Práctica 4:

Construcción de un servidor de archivos remoto

Objetivo de la práctica:

Construir un sencillo servidor de archivos en red sin estado utilizando (RPC) Llamadas a Procedimiento Remoto mediante la herramienta *rpcgen*.

1 Introducción

Vamos a construir un servidor de archivos sin estado que implementa sólo algunas funciones para manipular archivos y directorios. Estas son: *creat*, *write*, *read*, *rename* para archivos, *y mkdir y rmdir* para directorios. Como complemento se pueden añadir otras funciones para manipular archivos y directorios, u obtener propiedades (tamaño, fecha de modificación, etc.).

Para ilustrar algunos de los problemas que se presentan en un sistema distribuido real, tomemos la función siguiente:

```
int write_file (int fd, char *buf, int bytes)
{
      return put_block (fd, nbytes, buf );
}
```

donde la función `put_block` es una versión remota de la función `write` ordinaria.

Si diseñamos un servicio remoto, este es fiable si la red funciona correctamente y ninguno de los participantes sufre una caída. Las llamadas remotas se complican por el hecho de que el cliente o el servidor pueden fallar de forma independiente. Consideremos que un cliente emite una solicitud y no tiene respuesta. Después de cierto tiempo, el cliente repite la solicitud, ¿qué ocurre? La respuesta depende de la razón por la que no se recibió respuesta:

1. La solicitud se perdió en la red.
2. El servidor recibió la solicitud y la atendió, pero se perdió la respuesta.
3. El servidor no esta activo.

En el primer caso, el cliente puede volver a intentarlo sin consecuencias adversas. En el segundo, se produciría un resultado adverso. Si la primera solicitud se atendió, como resultado de la escritura, el puntero de lectura/escritura se verá adelantado y la repetición de la escritura producirá un resultado diferente. Se podría resolver este problema con números de secuencia de las solicitudes pero esto no resuelve el problema si falla el servidor.

Una estrategia para resolver ambos problemas, la perdida de respuesta y la caída del servidor, es realizar las operaciones para que se puedan repetir las solicitudes sin alterar los resultados. Se dice entonces que este tipo de operaciones son idempotentes. La función del ejemplo no es idempotente, pero podemos modificarla para que lo sea:

```
int put_block (int file, int offset, int count, char *data)
{
      int returncode = 0;
      if (lseek(file, offset, SEE_SET) == -1)
            returncode = -1;
```

```
            else
                  returncode = write(file, data, count);
            return returncode;
      }
```

La función anterior es idempotente en el sentido de que un cliente puede repetir la llamada y hacer que escriba los mismos datos en el mismo lugar del archivo. Por supuesto, la repetición de la función cambia la hora de modificación del archivo pero ese es un problema que no tendremos en cuenta.

Para invocar a la función `put_block` como llamada remota, el cliente mantiene un registro de la posición en la que escribir en el archivo, y lo actualiza cuando recibe la confirmación de que la función tuvo éxito. Por ejemplo,

```
      static int fd_offset = 0;
      Int write_file(int fd, char *buf, int nbytes)
      {
            int bytes_written;
            if ((bytes_written = put_block(fd, fd_offset, nbytes, buf)) != -1)
                  fd_offset += bytes_written;
            return bytes_written;
      }
```

La variable del cliente `fd_offset` del ejemplo, guarda la posición del archivo de la última solicitud con éxito. Antes del invocar a `write_file`, el programa debe abrir el archivo remoto para obtener un manejador para utilizarlo en las operaciones siguientes. En el ejemplo, se utiliza el entero fd, pero en una implemetación "real" podría ser una estructura opaca.

La función `put_block` no resuelve por completo el problema del servidor caído. Al caer el servidor y arrancar de nuevo, si recibe una petición de escritura no tiene forma de saber sobre que archivo tiene que hacerlo (ha perdido el descriptor del archivo). También, si el cliente falla, no podrá cerrar el archivo y este permanecería abierto en el servidor. La solución es diseñar un servidor que no mantenga el estado de sus clientes. La nueva versión de la función `put_block` abre el archivo, busca la posición apropiada, escribe el bloque y cierra el archivo. El cliente mantiene un registro local de la posición en el archivo para determinar la posición de búsqueda de las llamadas:

```
      int put_block (char *fname, int offset, int count, char *data)
      {
            int return code = 0;
            int file;
            if ((file = open(fname, 0_WRONLY | 0_CREAT, 0600) == -1)
                  returncode = -1;
            esle if (lseek(file, offset, SEEK_SET) == -1)
                  returncode = -1;
            else
                  returncode = write(file, data, count);
            close(file);
            return returncode;
      }
```

La especificación de la versión sin estado del servicio remoto de escritura sería:

```
/*  rfile.x   */
const MAX_BUF = 1024;
const MAX_STR = 256;

struct write_packet {
      string fname<MAX_STR>;
      int count;
      int offset;
      char data<MAX_BUF>;
};
```

```
program RFILEPROG {
      version RFILEVERS {
            int PUT_BLOCK(write_packet) = 1;
      } =1;
} = 0x31111112;
```

Se pide construir la aplicación distribuida propuesta que soporte las operaciones sobre archivos indicada al inicio: lectura, escritura, creación, y renombrado. Dejando la posibilidad de que estas operaciones se completen.

2 Desarrollo de la práctica: ¿cómo distribuir una aplicación?

En este apartado, vamos a completar los fundamentos vistos en teoría sobre las RPCs con los detalles de implementación necesarios cuando utilizamos *rpcgen*. Para ello, utilizaremos un sencillo ejemplo conocido por todos, el programa "hola, mundo". Además, aprovecharemos para mostrar los pasos que debemos de seguir para construir una aplicación distribuida.

Cuando queremos distribuir una aplicación utilizando RPCs deberemos de seguir el siguiente procedimiento:

1. Construir y probar una aplicación convencional que resuelva el problema. Debemos separar del programa anterior los procedimientos que se van a ejecutar en una máquina remota, y ponerlos en un fichero aparte.
2. Escribir una especificación *rpcgen* para el programa remoto, y compilarla.
3. Escribir las rutinas de interfaz con los tocones cliente y servidor.
4. Compilar y enlazar, por una parte, el programa cliente, y, por otra, el servidor.
5. Ejecutar el servidor en la máquina remota, y el cliente en la local.

Vamos a ver cada uno de los pasos anteriores, utilizando el ejemplo citado.

1. Construimos una aplicación que resuelve el problema

El código que se muestra a continuación es una reimplementación del programa "Hola mundo" en el que la función para imprimir el mensaje se ha realizado como un procedimiento, para poder distribuir la aplicación, ya que la impresión se realizará en el servidor.

Programa 1.- Código de "hola mundo" con impresión como procedimiento.

```
#include <stdio.h>
#include <stdlib.h>
void
main(void) {
     int    print_hola(void);
     if ( print_hola())
      printf("Mision cumplida\n");
     else
      printf("Incapaz de mostrar mensaje\n);
     exit(0);
}
int
print_hola(void) {
     return printf("Hola, mundo \n");
}
```

2. Creamos una especificación *rpcgen* y la compilamos

Una vez definida la estructura de la aplicación distribuida, preparamos la especificación para *rpcgen* que es básicamente una declaración de un programa remoto y de las estructuras de datos que utiliza. Este archivo de especificación puede contener:

- Declaraciones de constantes utilizadas por el cliente, y, más a menudo, del servidor.
- Declaraciones de los tipos de datos utilizados, especialmente como argumentos de los procedimientos remotos.
- Declaraciones de programas remotos, los procedimientos que estos contienen y los tipos de sus parámetros.

La especificación debe estar escrita en el lenguaje de programación RPC, que aunque es similar a C tiene sus diferencias. Además, hay que indicar que RPC utiliza números para nombrar los programas y procedimientos remotos, y el lugar para esta identificación en el archivo de especificación. Por ello, vamos a ver por separado cada uno de estos aspectos.

2.1 Lenguaje de especificación de interfaces RPC

El lenguaje RPC es una extensión del lenguaje XDR (e*Xtended Data Representation*) a la que se han añadido los tipos *program* y *version*.

Un archivo de especificación RPC consta de una serie de definiciones:

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

donde se reconocen seis tipos de definiciones:

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

- `struct` - se declaran casi exactamente como las de C. Su sintaxis es:

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

declaration-list:
      declaration ";"
      declaration ";" declaration-list
```

- `union` - son uniones discriminadas, y son algo diferentes que las de C. Son más parecidas a los registros variables de Pascal.

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

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

- `enumeration` - tienen la misma sintaxis que en C:

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

enum-value-list:
       enum-value
```

```
        enum-value "," enum-value-list

enum-value:
        enum-value-ident
        enum-value-ident "=" value
```

- `typedef` - Igual que en C:

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

- `constans` - simbolizan constantes que se pueden utilizar de la misma manera que constantes enteras. Por ejemplo, en la especificación del tamaño de una matriz.

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

- `program` - se declara de la siguiente manera:

```
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
```

- `declaration` - En XDR existen sólo cuatro tipos de declaraciones:

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

donde

- Declaraciones simples: como las declaraciones en C.

```
simple-declaration:
        type-ident variable-ident
```

- Declararación de matrices de longitud fija: como matrices en C:

```
fixed-array-declaration:
        type-ident variable-ident "[" value "]"
```

- Declaración de matrices de longitud variable: no existen en C. XDR inventa su sintaxis con los signos <valor>, donde valor especifica el tamaño máximo. Si se omite, la matriz puede ser de cualquier tamaño.

```
variable-array-declaration:
       type-ident variable-ident "<" value ">"
       type-ident variable-ident "<" ">"
```

- Declaración de punteros: exactamente como en C.

```
pointer-declaration:
        type-ident "*" variable-ident
```

- Existen algunas excepciones a las reglas vista anteriormente:
  - *Boolean* - La biblioteca RPC soporta el tipo `bool_t` que es bien `TRUE` o `FALSE`.
  - *String* - Las cadenas se declaran utilizando la palabra clave "string"
  - *Opaque* - El tipo opaque se utiliza para declarar datos sin tipo, sólo secuencias de bytes de tamaño arbitrario.
  - *Void* - en una declaración `void`, la variable no tiene nombre. Estas declaraciones sólo pueden ocurrir en: definiciones de uniones y definiciones de programas (como argumento o resultado de un procedimiento remoto).

A continuación vamos a ver un ejemplo de archivo de especificación RPC (**.x**) y el resultado de su procesamiento por el preprocesador que lo transforma en código C.

Programa 2.2.- Ejemplo de un archivo .x y su equivalente compilado.

Archivo .x	Archivo compilado por rpcgen
`const MAX = 1024;` `const DELIMITADOR = "@"`	`#define MAX = 1024;` `#define DELIMITADOR 0 "@";`

`enum primarios { rojo, amarillo = 4, azul };`	`enum primarios {`
	`Rojo = 0,`
	`Amarillo = 4,`
	`Azul = 4 + 1`
	`};`
	`typedef enum primarios primarios;`

`typedef colores extraño; /* simple */`	`typedef colores extraños;`
`typedef char linea[80]; /* matriz longitud fija */`	`typedef char linea[80];`
`typedef string var_linea<80>; /*matriz longitud variable */`	`typedef char *var_linea;`

`typedef int algun_ints< >; /*matriz long. variable sin máx.*/`	`typedef struct {`
	`u_int algun_ints_len;`
	`Int *algun_ints_val;`
	`} algun_ints;`

`typedef var_linea *linea_ptr; /* Puntero */`	`Typedef var_linea *linea_ptr;`

`struct registro {`
`var_linea nombre;`
`Int edad;`
`};`
	`struct registro {`
	`var_linea nombre;`
	`Int edad;`
	`};`
	`typedef struct registro registro;`

`union ret_valor conmuta( extern errno) {`
`case 0:`
`linea respuesta;`
`default:`
`void;`
`};`
	`struct ret_valor {`
	`extern errno;`
	`union {`
	`linea respuesta;`
	`} ret_valor_u;`
	`};`
	`typedef struct ret_valor ret_valor;`

Por último, indicar una lista de palabras clave que no pueden ser utilizadas por tanto como identificadores:

```
bool     const     enum      int          string     typedef
char     double    hyper     quadruple    switch     unsigned
void     case      default   float        struct     union
```

2.2 Identificación de programas y procedimientos remotos

El estándar RPC de Sun especifica que cada programa remoto debe poseer un entero único de 32 bits que lo identifica. Lo mismo ocurre con cada procedimiento dentro cada programa remoto. Los procedimientos se numeran secuencialmente desde 1 a N (el 0 se reserva para un procedimiento de eco que se utiliza para comprobar si el programa remoto es alcanzable). Además, cada programa remoto puede incluye un identificador de versión (normalmente la primera versión tiene el asignado el 1). El identificador de versión permite cambiar detalles de un procedimiento remoto sin tener que cambiar el número de programa. La especificación RPC permite que se puedan ejecutar en un computador múltiples versiones de un programa remoto simultáneamente, permitiendo la migración entre versiones durante los cambios.

Así cada RPC se identifica por la tripleta *(programa, versión, procedimiento)*. La Tabla 2.1 identifica los 8 grupos de número de programas que ha establecido Sun Microsystems para asegurar que no haya conflictos en la asignación de estos en organizaciones diferentes. La Tabla 2.2 muestra algunos de los números de programa asignados por Sun.

Tabla 2.1.- Número de programas utilizados en RPC.

Rango	Valores asignados por
0x00000000-0x1fffffff	Sun Microsystems
0x20000000-0x3fffffff	Administrador del sistema local
0x40000000-0x5fffffff	Transitorios (temporales)
0x60000000-0x7fffffff	Reservados (definidos por usuario)
0x80000000-0x9fffffff	Reservados (definidos por usuario)
0xa0000000-0xbfffffff	Reservados (definidos por usuario)
0xc0000000-0xdfffffff	Reservados (definidos por usuario)
0xe0000000-0xffffffff	Reservados (definidos por usuario)

Tabla 2.2.- Algunos ejemplos de números de programa asignados por Sun.

Nombre	Número	Descripción
.portmap	100000	Port mapper
.nfs	100003	Network File System
.ypserv	100004	NIS
.mountd	10005	Mount, showmount
.ypbind	100007	NIS binder
.lockd	100020	Gestor cerrojos locales
.lockd	100021	Gestor cerrojos de red

- **Definición del protocolo para nuestro ejemplo, archivo hola.x**

Siguiendo con nuestro ejemplo, el archivo de especificación `hola.x` tiene la forma:

```
/* hola.x -- archivo de definicion del protocolo escrito en lenguaje RPC que se
     ha de pasar a 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;
```

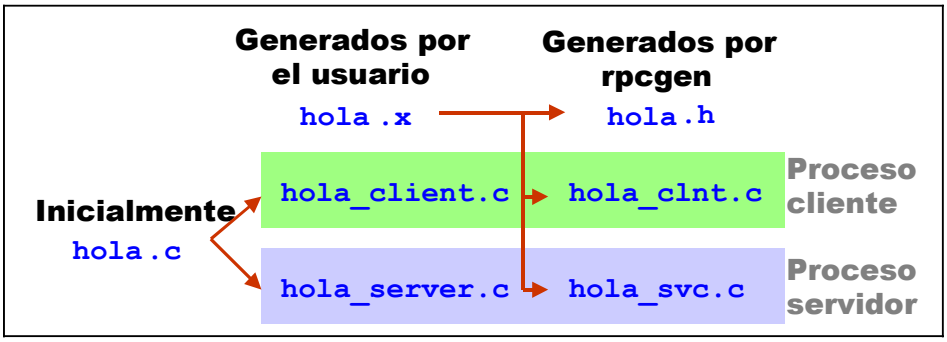
```
    } = 1
} = 0x20000001;
```

Una vez creado el archivo, lo compilamos con rpcgen:

```
% rpcgen -C hola.x
```

donde hemos compilado con la opción C que genera código conforme ANSI C. Otras opciones podéis verlas en la página de manual.

Los archivos creados y su relación aparecen en la figura.



- **El archivo hola.h**

```
/* 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 ((unsigned long)(0x2000001))
#define DISPLAY_VER ((unsigned long)(1))
#if defined(__STDC__) || defined(__cplusplus)
#define print_hola ((unsigned long)(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 ((unsigned long)(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 */
```

El archivo `hola.h` creado por *rpcgen* debe incluirse en los archivos tocones del cliente y del servidor. En este archivo encontramos los identificadores especificados 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 y en minúsculas. Siguiendo esta definición, tenemos dos prototipos para la función `print_hola`. El primer prototipo `print_hola_1` se utiliza en el tocón del cliente. El segundo, `print_hola_1_svc`, en el tocón del servidor. Aquí, la convención utilizada por *rpcgen* es añadir al nombre del procedimiento un subrayado (_) y el número de la versión (1), para el tocón del cliente, y lo mismo pero con `_svc` para el servidor.

3 Escribir la interfaz con los tocones, en el cliente y en el servidor

Como se observa en el siguiente código, el del programa cliente -programa 2.3-, se han realizado algunos cambios para acomodar las llamadas a procedimiento remoto.

Programa 2.3.- Programa cliente, `hola_client.c`.

```
/* Programa Cliente: hola_client.c */
#include <stdio.h>
#include <stdlib.h>
#include "hola.h"

void
main(int argc, char *argv[]) {
     CLIENT *client;
     int          *return_value, filler;
     char         *server;
/*Especificar el host donde se va a ejecutar como argmento 1*/
     if (argc != 2) {
       fprintf(stderr,"Uso:%s nombre del host\n", *argv);
       exit (1); }
     server= argv[1];
/*Generar handle cliente para llamar al servidor*/
if ((client=clnt_create(server, DISPLAY_PRG, DISPLAY_VER, "visible")) = (CLIENT
     *) NULL) {
     clnt_pcreateerror(server));
     exit(2);
}
return_value=print_hola_1((void *) &filler, client);
if (*return_value)
     printf("Mision completada\n");
else
     printf("Mensaje no impreso\n");
exit(0);
}
```

A continuación indicamos los cambios más notables. Primero, se han incluido dos archivos de cabecera adicionales. `stdlib.h` que contiene el prototipo de la función *exit*, y el archivo `hola.h`.

Además, en el código aparece declarada la estructura cliente, 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) ();        /* obten codigo 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;
```

El nombre de la máquina remota se toma del primer argumento de la línea de órdenes, y no se realizan comprobaciones de su validez y alcanzabilidad. A continuación se crea el *handle* del cliente a través de la función *clnt_create* que es parte del protocolo de funciones RPC.

La función `clnt_create` 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 OK, NULL si fallo.
```

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, y puede ser:

`NULL` o variable `netpath`,
`visible` - busca secuencialmente en **/etc/netconfig** para encontrar un protocolo con el indicador "v" -visible),
`circuit_v` - similar a visible pero busca transportes orientados a conexión (`tpi_costs` o `tpi_costs_ord`).
`datagram_v` - idem pero no orientados a conexión (`tpi_clts`)
`circuit_n` - similar a `netpath` pero elige transporte de datagramas orientados a conexión.
`datagram_n` - idem no orientados a conexión.
`udp` - protocolo UDP.
`tcp` - protocolo TPC.

Si falla la llamada a *clnt_create*, se devuelve NULL. En este caso, podemos 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 desconocido
`RPC_UNKNOWNPROT` - protocolo desconocido
`RPC_UNKNOWNADDR` - dirección desconocida
`RPC_UNKNOWNCAST` - sin soporte de broadcast

Para finalizar con el cliente, indicar que la invocación de la función `print_hola` utiliza su nuevo nombre `print_hola_1`, ahora devuelve un puntero a entero (frente a un entero), y tiene dos argumentos (frente a ninguno). 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 nuestra función no tiene originalmente ningún parámetro, se utiliza el identificador `filler` para rellenar el lugar. El segundo argumento es la referencia a la estructura cliente devuelta por *clnt_create*.

En cuanto al código del servidor, que aparece en el programa 2.4, hemos realizado las modificaciones que se indican a continuación. Para acomodar la llamada a procedimiento remoto, ahora, la función `print_hola` devuelve un puntero a entero. En nuestro ejemplo, la dirección devuelta esta asociada al identificador `ok`. Este identificador se declara como **static**. Es un imperativo que el identificador de retorno referenciado sea de tipo *static*, como opuesto a local. Los identificadores locales se almacena en la pila y una referencia a sus contenidos será invalido una vez que la función a retornado.

Como ya indicamos, el procedimiento cambia de nombre, ahora es `princt_hola_svc_1`. Pero no debemos de preocuparnos, la correspondencia entre el nombre `print_hola_1` del cliente y este se realiza en el tocón del servidor. Respecto al argumento, es un puntero. Si se necesitan

pasar múltiples argumentos estos se deben colocar en una estructura y pasar la referencia a la estructura (en versiones modernas, se puede dar la opción -N a *rpcgen* para escribir RPCs de múltiples argumentos, cuando se pasa un parámetro por valor no por referencia, o cuando se va a devolver un valor, no un puntero).

Se añade un segundo argumento, `struct svc_req *req`, que contiene información de invocación.

Programa 2.4.- Programa hola_server.c.

```
/* programa SERVIDOR:hola_server.c
     ejecutado por un proceso remoto */
#include <stdio.h>
#include "hola.h" /*generado por rpcgen*/

int *
print_hola_1_svc(void *filler, struct svc_req *req);
{
     static int   ok;
     ok = printf("Hola mundo\n");
     return(&ok);
}
```

Para finalizar con este apartado, hemos incluido por curiosidad los archivos de los tocones generados por *rpcgen*, `hola_clnt.c` y `hola_svc.c` (programas 2.5 y 2.6, respectivamente). El archivo contiene la llamada real a la función `print_hola_1`, que se realiza a través de la función clnt_call la cual es la que realiza realmente la llamada a procedimiento remoto. Esta función tiene como parámetros: el handle del cliente devuelto por `clnt_create`; la constante `print_hola` obtenida de `hola.h`; referencias a las rutinas XRD de codificación/decodificación de datos; a continuación, una referencia al argumento inicial que se pasará al procedimiento remoto por el servidor; y, por último, el valor `TIMEOUT`. Aunque el comentario inicial del archivo indica que no se modifique el contenido del mismo, este valor puede modificarse de su valor por defecto, 25, por otro razonable máximo impuesto por el usuario.

Programa 2.5.- Archivo hola_clnt.c.

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

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

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

int *
print_hola_1(void *argp, CLIENT *clnt)
{
     static int clnt_res;

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

```
    return (&clnt_res);
}
```

El código de `hola_svc.c` es bastante más largo que el del tocón del cliente, y no estudiaremos en detalle por su complejidad, y por no alargar innecesariamente este guión (el lector curioso puede descifrarlo). Este programa, además del código correspondiente a la RPC, contiene lo necesario para lanzar de fondo el proceso servidor.

Programa 2.6.- Archivo hola_scv.c.

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

#include "hola.h"
#include <stdio.h>
#include <stdlib.h>                     /* getenv, exit */
#include <rpc/pmap_clnt.h>              /* for pmap_unset */
#include <string.h>                     /* strcmp */
#include <signal.h>
#ifdef __cplusplus
#include <sysent.h>                     /* getdtablesize, open */
#endif /* __cplusplus */
#include <unistd.h>                     /* setsid */
#include <sys/types.h>
#include <memory.h>
#include <stropts.h>
#include <netconfig.h>
#include <sys/resource.h>               /* rlimit */
#include <syslog.h>

#ifndef     SIG_PF
#define     SIG_PF void(*)(int)
#endif
#ifdef      DEBUG
#define     RPC_SVC_FG
#endif

#define     _RPCSVC_CLOSEDOWN 120
static int _rpcpmstart;             /* Started by a port monitor ? */
/* States a server can be in wrt request */
#define     _IDLE 0
#define     _SERVED 1
#define     _SERVING 2
static int _rpcsvcstate = _IDLE;     /* Set when a request is serviced */
static
void _msgout(char* msg)
{
#ifdef RPC_SVC_FG
     if (_rpcpmstart)
       syslog(LOG_ERR, msg);
     else
       (void) fprintf(stderr, "%s\n", msg);
#else
     syslog(LOG_ERR, msg);
#endif
} static void
closedown(int sig) {
     if (_rpcsvcstate == _IDLE) {
       extern fd_set svc_fdset;
```

```
      static int size;
      int i, openfd;
      struct t_info tinfo;
      if (!t_getinfo(0, &tinfo) && (tinfo.servtype == T_CLTS))
            exit(0);
      if (size == 0) {
            struct rlimit rl;
            rl.rlim_max = 0;
            getrlimit(RLIMIT_NOFILE, &rl);
            if ((size = rl.rlim_max) == 0) {
                  return; }
      }
      for (i = 0, openfd = 0; i < size && openfd < 2; i++)
            if (FD_ISSET(i, &svc_fdset))
                  openfd++;
      if (openfd <= 1)
            exit(0);    }
     if (_rpcsvcstate == _SERVED)
       _rpcsvcstate = _IDLE;
     (void) signal(SIGALRM, (SIG_PF) closedown);
     (void) alarm(_RPCSVC_CLOSEDOWN/2);  }
static void
display_prg_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
     union {
       int fill;
     } argument;
     char *result;
     xdrproc_t xdr_argument, xdr_result;
     char *(*local)(char *, struct svc_req *);

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

     case print_hola:
       xdr_argument = (xdrproc_t) xdr_void;
       xdr_result = (xdrproc_t) xdr_int;
       local = (char *(*)(char *, struct svc_req *)) print_hola_1_svc;
       break;
default:
       svcerr_noproc(transp);
       _rpcsvcstate = _SERVED;
       return;
     }
     (void) memset((char *)&argument, 0, sizeof (argument));
     if (!svc_getargs(transp, xdr_argument, (caddr_t) &argument)) {
       svcerr_decode(transp);
       _rpcsvcstate = _SERVED;
       return;
     }
     result = (*local)((char *)&argument, rqstp);
     if (result != NULL && !svc_sendreply(transp, xdr_result, result)) {
       svcerr_systemerr(transp);
     }
     if (!svc_freeargs(transp, xdr_argument, (caddr_t) &argument)) {
       _msgout("unable to free arguments");
       exit(1);
     }
```

```
     _rpcsvcstate = _SERVED;
     return;
}
main() {
     pid_t pid;
     int i;
     char mname[FMNAMESZ + 1];
     if (!ioctl(0, I_LOOK, mname) &&
       (!strcmp(mname, "sockmod") || !strcmp(mname, "timod"))) {
       char *netid;
       struct netconfig *nconf = NULL;
       SVCXPRT *transp;
       int pmclose;

       rpcpmstart = 1;
       openlog("hola", LOG_PID, LOG_DAEMON);
       if ((netid = getenv("NLSPROVIDER")) == NULL) {
       /* started from inetd */
             pmclose = 1;
       } else {
             if ((nconf = getnetconfigent(netid)) == NULL)
                   _msgout("cannot get transport info");

             pmclose = (t_getstate(0) != T_DATAXFER);
       }

if (strcmp(mname, "sockmod") == 0) {
             if (ioctl(0, I_POP, 0) || ioctl(0, I_PUSH, "timod")) {
                   _msgout("could not get the right module");
                   exit(1);
             }
       }
       if ((transp = svc_tli_create(0, nconf, NULL, 0, 0)) == NULL) {
             _msgout("cannot create server handle");
             exit(1);
       }
       if (nconf)
             freenetconfigent(nconf);
       if (!svc_reg(transp, DISPLAY_PRG, DISPLAY_VER, display_prg_1, 0)) {
             _msgout("unable to register (DISPLAY_PRG, DISPLAY_VER).");
             exit(1);
       }
       if (pmclose) {
             (void) signal(SIGALRM, (SIG_PF) closedown);
             (void) alarm(_RPCSVC_CLOSEDOWN/2);
       }
       svc_run();
       exit(1);
     /* NOTREACHED */
     }else {
#ifndef RPC_SVC_FG
       int size;
       struct rlimit rl;
       pid = fork();
       if (pid < 0) {
             perror("cannot fork");
             exit(1); }
       if (pid)
             exit(0);
       rl.rlim_max = 0;
       getrlimit(RLIMIT_NOFILE, &rl);
       if ((size = rl.rlim_max) == 0)
             exit(1);
```

```
      for (i = 0; i < size; i++)
            (void) close(i);
      i = open("/dev/console", 2);
      (void) dup2(i, 1);
      (void) dup2(i, 2);
      setsid();
      openlog("hola", LOG_PID, LOG_DAEMON);
#endif
     }
     if (!svc_create(display_prg_1, DISPLAY_PRG, DISPLAY_VER, "netpath")) {
      _msgout("unable to create (DISPLAY_PRG, DISPLAY_VER) for netpath.");
      exit(1);
     }
     svc_run();
     _msgout("svc_run returned");
     exit(1);
     /* NOTREACHED */
}
```

---

4 Compilar el cliente y el servidor

1. Primero compilamos el cliente, indicando a enlazador que debe utilizar las bibliotecas de red:
   ```
   % gcc hola_client.c hola_clnt.c -o cliente -lnsl
   ```
2. Segundo, el servidor:
   ```
   % gcc hola_server.c hola_svc.c -o servidor -lnsl
   ```

En esta segunda compilación tendremos mensajes de error (identificadores declarados y no usados, etc.) que no evitan que se genere el ejecutable de servidor.

5 Ejecutar el servidor y el cliente

1. Podemos probar el ejemplo, ejecutando los procesos en la misma máquina:
   ```
   maq_local% servidor
   maq_local% cliente maq_local
   Mision cumplida
   ```

Curiosamente, vemos el mensaje "misión cumplida" pero no el de "Hola mundo". Esto se debe a que al lanzar de fondo el proceso servidor, lo desligamos del terminal de control y por tanto su salida es descartada.

2. Podemos probarlo de forma distribuida:

   ```
   maq_remota% servidor
   maq_local% cliente maq_remota
   ```

   o bien

   ```
   maq_local% servidor
   maq_local% rsh maq_remota $cwd/cliente maq_local
   Mision cumplida
   ```

Ampliación de la práctica (opcional)

Opcionalmente se puede implementar un servidor de archivos multihebrado utilizando la biblioteca de hebras de Linux que permita el servicio de archivos simultáneamente a varios procesos clientes.