámetros.

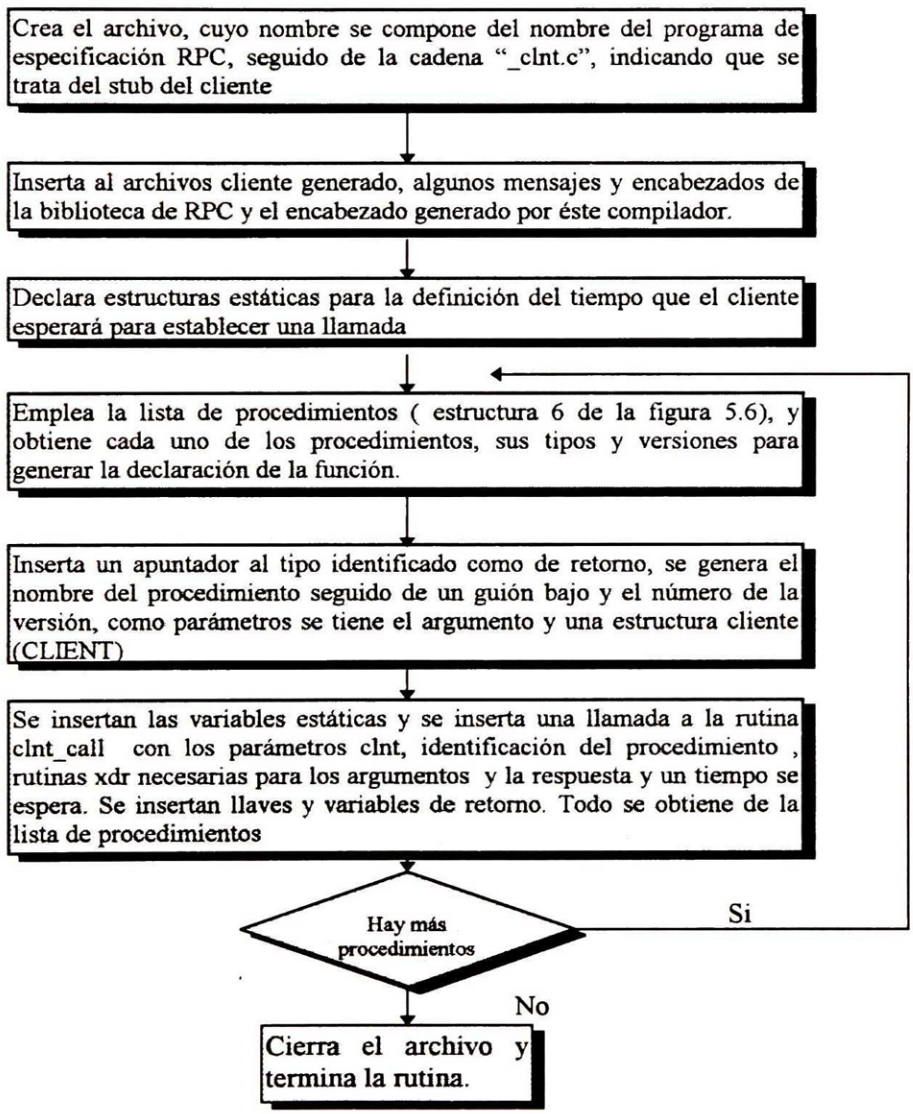


Figura 5-12 Algoritmo para la generación del Stub del cliente

5.5.4. Generador de rutinas del servidor (stub del servidor)

Para la generación del stub del servidor, se tiene almacenada en estructuras de datos, información que se emplea en cada stub y es fija. Por ejemplo siempre se llama a la rutina que crea el transporte UDP y siempre a la rutina que crea el transporte TCP, varían los parámetros que lleva. En la figura 5.13 se ilustra el algoritmo para la generación del stub del cliente.

El stub del servidor funciona como programa principal. Por lo que contiene el comando main() del lenguaje C, así como de un procedimiento general para cada declaración de programa y que se encarga de ejecutar cada petición remota.

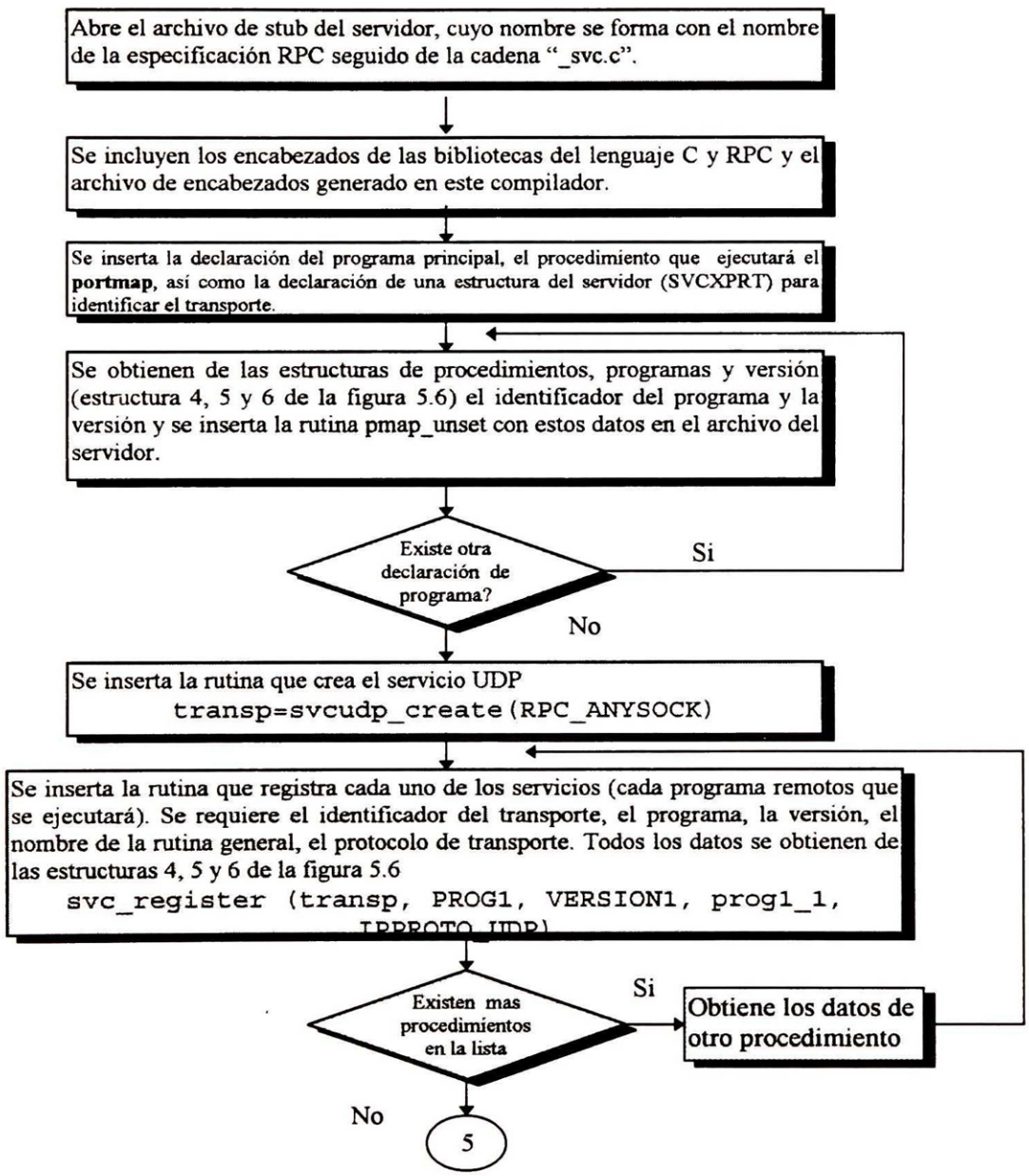


Figura 5-13. Algoritmo del stub del servidor

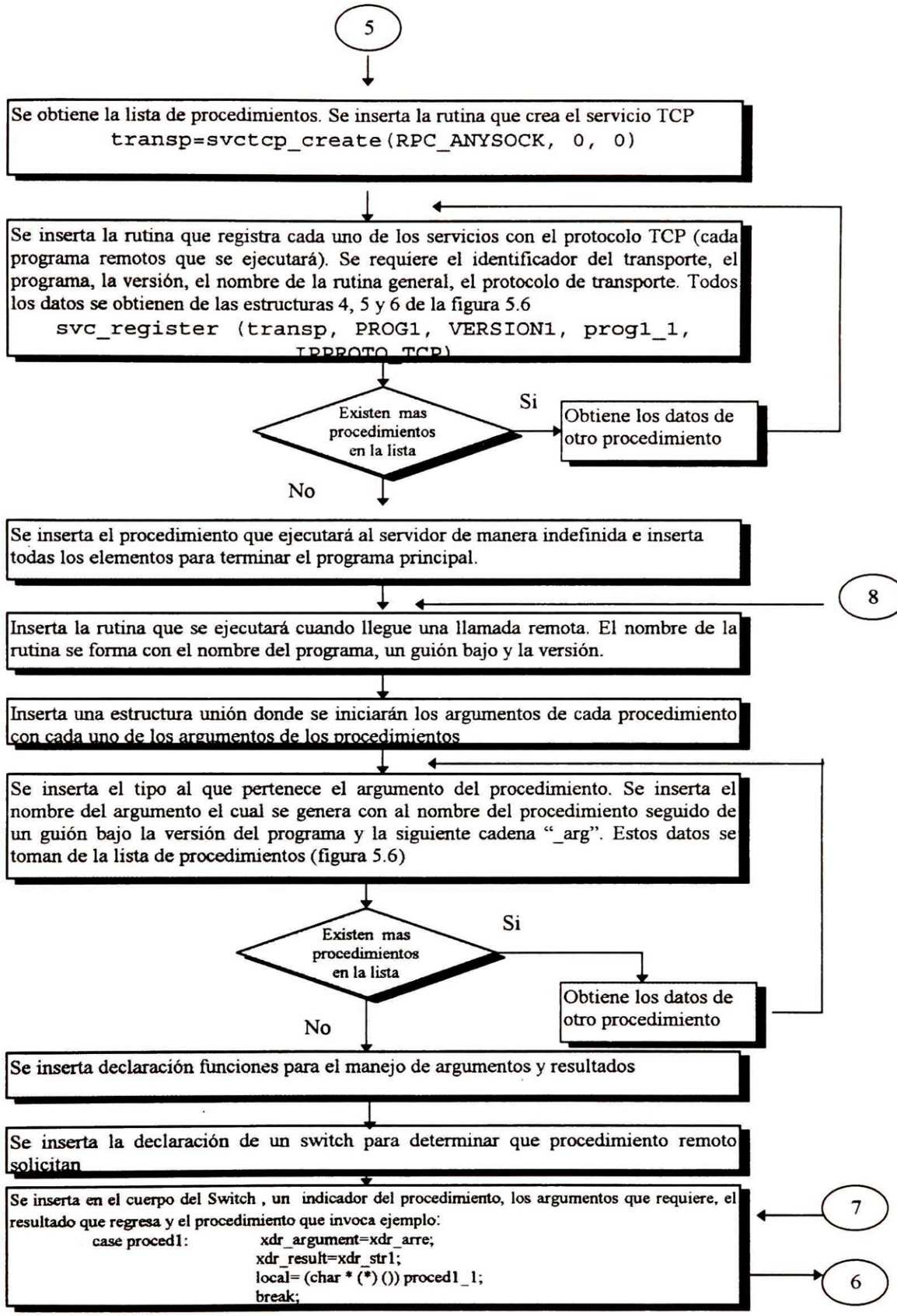


Figura 5-14. Continuación del algoritmo del stub del servidor

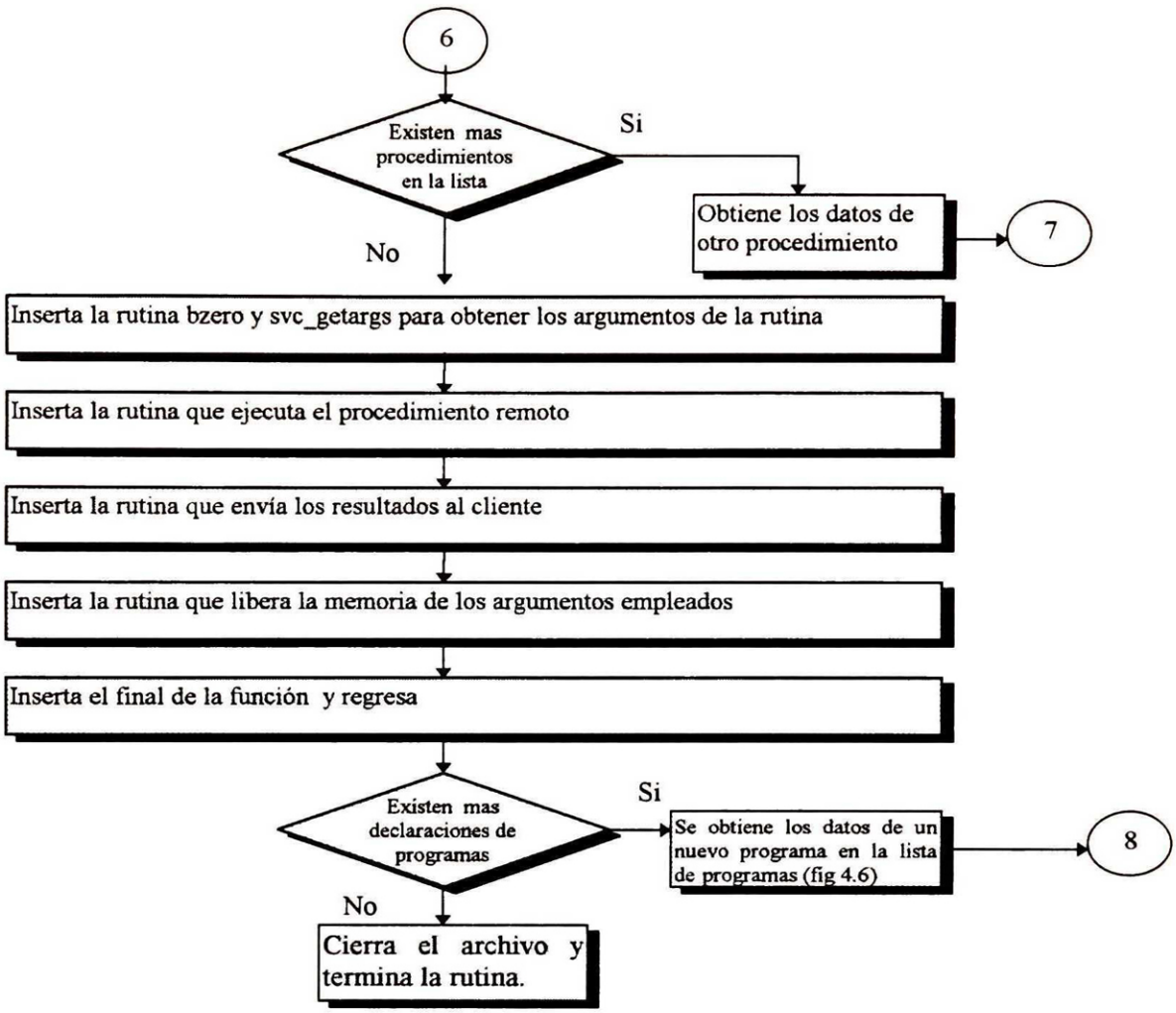


Figura 5-15. Continuación del algoritmo para la generación del stub del servidor

El compilador CRAD, diseñado particularmente para las computadoras personales, es similar al compilador rpcgen de SUN. La ventaja de tener un compilador para computadoras personales, es que no es necesario contar con una workstation con rpcgen para generar el código RPC (encabezados, stubs y rutinas XDR) y emplearlo con las rutinas de comunicación. Por lo tanto se consideró importante integrar todo un paquete (compilador, rutinas de comunicación y de conversión) para facilitar la programación de aplicaciones distribuidas.

CAPITULO

6

Ejemplo de generación de código

6

6. EJEMPLO DE GENERACION DE CODIGO

En este capítulo se presenta un ejemplo del empleo del compilador CRAD, la especificación en el lenguaje RPC, los encabezados, las rutinas xdr, los stubs del cliente y del servidor que genera el compilador, para la ejecución de un programa remoto.

6.1 *Planteamiento del problema*

El ejemplo que se plantea considera el manejo de una base de datos remota muy sencilla. Se exponen tres procedimientos para que se ejecuten de manera remota: Insertar un registro, consultar por apellido y consultar por numero telefónico.

Se maneja una estructura para el intercambio de información, conteniendo datos como nombre, apellido materno, apellido paterno, teléfono y dirección.

6.2 *Especificación del problema en lenguaje RPC*

Se genera una estructura denominada registro donde se declaran los campos `Nombre`, `ApellidoM`, `ApellidoP` y `dirección`, del tipo `string`, con una longitud máxima de `MAX_STR`. La variable teléfono del tipo `long` La especificación se presenta en el listado 6.1. El nombre del archivo RPC conteniendo la especificación es: `bd.rpc`

```
/* Definición para CRAD de una Base de Datos*/
const  MAX_STR = 255;
struct registro{
    string Nombre<MAX_STR>;
    string ApellidoM<MAX_STR>;
    string ApellidoP<MAX_STR>;
    long telefono;
    string dirección<MAX_STR>;
};
program PROGBD{
    version VERSBD{
          registro CVE_APP(string)=1;/*consulta por apellido paterno */
          registro CVE_TEL(int)=2;/*consulta por num. Telefónico */
          int INSERTA_REG(registro)=3;
    }=1;
}=0x30000001;
```

Listado 6.1. ***Ejemplo de la definición de protocolo RPC para una Base de datos (bd.rpc)***

6.3 Compilación de la especificación

Para la compilación de la especificación, se emplea el comando:

```
CRAD nombre del programa.rpc
```

Para la compilación del ejemplo se emplea:

```
CRAD bd.rpc
```

El compilador revisa las líneas de la especificación indicando los errores léxicos sintácticos y semánticos que detecte. La interfaz del compilador es sencilla. Los mensajes de error se muestran en pantalla indicando el error detectado y el número de línea donde se encuentra. En el caso de que el compilador no encuentre errores, genera los programas necesarios para establecer una llamada RPC.

El compilador genera el archivo de encabezados con el nombre del archivos de especificación y una extensión .h. En este caso el nombre del encabezado sería:

```
bd.h
```

Para las rutinas XDR se escribe el nombre del archivo de especificación seguido de la cadena "_xdr.c":

```
bd_xdr.c
```

Para el stub del cliente se emplea el nombre del archivo de especificación seguido de la cadena "_cln.c"

```
bd_cln.c
```

Para el stub del servidor se emplea el nombre del archivo de especificación seguido de la cadena "_svc.c"

```
bd_svc.c
```

Para el compilador es importante considerar el tamaño del nombre del archivo de especificación para evitar que al escribir los nombres se repitan.

6.4 Archivo de encabezados

En el listado 6.2 se presenta el archivo de encabezados generado por el compilador. Cada elemento de la especificación se codifica en el lenguaje C. (ver apéndice C y capítulo 3 para mayor referencia).

La versión del programa remoto es importante cuando existen múltiples generaciones de software dado que le permite a la aplicación cliente determinar si existe error por la versión que maneja el servidor.

El archivo de encabezados contempla la definición de constantes, la declaración de tipos y estructuras. La declaración de las rutinas XDR y de los procedimientos remotos.

```
/* Encabezados */
#include <types.h>

#define   MAX_STR 256
#define   MAX_STR 256
#define   PROGBD       ((u_long)0x30000001)
#define   VERSBD       ((u_long)1)
#define   CVE_APP      ((u_long)1)
#define   CVE_TEL      ((u_long)2)
#define   INSERTA_REG ((u_long)3)

typedef struct registro{
    char *Nombre;
    char *ApellidoM;
    char *ApellidoP;
    int telefono;
    char *dirección;
} registro;

bool_t xdr_registro(XDR *xdrs, registro *objp);

extern registro       *cve_app_1(char *argp, CLIENT *clnt);
extern registro       *cve_tel_1(int *argp, CLIENT *clnt);
extern int            *inserta_reg_1(registro *argp, CLIENT *clnt);
```

Listado 6.2. Archivo de encabezados

6.5 Código del cliente

El cliente en su programa principal inicia una estructura tipo CLIENT para establecer las sesiones remotas. Llama a la rutina `clnt_create ()` y si la rutina termina con éxito, el cliente puede llamar a los procedimientos remotos. Los nombres de procedimientos se forman con el nombre del procedimiento, guión bajo y la versión del programa. El código del programa principal del cliente se presenta en los listados 6.3 y 6.4.

```
/* Encabezados */
#include <stdio.h>
#include <ctype.h>
#include <rpc.h>
#include "bd.h"
#define DATABASE "personal.dat"
void imprime_reg(registro *)

void main(int argc, char *argv[]){
  CLIENT *cl;
  char *valor;
  int clave,p;
  registro *pr;

  if(argc!=4) {
    fprintf(stderr,"Usar: BaseD servidor cve_procedimiento valor");
    exit(1);
  }
  if (!(cl=clnt_create(argv[1],PROGBD,VERSBD,"tcp"))){
    clnt_createerror(argv[1]);
    exit(1);
  }

  valor=argv[3];
  clave=atol(argv[2]);
  switch (clave){

  case CVE_APP:
      imprime_reg (cve_app_1(&valor,cl));
      break;
  case CVE_TEL:
      if(!(sscanf(argv[3],"%ld",&p))!=1){
            fprintf(stderr, "la Rutina requiere un valor numérico");
            exit(1);
      }
      imprime_reg(cve_tel_1(&p,cl));
      break;

  case INSERTA_REG:
      pr=(registro *) malloc (sizeof(registro));
      pr->Nombre=(char *)malloc (MAX_STR);
      pr->ApellidoP=(char*)malloc (MAX_STR);
      pr->ApellidoM=(char *)malloc (MAX_STR);
      pr->direccion=(char *)malloc  (MAX_STR);
      if(sscanf(argv[3], "%s%s%s%d%s" ,pr->Nombre,pr->ApellidoP,
                  pr-> ApellidoM,&(pr->telefono),pr->direccion) != 5) {
            fprintf(stderr,"Datos incompletos para la rutina
                  inserta_reg");
            exit(1);
      }
```

Listado 6.3 Código del programa principal del cliente

```
        if(!(*inserta_reg(pr,cl))){
                    fprintf(stderr, "no se pudo insertar Registro");
                    exit(1);
        }
        break;
    default:
        fprintf(stderr, "%s: clave desconocida \n", argv[0]);
        exit(1);
    }
}
/*-----------------------------------
  Rutina IMPRIME_REG
-----------------------------------*/
void imprime_reg(registro *rg){
     printf("Nombre\tApellido M \t ApellidoP \tTeléfono \tDirección");
     printf("%s\t%s\t%s\t%d\t%s\n", rg->Nombre,rg->ApellidoM,\
            rg->ApellidoP,rg->telefono,rg->direcion);
}
```

Listado 6.4 Continuación del programa principal del cliente

• En este ejemplo se declara la rutina `imprime_reg()` empleada para imprimir un registro de la base de datos.

•Los argumentos del programa principal, indican que al ejecutar este programa debe recibir tres argumentos, el servidor remoto, la clave del procedimiento remoto que se ejecutará y el valor ó el dato que se procesará.

•Se declaran la estructura CLIENT y las variable valor, clave, pr.

•Se ejecuta la rutina `clnt_create`. El `argv[1]`, indica el servidor, y los parámetros restantes son el programa, la versión y el protocolo, en este caso `tcp`.

•En el caso de la consulta por apellido paterno, se envía la cadena recibida en el programa principal e imprime el resultado.

•En el caso de la consulta por número telefónico se verifica que el dato enviado desde el programa principal sea un valor numérico.

•En el caso del procedimiento de insertar registro, se solicita memoria para cada elemento de la estructura y solicita los datos para cada elemento. Se llama a la rutina se inserción.

6.6 *Stub del cliente*

En el stub que genera CRAD se presentan los procedimiento que se llaman en el programa principal. Estos procedimientos emplean las rutinas RPC para establecer la sesión remota adecuada. (ver capitulo 3 para mayor referencia y apéndice C). El stub que genera el compilador se presenta en el listado 6.5

```
#include <rpc.h>
#include "bd.h"
static struct timeval TIMEOUT= {25,0}

registro *cve_app_1(char *argp, CLIENT *clnt){
  static registro res;
  bzero ((char *)&res, sizeof(res));
  if(clnt_call(clnt,CVE_NOM,xdr_string, argp, xdr_registro, &res,
     TIMEOUT) != RPC_SUCCESS){
     return (NULL);
  }
  return(&res);
}
registro *cve_tel_1(int *argp, CLIENT *clnt){
  static regitro res;
  bzero ((char *)&res, sizeof(res));
  if(clnt_call(clnt,CVE_NOM,xdr_int, argp, xdr_registro, &res,
     TIMEOUT) != RPC_SUCCESS){
     return (NULL);
  }
  return(&res);
int *inserta_reg_1(registro *argp, CLIENT *clnt){
  static registro res;
  bzero ((char *)&res, sizeof(res));
  if(clnt_call(clnt,CVE_NOM,xdr_registro, argp, xdr_int, &res,
     TIMEOUT) != RPC_SUCCESS){
     return (NULL);
  }
  return(&res);
```

Listado 6.5. Stub del cliente

6.7 Filtros XDR

Para cualquier estructura declarada o nuevo tipo en la especificación se deben generar sus rutinas XDR. En el listado 6.6 se presenta el código del filtro XDR para la estructura registro.

```
#include <rpc.h>
#include "bd.h"

bool_t xdr_registro(XDR *xdrs, registro *objp){
   if (!xdr_string (xdrs, &objp->Nombre,MAX_STR)){
            return (FALSE);}
   if (!xdr_string (xdrs, &objp->ApellidoM,MAX_STR)){
            return (FALSE);}
   if (!xdr_string (xdrs, &objp->ApellidoP,MAX_STR)){
            return (FALSE);}
   if (!xdr_int(xdrs, &objp->telefono,MAX_STR)){
            return (FALSE);}
   if (!xdr_string (xdrs, &objp->dirección,MAX_STR)){
            return (FALSE);
   }
   return(TRUE);
}
```

Listado 6.6 Filtros XDR.

6.8 Stub del Servidor

El stub del servidor es el programa principal del servidor. En éste programa se inicia el portmap. En los listados 6.7 y 6.8 se presenta el código del stub del servidor.

```
#include <stdio.h>
#include <rpc.h>
#include "bd.h"

static void progbd_1();
main(){
   register SVCXPRT *transp;
   portmap_reg();
   (void) pmap_unset(PROGBD, VERSBD);
   transp=svcudp_create(RPC_ANYSOCK);
   if (transp==NULL){
      fprintf(stderr, "No se puede crear el servicio udp")
      exit(1);
   }
   if (!svc_register(transp, PROGBD,VERSBD,progbd_1, IPPROTO_UDP)){
      fprintf(stderr, "No se puede registrar a (PROGBD,VERSBD,udp)");
      exit(1);
   }

   transp=svctcp_create(RPC_ANYSOCK, 0, 0);
      if (transp==NULL){
      fprintf(stderr, "No se puede crear el servicio tcp")
      exit(1);
   }
   if (!svc_register(transp, PROGBD,VERSBD,progbd_1, IPPROTO_TCP)){
      fprintf(stderr, "No se puede registrar a (PROGBD,VERSBD,tcp)");
      exit(1);
   }
   svc_run();
   fprintf(stderr,"svc_run regreso");
   exit(1);
}

static void progbd_1(struct svc_req *rqstp, register SVCXPRT *transp){
   union{
      registro cve_app_1_arg;
      registro cve_tel_1_arg;
      int inserta_reg_1_arg;
   }argument;
   char *result;
   bool_t (*xdr_argument)(), (*xdr_result)();
   char *(*local)();
```

Listado 6.7. Stub del servidor

```
    switch(rqstp->rq_proc) {
       case NULLPROC:
             (void) svc_sendreply(transp, xdr_void, (char*)NULL);
             return;
       case CVE_APP:
             xdr_argument=xdr_string;
             xdr_result=xdr_registro;
             local=(char *)(*)())cve_app_1;
             break;
       case CVE_TEL:
             xdr_argument=xdr_int;
             xdr_result=xdr_registro;
             local=(char *)(*)())cve_tel_1;
             break;
       case INSERTA_REG:
             xdr_argument=xdr_registro;
             xdr_result=xdr_int;
             local=(char *)(*)())cve_app_1;
             break;
       default:
             svcerr_noproc(transp);
             return;
       }
       bzero((char *)&argument, sizeof(argument));
       if (!svc_getargs(transp, xdr_argument, &argument)) {
             svcerr_decode(transp);
             return;
       }
       result = (*local) (&argument,rqstp);
       if(result != NULL && !svc_sendreply (transp, xdr_result, result)){
             svcerr_systemerr(transp);
       }
       if (!svc_freeargs(transp, xdr_argument, &argument)){
             fprintf(stderr, "no se pueden liberar los argumentos");
             exit(1);
       }
       return;
}
```

Listado 6.8. Stub del servidor

6.9 Rutinas locales del servidor

El stub del servidor ejecuta las rutinas locales para atender las solicitudes del cliente. En los listados 6.9 y 6.10 se presentan las rutinas locales del servidor.

```
#include <stdio.h>
#include <string.h>
#include <rpc.h>
#include "bd.h"

FILE *fp = NULL;
static record rg =NULL;
int leeReg(void);

registro *cve_app_1(char **nom){
   if(!(arch=fopen(BASEDATOS ,"r")))
                  return((registro *)NULL);
   while (leeReg())
            if(!reg->ApellidoP, *nom))
                  break;
   if (feof(arch)){
            fclose(arch);
            return((registro *)NULL);
   }
   fclose(arch);
   return((registro *)reg);
}
registro *cve_tel_1(int **num){

   if(!(arch=fopen(BASEDATOS ,"r")))
            return((registro *)NULL);
   while (leeReg())
      if(!reg->telefono, *num))
            break;
   if (feof(arch)){
            fclose(arch);
            return((registro *)NULL);
   }
   fclose(arch);
   return((registro *)reg);
}
int *inserta_reg_1(registro *rg){
   static int edo;

   if(!(arch=fopen(BASEDATOS ,"a")))
            return(1);
   edo=fwrite(arch,rg);
   fclose(arch);
   return((int *)&edo);
}
```

Listado 6.9. Rutinas locales del servidor

```
int leeReg(){

   char buf [MAX_STR];
   if (!rg){
            rg= (record *) malloc (sizeof (record));
            rg->Nombre= (char *)malloc(MAX_STR);
            rg->dirección= (char *)malloc(MAX_STR);
            rg->ApellidoP= (char *)malloc(MAX_STR);
            rg->ApellidoM= (char *)malloc(MAX_STR);

   }
   if (!gets (buf, MAX_STR-1, fp))
            return (0);
   if (sscanf (buf, "%s%s%s%d%s", rg->Nombre, rg->dirección,
      rg->ApellidoP, rg->ApellidoM, &rg->telefono)!=5)
            return (0);
   return (1);
}
```

Listado 6.10. Continuación de las rutinas locales del servidor.

El ejemplo mostrado representa una base de datos muy elemental, en la que el cliente solicita en línea el servicio deseado. Se pueden generar aplicaciones tan complejas como se requieran. Este ejemplo sólo se empleó para demostrar el uso del compilador CRAD.

CONCLUSIONES

Conclusiones

En este proyecto de tesis se diseñó e implantó una herramienta para la programación de aplicaciones distribuidas la cual cumple con la mayoría de las características que definen el medio ambiente de la programación distribuida mencionadas en [Bloomer92], las cuales se citan en los siguientes puntos:

1) Herramientas de representación de datos independientes de la máquina para permitir el intercambio de datos entre las diferentes plataformas.
2) Un protocolo para especificar responsabilidades de bajo nivel de un cliente y un servidor durante una llamada a procedimiento remoto.
3) Un compilador de protocolo que toma una definición de procedimiento remoto, incluyendo argumentos y tipos y genera interfaces stub RPC. Un compilador de protocolo además de aislar la aplicación del protocolo básico y de la programación de bajo nivel en la red, genera programas para manejar la codificación, la decodificación y el registro de servicios.
4) Servicios de autenticación para aumentar la seguridad en los sistemas operativos estándar y es requerido como parte de cualquier acceso remoto.
5) Servicios de nombres de la red que permiten que las aplicaciones localicen los objetos de red convencionales (usuarios, hosts o mailboxes) y enlazar a servicios que se han registrado en la red.
6) Un tiempo de servicio de red para la sincronización del host.
7) Un sistema de archivo distribuido para eliminar redundancia.

Para cumplir con las características que definen el medio ambiente de la programación distribuida mencionadas en los puntos anteriores, este trabajo de tesis cumple con la mayoría de lo puntos citados. Se generó el compilador CRAD que cubre el ***punto 3***. Con CRAD la programación de las aplicaciones distribuidas se simplifica, porque sólo es necesario definir el procedimiento remoto en el lenguaje RPC (cuyo aprendizaje es simple) similar a la declaración de estructuras y tipos en el lenguaje C. Para probar el funcionamiento de los programas que genera el compilador (los stubs) fue necesario la programación de las rutinas para la codificación y decodificación de los datos en el lenguaje RPC ***(cumple en punto 2)*** y en el formato estándar XDR ***(cumple el punto 1).*** Además se requirió el establecimiento de la comunicación entre computadoras a través de un ambiente de red bajo la familia de protocolos TCP/IP Para contactar a las computadoras de manera remota, debe existir un servidor encargado de resolver los nombres de las máquinas conectadas en la misma red o en otras redes, o se puede configurar en cada una de las computadoras un archivo conteniendo nombres y direcciones de máquinas ***(cumple el punto 5).***

El ambiente de la programación distribuida se cumple en casi todos los aspecto en este trabajo. Señalamos lo anterior porque este compilador no cubre los protocolos de autenticación ***(cumple el punto 4)***. Para ello los usuarios deben programar sus propias rutinas ó protocolos de autenticación para proveer seguridad en sus sistemas. Aunque no se cubre el punto anterior, los programas desarrollados contemplan la posibilidad de anexar fácilmente los protocolos de autenticación como los de UNIX o DES.

Los clientes y servidores que genera CRAD y las rutinas de comunicación no guardan información acerca de las transacciones que efectuó el usuario ni se maneja una base de datos para redundancia en operaciones ***(cumple el punto 7)***.

Con respecto al ***punto 6***, se maneja un temporizador para la respuesta del cliente y la verificación de caídas del servidor. El programador puede cambiar este tiempo. Por el lado del servidor, se maneja una serie de rutinas para detectar caídas de clientes y desechar la respuestas y cerrar conexiones en caso necesario.

En el compilador CRAD se resolvieron algunos problemas no contemplados en el compilador del protocolo RPC de SUN (RPCGEN). Estos problemas, relacionados con la detección de errores, se manifestaban cuando se compilaban el cliente y el servidor con el compilador de C y otros se tornaban en errores lógicos y no sintácticos que son más difíciles de detectar.

Con CRAD, se pueden diseñar aplicaciones que corran en un ambiente donde los clientes y servidores sean computadoras personales o bien se puede instalar un cliente en una Workstation SUN y el servidor en una computadora personal o el servidor en la workstation SUN (cuya aplicación puede realizarse con el compilador rpcgen o programar de manera directa con las librerías RPC) y el cliente en una computadora personal (generado con el compilador CRAD o por medio de la programación directa con las librerías RPC y XDR elaboradas en esta tesis).

Este trabajo de tesis es una herramienta importante para la programación de aplicaciones distribuidas y puede emplearse en diversas aplicaciones. Es necesario recalcar que elimina mucho del trabajo de comunicación que se tiene que elaborar para dichas aplicaciones. Cabe mencionar que aunque actualmente existe mucha información al respecto y programas fuentes libres vía internet, este trabajo, se desarrolló en su totalidad y se adquirieron conocimientos referentes a la programación distribuida, protocolos de comunicación, interfaces de programación, compiladores y aplicaciones similares. Adicionalmente a las facilidades que proporciona este compilador, los conocimientos adquiridos durante su elaboración son parte importante de la formación personal.

Apéndice

A

XDR

Apéndice A . XDR : ESTÁNDAR PARA LA REPRESENTACIÓN DE DATOS EXTERNOS (EXTERNAL DATA REPRESENTATION STANDARD)

Introducción:

XDR es un estándar para la descripción y codificación de datos [RFC1014], empleado para la transferencia de datos entre máquinas con arquitecturas diferentes y utilizado en estaciones de trabajo como SUN, VAX, IBM-PC y CRAY.

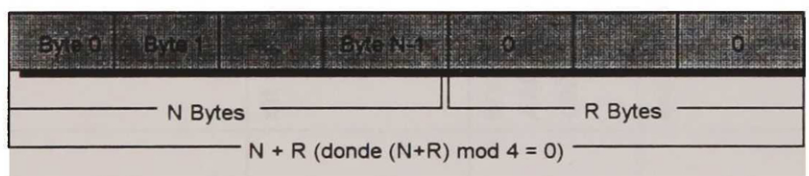
XDR contiene un conjunto de rutinas que permiten al programador de del lenguaje C describir arbitrariamente estructuras de datos

XDR emplea un lenguaje descriptivo (no de programación) para definir el formato de los datos. La sintaxis y semántica de los datos es similar al lenguaje C. Los protocolos como RPC de ONC y el NFS usan XDR para describir el formato de sus datos.

En éste estándar un dispositivo debe codificar los bytes (u octetos) que representan los datos de tal manera que el otro dispositivo pueda desempaquetar los bytes sin perder el significado. Por ejemplo el estándar Ethernet sugiere que los bytes se codifiquen en estilo "little-endian", o el primer bit menos significativo. XDR maneja maneja este mismo estilo.

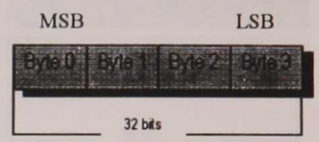
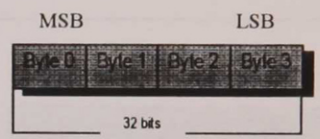
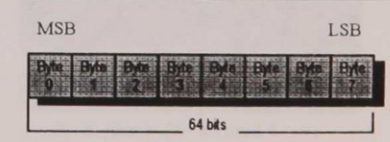
Tamaño básico de bloque.

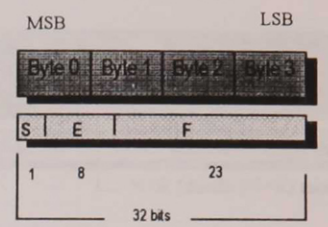
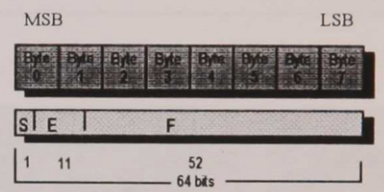
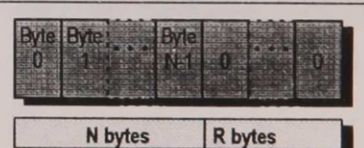
En XDR, la representación de todos los tipos de datos requiere de bloques múltiplos de cuatro bytes (32 bits). Los bytes se numeran de 0 a N-1. Si los N bytes de los datos, no son múltiplos de cuatro, entonces se anexan R octetos de 0 a 3 para tener un múltiplo de 4.

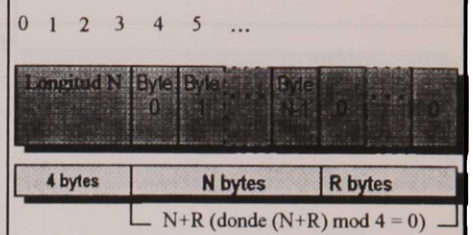
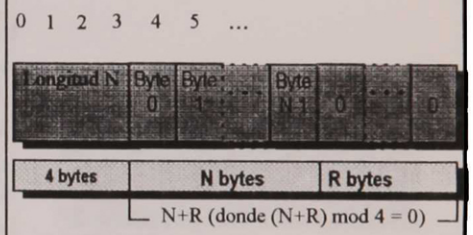
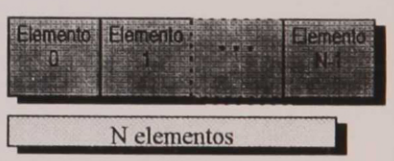


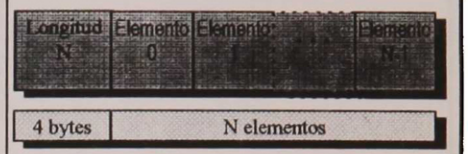
En la siguiente tabla se presenta una descripción de los tipos definidos en el estándar XDR así como la representación de cada uno.

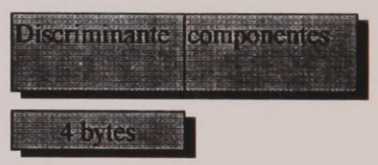
ESPECIFICACIÓN DEL PROTOCOLO XDR (TIPOS DE DATOS)

TIPO DE DATO	DECLARACION	EJEMPLO	NOTA	REPRESENTACIÓN GRÁFICA
Entero con signo	**int** *identificador;*	int *ejem;*	Se representa en notación complemento a dos. 32 bits Intervalo: -2147483648, 2147483647	
Entero sin signo	**int** *identificador;*	int *ejem;*	Intervalo: 0, 4294967295 32 bits	
Enumeración	**enum{** *nombre_identificador* = constante, ...**}** *identificador ;*	enum { *ROJO*=2, *AMARILLO*=3, *AZUL*=5} *colores;*	Se emplea para describir un subconjunto de enteros. Cada entero se representa como un entero con signo.	
Booleano	**bool** *identificador;*	bool *OK;*	Se representa como un entero con signo. Es equivalente a : enum {FALSE=0, TRUE=1} OK;	
Enteros largos (hyper) y enteros largos sin signo	**hyper** *identificador;* **unsigned hyper** *identificador;*	hyper *ejem;* unsigned hyper *ejemp;*	Números de 64 bits (8 bytes). Sus representación son las extensiones de entero y enteros sin signo definidas arriba. Representación en	

TIPO DE DATO	DECLARACION	EJEMPLO	NOTA	REPRESENTACIÓN GRÁFICA
			complemento a dos. 64 bits.	
Punto Flotante	**float** *identificador;*	float *ejem;*	Se codifica de acuerdo al estándar IEEE para normalización de punto flotante de precisión simple. 32 bits. Representación: S: Signo del número. Positivo=0, Negativo=1. E: Exponente del número (base 2, bias 127) F: Parte fraccionada de la mantisa) $(-1)^s \times 2^{(E-bias)} \times 1.F$	 LSB=31
Punto flotante de doble precisión	**double** *identificador;*	double ejem;	Se emplea una codificación similar a la definida para los datos de tipo punto flotante. Utiliza 64 bits.	
Punto flotante de cuádruple precisión	**quadruple** *identificador;*	quadruple ejem;	El estándar XDR actual (1995), define este tipo. Emplea 128 bits.	
Datos opaque de longitud fija	**opaque** *identificador* **[***n***];** n= Tamaño del arreglo en bytes. Si no es múltiplo de 4 se agregan R bytes de relleno.	opaque ejem [5];	Los datos opaque se emplean para el intercambio de caracteres especiales (no ASCCII).	 N+R (donde (N+R) mod 4 = 0)

TIPO DE DATO	DECLARACION	EJEMPLO	NOTA	REPRESENTACIÓN GRÁFICA
Datos opaque de longitud Variable	**opaque** *identificador <n>;* **opaque** *identificador <>;* *n = número máximo de bytes del arreglo*	opaque *datos_archivo <8192>;*	Si no se especifica el tamaño máximo del arreglo (n) se toma el valor de (2^{32}) - 1. Si se codifica una longitud máxima que la descrita en la especificación, se tomará como error. La longitud máxima de los datos se codifica en 4 bytes al inicio de la trama (un entero sin signo).	
Caracteres (string)	**string** *identificador <n>;* *ó* **string** *identificador <>;* *n = número máximo de bytes del arreglo*	string *nombre_archivo <255>;*	Un string se define como n bytes ASCII (enteros sin signo). Si no se especifica n, se asume como longitud máxima : (2^{32}) - 1. El valor n se codifica en 4 bytes al inicio de la trama.	
Arreglos de longitud fija	**Nombre_del_tipo** *identificador* [n];	int arreglo_de [3];	Se representan elementos homogéneos y se codifican individualmente. El tamaño de cada elemento debe ser múltiplo de 4. Aunque los elemento sean del mismo tipo, pueden tener tamaños diferentes. Por ejemplo: un arreglo de longitud fija de	

TIPO DE DATO	DECLARACION	EJEMPLO	NOTA	REPRESENTACIÓN GRÁFICA
			strings	
Arreglos de longitud variable	**Nombre_tipo** *identificador <n>;* *ó* **Nombre_tipo** *identificador <>;*	int arreglo_var <>;	Si no se especifica el número máximo de elementos en el arreglo (n) se asigna el tamaño de : $(2^{32}) - 1$	
Estructuras	**struct {** declaración _comp_A; declaración_comp_B; . . . **}** *identificador;*	struct { int a; string b <5>; } datos;	Los componentes de la estructura se codifican en el orden de sus declaraciones en la estructura. El tamaño de cada componente es múltiplo de 4 bytes.	

Union Discriminante	**union switch** (*declaración_del_discriminante*) **{**
	case *valor_del_disc_A:*
	declaración_de_A;
	case *valor_del_disc B:*
	declaración_de_B;
	. . .
	default: *declaración_default;*
	} *identificador;*
		union switch (int valor_rut){
		case 0:
		return(rutina_1());
		case 1:
		return(rutina_2());
		default:
		return(rutina_3));
		} ejecuta_rutina;
			Es un tipo compuesto por un discriminante que permite diferenciar o separar un tipo de acuerdo al valor que se le asigne. El discriminante puede ser un entero o un tipo bool y se emplean 4 bytes para su codificación.	

Void	**void;**	void	Tiene 0 bytes y se emplea	

TIPO DE DATO	DECLARACION	EJEMPLO	NOTA	REPRESENTACIÓN GRÁFICA
			para describir operaciones que no tienen datos de entrada o de salida.	
Constantes	**const** *nombre_del_identificador = n;*	const DOCE = 12;	Usado para definir nombres simbólicos para una constante No declara ningún dato	
Typedef. Definición de nuevos tipos	**typedef** *declaración;*	typedef art caja_art [DOCE];	No declara ningún dato, pero sirve para definir nuevos identificadores para declarar datos	

Datos Opcionales	**nombre_del_tipo** **identificador;*	int *pp;
		
		struct *stringlist {
		string campo<>;
		stringlist next;
		};
		
		Puede declararse como:
		
		union stringlist switch (bool opción){
		case TRUE:
		struct {
		string campo<>;
		stringlist next;
		} element;
		case FALSE:
		void;
		};
			Estos datos equivalen a una unión como la siguiente:
			union switch (bool opted) {
			case TRUE:
			Nombre_tipo_elemento;
			case FALSE:
			void;
			} identificador;
			Los datos opcionales no son tan interesantes en sí mismos, pero son muy útiles para describir estructuras de datos recursivas, tales como listas enlazadas y árboles	

Especificación del lenguaje xdr

La especificación del lenguaje, utiliza la notación de la forma Backus_Naur [Ullman90]. A continuación se presenta una pequeña introducción de ésta notación:

- Los caracteres '|', '(', ')', '[', ']', ' " ' y '*', son especiales.
- Los símbolos terminales son cadenas de cualquier tipo de caracteres, encerradas entre comillas dobles.
- Los símbolos no terminales son cadenas de caracteres no especiales.
- Cada parte o elemento alternativo se separa por una barra vertical.
- Las partes opcionales se encierran entre corchetes.
- Los elementos se agrupan encerrándolos entre paréntesis.
- un '*' que precede a un elemento indica que existe 0 o más ocurrencias de ése elemento.

Dentro de las consideraciones léxicas importantes tenemos:

- Los comentarios inician con '/*' y terminan con '*/'.
- Los espacios en blanco sirven para separar elementos y se ignoran al evaluar.
- Un identificador es una letra seguido por una secuencia opcional de letras, dígitos o guión bajo ('_').
- Una constante es una secuencia de uno o más dígitos decimales, opcionalmente precedido por un signo de menos ('-').

En el Listado A.1 se presenta la sintaxis de la especificación del lenguaje XDR

Sintaxis :

```
declaration:    type_specifier identifier
              | type_specifier identifier "[" value "]"
              | type_specifier identifier "<" [value] ">"
              | "opaque" identifier "[" value "]"
              | "opaque" identifier "<"  [value ] ">"
              | "string" identifier "<" [value] ">"
              | type_specifier "*" identifier
              | "void"

value:        constant
              | identifier

type_specifier: ["unsigned"] "int"
              | ["unsigned"] "hyper"
              | "float"
              | "double"
              | "quadruple"
              | "bool"
```

```
                |       enum-type-spec
                |       struct-type-spec
                |       union-type-spec
                |       identifier

enum-type-spec: "enum" enum-body


enum-body:              "{"     (identifier "=" value)
                        ("," identifier "=" value) *
                "}"

struct-type-spec:   "struct" struct-body

struct-body:        "{"     (declaration ";")
                        (declaration ";")*
                "}"
union-type-spec:    "union" union-body

union-body:         "switch" "(" declaration ")"  "{"
                        ("case" value ":" declaration ":" )
                        ("case" value ":" declaration ":" ) *
                        [ "default" ":" declaration ";" ]
                "}"
constant-def:       "const" identifier "=" constant ";"

type-def:           "typedef" declaration ";"
                |   "enum" identifier enum-body ";"
                |   "struct" identifier struct-body ";"
                |   "union" identifier union-body ";"

definition:         type-def
                |   constant-def

specification:      definition *
```

Listado A.1 Especificación del lenguaje XDR

Notas de la Sintáxis:

- Son palabras reservadas: `"bool", "case", "const", "default", "double", "quadruple", "enum", "float", "hyper","opaque", "string", "struct", "switch","typedef","union", "unsigned"` y `"void"`.

- Solamente se puede utilizar constantes sin signo como especificación del tamaño de los arreglos. Si se emplea algún identificador, debe declararse como una constante sin signo previamente, con la definición `"const"`.

- Las constantes y los identificadores de tipos dentro del alcance de una especificación, están en el mismo espacio y deben declararse únicamente dentro de este ámbito.

- Similarmente, los nombres de variables debe ser únicas dentro de alcance de la declaración de una estructura y unión.

- El discriminante de la unión debe ser de un tipo que evalúe a un entero. Esto es `"int"`, `"unsigned int"`, `"bool"`, un tipo enumerado o un tipo definido por `typedef`, que evalúe a uno de los tipos legales. También los valores de cada uno de los elementos de la unión (`case`), debe tener un valor legal y no puede declararse más de una vez.

Apéndice

B

Modelo cliente - servidor

Apéndice B MODELO CLIENTE-SERVIDOR

El modelo cliente/servidor permite compartir dispositivos conectados en red. En este medio ambiente, las computadoras están asociadas a uno o varios sistema que les permiten compartir recursos comunes (archivos, discos, impresoras, etc.). En la terminología de las LAN, el sistema que administra el dispositivo compartido se conoce como *servidor* (un servidor de archivo, un servidor de impresión, etc.). El cliente es la computadora que tiene acceso al recurso compartido. Otra manera de explicar este modelo es que el servidor proporciona recursos, mientras que el cliente los consume.

Para implementar el modelo RPC es necesario involucrarse con el diseño y tipos básicos de clientes y servidores, sus ventajas y desventajas así como el manejo del interfaz que permitiría establecer sesiones remotas. En este apéndice se explicarán estos detalles.

Implantación del modelo Cliente-Servidor

Al diseñar software cliente/servidor es importante que éste se apegue a estándares. En este proyecto de tesis se usa la familia de protocolos TCP/IP para la comunicación entre las computadoras. TCP/IP, como muchos protocolos de comunicación, proporciona mecanismos básicos para la transferencia de datos, soporte a muchos protocolos estándar como: TELNET, FTP, SMTP, etc. y nos permite entablar comunicación entre dos aplicaciones e intercambiar datos entre ellas. Así decimos que TCP/IP proporciona comunicación entre entidades pares (peer-to-peer). [Cisco93].

Además de apegarse a un estándar, la comunicación cliente/servidor puede elegir dos tipos de interacción: *Una orientada a conexión y la otra sin conexión*. Estos dos estilos corresponden a los dos protocolos de transporte que proporciona TCP/IP : el protocolo TCP ofrece un servicio con conexión mientras que el protocolo UDP ofrece un servicio sin conexión.

Las aplicaciones que usan TCP/IP deben especificar varios detalles como: funcionamiento como cliente o servidor; la dirección extremo o remota que empleará; el tipo de comunicación (protocolo orientado a conexión o sin conexión); las reglas de autorización y de protección; y detalles tales como el tamaño de los "buffers"

Para implementar el modelo cliente/servidor utilizando TCP/IP podemos emplear una interfaz para crear comunicación entre dos puntos de acceso al servicio, conocido como *Interfaz de socket.* [Woll90].

Interfaz de sockets

Un socket es un mecanismo que proporciona puntos terminales o extremos ("endpoints") de comunicación entre procesos independientes. Para un programa un socket es sólo un número (canal o descriptor de archivo) que está relacionado a una estructura de datos que contiene una dirección de red, además de información adicional.

Las aplicaciones deben tener asociadas direcciones de puntos extremos. Cada dirección de punto extremo está formada por:

- *Una dirección Internet (dirección IP)*. Es un entero de 4 bytes que identifica una dirección de host dentro de la familia de direcciones AF_INET.
- *Un Puerto*. es un entero sin signo de 2 bytes que identifica una aplicación en un host.

Los tipos de sockets se identifican de acuerdo a los servicios que desea el usuario. Existen sockets: "stream" y "datagram".

Los *"sockets stream"*, también conocidos como sockets con conexión, usan el protocolo TCP para proporcionar un flujo de datos bidireccional, confiable, en secuencia, y sin datos duplicados.

Un *socket Datagram* ó socket sin conexión, usa el protocolo UDP. Cada mensaje se direcciona (enruta) individualmente. los socket de datagramas no garantizan que el flujo de datos llegue en secuencia, sin pérdidas ni duplicados.

Al crear un socket, no se hace referencia a ninguna máquina o aplicación, sólo se convierte en un extremo potencial para la comunicación. El socket es útil cuando se enlaza a una dirección internet y a un número de puerto.

Consideraciones en el Diseño de Clientes

Las aplicaciones cliente, conceptualmente son más simples que los servidores por varias razones. Primero, muchos clientes no manejan explícitamente concurrencia con servidores múltiples. Segundo, el software del cliente se ejecuta normalmente en un programa convencional. Tercero, el software de cliente generalmente no necesita protección.

Para diseñar un cliente es necesario crear un punto extremo de comunicación (socket), comunicarse al puerto apropiado del servidor, enviar y recibir datos a/desde el servidor y finalmente cerrar el socket.

Consideraciones en el Diseño de Servidores

Conceptualmente, cada servidor sigue un algoritmo simple: crea un socket, lo enlaza a un puerto conocido sobre el cual se desea recibir la información. Posteriormente entra en un ciclo infinito en el cual se aceptan las solicitudes que arriban del cliente, procesa las solicitudes, formula las respuestas y las envía.

Dos de las características más importantes de los servidores que determinan su arquitectura son el número de solicitudes que éste procesa a la vez y el tipo de servicio de comunicación que se usa para el intercambio de información entre el cliente y el servidor.

De acuerdo con el número de solicitudes procesadas simultáneamente los servidores pueden clasificarse en:

Servidores concurrentes:

Servidores que procesan múltiples solicitudes a la vez. La razón principal para introducir concurrencia dentro de un servidor aparece por la necesidad de proporcionar un tiempo de respuesta más corto para múltiples clientes.

La concurrencia puede aplicarse al calcular respuestas para aprovechar mejor el uso del procesador. Mientras el procesador trabaja en estos cálculos, los dispositivos de entrada/salida pueden estar transfiriendo datos dentro de la memoria para otras respuestas. La concurrencia puede permitir que un solo procesador maneje respuestas que solamente requieren un monto pequeño de procesamiento sin necesidad de esperar por solicitudes que tomen grandes tiempos. Teniendo múltiples procesadores, las tareas se reparten entre ellos.

Estos servidores son más difíciles de diseñar pero podrían ofrecer un mejor desempeño.

Servidores iterativos:

Los servidores Iterativos son suficientes para aplicaciones triviales, ya que manejan una solicitud a la vez. Si otra solicitud llega mientras el servidor esta ocupado manejando una solicitud existente, el sistema pone en una cola la nueva solicitud. Una vez que el servidor finaliza el procesamiento de la solicitud, éste observa la cola para ver si tiene una nueva solicitud. Si el servidor no puede manejar solicitudes a una velocidad apropiada, su cola de espera de solicitudes eventualmente se sobrecargará. Para evitar el sobre flujo en servidores que puedan tener grandes tiempos de procesamiento de solicitudes, el diseñador debe considerar implantaciones concurrentes.

Un servidor iterativo es el más fácil de diseñar, programar depurar y modificar. Así, muchos programadores lo eligen cuando la ejecución proporciona una respuesta rápida. Usualmente los servidores iterativos trabajan mejor con servicios simples con acceso por medio de protocolos sin conexión.

De acuerdo con el servicio de comunicación cliente-servidor los servidores pueden clasificarse en

Servidores orientados a conexión:

Los servidores orientados a conexión emplean el protocolo TCP por lo que manejan la pérdida de paquetes y los problemas de entrega fuera de orden de manera automática. Mientras una conexión permanece abierta, TCP proporciona toda la confiabilidad necesaria, retransmite datos perdidos, verifica que los datos lleguen sin errores de transmisión y garantiza la secuencia de la información.

Una de las desventajas de los servidores orientados a conexión es que requieren un socket diferente por cada conexión, mientras que en un diseño sin conexión se permite la comunicación de múltiples host desde un solo socket. La desventaja más importante consiste en que TCP puede mantener conexiones ociosas. Por ejemplo, si suponemos que un cliente establece una conexión a un servidor, intercambia una solicitud y una respuesta y entonces se interrumpe la conexión. El cliente no enviará más solicitudes. El servidor a su vez, ya respondió a la solicitud, por lo que no enviará más datos al cliente. Este problema puede ocasionar que se agoten los recursos dado que el servidor tiene una estructura de datos asignada a la conexión, la cual no puede liberarse.

Servidores sin conexión

Los Servidores sin conexión emplean el protocolo UDP por lo que deben tomar la responsabilidad de entregar la información de manera confiable. Generalmente los clientes tienen la responsabilidad de retransmitir las solicitudes si no llega una respuesta, por lo tanto un servidor debe establecer una estrategia de retransmisión tan compleja como la que usa TCP.

Para determinar el tipo de servidor que se implementará, debemos considerar si los servicios ofrecerán comunicación tipo difusión ("broadcast") o tipo "multicast". El protocolo de transporte TCP ofrece comunicación punto a punto, por lo que no puede

suministrar ni comunicación broadcast ni multicast. Tales servicios requieren de UDP. Así, cualquier servidor que acepta o responda a comunicación multicast debe usar el protocolo sin conexión.

Tipos Básicos de Servidores

Generalizando, tenemos cuatro tipos básicos de servidores [Comer93].

- Servidores Iterativos sin conexión
- Servidores iterativos orientados a conexión
- Servidores concurrentes sin conexión
- Servidores concurrentes orientados a conexión

Servidores iterativo sin conexión:

Para este servidor se crea un socket, se enlaza a una dirección de puerto conocido para ofrecer el servicio, lee repetidamente las solicitudes del cliente , formula una respuesta y la envía al cliente de acuerdo al protocolo de aplicación.

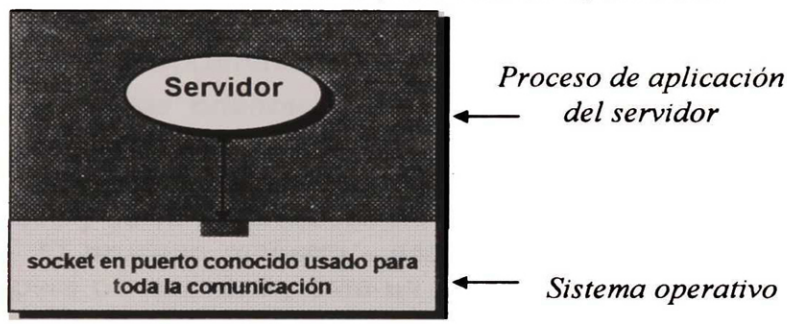


Figura 2.1 Servidor iterativo, sin conexión. Un solo proceso de servidor se comunica con varios clientes usando un socket.

Servidores iterativo orientado a conexión.

Un servidor de este tipo crea un socket lo enlaza a una dirección de puerto conocido para ofrecer el servicio. Al recibir una solicitud de conexión, la acepta y crea un nuevo socket para que atienda al cliente, recibe las solicitudes del cliente y le envía la respuesta de acuerdo al protocolo de aplicación. Cuando finalice el cliente, se cierra ese socket y espera otra solicitud para aceptar una nueva conexión.

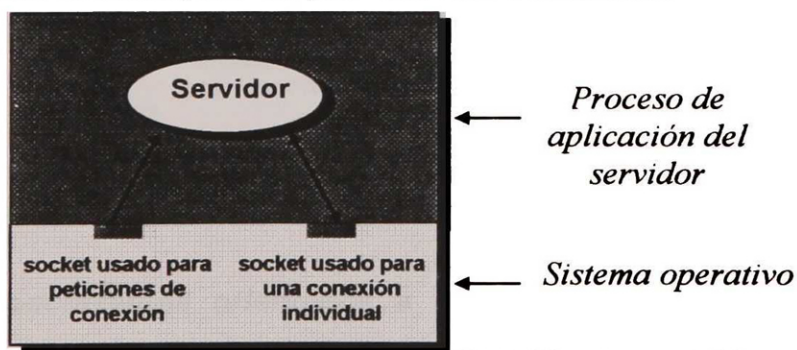


Figura 2.2. Servidor iterativo, orientado a conexión. El servidor espera peticiones de conexión por el puerto predeterminado y entonces se comunica con el cliente usando esa conexión.

Servidor concurrente sin conexión

El servidor Maestro crea un socket y se enlaza a una dirección de puerto conocido para ofrecer el servicio. Deja el socket desconectado. Llama repetidamente a la rutina apropiada para recibir la solicitud del cliente y crea un nuevo proceso esclavo para manejar la respuesta. El esclavo recibe una solicitud específica sobre la creación así como acceso al socket. Forma una respuesta de acuerdo al protocolo de aplicación y lo envía al cliente. Al terminar de atender al cliente acaba su función.

Servidor concurrente orientado a conexión

Los protocolos de aplicación concurrente usan una conexión como el paradigma básico para comunicación. Ello le permite al cliente establecer una conexión con el servidor, comunicarse sobre la conexión y entonces descartarla. En muchos casos, la conexión entre cliente y servidor maneja más de una solicitud: el protocolo permite al cliente enviar solicitudes repetidamente y recibir respuestas sin terminar la conexión o crear una nueva, así los servidores orientados a conexión implementan concurrencias entre conexiones más que entre solicitudes individuales.

Aunque es posible que un servidor lleve a cabo cierta concurrencia utilizando un solo proceso, muchos servidores concurrentes usan múltiples procesos.

•Servidores concurrentes orientados a conexión (implantación con múltiples procesos):

Para la implantación de éste tipo de servidores se requiere de un proceso servidor maestro y de varios procesos servidores - esclavos: como se muestra en la figura 2.3. ***El proceso maestro - esclavo*** abre un socket sobre un puerto conocido, espera por solicitud y crea un *proceso servidor esclavo* para manejar cada solicitud. El servidor maestro nunca se comunica directamente con el cliente, pasa la responsabilidad al esclavo. Cada proceso esclavo maneja comunicación con otro cliente. Después el esclavo forma una respuesta y la envía al cliente si éste existe

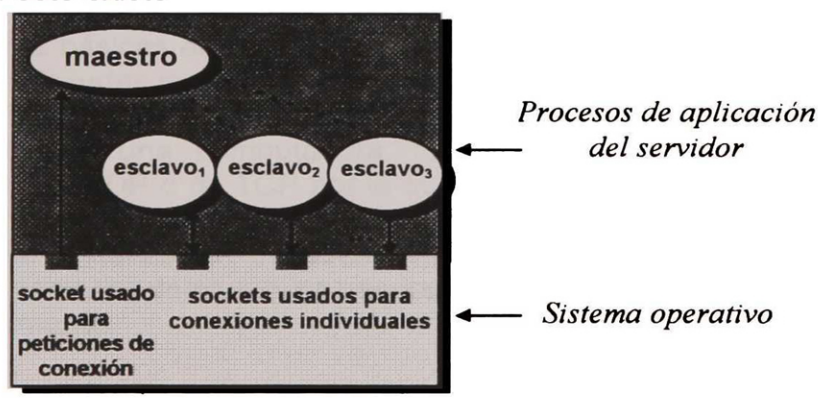


Figura 2.3 Estructura de procesos de un servidor concurrente orientado a conexión. Un proceso servidor maestro acepta cada conexión entrante y crea un proceso esclavo para manejarla.

•Servidores concurrentes orientados a conexión (implantación con un solo proceso):

En este caso, un único proceso en el servidor mantiene varias conexiones TCP abiertas para comunicación con varios clientes proporcionando una concurrencia aparente. Mientras espera datos de los clientes, el proceso se

mantiene bloqueado. Cuando llega un dato por cualquier conexión, el proceso despierta, atiende la petición y envía una respuesta para volver a un estado de espera. Mientras el CPU sea lo suficientemente rápido para atender la carga presentada al servidor, la versión de un solo proceso maneja las peticiones tan bien como una versión con varios procesos. Incluso es posible que una versión de un solo proceso maneje más rápidamente cierta carga que una implantación con varios procesos, debido a que la primera requiere menor conmutación entre diferentes procesos.

La figura 2.4. ilustra la estructura y sockets de un servidor concurrente con un proceso único.

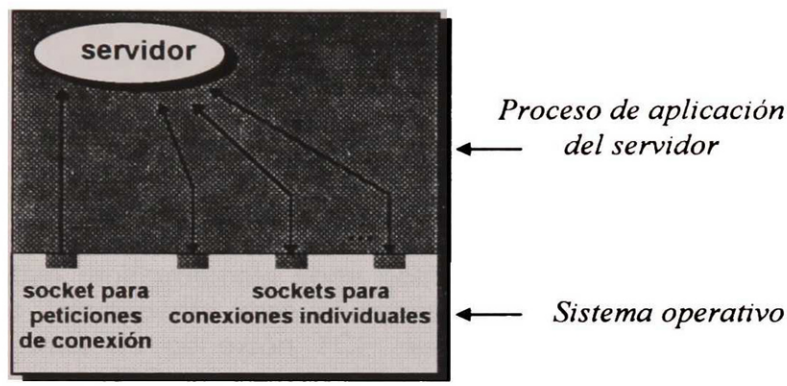


Figura 2.4. La estructura de procesos de un servidor orientado a conexión que proporciona concurrencia con un solo proceso. El proceso maneja sockets múltiples.

El único proceso debe realizar las tareas maestro y esclavo. Mantiene un conjunto de sockets en donde uno relacionado con un puerto conocido sirve al maestro para aceptar las peticiones de conexión. Los demás sockets corresponden a conexiones por medio de los cuales los esclavos atienden a las peticiones.

Servidores Multiprotocolo (TCP/UDP)

En los servidores básicos descritos anteriormente, se tiene de manera independiente un servidor para solicitudes que llegan a través de UDP y otro para solicitudes que llegan a través de TCP. El tener servidores separados permite ejercer control sobre los protocolos que ofrece una computadora. Sin embargo, muchos servicios pueden solicitarse a través de UDP o de TCP por lo que cada servicio requeriría dos servidores. Además la mayor parte del código de ambos servidores puede ser el mismo, por lo que la depuración y la actualización de estos implica modificaciones y administración de ambos. Por otra parte, el consumo de recursos (como las localidades de la tabla de procesos) se incrementa.

Servidores multiprotocolo Iterativo

Un servidor multiprotocolo consiste de un proceso único que usa operaciones asíncronas de E/S (entrada/salida) para manejar la comunicación sobre UDP ó TCP. El servidor inicialmente abre dos sockets: uno para transporte sin conexión (UDP) y el otro que usa transporte orientado a conexión (TCP). Por ejemplo, cuando un cliente realiza una petición de conexión TCP, el socket TCP correspondiente se encuentra listo. Realizará las operaciones para aceptar una nueva conexión que se empleará para atender al cliente; si el cliente envía una petición en un datagrama UDP, el socket UDP

está listo, el servidor lee la solicitud y registra la dirección del emisor. Después de formular la respuesta, el servidor la envía al cliente.

La figura 2.5 ilustra la estructura del proceso de ***un servidor multiprotocolo iterativo.***

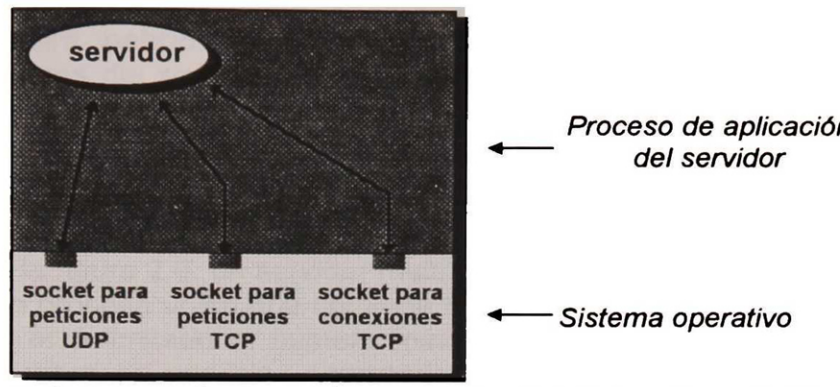


Figura 2.5. La estructura de procesos de un servidor multiprotocolo iterativo. En cualquier momento, el servidor tiene como máximo tres sockets abiertos: uno para peticiones UDP, otro para peticiones de conexión TCP y uno temporal para una conexión TCP individual.

El servidor iterativo multiprotocolo tiene como máximo tres sockets abiertos en un tiempo determinado. Inicialmente abre un socket para aceptar datagramas UDP y otro para aceptar peticiones de conexión TCP, espera las peticiones. Cuando llega un datagrama en el socket UDP el servidor determina la respuesta y la regresa al cliente usando el mismo socket. Cuando llega una petición de conexión por el socket TCP, el servidor la acepta y obtiene una nueva conexión, esto crea un nuevo socket que permanece abierto durante el tiempo que dura la comunicación con el cliente.

Un servidor multiprotocolo permite al diseñador 1) crear un procedimiento individual que responda a las peticiones de un servicio dado y 2) llamar a ese procedimiento sin importar el transporte usado en las peticiones (UDP ó TCP). En otras palabras, un servidor multiprotocolo permite encapsular todo el código destinado a un servicio particular en un programa único, eliminar duplicidad y facilitar los cambios en el mismo, además de optimar el uso de recursos del sistema (como los procesos).

El mantener el código compartido en un lugar único facilita su mantenimiento y garantiza que el servicio ofrecido por los dos protocolos de transporte es el mismo.

Servidores multiprotocolo concurrentes

Los servidores concurrentes multiprotocolo pueden usar un método iterativo para manejar las peticiones siempre que el procesamiento que se requiera para cada respuesta sea pequeño.

En el caso más simple, el servidor multiprotocolo puede crear un nuevo proceso para manejar cada conexión TCP de forma concurrente, mientras las peticiones UDP son atendidas de forma iterativa. En el diseño multiprotocolo también se puede implantar un proceso único que proporcione concurrencia aparente entre las peticiones que usan conexiones TCP o UDP.

Servidores Multiservicio (TCP, UDP)

En muchos casos los programadores diseñan un servidor para cada servicio, por ejemplo uno para pruebas en red, otro para depuración de programas en red, otro para mantenimiento de la red, etc., esto implica soportar varios procesos en una máquina que posiblemente no recibirá peticiones de todos esos servicios, con la consiguiente ocupación de los recursos del sistema. Esta situación ha motivado la consolidación de servidores múltiples dentro de un único servidor multiservicio, con el objetivo de reducir el número procesos en ejecución.

Servidores multiservicio sin conexión

Como se ilustra en la figura 2.6, un servidor multiservicio iterativo y sin conexión por lo general consiste de un proceso único que contiene el código necesario para los servicios que ofrece (en UNIX está limitado el número máximo de sockets abiertos por lo que también lo está el número de servicios soportados). El servidor abre un conjunto de sockets y relaciona cada socket con un puerto conocido para cada servicio que se ofrece. Usa una pequeña tabla para relacionar sockets con servicios. Para cada descriptor de socket existe un registro de la dirección del procedimiento asociado al servicio. Cuando llega el datagrama, el servidor llama el procedimiento apropiado (usando la tabla para relacionar el descriptor del socket con el procedimiento que soporta el servicio) para formular la respuesta y enviarla.

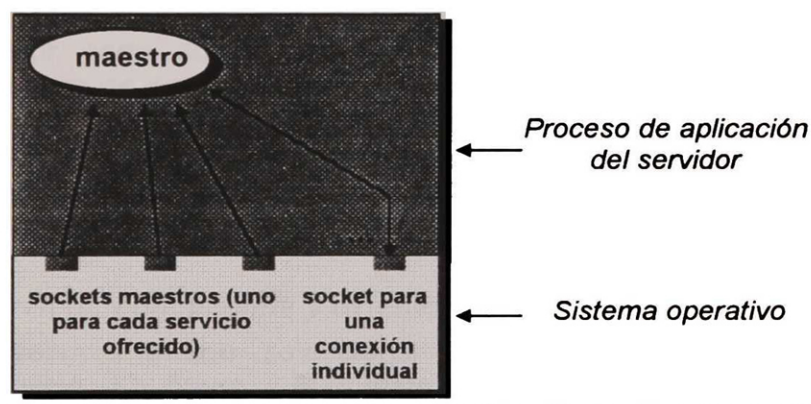


Figura 2.6. Un servidor multiservicio, iterativo y sin conexión. El servidor espera un datagrama de cualquier socket, donde cada socket corresponde a un servicio individual.

Servidores multiservicio orientado a conexión

Un servidor multiservicio orientado a conexión puede seguir un algoritmo iterativo. En principio, este servidor debe realizar las mismas tareas que haría un conjunto de servidores iterativos y orientados a conexión. Es decir, el proceso individual en un servidor multiservicio reemplaza el servidor maestro del conjunto de servidores orientados a conexión. El servidor multiservicio usa transacciones de E/S para realizar sus labores. La figura 2.7 se muestra la estructura de procesos de un servidor como éste.

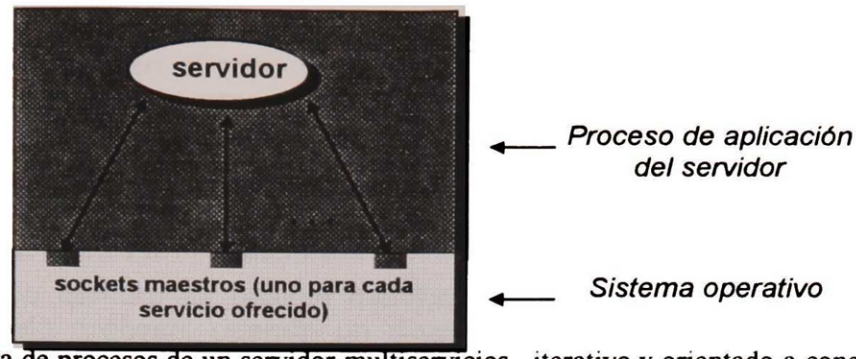


Figura 2.7. La estructura de procesos de un servidor multiservicios , iterativo y orientado a conexión. En cualquier momento, el servidor tiene un socket abierto para cada servicio y hasta un socket adicional para manejar una conexión particular.

Cuando se inicia el servidor multiservicio crea un socket por cada servicio que ofrece, relaciona cada socket a un puerto conocido y usa un comando (`select`) para esperar la llegada de alguna petición por cualquier socket. Al llegar una petición de servicio (socket listo), el servidor acepta la conexión (`accept`) y crea un nuevo socket que se usa para comunicación cliente-servidor. Este nuevo socket se cierra cuando termina la comunicación. Entonces, además de un socket maestro para cada servicio, el servidor tiene como máximo otro socket abierto.

Al igual que en el caso de servicios sin conexión, el servidor mantiene una tabla para decidir cómo manejar cada conexión que llega. Al inicio, el servidor acomoda en la tabla sockets maestros. Por cada socket maestro, el servidor agrega en la tabla la especificación del número de socket y el procedimiento que soporta el servicio ofrecido por tal socket. Después de tener todos los sockets maestros, el servidor espera por peticiones de conexión (`select`). Una vez que llega una conexión, el servidor usa la tabla para decidir cuál de los procedimientos internos se llamará para manejar el servicio que pidió el cliente.

Un servidor multiservicio, concurrente y orientado a conexión

El servidor multiservicio puede seleccionar el manejo iterativo o concurrente dependiendo del servicio. La figura 2.8 muestra la estructura de procesos para un servidor multiservicio, concurrente y orientado a conexión.

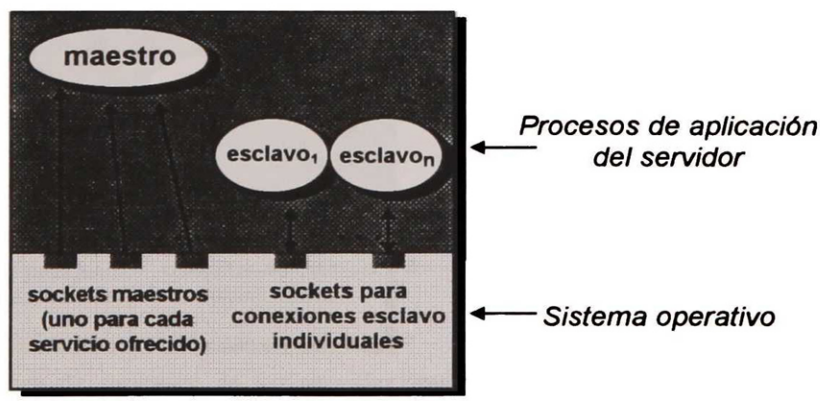


Figura 2.8. La estructura de procesos para un servidor multiservicio, concurrente, orientado a conexión. El proceso maestro maneja las peticiones de conexión, mientras que un proceso esclavo maneja cada conexión.

En una implantación iterativa, una vez que concluye la comunicación cliente-servidor se cierra la nueva conexión. En el caso concurrente, el proceso maestro del servidor cierra la conexión del esclavo; este proceso se comunica con el cliente (recibe peticiones y envía respuestas) usando una nueva conexión que permanece abierta hasta que termina la iteración con el cliente.

Un servidor multiservicio con un proceso único

Aún cuando es poco común, es posible manejar toda la actividad en un servidor multiservicio con un solo proceso, usando un diseño similar al servidor con proceso único. En lugar de crear un proceso esclavo para cada conexión, el proceso único añade el socket para cada nueva conexión. Si algún socket maestro se encuentra listo, el proceso único acepta una conexión, si uno de los sockets esclavos está listo, el proceso obtiene la petición del cliente, arma y envía la respuesta al cliente.

En un servidor multiservicio, cualquier cambio en un servicio implica la compilación del código fuente del servidor, la terminación de los procesos del servidor y el inicio del servidor usando el nuevo código.

Si el servidor multiservicio ofrece gran variedad de servicios es posible que alguno de ellos lo utilice un cliente en el momento de la terminación de los procesos del servidor, lo cual causaría problemas a los clientes. Además entre mayor sea el número de servicios ofrecidos, aumenta la probabilidad de actualización en alguno de ellos.

Los diseñadores optan por dividir el servidor multiservicio en componentes independientes usando programas compilados de forma separada para manejar cada servicio.

Consideremos el servidor multiservicio de la figura 2.9, el servidor maestro espera la petición de conexión de un conjunto de sockets maestros. Una vez que llega una petición de conexión, el proceso maestro crea un proceso esclavo que manejará la conexión. El servidor debe contener el código que soporta todos los servicios. La figura 2.9 ilustra una forma de modificar el diseño para obtener piezas separadas.

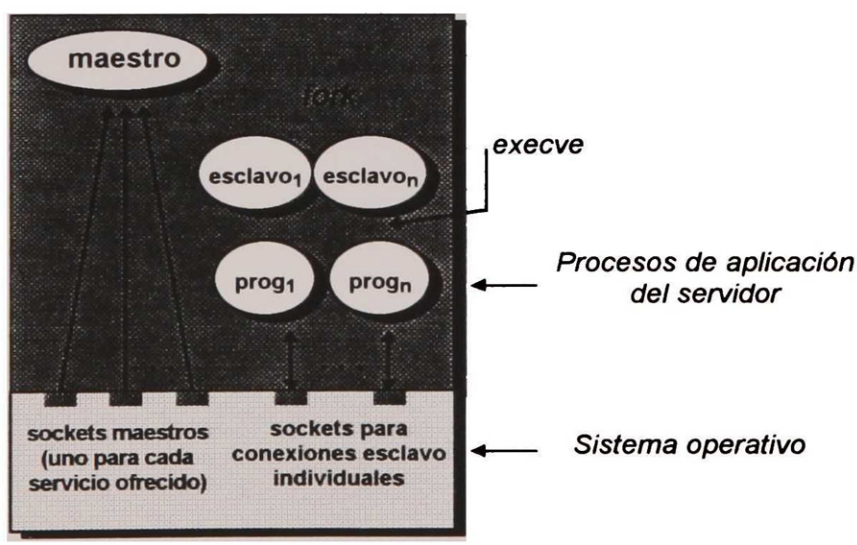


Figura 2.9 La estructura de procesos de un servidor multiservicio orientado a conexión que usa el comando *execve* para ejecutar un programa separado por cada conexión.

El servidor maestro crea nuevos procesos que manejan cada conexión. Sin embargo, el proceso esclavo reemplaza el código original por un programa nuevo que maneja toda la comunicación con el cliente (llama a `execve`).

Debido a que el esclavo llama a un programa nuevo desde un archivo, este diseño permite al administrador del sistema reemplazar el archivo sin tener que compilar nuevamente el código del servidor multiservicio, terminar la ejecución de los procesos del servidor, ni iniciar la ejecución del servidor.

En un servidor multiservicio, la llamada a `execve` de UNIX hace que se separe el código que maneja un servicio individual, del código que maneja las peticiones individuales de los clientes.

Servidores Multiservicio Multiprotocolo

El diseño multiprotocolo permite a un servidor manejar sockets UDP ó TCP para un mismo servicio. En el caso de un servidor multiservicio, el servidor puede manejar sockets UDP ó TCP para algunos o todos los servicios que ofrece.

Es común el uso del término *super servidor* para referirse al servidor multiservicio, multiprotocolo. En estos servidores, inicialmente se abren uno o dos sockets maestros por cada servicio que se ofrece. Los sockets maestros de un servicio determinado corresponden a un transporte sin conexión u orientado a conexión. El servidor espera solicitudes en cualquier socket. Si un socket UDP recibe una petición, el servidor llama a un procedimiento para atenderla en el socket, formula una repuesta y la envía. Si llega una petición a un socket TCP, el servidor llama un procedimiento que acepta la siguiente conexión del socket y la maneja. El servidor puede manejar la conexión directamente, de forma iterativa o puede crear un nuevo proceso para manejar la conexión en modo concurrente.

Apéndice

C

Compilador del Protocolo RPC Para la programación de aplicaciones distribuidas

PROTOCOLO DE MENSAJES RPC

LENGUAJE RPC	LENGUAJE C

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

Las **uniones** del lenguaje RPC, tienen una representación mas detallada en el lenguaje C. Al igual que las estructuras en el lenguaje RPC, no es necesario el typedef	

union lee_resultado switch (int numerr){
case 0:
opaque datos [1024];
default:
void;
};
.
	struct lee_resultado {
	int numerr;
	union {
	char datos [1024];
	} lee_resultado_u;
	};
	typedef struct lee_result lee_result;

La definición del **programa** que se ejecutará de manera remota, se representa como:	

program PROGTIEMPO {
version VERTIEMPO{
unsigned int OBTIEMP(void)= 1;
void INITIEMP (unsigned int)=2;
}=1;
}= 44;
	#define PROGTIEMPO 44
	#define VERTIEMPO 1
	#define OBTIEMP 1
	#define INITIEMP 2

Casos Especiales	
bool casado;	bool_t casado;
string nombre<32>;	char *nombre;
string lognombre<>;	char *longnombre;
opaque bloquesdsk [512];	char bloquesdsk [512];
opaque datosarch<1024>;	struct { u_int datosarch_len; char * datosarch_val; }datosarch;
void	void

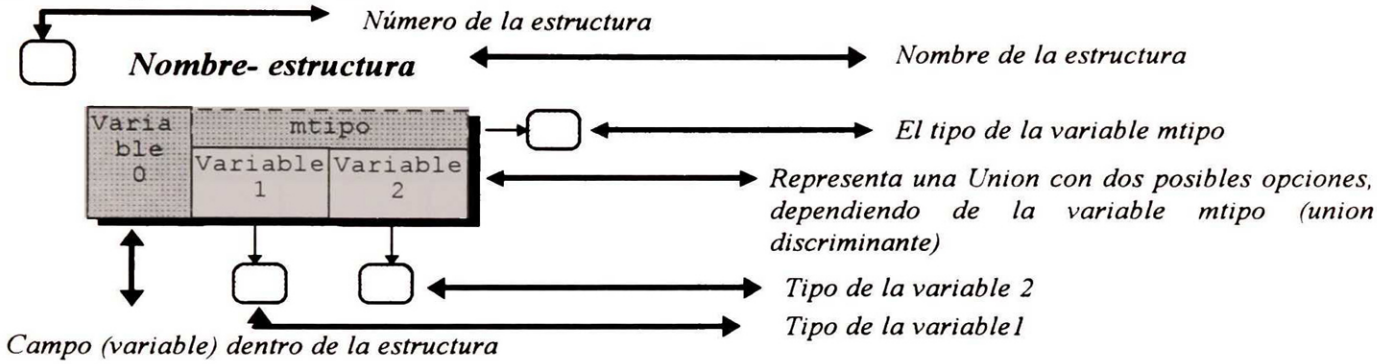
Tabla 4.1. *Comparaciones entre los lenguajes RPC y C*

Protocolo de Mensajes RPC

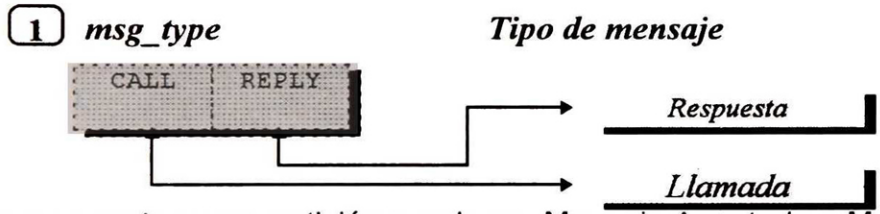
Para cumplir con los requerimientos del protocolo RPC, SUN diseñó una serie de reglas para la transmisión de los mensajes, la autenticidad, el manejo del lenguaje y las rutinas para la administración de procesos (port map). A continuación se especifica el protocolo de mensajes usado en la implantación del compilador RPC.

Mensajes RPC

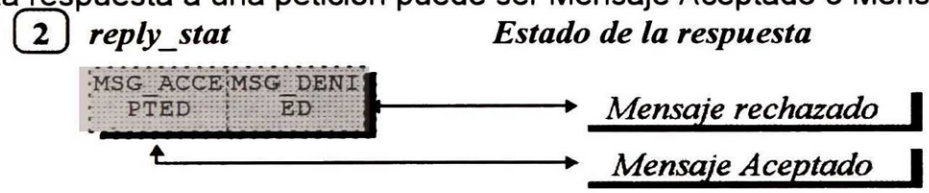
Para la representación de manera gráfica de las estructuras de los mensajes RPC se usará la siguiente notación



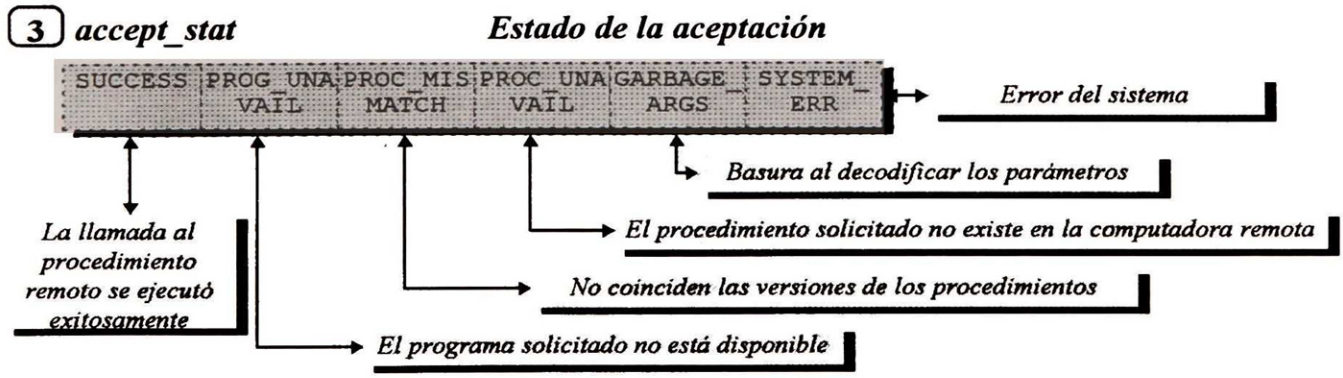
Los tipos de mensajes que se pueden enviar son llamada o respuesta



La respuesta a una petición puede ser Mensaje Aceptado o Mensaje Rechazado.

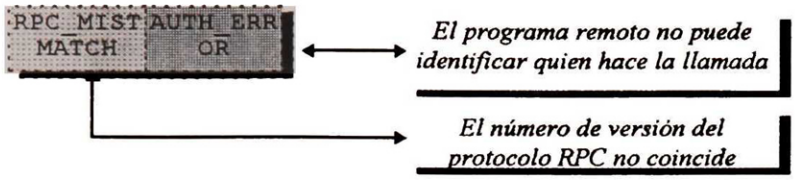


Cuando se acepta un mensaje, se regresa la siguiente estructura con los estados indicados:



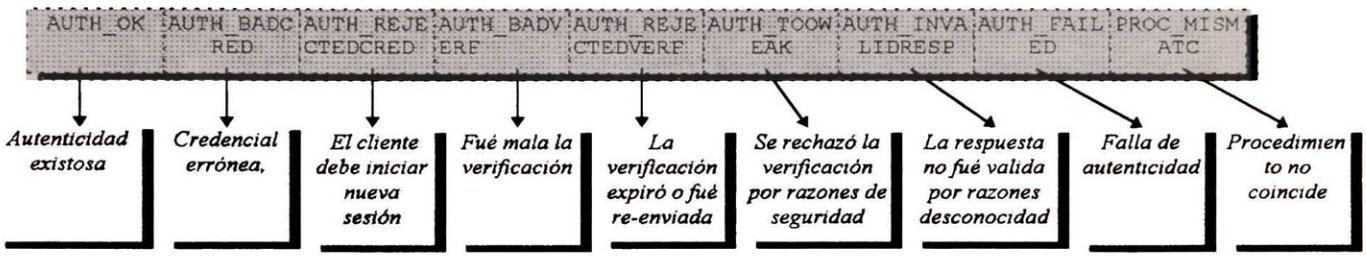
Un mensaje puede rechazarse por las siguientes razones:

4 ***reject_stat*** ***Estado de rechazo***



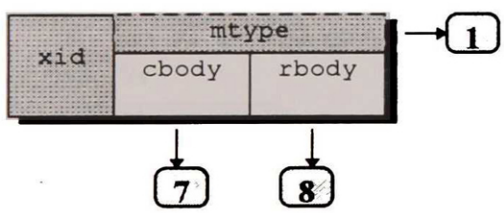
Para representar el resultado de un proceso de verificación de autenticidad del procedimiento remoto se tienen los siguientes estados:

5 ***auth_stat***



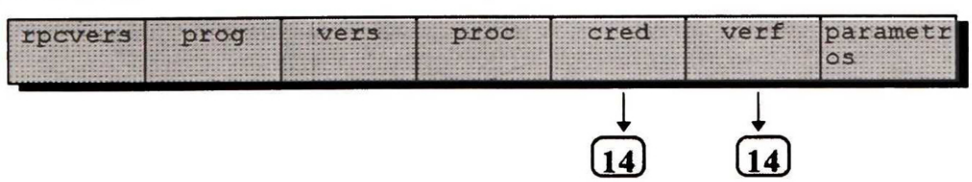
Todos los mensajes RPC inician con un identificador de la transacción (XID), seguido por una unión discriminante, representada en formato XDR como se describe en el apéndice A. El xid del mensaje respuesta (REPLY) debe coincidir con el mensaje de solicitud (CALL).

6 ***rpc_msg*** ***Mensaje RPC***



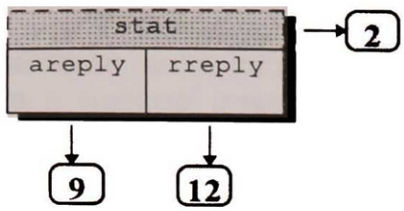
El cuerpo de una llamada RPC debe contener la versión del protocolo RPC que se maneje (en este caso la versión 2), el programa, la versión y el procedimiento remoto que se ejecutará y dos parámetros para autenticar: Cred (validación por credenciales) y verf (verificador de la autenticidad). En el resto del cuerpo deberán colocarse los parámetros para el procedimiento remoto :

7 ***call_body*** ***Cuerpo de la llamada***



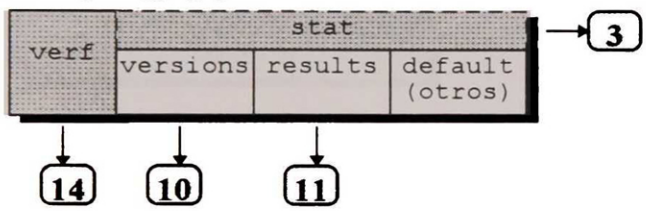
El cuerpo de la respuesta puede ser *Mensaje aceptado* o *Mensaje Rechazado*

8 ***reply_body*** ***Cuerpo de la respuesta***

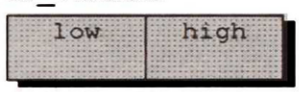


- La respuesta a una llamada RPC que regresa un mensaje aceptado deben Verificar la autenticidad que genera el servidor para validarse así mismo con el cliente. Por otro lado, la respuesta tiene un *unión discriminante* en la que se analiza un código de estado de la llamada. Si la llamada fue exitosa se regresa el resultado. Si el programa no coincidió (PROG_MISTMATCH), se regresan las versiones registradas. Para otro tipo de errores, se asigna void.

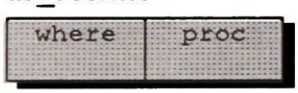
9 ***accepted_reply*** ***Respuesta aceptada***



10 ***ar_version*** ***Versiones***



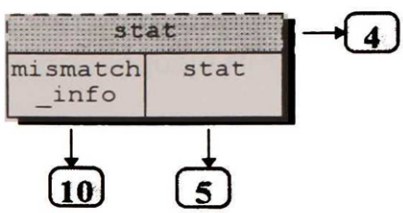
11 ***ar_results*** ***Resultados***



El rechazo de una llamada RPC puede deberse a :

- El servidor no está corriendo una versión compatible del protocolo RPC (RPC_MISMATCH). En éste caso el servidor regresa los números de versiones del protocolo RPC que soporta.
- El servidor rechaza la identificación de cliente (AUTH_ERROR). En éste caso el servidor regresa el motivo de la falla.

12 ***rejected_reply*** ***Respuesta rechazada***



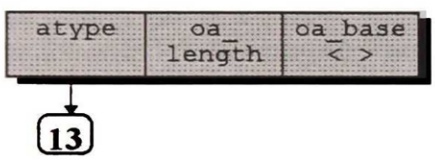
Protocolo de Autenticación

Una Llamada a procedimiento remoto define varias formas de identificación, incluyendo un esquema de identificación simple como lo es el de Unix o uno más complicado como el que usa el Estándar de encritado de datos (*Data Encryptation Standar [DES])* , Originalmente publicado por National Bureau Of Standards (NBS ha cambiado su nombre a National Institut For Standards and Technology [NIST]). La información de autenticidad puede tener uno de los siguientes tipos:

13 ***auth_type*** ***Tipo de autenticación***

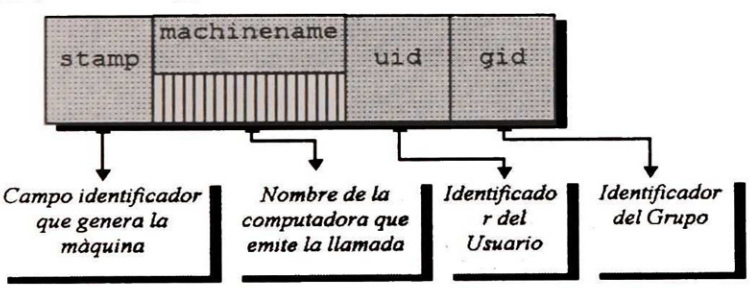
Para declarar el tipo de autenticidad en el formato de los mensajes RPC, se emplea la estructura opaque_auth:

14 ***opaque_auth***



Cada método de autenticidad usa un formato específico para codificar los datos. Por ejemplo, muchos sistemas que envían mensajes RPC, emplean el Sistema Operativo UNIX. La autenticidad en UNIX define la estructura siguiente:

15 ***auth_unix***



Lenguaje RPC.

El lenguaje RPC es una extensión del lenguaje XDR, en la que se agregan las declaraciones de "program", "procedure", y "version". A continuación se describe la sintaxis del lenguaje RPC y un ejemplo.

program_def :

"*program*" identificador "*{*"
version_def
version_def

"}" "=" constante ";"

version_def:

"version" identificador "{"
procedure_def
procedure_def
. .

"}" "=" constante ";"

procedure_def:

especificador_tipo identificador "(" especificador de tipo [",", especificador de tipo]... ")" "=" constante ";"

1. En esta especificación las palabras reservadas son "program" y "version".
2. El nombre y el número de la versión no puden repetirse dentro del espacio de una definición de programa.
3. El nombre y el número de procedimiento no pueden repetirse dentro del espacio de una definición de versión.
4. Los identificadores de programas están en el mismo espacio que las constantes y los tipos de identificadores.
5. Solamente pueden asignarse constantes sin signo a los programas, versiones y procedimientos.

Un ejemplo de un programa en el lenguaje RPC seria:

```
program FECHA_PROG{
        version FECH_VER{
                string proc1_fecha(void)=1;
                int mes(string)=2;
        }=1;
        version FECH_VER2{
                string proc1_fecha(void)=1;
        }=2;
}=100;
```

Protocolo del programa PORT MAPPER

Los protocolos de transporte UDP y TCP emplean números de puertos de 16 bits para identificar los extremos de comunicación. SUN emplea números de 32 bits para identificar un programa remoto. Por esto es imposible identificar a los programas RPC con los números de los puertos.
El cliente se enfrenta al problema de identificar a qué puerto remoto se tiene que conectar para ejecutar el procedimientos deseado. Esta identificación del puerto debe ser dinámica, ya que el servidor puede asignar números diferentes cada vez.

Para permitir a los clientes, contactar cualquier programa remoto, SUN ofrece un mecanismo de llamada a una entidad que se conoce como port mapper y que se localiza en el servidor. Para esto en el lado del servidor se mantiene una pequeña base de datos donde se registra cada servicio RPC indicando el número de programa , número de puerto y el protocolo que maneja. De ésta manera el cliente contacta al port mapper y le solicita el puerto del programa remoto deseado. En la tabla 1 se presenta una serie de constantes definida por SUN, que maneja el protocolo de mensajes

CONSTANTES	
PMAPPORT	111
PMAPPROG	100000
PMAPVER	2
PMAPVER_PROTO	2
PMAPVER_ORIG	1
PMAPPROC_NULL	0
PMAPPROC_SET	1
PMAPPROC_UNSET	2
PMAPPROC_GETPORT	3
PMAPPROC_DUMP	4
PMAPPROC_CALLIT	5
IPPROTO_TCP	6
IPPROTO_UDP	17

Tabla 1. Constantes definidas en el protocolo de mensajes

Los procedimientos del Port Mapper son los siguientes:

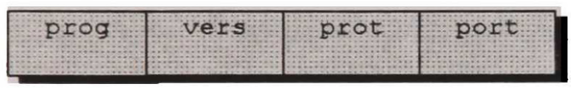
```
program PMAPPROG {
        version PMAPVERS{
                void          PMAPPROC_NULL(void)          =0;
                bool          PMAPPROC_SET (mapping)       =1;
                bool          PMAPPROC_UNSET(mapping)      =2;
                unsigned int  PMAPPROC_GETPORT(mapping)    =3;
                pmaplist      PMAPPROC_DUMP(void)          =4;
                call_result   PMAPPROC_CALLIT(call_args)   =5;
        }=2;
}=1000;
```

El procedimiento PMAPPROC_NULL, es un procedimiento que no ejecuta ninguna acción, ya que algunos programas no envían parámetros.
El procedimiento PMAPPROC_SET se emplea para registrarse con el port mapper. Necesita como parámetros, la estructura *mapping* (programa, versión, protocolo de transporte y numero de puerto en el cual atenderá solicitudes). El procedimiento regresa TRUE, si se registró correctamente y FALSE en caso contrario. El procedimiento rechaza la solicitud de inserción si ya existe los tres datos "prog, vers, prot".

El procedimiento PMAPPROC_UNSET elimina un registro de la base de datos del port mapper.
El procedimiento PMAPPROC_GETPORT regresa el número de puerto en el que se encuentra el número de programa "prog", número de versión "ver" y el número de protocolo de transporte "prot" que recibe como argumento. Si regresa un cero indica que no se encuentra registrado el programa que le indican. El campo port es ignorado en la estructura *mapping*.
El procedimiento PMAPPROC_DUMP regresa una lista de los programas registrados en la base de datos del port mapper.
El procedimiento PMAPPROC_CALLIT. Le permite al cliente llamar a otro procedimiento remoto en la misma máquina sin conocer el número de puerto del protocolo. Este procedimiento solamente envía un reply si el procedimiento se ejecutó exitosamente y si no, se mantiene en "silencio" (no envía un reply). El portmapper se comunica con el programa remoto usando solamente UDP.

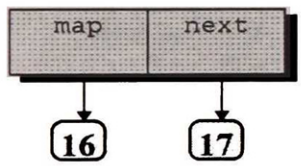
Los mensajes definidos para el protocolo Port Mapper son los siguientes:

(16) ***mapping***



lista de programas :

(17) ***pmaplist***



Argumentos para el procedimiento callit:

(18) ***call_args***

El resultado del procedimiento callit:

(19) ***call_result***

BIBLIOGRAFÍA

- **[Bloo91]** John Bloomer. "*Ticket to Ride. Remote Procedure Calls in a Network Environment*", SunWorld Noviembre 1991. Revista
- **[Bloo92]** John Bloomer, "Power programming with RPC", O'Reilly & Associates, Inc, 1992.
- **[Cano84]** José Canosa, "*El sistema UNIX y sus aplicaciones*", México-Barcelona, Publicaciones Marcombo, S.A., 1984.
- **[cisco93]** , Cisco Systems Inc., "*Internetworking Technology Overview* ", USA 1993.
- **[CN2191]** Computer Network and ISDN systems, "*Communication in the Raid Distributed Database System*", Vol 21 No. 2, Abril 1991.
- **[CN2391]** Computer Network and ISDN systems, "*Remote Procedure Call: a stepping stone towards ODP*", Vol 23, No. 1-3, Noviembre 1991.
- **[comer93]** Douglas E. Comer and David L. Stevens, "*Internetworking with TCP/IP Vol III: Client-Server Programming and Applications*", Pretince Hall, New Jersey, 1993
- **[comer96]** Douglas E. Comer, "*Redes Globales de información con Internet y TCP, principios básicos, protocolos y arquitectura*", México 3ª edición 1996. Prentice-Hall Hiispanoamericana , S.A.
- **[Comm92]** Communications of the ACM, "*Client-Server computing*", Vol 35, No. 7, Julio 1992
- **[Corb91]** John R. Corbin, "*The Art of Distributed Applications. Programming Techniques for RPC* ", Spring-Verlag 1991.
- **[DB0790]** Distributed Computing, "*Retargetable stub generator for a remote procedure call facility*", Vol 13, No. 6, Julio/agosto 1990
- **[Johnson75]** Johnson S.C., "Yacc-yet another compiler compiler", Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, Nueva Jersey. 1975.
- **[Lesk75]** Lesk, M.E., "Lex.- A lexical analyzer generator", Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, Nueva Jersey. 1975.
- **[LEX]** "lex compiler to Personal Computers"
- **[RFC1014]** Sun Microsystems, Inc. , "External Data Representation Standard: Protocol Specification", Junio 1987.
- **[RFC1057]** Sun Microsystems,Inc;"*Remote Procedure Call Protocol Specification Version 2*"; RFC 1057, Junio 1988.
- **[RFC1831]** Sun Microsystems,Inc;"*Remote Procedure Call Protocol Specification Version 2*"; RFC 1831, Junio 1995.
- **[RFC1831]** Sun Microsystems, Inc. , "External Data Representation Standard: Protocol Specification", Junio 1995.
- **[Schild87]** Herbert Schild, "Lenguaje C programación Avanzada", OSBORNE/McGraw Hill 1987

- **[Stal91]** W. Stallings, "*Data and Computer Communications*", Macmillan Publishing Company, Nueva York, 3ª edición, 1991.
- **[Stevens90]** W. Richard Stevens, "Unix Networking Programming", Prentice Hall Inc. Software Series, New Jersey, 1990
- **[SUN90]** Sun Microsystems, Inc. "*Network Programming Guide*", Revision A, 27 de Marzo 1990. Mountain View, Calif.
- **[Tanen91]** Andrew S. Tanenbaum. "*Computer Networks*", Prentice-Hall
- **[Ullman90]** Jeffrey D. Ullman, Alfred V. Aho, Revi Sethi, "Compiladores: principios, técnicas y herramienta",Addison-Wesley Iberoamericana, S.A., 1990.
- **[woll90]** Wollongong , "*WIN/API for DOS, programming guide*", The wollongong Group, Inc, San Antonio Road Palo Alto CA. Release 4.1
- **[Woll91]** Wollongong, "Pathway API programming Guide, Dos Operating System", The wollongong Group, Inc, San Antonio Road Palo Alto CA, Release 1.1, Marzo 1991
- **[Woll92Ac]** Wollongong, "*Pathway Access , for dos & Microsoft Windows Systems* ", The wollongong Group. Palo Alto CA. Release 1.2
- **[Woll92Ru]** Wollongong, ""*Pathway Runtime, for dos & Microsoft Windows Systems* ", release 1,2
- **[Woll94AcD]** Wollongong, "*Pathway Access , for dos & Microsoft Windows Systems. User Guide DOS Systems*", The wollongong Group. Palo Alto CA. Release 3.0
- **[Woll94AcW]** Wollongong, "*Pathway Access , for dos & Microsoft Windows Systems. User Guide Microsoft windows Systems*", The wollongong Group. Palo Alto CA. Release 3.0
- **[Woll94Ru]** Wollongong, ""*Pathway Runtime, for dos & Microsoft Windows Systems User Guide* ", release 2.0
- **[Woll94RuR]** Wollongong, ""*Pathway Runtime, for dos & Microsoft Windows Systems. Reference Pathway Runtime* ", release 2.0
- **[YAC]** "Yac compiler to personal Computers"

Los abajo firmantes, integrantes del jurado para el examen de grado que sustentará la **C. Graciela Judith Esparza Azcoitia**, declaramos que hemos revisado la tesis titulada:

"Compilador del Protocolo RPC para la programación de aplicaciones Distribuidas"

y consideramos que cumple con los requisitos para obtener el grado de Maestra en Ciencias, con especialidad en Ingeniería Eléctrica.

Atentamente

Dr. Héctor Ruíz Barradas _____

Dr. Manuel Mauricio Lara Barrón _____

M. en C. Raúl García Ruíz _____