PRÁCTICA 2

Llamada remota a procedimiento (RPC)

1.1 Introducción

La independencia del transporte de RPC aísla a las aplicaciones de los elementos físicos y lógicos de los mecanismos de comunicaciones de datos y permite a la aplicación utilizar varios transportes. RPC permite a las aplicaciones de red utilizar llamadas a procedimientos que ocultan los detalles de los mecanismos de red subyacentes.

Características de RPC de Sun:

1) Cada procedimiento RPC se identifica unívocamente. Se lleva a cabo mediante:
   a. Un **número de programa** que identifica un grupo de procedimientos remotos relacionados.
   b. Un **número de versión** para que cuando se realicen cambios en un servicio remoto (p.e. añadir un nuevo procedimiento) no tenga que asignarse un nuevo número de programa.
   c. Un **número de procedimiento** único.
2) Selección de red. Permite a los usuarios y aplicaciones seleccionar dinámicamente un transporte de los disponibles. Para esto se utilizan dos mecanismos:
   a. El archivo */etc/netconfig* que contiene una lista de transportes disponibles para la máquina y los identifica por tipos:
      - Campo 1: es el identificador de red del transporte.
      - Campo 2: tipo de transporte, *tpi_clts* son sin conexión, *tpi_cots* con conexión y *tpi_cots_ord* conexión con ordenación.
      - Campo 3: contiene *flags* que identifican el tipo de transporte (p.e. *v* para uno que pueda ser seleccionado).
      - Último campo: contiene uno más módulos enlazables de tiempo de ejecución que contienen las rutinas de traducción de nombres a direcciones.
      - Los transportes *loopback* se utilizan para registrar servicios con *rpcbind* y son sólo transportes locales.

b. La variable de entorno *NETPATH* especifica el orden en el cual la aplicación intenta los transportes disponibles. Su formato es una lista ordenada de identificadores de red separados por dos puntos (p.e. *tcp:udp*). RPC selecciona el transporte según se especifique:

   - *netpath*. Escoge los transportes especificados en la variable de entorno del mismo nombre. Si ésta no se ha definido, utiliza los transportes del archivo */etc/netconfig* que tengan el *flag v* y según el orden en que aparecen.
   - *visible*. Similar al caso anterior cuando utiliza el archivo.
   - *tcp*. Utiliza el protocolo TCP/IP.
   - *udp*. Utiliza el protocolo UDP.

3) Utilidad *rpcbind* que en versiones anteriores se denominaba *portmap*. Los servicios de transporte no proporcionan servicios de búsqueda de direcciones, sólo proporcionan transferencia de mensajes a través de la red. Esta utilidad proporciona el medio para que un cliente pueda obtener la dirección de un programa servidor. Proporciona las siguientes operaciones:

   - Registro de direcciones. Asocia servicios RPC con direcciones. *rpcbind* es el único servicio RPC que tiene dirección conocida (puerto número 111 tanto para TCP como UDP). Un servicio hace su dirección disponible a los clientes cuando la registra con *rpcbind*. Ningún cliente ni servidor puede asumir las direcciones de red de un servicio RPC. La biblioteca RPC (*libnsl*) proporciona una interfaz a todos los procedimientos de *rpcbind*.
   - Borrar registros.
   - Obtener la dirección de un específico programa.
   - Obtener la lista de registros completa. La lista de registros se puede obtener con la orden *rpcinfo*.
   - Realizar una llamada remota para un cliente.
   - Devolver la hora.

4) Varios niveles. Los servicios de RPC se pueden utilizar a diferentes niveles. Los servicios del nivel más bajo no son necesarios cuando se utiliza la utilidad *rpcgen* para generar los programas RPC.

5) Representación externa de datos (XDR). Para que RPC funcione en varias arquitecturas de sistemas requiere una representación de datos estándar. XDR es una descripción de datos independiente de la máquina y un protocolo de codificación. Con XDR, RPC consigue manejar estructuras de datos arbitrarias sin importar la ordenación de *bytes* que haga cada máquina o las cuestiones en el diseño de estructuras.

1.2 rpcgen

Esta utilidad genera módulos interfaz de programas remotos compilando código fuente escrito en el lenguaje RPC. Por defecto, la salida de *rpcgen* es:

- Un archivo cabecera con las definiciones comunes al servidor y al cliente.
- Un conjunto de rutinas XDR que traducen cada tipo de dato definido en el archivo cabecera.
- Un programa *stub* para el servidor y otro para el cliente.

Además, *rpcgen* opcionalmente genera una tabla informe de RPC para comprobar autorizaciones y para invocar rutinas de servicio. Otras opciones incluyen especificar plazos de tiempo para servidores, seleccionar varios transportes, código conforme al ANSI-C pasando argumentos estilo C.

Pasos en el desarrollo de un programa distribuido:

1) Determinar los tipos de datos de todos los argumentos del procedimiento llamado y escribir la especificación del protocolo en el lenguaje RPC. Por ejemplo, para un procedimiento llamado *printmessage()* al cual se le pasa una cadena de caracteres y devuelve un entero, el código fuente para la especificación del protocolo sería:

```
/* Archivo msg.x: Protocolo de impresion de un mensaje remoto */
   program MESSAGEPROG {
     version PRINTMESSAGEVERS {
       int PRINTMESSAGE (string) = 1;
     } = 1;
   } = 0x20000001;
```

Este protocolo declara el procedimiento *PRINTMESSAGE* como procedimiento número 1, en la versión número 1 del programa *MESSAGREPROG*, cuyo número de programa es *0x20000001*. Los números de programa se dan según la siguiente tabla:

0 - 1fffffff	Definidos por Sun
20000000 - 3fffffff	Definidos por los usuarios para programas particulares
40000000 - 5fffffff	Reservados para programas que generan números de programas dinámicamente
60000000 - ffffffff	Reservados para uso futuro

Por convención, los nombres se suelen escribir en mayúsculas y el nombre del archivo que contiene la especificación acaba en *.x*.

El tipo de argumento *string* se corresponde con *char* * de C.

2) Escribir el procedimiento remoto y el programa cliente principal que lo llama.

```
/* msg_proc.c: implementacion del procedimiento remoto */

  #include <stdio.h>
  #include "msg.h"             /* el archivo lo genera rpcgen */

  int *
  printmessage_1(msg, req)
    char **msg;
    struct svc_req *req;  /* detalles de la llamada */

    {
      static int result;   /* es obligatorio que sea estatica */
      FILE *f;
      f = fopen("/dev/console", "w");
      if (f == (FILE *)NULL) {
        result = 0;
        return (&result);
      }
      fprintf(f,"%s\n", *msg);
      fclose(f);
      result = 1;
      return (&result);
    }
```

Observe lo siguiente en la declaración del procedimiento remoto:

- Toma un puntero a un array de caracteres. Esto ocurre cuando no se utiliza la opción *-N* con *rpcgen*. Sin esta opción, los procedimientos remotos siempre se llaman con un único argumento, si es necesario más de uno se pasan en una estructura.
- El segundo parámetro contiene información sobre el contexto de una invocación (programa, versión, procedimiento, un puntero a una estructura que contiene información de transporte, ...). Esta información se hace disponible para el caso de que el procedimiento invocado la requiera para realizar la petición.

- Devuelve un puntero a un entero en lugar del entero. Esto también ocurre cuando no se utiliza la opción *-N* con *rpcgen*. Siempre se declara el resultado como *static*, ya que si se declara local al procedimiento remoto, las referencias a él por el *stub* del servidor son inválidas después de la vuelta del procedimiento.
- Un *_1* se añade al nombre según el número de versión, lo cual permite múltiples versiones del mismo nombre.

```
/* Archivo  rprintmsg.c: programa cliente */
  #include <stdio.h>
  #include "msg.h"

  main(argc, argv)
    int argc;
    char *argv[];
  {
    CLIENT *clnt;
    int *result;
    char *server;
    char *message;
    if (argc != 3) {
      fprintf(stderr, "uso: %s maquina mensaje\n", argv[0]);
      exit(1);
    }
    server = argv[1];
    message = argv[2];

    /* Crea estructura de datos(handle) del proceso
       cliente para el servidor designado     */
    clnt = clnt_create(server, MESSAGEPROG, PRINTMESSAGEVERS,"visible");
    if (clnt == (CLIENT *)NULL) {
      /* No se pudo establecer conexion con el servidor */
      clnt_pcreateerror(server);
      exit(1);
    }

    /* Llamada remota al procedimiento en el servidor */
    result = printmessage_1(&message, clnt);
    if (result == (int *)NULL) {
      /* Ocurrio un error durante la llamada al servidor */
      clnt_perror(clnt, server);
      exit(1);
    }
    if (*result == 0) {
```

```
      /* El servidor fue incapaz de imprimir nuestro mensaje */
      fprintf(stderr, "%s: no pudo imprimir el mensaje\n", argv[0]);
      exit(1);
    }
    printf("Mensaje enviado a %s\n", server);
    clnt_destroy( clnt );
    exit(0);
  }
```

Observe lo siguiente en la declaración del cliente:

- La rutina de biblioteca *clnt_create()* crea una estructura de datos (*handle*) del cliente. Ésta se pasa a la rutina *stub* que llama al procedimiento remoto. Cuando no se van a realizar más llamadas se destruye la estructura de datos con *clnt_destroy()* para conservar los recursos del sistema.
- El último parámetro *visible* de *clnt_create()* especifica cualquier transporte con el *flag v* en el archivo */etc/netconfig*.

3) Generación del archivo de cabecera *msg.h*, *stub* del cliente (*msg_clnt.c*) y *stub* del servidor (*msg_svc.c*) con:

```
rpcgen msg.x
```

4) Compilación de los dos programas, cliente y servidor, y enlazado de cada uno con la biblioteca *libnsl* que contiene todas las funciones de red (incluyendo RPC y XDR):

```
cc rprintmsg.c msg_clnt.c -o rprintmsg -lnsl
cc msg_proc.c msg_svc -o msg_server -lnsl
```

5) Ejecución del servidor y posteriormente del cliente. El servidor generado por rpcgen siempre se ejecutará en segundo plano (background) sin necesidad de invocarlo con &. Además estos servidores pueden ser invocados por monitores de puertos como *inetd*.

1.3 Ejemplo completo de servicio RPC

En el ejemplo de la sección anterior no fueron generadas rutinas XDR (archivo *msg_xdr.c*) por *rpcgen* debido a que el programa sólo utiliza tipos básicos que están incluidos en *libnsl*. Si se hubieran definido tipos de datos en el archivo *.x* si se habrían generado estas rutinas. Estas rutinas

se utilizan para convertir estructuras de datos locales al formato XDR y viceversa. Por cada tipo de dato definido en el archivo *.x*, *rpcgen* genera una rutina con el prefijo *xdr_* que se incluye en el archivo *_xdr.c*. A continuación se presenta un servicio RPC de listado de directorio remoto:

```
/* dir.x : Protocolo de listado de directorio remoto */
  const MAX= 255;       /* longitud maxima de la entrada directorio */
  typedef string nametype<MAX>;               /* entrada directorio */
  typedef struct namenode *namelist;        /* enlace en el listado */

  struct namenode{
    nametype name;        /* nombre de la entrada de directorio */
    namelist next ;                         /* siguiente entrada */
  };

  /* La siguiente union se utiliza para discriminar entre llamadas
   *                 con exito y llamadas con errores               */
  union readdir_res switch (int errno) {
    case 0:
      namelist list; /* sin error: listado del directorio */
    default:
      void;          /* con error: nada                   */
  };

  program DIRPROG {
    version DIRVER {
      readdir_res READDIR(nametype) = 1;
    } =1;
  } = 0x20000155;
```

Se pueden redefinir tipos (como *readdir_res*) usando las palabras reservadas *struct*, *union* y *enum* (en una sección posterior se introduce el lenguaje RPC). *rpcgen* compila uniones RPC en estructuras de C. Utilizando *rpcgen* sobre *dir.x* genera, además de los archivos ya vistos, el archivo *dir_xdr.c*.

Para generar los *stubs*, plantillas, ..., ejecutaremos (todas las opciones de *rpcgen* se comentan en la siguiente sección):

```
turing% rpcgen -NCa dir.x
```

```
/* dir_client.c: codigo del cliente       */
  /*
   * 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 "dir.h"             /* creado por rpcgen */
  extern int errno;

  void
  dirprog_1(char *server,  nametype dir)
  {
    CLIENT *clnt;
    readdir_res  *result;
    namelist nl;

    #ifndef DEBUG
    clnt = clnt_create(server, DIRPROG, DIRVER, "netpath");
    if (clnt == (CLIENT *) NULL) {
      clnt_pcreateerror(server);
      exit(1);
    }
    #endif  /* DEBUG */

    result = readdir_1(dir, clnt);
    if (result == (readdir_res *) NULL) {
      clnt_perror(clnt, server);
      exit(1);
    }

    if (result->errno != 0) {
      errno =result->errno;
      perror(dir);
      exit(1);
    }

    for (nl = result->readdir_res_u.list; nl != NULL; nl = nl->next)
      printf("%s\n", nl->name);

    xdr_free (xdr_readdir_res,  result);

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

```
    exit(0);
  }
   main(int argc, char *argv[])
  {
    char *server;
    nametype dir;
    setbuf(stdout, NULL);
    if (argc < 3) {
      printf("usage:  %s server_host directory\n", argv[0]);
      exit(1);
    }
    server = argv[1];
    dir = argv[2];
    dirprog_1(server, dir);
  }
```

Observe que el código de cliente generado por *rpcgen* no libera la memoria asignada por la llamada RPC, para esto se utiliza *xdr_free()*. Es similar a la función *free()*, excepto en que se pasa la función XDR para el resultado devuelto.

```
/* dir_server.c : codigo del servidor   */
  /*
   * 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 "dir.h"              /* creado por rpcgen */
  #include <dirent.h>

  extern int errno;

  readdir_res *
  readdir_1_svc(nametype dirname, struct svc_req *rqstp)
  {
    DIR *dirp;
    struct dirent *d;
    namelist nl;
    namelist *nlp;

    static readdir_res  result;     /* tiene que ser estatica */
```

```
    dirp = opendir(dirname);
    if  (dirp == (DIR *)NULL) {
      result.errno = errno;
      return (&result);
    }
    /*
     * La siguiente linea de codigo libera la memoria que se asigno
     * (para el resultado) en una ejecucion previa del servidor
     */
    xdr_free(xdr_readdir_res, &result);

    nlp = &result.readdir_res_u.list;
    while (d = readdir(dirp)) {
      nl = *nlp = (namenode *) malloc(sizeof(namenode));
      if (nl == (namenode *) NULL) {
        result.errno =  10;
        closedir(dirp);
        return (&result);
      }
      nl->name = strdup(d->d_name);
      nlp = &nl->next;
    }
    *nlp = (namelist)NULL;
    result.errno = 0;
    closedir (dirp);
    return (&result);
  }
```

Los archivos serán compilados como sigue:

```
turing% cc dir_client.c dir_clnt.c dir_xdr.c -o cliente -lnsl
turing% cc dir_server.c dir_svc.c dir_xdr.c -o servidor -lnsl
```

o como alternativa a lo anterior:

```
turing% make -f makefile.dir
```

Y el programa se ejecutará como sigue:

```
ws1% servidor
turing% cliente ws1 /etc
```

Este ejemplo, también ilustra como resuelve *rpcgen* por si sólo el problema de que pasar punteros del espacio de direcciones de una máquina a otra, en la cual no serían válidos. XDR proporciona en el archivo *_xdr.c* una función filtro especial denominada *xdr_pointer()*, el cual es capaz de seguir cadenas de punteros y codificar el resultado en una cadena de *bytes*. Esto funciona correctamente en algunos tipos recursivos de estructuras de datos tales como listas enlazas y árboles binarios, ya que proporcionan un final mediante punteros nulos, con lo cual son estructuras enlazadas acíclicas. Sin embargo, esta técnica no funciona con estructuras de datos cíclicas tales como listas circulares, listas doblemente enlazadas, etc.

1.4 Opciones de rpcgen

Las opciones de tiempo de compilación son:

- Código estilo C (*-N*). Provoca que *rpcgen* genere código con los argumentos pasados por valor y si hay varios argumentos estos se pasan sin *struct*.

```
/* archivo add.x: Demuestra el paso de argumentos por defecto.
                  Solo se puede pasar un argumento    */
  struct add_arg {
    int first;
    int second;
  }
  program ADDPROG {
    version ADDVER {
      int ADD (add_arg) = 1;
    } = 1;
  } = 0x20000199;
/* archivo add.x: Demuestra el paso de argumento estilo C */
  program ADDPROG {
    version ADDVER {
      int ADD (int, int) = 1;
    } = 1;
  } = 0x20000199;
```

- Archivos plantilla. Genera código ejemplo que puede servir como guía o se puede utilizar directamente rellenando las partes omitidas. Posibilidades:

Opción	Función
`-a`	Genera todos los archivos plantilla
`-Sc`	Genera la plantilla para el cliente
`-Ss`	Genera la plantilla para el servidor
`-Sm`	Genera la plantilla del archivo para la utilidad make

- Código compatible con ANSI-C o SPARCompiler C++ 3.0 (*-C*). Esta opción se suele utilizar junto con la opción *-N*. Observe que los nombres de los procedimientos remotos en el servidor requieren el sufijo *_svc*.

- Código MT-seguro (*-M*). Por defecto el código generado por *rpcgen* no es MT-seguro, ya que utiliza variables globales sin proteger y devuelve resultados mediante variables estáticas. Con esta opción se genera código MT-seguro el cual puede utilizarse en un entorno multi-hebras. La opción se puede utilizar junto con las opciones *-N* y *-C*.

1.5 Referencia del lenguaje RPC

Este lenguaje es una extensión del lenguaje XDR. A continuación se describe su sintaxis junto con algunos ejemplos y el resultado de la compilación de las definiciones de tipos del lenguaje RPC, lo cual se incluye en el archivo de cabecera *.h* de la salida de *rpcgen*.

El lenguaje consiste en una serie de definiciones:

```
<lista-definiciones> ::= <definicion>; | <lista-definiciones>;
<definicion> ::= <definicion-enum> |
                 <definicion-const> |
                 <definicion-typedef> |
                 <definicion-struct> |
                 <definicion-union> |
                 <definicion-program>
```

Definiciones no son lo mismo que declaraciones, no se asigna espacio en una definición (sólo la definición de tipo de uno o varios elementos de datos). Es decir, las variables tendrán que ser declaradas posteriormente en el programa.

1.5.1 Enumeraciones

Tienen la misma sintaxis que las de C:

```
<definicion-enum> ::= enum <ident-enumeracion> "{"
                            <lista-valores-enum>
                      "}"
<lista-valores-enum> ::= <valor-enum> |
                         <valor-enum> , <lista-valores-enum>
<valor-enum> ::= <ident-valor-enum> |
                 <ident-valor-enum> = <valor>
```

A continuación se da un ejemplo de definición y su traducción a C:

```
enum tipocolor {                        enum tipocolor {
   ROJO = 0,                               ROJO = 0,
   VERDE = 1,            --->              VERDE = 1,
   AZUL = 2                                AZUL = 2,
};                                      };
                                        typedef enum tipocolor tipocolor;
```

1.5.2 Constantes

Las constantes simbólicas pueden usarse donde una constante entera se usa.

```
<definicion-const> ::= const <ident-constante> = <entero>
```

Ejemplo:

```
const DOCENA = 12;          --->      #define DOCENA 12
```

1.5.3 Definiciones de Tipos

Tienen exactamente la misma sintaxis que en C:

```
<definicion-typedef> ::= typedef <declaracion>
```

El ejemplo siguiente define un tipo utilizado para declarar cadenas de nombres de archivo que tienen una longitud máxima de 255 caracteres:

```
typedef string tipo_nombref<255>;   --->   typedef char *tipo_nombref;
```

1.5.4 Declaraciones

No se debe confundir declaraciones de variables con declaraciones de tipos. *rpcgen* no soporta declaraciones de variables. Hay cuatro clases de declaraciones. Estas declaraciones tienen que ser parte de un *struct* o un *typedef*; no se pueden encontrar solas:

```
<declaracion> ::= <declaracion-simple> |
                  <declaracion-array-fijo> |
                  <declaracion-array-variable> |
                  <declaracion-puntero>
<declaracion-simple> ::= <ident-tipo> <ident-variable>
```

Ejemplo de declaración simple:

```
tipocolor color;          --->         tipocolor color;
```

```
<declaracion-array-fijo> ::= <ident-tipo>  <ident-variable> "["<valor>"]"
```

Ejemplo de array de tamaño fijo:

```
tipocolor paleta[8];      --->         tipocolor paleta[8];
```

La declaración de arrays variables no tiene sintaxis explicita en C. Se puede especificar un tamaño máximo entre los ángulos. Estas declaraciones se traducen a declaraciones *struct* de C.

```
<declaracion-array-variable>::=<ident-tipo> <ident-variable> "<"valor">" |
                               <ident-tipo> <ident-variable> "<" ">"
```

Ejemplo:

```
int altura<12>;          --->       struct {
                                      u_int altura_len;
                                      int *altura;
                                    } altura;
```

La declaración de punteros es igual a la de C. Los punteros de direcciones no se envían realmente por la red, pero son útiles para enviar tipos de datos recursivos tales como listas o árboles.

```
<declaracion-puntero> ::= <ident-tipo> *<ident-variable>
```

Ejemplo:

```
listaelementos *siguiente;     --->    listaelementos *siguiente;
```

1.5.5 Estructuras

Se declaran igual que en C.

```
<defincion-struct> ::= struct <ident-estructura> "{"
                                <lista-declaraciones>
                       "}"
<lista-declaraciones> ::= <declaracion> ; |
                          <declaracion> ; <lista-declaraciones>
```

Ejemplo:

```
struct coord {                      struct coord
  int x;       --->                   int x;
```

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

1.5.6 Uniones

Las uniones de RPC son diferentes a las uniones de C. Son similares a los registros variantes de PASCAL.

```
<definicion-union>::=union <ident-union> switch (<declaracion-simple>) "{"
                          <lista-case>
                     "}"
<lista-case> ::= case <valor> : <declaracion> ; |
                 case <valor> : <declaracion> ; <lista-case> |
                 default : <declaracion> ;
```

El siguiente es un ejemplo de un tipo devuelto como resultado de una operación de lectura de un dato: si no hay error, devuelve un bloque de datos; en otro caso, no devuelve nada.

```
union resultado_leido switch (int errno) {
      case 0:
        tipodato dato;
      default:
        void;
};
```

Y se traduce a C:

```
struct resultado_leido {
      int errno;
      union {
        tipodato dato;
      } resultado_leido_u;
};
typedef struct resultado_leido resultado_leido;
```

1.5.7 Programas

Los programas RPC se declaran con la siguiente sintaxis:

```
<definicion-program> ::= program <ident-programa> "{"
                                  <lista-versiones>
                            "}" = <valor>;
<lista-versiones> ::= <version> ; |
                      <version> ; <lista-versiones>
<version> ::= version <ident-version> "{"
                      <lista-procedimientos>
              "}" = <valor> ;
<lista-procedimientos> ::= <procedimiento> ; |
                           <procedimiento> ; <lista-procedimientos>
<procedimiento> ::= <ident-tipo>  ( <ident-tipo> ) = <valor>;
```

Cuando se especifica la opción *-N* en *rpcgen*, reconoce la siguiente sintaxis:

```
<procedimiento> ::= <ident-tipo>  ( <lista-ident-tipo> ) = <valor>;
<lista-ident-tipo> ::= <ident-tipo> |
                       <ident-tipo> , <lista-ident-tipo>
```

1.5.8 Casos especiales

- **Lógicos**. La biblioteca RPC utiliza un tipo lógico llamado *bool_t* que es *TRUE* o *FALSE*. Los parámetros declarados de tipo *bool* en el lenguaje RPC se traducen a *bool_t* en el archivo de cabecera de salida de *rpcgen*.

- **Cadenas de caracteres**. En el lenguaje RPC las cadenas de caracteres se declaran con la palabra reservada *string*, y se traduce al tipo *char ** en el archivo de cabecera. El tamaño máximo contenido entre los ángulos especifica el número máximo de caracteres permitidos en la cadena de caracteres (sin contar el carácter *NULL*). El tamaño máximo puede quedar sin implementación, indicando una cadena de caracteres de longitud arbitraria. Ejemplo:

```
string nombre<32>;          --->     char *nombre;
string nombrecompleto<>;    --->     char *nombrecompleto;
```

Observe que cadenas de caracteres *NULL* no pueden pasarse; sin embargo, una cadena de longitud cero sí (sólo con el byte *NULL*).

- **Datos opacos**. Se utilizan para describir datos sin tipo, es decir, secuencias de *bytes* arbitrarios. Pueden declararse como un array bien de longitud fija o variable. Ejemplos:

```
opaque bloquedisco[512];  --->   char bloquedisco[512];
opaque datosf<1024>;      --->   struct {
                                    u_int datosf_len;
                                    char *datosf_val;
                                 } datosf;
```

- ***void***. Declaraciones *void* pueden tener lugar sólo en dos lugares:
  - definiciones de uniones
  - definiciones de programas, como argumento (p.e. indicando que no se pasan argumentos) o resultado de un procedimiento remoto.

1.6 Ejercicio 1

Escriba un programa distribuido, utilizando *rpcgen*, que realice las operaciones aritméticas suma, resta, multiplicación o división sobre enteros, según las siguientes especificaciones:

- El cliente se encargará de filtrar la línea de orden introducida por el usuario la cual podrá tener el siguiente formato:

```
<programa> <maquina> <entero> <operador> <entero>
```

  donde <*operador*> puede ser + | - | *x* | /.

  El programa cliente será el encargado de llamar a la operación correspondiente en el servidor.

- El servidor sólo realizará la evaluación de la expresión y devolverá el resultado.

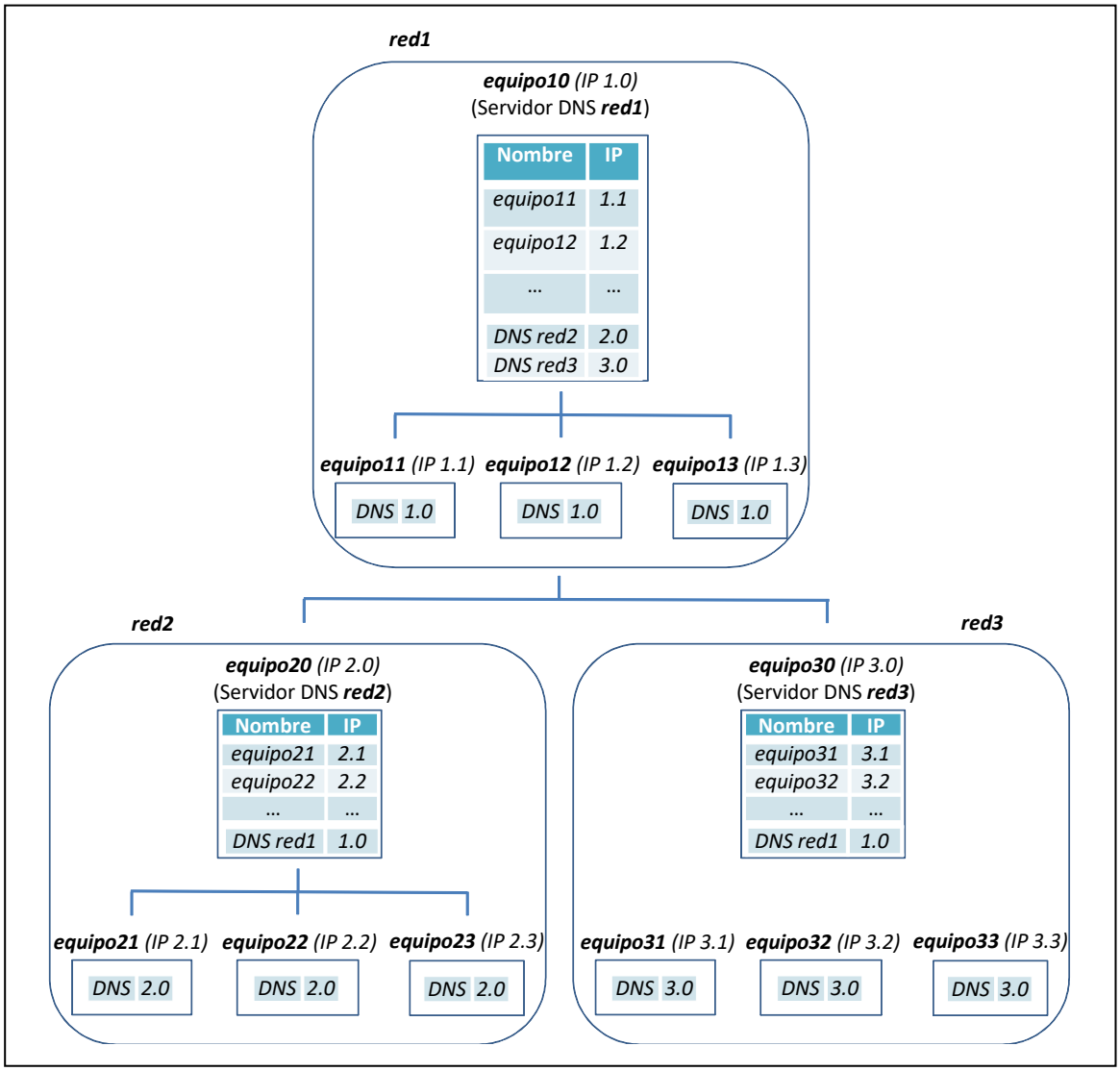
1.7 Ejercicio 2

Utilizando *rpcgen*, programe una simplificación de un servicio DNS (*Domain Name Server*), el cual es un servicio jerárquico organizado en base a subredes para resolver la obtención de la dirección IP de un equipo a partir de su nombre, independientemente de la subred donde se encuentre el equipo.

Cada subred dispone de un servidor DNS que consiste en un **servicio básico de registro** de nombres de equipos y sus correspondientes IPs, los equipos pertenecen a dicha subred (véase más adelante la figura con un ejemplo ilustrativo). En el caso de subredes situadas en los nodos hoja de la jerarquía, dicho servidor conoce y registra la dirección del servidor superior en la jerarquía al que tiene que consultar en caso de que no pueda resolver la obtención de la IP de un equipo que no pertenezca a su subred. En el caso de que una subred sea intermedia en la jerarquía, además de la dirección del servidor superior en la jerarquía, dicho servidor también conoce y registra las direcciones de los servidores de cada una de las subredes que son nodos inferiores en la jerarquía. Ello permite que una petición recibida desde un servidor inferior en la jerarquía sea propagada al servidor de la subred que corresponde al equipo buscado.

Un equipo se ha de nombrar de forma que se especifique nombre de equipo y subred a la que pertenece (e.g. *equipo21.red2*). Un registro consiste en el nombre del equipo junto con su correspondiente IP (ambos campos serán cadenas de caracteres). El servidor DNS tendrá que implementar operaciones de alta y baja de registros de equipos que pertenecen a su subred, y obtención de IPs de equipos dados sus nombres ya sean pertenecientes a su subred o no. Tenga en cuenta que cuando un servidor DNS tiene que resolver la obtención de IP de un equipo que no pertenece a su subred, el mismo servidor es el responsable de transmitir petición al DNS siguiente en la jerarquía, y así sucesivamente hasta que la petición alcance al DNS que mantiene la información. Cada servidor DNS también tendrá registradas de la forma más conveniente las direcciones de los servidores DNS con los que conecta directamente en la jerarquía.

Diseñe e implemente una solución basada en RPC al ejemplo mostrado en la figura siguiente.


red1
equipo10 (IP 1.0)
(Servidor DNS red1)
Nombre
IP
equipo11
1.1
equipo12
1.2
...
...
DNS red2
2.0
DNS red3
3.0
equipo11 (IP 1.1)
equipo12 (IP 1.2)
equipo13 (IP 1.3)
DNS 1.0
DNS 1.0
DNS 1.0
red2
equipo20 (IP 2.0)
(Servidor DNS red2)
Nombre
IP
equipo21
2.1
equipo22
2.2
...
...
DNS red1
1.0
equipo21 (IP 2.1)
equipo22 (IP 2.2)
equipo23 (IP 2.3)
DNS 2.0
DNS 2.0
DNS 2.0
red3
equipo30 (IP 3.0)
(Servidor DNS red3)
Nombre
IP
equipo31
3.1
equipo32
3.2
...
...
DNS red1
1.0
equipo31 (IP 3.1)
equipo32 (IP 3.2)
equipo33 (IP 3.3)
DNS 3.0
DNS 3.0
DNS 3.0