Lección 10
Las RPC de Sun Microsystems

2ª parte: Nivel simplificado y otras funciones

Nivel Simplificado

Sólo se usan cuatro funciones:

En el cliente	En el servidor
`callrpc()`	`registerrpc()`
`clnt_perrno()`	`svc_run()`

Nivel Simplificado. Cliente

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

Nombre de la máquina del servidor

Identificación del procedimiento: Nº de programa, nº de versión y nº de procedimiento

Parámetro que recibe el procedimiento y filtro XDR que se le debe aplicar.

Parámetro que retorna el procedimiento y filtro XDR que se le debe aplicar.

Nivel Simplificado. Cliente

Retorno de la función:

```
enum clnt_stat {
   RPC_SUCCESS = 0,                /* llamada correcta */
   RPC_CANTENCODEARGS = 1,         /* No se pudo codificar los argumentos */
   RPC_CANTDECODERES = 2,          /* No se pudo decodificar el resultado */
   RPC_CANTSEND = 3,               /* Fallo en el envío de la llamada */
   RPC_CANTRECV = 4,               /* Fallo al recibir el resultado */
   RPC_TIMEDOUT = 5,               /* Tiempo de espera excedido */
   RPC_VERSMISMATCH = 6,           /* Versión de la rpc incompatible */
   RPC_AUTHERROR = 7,              /* Fallo de autentificación */
   RPC_PROGUNAVAIL = 8,            /* Programa no disponible */
   RPC_PROGVERSMISMATCH = 9,       /* Versión del programa incompatible */
   RPC_PROCUNAVAIL = 10,           /* Procedimiento no disponible */
   RPC_UNKNOWNHOST = 13,           /* Máquina del servidor desconocida */
   RPC_RPCBFAILURE = 14,           /* El conector falló en la llamada */
   RPC_PROGNOTREGISTERED = 15,     /* Programa no registrado */
};
```

Nivel Simplificado. Cliente

```
void clnt_perrno (
            const enum clnt_stat estado
                );
```

La función envía a la salida estándar un mensaje explicando el error indicado por el parámetro `estado`.

Nivel Simplificado. Servidor

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

Identificación del procedimiento: Nº de programa, nº de versión y nº de procedimiento

Puntero a la función que contiene el código del servicio.

Filtros XDR que se deben aplicar al Parámetro y al retorno.

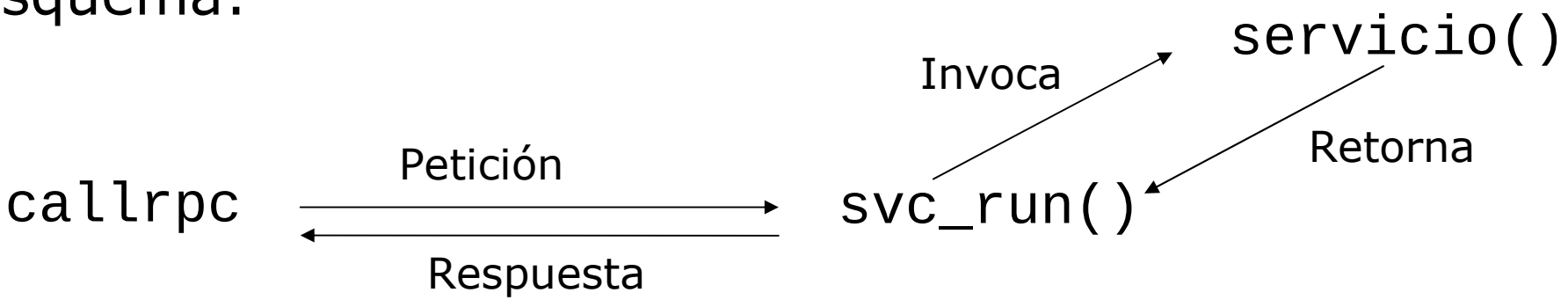
Nivel Simplificado. Servidor

```
void svc_run ();
```

Sin parámetros. No retorna nunca. Es la última función a invocar en el servidor.

Esta función deja residente el servidor y le transfiere el control al *runtime* de las RPC para que reciba las peticiones de servicio de los clientes y las gestione.

Esquema:



Nivel Simplificado. Ejemplo Código

Programa que multiplica por dos un entero.

Definimos unas constantes comunes a cliente y servidor.

simple.h

```
#define DUPLICADOR ((u_long) 0x20821040)
#define PRIMERA ((u_long) 1)
#define DUPLICA ((u_long) 2)
#define ACABADOR ((u_long) 3)
```

Nivel Simplificado. Ejemplo Código

Cliente.c

```
#include <rpc/rpc.h>
#include "simple.h"
main(int narg, char *arg[]) {
  enum clnt_stat estado;
  int leido, result;
  printf("\nIntroduce un numero: ");
  scanf("%d",&leido);
  estado=callrpc(arg[1],DUPLICADOR, PRIMERA, DUPLICA,
                 (xdrproc_t)xdr_int, (char *)&leido,
                 (xdrproc_t)xdr_int, (char *)&result);
  if ( estado != RPC_SUCCESS )   {
      clnt_perrno(estado);
      exit(-1); }
  printf("\n\nEl resultado es:  %d\n",result);
  printf("\nPulsa <return> para acabar\n");
  exit(0);  }
```

Ficheros de cabecera necesarios

Llamada al procedimiento

Control de errores

Uso del resultado

Nivel Simplificado. Ejemplo Código

Servidor.c

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

Ficheros de cabecera necesarios

```
int * duplica (int *valor)
{
  static int resultado;
  resultado= *valor * 2;
  return(&resultado);
}
```

Código fuente del servicio. (Detalles en la siguiente transparencia)

```
main()  {
  int bien;
  bien=registerrpc(DUPLICADOR,PRIMERA,DUPLICA,duplica,xdr_int,xdr_int);
  if ( bien == -1 )    {
      printf("\nError al registrar el procedimiento duplica\n");
      exit(-1);    }
  svc_run();
}
```

Registro del servicio

Llamada al *runtime*

Nivel Simplificado. Ejemplo Código

Detalle del servicio

```
int * duplica (int *valor)
{
  static int resultado;

  resultado= *valor * 2;
  return(&resultado);
}
```

¡Limitación! Un único parámetro de entrada al servicio. Puede ser cualquier tipo de dato (estructuras, etc).

Retorno y parámetro siempre son un puntero al tipo de dato que maneja el filtro XDR.
Si filtro es **`xdr_int`** ➔ tipo de dato es **`int *`**

La variable de retorno siempre tiene que ser declarada de tipo **`static`**

Concurrencia en el servidor

¿Qué ocurre cuando llegan dos peticiones de servicio simultáneamente?

¿Cuál es la política de servicio aplicada?

El servidor atiende una por una las peticiones recibidas. No pasa a la segunda hasta que acaba con la primera.

En el caso de un servicio *pesado* con un gran número de clientes simultáneos ¿Es posible conseguir concurrencia en el servidor?

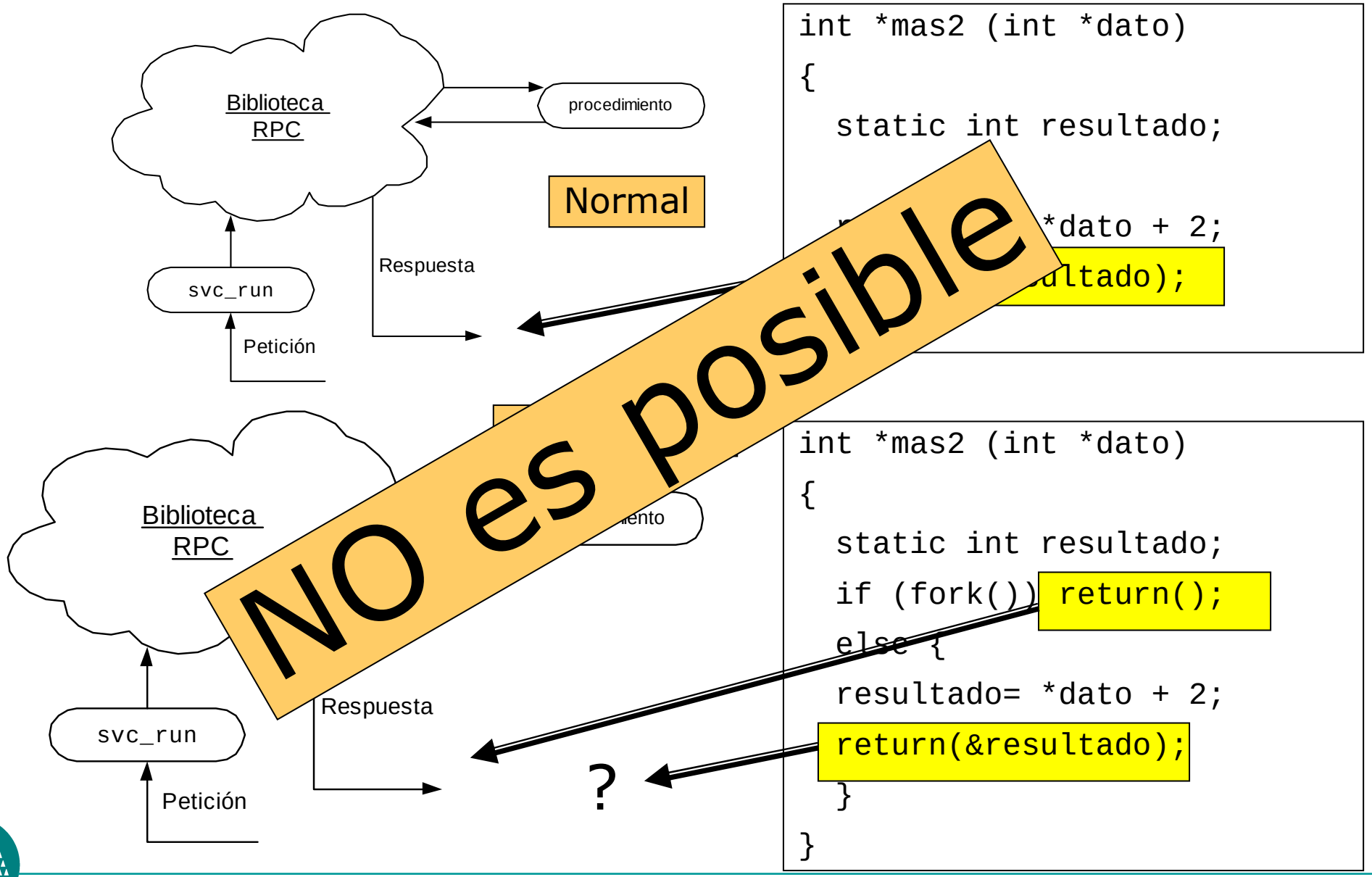
Sí. Usando ***`fork()`****, pero...¿Dónde lo colocamos?*

Nivel simplificado:

Tres funciones: `callrpc, registerrpc, svc_run.`

Única posibilidad: en la rutina de servicio.

Concurrencia en el servidor



Concurrencia en el servidor

Usando `rpcgen`:

En este caso la *nube* está identificada: es el extremo del servidor.

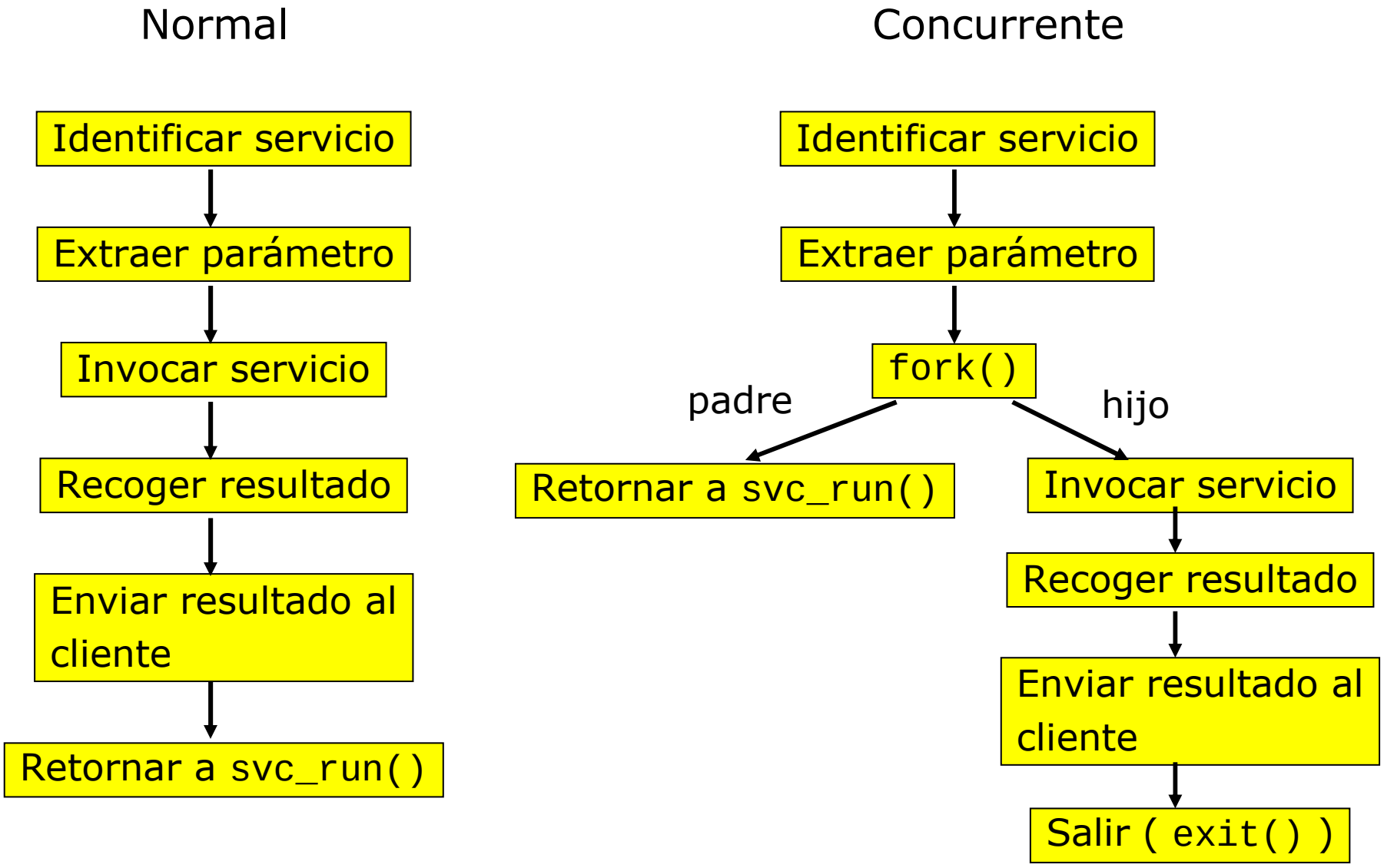
En el extremo del servidor tenemos la rutina repartidora (*dispatcher)* que realiza la siguientes tareas:

1. Identifica y selecciona el procedimiento que invoca el cliente inicializando los filtros XDR adecuados. Devuelve un mensaje de error si el procedimiento no existe.
2. Extrae del paquete que recibe por la red el argumento del procedimiento.
3. Invoca el procedimiento con el parámetro recibido.
4. Recoge el resultado devuelto por el servicio y se lo envía al cliente.
5. Retorna a `svc_run()`.

¿Podemos llamar a `fork` en algún momento?

Sí, en cualquier momento antes del punto 3 el repartidor puede crear un hijo que haga el trabajo y retornar él a `svc_run()`.

Concurrencia en el servidor



Concurrencia en el servidor. Ejemplo

```
static void sumador_3(struct svc_req *rqstp, register SVCXPRT *transp) {
  union {  datos suma_1_arg; } argument;  char *result;  xdrproc_t xdr_argument,
xdr_result;  char *(*local)(char *, struct svc_req *);
  switch (rqstp->rq_proc) {
  case NULLPROC:  (void) svc_sendreply(transp, (xdrproc_t) xdr_void, (char *)NULL);
    return;
  case SUMA:
    xdr_argument = (xdrproc_t) xdr_datos;
    xdr_result = (xdrproc_t) xdr_int;
    local = (char *(*)(char *, struct svc_req *)) suma_3;
    break;
  default:  . . . . .
  }
if (!svc_getargs(transp, xdr_argument, (caddr_t) &argument)) {
    svcerr_decode(transp);  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)) {
    fprintf(stderr, "unable to free arguments"); exit(1);
  }
  return;
}
```

if fork() return();

exit()

Nota: También existe el problema de los procesos errantes (zombies). Usar `signal()`.

La utilidad `rpcinfo`.

Interroga al localizador (*binder)* de Sun para obtener información de los servicios instalados en una máquina. Uso:

`Rpcinfo –p [máquina]` .- Muestra servicios registrados.

```
   program vers proto   port  service
    100000    4   tcp    111  rpcbind
    100000    3   tcp    111  rpcbind
    100000    4   udp    111  rpcbind
    100000    3   udp    111  rpcbind
    100011    1   udp  32772  rquotad
    100002    2   tcp  32771  rusersd
    100012    1   udp  32774  sprayd
    100008    1   udp  32775  walld
    100001    4   udp  32776  rstatd
    100221    1   tcp  32772
    100003    2   udp   2049  nfs
    100003    3   udp   2049  nfs
 545394752    1   udp  34117
 545394752    1   tcp  33010
```

Opciones de `rpcinfo`.

`-u maquina prognum [versnum]` .- Realiza una llamada al procedimiento número cero del programa `prognum` en `máquina`. Usará el protocolo udp.

`-t maquina prognum [versnum]` .- Es un "ping" como el caso anterior. En este caso usando tcp.

`-d prognum versnum` .- Elimina el registro en las tablas del localizador del programa `prognum` de versión `versnum`. Sólo puede hacerlo el propietario del programa (quién lo registró) o root. No disponible en todas las máquinas.

`-b prognum versnum` .- Realiza una llamada *broadcast* al procedimiento número cero del programa `prognum` y versión `versnum` en todas las máquinas de un segmento de red que los tengan instalado. Devuelve una lista con todas las máquinas que han respondido a la llamada.

Procesamiento en lotes (*batch*) de las RPC.

Usando `clnt_control` se puede conseguir que un cliente realice múltiples RPC sin esperar por respuesta del servidor.

Condiciones:

- El tiempo de espera del cliente tiene que ser cero.
- El servidor no debe enviar ninguna respuesta.
- Se debería utilizar un protocolo de transporte fiable como tcp.
- Se recomienda terminar la serie de peticiones con una RPC "normal" para garantizar el envío de las anteriores.

Ejemplo:

Procesamiento en lotes (*batch*) de las RPC.

```
CLIENT *clnt;
struct timeval espera;
                        .......
clnt=clnt_create(.....);
                        .......
espera.tv_sec=espera.tv_usec=0;
if (clnt_control(clnt,CLSET_TIMEOUT,&espera)==FALSE) {
    printf("Error");    exit(-1);
  }
for (i=0; i<10; i++)
    enlotes_1 (&i, clnt);
espera.tv_sec=25;
if (clnt_control(clnt,CLSET_TIMEOUT,&espera)==FALSE) {
    printf("Error"); exit(-1);
  }
resul=finlotes_1 (&dato, clnt);
                        ......
```

Gestión del localizador

El localizador o *binder* es un servidor implementado como un servidor de RPC.

Está registrado con el número de programa 100000 y escucha en el puerto 111 tanto de udp como de tcp.

El localizador se utiliza para realizar las llamadas en multidifusión o *broadcast.*

Para registrar un servicio, borrarlo o localizarlo es necesario realizar una RPC.

El ONC RPC de Sun suministra un conjunto de funciones para acceder a los servicios del localizador.

Gestión del localizador (II)

```
bool_t pmap_set (prognum,versnum, proto, port);
unsigned long prognum, versnum;
int proto;
unsigned short port;
```

Registra con el localizador el servidor de número `prognum` y versión `versnum` para el protocolo indicado en `proto`. El servicio escucha en el puerto `port`. Es usada por `svc_register` en el extremo del servidor.

```
bool_t pmap_unset (prognum,versnum);
unsigned long prognum, versnum;
```

Desrregistra con el localizador el servidor indicado.

Gestión del localizador (III)

```
struct pmaplist *pmap_getmaps (addr);
struct sockaddr_in addr;
```

Devuelve la lista de servicios registrados en el localizador de la máquina de dirección addr. La función devuelve NULL si no se puede contactar.

```
struct pmaplist {
        PMAP pml_map;
        struct pmaplist *pml_next;
 }
                    struct PMAP {
                            unsigned long pm_prog; /*programa*/
                            unsigned long pm_vers; /* versión*/
                            unsigned long pm_prot; /*Id protocolo*/
                            unsigned long pm_port; /* puerto*/
                            }
```

Gestión del localizador (IV)

```
enum clnt_stat pmap_rmtcall (
        struct sockaddr_in addr,
        unsigned long prognum,
        unsigned long versnum,
        unsigned long procnum,
            xdrproc_t inproc,
                char *in,
            xdrproc_t outproc,
                char *out,
        struct timeval timeout,
        unsigned long *portp
        );
```

Le solicita al localizador de la máquina **addr** que realice una llamada local al procedimiento indicado con los parámetros que se le pasan.

Si la llamada ha sido realizada de manera correcta, en el parámetro **portp** obtendremos el puerto de protocolo en el que escucha el servicio.

Si el servicio no está registrado, la función retorna con un *timeout.*

Multidifusión de RPC (*broadcast RPC*)

Es posible realizar una llamada en multidifusión (*broadcast*) a todos los servidores que provean un determinado servicio en una red.

Características:

1. La llamada se realiza a través del localizador, no directamente.
2. Una RPC normal devuelve un único resultado. Una RPC en multidifusión puede devolver un resultado por cada servidor instalado.
3. Se utiliza únicamente el protocolo UDP.
4. El tamaño máximo de los parámetros está limitado por la MTU ("*Maximun Transfer Unit*") de la red local. (Ethernet=1500 bytes)
5. Todas las llamadas en multidifusión utilizan por defecto el esquema de seguridad `AUTH_UNIX`.

Prototipo de `clnt_broadcast`

```
enum clnt_stat clnt_broadcast (
        unsigned long prognum,
        unsigned long versnum,
        unsigned long procnum,
            xdrproc_t inproc,
                  char *in,
            xdrproc_t outproc,
                  char *out,
      resultproc_t respuesta
              );
```

Solicita a los localizadores del segmento de red en el que se encuentra, que realicen una llamada local al procedimiento indicado, con los parámetros que se le pasan.

Si la llamada ha sido realizada de manera correcta, para cada uno de los servicios que respondan enviando un resultado **`clnt_broadcast`** invocará a la función **`respuesta`**.

Prototipo de `clnt_broadcast`

```
bool_t respuesta (
                    char *out,
                    struct sockaddr_in * addr
                    );
```

El parámetro **out** que recibe esta función es el mismo que el de la función **clnt_broadcast** y es la respuesta del servicio a la llamada.

El parámetro **addr** contiene todos los datos relativos a la máquina en la que se encuentra el servicio que ha contestado.

Si la función respuesta retorna **FALSE** (0), **clnt_broadcast** seguirá esperando por más respuestas de servidores.

Si la función respuesta retorna **TRUE** (1), **clnt_broadcast** no seguirá esperando por más respuestas de servidores, retornando el control.

Ejemplo de multidifusión

```
#include <netdb.h>
int contador=0;
static bool_t respuesta ( int  *viene,struct sockaddr_in *donde) {
        struct hostent *maqui;
        maqui= (struct hostent *)gethostbyaddr( (char *) &donde->sin_addr,
                        sizeof donde->sin_addr, AF_INET);
        printf("\nRespuesta recibida: %d",*viene);
        printf("\nMaquina que la envia: %s ",maqui->h_name);
        if (++contador < 6) return(FALSE);
}
main()  {
        enum clnt_stat estado;
        int dato;
        estado= clnt_broadcast(DUPLICADOR, PRIMERA, IDENTIFICA,xdr_void,
                (char *) NULL, xdr_int, (char *) &dato,respuesta);
        printf("\nValor de dato: %d\n", dato);
        if ( (estado != RPC_SUCCESS) && (estado != RPC_TIMEDOUT)) {
                clnt_perrno(estado);  exit(1);
        }
}
```