Sistemas Distribuidos

RPC (*Remote Procedure Call*)

Tabla de Contenidos

Parte I. Introducción a RPC con *rpcgen*

Las distribuciones de Linux suelen incluir la herramienta llamada ***rpcgen*** que permite desarrollar RPCs, facilitando la escritura de aplicaciones distribuidas. Esta herramienta se basa en el ONC RPC (Open Network Computing RPC), desarrollado originalmente por la empresa SUN Microsystems, para el sistema operativo Unix.

Este tipo de RPC, define un modelo de llamadas de procedimientos remotos para aplicaciones distribuidas, haciendo uso del estándar de representación de datos XDR. Esta es una convención "vendor-independent" y la idea es que al comunicarse dos equipos en la red, no existan problemas con la representación de los datos. De esta manera, el tipo de datos "no viaja por la red" *y* la cadena de bits se codifica *y* decodifica de acuerdo con el tipo de datos que se consideren, independientemente del equipo origen y destino. Los datos se decodifican localmente en cada computadora de acuerdo a la representación que ésta tenga/haga de cada uno de ellos.

En este apunte se presentan los fundamentos de RPCs, con los detalles de implementación necesarios, cuando se utiliza `rpcgen`. Para ello, se hará uso de un ejemplo sencillo, viendo además los pasos a tener en cuenta para desarrollar una aplicación distribuida.

1. Un ejemplo con *rpcgen*

Cuando se distribuye una aplicación utilizando RPCs, en general se sigue una secuencia bastante conocida y bien determinada de pasos:

1. Aunque es opcional, se considera apropiado al menos cuando no se tiene experiencia previa construir y probar una aplicación convencional (no distribuida) que resuelva el problema. Luego, se debe *distribuir* el programa anterior, identificando los procedimientos que se van a ejecutar en una máquina remota.
2. Escribir una especificación `rpcgen` para el programa distribuido y compilarla. En general esto implica especificar en el *lenguaje* de RPC la especificación de procedimientos remotos, que son aquellos procedimientos que se llamarán como si fueran locales pero en realidad se resolverán/ejecutarán en una computadora diferente de la que corre la aplicación que hace la llamada,
3. Escribir las rutinas/programas del que ejecuta el RPC (el llamador) y el que contiene las rutinas mismas (el proceso que tiene y ejecuta los procedimientos llamados de manera remota).
4. Compilar y enlazar por una parte el programa cliente y, por otra, el servidor. En el caso de computadoras con la misma arquitectura y sistema operativo se puede generar todo en una sola de ellas y distribuir el binario de la aplicación.
5. Ejecutar el servidor en la máquina remota, y el cliente en la local.

Se verán a continuación cada uno de los pasos anteriores con una aplicación muy sencilla, que en sí misma no representa un problema sino un ejemplo inicial de uso de rpcgen para RPC.

1.1. Construir una Aplicación que Resuelve el Problema

El código que se muestra a continuación, es una re-implementación del conocido "Hola mundo", en el que la función para imprimir el mensaje se ha realizado como un procedimiento. Esto facilitará luego "distribuir" la aplicación, ya que la impresión se realizará en el proceso que corresponda (usualmente denominado *servidor*). La idea básica en este sentido es que todo lo que se quiera resolver en la aplicación se pueda expresar como una o más funciones en C. Estas funciones en C serán luego los *procedimientos remotos* a llamar desde otra/s aplicación/es. En este caso en particular se asume que todo lo que se quiere hacer es imprimir en pantalla "Hola, mundo" y además verificar si tal tarea se pudo llevar a cabo satisfactoriamente o no.

De hecho, en el contexto del desarrollo de sistemas en ambientes distribuidos es relativamente raro comenzar con una aplicación secuencial y no distribuida, pero como se explicó antes, es útil para aprender y desarrollar un poco de experiencia en el uso y programación de/con RPC. Lo más usual en ambientes distribuidos suele ser que las tareas a realizar tanto por los clientes como por los servidores son bastante predeterminadas, o directamente se piensa en resolver un problema

siguiendo el modelo cliente/servidor con lo que el desarrollo de una aplicación no distribuida inicial suele ser un paso innecesario.

Código de "Hola, mundo" con impresión como procedimiento.

```
#include <stdio.h>
#include <stdlib.h>

int print_hola(void)
{
   return printf("Hola, mundo\n");
}

void main(void)
{
   int print_hola(void);

   if (print_hola() > 0)
      printf("Mision cumplida\n");
   else
      printf("Incapaz de mostrar mensaje\n");

   exit(0);
}
```

1.2. Crear una Especificación *rpcgen y C*ompilarla

Una vez definidos los procedimientos que tendrá la estructura de la aplicación distribuida, se prepara la especificación para ***rpcgen.*** Básicamente, en general es la declaración de uno o varios procedimientos (*funciones* en C) remoto/s y de las estructuras de datos que utiliza/n. Este archivo de especificación puede contener:

- Declaraciones de constantes utilizadas por el cliente y/o por el servidor.
- Declaraciones de los tipos de datos utilizados.
- Declaraciones de programas remotos, de los procedimientos que éstos contienen y los tipos de sus parámetros.

La especificación debe estar escrita en el lenguaje de programación de ***rpcgen,*** que si bien es diferente al lenguaje C, presenta cierta similitud. Además, cabe aclarar que RPC utiliza números para nombrar los programas y procedimientos remotos, siendo el lugar para esta identificación el archivo de especificación. Por ello, se verán por separado cada uno de estos aspectos.

1.2.1 Lenguaje de Especificación de Interfaces RPC

El lenguaje RPC es una extensión del lenguaje XDR (eXtended Data Representation) a la que se han añadido los tipos program y version. El RPC consta de un formato de datos que utiliza la representación de datos XDR. Este formato define un lenguaje de descripción de datos y estandariza una sintaxis de transferencia. Dentro del lenguaje de descripción de datos, XDR dispone de una serie de tipos elementales (int, bool, string, etc) y una serie de constructores para declarar tipos de datos más complejos (arreglos, estructuras, uniones). Casi todos estos tipos y constructores tienen su equivalente en los lenguajes de programación más habituales.

Esta convención nace como el XDR(1), estándar de facto de Sun, estableciéndose luego el estándar XDR(2), en donde se incluyeron algunas facilidades, como por ejemplo permitir definir datos compuestos. Este estándar se describe en el RFC 1014 y las extensiones para RPC en el RFC 1050.
Como características generales, podemos citar:

- Cada dato ha de ser múltiplo de 32 bits. Tipos básicos:
    - 32 bits: int, unsigned int, bool, float
    - 64 bits: hyper, unsigned hyper, double
- void: 0 bytes
- Codificación big endian

En la Tabla 1, se ejemplifican especificaciones en el lenguaje RPC y su equivalente en lenguaje C.

Tabla 1: Definiciones en XDR

Especificación	XDR	Lenguaje C
Constantes	Const MAX = 12;	#define MAX 12;
Entero con signo	int a;	int a;
Entero sin signo	unsigned a;	unsigned a;
Valor lógico	bool a;	enum bool_t {TRUE = 1, FALSE=0}; Typedef enum bool_t bool_t; bool_t a;
Coma Flotante	float a;	float a;
Cadenas de bytes	opaque a[20];	char a[20];
Cadenas de bytes de long. variable	opaque<37>; opaque b<>;	struct { int a_len; char *a_val; } a;
Cadena de caracteres	string a<37>; string b<>;	char *a; char *b;
Arreglo de tamaño fijo	int a[12];	int a[12];
Arreglo de tamaño variable	int a<12>; float b<>;	struct { int a_len; int *a_val; } a; struct { int b_len; float *b_val; } b;

Estructuras	struct t {
	int c1;
	string c2<20>;
	};
	t a;
		struct t {
		int c1;
		string *c2;
		};
		typedef struct t t;
		t a;

En RPC, se utiliza el término *program* para hacer referencia a un programa que puede tener uno o más procedimientos a llamar de manera remota. Además, un programa está identificado por un número, es decir, el número de programa identifica a un grupo funcional de procedimientos. Por ejemplo en un sistema de archivos, incluiría procedimientos individuales como "leer" y "escribir". Los procedimientos individuales se identifican con un número de procedimiento único dentro del programa remoto. A medida que el programa remoto evoluciona, a cada versión se le asigna un número de versión. Cada programa puede tener varias versiones. Los programas, versiones y procedimientos se identifican mediante números enteros. De acuerdo con esto, cada procedimiento queda identificado de manera única por la terna <programa, versión, procedimiento>. En la Tabla 2, se muestran los rangos disponibles para el número de programa, dado que la especificación de RPC hecha por Sun incluye en sí misma algunos números de programa o *rangos* de números de programa de manera predeterminada.

Tabla 2: Intervalo para Números de Programas.

Intervalo Hexadecimal	Uso
0x00000000- 0x1FFFFFFF	Definidos por Sun
0x20000000- 0x3FFFFFFF	Definidos por el Usuario
0x40000000- 0xFFFFFFFF	Reservados

El archivo con la especificación tiene extensión ".x". La especificación de una aplicación consiste en la definición de un conjunto de procedimientos dentro de un programa. De manera más formal, la definición de un protocolo de aplicación debe seguir la siguiente sintaxis.

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

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

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

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

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

Para el ejemplo que se ha presentado, todo lo que hay que especificar es un programa que *contenga* la función que hace la impresión de "Hola, mundo" con el lenguaje de RPC. El archivo de especificación, hola.x (que también se incluye en el Anexo I.1), entonces resulta ser:

```
/* hola.x -- archivo de definicion escrito en lenguaje RPC que se
ha de utilizar con rpcgen. Cada procedimiento es a parte de un
programa remoto. Cada procedimiento tiene un nombre y numero. Se
suministra un numero de version para poder generar diferentes
versiones del mismo procedimiento */

program display_prg
{
    version display_ver
    {
        int print_hola (void) = 1;
    } = 1;
} = 0x20000001;
```

1.2.2 Ejecutando rpcgen

La Fig. 1, muestra el esquema de la compilación con rpcgen*.* Básicamente, se generan tres archivos que corresponden al archivo de encabezados ".h" y los *stubs* (*partes de*, *representantes*, *resguardos*, *talones*), tanto del lado del que llama (denominado *cliente* a partir de ahora) como del lado del que es llamado (denominado *servidor* a partir de ahora).

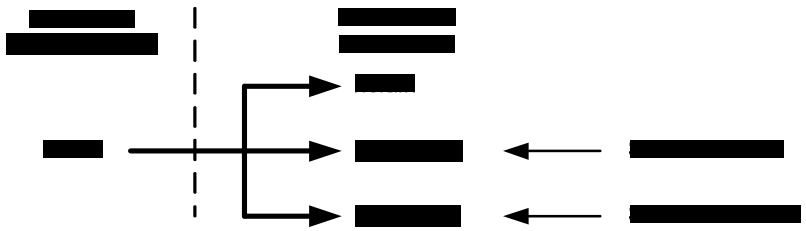

Figura 1: Ejecución del Comando *rpcgen.*

Notar la denominación asignada por rpcgen *a* estos archivos:

- El encabezado, con el mismo nombre que el archivo original, extensión ".h".
- El *stub* del cliente, con el mismo nombre seguido por "_clnt", extensión ".c".
- El *stub* del servidor, con el mismo nombre seguido por "_svc", extensión ".c".

Los stubs no son más que lo necesario para que el cliente pueda llamar al servidor (ejecutar el RPC) y que el servidor pueda *responder*, es decir ejecutar el procedimiento y retornar al menos el control y según el *procedimiento* llamado (función en C), un valor. Se debe notar que estos *stubs* son necesarios porque en ambientes distribuidos no hay forma de conocer la *otra* aplicación de manera explícita. Aunque se intenta mantener la sintaxis y la semántica de un procedimiento, en RPC la aplicación está distribuida y, por lo tanto, no se tiene un único ejecutable que contiene todo (ni siquiera referencias a código/bibliotecas de enlace dinámico).

El archivo hola.h creado por rpcgen, que se muestra en el Anexo I.2, debe incluirse en los archivos stubs del cliente y del servidor. En este archivo están los identificadores especificados para el número de programa y versión como constantes de tipo unsigned long integer. Estas constantes tienen asignadas el valor indicado en el archivo de especificación del protocolo. El nombre del procedimiento se define del mismo tipo. También se observa que hay dos prototipos para la función (o *asociados* a la función) print_hola. El primer prototipo print_hola_1 se utiliza en el stub del cliente. El segundo, print_hola_1_svc, en el stub del servidor. Aquí, la convención utilizada por rpcgen es añadir al nombre del procedimiento el carácter "_" y el número de la versión (1) para el stub del cliente y lo mismo, pero con "_svc", para el servidor.

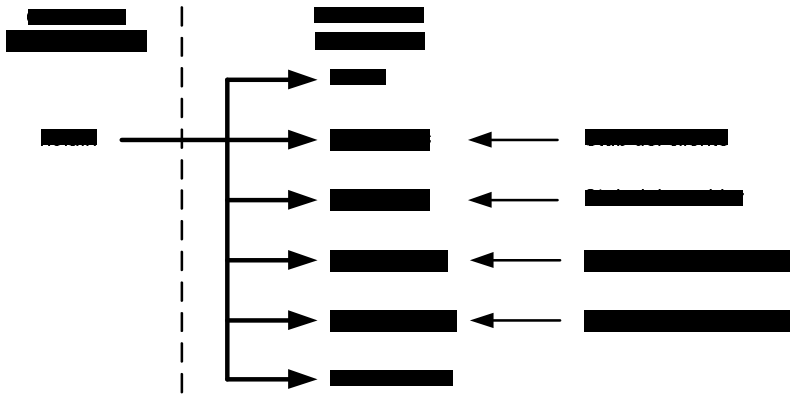

Figura 2: Ejecución del Comando *rpcgen –a.*

Al utilizar la opción –a del rpcgen se genera el código base (como un *esqueleto*) tanto del programa cliente como del servidor "hola_client.c" y "hola_server.c" tal como se muestra en la Fig. 2. El comando rpcgen no podría hacer más que los *esqueletos* de estos procesos/programas dado que no conoce más que la interfase de la función a usar con RPC. A estos *esqueletos* se le deben agregar las correspondientes modificaciones para obtener los programas completamente operativos para el cliente ("hola_client.c") y el servidor ("hola_server.c"). En cierta forma, hola_client.c y hola_server.c son generados *inicialmente* por rpcgen y completados por el usuario/programador.

También en la Fig. 2 se puede notar que rpcgen también genera de manera automática el archivo Makefile.hola, que no es más que el archivo necesario para el comando make para la compilación y enlazado necesarios para generar los binarios de las aplicaciones cliente y servidor.

La Fig. 3 muestra el esquema general de la obtención del código de programa en la máquina cliente y en la servidora. Nótese que el formato de representación de datos XDR, por ejemplo, tiene relación directa y completa solamente con los *stubs*.

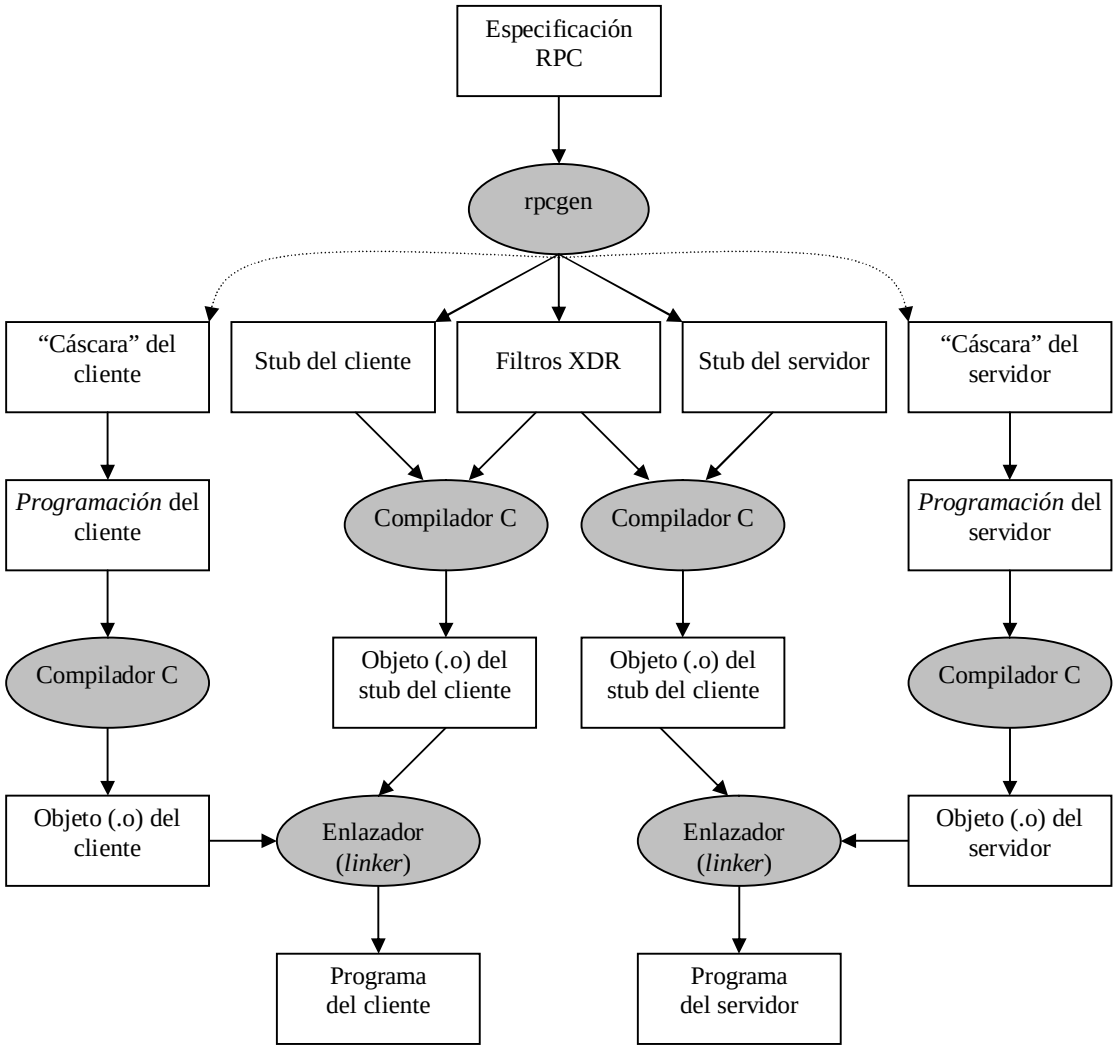

Figura 3: Esquema de Aplicación Distribuida con *rpcgen.*

Esto es consecuencia de la distribución de código (ahora suele estar asociado al término *deploy* de una aplicación distribuida) en diferentes computadoras, donde los datos pueden ser representados de manera diferente en cada una de ellas (desde la cantidad de bytes por cada dato simple hasta la codificación little/big endian). Las aplicaciones *completas* no pueden ser resueltas por rpcgen y tampoco RPC fue definido para esta tarea. RPC se utiliza para resolver el problema de llamada a

procedimientos remotos (no para las aplicaciones completas) y rpcgen no tiene forma de conocer los detalles del *contenido* de cada uno de los procedimientos que se definen, tiene solamente la interfase de cada uno de ellos.

Por otro lado, también se nota la fuerte relación de rpcgen con el lenguaje C, dado que tanto los stubs como los programas *incompletos* generados con la opción –a de rpcgen se expresan en lenguaje C. También en la Fig. 3 se puede notar que tanto el cliente como el servidor *solamente* necesitan los stubs compilados para obtener los ejecutables con el programa ligador (*linker*), sin conocer el resto de los detalles del *otro* programa/ aplicación, solamente se conoce el *protocolo* a nivel de aplicación: básicamente los datos que se intercambian entre ambas aplicaciones. No es necesario conocer el *tipo* o *semántica* de la comunicación porque es, justamente, la de una llamada a procedimiento. Todo lo relacionado con la representación de los datos es manejado de manera *transparente* (para el programador) a través de XDR. Todo lo relacionado con el manejo del control (para obtener/ implementar la *semántica* de RPC) es manejado básicamente a través de los stubs.

1.3. Escribir los Procesos/Programas Ll*amador* (Cliente) y L*lamado* (Servidor)

En el Anexo I.3, se muestra el código ejemplo del cliente que genera el rpcgen, que puede ser utilizado sin modificaciones en este caso (aunque no sería *exactamente* lo mismo que en el ejemplo original de hola.c). Se observa que automáticamente se incluye el archivo de encabezados “hola.h”. La función *main* tiene definida una línea de comandos, en donde el segundo parámetro es el argumento con que se invoca a la misma. Este resulta ser igual al nombre de la máquina remota. Vale aclarar que en este ejemplo generado automáticamente por el rpcgen**,** no se realiza ninguna comprobación de la validez y alcanzabilidad de la máquina remota. A continuación se invoca a la función que propiamente maneja la invocación remota. Esta función declara la estructura CLIENT, cuya forma es la siguiente (obtenida de <rpc/cntl.h>):

```
typedef struct {
        AUTH *cl_auth; /* autentitificador */
        struct clnt_ops {
                enum clnt_stat (*cl_call) (); /* r.p.c. */
                void (*cl_abort) (); /* aborta una llamada */
                void (*cl_geterr) (); /* obtiene código de error */
                bool_t(*cl_freeres) (); /* libera resultados */
                void (*cl_destroy) (); /* destruye la estructura */
                bool_t(*cl_control) (); /* ioctl de rpc */
        } *cl_ops;
        caddr_t cl_private; /* información privada */
        char *cl_netid; /* identificador red */
        char *cl_tp; /* nombre dispositivo */
} CLIENT;
```

Luego se crea el handle del cliente a través de la función *clnt_create* que es parte del protocolo de funciones RPC. Esta función tiene la forma:

```
#include <rpc/rpc.h>

CLIENT *clnt_create(const char *host, const u_long prognum, const u_long versum, const char
*nettype );
```

Retorna: handle cliente si la ejecución fue satisfactoria o NULL si falló. Donde:

- host es una cadena de caracteres que indica el nombre de la máquina remota.
- prognum *y* versnum, son respectivamente los números de programa *y* versión.
- nettype especifica el tipo de protocolo de transporte.

Como se ve en el código, **el protocolo que elige rpcgen para comunicar procesos es UDP**.

Si falla la llamada a *clnt_creat*e, se devuelve NULL. En este caso, se puede invocar a la rutina de la biblioteca *clnt_pcreateerror* para mostrar un mensaje que indica la razón del fallo. Esta función tiene la forma:

```
#include <rpc/rpc.h>
void clnt_pcreateerror (const char *s);
```

Donde los mensajes de error pueden ser:

- RPC_UNKNOWNHOST - anfitrión (host) desconocido.
- RPC_UNKNOWNPROT - protocolo desconocido.
- RPC_UNKNOWNADDR - dirección desconocida.
- RPC_UNKNOWNCAST - sin soporte de broadcast.

Por último, la invocación de la función print_hola utiliza su nuevo nombre print_hola_1, que devuelve un puntero a entero (frente a un entero que devolvía la versión local) y tiene dos argumentos (frente a ninguno para el caso original). Por diseño, todas las RPCs devuelven un puntero. En general, todos los argumentos pasados a la RPC se pasan por referencia, no por valor. Como esta función no tiene originalmente ningún parámetro, *rpcgen* al crear el código fuente, pasa como referencia para "rellenar" este campo, el puntero a char declarado dentro de display_prg_1. El segundo argumento es la referencia a la estructura cliente devuelta por *clnt_creat*e.

Se puede ver ahora el código del servidor que se muestra en el Anexo I.4.1. La función print_hola devuelve ahora un puntero a entero. En este ejemplo, la dirección devuelta está asociada a result que se declara como **static**. Es necesario que el identificador de retorno referenciado sea de tipo static y no *local.* Los identificadores locales se almacenan en la pila y una referencia a sus contenidos, será inválida una vez que la función ha retornado.

Como ya se indicó, el procedimiento cambia de nombre, ahora es princt_hola_svc_1. La correspondencia entre este nombre y el nombre print_hola_1 del cliente se realiza en el stub del servidor. Respecto al argumento, es un puntero. Si se necesitan pasar múltiples argumentos, éstos se deben colocar en una estructura y pasar la referencia a la estructura (o utilizar la opción –N de rpcgen*,* que no es compatible con las versiones originales de RPC de SUN). Se añade un segundo argumento, struct svc_req *req, que contiene información de invocación. El Anexo I.4.2 muestra el agregado necesario para hacer que el procedimiento remoto imprima "Hola, mundo".

1.4. Ejecutar el Servidor en la Máquina Remota, y el Cliente en la Local

La forma más sencilla de probar la aplicación RPC, es corriendo cliente y servidor en la misma máquina. Cada programa remoto está "conectado" a un puerto. El número de este puerto se puede elegir libremente, exceptuando los puertos reservados para "servicios bien conocidos". Es evidente que el "llamador" tendrá que conocer el número de puerto usado por el programa remoto. Para hacer posible la comunicación, a cada servicio del servidor, se le asigna un número de puerto de 16 bits que deberán conocer todos los clientes. El cliente inicialmente sólo conoce el host y el número de programa. Debe ser capaz de obtener el puerto.

Cada máquina que ofrece un programa RPC cuenta con un mecanismo que permite al cliente obtener el número de puerto del servidor (port mapper). La aplicación **portmap** corre en el servidor, y *mapea* el número de programa y de versión, al puerto usado por ese programa (convierte el número de programa a un número de puerto DARPA). Debido a que portmap tiene asignado el número de puerto reservado (111), todo lo que tiene que hacer el cliente es preguntarle al servicio portmap, en el servidor remoto, por el puerto usado por el programa servidor. El servidor portmap sólo tiene conocimiento de los programas de su host (sólo programas RPC en el host local) y debe ser arrancado antes que el servidor RPC. Cada programa RPC, deberá registrarse con el portmap local cuando arranca.

Por otro lado, el comando **rpcinfo** permite conocer qué servicios RPC están activos. Este comando informa qué programas están registrados en el servidor. Un ejemplo de cómo es la salida de la ejecución de este comando sería:

> **rpcinfo –p localhost**

program	**vers**	**proto**	**Port**
100000	2	tcp	111 portmapper
100000	2	udp	111 portmapper
536870913	1	udp	1221
536870913	1	tcp	1223

donde se muestra el número de programa, de versión, el protocolo utilizado y el port asignado a cada servicio. Notar que el programa 536870913 (0x20000001) versión 1 es el programa que corresponde a la definición de "hola.x". La ejecución de rpcinfo permite saber si el servicio portmapper está activo. De no ser así, se lo deberá ejecutar/activar.

En la salida del comando rpcinfo*,* la columna de la derecha aparece en blanco para el programa de ejemplo que se viene desarrollando. Para asociar el número de programa a su nombre, se puede registrar el nombre del servicio remoto, insertando una línea con la sintaxis
"name_of_server_for_rpc_program rpc_program_number [aliases]"
en el archivo /etc/rpc:

```
hola_server                536870913
```

El campo "aliases" es opcional *y* lo se puede omitir. Así, el comando rpcinfo daría:

program	**vers**	**proto**	**Port**
100000	2	tcp	111 portmapper
100000	2	udp	111 portmapper
536870913	1	udp	1221 hola_server
536870913	1	tcp	1223 hola_server

En donde ahora aparece el nombre registrado del servicio que corresponde al programa 0x20000001. Esto naturalmente resulta más fácil de manejar a la hora de realizar las pruebas de una aplicación.

2. Algunos Cambios que Agregan Legibilidad

Tal como se ha desarrollado la explicación y el ejemplo, se dieron las pautas básicas de RPC junto con la utilización de rpcgen. Una vez que se está familiarizado con la forma de programar con RPC-rpcgen hay tareas que son relativamente conocidas y automáticas. De hecho, excepto el primer paso de programar una aplicación no distribuida como tarea inicial, el resto es común a todas las aplicaciones distribuidas en este contexto RPC-rpcgen. Es importante identificar las tareas específicas de quien hace el desarrollo:

- Especificación RPC.
- Programación del Cliente.
- Programación del Servidor.

Y, a partir de aquí, se sigue el ciclo común de verificación, instalación (que tiene algunas características *propias* en los sistemas distribuidos), mantenimiento, etc. De las tres tareas mencionadas antes, tanto la especificación RPC como la programación del servidor son claras en cuanto a lo que involucran y su complejidad es casi la misma que la complejidad de la aplicación. Quizás convenga mirar con más detalle lo referente a la programación del servidor, dado que es posible hacer algunos cambios que son conceptualmente menores pero que agregan mucho en cuanto a legibilidad.

Una de las características más llamativas de la generación *automática* de código por parte de rpcgen consiste en generar una función main() que no hace mucho más que llamar a *otra* función (que se *deriva* de la especificación RPC). En el Anexo I.3 esta función es display_prg_1(char *host). Esta *otra* función tiene dos características importantes y algo llamativas:

1. Lleva a cabo lo se podría llamar de verificación de la conexión con el servidor: básicamente que el servidor exista, esté registrado con RPC, etc., esté operativo para este cliente.
2. Hace una a una (incondicionalmente) todas las llamadas a procedimientos del servidor, en el orden en que fueron especificadas en RPC.

En el ejemplo del Anexo I.3, en la función display_prg_1(char *host), lo que está entre
#ifndef DEBUG
y
#endif /* DEBUG */
es, como se explicó antes, para la generación del handle que después se utilizará para todas las llamadas a procedimientos remotos. A menos que se quiera cambiar algo de esta generación del handle (y que es posible que se lleve a cabo con otros cambios), no es necesario que el programador intervenga en esta tarea. Una de las primeras tareas que el programador puede realizar es, justamente, separar esta parte del código, que se genera automáticamente y no necesita ninguna modificación específica, del resto del código que no tiene estas características.

Por otro lado, la llamada en secuencia y de manera incondicional a todos los procedimientos remotos especificados en RPC no parece tener más lógica que la de dejar ejemplos explícitos de cómo se codifican las llamadas. En el ejemplo dado, esto no genera ningún problema en sí mismo, pero al menos se puede sospechar que si se definen, por ejemplo, 5 procedimientos remotos a priori no tiene mucho sentido que el proceso del cliente, sin intervención del usuario, haga los cinco llamados en secuencia. Además, sería interesante que los procedimientos *sigan siendo vistos como locales*. Esto significaría que sería interesante contar con una función con perfil int print_hola(void) visible desde el main() en vez de una función Int* print_hola_1(void*, CLIENT*) dentro de la función especificada en RPC como program.

La separación de lo que se podría denominar como *de inicialización* del resto tiene relación con la generación del handle, que es lo que necesariamente debe existir para las llamadas remotas. La disponibilidad de las funciones remotas como si fueran locales tiene relación con la generación de *wrappers (envolturas)* o *adaptadores* que dan una interfase más conocida o familiar para los programadores que la que se genera con rpcgen, además de no realizar llamadas en sí mismas, sino por petición del usuario que es quien debe decidir el *comportamiento* o ejecución del proceso cliente. Está claro que estas modificaciones mejoran la legibilidad para el programador, pero no son sin costo adicional.

A modo de ejemplo, se pueden ver estos cambios en el mismo programa que se ha presentado y desarrollado con RPC. Una de las decisiones involucradas en los cambios es la de cómo hacer llegar el handle a los *wrappers* de los llamados remotos. Como casi siempre, hay dos formas: como parámetro o de manera global. Se decide hacerlo global a partir de ahora dado que mejora la legibilidad de los wrappers, pero siempre se pueden discutir las alternativas. Esto significa que la declaración

```
CLIENT *clnt;
```

ya no sería local a ninguna función, ni al main() ni a display_prg_1(). Además, esta función display_prg_1() no tendría más que la generación del handle, las llamadas a procedimientos remotos ya no se harían dentro de esta función, es decir que quedaría:

```
display_prg_1(char *host)
{
#ifndef DEBUG
	clnt = clnt_create (host, display_prg, display_ver, "udp");
	if (clnt == NULL) {
		clnt_pcreateerror (host);
		exit (1);
	}
#endif	/* DEBUG */
}
```

Dado que la “destrucción” (o finalización correcta) del handle ya no puede estar en esta función, se puede generar otra, display_prg_1_end() que lleve a cabo esa tarea (para no incluirla directamente en el propio main(), es decir que habría una nueva función:

```
display_prg_1_end()
{
#ifndef DEBUG
	clnt_destroy (clnt);
#endif	/* DEBUG */
}
```

Todo lo que quedaría por resolver sería la propia llamada a la función remota, dado que tanto display_prg_1() como display_prg_1_end() no son más que “administrativas” en cuanto a que son necesarias para el propio mecanismo de RPC, haya uno o más procedimientos a llamar de manera remota. Se agrega una función más en el main() que sea el wrapper del llamado remoto (para realizarlo como si fuera local). Este wrapper no hace más que definir un “acceso” al procedimiento remoto como si fuera local, es decir como si fuera el de la rutina original. Inicialmente, alcanzaría con que el wrapper realice el llamado remoto y funcione de manera similar al código que genera la herramienta rpcgen. Esto se logra directamente con:

```
void wrap_print_hola()
{
	int  *result_1;
	char *print_hola_1_arg;
	result_1 = print_hola_1((void*)&print_hola_1_arg, clnt);
	if (result_1 == (int *) NULL) {
		clnt_perror (clnt, "call failed");
	}
}
```

Está claro que además de definir una interfase similar (o la misma, que en este caso es posible y general casi siempre lo es) debe realizar el llamado a procedimiento remoto tal como lo prepara rpcgen. De hecho, ya que se tiene un wrapper, se lo podría hacer más semejante a la función local original, tal como está definida en el código de "Hola, mundo" con impresión como procedimiento. Teniendo todo esto en cuenta, la función wrapper que llama al procedimiento remoto (que se ejecutará en el servidor) se define como:

```
int wrap_print_hola()
{
	int  *result_1;
	char *print_hola_1_arg;
	result_1 = print_hola_1((void*)&print_hola_1_arg, clnt);
	if (result_1 == (int *) NULL)
		return 0;
	else
		return *result_1;
}
```

El Anexo I.5 muestra el código modificado completo del hola_client.c para el cual se tomó el generado por rpcgen como punto de partida y se llevaron a cabo los cambios mencionados. Es interesante notar que si se utiliza *solamente* este cambio (la *nueva* función *wrapper*) el programa cliente mostrará impresiones en pantalla como si no funcionara correctamente aunque sí lo haga. El cambio que resta para evitar estos mensajes es sencillo y se deja como tarea.

Quizás el costo más grande de estos cambios es que no se pueden mantener de manera automática cuando cambia la especificación RPC y se tiene que utilizar rpcgen nuevamente. En este caso, se tiene el “cliente” generado automáticamente sin los cambios anteriores y, por lo tanto, habría que rehacerlos. Quizás una manera más sencilla de obtener lo mismo sea teniendo una copia de seguridad del cliente modificado de forma tal que se puedan agregar las partes nuevas (a partir de una nueva especificación RPC y utilización de rpcgen) con la nueva “cáscara” generada por el rpcgen.

3. Bibliografía

- Digital Equipment Corporation, **Programming with ONC RPC,** 1996. (http://www.cs.arizona.edu/computer.help/policy/DIGITAL_unix/AA-Q0R5B-TET1_html/TOC.html)
- Cisco Systems Inc., **Cisco IOS for S/390 RPC/XDR Programmer’s Reference,** 1998. (http://www.cisco.com/univercd/cc/td/doc/product/software/ioss390/ios390rp)
- Red Hat Inc., Red Hat Linux 7.3: Manual oficial de referencia de Red Hat Linux, 2002. (http://www.europe.redhat.com/documentation/rhl7.3/rhl-rg-es-7.3/index.php3)

Anexo I.1: Archivo de especificación “hola.x”.

```
program display_prg
{
   version display_ver
   {
      int print_hola (void) = 1;
   } = 1;
} = 0x20000001;
```

Anexo I.2: Archivo “hola.h” generado por ***rpcgen.***

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

#ifndef _HOLA_H_RPCGEN
#define _HOLA_H_RPCGEN

#include <rpc/rpc.h>


#ifdef __cplusplus
extern "C" {
#endif


#define display_prg 0x20000001
#define display_ver 1

#if defined(__STDC__) || defined(__cplusplus)
#define print_hola 1
extern  int * print_hola_1(void *, CLIENT *);
extern  int * print_hola_1_svc(void *, struct svc_req *);
extern int display_prg_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);

#else /* K&R C */
#define print_hola 1
extern  int * print_hola_1();
extern  int * print_hola_1_svc();
extern int display_prg_1_freeresult ();
#endif /* K&R C */

#ifdef __cplusplus
}
#endif

#endif /* !_HOLA_H_RPCGEN */
```

Anexo I.3: Código ejemplo “hola_client.c” generado con ***rpcgen.***

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

#include "hola.h"


void
display_prg_1(char *host)
{
	CLIENT *clnt;
	int  *result_1;
	char *print_hola_1_arg;

#ifndef DEBUG
	clnt = clnt_create (host, display_prg, display_ver, "udp");
	if (clnt == NULL) {
		clnt_pcreateerror (host);
		exit (1);
	}
#endif	/* DEBUG */

	result_1 = print_hola_1((void*)&print_hola_1_arg, clnt);
	if (result_1 == (int *) NULL) {
		clnt_perror (clnt, "call failed");
	}
#ifndef DEBUG
	clnt_destroy (clnt);
#endif	 /* DEBUG */
}


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

	if (argc < 2) {
		printf ("usage: %s server_host\n", argv[0]);
		exit (1);
	}
	host = argv[1];
	display_prg_1 (host);
exit (0);
}
```

Anexo I.4.1: Código ejemplo “hola_server.c” generado con ***rpcgen.***

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

#include "hola.h"

int *
print_hola_1_svc(void *argp, struct svc_req *rqstp)
{
	static int  result;

	/*
	 * insert server code here
	 */

	return &result;
}
```

Anexo I.4.2: Código del programa servidor (“hola_server.c”) modificado.

```
/*
* CODIGO MODIFICADO PARA EL SERVIDOR
*/

#include “hola.h”

int *
print_hola_1_svc(void *argp, struct svc_req *rqstp)
{
		static int result;
		printf (“hola mundo\n”);
		return &result;
}
```

Anexo I.5: Código del programa cliente (“hola_client.c”) modificado.

```
/*
* CODIGO MODIFICADO PARA EL CLIENTE
*/
#include "hola.h"

/* Declaracion global para el handle del cliente (usado en todos los RPC */
CLIENT *clnt;

/* Creacion del handle */
display_prg_1(char *host)
{
#ifndef DEBUG
	clnt = clnt_create (host, display_prg, display_ver, "udp");
	if (clnt == NULL) {
		clnt_pcreateerror (host);
		exit (1);
	}
#endif  /* DEBUG */
}

/* Destrucción del handle */
display_prg_1_end()
{
#ifndef DEBUG
	clnt_destroy (clnt);
#endif   /* DEBUG */
}

/* Wrapper para el RPC */
int wrap_print_hola()
{
	int  *result_1;
	char *print_hola_1_arg;
	result_1 = print_hola_1((void*)&print_hola_1_arg, clnt);
	if (result_1 == (int *) NULL)
		return 0;
	else
		return *result_1;
}

int main (int argc, char *argv[])
{
	char *host;
	if (argc < 2) {
		printf ("usage: %s server_host\n", argv[0]);
		exit (1);
	}
	host = argv[1];

	display_prg_1(host);			/* Creación del handle */

	 if (wrap_print_hola() > 0)		/* Igual al código original (por el wrapper) */
		printf(“Mision cumplida\n”);
	 else
		printf(“Incapaz de mostrar mensaje\n”);

	display_prg_1_end();			/* Destrucción del handle */
	exit (0);
}
```

Parte II. RPC para RFS (*Remote File System*)

En la primera parte se presentó un ejemplo sencillo para ejercitar RPC con la herramienta rpcgen. El objetivo de esta parte es mostrar cómo programar un servidor sencillo para acceso a (un sistema de) archivos remotos. Dicho servidor, debe contar con las funciones necesarias para poder realizar las siguientes operaciones:

- Leer un archivo
- Copiar un archivo
- Borrar un archivo
- Listar un directorio
- Cambiar de directorio
- Crear un directorio

A manera de ejemplo, en esta parte se comenta el desarrollo de lo que podría ser la función remota "leer archivo". El resto de las funciones tienen un desarrollo similar.

4. Archivo de especificación

En el Anexo II.1 se lista el archivo rfs.x en donde está declarado el programa que permite la operación de lectura remota sobre un archivo. Se incluye aquí también para dar algunas explicaciones respecto a su contenido:

```
/* rfs.x */
typedef opaque file_data<>;

struct open_record
{
  string file_name<>;
  int flags;
};

struct read_record
{
  int fd;
  int count;
};

program RFS
{
  version RFS_VERS_1
  {
    int RFS_OPEN(open_record r) = 1;
    file_data RFS_READ(read_record r) = 2;
    int RFS_CLOSE(int fd) = 3;
  } = 1;
} = 0x20000001;
```

Como se puede ver en el archivo de especificación rfs.x están declarados los procedimientos que corresponden a las tres operaciones básicas que se requieren cuando se lee un archivo: open, read y close. Como cada una de estas operaciones necesita parámetros y retorna resultados, también se observa que se ha definido lo siguiente:

- **file_data**: una cadena de bytes de longitud variable para *contener* o *representar* los datos de los archivos (a leer o escribir, según sea el caso). Es lo que retorna, en el ejemplo, la función de lectura *remota* de un archivo.
- **open_record**: estructura que permite definir el nombre y los permisos del archivo a leer (los dos campos de la estructura).

- **read_record**: estructura que especifica el descriptor asociado al archivo abierto que a leer y la cantidad de caracteres que se pedirán en cada operación de lectura que se realice.

5. Programa Servidor

Como se detalló en la primera parte, el código generado en forma automática del lado del servidor sólo crea la definición de cada procedimiento incluido en el archivo rfs.x. Por lo tanto, las modificaciones realizadas corresponden al código de las funciones creadas que definen su comportamiento, el procesamiento de cada uno de los servicios. Cada una de estas funciones no son mucho más que las correspondientes llamadas al sistema para operar sobre el sistema de archivos local del servidor. En el Anexo II.2 se muestra una versión *completa* del servidor.

6. Programa Cliente

Ejecutando el programa **rpcgen –a**, se genera el código básico del programa ejemplo para el cliente y el servidor (los *esqueletos* de ambos programas). En el Anexo II.3 se ha incluido el programa cliente modificado, sobre la base del ejemplo. Sobre los códigos creados automáticamente por rpcgen, además de incluir algunos encabezados, se pueden mencionar como modificaciones relevantes, las siguientes:

- En la función main():
  - El manejo del nombre del archivo a leer, ya que ***rpcgen*** genera en forma automática solamente el manejo del nombre del host. Esto implica la declaración de una variable local más y también la asignación del parámetro de línea de comandos que le corresponde.
- En la función rfs_1():
  - Se agrega el parámetro para el nombre del archivo.
  - Se agregan las variables locales fd y n. La variable fd se usará para contener el valor de retorno de la función rfs_open_1(). La variable n se usará como índice de la iteración definida para imprimir de los datos de retorno de la función rfs_read_1().
  - Luego de la creación del handle del cliente y previo a la invocación de la función de apertura del archivo, se asignan el nombre del mismo y los correspondientes permisos, según la estructura definida para esto en rfs.x.
  - Previa a la operación de lectura, se definen los miembros de la estructura declarada en rfs.x con el descriptor del archivo y el tamaño del paquete a leer.
  - Se muestran en pantalla los datos retornados por la función de lectura del archivo remoto.

Se debe notar cómo ha "crecido" la función rfs_1() comparativamente, respecto del ejemplo dado en la Parte I, donde solamente había definido un sólo procedimiento a llamar de manera remota. En este Anexo II.2 no se han hecho las modificaciones explicadas en la última sección de la Parte I y por lo tanto la función rfs_1() es proporcionalmente mayor y con mayores funcionalidades (creación del handle y llamado de las tres operaciones remotas definidas en rfs.x).

El programa del cliente, dado en el Anexo II.3 tiene un *pequeño* error pero funciona *correctamente* (aunque parece una contradicción) de todas maneras. Se deja como tarea encontrar y corregir el error. A partir de lo explicado en la siguiente sección quizás quede más claro cuál es este error y cómo llevar a cabo la corrección.

7. Otras Modificaciones al Programa Cliente

Tal como se explicó en la Parte I, es interesante hacer otras modificaciones al programa cliente más allá de lo que no puede llevar a cabo la herramienta rpcgen: separar el manejo del handle de todas las demás operaciones y generar wrappers para cada uno de los llamados remotos. Los cambios necesarios para la asignación/creación y destrucción del handle son exactamente los mismos que en el ejemplo dado en la Parte I, los wrappers siempre dependen de los propios procedimientos a llamar de manera remota, básicamente de los parámetros.

En el Anexo II.4 se muestra el programa cliente completo con las modificaciones incluidas. Es interesante notar que utilizando los wrappers no solamente se proporciona una interfase más familiar

para las operaciones a ejecutar de manera remota se puede llevar a cabo una separación más *natural* del control de errores. En este punto es necesario aclarar que, a diferencia de las funciones y/o procedimientos *locales* estándares, las llamadas remotas tienen un error posible *agregado*: la falla de las comunicaciones con el servidor o, en general, que el servidor no esté disponible por alguna razón. Como se puede notar en cada uno de los wrappers del Anexo II.4, en cada uno de ellos se controla si lo retornado es NULL o no. De hecho, el retorno NULL de los RPC *representa* el error de no haber podido llegar hasta el servidor, el servidor no se pudo contactar y no se pudo resolver el RPC de manera satisfactoria en ejecución. Uno de los ejemplos más explícitos del *doble* control de errores se puede verificar en lo relacionado con la apertura del archivo. En la función

int wrap_rfs_open(char *file_name)

se *controla* o al menos se *notifica* si hay errores con respecto a la resolución del mecanismo de RPC y en la función

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

(que es desde donde se utiliza), se controla si la propia operación de apertura falló o no, de acuerdo a lo reportado desde el servidor, con lo que el valor de retorno del servidor. Quizás en este punto se pueda notar con mayor claridad el pequeño error que se ha mantenido en el código del cliente del Anexo II.4 desde el código correspondiente al del Anexo II.3.

8. Bibliografía

- Sun Microsystems, rpcgen Programming Guide, 2003. (http://www.docs.sun.com/)

Anexo II.1: Archivo de especificación “rfs.x”.

```
/* rfs.x */

typedef opaque file_data<>;

struct open_record
{
    string file_name<>;
    int flags;
};

struct read_record
{
    int fd;
   int count;
};

program RFS
{
   version RFS_VERS_1
   {
      int RFS_OPEN(open_record r) = 1;
       file_data RFS_READ(read_record r) = 2;
      int RFS_CLOSE(int fd) = 3;
   } = 1;
} = 0x20000001;
```

Anexo II.2: Programa Servidor.

```
/*
* programa servidor desarrollado sobre la base del ejemplo que genera rpcgen
*/

#include "rfs.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int * rfs_open_1_svc(open_record *argp, struct svc_req *rqstp)
{
    static int result;

    printf("llamada open\n");
    result = open(argp->file_name, argp->flags);
    return &result;
}

file_data * rfs_read_1_svc(read_record *argp, struct svc_req *rqstp)
{
    static file_data result;
    int n;

    printf("llamada read\n");
    result.file_data_val = (char *) malloc(argp->count);
    if (result.file_data_val == 0)
        result.file_data_len = 0;
    else
        result.file_data_len = read(argp->fd, result.file_data_val, argp->count);
    return &result;
}

int * rfs_close_1_svc(int *argp, struct svc_req *rqstp)
{
    static int result;

    printf("llamada close\n");
    result = close(*argp);
    return &result;
}
```

Anexo II.3: Programa Cliente.

```
/*
* programa cliente desarrollado sobre la base del ejemplo que genera rpcgen
*/

#include "rfs.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void rfs_1(char *host, char *file_name)
{
    CLIENT *clnt; int *result_1;
    open_record rfs_open_1_arg;
    file_data *result_2;
    read_record rfs_read_1_arg;
    int *result_3;
    int rfs_close_1_arg;

    int fd, n;

#ifndef DEBUG
    clnt = clnt_create (host, RFS, RFS_VERS_1, "udp"); if (clnt == NULL) {
    clnt_pcreateerror (host);
    exit (1);
}
#endif /* DEBUG */

    rfs_open_1_arg.file_name = file_name; /* define archivo a leer */
    rfs_open_1_arg.flags = O_RDWR;        /* define permisos */
    result_1 = rfs_open_1(&rfs_open_1_arg, clnt);
    if (result_1 == (int *) NULL) {
        clnt_perror (clnt, "Fallo llamada open");
    }

    fd = *result_1;
    if (fd == -1) {
        printf("Error al abrir el archivo\n"); return;
    }

    rfs_read_1_arg.fd = fd;
    rfs_read_1_arg.count = 20;

    do {
        result_2 = rfs_read_1(&rfs_read_1_arg, clnt);
        if (result_2 == (file_data *) NULL) {
            clnt_perror (clnt, "Fallo llamada read");
        }

        for (n=0; n < result_2->file_data_len; ++n)
            putchar(result_2->file_data_val[n]);
        } while (result_2->file_data_len == 20);
    putchar('\n');

    result_3 = rfs_close_1(&rfs_close_1_arg, clnt);
    if (result_3 == (int *) NULL) {
        clnt_perror (clnt, "fallo llamada close");
    }
```

```
#ifndef DEBUG
    clnt_destroy (clnt);
#endif /* DEBUG */
}

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

    /* Se deben pasar nombre de host y de archivo => argc=3 */
    if (argc < 3) {
        printf ("usage: %s server_host filename\n", argv[0]);
        exit (1);
    }

    host = argv[1];          /* nombre del host remoto */
    file_name = argv[2];  /* nombre del archivo a leer */
    rfs_1 (host, file_name);
    exit (0);
}
```

Anexo II.4: Programa Cliente con otras Modificaciones.

```
/*
* programa cliente desarrollado sobre la base del ejemplo que genera rpcgen
*/

#include "rfs.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/* Handle para las llamadas remotas (creado/asignado en rfs_1() */
CLIENT *clnt;


/* Crea/asigna el handle */
void rfs_1(char *host)
{
#ifndef DEBUG
    clnt = clnt_create (host, RFS, RFS_VERS_1, "udp");
    if (clnt == NULL)
    {
      clnt_pcreateerror (host);
      exit (1);
    }
#endif /* DEBUG */
}

/* Destruye el handle */
void rfs_1_end()
{
#ifndef DEBUG
    clnt_destroy (clnt);
#endif /* DEBUG */
}

/* Wrapper para rfs_open */
int wrap_rfs_open(char *file_name)
{
    open_record rfs_open_1_arg;
    int *result_1;

    rfs_open_1_arg.file_name = file_name; /* define archivo a leer */
    rfs_open_1_arg.flags = O_RDWR;        /* define permisos */

    result_1 = rfs_open_1(&rfs_open_1_arg, clnt);
    if (result_1 == (int *) NULL)
      clnt_perror (clnt, "Fallo llamada open");

    return *result_1;
}

/* Wrapper para rfs_read */
file_data *wrap_rfs_read(int fd, int count)
{
    read_record rfs_read_1_arg;
    file_data *result_2;

    rfs_read_1_arg.fd = fd;
    rfs_read_1_arg.count = count;
```

```
    result_2 = rfs_read_1(&rfs_read_1_arg, clnt);
    if (result_2 == (file_data *) NULL)
      clnt_perror (clnt, "Fallo llamada read");

    return result_2;
}

/* Wrapper para rfs_close */
int wrap_rfs_close()
{
    int *result_3;
    int rfs_close_1_arg;

    result_3 = rfs_close_1(&rfs_close_1_arg, clnt);
    if (result_3 == (int *) NULL) {
        clnt_perror (clnt, "Fallo llamada close");
    }

    return *result_3;
}


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

    int      fd, n;
    file_data *result_2;

    /* Se deben pasar nombre de host y de archivo => argc=3 */
    if (argc < 3) {
        printf ("usage: %s server_host filename\n", argv[0]);
        exit (1);
    }

    host = argv[1];                    /* nombre del host remoto */
    rfs_1 (host);                 /* creacion/asignacion del handle */

    file_name = argv[2];           /* nombre del archivo a abrir/leer */
    fd = wrap_rfs_open(file_name);
    if (fd == -1)
    {
      printf("Error al abrir el archivo\n");
      return;
    }

    do {                          /* lecturas/uso del archivo */
      result_2 = wrap_rfs_read(fd, 20);
      for (n = 0; n < result_2->file_data_len; ++n)
        putchar(result_2->file_data_val[n]);
    } while (result_2->file_data_len == 20);
    putchar('\n');

    wrap_rfs_close();              /* cierre del archivo */

    rfs_1_end();                  /* destruccion del handle */
    exit (0);
}
```