# Guía de programación con las RPC de Sun/ONC

## Introducción

El mecanismo de llamada a procedimiento remoto del *Open Network Computing* de Sun es uno de los más extendidos, debido al indiscutible éxito de algunos de los productos que se sustentan sobre el mismo, como NFS. Eso hace que, a pesar de sus evidentes limitaciones frente a otros mecanismos de RPC, como el proporcionado por el entorno DCE, resulte una opción interesante a la hora de familiarizarse, desde un punto de vista docente, con la programación de servicios utilizando RPCs, ya que cualquier sistema UNIX dispone en su instalación básica de todo el software requerido para su utilización.

No existe, sin embargo, mucha documentación sobre cómo programar usando este mecanismo (en mi opinión, la mejor referencia es el capítulo 16 del libro de Richard Stevens *UNIX Network Programming, Volume 2, Second Edition: Interprocess Communications*, Prentice Hall, 1999, ISBN 0-13-081081-9). Este documento nace, por tanto, con el objetivo de proporcionar una guía rápida para la programación en este entorno, de manera que no se requiera consultar ningún documento adicional a la hora de acometer las prácticas de la asignatura que usan esta tecnología.

Dado el carácter aplicado que pretende tener esta guía, la presentación se realizará usando programas de ejemplo de progresiva dificultad (este enlace permite la descarga de los programas de ejemplo):

- Primer caso: servicios con tipos de datos simples. El código se encuentra en los directorios `caso_1_erroneo`, una versión inicial intencionadamente errónea, y `caso1_OK`.
- Segundo caso: servicios con tipos de datos compuestos. El código se encuentra en el directorio `caso_2`.
- Uso de tipos de XDR no soportados directamente en C:
  - Tercer caso: uso de tipo `union` de XDR. El código se encuentra en el directorio `caso_3`.
  - Cuarto caso: uso de vectores de tamaño variable de XDR. El código se encuentra en el directorio `caso_4`.

**NOTA:** En algunas distribuciones de Linux no se permite, por defecto, que un usuario normal pueda arrancar servidores que usen este tipo de RPC. Para poder hacerlo, hay dos posibilidades:
- Lanzar el servidor como superusuario (`sudo`). Nótese que el cliente lo puede ejecutar cualquier usuario.
- Rearrancar el proceso `rpcbind` con la opción `-i` para que sí permita que los usuarios normales puedan arrancar servidores RPC.

## Primer caso: servicios con tipos de datos simples

Se pretende desarrollar un servidor que ofrezca a los clientes dos operaciones:
- Dado un `uid`, averiguar cuál es el nombre de usuario que tiene asignado ese identificador en la máquina donde ejecuta el servidor. Por tanto, se trata de un servicio que requiere un único parámetro de tipo entero y retorna un único valor: el *string* correspondiente al nombre de usuario asociado al identificador.
- Dado un nombre de usuario, averiguar cuál es el `uid` que tiene asignado en la máquina donde ejecuta el servidor. En este caso, recibe un único parámetro de tipo *string* y retorna un entero.

### Definición del servicio

El primer paso a la hora de desarrollar un servicio es especificar en un fichero IDL, con extensión `.x`, el nombre del servicio (palabra reservada `program`) y su número de versión (`version`), asignándoles, además de un nombre, sendos identificadores numéricos. En el caso del servicio usado como ejemplo, la definición podría ser como la que aparece en el siguiente listado.

```
program RUSUARIOS_SERVICIO {
     version RUSUARIOS_VERSION {
     /* pendiente de rellenar */
     }=1;
}=666666666;
```

Observe el uso de mayúsculas en los nombres de los identificadores. Esta convención reduce la posibilidad de conflictos entre los nombres generados automáticamente por el sistema de RPCs y los que usa la aplicación.

El próximo paso sería declarar los servicios que proporcionará el servidor. El siguiente listado (fichero `rusuarios.x` del directorio `caso1_erroneo/idl`) muestra esta declaración, donde nuevamente se han usado nombres en mayúsculas por los motivos antes explicados y se le han asignado valores enteros sucesivos a partir de 1 a cada función (el 0 está reservado para un servicio nulo que incluye automáticamente el sistema de RPCs en cada servidor).

```
program RUSUARIOS_SERVICIO {
     version RUSUARIOS_VERSION {
                int OBTENER_UID(string nombre)=1;
                string OBTENER_NOMBRE(int uid)=2;
     }=1;
}=666666666;
```

En este momento ya se pueden generar los resguardos del cliente y del servidor. Para ello, hay que procesar el fichero IDL que se acaba de definir usando el mandato `rpcgen`:

```
$ rpcgen rusuarios.x
$ ls
  rusuarios_clnt.c  rusuarios.h  rusuarios_svc.c  rusuarios.x
```

Como se puede observar, `rpcgen` ha generado el resguardo del cliente (`rusuarios_clnt.c`), el del servidor (`rusuarios_svc.c`) y un fichero de cabecera (`rusuarios.h`) que debería usarse en ambos lados de la comunicación (en este ejemplo, no se ha generado un fichero con extensión `.xdr` al usarse sólo tipos de datos básicos). En los ficheros de ejemplo que acompañan a esta guía, para evitar tener que copiar los respectivos resguardos y el fichero de cabecera, se han definido enlaces simbólicos a los mismos en los directorios que contienen el software del cliente (directorio`caso1_erroneo/cliente`) y del servidor (directorio `caso1_erroneo/servidor`).

## Implementación del servidor

La mejor manera de comenzar la programación del servidor es usar el mandato `rpcgen` sobre el fichero IDL pero esta vez usando la opción `-Ss`, que genera un esqueleto del código del servidor:

```
$ rpcgen -Ss rusuarios.x > ../servidor/rusuariosd.c
```

El contenido inicial del fichero del servidor (al que hemos llamado `rusuariosd.c`) sería el siguiente:

```
#include "rusuarios.h"

int *
obtener_uid_1_svc(char **argp, struct svc_req *rqstp)
{
     static int  result;

     /*
      * insert server code here
      */

     return &result;
}

char **
obtener_nombre_1_svc(int *argp, struct svc_req *rqstp)
{
     static char * result;

     /*
      * insert server code here
      */

     return &result;
}
```

En ese código generado automáticamente, se pueden resaltar varios aspectos:

- Las funciones de servicio tienen el mismo nombre que se les dio en el fichero IDL (`.x`), pero en minúsculas y con un sufijo que está compuesto por: `_N_svc`, donde `N` se corresponde con el número de versión.
- El primer parámetro que recibe cada función se corresponde con el definido en el fichero IDL, pero con un nivel de indirección:
  - El tipo *string* especificado en la primera función, que corresponde al tipo `char *` de C puesto que C no soporta directamente ese tipo (recuerde que el lenguaje de IDL de las RPC de Sun/ONC no es directamente C, sino una extensión del mismo), se ha convertido en un tipo `char **`.
  - De manera similar, el entero definido en la segunda función de servicio se ha transformado en un parámetro de tipo `int *`.
- Ambas funciones reciben un segundo parámetro que incluye información sobre el cliente que realiza la petición (por ejemplo, información de autenticación). A efectos del desarrollo de las prácticas de la asignatura, ignoraremos la información contenida en ese parámetro.
- De manera similar a lo que ocurre con el primer parámetro de cada función, el resultado corresponde a lo definido en el fichero IDL, pero con un nivel de indirección añadido: de `int` a `int *` y de `string` (es decir, `char *` en C) a `char **`.
- En el código generado automáticamente, en cada función aparece una variable estática (`result`) del tipo correspondiente al definido en el fichero IDL, y tal que la función termina retornando su dirección como valor de retorno. Este artificio merece una pequeña explicación. Hay que tener en cuenta que cuando se completa la ejecución de la función de servicio, el resguardo del servidor toma control y

procede a *empaquetar* el resultado. Si la función de servicio terminara retornando la dirección de una variable local *automática* (es decir, residente en el registro de activación almacenado en la pila), se produciría un error puesto que esa variable desaparece al completarse la función de servicio. Al definir la variable local como estática (es decir, como una variable que no reside en la pila y cuyo valor se mantiene después de completarse la función), se elimina el problema.

Sobre ese esqueleto, se pasa directamente a programar los servicios, usando las llamadas al sistema de UNIX `getpwnam` y `getpwuid`, respectivamente.

```
#include <string.h>
#include <pwd.h>
#include <sys/types.h>
#include "rusuarios.h"

int *
obtener_uid_1_svc(char **argp, struct svc_req *rqstp)
{
     static int  result;
     struct passwd *desc_usuario;

     result=-1;
     desc_usuario=getpwnam(*argp);
     if (desc_usuario)
          result=desc_usuario->pw_uid;

     return &result;
}

char **
obtener_nombre_1_svc(int *argp, struct svc_req *rqstp)
{
     static char * result;
     struct passwd *desc_usuario;

     desc_usuario=getpwuid(*argp);
     if (desc_usuario)
          result=desc_usuario->pw_name;
     else
          result=NULL;

     return &result;
}
```

Nótese que en la primera función, en caso de error, se ha devuelto un valor de `-1`, mientras que en la segunda se ha retornado un valor `NULL` (más adelante, veremos que eso es erróneo).

En este punto, se puede compilar y ejecutar el servidor. Cuando ya esté arrancado, puede usar el mandato `rpcinfo` para ver cómo el servidor se ha dado de alta en el `portmap` (o `binder`):

```
$ rpcinfo -p
 programa vers proto   puerto
```

```
      ..........................
      666666666    1   udp  60659
         666666666    1   tcp  60674
      ..........................
```

En la salida generada por el mandato, puede observar los puertos TCP y UDP que usa el servidor (por defecto, un servidor en este entorno da servicio a través de ambos protocolos) que podrán ser diferentes cada vez que se vuelva a ejecutar. Asimismo, se puede probar si está listo para dar servicio usando los siguientes mandatos (el primero prueba el servicio TCP y el segundo el UDP):

```
$ rpcinfo -t localhost 666666666 1
 el programa 666666666 versión 1 está listo y a la espera
$ rpcinfo -u localhost 666666666 1
 el programa 666666666 versión 1 está listo y a la espera
```

## Implementación de clientes

De manera similar a lo que ocurre con el servidor, puede sernos útil comenzar a programar un cliente usando la salida generada por el mandato `rpcgen` aplicado al fichero IDL pero esta vez usando la opción `-Sc`, que genera un fragmento de código que muestra la manera de invocar a cada una de las funciones de servicio. Tomando como punto de partida ese código de muestra, se han desarrollado dos clientes, cada uno de los cuales usa uno de los servicios implementados.

El cliente `getuid.c` hace uso del servicio que retorna el `uid` dado un nombre de usuario para obtener esa información de todos los nombres de usuario que recibe como argumentos de la línea de mandatos (excepto `argv[1]` donde se especificará la máquina donde ejecuta el servidor).

```
#include <stdio.h>
#include "rusuarios.h"

int main (int argc, char *argv[]) {
     CLIENT *clnt;   // PUNTO 1
     int  *result;
     int i;

     if (argc < 3) {
          printf ("usage: %s server_host nombre...\n", argv[0]);
          exit (1);
     }
     // PUNTO 2
     clnt = clnt_create (argv[1], RUSUARIOS_SERVICIO, RUSUARIOS_VERSION,
"tcp");
     if (clnt == NULL) { // PUNTO 3
          clnt_pcreateerror (argv[1]);
          exit (1);
     }
     for (i=2; i<argc; i++) {
          result = obtener_uid_1(&argv[i], clnt); // PUNTO 4
          if (result == NULL) // PUNTO 5
```

```
                clnt_perror (clnt, "error en la llamada");
            else if (*result == -1)  // PUNTO 6
                printf("%s: no existe ese usuario\n", argv[i]);
            else
                printf("%s: %d\n", argv[i], *result);
        }
        clnt_destroy (clnt); // PUNTO 7
        exit (0);
}
```

A continuación, se explican los puntos más relevantes de este código de cliente (identificados en el listado como comentarios con el formato `PUNTO X`):

1. Hay que declarar una variable de tipo `CLIENT *` para almacenar el descriptor que retorna la llamada `clnt_create`, el cual se usará posteriormente a la hora de invocar las RPCs.
2. La llamada `clnt_create` realiza todo el proceso de *enlace* con el servidor (su localización y la conexión con el mismo). En ella el cliente debe especificar los siguientes parámetros:
   - Nombre de la máquina donde ejecuta el servidor: en el ejemplo se obtiene del primer argumento de la línea de mandatos.
   - Nombre del servicio: `RUSUARIOS_SERVICIO`.
   - Versión del servicio: `RUSUARIOS_VERSION`.
   - Protocolo: en el ejemplo, TCP.
3. En caso de que se produzca un error en el enlace con el servidor, la llamada `clnt_create` devuelve un valor nulo, y se puede usar la función `clnt_pcreateerror` para imprimir más información sobre el error.
4. La función de servicio tiene el mismo nombre que se le dio en el fichero IDL (`.x`), pero en minúsculas y con un sufijo que está compuesto por: `_N`, donde `N` se corresponde con el número de versión. El primer parámetro que se especifica en la invocación de la función de servicio se corresponde con el definido en el fichero IDL, pero con un nivel de indirección: en vez de un `string`(es decir, `char *` en C) se le pasa un `char **`. El segundo parámetro es el descriptor devuelto por la llamada `clnt_create`. En cuanto al valor de retorno, corresponde al definido en el fichero IDL, pero nuevamente con un nivel de indirección añadido: de `int` a `int *`.
5. En caso de que se produzca algún tipo de error en la invocación de la RPC, se devuelve un valor nulo, y se puede usar la función `clnt_perror` para imprimir más información sobre el error.
6. Para acceder al valor devuelto por la RPC, hay que, evidentemente, deshacer ese nivel de indirección (`*result`).
7. La llamada `clnt_destroy` rompe el *enlace* con el servidor.

En este momento ya se puede compilar y ejecutar este programa cliente:

```
$ ./getuid localhost root nadie fperez
  root: 0
  nadie: no existe ese usuario
  fperez: 1001
```

El segundo cliente (`getname.c`) es similar al primero pero haciendo uso del segundo servicio, que retorna el nombre de usuario asociado a un `uid` dado.

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

```
int main (int argc, char *argv[]) {
     CLIENT *clnt;
     char  **result;
     int i;
     int uid;

     if (argc < 3) {
          printf ("usage: %s server_host uid...\n", argv[0]);
          exit (1);
     }
     clnt = clnt_create (argv[1], RUSUARIOS_SERVICIO, RUSUARIOS_VERSION,
"tcp");
     if (clnt == NULL) {
          clnt_pcreateerror (argv[1]);
          exit (1);
     }
     for (i=2; i<argc; i++) {
          uid=atoi(argv[i]);
          result = obtener_nombre_1(&uid, clnt);
          if (result == NULL)
               clnt_perror (clnt, "error en la llamada");
          else if (*result)
               printf("%d: %s\n", uid,
                    *result);
          else
               printf("%d: no existe ese UID\n", uid);
     }
     clnt_destroy (clnt);
     exit (0);
}
```

Observe el uso de la expresión `*result` para acceder al *string* que se retorna como resultado del servicio. Intentemos ejecutar este programa cliente:

```
$ ./getname localhost 0 877 1001
  0: root
  error en la llamada: RPC: Can't decode result
  1001: fperez
```

Se ha producido un error cuando se trata un `uid` que no corresponde a ningún usuario en la máquina donde ejecuta el servidor. ¿A qué se debe ese error?

## Errores en la gestión de *strings*

El problema está en el manejo del tipo `string` de XDR. Cuando se usa este tipo en las declaraciones de un fichero XDR (ya sea como argumento de una función, como valor retornado, o como un campo de una estructura), hay que asegurarse en el código del cliente y/o en el del servidor de que una variable de este tipo siempre apunta a un *string* válido, es decir, a una cadena de 0 ó más caracteres que termina con un carácter nulo (`\0`) adicional.

En el ejemplo, cuando el servidor, dentro del servicio que obtiene el nombre de usuario asociado a un `uid`, encuentra que ese identificador de usuario no existe, realiza la siguiente asignación:

```
result=NULL;
```

Pero un valor nulo no es un *string* válido (para comprobarlo, pruebe a pasárselo a la función `strlen`). Hay que asegurarse de que la variable `result` hace referencia a un *string* válido también en este caso. Una posible solución sería asignarle un *string* vacío:

```
result="";
```

Con esa estrategia, habría que modificar el código del cliente afectado por ese cambio (sólo se muestra el fragmento afectado; consúltese el fichero `caso1_OK/cliente/getname.c` para ver el código completo):

```
result = obtener_nombre_1(&uid, clnt);
if (result == NULL)
     clnt_perror (clnt, "error en la llamada");
else if (strlen(*result)!=0)
     printf("%d: %s\n", uid,
          *result);
else
     printf("%d: no existe ese UID\n", uid);
```

Nótese como en este caso para determinar si el `uid` no se corresponde con ningún nombre de usuario, comprobamos con `strlen` si el *string* retornado (`*result`) tiene longitud 0.
Con esa modificación, el servicio parece funcionar correctamente. Sin embargo, hay un error latente en el mismo.

Si revisamos el manual de la función `getpwuid`, observamos que explica que una llamada a esta función puede sobreescribir los datos devueltos por una llamada previa. Para entenderlo mejor, podemos usar el siguiente ejemplo, que, obviamente, no tiene nada que ver con las RPC.

```
#include <stdio.h>
#include <pwd.h>
int main() {
        struct passwd *desc_usuario;

        desc_usuario=getpwuid(0);
        getpwuid(1);
        printf("UID 0: %s\n", desc_usuario->pw_name);
        return 0;
}
```

Al ejecutar esa prueba, el nombre de usuario que se imprime no es el del *root*, sino el correspondiente al identificador de usuario 1.

En la función de servicio desarrollada (`obtener_nombre_1_svc`) sólo hay una llamada a `getpwuid`, pero el propio resguardo del servidor podría también hacer esa llamada, sobreescribiendo los datos obtenidos. Por tanto, lo más razonable es crear una copia en el *heap*:

```
     if (desc_usuario)
          result=strdup(desc_usuario->pw_name);
```

Sin embargo, ahora surge un problema adicional: ¿quién se encarga de liberar la memoria dinámica reservada por `strdup` (recuerde que `strdup` realiza un `malloc`)? Téngase en cuenta que la memoria dinámica no se puede liberar al final de la función, puesto que el resguardo del servidor debe tener acceso a los datos almacenados en la misma cuando toma control al completarse la ejecución de la función.

## Uso de `xdr_free`

Para resolver este problema, el sistema de RPC de Sun/ONC ofrece la función `xdr_free`. Si una función de servicio requiere el uso de memoria dinámica, para evitar *goteras de memoria*, debe incluir en su parte inicial una llamada a `xdr_free` especificando cuál es el tipo del resultado devuelto por la función, así como la dirección de la variable estática que almacena el resultado de la invocación previa de la función:

```
char **
obtener_nombre_1_svc(int *argp, struct svc_req *rqstp)
{
     static char * result;
     struct passwd *desc_usuario;

     xdr_free((xdrproc_t)xdr_string, (char *)&result);
     ..................................
```

Téngase en cuenta que siempre se debe llamar a `xdr_free` pasándole el tipo y la dirección de la variable estática que almacena el resultado, incluso aunque en algunos casos, la memoria dinámica no esté asociada directamente a la variable que almacena el resultado, como sí ocurre en este ejemplo, sino a un campo interno de la misma, como veremos en posteriores ejemplos.

Falta un último detalle para completar una solución válida. Al usar `xdr_free` al principio de la función `obtener_nombre_1_svc`, hay que asegurarse de que la variable `result` siempre hace referencia a información almacenada en el *heap*. Eso no ocurre si en la invocación previa el `uid` no existía, ya que en ese caso `result` apuntará a un literal vacío de tipo *string* (`""`) almacenado de forma estática.

Para resolver este problema, se puede usar `strdup` también en el caso de que el `uid` no exista:

```
          result=strdup("");
```

Con todas estas modificaciones, el servidor (fichero `caso1_OK/servidor/rusuariosd.c`) queda de la siguiente forma (sólo se incluye la parte correspondiente al servicio `obtener_nombre_1_svc`):

```
char **
obtener_nombre_1_svc(int *argp, struct svc_req *rqstp)
{
```

```
     static char * result;
     struct passwd *desc_usuario;

     xdr_free((xdrproc_t)xdr_string, (char *)&result);

     desc_usuario=getpwuid(*argp);
     if (desc_usuario)
          result=strdup(desc_usuario->pw_name);
     else
          result=strdup("");

     return &result;
}
```

## Segundo caso: servicios con tipos de datos compuestos

Las RPC de Sun/ONC sólo permiten definir un valor de entrada y uno de salida (el valor de retorno) por cada función de servicio. Si una determinada función de servicio requiere pasar y/o devolver varios datos, es necesario definir estructuras que engloben a esos datos, y que se usarán como parámetro de entrada y/o valor de retorno.
Para practicar con esta característica, vamos a extender el servicio desarrollado como ejemplo incluyendo dos nuevas funciones de servicio:

- Una función que reciba dos `uid` y compruebe si pertenecen al mismo grupo. Requiere, por tanto, pasarle dos argumentos, por lo que habrá que definir una estructura que los contenga.
- Una función que recibe un nombre de usuario y retorna su `uid` y su `gid`. Necesita, por consiguiente, devolver dos valores, lo que requiere definir una estructura que los incluya.

La nueva versión del fichero IDL que incluye estas dos nuevas funciones de servicio sería la siguiente:

```
struct pareja_uids {
     int uid1;
     int uid2;
};
struct respuesta_ugid  {
        int uid;
        int gid;
};
program RUSUARIOS_SERVICIO {
     version RUSUARIOS_VERSION {
          int OBTENER_UID(string nombre)=1;
          string OBTENER_NOMBRE(int uid)=2;
          bool MISMO_GID(pareja_uids)=3;
          respuesta_ugid OBTENER_UGID(string nombre)=4;
     }=1;
}=666666666;
```

Observe la definición de las dos estructuras: la primera para recoger los dos parámetros de entrada de la tercera función de servicio y la segunda estructura para contener los dos valores retornados por la cuarta. Asimismo, conviene destacar el uso del tipo booleano específico de XDR (`bool`).
Si aplicamos el mandato `rpcgen` a este fichero IDL, podemos comprobar que aparece un nuevo tipo de

fichero (`rusuarios_xdr.c`):

```
     $ rpcgen rusuarios.x
     $ ls
       rusuarios_clnt.c  rusuarios.h  rusuarios_svc.c  rusuarios.x
rusuarios_xdr.c
```

Este fichero, requerido tanto por el cliente como por el servidor, contiene la lógica para el *marshalling/unmarshalling* de los tipos definidos por el programador (en el ejemplo del primer caso no se generaba este fichero puesto que no había tipos definidos por el programador).
El siguiente listado muestra el código del servidor (fichero `caso2/servidor/rusuariosd.c`), incluyendo únicamente la parte correspondiente a los dos nuevos servicios):

```
#include <string.h>
#include <pwd.h>
#include <sys/types.h>
#include "rusuarios.h"

bool_t *
mismo_gid_1_svc(pareja_uids *argp, struct svc_req *rqstp)
{
     static bool_t  result;
     struct passwd *desc_usuario;
     int gid1, gid2;

     result=FALSE;
     desc_usuario=getpwuid(argp->uid1);
     if (desc_usuario) {
          gid1=desc_usuario->pw_gid;
          desc_usuario=getpwuid(argp->uid2);
          if (desc_usuario) {
               gid2=desc_usuario->pw_gid;
               result=(gid1==gid2);
          }
     }
     return &result;
}

respuesta_ugid *
obtener_ugid_1_svc(char **argp, struct svc_req *rqstp)
{
        static respuesta_ugid  result;
        struct passwd *desc_usuario;

        result.uid=-1;
        result.gid=-1;
        desc_usuario=getpwnam(*argp);
        if (desc_usuario) {
                result.uid=desc_usuario->pw_uid;
```

```
                result.gid=desc_usuario->pw_gid;
        }
        return &result;
}
```

Nótese que el manejo de las estructuras es igual que el que se realiza en cualquier programa en C, puesto que hay una correspondencia directa entre el tipo `struct` de XDR y el de C.
A continuación, se muestra un programa cliente (fichero `caso2/cliente/getugid`) del servicio que devuelve el `uid` y `gid` de un usuario.

```
#include <stdio.h>
#include "rusuarios.h"

int main (int argc, char *argv[]) {
     CLIENT *clnt;
     respuesta_ugid  *result;
     int i;

     if (argc < 3) {
          printf ("usage: %s server_host nombre...\n", argv[0]);
          exit (1);
     }
     clnt = clnt_create (argv[1], RUSUARIOS_SERVICIO, RUSUARIOS_VERSION,
"tcp");
     if (clnt == NULL) {
          clnt_pcreateerror (argv[1]);
          exit (1);
     }
     for (i=2; i<argc; i++) {
          result = obtener_ugid_1(&argv[i], clnt);
          if (result == NULL)
               clnt_perror (clnt, "error en la llamada");
          else if ((result->uid == -1)||(result->gid == -1))
               printf("%s: no existe ese usuario\n", argv[i]);
          else
               printf("%s: UID %d GID %d\n", argv[i], result->uid,
                         result->gid);
     }
     clnt_destroy (clnt);
     exit (0);
}
```

El siguiente programa cliente muestra el uso del servicio que comprueba si dos `uid` pertenecen al mismo grupo. Dado que hasta este momento todos los programas utilizaban un único servicio, en este ejemplo, un poco más complejo que los precedentes, se ilustra el uso de varios servicios. El programa recibe como argumentos `uids` e imprime parejas (nombre de usuario, `uid`) agrupadas (un grupo por linea) por aquéllas que tienen asociado el mismo `gid`.

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

```
#include "rusuarios.h"

int imprime_nombre(int uid, CLIENT *cl) {
     int ok=FALSE;
     char  **result_nombre;

     result_nombre = obtener_nombre_1(&uid, cl);
     if (result_nombre == NULL)
          clnt_perror (cl, "error en la llamada");
     else if (strlen(*result_nombre) == 0)
               printf("%d: no existe ese usuario\n", uid);
     else {
          ok=TRUE;
          printf("(%s:%d)", *result_nombre, uid);
     }
     return ok;
}

int main (int argc, char *argv[]) {
     CLIENT *clnt;
     int uid1, uid2;
     pareja_uids  mismo_gid_arg;
     bool_t  *result_mismo;
     int i, j;

     if (argc < 3) {
          printf ("usage: %s server_host uid...\n", argv[0]);
          exit (1);
     }
     clnt = clnt_create (argv[1], RUSUARIOS_SERVICIO, RUSUARIOS_VERSION,
"tcp");
     if (clnt == NULL) {
          clnt_pcreateerror (argv[1]);
          exit (1);
     }
     for (i=2; i<argc; i++) {
          if (argv[i]) {
               uid1=atoi(argv[i]);
               if (!imprime_nombre(uid1, clnt))
                    continue;
               mismo_gid_arg.uid1=uid1;
               for (j=i+1; j<argc; j++) {
                    if (argv[j]) {
                         uid2=atoi(argv[j]);
                         mismo_gid_arg.uid2=uid2;
                         result_mismo=mismo_gid_1(&mismo_gid_arg,clnt);
                         if (result_mismo == NULL)
                              clnt_perror (clnt, "error en la llamada");
                         else if (*result_mismo) {
                              imprime_nombre(uid2, clnt);
                              argv[j]=NULL;
                         }
```

```
                    }
               }
               printf("\n");
          }
     }
     clnt_destroy (clnt);
     exit (0);
}
```

# Tercer caso: uso de tipo `union` de XDR

La solución usada en el segundo servicio (`OBTENER_NOMBRE` para indicar que ha habido en error (el `uid` no corresponde con ningún usuario) es un poco artificiosa: se ha usado un cierto valor (un *string*vacío) que no puede aparecer si el servicio procede correctamente. A veces, esa estrategia puede ser difícil de emplear o demasiado forzada.
En esta sección, se plantea una solución alternativa basada en el tipo `union` de XDR, que es distinto del tipo `union` de C. El tipo de XDR se parece más a los registros variantes presentes en lenguajes como Ada, en los que hay un campo discriminante que determina cuál es el contenido del registro en cada caso. Esto nos va a permitir también analizar cómo se manejan los tipos XDR que no corresponden directamente a tipos de C.

A continuación, se incluye una versión del fichero IDL (`caso3/idl/rusuarios.x`) que usa este tipo de datos específico de XDR.

```
union respuesta_nombre switch(bool existe){
     case FALSE: void;
     case TRUE: string nombre<>;
};
program RUSUARIOS_SERVICIO {

     version RUSUARIOS_VERSION {
          int OBTENER_UID(string nombre)=1;
          respuesta_nombre OBTENER_NOMBRE(int uid)=2;
          bool MISMO_GID(pareja_uids)=3;
          respuesta_ugid OBTENER_UGID(string nombre)=4;
     }=1;
}=666666666;
```

A la hora de programar el servidor y los posibles clientes, es necesario averiguar cómo se ve el tipo `union` de XDR desde el código C. Para ello, es recomendable editar el fichero de cabecera (`.h`), generado automáticamente por `rpcgen`, para poder ver esa correspondencia.

A continuación, se muestra el fragmento del fichero `rusuarios.h` que contiene la definición que nos interesa:

```
struct respuesta_nombre {
     bool_t existe;
```

```
     union {
          char *nombre;
     } respuesta_nombre_u;
};
```

Nótese que se ha generado automáticamente un nivel más de anidamiento y que, por tanto, para acceder al nombre usando la variable `result` habrá que especificar lo siguiente:`result.respuesta_nombre_u.nombre`.
El siguiente listado muestra el código del servidor (fichero `caso3/servidor/rusuariosd.c`), incluyendo únicamente la parte correspondiente al servicio `obtener_nombre_1_svc`):

```
respuesta_nombre *
obtener_nombre_1_svc(int *argp, struct svc_req *rqstp)
{
        static respuesta_nombre  result;
     struct passwd *desc_usuario;

     xdr_free((xdrproc_t)xdr_respuesta_nombre, (char *)&result);

     desc_usuario=getpwuid(*argp);
     if (desc_usuario) {
          result.existe=TRUE;
          result.respuesta_nombre_u.nombre=strdup(desc_usuario-
>pw_name);
     }
     else
          result.existe=FALSE;

     return &result;
}
```

Observe el uso de `xdr_free`, que se aplica a la variable `result`, incluso aunque la memoria dinámica esté asociada al campo `nombre`.
A continuación, se lista el cliente afectado por el cambio (fichero `caso3/cliente/getname.c`).

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

int main (int argc, char *argv[]) {
     CLIENT *clnt;
     respuesta_nombre  *result;
     int i;
     int uid;

     if (argc < 3) {
          printf ("usage: %s server_host uid...\n", argv[0]);
          exit (1);
```

```
     }
     clnt = clnt_create (argv[1], RUSUARIOS_SERVICIO, RUSUARIOS_VERSION,
"tcp");
     if (clnt == NULL) {
          clnt_pcreateerror (argv[1]);
          exit (1);
     }
     for (i=2; i<argc; i++) {
          uid=atoi(argv[i]);
          result = obtener_nombre_1(&uid, clnt);
          if (result == NULL)
               clnt_perror (clnt, "error en la llamada");
          else if (result->existe == FALSE)
               printf("%d: no existe ese UID\n", uid);
          else
               printf("%d: %s\n", uid,
                    result->respuesta_nombre_u.nombre);
     }
     clnt_destroy (clnt);
     exit (0);
}
```

Dado que el cliente `agrupa_por_gids` también usa ese servicio, también es necesario modificarlo, aunque no se incluye el código en este documento (véase el fichero`caso3/cliente/agrupa_por_gids.c`).

# Cuarto caso: uso de vectores de tamaño variable de XDR

El otro tipo de datos específico de XDR, que no tiene una proyección directa sobre un tipo del lenguaje C, es el que corresponde con los vectores de longitud variable (`tipo vector<>`). Puede ser interesante usar este tipo cuando hay que gestionar un número variable de datos (por ejemplo, el contenido de un fichero). Si se usara un vector estático (`tipo vector[MAX]`, que tiene correspondencia directa en C), habría que fijar un tamaño máximo y siempre se gastaría el ancho de banda y el coste de procesamiento correspondientes a ese tamaño máximo.
Para ilustrar este mecanismo, vamos a extender el servicio con una nueva función que recibe una lista de `uids` y retorna el conjunto de `gids` correspondientes. A continuación, se incluye el nuevo fichero IDL (`caso4/idl/rusuarios.x`) que usa vectores de tamaño variable.

```
struct pareja_uids {
     int uid1;
     int uid2;
};
struct respuesta_ugid  {
        int uid;
        int gid;
};
union respuesta_nombre switch(bool existe){
     case FALSE: void;
     case TRUE: string nombre<>;
};
```

```
struct arg_gids  {
        int uids<>;
};
struct respuesta_gids {
        int gids<>;
};
program RUSUARIOS_SERVICIO {
     version RUSUARIOS_VERSION {
          int OBTENER_UID(string nombre)=1;
          respuesta_nombre OBTENER_NOMBRE(int uid)=2;
          bool MISMO_GID(pareja_uids)=3;
          respuesta_ugid OBTENER_UGID(string nombre)=4;
          respuesta_gids OBTENER_GIDS(arg_gids lista_uids)=5
     }=1;
}=666666666;
```

A la hora de programar el servidor y los posibles clientes, es necesario averiguar cómo se ve este nuevo tipo desde el código C. Para ello, es recomendable editar el fichero de cabecera (`.h`), generado automáticamente por `rpcgen`, para poder ver esa correspondencia.
A continuación, se muestra el fragmento del fichero `rusuarios.h` que contiene las definiciones que nos interesan:

```
struct arg_gids {
     struct {
          u_int uids_len;
          int *uids_val;
     } uids;
};
typedef struct arg_gids arg_gids;

struct respuesta_gids {
     struct {
          u_int gids_len;
          int *gids_val;
     } gids;
};
typedef struct respuesta_gids respuesta_gids;
```

Nótese que el vector se ha proyectado en un `struct` de C con dos campos: el primero (`gids_len`) debe contener el tamaño del vector, mientras que el segundo (`gids_val`) hará referencia a los datos del vector.
El siguiente listado muestra el código del servidor (fichero `caso4/servidor/rusuariosd.c`), incluyendo únicamente la parte correspondiente al nuevo servicio:

```
respuesta_gids *
obtener_gids_1_svc(arg_gids *argp, struct svc_req *rqstp)
{
     static respuesta_gids  result;
     struct passwd *desc_usuario;
```

```
     int tam, i;

     xdr_free((xdrproc_t)xdr_respuesta_gids, (char *)&result);

     tam= argp->uids.uids_len;
     result.gids.gids_val=malloc(sizeof(int) * tam);
     result.gids.gids_len=tam;
     for (i=0; i<tam; i++) {
          desc_usuario=getpwuid(argp->uids.uids_val[i]);
          if (desc_usuario)
               result.gids.gids_val[i]=desc_usuario->pw_gid;
          else
               result.gids.gids_val[i]=-1;
     }
     return &result;
}
```

Nótese el uso de `xdr_free` sobre la variable `result` para liberar la memoria dinámica reservada en cada invocación. Asimismo, observe que hay que rellenar adecuadamente el campo `gids_len`.
A continuación, se muestra un programa cliente (fichero `caso4/cliente/getgids`) del nuevo servicio.

```
#include <stdio.h>
#include "rusuarios.h"

int main (int argc, char *argv[]) {
     CLIENT *clnt;
     respuesta_gids  *result;
     arg_gids  arg;
     int i, tam;

     if (argc < 3) {
          printf ("usage: %s server_host uid...\n", argv[0]);
          exit (1);
     }
     clnt = clnt_create (argv[1], RUSUARIOS_SERVICIO, RUSUARIOS_VERSION,
"tcp");
     if (clnt == NULL) {
          clnt_pcreateerror (argv[1]);
          exit (1);
     }
     tam=argc-2;
     arg.uids.uids_len=tam;
     arg.uids.uids_val=malloc(tam*sizeof(int));

     for (i=0; i<tam; i++)
          arg.uids.uids_val[i]=atoi(argv[i+2]);

     result = obtener_gids_1(&arg, clnt);
     if (result == NULL)
          clnt_perror (clnt, "error en la llamada");
```

```
     else  {
          tam=result->gids.gids_len;
          for (i=0; i<tam; i++)
               if (result->gids.gids_val[i] == -1)
               printf("%d: no existe ese usuario\n",
arg.uids.uids_val[i]);
          else
               printf("UID %d: GID %d\n", arg.uids.uids_val[i],
                    result->gids.gids_val[i]);
     }

     clnt_destroy (clnt);
     exit (0);
}
```