

Clase 15-08-2006: Funciones setjmp y longjmp:

C tiene la (infame) sentencia “**goto**” que permite que un programa siga ejecutando otra sentencia no contigua al **goto**. Este **goto** es **LOCAL**: el **goto** y la **etiqueta** de destino (el lugar al cuál se tiene que saltar) tienen que estar en la misma función.

```
{  
...  
    goto A;  
...  
A:      //etiqueta  
...  
}
```

Sin embargo hay lenguajes que tienen **goto NO LOCAL**. Dos ejemplos son :

- PASCAL
- PL/I (década del '60)

La filosofía de ambos era distinta.

Hay una distinción entre “**PALABRAS CLAVE**” y “**PALABRAS RESERVADAS**”

¿Qué significa que tienen **goto NO LOCAL**?

Por ejemplo en PASCAL se debea hacer las declaraciones locales:

```
Program P;  
label 99;  
procedure p;  
...  
begin  
...  
    goto 99:  
...  
end  
...  
begin  
    99:  
end
```

PASCAL solo permite **goto** “de adentro hacia afuera” es un **goto SEGURO**: pues no permite pasar el flujo de ejecución hacia algo que no existe.

PL/I implementa el **goto ASIGNADO**:

```
I = L      //sea L una etiqueta definida con anterioridad  
GOTO I
```

También se puede hacer en **PL/I**:

```
I = 3.1415  
GOTO I      //CUIDADO: no es demasiado seguro
```

Clase 15-08-2006: Funciones setjmp y longjmp:

Siguiendo con lo nuestro, vamos a ver que **C** eligió unir **lo peor de PASCAL** con **lo peor de PL/I**.

En el archivo de cabecera: **/usr/include/setjmp.h**

Se define un **TIPO DE DATOS :**

jmp_buf
y dos funciones, cuyos prototipos son:

```
int setjmp( jmp_buf )  
  
void longjmp( jmp_buf, int )
```

¿Qué hacen estas funciones?

setjmp guarda en un **jmp_buf** el **ESTADO DEL PROCESO** en ese momento.

¿Qué quiere decir el **ESTADO DEL PROCESO**?

Es

- qué valor posee el **CONTADOR DE INSTRUCCIONES (registro del microprocesador)**
- qué valor tiene el **PUNTERO A PILA (STACK POINTER)**
- etc

setjmp devuelve CERO SIEMPRE.

¿Qué hace **longjmp**?

```
longjmp( jmp_buf b, int i )
```

TOMA la información del ESTADO DEL PROCESO que está guardado en el jmp_buf llamado, en este ejemplo b.

El **PROCESO** parece que vuelve al interior de **setjmp**.

La única diferencia es que volverá retornando el valor de i, de acuerdo a lo siguiente:

Si i es CERO longjmp lo cambia a UNO silenciosamente

Clase 15-08-2006: Funciones setjmp y longjmp:

Veamos un ejemplo:

```
/* Archivo: prueba-setjmp.c */
#include <stdio.h>
#include <setjmp.h>

jmp_buf buf;

void f (int i)
{
    printf("en f\n");
    if(i!=0) longjmp(buf , 1);
    printf("salimos de f\n");
}

int main()
{
    printf("uno\n");
    if( setjmp(buf) != 0){
        printf("!!!\n");
        return 0;
    }

    f(0); f(1);
    printf("chau\n");
    return 0;
}
//Fin Archivo: prueba-setjmp.c
```

Que compilamos con la orden:

gcc -Wall prueba-setjmp.c -o prueba-setjmp.out

Y Ejecutamos mediante: **./prueba-setjmp.out**

Cuya salida en pantalla es:

**uno
en f
salimos de f
en f
!!!**

Esto funciona como un **goto**, pero puede estar en dos funciones en localidades distintas y distantes entre ellas, es decir en diferentes ámbitos de acceso.

Sirve usar las funciones **setjmp** y **longjmp** como salida de emergencia.

Clase 15-08-2006: Funciones setjmp y longjmp:

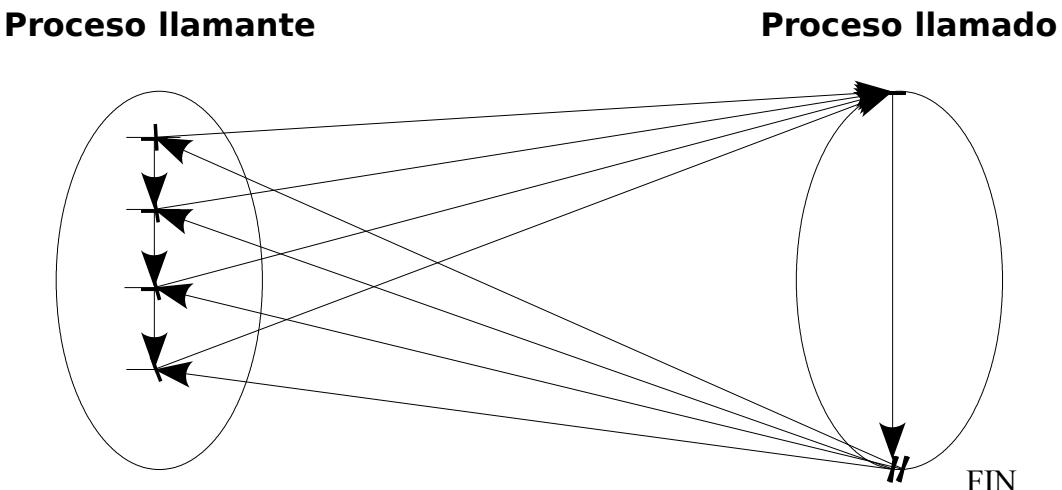
Para obtener una **multitarea “real”** se necesitaría de **un procesador dedicado para ejecutar las instrucciones de cada “proceso”**.

Se puede implementar una **multitarea trucha** sacando la información del **estado de un proceso** mediante **setjmp** y **restaurando el estado con longjmp**, para cada proceso e **ir turneando entre cada uno de los procesos**.

setjmp y **longjmp** permiten implementar una **multitarea sencilla**.

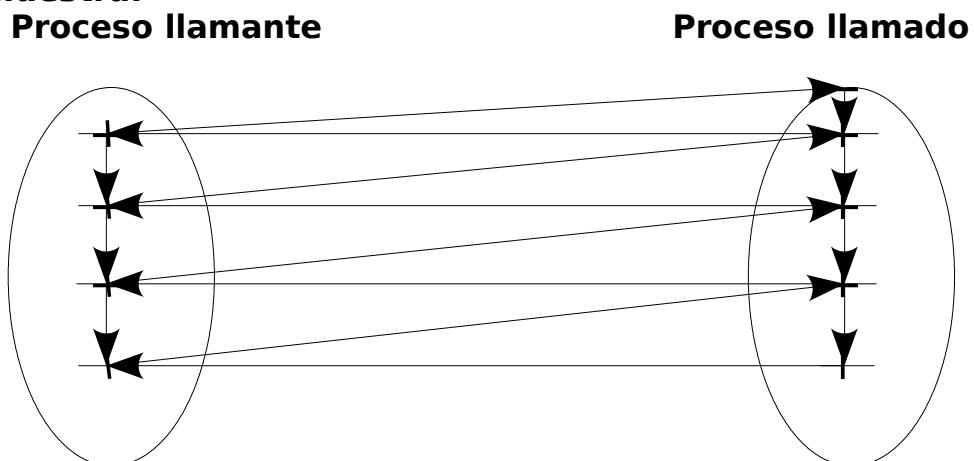
SUBRUTINAS:

Veamos cómo es el esquema correspondiente a una **subrutina**



Este es un modelo **asimétrico**, **todas las llamadas a subrutinas van hacia el mismo lugar, en el proceso llamado, y retornan hacia el mismo lugar, en el proceso llamante**.

En cambio, en el **modelo de corrotinas** el esquema resulta **simétrico** como se muestra:



Clase 15-08-2006: Funciones setjmp y longjmp:

El modelo de corrutinas es usado en los **REDIRECCIONAMIENTOS DE ENTRADA-SALIDA (por ejemplo: ls | wc)**. Es la **multitarea que venía con Windows 3.11**.

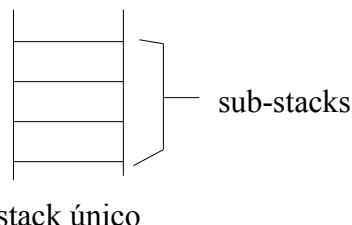
Hay un problema: **C** genera código que asume que el **proceso tiene en un instante dado, un solo estado legítimo**, hay un sólo stack.

Si queremos **implementar corrutinas, cada proceso** interviniente **deberá tener su propio stack**.

Trataremos de hacerlo con el mínimo gasto posible (sin usar **assembler**, ni cosas raras).

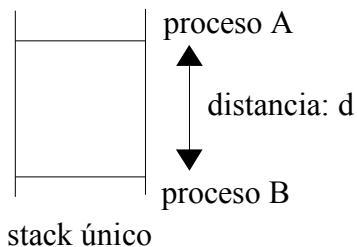
Algunos detalles:

- **CORRUTINA <====> FUNCIÓN**
- Con respecto a múltiples stack: el esquema sería el siguiente



stack único

¿Cómo podemos pasar de un proceso A a otro proceso B en el siguiente esquema?



Pués, con **recursión**. Bosquejemos una función que haga esto.

```
/**  
 * aux  
 * Función que hace sub-stacks y lo guarda en un jmp_buf  
 * Argumentos:  
 *     unsigned int d  
 *     jmp_buf buf  
 *     void(*f)()  
 */
```

buf es del tipo **jmp_buf** definido **setjmp.h**
Puntero a una función con argumento **void**
que retorna **void**

Clase 15-08-2006: Funciones setjmp y longjmp:

```
void aux( unsigned int d, char * p, jmp_buf buf, void(*f)() )
{
    if( p - (char *)&d < d )           //esto es la distancia
    {
        aux( d, p, buf, f);           //se hace una llamada recursiva
    }
    else
    {
        if( setjmp( buf ) != 0 )
        {
            f();                      //que toma como argumento
            abort();                   //definida en stdlib.h
        }
    }
}
```

Ahora veamos un código fuente concreto:

```
// Archivo: multitarea.c      Ejemplo de CORRUTINAS
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

/* DEFINICIONES */
#define MAX_T 16
#define T_STACK (4*1024)

/* Para cambiar entre corrutinas */
#define TRANSFER(o,d) (setjmp(o)==0?(longjmp(d,1),0):0)

/* SINONIMOS */
typedef jmp_buf task_t;
typedef void (*pfunc)();

/* VARIABLES GLOBALES */
task_t t1, t2, tmain;
```

Clase 15-08-2006: Funciones setjmp y longjmp:

```
// Archivo: multitarea.c      Ejemplo de CORRUTINAS - continuación
/* FUNCIONES */
void aux(unsigned int d, char * p, jmp_buf buf, void (*f)() )
{
    if( p - (char*)&d < d) aux( d, p, buf, f);
    else
    {
        if( setjmp(buf) != 0 )
        {
            f(); abort();
        }
    }
}
void hace_stack(task_t t, pfunc f)
{
    static int ctas = MAX_T;
    char dummy;
    aux (ctas*T_STACK,&dummy,t,f);
    ctas--;
}
void f1()
{
    double d;
    for(d=0;d<1000;d+=0.01){
        printf("en f1, d=%f\n",d);
        TRANSFER(t1,t2);
    }
}
void f2()
{
    int i;
    for( i=0 ; i<50 ; i++ )
    {
        printf("en f2, i=%d\n",i);
        TRANSFER(t2,t1);
    }
    TRANSFER(t2,tmain);
}
/* FUNCION PRINCIPAL MAIN */
int main()
{
    hace_stack(t1,f1);
    hace_stack(t2,f2);
    TRANSFER(tmain,t1);
    return 0;
}// Fin Archivo: multitarea.c
```

Clase 15-08-2006: Funciones setjmp y longjmp:

Que se compila mediante:

gcc -Wall multitarea.c -o multitarea.out

Al ejecutarlo mediante la orden:

./multitarea.out

Se observa una salida en pantalla similar a la siguiente:

```
...
en f1, d=0.000000
en f2, i=0
en f1, d=0.010000
en f2, i=1
en f1, d=0.020000
en f2, i=2
en f1, d=0.030000
...
```

Clase 22-08-2006: Ayuda en resolución de ejercicios - qsort:

Vamos a ver cómo usar una función denominada **qsort** que implementa un algoritmo de ordenación rápida.

Se puede consultar su página de manual en línea mediante: **man qsort**

Su prototipo es:

```
void qsort(
    void * base,           //Arreglo a ordenar
    size_t nelem,          //Cantidad de elementos del arreglo
    size_t tam,             //tamaño de cada elemento del arreglo

    int(*cmp)( const void * , const void * )
    /* puntero a una función de comparación que devuelve un
     ** entero de acuerdo al siguiente criterio:
     **      < 0  si *(primer-puntero) < *(segundo-puntero)
     **      = 0  si *(primer-puntero) = *(segundo-puntero)
     **      > 0  si *(primer-puntero) > *(segundo-puntero)
     */
)
```

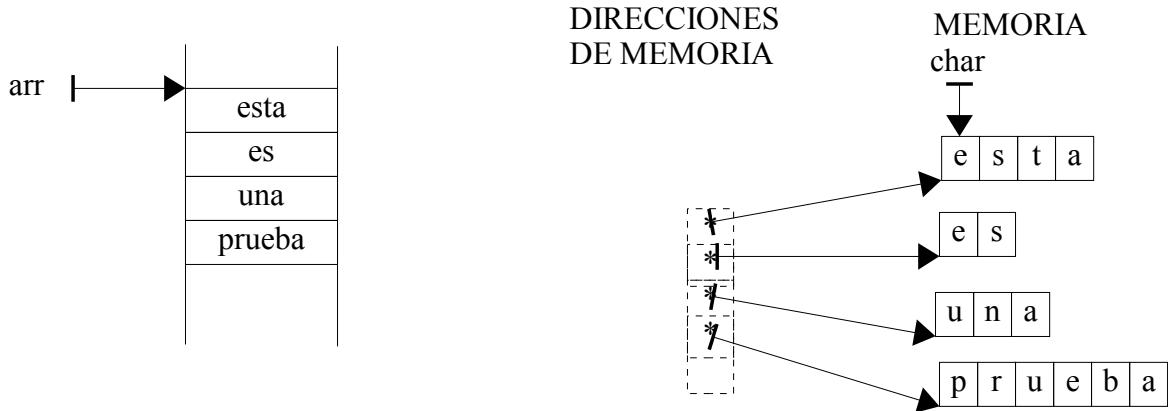
Ejemplo: se puede utilizar para ordenar un arreglo de “**strings**”, más precisamente, un **arreglo de punteros a caracteres**, pues el tipo de dato string no está contemplado en **C**.

La declaración en **C** de un **arreglo de punteros a caracteres** como el siguiente {"esta", "es", "una", "prueba"} } es, en su forma inicializada:

```
char * arr[] = {"esta", "es", "una", "prueba"};
// arr es un arreglo de punteros a caracteres
```

Clase 22-08-2006: Ayuda en resolución de ejercicios – qsort:

El esquema que presenta este arreglo en memoria es



Ahora veamos un código fuente de ejemplo que hace uso de **qsort**:

```
//Archivo: ejemplo-qsort.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Arreglo de punteros a cadenas de caracteres */
static char *arr[]=
{
    "Esta", "es", "una", "prueba"
};

static int cmp(const void *a, const void *b)
{
    const char **pa=a , **pb=b;
    return strcmp((*pa),(*pb));
}

int main ()
{
    int i;
    qsort(arr, sizeof(arr)/sizeof(arr[0]), sizeof(arr[0]), cmp);
    for(i=0; i<sizeof(arr)/sizeof(arr[0]); i++)
        printf("%s\n",arr[i]);
    return 0;
}
//Fin del Archivo: ejemplo-qsort.c
```

Que **se compila** con: **gcc -Wall ejemplo-qsort.c -o ejemplo-qsort.out**
Y **se ejecuta** con: **./ejemplo-qsort.out**

Clase 22-08-2006: Ayuda en resolución de ejercicios – qsort:

Obteniéndose en pantalla:

**Esta
es
prueba
una**

Es decir el, arreglo de punteros a cadenas de caracteres en **orden lexicográfico ascendente**.

Ahora vamos a ver una **implementación sencilla en C del algoritmo de BÚQUEDA BINARIA**, este **funciona mientras el arreglo se encuentre ordenado**.

```
/* Función que busca un elemento en un arreglo ordenado.  
 * Retorna el índice en donde se encuentra en la primera aparición del  
 * elemento, de lo contrario retorna -1  
 */  
int mibsearch (      unsigned int * arreglo ,  
                    unsigned int tamano ,  
                    unsigned int buscado  
                )  
{  
    unsigned int min = 0 , max = (tamano - 1) ;  
    unsigned int central;           //= ( min + ( max - min )/ 2 )  
    if( buscado >= arreglo[min] && buscado <= arreglo[max] )  
    {  
        while( min <= max )  
        {  
            //para asegurarse que caiga en uno de los elementos del arreglo  
            central = (unsigned int)( min + ( max - min )/2 );  
            if( buscado == arreglo[central] )  
                return central; // lo encontró y retorna su índice  
            else if( buscado > arreglo[central] )  
                min = central + 1;  
            else  
                max = central - 1;  
            central = ( min + ( max - min )/2 );  
        }  
        /* si no lo encuentra retorna -1 pues los elementos del arreglo  
         **son todos unsigned int y de esta manera es fácil distinguir  
         **que no se ha encontrado el elemento buscado  
        */  
        return (-1);  
    }
```

Clase 22-08-2006: Ayuda en resolución de ejercicios – qsort:

```
//Continua implementación de algoritmo de búsqueda binaria
else if( buscado < arreglo[min] || buscado > arreglo[max] )
{
    printf("USTED INGRESÓ EL VALOR: %u \n" , buscado );
    printf("QUE ES MENOR QUE EL MÍNIMO VALOR DEL ARREGLO,
           O MAYOR QUE EL MÁXIMO VALOR DEL ARREGLO\n");
    printf("ES DECIR: NO SE ENCUENTRA EN EL ARREGLO! \n" );
    return -1;
}
return -1;
}
//Fin de la implementación del algoritmo de búsqueda binaria
```

Ahora veamos cómo se puede usar en un código fuente:

```
//Archivo: ej2a.c
//Ejercicio 2a de SISTEMAS OPERATIVOS – DIEGO BOTTALLO

#include <stdio.h>
#include <stdlib.h> // para "qsort" y "rand"

/********************* VARIABLES GLOBALES *****/
unsigned int cuantos, buscado, resultado;

/********************* PROTOTIPOS *****/
int compara( const void * , const void * );
int mibsearch( unsigned int * , unsigned int , unsigned int );
void inicializa( unsigned int * );
void exibe( unsigned int * );
void verificaYMuestra( unsigned int * , unsigned int );

/********************* FUNCIONES *****/
/* Función de comparación: solo en modo ascendente, pues el algoritmo
 * de búsqueda binaria requiere que los elementos del arreglo estén
 * ordenados.
 */
int compara( const void * arg1 , const void * arg2 )
{
    return ( *((unsigned int *)arg1) - *((unsigned int *)arg2) );
}
```

Clase 22-08-2006: Ayuda en resolución de ejercicios – qsort:

```
//Continuación del archivo: ej2a.c
/* Función que inicializa los valores del arreglo en forma aleatoria usando
 * la función "rand"
 */
void inicializa( unsigned int * arreglo )
{
    int i;
    printf("\nINICIALIZANDO LOS ELEMENTOS DEL ARREGLO EN
          FORMA ALEATORIA...\n");
    for( i=0 ; i < cuantos ; i++ )
    {
        arreglo[i] = 1 + (int)( 10000.0 * rand()/(RAND_MAX + 1.0 ) );
        // se generan números aleatorios entre 1 y 10000
        printf("... ARREGLO[%d] = %d ...\n" , i , arreglo[i] );
    }
    printf("\n");
}

// Rutina de exhibición de los elementos del arreglo en pantalla
void exibe( unsigned int * arreglo )
{
    unsigned int i;
    printf("\n");
    for( i=0 ; i < cuantos ; i++ )
    {
        printf(" ARREGLO[%u] = %u \n" , i , arreglo[i] );
    }
    printf("\n");
}
```

Clase 22-08-2006: Ayuda en resolución de ejercicios – qsort:

```
//Continuación del archivo: ej2a.c

/* Función que busca un elemento en un arreglo ordenado.
 * Retorna el índice en donde se encuentra en la primera aparición del
 * elemento, de lo contrario retorna -1
 */
int mibsearch(    unsigned int * arreglo , unsigned int tamanio ,
                  unsigned int buscado )
{
    unsigned int min = 0 , max = (tamanio - 1) ;
    unsigned int central;           //= ( min + ( max - min )/ 2 )
    if( buscado >= arreglo[min] && buscado <= arreglo[max] )
    {
        while( min <= max )
        {
            central = (unsigned int)( min + ( max - min )/2 );
            //para asegurarse que caiga en uno de los elementos del
            //arreglo
            if( buscado == arreglo[central] )
                return central; // lo encontró y retorna su índice
            else if( buscado > arreglo[central] )
                min = central + 1;
            else
                max = central - 1;
            central = ( min + ( max - min )/2 );
        }
        // si no lo encuentra retorna -1 pues los elementos del arreglo
        // son todos unsigned int y de esta manera es fácil distinguir
        // que no se ha encontrado el elemento buscado
        return (-1);
    }
    else if( buscado < arreglo[min] || buscado > arreglo[max] )
    {
        printf("USTED INGRESÓ EL VALOR: %u \n" , buscado );
        printf("QUE ES MENOR QUE EL MÍNIMO VALOR DEL ARREGLO,
              O MAYOR QUE EL MÁXIMO VALOR DEL ARREGLO\n");
        printf("ES DECIR: NO SE ENCUENTRA EN EL ARREGLO! \n" );
        return -1;
    }
    return -1;
}
```

Clase 22-08-2006: Ayuda en resolución de ejercicios – qsort:

```
//Continuación del archivo: ej2a.c

void verificaYmuestra( unsigned int * arreglo,
                      unsigned int indiceResultado )
{
    if( indiceResultado == (-1) )
        printf("\nEL ENTERO SIN SIGNO BUSCADO NO SE ENCUENTRA
               EN EL ARREGLO\n" );
    else if( indiceResultado != (-1) )
    {
        printf("\nEL ENTERO SIN SIGNO: %u \n" ,
               arreglo[indiceResultado] );
        printf("SE ENCUENTRA EN LA POSICIÓN: %u \n" ,
               indiceResultado );
    }
}

/***************** FUNCIÓN PRINCIPAL MAIN *****/
int main()
{
    printf("INGRESE LA CANTIDAD DE ENTEROS SIN SIGNO QUE
          CONTENDRÁ EL ARREGLO: \n" );
    scanf( "%u" , &cuantos );
    unsigned int arreglo[cuantos];
    inicializa( arreglo );
    qsort( arreglo , sizeof(arreglo)/sizeof(arreglo[0]) , sizeof(arreglo[0]) ,
           compara );
    printf("EL ARREGLO ORDENADO EN FORMA ASCENDENTE: \n" );
    exibe( arreglo );
    printf("INGRESE EL VALOR DEL ENTERO SIN SIGNO QUE DESEA
          BUSCAR EN EL ARREGLO: \n" );
    scanf( "%u" , &buscado );

    resultado = mibsearch( arreglo , cuantos , buscado );
    // posición en el arreglo ó -1 si no se encuentra

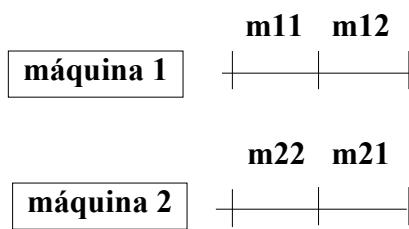
    verificaYmuestra( arreglo , resultado ) ;
    return 0;      // informa al entorno finalización correcta
}
// Fin del archivo: ej2a.c
```

Que se compila con: **gcc -Wall ej2a.c -o ej2.out**

Y se ejecuta con: **./ej2a.out**

Clase 29-08-2006: Historia de internet – TCP/IP:

- **1965** **Jon Postel** y otros consideran **multiplexar** temporalmente el envío de transmisiones entre computadoras (**PACKET MESSAGING**)



REFERENCIA:

m12: paquete desde **máquina 1** hacia **máquina 2**

m21: paquete desde **máquina 2** hacia **máquina 1**

Se puede mantener una conexión entre varias máquinas siempre y cuando se pueda discernir desde dónde y hacia dónde se dirige cada paquete.

- **1966** Prueba exitosa entre el **MIT** y **UCLA** (conectan de un extremo al otro de los Estados Unidos).
El Departamento de Defensa de los Estados Unidos sponsorea las investigaciones.
- **1974** **Bolt, Beranack & Newman:** **PROTOCOLO TCP/IP (con modelo de capas)**
- **1975** **Berkeley** es comisionada para **implementar el PROTOCOLO TCP/IP**
- **1982** **Sale BSD 3.0 con TCP/IP** y nace la **ARPANET**
- **1992** **Desaparece ARPANET y nace INTERNET**

¿Qué es un protocolo?

Son **normas y reglas** para la **comunicación e intercambio** de la **información**.

También **daremos protocolo a su implementación:** todos los **módulos y demás cosas del sistema operativo que hacen que funcione**.
EL protocolo es para comunicar dos PROCESOS.

Necesitamos:

- Un medio físico**
- Un protocolo COMÚN que ofresca: un ESPACIO DE NOMBRES (DIRECCIONES)**

Clase 29-08-2006: Historia de internet – TCP/IP:

TCP/IP usa como espacio de direcciones 2 números:

- Un entero de 32 bits llamado **DIRECCIÓN IP (IP ADDRESS)** que denota a la **COMPUTADORA (HOST)**
- Un entero de 16 bits llamado **PUERTO (PORT)** que denota a **UN PROCESO DENTRO DE UN HOST**

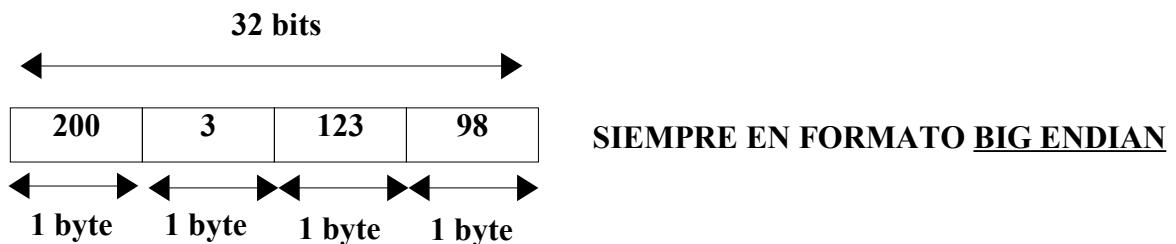
Analogía: Comparando con un EDIFICIO DE DEPARTAMENTOS:

- LA **DIRECCIÓN o el TELÉFONO DEL EDIFICIO** sería la **IP**
- EL **NÚMERO DE DEPARTAMENTO o el INTERNO TELEFÓNICO** sería el **PORT**

Las **IP Address** se acostumbran a dar así:

200.3.123.98 (eva_fceia_unr.edu.ar)

¿Qué significa?



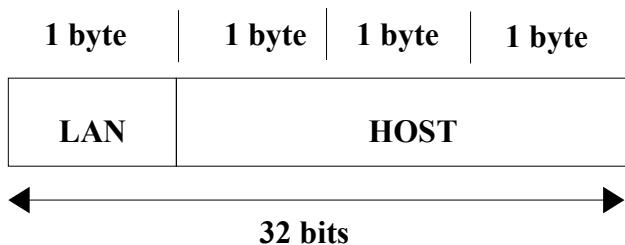
Ahora, por lo general, **los host se agrupan en REDES LOCALES (LANs)**

Analogía con las **CENTRALES TELEFÓNICAS**:

- Un **interno en un departamento**: (0341) 437-1234
- Otro **interno en el mismo departamento**: (0341) 437-0381

En una **LAN necesitamos algo parecido a una CENTRAL TELEFÓNICA.**

Necesitamos PARTIR LAS IP ADDRESS:



Según donde se ubique la partición de la IP ADDRESS tendremos para elegir:

- **POCAS REDES LANs con MUCHOS HOSTs**
- **MUCHAS REDES LANs con POCOS HOSTs**

Clase 29-08-2006: Historia de internet – TCP/IP:

No vamos a partir así **LAS IP ADDRESS** sino que usaremos:
CATEGORÍAS DE REDES

CLASE O CATEGORÍA	LAN	HOST	VALOR DEL 1º BYTE EN DECIMAL
A	8 BITS 1º BYTE	24 BITS 2º, 3º Y 4º BYTE	1 – 127
B	16 BITS 1º Y 2º BYTES	16 BITS 3º Y 4º BYTES	128 – 191
C	24 BITS 1º, 2º Y 3º BYTES	8 BITS 4º BYTE	192 – 223
D	---	MULTICAST	224 – 239
E	---	EXPERIMENTAL	240 – 255

MODELO DE CAPAS:

Los que se comunican son procesos, entre distintos host



EJEMPLO:

PERMITE
DISERNIR
CUÁL PLACA
USAR

INTERNET PROTOCOL

CAPA DE RED (IP)

DRIVERS

CAPA DE ENLACE

HARDWARE

CAPA FÍSICA

Clase 29-08-2006: Historia de internet – TCP/IP:

El PROTOCOLO TCP/IP NÓ TIENE 7 CAPAS COMO EL MODELO OSI SINÓ QUE TIENE 4 CAPAS.

El **socket** es una **abstracción** que **permite** en principio **a un proceso** **comunicarse con la implementación del PROTOCOLO EN LA CAPA DE TRANSPORTE ELIGIENDO USAR TCP O UDP.**

127.0.0.1 Denota la IP ADDRESS DE LA PROPIA COMPUTADORA

¿Por qué 2 SUB-PROTOCOLOS?

TCP: TRANSMISION CONTROL PROTOCOL

Es un **protocolo “seguro”**: garantiza que hace el **máximo esfuerzo para evitar la DUPLICACIÓN DE PAQUETES, la DESAPARICIÓN DE PAQUETES y también SU CORRUPCIÓN.**

Por esto, **TCP** es un protocolo **ORIENTADO A SESIÓN**.

UDP: UNIVERSAL DATAGRAM PROTOCOL

Garantiza que NO HARÁ NADA PARA REMEDIAR ESTOS INCONVENIENTES.

Por esto, **UDP** está **ORIENTADO A PACKETS**.

En la práctica, UDP es muy seguro.

Socket: La implementación que permite ver protocolos, no sólo TCP/IP, a un proceso.

Ejemplos: Veremos que pasará lo mismo que con los DESCRIPTORES DE ARCHIVOS: **un SOCKET es UN VALOR ENTERO**

Para poder usarlos se deben incluir los siguientes archivos de cabecera:

```
#include <sys/types.h>
#include <sys/socket.h>
```

- Creación de un **socket** para el **PROTOCOLO TCP/IP con el SUB-PROTOCOLO TCP**:

```
int sock;
sock = socket( PF_INET, SOCK_STREAM, 0);
if( sock < 0 ) error;

/*      PF_INET      significa PROTOCOL FAMILY INTERNET
**      SOCK_STREAM   "      SUBPROTOCOL TCP
*/

```

Clase 29-08-2006: Historia de internet – TCP/IP:

- Creación de un **socket** para el **PROTOCOLO TCP/IP con el SUB-PROTOCOLO UDP**

```
int sock;
sock = socket( PF_INET, SOCK_DGRAM, 0);

/*      PF_INET      significa PROTOCOL FAMILY INTERNET
**      SOCK_DGRAM    "      SUBPROTOCOL UDP
*/
```

Para ampliar la información se recomienda leer obligatoriamente la página del manual en línea de comandos: man socket

Un socket se debe cerrar, luego de usarlo, y para ello se utiliza:
close(sock);

Un socket cerrado no se puede usar.

Vamos a poder usar como archivo un socket TCP.

¿Cómo usar un socket?

Depende del **SUB-PROTOCOLO**.

Se sigue los lineamientos del **MODELO CLIENTE/SERVIDOR (CLIENT/SERVER)**.

Un **PAR CLIENT/SERVER** es un **PAR DE PROCESOS QUE SE COMUNICAN ENTRE SÍ**.

La única diferencia entre ambos (desde nuestro punto de vista), es que la:

PRIMERA ACCIÓN del **SERVIDOR** es **ESPERAR** un **contacto**, mientras que la

PRIMERA ACCIÓN del **CLIENTE** es **INICIAR** ese **contacto**.

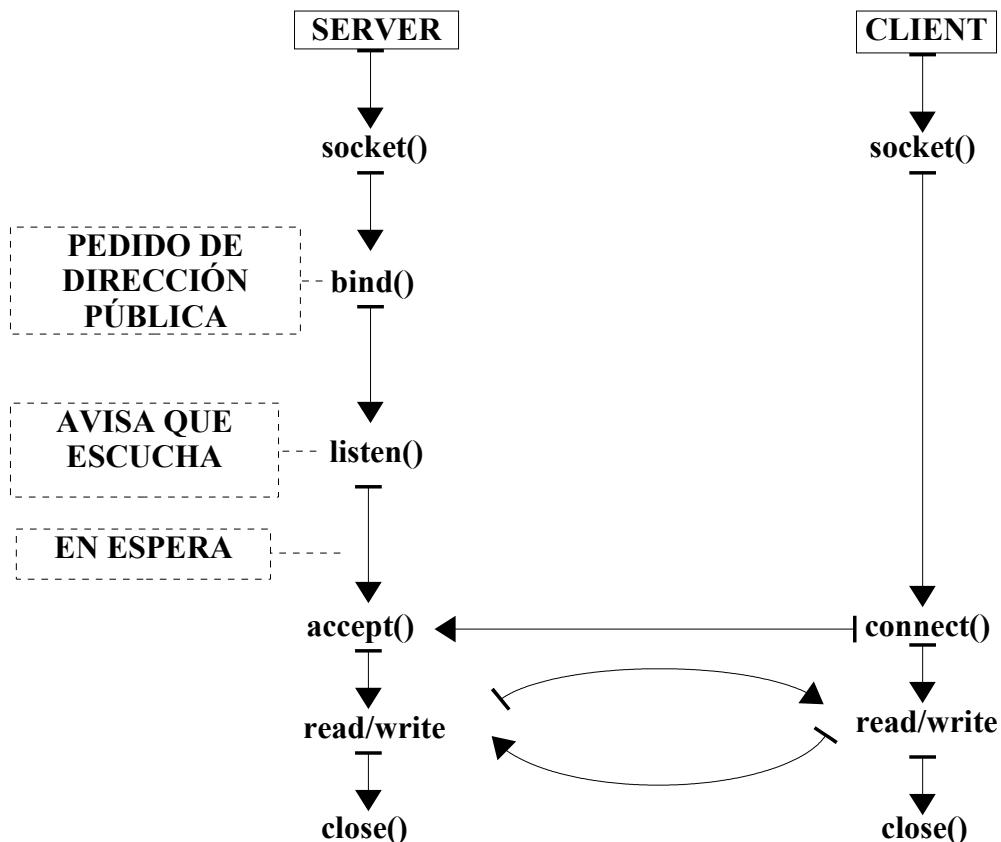
PROCESO	
SERVIDOR	CLIENTE
ESPERA	INICIA

Esto es **ASIMÉTRICO** y se nota en un requisito obvio: el **SERVIDOR debe hacer PÚBLICA SU DIRECCIÓN**.

Recién después de esto podría **ESPERAR POR UNA COMUNICACIÓN**.

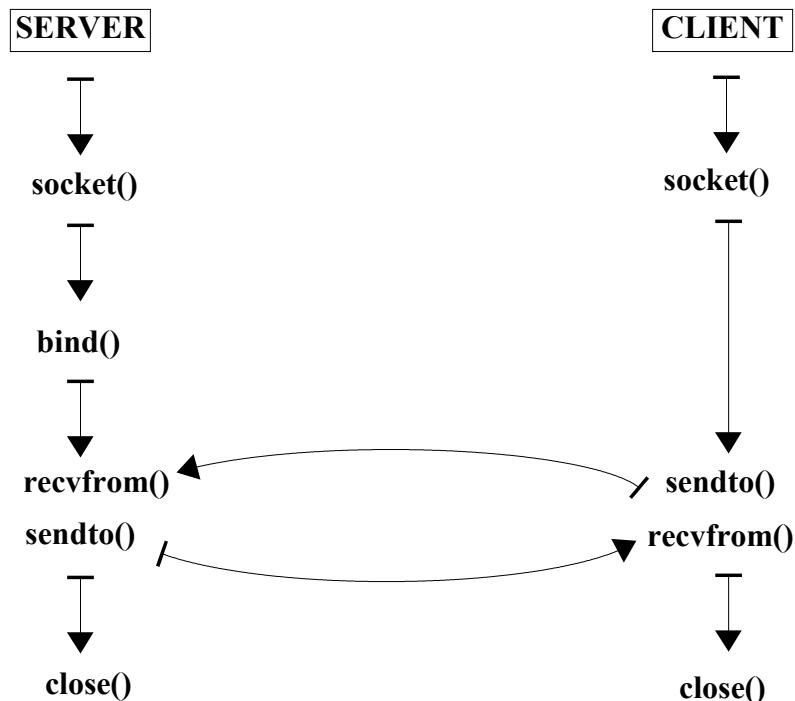
Clase 29-08-2006: Historia de internet – TCP/IP:

**SUMARIO DE ACCIONES PARA HACER UNA COMUNICACIÓN
CLIENTE/SERVIDOR USANDO EL SUB-PROTOCOLO TCP**



Clase 29-08-2006: Historia de internet – TCP/IP:

**SUMARIO DE ACCIONES PARA HACER UNA COMUNICACIÓN
CLIENTE/SERVIDOR USANDO EL SUB-PROTOCOLO UDP**



Veamos un **primer par de códigos fuente de ejemplo:**

- **hola-mundo-servidor-tcp.c**
- **hola-mundo-cliente-tcp.c**

Clase 29-08-2006: Historia de internet – TCP/IP:

```
/* Archivo: hola-mundo-servidor-tcp.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

/* DEFINICIONES */
#define PORT 5000
#define SIZE 1024

/* SINONIMOS */
typedef struct sockaddr * sad;
//    sad    puntero a una struct sockaddr

/* FUNCIONES */
void error(char * s)
{
    perror(s);
    exit(-1);
}

/* FUNCION PRINCIPAL */
int main()
{
    int s, s1;
    char linea[SIZE];
    int cuanto, L;

    if( (s = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
        error("socket");

    struct sockaddr_in sin, sin1; //Leer: man 7 ip
    //sin y sin1 son del tipo: struct sockaddr_in

    sin.sin_family = AF_INET;      //Leer: man 7 ip
    sin.sin_port = htons(PORT);
    //Host To Network Short. Leer: man 3 htons
    sin.sin_addr.s_addr = INADDR_ANY; //Cualquier interface.

    if( bind(s, (sad)&sin, sizeof sin) < 0 )
        error("bind");
```

Clase 29-08-2006: Historia de internet – TCP/IP:

```
/* Continuacion Archivo: hola-mundo-servidor-tcp.c */

    if( listen(s, 5) < 0 )
        error("listen");

    L = sizeof(sin1);

    //Recibe por s. Atiende por s1.
    if((s1 = accept(s, (sad)&sin1, &L)) < 0)
        error("accept");

    cuanto = read(s1, linea, SIZE); //Leer: man 2 read
    linea[cuento] = 0;
    printf("De %s : %d llega …> %s \n",
           inet_ntoa(sin1.sin_addr),
           ntohs(sin1.sin_port),
           linea);
    write(s1, "CHAU !", 5);

    //Los sockets abiertos deben cerrarse.
    close(s1), close(s);

    return 0;      //Finalización exitosa
}

/* Fin Archivo: hola-mundo-servidor-tcp.c */
```

Que se compila mediante la orden:

gcc -Wall hola-mundo-servidor-tcp.c -o hola-mundo-servidor-tcp.out

Que se ejecuta mediante la orden:

./hola-mundo-servidor-tcp.out

De esta manera **se inicia el PROCESO SERVIDOR** que se **QUEDA A LA ESPERA** de **CONEXIONES** que **VENDRÁN DEL PROCESO CLIENTE UNA VEZ QUE ÉSTE SE EJECUTE.**

Clase 05-09-2006: Historia de internet – TCP/IP:

Ahora veamos entonces el código fuente de **hola-mundo-cliente-tcp.c :**

```
/* Archivo: hola-mundo-cliente-tcp.c */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

Clase 05-09-2006: Historia de internet – TCP/IP:

```
/* Continuacion Archivo: hola-mundo-cliente-tcp.c */
/* DEFINICIONES */
#define PORT 5000
#define SIZE 1024

/* SINONIMOS */
typedef struct sockaddr *sad;

/* FUNCIONES */
void error(char*s)
{
    perror(s);
    exit(-1);
}
/* FUNCION PRINCIPAL */
int main(int argc, char **argv)
{
    int sock;
    struct sockaddr_in sin;
    char linea [SIZE];
    int cuanto;

    if(argc < 2){
        fprintf(stderr,"Uso: %s ipaddr\n", argv[0]);
        exit(-1);
    }

    if((sock = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
        error("socket");

    sin.sin_family = AF_INET;
    sin.sin_port = htons(PORT); //Host to Network Short

    inet_aton(argv[1],&sin.sin_addr); //Ascii To Network

    if( connect(sock, (sad)&sin, sizeof sin) < 0)
        error("connect");

    write(sock, "Hola mundo!\n", 12);
    cuanto = read(sock, linea, SIZE);
    linea[cuento]=0;
    printf("Contestaron: %s\n", linea);
    close (sock);
    return 0;
}
/* Fin Archivo: hola-mundo-cliente-tcp.c */
```

Clase 05-09-2006: Historia de internet – TCP/IP:

Que se compila mediante la orden:

gcc -Wall hola-mundo-cliente-tcp.c -o hola-mundo-cliente-tcp.out

Como **primera prueba de ejecución** podemos invocarlo mediante la siguiente orden para **ver el mensaje que nos muestra**:

./hola-mundo-cliente-tcp.out

El mensaje que exibe en pantalla es el siguiente:

Uso:./hola-mundo-cliente-tcp.out ipaddr

Entonces podemos ahora **invocarlo mediante la orden** siguiente, **pasando como argumento de línea de comandos** la **DIRECCIÓN IP 127.0.0.1** que **hace referencia a la PROPIA COMPUTADORA**:

./hola-mundo-cliente-tcp.out 127.0.0.1

Para ver bién tanto el mensaje del **PROCESO CLIENTE** como el del **PROCESO SERVIDOR** es aconsejable **invocar a cada uno en consolas virtuales diferentes**.

De este modo:

- En la consola donde se ejecuta **hola-mundo-servidor-tcp.out** se observara, luego de ejecutar en la otra consola **hola-mundo-cliente-tcp.out**, el siguiente mensaje:

De 127.0.0.1 : 33095 llega ...> Hola mundo!

- En la consola donde se ejecuta **hola-mundo-cliente-tcp.out** por su parte se observará:

Contestaron: CHAU

Se debe notar que este **PROCESO SERVIDOR no finaliza** su ejecución **hasta que recibe una CONEXIÓN del PROCESO CLIENTE y la ATIENDE (PAR CLIENTE/SERVIDOR SUB-PROTOCOLO TCP)**.

NOTA: Si se quiere ejecutar este **PAR DE PROCESOS** más de una vez, **se deberá esperar un intervalo de tiempo**, de lo contrario se observará en pantalla, al invocar **hola-mundo-servidor-tcp.out** el siguiente mensaje:

bind: Address already in use

Pasaremos ahora a implementar otro **PAR DE PROCESOS CLIENTE/SERVIDOR** pero **ahora usando el SUB-PROTOCOLO UDP.**

Clase 05-09-2006: Historia de internet – TCP/IP:

```
/* Archivo: hola-mundo-servidor-udp.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

/* DEFINICIONES */
#define PORT 5000
#define SIZE 1024

/* SINONIMOS */
typedef struct sockaddr * sad;

void error (char *s)
{
    perror(s);
    exit (-1);
}

/* FUNCION PRINCIPAL */
int main ()
{
    int sock;
    struct sockaddr_in sin;
    char linea[SIZE];
    int cuanto, L;
    if ((sock = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
        error("socket");
    sin.sin_family = AF_INET;
    sin.sin_port = htons(PORT);
    sin.sin_addr.s_addr = INADDR_ANY; //Cualquier interface.
    if( bind(sock, (sad)&sin, sizeof sin) < 0)
        error("bind");
    L = sizeof (sin);
    if((cuanto = recvfrom(sock, linea, SIZE, 0, (sad)&sin, &L)) < 0)
        error("recv from");
    //linea[0]++;
    if( sendto(sock, linea, cuanto, 0, (sad)&sin, L) < 0)
        error ("send to");
    close(sock);
    return 0; //FINALIZACION EXITOSA
}
/* Fin Archivo: hola-mundo-servidor-udp.c */
```

Clase 05-09-2006: Historia de internet – TCP/IP:

Que se compila mediante la orden:

gcc -Wall hola-mundo-servidor-udp.c -o hola-mundo-servidor-udp.out

Ahora veamos el **código fuente** de lo que será el **PROCESO CLIENTE**:

```
/* Archivo: hola-mundo-cliente-udp.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

/* DEFINICIONES */
#define PORT 5000
#define SIZE 1024

/* SINONIMOS */
typedef struct sockaddr * sad;

/* FUNCIONES */
void error (char *s)
{
    perror(s);
    exit (-1);
}

/* FUNCION PRINCIPAL */
int main (int argc, char **argv)
{
    if(argc < 2)
    {
        fprintf(stderr, "Uso: %s ipaddr\n", argv[0]);
        exit(-1);
    }
    int sock;
    struct sockaddr_in sini, sino;
    char linea[SIZE];
    int cuanto, L;
    if ((sock = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
        error("socket");
    sino.sin_family = AF_INET;
    sino.sin_port = htons(PORT);
    inet_pton(AF_INET, argv[1], &sino.sin_addr);
    //Posix To Network. Leer: man 3 inet_ntop
```

Clase 05-09-2006: Historia de internet – TCP/IP:

/* Continuacion Archivo: hola-mundo-cliente-udp.c */

Que se compila mediante la orden:

gcc -Wall hola-mundo-cliente-udp.c -o hola-mundo-cliente-udp.out

Ahora invocaremos a cada programa ejecutable desde diferentes consolas virtuales y veremos:

- Para ejecutar el programa servidor, y así ver como funciona el **PROCESO SERVIDOR (SUB-PROTOCOLO UDP)** tecleamos en una de las consolas:
./hola-mundo-servidor-udp.out
- Para ejecutar el programa cliente, y así ver cómo se desenvuelve el **PROCESO CLIENTE (SUB-PROTOCOLO UDP)** tecleamos:
./hola-mundo-cliente-udp.out 127.0.0.1

Una vez que se han **ejecutado ambos PROCESOS** veremos **en la consola del PROCESO CLIENTE** el siguiente mensaje:

**De 127.0.0.1 : 5000 --> Mensaje[Hola mundo!
]**

A continuación aclararemos algunas cuestiones relacionadas a estos ejemplos:
En la página de manual en línea de comandos: **man 7 ip** se explica:

IP(7) Manual del Programador de Linux IP(7)

NOMBRE

ip - Implementación Linux del protocolo IPv4

SINOPSIS

```
#include <sys/socket.h>
#include <net/netinet.h>

tcp_socket = socket(PF_INET, SOCK_STREAM, 0);
raw_socket = socket(PF_INET, SOCK_RAW, protocol);
udp_socket = socket(PF_INET, SOCK_DGRAM, protocol);
```

DESCRIPCIÓN

Linux implementa el Protocolo de Internet (Internet Protocol, IP), version 4, descrito en RFC791 y RFC1122. ip contiene una implementación de multidestino del Nivel 2 según el RFC1112. También contiene **un enrutador IP incluyendo un filtro de paquetes.**

La **interfaz del programador es compatible con la de los conectores BSD**. Para más información sobre conectores, vea **socket(7)**.

Un **conector IP se crea llamando a** la función **socket(2)** de la forma

socket(PF_INET, socket_type, protocol).

Los **tipos de conectores válidos** son

SOCK_STREAM para abrir un **conector tcp(7)**,

SOCK_DGRAM para abrir un **conector udp(7)** o

SOCK_RAW para abrir un **conector directo (raw(7)) para acceder al protocolo IP directamente.**

protocol es el **protocolo IP en la cabecera IP a enviar o recibir**. Los **únicos valores válidos para protocol** son

0 y IPPROTO_TCP para conectores TCP, y

0 y IPPROTO_UDP para conectores UDP.

Para **SOCK_RAW** debe **especificar un protocolo IP válido del IANA definido por** uno de los números asignados en el **RFC1700**.

NOTA:

La entidad **IANA** es la “**INTERNET ASSIGNED NUMBERS AUTHORITY**”
Más información en: <http://www.iana.org>

Cuando un **proceso** quiere **recibir** nuevos

paquetes de entrada o conexiones,

debe

enlazar un conector a una dirección de la interfaz local usando bind(2).

Sólo se puede ligar **un conector IP a un par (dirección, puerto)** dado.

Cuando en la llamada a bind se especifica INADDR_ANY, el conector será ligado a todas las interfaces locales.

Cuando se llama a **listen(2)** o **connect(2)** con un **conector no enlazado**, el **conector será automáticamente ligado a un puerto aleatorio libre** cuya dirección local sea **INADDR_ANY**.

Una dirección local de conector TCP que haya sido enlazada, no estará disponible durante un cierto tiempo después de que se cierre, a menos que se haya activado la opción SO_REUSEADDR. Se debe tener cuidado al usar esta opción ya que hace que TCP sea menos fiable.

FORMATO DE LAS DIRECCIONES

Una dirección de conector IP (dirección de un socket IP) se define como una combinación de una dirección de interfaz IP y un número de puerto.

El protocolo IP básico no proporciona números de puerto. Estos son implementados por protocolos de un nivel más alto como udp(7) y tcp(7). En los conectores directos (raw(7)), a sin_port se le asigna el protocolo IP.

```
struct sockaddr_in
{
    sa_family_t sin_family; /* familia de direcciones: AF_INET */
    u_int16_t    sin_port;   /* puerto con los bytes en el orden de red*/
    struct in_addr sin_addr; /* dirección de Internet */
};

/* Dirección de Internet.*/
struct in_addr
{
    u_int32_t s_addr; /* dirección con los bytes en el orden de red */
};
```

A sin_family siempre se le asigna el valor AF_INET. Este valor es necesario. En Linux 2.2, la mayoría de las funciones de red devuelven EINVAL cuando se ha omitido este valor.

sin_port contiene el puerto con los bytes en orden de red. Los números de puerto por debajo de 1024 se llaman puertos reservados. Sólo los procesos con identificador de usuario efectivo 0 o la capacidad CAP_NET_BIND_SERVICE pueden realizar enlaces mediante bind(2) a estos conectores.

Dese cuenta que el protocolo IPv4 puro no posee como tal el concepto de puerto. Estos son implementados por protocolos superiores como tcp(7) y udp(7).

sin_addr es

la dirección IP del anfitrión (host).

El miembro **s_addr** de **struct in_addr** contiene la dirección de la interfaz del anfitrión con los bytes en orden de red.

Sólo se debería acceder a **in_addr** usando las funciones de biblioteca

inet_aton(3),

inet_addr(3) y

inet_makeaddr(3),

o directamente mediante el

mecanismo de resolución de nombres (vea gethostbyname(3)).

Las direcciones IPv4 se dividen en direcciones unidestino, de difusión y multidestino.

Las direcciones

unidestino especifican una **única interfaz de un anfitrión**,

de difusión especifican **todos los anfitriones de una red** y

multidestino identifican a **todos los anfitriones de un grupo multidestino**.

Sólo se pueden

enviar datagramas a o

recibir datagramas de

direcciones de difusión cuando está activa la opción de conector **SO_BROADCAST**.

En la implementación actual, los **conectores orientados a conexión (sockets TCP)** sólo pueden usar direcciones unidestino.

Dese cuenta que la

dirección y el puerto

se almacenan

siempre en orden de red.

En particular, esto significa que necesita llamar a

htons(3) con el número que se ha asignado al puerto.

Todas las funciones de manipulación de dirección/puerto en la biblioteca estándar trabajan en orden de red.

Existen varias **direcciones especiales**:

INADDR_LOOPBACK

(127.0.0.1) siempre se refiere al ordenador local a través del dispositivo 'loopback'.

INADDR_ANY	(0.0.0.0) significa cualquier dirección para enlazar.
INADDR_BROADCAST	(255.255.255.255) significa cualquier ordenador y, por razones históricas, tiene el mismo efecto en el enlace que INADDR_ANY.

OPCIONES DE LOS CONECTORES

IP soporta algunas opciones de conector específicas del protocolo que se pueden configurar con **setsockopt(2)** y leer con **getsockopt(2)**.

El nivel de opciones de conector para IP es **SOL_IP**.

Una opción entera booleana es cero cuando es falsa y cualquier otra cosa cuando es cierta.

IP_OPTIONS

Establece u obtiene las opciones IP a enviar con cada paquete desde este conector. Los **argumentos son punteros a un buffer de memoria que contiene las opciones y la longitud de las opciones.** La llamada

setsockopt(2) establece las opciones IP asociadas a un conector. El tamaño máximo de opción para IPv4 es de 40 bytes. Vea **RFC791** para las **opciones permitidas**.

Cuando el paquete inicial de petición de conexión para un conector **SOCK_STREAM (SOCKET TCP)** contiene opciones IP, las opciones IP se configurarán automáticamente al valor de las opciones del paquete inicial con las cabeceras de enrutamiento invertidas. No se permite que los paquetes de entrada cambien las opciones después de que la conexión se haya establecido. El procesamiento de todas las opciones de enrutamiento de la fuente de entrada está desactivado por defecto y se puede activar usando la **sysctl accept_source_route**. Otras opciones, como las marcas de tiempo, todavía se siguen manejando. Para los conectores de **datagramas (SOCKETS UDP)**, las opciones IP sólo pueden ser configuradas por el usuario local.

Llamar a **getsockopt(2)** con **IP_OPTIONS** coloca en el buffer proporcionado las opciones IP actuales usadas para enviar.

IP_PKTINFO

Pasa un mensaje auxiliar **IP_PKTINFO** que contiene una estructura **pktinfo** que proporciona alguna información sobre los paquetes de entrada. Esto sólo funciona para conectores orientados a **datagramas (SOCKETS UDP)**.

```
struct in_pktinfo
{
    unsigned int    ipi_ifindex; /* Índice de la interfaz */

    struct in_addr ipi_spec_dst;
    /* Dirección de destino del enrutamiento */

    struct in_addr ipi_addr;
    /* Dirección de destino en la cabecera */
};
```

ipi_ifindex es el **índice de la interfaz en la que se recibió el paquete.**

ipi_spec_dst es la **dirección de destino de la entrada de la tabla de enrutamiento** y

ipi_addr es la **dirección de destino en la cabecera del paquete.**

Si se pasa **IP_PKTINFO** a **sendmsg(2)**, el **paquete de salida se enviará a través de la interfaz especificada en ipi_ifindex con la dirección de destino indicada en ipi_spec_dst.**

IP_RECVTOS

Cuando está activa, se pasa el mensaje auxiliar **IP_TOS** con los paquetes de entrada. Contiene un byte que especifica el campo **Tipo de Servicio/Precedencia** de la cabecera del paquete.
Espera una opción entera booleana.

IP_RECVTTL

Cuando esta opción está activa, pasa un mensaje de control **IP_RECVTTL** con el campo "**tiempo de vida**" (**time to live**) del paquete recibido dado por un byte.
No soportada por conectores SOCK_STREAM (SOCKETS TCP).

IP_RECVOPTS

Pasa todas las opciones IP de entrada al usuario en un mensaje de control **IP_OPTIONS**. La cabecera de enrutamiento y otras opciones ya las completa el anfitrión local.
No soportada para conectores SOCK_STREAM (SOCKETS TCP).

IP_RETOPTS

Identica a **IP_RECVOPTS** pero devuelve opciones directas sin procesar cuyas marcas de tiempo y opciones del registro de ruta no son completadas por este anfitrión.

IP_TOS

Establece o devuelve el campo

Tipo de Servicio (Type-Of-Service, TOS) a enviar con cada paquete IP creado desde este conector. Se usa para priorizar los paquetes en la red. **TOS es un byte.** Existen algunas opciones TOS estándares definidas:

IPTOS_LOWDELAY para minizar los retrasos en el caso de tráfico interactivo,

IPTOS_THROUGHPUT para optimizar el rendimiento,

IPTOS_RELIABILITY para optimizar la fiabilidad e

IPTOS_MINCOST que se debería usar para "datos de relleno" donde no tenga sentido una transmisión lenta.

Como mucho, se puede especificar uno de estos valores TOS. Los otros bits son inválidos y se limpiarán. Por defecto, Linux envía primero datagramas **IPTOS_LOWDELAY** pero el comportamiento exacto depende de la disciplina de encolamiento configurada. Algunos niveles de prioridad alta pueden necesitar un identificador de usuario efectivo 0 o la capacidad **CAP_NET_ADMIN**. La prioridad también se puede configurar de una manera independiente del protocolo mediante la opción de conector (**SOL_SOCKET, SO_PRIORITY**) (vea **socket(7)**).

IP_TTL

Establece u obtiene el campo "tiempo de vida" actual que se envía en cada paquete enviado desde este conector.

IP_HDRINCL

Cuando está activa, el usuario proporciona una cabecera IP delante de los datos de usuario.

Sólo válida para conectores **SOCK_RAW**.

Vea **raw(7)** para más información.

Cuando esta opción está activa los valores configurados mediante **IP_OPTIONS, IP_TTL** y **IP_TOS** se ignoran.

IP_RECVERR

Habilita el paso adicional fiable de mensajes de error. Cuando se activa en un conector de datagramas todos los errores generados se encolarán en una cola de errores por conector.

Cuando el usuario recibe un error procedente de una operación con un conector, se puede recibir el error llamando a **recvmsg(2)** con la opción **MSG_ERRQUEUE** activa. La estructura **sock_extended_err** que describe el error se pasará en un mensaje

auxiliar con el tipo **IP_RECVERR** y el nivel **SOL_IP**. Esto es útil para el manejo fiable de errores en conectores no conectados. La parte de datos recibida de la cola de errores contiene el paquete de error.

IP usa la estructura **sock_extended_err** como sigue:

a **ee_origin** se le asigna el valor **SO_EE_ORIGIN_ICMP** para errores recibidos en un paquete ICMP o **SO_EE_ORIGIN_LOCAL** para errores generados localmente.

A **ee_type** y **ee_code** se les asignan los campos tipo y código de la cabecera ICMP.

ee_info contiene la MTU descubierta para errores EMSGSIZE.

ee_data no se usa actualmente.

En el caso de un error originado en la red, todas las opciones IP (IP_OPTIONS, IP_TTL, etc.) activas en el conector y contenidas en el paquete de error, se pasan como mensajes de control.

El contenido útil del paquete que ha provocado el error se devuelve como datos normales.

En el caso de conectores **SOCK_STREAM**, **IP_RECVERR** tiene una semántica ligeramente diferente. En lugar de guardar los errores para cuando expire el siguiente plazo de tiempo, pasa todos los errores de entrada inmediatamente al usuario. Esto podría ser útil para conexiones TCP breves que necesitan un manejo rápido de errores. Use esta opción con cuidado: hace que TCP no sea fiable al no permitirle recuperarse adecuadamente de los cambios de enrutamiento y de otras condiciones normales, y rompe la especificación del protocolo. Dese cuenta que TCP no posee una cola de errores. **MSG_ERRQUEUE** es ilegal en conectores **SOCK_STREAM**. Por tanto, todos los errores son devueltos sólo por funciones de conector o mediante **SO_ERROR**.

Para conectores directos (raw), **IP_RECVERR** activa el paso de todos los errores ICMP recibidos a la aplicación. En otro caso, sólo se informa de los errores que se producen en conectores conectados.

Esta opción establece u obtiene un valor booleano entero. Por defecto, **IP_RECVERR** está desactivada.

IP_PMTU_DISCOVER

Establece o recibe la configuración del "descubrimiento de la MTU de la ruta" para el conector. Cuando se activa, Linux realizará el descubrimiento de la MTU de la ruta en este conector tal y

como se define en RFC1191. La opción de "no fragmentar" se activa en **todos los datagramas de salida.** El **valor global** por defecto del sistema **se controla mediante la sysctl ip_no_pmtu_disc** para los **conectores SOCK_STREAM** y para todos los demás está **desactivado.** Para conectores que **no son SOCK_STREAM** es **responsabilidad del usuario enpaquetar los datos en trozos de tamaño MTU y realizar la retransmisión si es necesario.** El **núcleo rechazará aquellos paquetes que sean más grandes que la MTU de ruta conocida si esta opción** está **activa (con EMSGSIZE).**

Opciones del descubrimiento de la MTU de la ruta	Significado
IP_PMTUDISC_WANT	Usar configuraciones por ruta.
IP_PMTUDISC_DONT	Nunca realizar el descubrimiento de la MTU de la ruta.
IP_PMTUDISC_DO	Realizar siempre el descubrimiento de la MTU de la ruta.

Cuando se **activa el descubrimiento de la MTU de la ruta**, el **núcleo** automáticamente **memoriza la MTU de la ruta por anfitrión de destino.** Cuando se está conectado a un extremo específico mediante **connect(2)**, se puede obtener convenientemente la MTU de la ruta conocida actualmente usando la opción de conector **IP_MTU** (por ejemplo, después de que haya ocurrido un error **EMSGSIZE**). La **MTU puede cambiar con el tiempo.** Para conectores no orientados a conexión con muchos destinos, también se puede acceder a la nueva MTU usando la cola de errores (vea **IP_RECVERR**). Se encolará un nuevo error para cada actualización que llegue de la MTU.

Mientras se está realizando el descubrimiento de la MTU, se pueden perder paquetes iniciales de los conectores de datagramas. Las aplicaciones que usan UDP deben ser conscientes de esto y no tenerlo en cuenta para sus estrategias de retransmisión de paquetes.

Para iniciar el proceso de descubrimiento de la MTU de la ruta en conectores no orientados a conexión, es posible comenzar con un tamaño grande de datagramas (con longitudes de bytes de hasta 64KB en las cabeceras) y dejar que se reduzca mediante actualizaciones de la MTU de la ruta.

Para obtener una estimación inicial de la MTU de la ruta, conecte un conector de datagramas a una dirección de destino usando **connect(2)** y obtenga la MTU llamando a **getsockopt(2)** con la opción **IP_MTU**.

IP_MTU

Obtiene la **MTU de la ruta** conocida actualmente para el conector actual. Sólo válida cuando el conector ha sido conectado.
Devuelve un entero. Sólo válida para getsockopt(2).

IP_ROUTER_ALERT

Pasar a este conector todos los paquetes "a reenviar" que tengan activa la opción "alarma del enrutador IP" (IP Router Alert). Sólo válida para conectores directos. Esto es útil, por ejemplo, para demonios RSVP en el espacio de usuario. Los paquetes interceptados no son reenviados por el núcleo, es responsabilidad de los usuarios enviarlos de nuevo. Se ignora el enlace del conector, tales paquetes sólo son filtrados por el protocolo. Espera una opción entera.

IP_MULTICAST_TTL

Establece o lee el valor "tiempo de vida" (time-to-live, TTL) de los paquetes multidestino de salida para este conector. Es muy importante para los paquetes multidestino utilizar el TTL más pequeño posible. El valor por defecto es 1 lo que significa que los paquetes multidestino no abandonarán la red local a menos que el programa de usuario lo solicite explícitamente. El argumento es un entero.

IP_MULTICAST_LOOP

Establece o lee un argumento entero booleano que indica si los paquetes multidestino enviados deben o no ser devueltos a los conectores locales.

IP_ADD_MEMBERSHIP

Unirse a un grupo multidestino. El argumento es una estructura struct ip_mreqn.

```
struct ip_mreqn
{
    struct in_addr imr_multiaddr;
    /* Dirección IP del grupo multidestino */

    struct in_addr imr_address;
    /* Dirección IP de la interfaz local */

    int imr_ifindex; /* Índice de la interfaz */
};
```

imr_multiaddr

contiene la dirección del grupo multidestino al que la aplicación se quiere unir o quiere dejar. Debe ser una dirección multidestino válida.

imr_address	es la dirección de la interfaz local con la que el sistema debe unirse al grupo multidestino. Si es igual a INADDR_ANY el sistema elige una interfaz adecuada.
imr_ifindex	es el índice de la interfaz que debe unirse a o dejar el grupo imr_multiaddr , o 0 para indicar cualquier interfaz .

Por compatibilidad, todavía **se soporta la antigua estructura ip_mreq**. Difiere de **ip_mreqn** sólo en que no incluye el campo **imr_ifindex**. Ésta **opción sólo es válida para setsockopt(2)**.

IP_DROP_MEMBERSHIP

Dejar un grupo multidestino. El **argumento** es una **estructura ip_mreqn o ip_mreq similar a la de IP_ADD_MEMBERSHIP**.

IP_MULTICAST_IF

Establece el dispositivo local para un conector multidestino. El **argumento** es una **estructura ip_mreqn o ip_mreq similar a la de IP_ADD_MEMBERSHIP**.

Cuando se pasa una opción de conector inválida, se **devuelve el error ENOPROTOOPT**.

SYSCTLs

El **protocolo IP soporta** la interfaz **sysctl** para **configurar** algunas **opciones globales**. Se **puede acceder** a las sysctls **leyendo o escribiendo los ficheros /proc/sys/net/ipv4/* o usando la interfaz sysctl(2)**.

ip_default_ttl

Establece el valor "tiempo de vida" (TTL) por defecto de los paquetes de salida. Éste se **puede cambiar para cada conector con la opción IP_TTL**.

ip_forward

Activa el reenvío IP con una opción booleana. También se **puede configurar el reenvío IP interfaz a interfaz**.

ip_dynaddr

Activa la reescritura dinámica de la dirección del conector y de las entradas de enmascaramiento (masquerading) para cuando cambie la dirección de la interfaz. Esto es **útil para interfaces dialup (como las telefónicas) con direcciones IP cambiantes**. **0 significa no reescritura, 1 la activa y 2 activa el modo verbose**.

ip_autoconfig

No documentado.

ip_local_port_range

Contiene **dos enteros** que **definen el intervalo de puertos locales** por defecto **reservados para los conectores**. La reserva comienza con el primer número y termina con el segundo. Dése cuenta que **estos no deben entrar en conflicto con los puertos usados por el enmascaramiento (aunque se trate el caso)**. También, las **elecciones arbitrarias pueden producir problemas con algunos filtros de paquetes del cortafuegos** que realizan suposiciones sobre los puertos locales en uso. El **primer número debe ser al menos >1024, mejor >4096 para evitar conflictos con puertos bien conocidos y para minimizar los problemas con el cortafuegos**.

ip_no_pmtu_disc

Si está **activa**, por defecto **no realiza el descubrimiento de la MTU de la ruta para los conectores TCP**. El descubrimiento de la MTU de la ruta puede fallar si se encuentran en la ruta cortafuegos mal configurados (como los que pierden todos los paquetes ICMP) o interfaces mal configuradas (por ejemplo, un enlace punto a punto en donde ambos extremos no se ponen de acuerdo en la MTU). Es mejor arreglar los enrutadores defectuosos de la ruta que desactivar globalmente el descubrimiento de la MTU de la ruta ya que el no realizarlo incurre en un alto coste para la red.

ipfrag_high_thresh, ipfrag_low_thresh

Si el **número de fragmentos IP encolados alcanza el valor ipfrag_high_thresh**, la **cola se recorta al valor ipfrag_low_thresh**. Contiene un **entero con el número de bytes**.

ip_always_defrag

[Nueva con la versión 2.2.13 del núcleo. En anteriores versiones del núcleo la característica era controlada en tiempo de compilación por la opción CONFIG_IP_ALWAYS_DEFRAG]

Cuando esta **opción booleana se habilita (es distinta de 0)** los **fragmentos de entrada (partes de paquetes IP que aparecen cuando algún anfitrión entre el origen y el destino decidió que los paquetes eran demasiado grandes y los dividió en pedazos) se reensamblarán (desfragmentarán) antes de ser procesados, incluso aunque vayan a ser reenviados**.

Sólo habilítelo cuando tenga en funcionamiento un cortafuegos que sea el único enlace de su red o un proxy transparente. Nunca lo active para un enrutador u ordenador normal. En otro caso, se **puede perturbar la comunicación fragmentada cuando los**

fragmentos viajen a través de diferentes enlaces. La desfragmentación también **tiene un alto coste de tiempo de CPU y de memoria.**

Esto **se activa automágicamente cuando se configura un enmascaramiento o un proxy transparente.**

neigh/*

Vea **arp(7)**.

IOCTLs

Todas las **ioctls** descritas en **socket(7)** se aplican a IP.

Las **ioctls para configurar el cortafuegos se documentan en la página ipfw(7) del paquete ipchains.**

Las **ioctls para configurar los parámetros de los dispositivos genéricos se describen en netdevice(7).**

NOTAS

Tenga **mucho cuidado con la opción SO_BROADCAST** (no es privilegiada en Linux). **Es fácil sobrecargar la red realizando difusiones sin tomar precauciones.** Para los **nuevos protocolos de aplicación** es **mejor usar un grupo multidestino que usar la difusión.** La difusión no está recomendada.

Otras **implementaciones de conectores BSD proporcionan las opciones** de conector **IP_RCVDSTADDR** y **IP_RECVIF** para **obtener la dirección de destino y la interfaz de los datagramas recibidos.** Linux posee la opción más general **IP_PKTINFO** para la misma tarea.

ERRORES

ENOTCONN

La operación sólo está definida en conectores conectados, pero el conector no lo está.

EINVAL

Se ha pasado un **argumento inválido.** Para las operaciones de envío, éste se puede producir al enviar a una ruta blackhole.

EMSGSIZE

El datagrama es mayor que una MTU de la ruta y no puede ser fragmentado.

EACCES

El usuario **ha intentado ejecutar una operación sin los permisos necesarios.** Estos incluyen:

enviar un paquete a una dirección de difusión sin haber activado la opción SO_BROADCAST,
enviar un paquete a través de una ruta prohibida,
modificar la configuración del cortafuegos sin tener la capacidad CAP_NET_ADMIN ni un identificador de usuario efectivo 0,
y realizar un enlace a un puerto reservado sin la capacidad CAP_NET_BIND_SERVICE ni un identificador de usuario efectivo 0.

EADDRINUSE

Se ha intentado el enlace a una dirección ya en uso.

ENOMEM y ENOBUFS

No hay suficiente memoria disponible.

ENOPROTOOPT y EOPNOTSUPP

Se han pasado una opción de conector inválida.

EPERM

El usuario no tiene permiso para establecer una prioridad alta, cambiar la configuración o enviar señales al proceso o grupo solicitado.

EADDRNOTAVAIL

Se ha solicitado una interfaz inexistente o la dirección fuente solicitada no es local.

EAGAIN

La operación se bloquearía en un conector bloqueante.

ESOCKTNOSUPPORT

El conector no está configurado o se ha solicitado un tipo de conector desconocido.

EISCONN

Se ha llamado a connect(2) con un conector ya conectado.

EALREADY

Ya se está realizando una operación de conexión sobre un conector no bloqueante.

ECONNABORTED

Se ha cerrado la conexión durante un accept(2).

EPIPE

La conexión se ha cerrado inesperadamente o el otro extremo la ha cancelado.

ENOENT

Se ha llamado a SIOCGSTAMP con un conector en donde no ha llegado ningún paquete.

EHOSTUNREACH

Ninguna entrada válida de la tabla de enrutamiento coincide con la dirección de destino. Este error puede ser provocado por un mensaje ICMP procedente de un enrutador remoto o por la tabla local de enrutamiento.

ENODEV

Dispositivo de red no disponible o incapaz de enviar paquetes IP.

ENOPKG

No se ha configurado un subsistema del núcleo.

ENOBUFS, ENOMEM

No hay suficiente memoria libre. Esto a menudo significa que la reserva de memoria está limitada por los límites del búfer de conectores, no por la memoria del sistema, aunque esto no es coherente al 100%.

Los protocolos superpuestos pueden generar otros errores.

Vea **tcp(7)**, **raw(7)**, **udp(7)** y **socket(7)**.

VERSIONES

IP_PKTINFO, IP_MTU, IP_PMTU_DISCOVER, IP_PKTINFO, IP_RECVERR y **IP_ROUTER_ALERT** son opciones nuevas del núcleo 2.2 de Linux.

struct ip_mreqn es nueva en Linux 2.2. Linux 2.0 sólo soportaba **ip_mreq**.

Las **sysctls** se introdujeron en la versión 2.2 de Linux.

COMPATIBILIDAD

Por compatibilidad con Linux 2.0, todavía **se soporta** la sintaxis obsoleta

socket(PF_INET, SOCK_RAW, protocol)

para abrir un conector de paquetes (**packet(7)**). Se recomienda **no usar** esta sintaxis y debería reemplazarse por

socket(PF_PACKET, SOCK_RAW, protocol).

La principal diferencia es la nueva estructura de direcciones `sockaddr_ll` para la información genérica de la capa de enlace en lugar de la antigua `sockaddr_pkt`.

FALLOS

Existen demasiados valores de error inconsistentes.

No se han descrito las `ioctl`s para configurar las opciones de interfaz específicas de IP y las tablas ARP.

AUTORES

Esta página de manual fue **escrita por Andi Kleen**.

VÉASE TAMBIÉN

`sendmsg(2)`, `recvmsg(2)`, `socket(7)`, `netlink(7)`, `tcp(7)`, `udp(7)`, `raw(7)`, `ipfw(7)`.

RFC791 para la especificación IP original.

RFC1122 para los requerimientos IPv4 para los anfitriones.

RFC1812 para los requerimientos IPv4 para los enrutadores.

Página man de Linux

11 mayo 1999

IP(7)

NOTA: La **unidad máxima de transferencia** (*Maximum Transfer Unit - MTU*) es un término de [redes de computadoras](#) que expresa el tamaño en [bytes](#) del [datagrama](#) más grande que puede pasar por una capa de un [protocolo de comunicaciones](#).
Ejemplos de MTU:

- [Ethernet](#): 1500 bytes
- [ATM](#) (AAL5): 8190 bytes
- [FDDI](#): 4470 bytes
- [PPP](#): 576 bytes

Los datagramas pueden pasar por varios tipos de redes con diferentes protocolos antes de llegar a su destino. Por tanto, para que un datagrama llegue sin fragmentación al destino, ha de ser menor o igual que el mínimo MTU de las redes por las que pase.

Veamos ahora la página de manual en línea de comandos: **man 7 tcp**

TCP(7)

Manual del Programador de Linux

TCP(7)

NOMBRE

tcp - Protocolo TCP.

SINOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
tcp_socket = socket(PF_INET, SOCK_STREAM, 0);
```

DESCRIPCIÓN

Ésta es una implementación del protocolo TCP definido en **RFC793**, **RFC1122** y **RFC2001** con las **extensiones NewReno y SACK**. Proporciona una **conexión bidireccional, fiable y orientada a conexión entre dos conectores encima de ip(7)**. TCP garantiza que los datos llegan en orden y retransmite los paquetes perdidos. Genera y comprueba una suma de verificación por paquete para detectar errores de transmisión. TCP no conserva los límites entre registros.

Un conector TCP recién creado no tiene ni dirección remota ni local y no está totalmente especificado. Para crear una conexión TCP de salida use **connect(2)** para establecer una **conexión con otro conector TCP**. Para recibir nuevas conexiones de entrada, primero **enlace el conector a una dirección local y puerto mediante bind(2)** y a continuación llame a **listen(2)** para colocar el conector en **estado de escucha**. Después de esto, se **puede aceptar un nuevo conector para cada conexión de entrada usando accept(2)**. Un **conector sobre el que se han ejecutado con éxito accept o connect** está totalmente especificado y puede transmitir datos. No se pueden transmitir datos a través de conectores en estado de escucha o no conectados todavía.

La **versión 2.2 de Linux soporta las extensiones RFC1323 para TCP de alto rendimiento**. Éstas incluyen **grandes ventanas TCP para soportar enlaces con una alta latencia o gran ancho de banda**. Para usarlas, se deben incrementar los tamaños de los buffers de envío y recepción.

ANALISTA UNIVERSITARIO DE SISTEMAS

SISTEMAS OPERATIVOS – Prof. Macchi Guido

45/

Se pueden configurar globalmente con las

**sysctls net.core.wmem_default y
net.core.rmem_default**,

o para cada conector individual mediante las **opciones de conector SO_SNDBUF y SO_RCVBUF**. Los **tamaños máximos de los buffers de los conectores** están limitados por las **sysctls globales net.core.rmem_max y net.core.wmem_max**. Vea **socket(7)** para más información.

TCP soporta datos urgentes. Los datos urgentes se usan para indicar al receptor que algún mensaje importante es parte del flujo de datos y que debe ser procesado tan pronto como sea posible. Para enviar datos urgentes, especifique la opción **MSG_OOB** en **send(2)**. Cuando se reciben datos urgentes, el núcleo envía una señal **SIGURG** al proceso lector o al proceso o grupo de procesos que han sido configurados para el conector usando la **ioctl FIOCSPGRP o FIOCSETOWN**. Cuando la opción de conector **SO_OOBINLINE** está activa, los datos urgentes se colocan en el flujo normal de datos (y se pueden examinar mediante la **ioctl SIOCATMARK**). En otro caso, sólo se pueden recibir cuando se ha configurado la opción **MSG_OOB** en **sendmsg(2)**.

FORMATO DE LAS DIRECCIONES

TCP está construido encima de IP (vea ip(7)). Los **formatos de dirección definidos por ip(7) se aplican a TCP**. TCP sólo soporta comunicaciones punto a punto. La difusión y el envío multidestino no están soportados.

SYSCTLs

Estas **sysctls** pueden ser **accedida mediante los ficheros /proc/sys/net/ipv4/* o con la interfaz sysctl(2)**. Además, las **mayoría de las sysctls de IP también se aplican a TCP**. Vea ip(7).

tcp_window_scaling

Habilita la adaptación de ventanas TCP RFC1323.

tcp_sack

Habilita los reconocimientos selectivos TCP RFC2018.

tcp_timestamps

Habilita las marcas de tiempo TCP RFC1323.

tcp_fin_timeout

Cantidad de segundos a esperar un paquete FIN final antes de que el conector sea cerrado a la fuerza. Esto es estrictamente una violación de la especificación TCP pero necesario para evitar ataques de "denegación de servicio".

tcp_keepalive_probes

Número máximo de sondeos "sigue vivo" (keep-alive) de TCP a enviar antes de darse por vencido. Este tipo de sondeos sólo se envía cuando la opción de conector SO_KEEPALIVE está activa.

tcp_keepalive_time

Cantidad de segundos después de que no se haya transmitido ningún dato antes de que se envíen sondeos "sigue vivo" a través de una conexión. El valor por defecto es de 10800 segundos (3 horas).

tcp_max_ka_probes

Cuántos sondeos "sigue vivo" se envían por cada periodo atrasado del cronómetro. Para evitar ráfagas, este valor no debería ser demasiado alto.

tcp_stdurg

Activa la interpretación estricta del RFC793 para el campo "puntero urgente" de TCP. Por defecto, se usa la interpretación de "puntero urgente" compatible con BSD, apuntando al primer byte después de los datos urgentes. La interpretación RFC793 es hacerlo apuntar al último byte de los datos urgentes. Activar esta opción puede conducir a problemas de interoperabilidad.

tcp_syncookies

Habilita los "syncookies" de TCP. Se **debe compilar el núcleo con CONFIG_SYN_COOKIES**. Los "syncookies" protegen a un conector de la sobrecarga cuando intentan llegar demasiadas conexiones. Puede ser que las máquinas clientes ya no sean capaces de detectar una máquina sobrecargada con un plazo de tiempo pequeño cuando se activan los "syncookies".

tcp_max_syn_backlog

Longitud de la cola de acumulación por conector. A partir de la versión 2.2 de Linux, la **acumulación especificada en listen(2) sólo indica la longitud de la cola de acumulación de los conectores ya establecidos**. Esta sysctl establece la **longitud máxima de la cola de conectores todavía no establecidos** (en estado SYN_RECV) para cada conector en escucha. Cuando llegan más solicitudes de conexión, Linux comienza a perder paquetes. Cuando se activan los "syncookies" todavía se responde a los paquetes y esta valor se ignora.

tcp_retries1

Define **cuántas veces se retransmite una respuesta a una solicitud de conexión TCP antes de darse por vencido**.

tcp_retries2

Define **cuántas veces se retransmite un paquete TCP en el estado "establecido" antes de dejarlo**.

ANALISTA UNIVERSITARIO DE SISTEMAS

SISTEMAS OPERATIVOS – Prof. Macchi Guido

47/

tcp_syn_retries

Define **cuántas veces se intenta enviar un paquete SYN inicial a un anfitrión remoto antes de abandonar y devolver un error**. El valor debe estar por debajo de 255. Éste sólo es el **plazo de tiempo para las conexiones de salida**. Para las conexiones de entrada el número de retransmisiones se define en **tcp_retries1**.

tcp_retrans_collapse

Intentar enviar paquetes totalmente formados durante las retransmisiones. Esto se usa para solucionar temporalmente los fallos TCP de algunas pilas de protocolos.

OPCIONES DE LOS CONECTORES

Para establecer u obtener la **opción de un conector TCP**, llame a **getsockopt(2)** para leerla o a **setsockopt(2)** para escribirla, **asignando SOL_TCP** al argumento de la familia del conector. Además, la mayoría de las **opciones de conector SOL_IP** son válidas para **conectores TCP**. Para más información vea **ip(7)**.

TCP_NODELAY

Desactiva el algoritmo de Nagle. Esto significa que **los paquetes se envían siempre tan pronto como sea posible y no se introduce ningún retraso innecesario, a costa de más paquetes en la red.** Espera una opción booleana entera.

TCP_MAXSEG

Establece o recibe el tamaño máximo de segmento para los paquetes TCP de salida. Si se configura esta opción antes del establecimiento de la conexión, también cambia el valor **MSS** comunicado al otro extremo en el paquete inicial. Los valores mayores que la MTU de la interfaz se ignoran y no tienen ningún efecto.

TCP_CORK

No se envían tramas parciales cuando está activa. Todas las tramas parciales encoladas se envían cuando esta opción se desactiva de nuevo. Esto es útil para ir añadiendo cabeceras antes de llamar a **sendfile(2)** o para optimizar el rendimiento. Esta opción no se puede combinar con **TCP_NODELAY**.

IOCTLs

Estas **ioctls** pueden ser accedidas usando **ioctl(2)**. La sintaxis correcta es:

```
int value;  
error = ioctl(tcp_socket, ioctl_type, &value);
```

FIONREAD

Devuelve la cantidad de datos encolados sin leer en el buffer de recepción. El argumento es un **puntero a un entero**.

SIOCATMARK

Devuelve cierto cuando el programa de usuario ya ha recibido todos los datos urgentes. Esto se usa junto con **SO_OOBINLINE**. El argumento es un **puntero a un entero** para el **resultado de la comprobación**.

TIOCOUTQ

Devuelve la cantidad de datos sin enviar en la cola de envío del conector en el puntero a un valor entero pasado.

MANEJO DE ERRORES

Cuando se produce un **error de red**, **TCP intenta reenviar el paquete**. Si no tiene éxito después de un cierto tiempo, informa o bien de un **error ETIMEDOUT** o bien del último error recibido sobre esta conexión.

Algunas **aplicaciones necesitan una notificación más rápida del error**. Esto se puede hacer con la opción de conector **IP_RECVERR** del nivel **SOL_IP**. Cuando se activa esta opción, todos los errores de entrada son pasados inmediatamente al programa de usuario. Use esta opción

con cuidado (hace que TCP sea menos tolerante a cambios de enrutamiento y a otras condiciones de red normales).

NOTAS

Cuando se produce un **error, al configurar una conexión, durante la escritura en un conector, sólo se produce una señal SIGPIPE cuando está activa la opción de conector SO_KEEPOPEN.**

TCP no posee verdaderos datos fuera de orden, posee datos urgentes. En Linux esto significa que si el otro extremo envía datos fuera de orden recientes, los anteriores datos urgentes se insertarán como datos normales en el flujo (incluso cuando SO_OOBINLINE no está activa). Esto difiere de las pilas de protocolo basadas en BSD.

Linux usa por defecto una interpretación del campo puntero urgente compatible con BSD. Esto viola el RFC1122 pero se necesita por interoperatividad con otras pilas. Se puede cambiar con la sysctl tcp_stdurg.

ERRORES

EPIPE

El otro extremo ha cerrado el conector inesperadamente o se ha intentado leer de un conector desconectado.

ETIMEDOUT

El otro extremo no ha reconocido los datos retransmitidos después de cierto tiempo.

EAFNOSUPPORT

El tipo de dirección de conector pasado en sin_family no es AF_INET.

TCP también puede devolver cualquier error definido por ip(7) o la capa de conectores genéricos.

FALLOS

No se han documentado todos los errores.

No se ha descrito IPv6.

No se han descrito las opciones de proxy transparente.

VERSIONES

Las **sysctls** son nuevas en la versión 2.2 de Linux. **IP_RECVERR** es una característica nueva de la versión 2.2 de Linux. **TCP_CORK** es nueva en la versión 2.2.

VÉASE TAMBIÉN

socket(7), socket(2), ip(7), sendmsg(2), recvmsg(2).

RFC793 para la especificación de TCP.

RFC1122 para los requisitos de TCP y una descripción del algoritmo Nagle.

RFC2001 para algunos algoritmos de TCP.

Página man de Linux

25 abril 1999

TCP(7)

También observemos las particularidades de la página de manual en línea de comandos de: **man 7 udp**

UDP(7)

Manual del Programador de Linux

UDP(7)

NOMBRE

udp - Protocolo UDP sobre IPv4.

SINOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
udp_socket = socket(PF_INET, SOCK_DGRAM, 0);
```

DESCRIPCIÓN

Ésta es una implementación del **protocolo UDP (User Datagram Protocol) descrito en RFC768**. Implementa un **servicio de paquetes de datagramas no fiable y sin conexión**. Los paquetes pueden ser reordenados o duplicados antes de que lleguen. UDP genera y comprueba sumas de verificación (checksums) para detectar errores de transmisión.

Cuando se crea un conector (socket) UDP, sus direcciones local y remota están sin especificar. Se pueden enviar datagramas inmediatamente usando sendto(2) o sendmsg(2) con una dirección de destino válida como argumento. Cuando se llama a connect(2) sobre el conector, se envía la dirección de destino por defecto y a partir de ese momento se pueden enviar datagramas usando send(2) o write(2) sin especificar una dirección de destino. Todavía es posible realizar envíos a otros destinos pasando una dirección a sendto(2) o sendmsg(2). Para poder recibir paquetes, primero se debe ligar el conector a una dirección local usando bind(2). Cuando éste no sea el caso, la capa de conectores le asignará automáticamente un puerto local en la primera petición de recepción del usuario.

Todas las **operaciones de recepción sólo devuelven un paquete. Cuando el paquete es más pequeño que el buffer pasado, sólo se**

devuelven los datos del paquete y, cuando es mayor, el paquete se trunca y la bandera MSG_TRUNC se activa.

Se pueden enviar o recibir opciones IP usando las opciones de conectores descritas en ip(7). Estas son procesadas por el núcleo sólo cuando está activa la sysctl adecuada (pero todavía se pasan al usuario incluso cuando está desactivada). Vea ip(7).

Cuando en un envío está activa la opción MSG_DONTROUTE, la dirección de destino debe referirse a la dirección de una interfaz local y el paquete sólo se envía a esa interfaz.

UDP fragmenta un paquete cuando su longitud total excede la MTU (Unidad de Transmisión Máxima) de la interfaz. Una alternativa de red más amigable es usar el descubrimiento de la MTU de la ruta como se describe en la sección IP_PMTU_DISCOVER de ip(7).

FORMATO DE DIRECCIÓN

UDP usa el formato de dirección sockaddr_in de IPv4 descrito en ip(7).

MANEJO DE ERRORES

Todos los errores fatales serán pasados al usuario como un resultado de error incluso cuando el conector no esté conectado. Este comportamiento difiere de muchas otras implementaciones de conectores BSD que no pasan ningún error al menos que el conector esté conectado. El comportamiento de Linux viene mandado por el RFC1122.

Por compatibilidad con código anterior es posible activar la opción SO_BSDCOMPAT de SOL_SOCKET para recibir errores remotos (excepto EPROTO y EMSGSIZE) sólo cuando el conector se ha conectado. Es mejor arreglar el código para manejar adecuadamente los errores que habilitar esta opción. Los errores generados localmente siempre se pasan.

Cuando se activa la opción IP_RECVERR todos los errores se almacenan en la cola de errores de conector y se pueden recibir mediante recvmsg(2) con la opción MSG_ERRQUEUE activa.

ERRORES

Una operación de enviar o recibir sobre un conector UDP puede devolver cualquier error documentado en socket(7) o ip(7).

ECONNREFUSED No se ha asociado un receptor a la dirección de destino. Esto podría ser provocado por un paquete anterior enviado por el conector.

VERSIONES

IP_RECVERR es una nueva característica de la versión 2.2 de Linux.

CREDITOS

Esta página de manual fue **escrita por Andi Kleen**.

VÉASE TAMBIÉN

ip(7), socket(7), raw(7).

RFC768 para el protocolo UDP.

RFC1122 para los requisitos del anfitrión (host).

RFC1191 para una descripción del descubrimiento de la MTU de la ruta.

Página man de Linux

2 octubre 1998

UDP(7)

También hechamos un vistazo a la “man page”: **man 7 raw**
RAW(7) Manual del Programador de Linux

RAW(7)

NOMBRE

raw, SOCK_RAW - Conectores directos (raw) IPv4 de Linux

ANALISTA UNIVERSITARIO DE SISTEMAS

SISTEMAS OPERATIVOS – Prof. Macchi Guido

52/

SINOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
raw_socket = socket(PF_INET, SOCK_RAW, int protocol);
```

DESCRIPCIÓN

Los **conectores directos** permiten implementar nuevos protocolos IPv4 en el espacio de usuario. Un conector directo **recibe o envía el datagrama crudo sin incluir cabeceras del nivel de enlace**.

La **capa IPv4 genera una cabecera IP cuando se envía un paquete, a menos que se active la opción IP_HDRINCL en el conector. Cuando se activa, el paquete debe contener una cabecera IP. En la recepción, la cabecera IP siempre está incluida en el paquete.**

Sólo los procesos con un identificador de usuario efectivo de 0 o la capacidad CAP_NET_RAW pueden abrir conectores directos.

Todos los paquetes o errores cuyo protocolo coinciden con el número protocolo especificado por el conector directo, se pasan a este conector. Para una lista de los protocolos permitidos vea los números asignados en RFC1700 y getprotobynumber(3).

Un **protocolo IPPROTO_RAW** implica que **IP_HDRINCL** está activa y recibe todos los protocolos IP. El envío no está permitido.

Campos de cabecera IP modificados en el envío por IP_HDRINCL	
Suma de comprobación IP	Siempre se rellena
Dirección fuente	Se rellena cuando es cero
Identificador del paquete	Se rellena cuando es cero
Longitud total	Siempre se rellena

Si se especifica **IP_HDRINCL** y la cabecera IP tiene una dirección de destino distinta de cero, la dirección de destino del conector se utiliza para enrutar el paquete. Cuando se especifica **MSG_DONTROUTE**, la dirección de destino debe referirse a una interfaz local, de lo contrario, se realiza una búsqueda en la tabla de enrutamiento, aunque se ignoran las rutas que se dirigen a enrutadores.

Si no se activa **IP_HDRINCL**, se pueden configurar las opciones de la cabecera IP de los conectores directos con **setsockopt(2)**. Vea **ip(7)** para más información.

En Linux 2.2 todas las **opciones y campos de las cabeceras IP** se pueden configurar usando las opciones de los conectores IP. Esto significa que los conectores directos son usualmente necesarios sólo para protocolos nuevos o protocolos que no tienen una interfaz de usuario (como ICMP).

Cuando se recibe un paquete, se pasa a cualquier conector directo que haya sido asociado a su protocolo antes de que sea pasado al manejador de cualquier otro protocolo (por ejemplo, los módulos de protocolo del núcleo).

FORMATO DE LAS DIRECCIONES

Los conectores directos usan la estructura de direcciones estándar **sockaddr_in** definida en **ip(7)**.

El campo **sin_port** se podría usar para especificar el número de protocolo IP, pero en Linux 2.2 se ignora al enviar y siempre debería valer 0 (vea FALLOS). Para los paquetes de entrada, a **sin_port** se le asigna el protocolo del paquete. Vea el fichero cabecera **<netinet/in.h>** para protocolos IP válidos.

OPCIONES DE LOS CONECTORES

Las opciones de los conectores directos se pueden configurar con **setsockopt(2)** y leer con **getsockopt(2)**, pasando la opción de familia **SOL_RAW**.

ICMP_FILTER

Activa un filtro especial para los conectores directos asociados al protocolo **IPPROTO_ICMP**.

El valor tiene un bit activo para cada tipo de mensaje ICMP que debe filtrarse. Por defecto, no se filtra ningún mensaje ICMP.

Además, **se soportan todas las opciones SOL_IP de ip(7) válidas.**

NOTAS

Los **conectores directos fragmentan un paquete cuando su longitud total excede la MTU de la interfaz** (no obstante, vea FALLOS). Una alternativa más rápida y favorable para la red es implementar el descubrimiento del MTU de la ruta como se describe en la sección **IP_PMTU_DISCOVER** de ip(7).

Se puede asociar un conector directo a una dirección local específica usando la llamada **bind(2)**. Si no está asociado, se reciben todos los paquetes con el protocolo IP especificado. Además, se puede asociar un conector directo a un dispositivo de red específico usando **SO_BINDTODEVICE**. Vea **socket(7)**.

Un **conector IPPROTO_RAW** es sólo de envío. Si verdaderamente quiere recibir todos los paquetes IP, use un conector **packet(7)** con el protocolo **ETH_P_IP**. Dese cuenta que, a diferencia de los conectores directos, los conectores de paquete no reensamblan fragmentos IP.

ANALISTA UNIVERSITARIO DE SISTEMAS

SISTEMAS OPERATIVOS – Prof. Macchi Guido

54/

Si quiere recibir todos los paquetes ICMP para un conector de **datagramas**, normalmente es mejor usar **IP_RECVERR** en ese conector particular. Vea **ip(7)**.

Los **conectores directos pueden interceptar todos los protocolos IP de Linux, incluso protocolos como ICMP o TCP que poseen un módulo de protocolo dentro del núcleo**. En este caso, los paquetes se pasan tanto al módulo del núcleo como al conector (o conectores) directo. No se debería confiar en esto en programas transportables ya que muchas otras implementaciones de conectores BSD tienen limitaciones aquí.

Linux nunca cambia las cabeceras pasadas por el usuario (salvo para llenar algunos campos de valor 0 como se ha descrito en **IP_HDRINCL**). Esto es diferente de muchas otras implementaciones de conectores directos.

Generalmente, los **conectores directos son poco transportables** y deberían evitarse en programas destinados a ser transportables.

En el **envío a través de conectores directos** se debería tomar el protocolo IP de **sin_port**. Esta capacidad se perdió en Linux 2.2. La forma de solucionar esto es usar **IP_HDRINCL**.

MANEJO DE ERRORES

Sólo se pasan al usuario los errores generados por la red cuando el conector está conectado o está activa la opción IP_RECVERR. Para conectores conectados, sólo se pasan **EMSGSIZE** y **EPROTO** por compatibilidad. Con **IP_RECVERR** todos los **errores de red se guardan en la cola de errores.**

ERRORES

EMSGSIZE

Paquete demasiado grande. O bien el **descubrimiento del MTU de la ruta está activo** (la opción **IP_PMTU_DISCOVER** de los conectores) o bien el **tamaño del paquete excede el máximo tamaño de 64KB permitido por IPv4.**

EACCES El usuario ha intentado enviar a una dirección de difusión sin tener activa la opción de difusión en el conector.

EPROTO Ha llegado un **error ICMP informando de un problema de parámetros.**

EFAULT Se ha pasado una dirección de memoria inválida.

EOPNOTSUPP

Se ha pasado a la llamada socket una opción inválida (como **MSG_OOB**).

EINVAL Argumento inválido.

EPERM El usuario no tiene permiso para abrir conectores directos. Sólo los procesos con un identificador de usuario efectivo de 0 o el atributo **CAP_NET_RAW** pueden hacerlo.

VERSIONES

IP_RECVERR y **ICMP_FILTER** son nuevos en la versión 2.2 de Linux. Ambos son extensiones de Linux y no deberían usarse en programas transportables.

La versión 2.0 de Linux activaba cierta compatibilidad fallo a fallo con BSD en el código de los conectores directos cuando se activaba la opción **SO_BSDCOMPAT**. Ésto se ha eliminado en la versión 2.2.

FALLOS

No se han descrito las extensiones de proxy transparente.

Cuando se activa la opción **IP_HDRINCL**, los **datagramas no se fragmentan y están limitados por la MTU de la interfaz.** Ésta es una limitación de la versión 2.2 de Linux.

La posibilidad de especificar el protocolo IP en **sin_port** durante el envío desapareció en Linux 2.2.

Siempre se usa el protocolo al que se enlazó el conector o el que se especificó en la llamada inicial a **socket(2)**.

AUTOR

Esta página de manual fue escrita por **Andi Kleen**.

VÉASE TAMBIÉN

ip(7), socket(7), recvmsg(2), sendmsg(2).

RFC1191 para el descubrimiento del MTU de la ruta.

RFC791 y el fichero cabecera <linux/ip.h> para el protocolo IP.

Página man de Linux

2 octubre 1998

RAW(7)

Nos dedicaremos ahora a ultimar detalles acerca de la página del manual en línea de comandos: **man 7 socket**

SOCKET(7)

Manual del Programador de Linux

SOCKET(7)

NOMBRE

socket - Interfaz de conectores (sockets) de Linux

SINOPSIS

```
#include <sys/socket.h>
mysocket = socket(int socket_family, int socket_type, int protocol);
```

DESCRIPCIÓN

Esta página de manual describe la **interfaz de usuario de la capa de conectores de red de Linux**. Los **conectores** compatibles con BSD **son la interfaz uniforme entre el proceso de usuario y las pilas de protocolos de red dentro del núcleo**. Los **módulos de protocolo se agrupan en familias de protocolos** como **PF_INET, PF_IPX y PF_PACKET**, y **tipos de conectores** como **SOCK_STREAM o SOCK_DGRAM**. Vea **socket(2)** para obtener más información sobre las familias y los tipos.

FUNCIONES DE LA CAPA DE CONECTORES

Estas funciones las usa el proceso de usuario para enviar o recibir paquetes y para realizar otras operaciones con conectores. Para más información vea sus páginas de manual respectivas.

socket(2) crea un conector, **connect(2) conecta** un conector a una dirección de conector remota, la función **bind(2) enlaza** un conector a una dirección de conector local, **listen(2) indica al conector que se aceptarán nuevas conexiones** y **accept(2)** se usa **para obtener un nuevo conector con una nueva conexión de entrada**. **socketpair(2)** devuelve dos

conectores anónimos conectados (sólo implementado para unas pocas familias locales como **PF_UNIX**)

send(2), **sendto(2)** y **sendmsg(2)** envían datos a través de un conector y **recv(2)**, **recvfrom(2)** y **recvmsg(2)** reciben datos de un conector. **poll(2)** y **select(2)** esperan la llegada de datos o la posibilidad de enviar datos. Además, se pueden usar las operaciones estándares de E/S como **write(2)**, **writev(2)**, **sendfile(2)**, **read(2)** y **readv(2)** para leer y escribir datos.

getsockname(2) devuelve la dirección de un conector local y **getpeername(2)** devuelve la dirección de un conector remoto. **getsockopt(2)** y **setsockopt(2)** se usan para configurar o consultar opciones de los protocolos o las capas. **ioctl(2)** se puede usar para configurar o consultar otras opciones determinadas.

close(2) se usa para cerrar un conector. **shutdown(2)** cierra partes de una conexión bidireccional entre conectores.

Las búsquedas o las llamadas a **pread(2)** o **pwrite(2)** con una posición distinta de cero, no están soportadas en conectores.

Es posible realizar E/S no bloqueante con conectores activando la opción **O_NONBLOCK** sobre el descriptor de fichero de un conector usando **fcntl(2)**. **O_NONBLOCK** se hereda durante una llamada a **accept**. A continuación, todas las operaciones que normalmente se bloquearían devolverán (usualmente) el error **EAGAIN**. **connect(2)** devuelve un error **EINPROGRESS** en este caso. Más tarde, el usuario puede esperar diferentes eventos mediante **poll(2)** o **select(2)**.

<i>Eventos de E/S</i>		
Evento	Opción de poll	Ocurrencia
Lectura	POLLIN	Han llegado nuevos datos.
Lectura	POLLIN	Se ha completado una nueva solicitud de conexión (para conectores orientados a conexión).
Lectura	POLLHUP	El otro extremo ha iniciado una solicitud de desconexión.
Lectura	POLLHUP	Se ha roto una conexión (sólo para protocolos orientados a conexión).
		Cuando se escribe en el conector, también se envía la señal SIGPIPE.

<i>Eventos de E/S</i>		
Escritura	POLLOUT	El conector tiene suficiente espacio en el buffer de envío para escribir nuevos datos.
Lectura/Escritura	POLLIN POLLOUT	Ha finalizado un connect(2) de salida.
Lectura/Escritura	POLLERR	Se ha producido un error asíncrono.
Lectura/Escritura	POLLHUP	El otro extremo ha cerrado una dirección de la conexión.
Excepción	POLLPRI	Han llegado datos fuera de orden, lo que hace que se envíe la señal SIGURG.

Una alternativa a **poll/select** es dejar que el núcleo informe de los eventos a la aplicación mediante una señal **SIGIO**. Para ello, se debe activar la opción **FASYNC** en el descriptor de fichero de un conector mediante **fcntl(2)** y se debe instalar un manejador de señales válido para **SIGIO** mediante **sigaction(2)**. Vea la discusión sobre SEÑALES más abajo.

OPCIONES DE LOS CONECTORES

Estas opciones de conector se pueden **configurar usando setsockopt(2)** y **consultar con getsockopt(2)** con el **nivel de conectores** fijado a **SOL_SOCKET** para todos los conectores:

SO_KEEPALIVE

Habilita el **envío de mensajes "sigue vivo"** (keep-alive) en conectores orientados a conexión(TCP).

Espera una **opción booleana entera**.

SO_OOBINLINE

Si se habilita esta opción, los **datos fuera de orden** se colocan directamente en el **flujo de recepción de datos**. En **otro caso**, los **datos fuera de orden** sólo se pasan cuando se activa la opción **MSG_OOB** durante la recepción.

SO_RCVLOWAT y SO SNDLOWAT

Especifican el **número mínimo de bytes** en el buffer para que la capa de conectores **pase los datos al protocolo** (**SO SNDLOWAT**) o **al usuario durante la recepción** (**SO RCVLOWAT**).

Estos dos **valores no se pueden cambiar** en Linux y sus **argumentos de tamaño siempre tienen el valor de 1 byte**. **getsockopt** es **capaz de leerlos**. **setsockopt** siempre devolverá **ENOPROTOOPT**.

SO_RCVTIMEO y SO_SNDFTIMEO

Especifica los plazos de tiempo (timeouts) para enviar y recibir antes de informar de un error. En Linux el valor de ambos es fijo y viene dado por una configuración específica del protocolo y no se pueden ni leer ni modificar. Su funcionalidad se puede emular usando `alarm(2)` o `setitimer(2)`.

SO_BSDCOMPAT

Habilita la compatibilidad fallo a fallo con BSD. Esto lo usa sólo el módulo del protocolo UDP y está previsto que se elimine en el futuro. Cuando está activa, los errores ICMP recibidos por un conector UDP no se pasan al programa de usuario. Linux 2.0 también habilitaba las opciones de compatibilidad fallo a fallo con BSD (cambio aleatorio de las cabeceras, omisión de la opción de difusión) para los conectores directos con esta opción, pero esto se ha eliminado en la versión 2.2 de Linux. Es mejor corregir los programas de usuario que habilitar esta opción.

SO_PASSCRED

Habilita o deshabilita la recepción del mensaje de control `SCM_CREDENTIALS`. Para más información, vea `unix(7)`.

SO_PEERCRED

Devuelve las credenciales del proceso externo conectado a este conector. Sólo útil para conectores `PF_UNIX`. Vea `unix(7)`. El argumento es una estructura `ucred`. Esta opción sólo es válida para `getsockopt`.

SO_BINDTODEVICE

Enlaza este conector a un dispositivo particular, como "eth0", especificado en el nombre de interfaz pasado. Si el nombre es una cadena vacía o la longitud de las opciones es cero, se elimina el enlace entre el dispositivo y el conector. La opción pasada es una cadena (terminada en \0) de longitud variable con el nombre de la interfaz, con un tamaño máximo de `IFNAMSIZ`. Si el conector está ligado a una interfaz, éste sólo procesará los paquetes recibidos desde la interfaz particular.

SO_DEBUG

Activa la depuración de los conectores. Sólo permitida para los procesos con la capacidad `CAP_NET_ADMIN` o un identificador de usuario efectivo 0.

SO_REUSEADDR

Indica que las reglas usadas para validar las direcciones proporcionadas en una llamada `bind(2)` deben permitir la reutilización de las direcciones locales. Para los conectores `PF_INET` esto significa que un conector se puede enlazar a una dirección, excepto cuando hay un conector activo escuchando asociado a la dirección. Cuando el conector que está escuchando está asociado a `INADDR_ANY` con un

puerto específico, entonces no es posible realizar enlaces a este puerto para ninguna dirección local.

SO_TYPE

Obtiene el tipo de conector como un valor entero (como **SOCK_STREAM**). Sólo puede ser leído con **getsockopt**.

SO_DONTROUTE

No enviar a través de un enrutador, sólo enviar a ordenadores directamente conectados. Se puede conseguir el mismo efecto activando la opción **MSG_DONTROUTE** en una operación **send(2)** sobre un conector. Espera una opción booleana entera.

SO_BROADCAST

Establece o consulta la opción de difusión. Cuando está activa, los conectores de datagramas reciben los paquetes enviados a una dirección de difusión y se les permite enviar paquetes a una dirección de difusión. Esta opción no tiene efecto en conectores orientados a conexión (**TCP**).

SO_SNDBUF

Establece u obtiene, en bytes, el máximo buffer de envío de un conector. El valor por defecto se configura con la **sysctl wmem_default** y el máximo valor permitido se establece con la **sysctl wmem_max**.

SO_RCVBUF

Establece u obtiene, en bytes, el máximo buffer de recepción de un conector. El valor por defecto se configura con la **sysctl rmem_default** y el máximo valor permitido se establece con la **sysctl rmem_max**.

SO_LINGER

Establece u obtiene la opción **SO_LINGER**. El argumento es una estructura **linger**.

```
struct linger {
    int l_onoff; /* activar/desactivar demora */
    int l_linger; /* segundos de demora */
};
```

Cuando esta opción está activa, un **close(2)** o **shutdown(2)** no regresará hasta que todos los mensajes encolados para el conector hayan sido enviados con éxito o se haya alcanzado el plazo de tiempo de demora. En otro caso, la llamada regresa inmediatamente y el cierre se realiza en segundo plano. Cuando el conector se cierra como parte de una llamada **exit(2)**, siempre se demora en segundo plano.

SO_PRIORITY

Asigna a todos los paquetes a enviar a través de este conector la prioridad definida por el protocolo. Linux usa este valor para ordenar las colas de red: los paquetes con una prioridad mayor se pueden procesar primero dependiendo de la disciplina de encolamiento del dispositivo seleccionado. Para ip(7), esto también establece el campo "tipo de servicio IP" (TOS) para los paquetes de salida.

SO_ERROR

Obtiene y borra el error de conector pendiente. Sólo válida para **getsockopt**. Espera un entero.

SEÑALES

Cuando se escribe en un conector orientado a conexión (TCP) que ha sido cerrado (por el extremo local o remoto) se envía una señal **SIGPIPE** al proceso escritor y se devuelve el valor de error EPIPE. No se envía la señal cuando la llamada para escritura especifica la opción **MSG_NOSIGNAL**.

Cuando se solicita con la **fcntl FIOCSETOWN** o la **ioctl SIOCSGRP**, la señal **SIGIO** se envía cuando se produce un evento de E/S. Es posible usar **poll(2)** o **select(2)** en el manejaror de la señal para averiguar sobre qué conector se produjo el evento. Una alternativa (en Linux 2.2) es configurar una señal de tiempo real usando la **fcntl F_SETSIG**. Se llamará al manejaror de la señal de tiempo real con el descriptor de fichero en el campo **si_fd** de su estructura **siginfo_t**. Vea **fcntl(2)** para más información.

Bajo determinadas circunstancias (por ejemplo, varios procesos accediendo a un único conector), la condición que ha provocado la señal **SIGIO** puede haber desaparecido ya cuando el proceso reaccione a la señal. Si esto ocurre, el proceso debería esperar de nuevo ya que Linux reenviará la señal **SIGIO** más tarde.

SYSCTLs

Se puede acceder a las sysctls fundamentales de red de los conectores usando los ficheros **/proc/sys/net/core/*** o mediante la interfaz **sysctl(2)**.

rmem_default

contiene el valor por defecto, en bytes, del buffer de recepción de un conector.

rmem_max

contiene el tamaño máximo, en bytes, del buffer de recepción de un conector que el usuario puede establecer usando la opción de conector **SO_RCVBUF**.

wmem_default

contiene el **valor por defecto, en bytes, del buffer de envío de un conector.**

wmem_max

contiene el **tamaño máximo, en bytes, del buffer de envío de un conector** que un **usuario puede configurar usando** la opción de conector **SO_SNDBUF.**

message_cost y **message_burst**

configuran el filtro de cubetas de fichas usado para cargar los mensajes límites de advertencia (to load limit warning messages) **provocados por eventos de red externos.**

netdev_max_backlog

Número máximo de paquetes en la cola de entrada global.

optmem_max

Longitud máxima de los datos auxiliares y de los datos de control del usuario, como los iovecs por conector.

IOCTLs

Se puede **acceder a estas ioctl's usando ioctl(2):**

```
error = ioctl(ip_socket, ioctl_type, &value_result);
```

SIOCGSTAMP

Devuelve una struct timeval con la marca de tiempo recibida del último paquete pasado al usuario. Esto es útil para realizar medidas exactas del tiempo de ida y vuelta o tiempo de viaje. Vea **setitimer(2)** para una descripción de struct timeval.

SIOCSPGRP

Configura el proceso o grupo de procesos al que **enviar la señal SIGIO o SIGURG cuando termina una operación de E/S asíncrona o hay disponibles datos urgentes.** El argumento es un puntero a un pid_t. Si el argumento es positivo, las señales se envian a ese proceso. Si es negativo, las señales se envían al grupo de procesos cuyo identificador es el valor absoluto del argumento. El proceso sólo puede seleccionar a él mismo o a su propio grupo de procesos para que reciban las señales, a menos que posea la capacidad CAP_KILL o un identificador de usuario efectivo 0.

FIOASYNC

Modifica la opción **O_ASYNC** para habilitar o deshabilitar el modo de E/S asíncrona del conector. El modo de E/S asíncrona significa que se producirá una señal **SIGIO**, o la señal establecida mediante **F_SETSIG**, cuando se produzca un nuevo evento de E/S.

El argumento es una **opción booleana entera**.

SIOCGPGRP

Obtiene el **proceso o grupo de procesos actual** que recibe las señal **SIGIO** o **SIGURG**, o **0** cuando no hay ninguno.

Fcntls válidas:

FIOCGETOWN

Idéntica a la ioctl **SIOCGPGRP**.

FIOCSETOWN

Idéntica a la ioctl **SIOCSPGRP**.

OBSERVACIONES

Linux asume que se usa la mitad del buffer de envío/recepción para estructuras internas del núcleo.

Por tanto, las **sysctl**s son el doble de lo que se puede observar en última instancia.

FALLOS

No se han documentado las **opciones de conector SO_ATTACH_FILTER** y **SO_DETACH_FILTER** de **CONFIG_FILTER**. La interfaz sugerida para usarlas es la biblioteca **libpcap**.

VERSIONES

SO_BINDTODEVICE se introdujo en la versión 2.0.30 de Linux. **SO_PASSCRED** es nueva en la versión 2.2 del núcleo. Las **sysctl**s son nuevas en Linux 2.2.

AUTORES

Esta página de manual fue escrita por **Andi Kleen**.

VÉASE TAMBIÉN

socket(2), **ip(7)**, **setsockopt(2)**, **getsockopt(2)**, **packet(7)**, **ddp(7)**

SOCKET(2) Manual del Programador de Linux SOCKET(2)

NOMBRE

socket - crea un extremo de una comunicación

SINOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int socket(int dominio, int tipo, int protocolo);
```

DESCRIPCIÓN

Socket **crea un extremo de una comunicación y devuelve un descriptor.**

El parámetro **dominio** especifica un **dominio de comunicaciones**. Esto selecciona la **familia de protocolo que se usará para la comunicación**. Estas **familias se definen en <sys/socket.h>**.

Los **formatos actualmente reconocidos** incluyen:

Nombre	Propósito	Página de manual
PF_UNIX,PF_LOCAL	Comunicación local	unix(7)
PF_INET	Protocolos de Internet IPv4	ip(7)
PF_INET6	Protocolos de Internet IPv6	
PF_IPX	Protocolos IPX - Novell	
PF_NETLINK	Dispositivo de la interfaz de usuario del núcleo	netlink(7)
PF_X25	Protocolo ITU-T X.25 / ISO-8208	x25(7)
PF_AX25	Protocolo AX.25 de radio para aficionados	
PF_ATMPVC	Acceso directo a PVCs ATM	
PF_APPLETALK	Appletalk	ddp(7)
PF_PACKET	Interfaz de paquetes de bajo nivel	packet(7)

El **conector tiene el tipo** indicado, **que especifica la semántica de la comunicación**. Los **tipos** definidos **en la actualidad son**:

SOCK_STREAM

Proporciona **flujos de bytes** basados en una **conexión bidireccional secuenciada, confiable**. Se puede admitir un mecanismo de **transmisión de datos fuera-de-banda**.

SOCK_DGRAM

Admite **datagramas (mensajes no confiables, sin conexión, de una longitud máxima fija)**.

SOCK_SEQPACKET

Proporciona un camino de **transmisión de datos basado en conexión bidireccional secuenciado, confiable**, para **datagramas de longitud máxima fija**; se **requiere un consumidor para leer un paquete entero con cada llamada al sistema de lectura**.

SOCK_RAW

Proporciona **acceso directo a los protocolos de red**.

SOCK_RDM

Proporciona una **capa de datagramas fiables** que **no garantiza el orden**.

SOCK_PACKET

Obsoleto y no debería utilizarse en programas nuevos. Vea **packet(7)**.

Algunos tipos de conectores pueden no ser implementados por todas las familias de protocolos. Por ejemplo, **SOCK_SEQPACKET no está implementado para AF_INET**.

El **protocolo** especifica un protocolo particular para ser usado con el conector. Normalmente sólo existe un protocolo que admite **un tipo particular de conector dentro de una familia de protocolos** dada. Sin embargo, es posible que puedan existir varios protocolos, en cuyo caso un protocolo particular puede especificarse de esta manera. El **número de protocolo a emplear es específico al "dominio de comunicación"** en el que la comunicación va a tener lugar; vea **protocols(5)**. Consulte **getprotoent(3)** para ver **cómo asociar unas cadenas con el nombre de un protocolo a un número de protocolo**.

Los **conectores del tipo SOCK_STREAM** son **flujos de bytes bidireccionales, similares a tuberías**, que **no conservan los límites de registro**. Un conector de flujo debe estar en un estado conectado antes de que cualquier dato pueda ser enviado o recibido en él. Se crea una conexión con otro conector mediante la llamada `connect(2)`. Una vez hecha la conexión, los datos pueden transferirse utilizando llamadas `read(2)` y `write(2)` o alguna variante de las llamadas `send(2)` y `recv(2)`. Cuando una sesión se ha completado, se puede efectuar un `close(2)`. Los **datos fuera-de-banda** pueden **transmitirse** también como se describe en `send(2)` y `recibirse` según se describe en `recv(2)`.

Los **protocolos de comunicaciones que implementan un SOCK_STREAM aseguran** que los **datos no se pierden ni se duplican**. Si un trozo de dato para el cual el protocolo de la pareja tiene espacio de búfer no puede ser transmitido satisfactoriamente en un período razonable de tiempo, entonces la conexión se considera muerta. Cuando se activa **SO_KEEPALIVE** en el conector el protocolo comprueba de una manera específica del protocolo **si el otro extremo todavía está vivo**. Se lanza una señal **SIGPIPE** si un proceso envía o recibe en un flujo roto; esto provoca que procesos simples, que no manejan la señal, acaben.

Los **conectores SOCK_SEQPACKET emplean las mismas llamadas al sistema que los SOCK_STREAM**. La única diferencia es que las llamadas a **read(2)** devolverán solamente la cantidad de datos pedidos, y los que queden en el paquete que llega se perderán. También se conservarán todos los límites de mensaje en los datagramas que lleguen.

Los conectores **SOCK_DGRAM** y **SOCK_RAW** permiten el envío de datagramas a los correspondientes nombrados en llamadas a **send(2)**. Los datagramas se reciben generalmente con **recvfrom(2)**, que devuelve el siguiente datagrama con su dirección de retorno.

SOCK_PACKET es un **tipo de conector obsoleto** para recibir paquetes crudos directamente desde el manejador de dispositivo. Use **packet(7)** en su lugar.

Una llamada a **fcntl(2)** con el argumento **F_SETOWN** puede utilizarse para especificar que un grupo de proceso reciba una señal **SIGURG** cuando lleguen los datos fuera-de-banda o la señal **SIGPIPE** cuando una conexión **SOCK_STREAM** se rompa inesperadamente. También puede usarse para configurar el proceso o grupo de procesos que recibirán la E/S y la notificación asíncrona de los eventos de E/S a través de **SIGIO**. Usar **F_SETOWN** es equivalente a una llamada a **ioctl(2)** con el argumento **SIOSETOWN**.

Cuando la red señala una condición de error al módulo del protocolo (por ejemplo, usando un mensaje ICMP para IP) se activa la bandera de error pendiente para el conector. La siguiente operación sobre ese conector devolverá el código de error del error pendiente. Para algunos protocolos es posible habilitar una cola de error por conector para obtener información detallada del error. Vea **IP_RECVERR** en **ip(7)**.

La operación de los conectores se controla por opciones en el nivel de los conectores. Estas opciones se definen en **<sys/socket.h>**. **setsockopt(2)** y **getsockopt(2)** se emplean para establecer y obtener opciones, respectivamente.

VALOR DEVUELTO

Se devuelve un **-1 si ocurre un error; en otro caso el valor devuelto es un descriptor para referenciar el conector.**

ERRORES

EPROTONOSUPPORT

El tipo de protocolo, o el **protocolo especificado, no es reconocido dentro de este dominio.**

ENFILE No hay suficiente memoria en el núcleo para reservar una nueva estructura de conector.

EMFILE Se ha desbordado la tabla de ficheros del proceso.

EACCES Se deniega el permiso para crear un conector del tipo o protocolo especificado.

ENOBUFS o ENOMEM

No hay suficiente memoria disponible. El conector no puede crearse hasta que no queden libres los recursos suficientes.

EINVAL

Protocolo desconocido o familia de protocolo no disponible.

Los módulos de los protocolos subyacentes pueden generar otros errores.

CONFORME A

4.4BSD (la llamada a función socket apareció en 4.2BSD). Generalmente transportable a o desde sistemas no BSD que admitan clones de la capa de conectores de BSD (incluyendo variantes **System V**).

NOTA

Las **constantes evidentes usadas en BSD 4.* para las familias de protocolos son PF_UNIX, PF_INET, etc., mientras que AF_UNIX, etc. se usan para las familias de direcciones.** Sin embargo, ya la **página de manual BSD promete: "La familia de protocolos generalmente es la misma que la familia de direcciones" y los estándares subsiguientes usan AF_* en todas partes.**

FALLOS

SOCK_UUCP todavía no está implementado.

VÉASE TAMBIÉN

accept(2), bind(2), connect(2), getprotoent(3), getsockname(2), getsockopt(2), ioctl(2), listen(2), read(2), recv(2), select(2), send(2), shutdown(2), socketpair(2), write(2)

"An Introductory 4.3 BSD Interprocess Communication Tutorial" está reimpresso en UNIX Programmer's Supplementary Documents Volume 1.

"BSD Interprocess Communication Tutorial" está reimpresso en UNIX Programmer's Supplementary Documents Volume 1.

Página man de Linux

24 abril 1999

SOCKET(2)

Ahora observaremos: **man 7 unix:**

UNIX(7)

Manual del Programador de Linux

UNIX(7)

NOMBRE

unix, PF_UNIX, AF_UNIX, PF_LOCAL, AF_LOCAL - Conectores para la comunicación local entre procesos.

SINOPSIS

```
#include <sys/socket.h>
#include <sys/un.h>

unix_socket = socket(PF_UNIX, type, 0);
error = socketpair(PF_UNIX, type, 0, int *sv);
```

DESCRIPCIÓN

La **familia de conectores PF_UNIX** (también conocida como **PF_LOCAL**) se usa **para comunicar procesos en la misma máquina** de manera eficiente. Los **conectores Unix** pueden ser o bien **anónimos (creados mediante socketpair(2))** o bien **estar asociados con un fichero de tipo conector**. Linux también soporta un espacio de nombres abstracto que es independiente del sistema de ficheros.

Los tipos válidos son **SOCK_STREAM** para un conector orientado a conexión y **SOCK_DGRAM** para un conector orientado a datagramas que conserva las fronteras entre los mensajes. Los **conectores Unix siempre son fiables y no reordenan los datagramas**.

Los conectores Unix **soportan el paso de descriptores de fichero o credenciales de proceso a otros procesos, como datos auxiliares a los datagramas**.

FORMATO DE LAS DIRECCIONES

Una dirección Unix se define como un nombre de fichero en el sistema de fichero o como una cadena única en el espacio de nombres abstracto. Los **conectores creados mediante socketpair(2) son anónimos**. Para **conectores no anónimos** la **dirección del destino se puede configurar usando connect(2)**. La dirección local se puede

configurar usando bind(2). Cuando un conector se conecta y no tiene todavía una dirección local, se genera automáticamente una dirección única en el espacio de nombres abstracto.

```
#define UNIX_PATH_MAX 108

struct sockaddr_un {

    sa_family_t sun_family;
    /* AF_UNIX */

    char     sun_path[UNIX_PATH_MAX];
    /* nombre de la ruta */
};
```

sun_family siempre contiene **AF_UNIX**. **sun_path** contiene el **nombre de ruta (terminado en cero) del conector en el sistema de ficheros**. Si **sun_path** comienza con un byte cero se refiere al **espacio de nombres abstracto mantenido por el módulo del protocolo Unix**. La dirección del conector en este espacio de nombres viene dada por el resto de los bytes en **sun_path**. Dese cuenta que los nombres en el espacio de nombres abstracto no terminan en cero.

OPCIONES DE LOS CONECTORES

Por razones históricas estas opciones de los conectores se especifican con un tipo **SOL_SOCKET**, aunque sean específicas de **PF_UNIX**. Se pueden configurar con **setsockopt(2)** y leer con **getsockopt(2)** especificando **SOL_SOCKET** como familia del conector.

SO_PASSCRED

Habilita la **recepción de las credenciales del proceso emisor en un mensaje auxiliar**. Cuando esta opción está activa y el conector no está conectado aún, se genera automáticamente un nombre único en el espacio de nombres abstracto. Espera una **bandera booleana entera**.

MENSAJES AUXILIARES

Por razones históricas, estos tipos de **mensajes auxiliares se especifican con un tipo SOL_SOCKET**, aunque son **específicos de PF_UNIX**. Para enviarlos, asigne al campo **cmsg_level** de la estructura **cmsghdr** el valor **SOL_SOCKET** y al campo **cmsg_type** el **tipo**. Para más información, vea **cmsg(3)**.

SCM_RIGHTS

Enviar o recibir un **conjunto de descriptores de fichero abiertos a/desde otro proceso**. La parte de datos contiene un **array de enteros con los descriptores de fichero**. Los **descriptores de fichero pasados** se comportan como si hubieran sido creados con **dup(2)**.

SCM_CREDENTIALS

Enviar o recibir credenciales Unix. Esto se puede usar para autenticación. Las credenciales se pasan como un mensaje auxiliar **struct ucred**.

```
struct ucred {
    pid_t pid; /* PID del proceso emisor */
    uid_t uid; /* UID del proceso emisor */
    gid_t gid; /* GID del proceso emisor */
};
```

El núcleo comprueba las credenciales que el emisor especifica. Un proceso con identificador de usuario efectivo 0 puede especificar valores que no coincidan con los suyos. El emisor debe especificar su propio identificador de proceso (a menos que posea la capacidad **CAP_SYS_ADMIN**), sus identificador de usuario, identificador de usuario efectivo o identificador de usuario de conjunto (a menos que posea la capacidad **CAP_SETUID**) y sus identificador de grupo, identificador de grupo efectivo o identificador de grupo de conjunto (a menos que posea la capacidad **CAP_SETGID**). Para recibir un mensaje **struct ucred** la opción **SO_PASSCRED** debe estar activa en el conector.

VERSIONES

SCM_CREDENTIALS y el espacio de nombres abstracto fueron introducidos en la versión 2.2 de Linux y no deberían usarse en programas transportables.

NOTAS

En la implementación de Linux, los conectores que son visibles en el sistema de ficheros respetan los permisos del directorio en el que están. Se pueden cambiar sus propietarios, grupos y permisos.

La creación de un nuevo conector fallará si el proceso no tiene permisos de escritura y búsqueda (ejecución) en el directorio en el que se crea el conector. La conexión al objeto conector requiere permiso de lectura/escritura. Este comportamiento difiere del de muchos sistemas derivados de BSD que ignoran los permisos para los conectores Unix. Por seguridad, los programas transportables no deberían confiar en esta característica.

Ligar un conector con un nombre de fichero crea un conector en el sistema de ficheros que debe ser borrado por el invocador cuando no se necesite más (usando **unlink(2)**). Se aplica la semántica habitual de Unix detrás de una operación de cierre: el conector puede ser desligado en cualquier instante y será finalmente eliminado del sistema de ficheros cuando se cierre la última referencia a él.

Para pasar descriptores de fichero o credenciales necesita enviar/leer al menos un byte.

ERRORES

ENOMEM No hay suficiente memoria.

ECONNREFUSED

Se ha llamado a **connect(2)** con un objeto **conector** que no está **escuchando**. Esto **puede ocurrir cuando no existe el conector remoto** o el **nombre de fichero no es un conector**.

EINVAL Se ha pasado un argumento inválido. Una causa común es olvidar asignar **AF_UNIX** al **campo sun_type de las direcciones pasadas** o que el conector se encuentre en un estado inválido para la operación aplicada.

EOPNOTSUPP

Se ha invocado una operación orientada a conexión sobre un conector no orientado a conexión o se ha intentado usar la opción de "datos fuera de orden".

EPROTONOSUPPORT

El **protocolo** pasado **no es PF_UNIX**.

ESOCKTNOSUPPORT

Tipo de conector desconocido.

EPROTOTYPE

El **tipo del conector remoto no coincide con el tipo del conector local (SOCK_DGRAM contra SOCK_STREAM)**

EADDRINUSE

La **dirección local seleccionada ya está en uso o el objeto conector del sistema de ficheros ya existe**.

EISCONN

Se ha llamado a **connect(2) sobre un conector ya conectado** o se **ha especificado una dirección de destino en un conector conectado**.

ENOTCONN

La **operación del conector necesita una dirección de destino pero el conector no está conectado**.

ECONNRESET

Se **ha cerrado inesperadamente el conector remoto**.

EPIPE Se ha cerrado el conector remoto de un conector orientado a conexión. Si se ha activado, también se enviará una señal **SIGPIPE**. Esto se puede **evitar pasando** la opción **MSG_NOSIGNAL** a **sendmsg(2)** o a **recvmsg(2)**.

EFAULT La **dirección de memoria de usuario no es válida**.

EPERM El emisor ha pasado credenciales inválidas en struct ucred.

La **capa de conectores genérica**, o el **sistema de ficheros** al generar un objeto conector en el sistema de fichero, **pueden producir otros errores**. Vea las páginas de manual adecuadas para más información.

VÉASE TAMBIÉN

**recvmsg(2), sendmsg(2), socket(2), socketpair(2), cmsg(3),
socket(7)**

CREDITOS

Esta página de manual fue **escrita por Andi Kleen**.

Página man de Linux

7 mayo 1999

UNIX(7)

Ahora veremos la página de manual: **man 7 netlink**

NETLINK(7)

Manual del Programador de Linux

NETLINK(7)

NOMBRE

netlink, PF_NETLINK - Comunicación entre el núcleo y el usuario.

SINOPSIS

```
#include <asm/types.h>
#include <sys/socket.h>
#include <linux/netlink.h>
```

```
netlink_socket = socket(PF_NETLINK, socket_type, netlink_family);
```

DESCRIPCIÓN

Netlink se utiliza para transferir información entre los módulos del núcleo y los procesos del espacio de usuario. Consiste en una interfaz basada en conectores estándares para los procesos de usuario y una API del núcleo interna para los módulos del núcleo. La interfaz del núcleo interna no se documenta en esta página de manual. También existe una interfaz netlink obsoleta mediante dispositivos de caracteres netlink. Esta interfaz no se documenta aquí y sólo se proporciona por compatibilidad hacia atrás.

Netlink es un servicio orientado a datagramas. Tanto **SOCK_RAW** como **SOCK_DGRAM** son **valores válidos para socket_type**. Sin embargo, el **protocolo netlink no distingue entre conectores de datagrama y conectores directos (raw)**.

netlink_family selecciona el módulo del núcleo o el grupo netlink con el que comunicarse. Las familias netlink asignadas actualmente son:

NETLINK_ROUTE

Recibe actualizaciones de enrutamiento y puede usarse para modificar la tabla de enrutamiento de IPv4 (vea **rtnetlink(7)**).

NETLINK_FIREWALL

Recibe paquetes enviados por el código del cortafuegos de IPv4.

NETLINK_ARPD

Para gestionar la tabla ARP en el espacio de usuario.

NETLINK_ROUTE6

Recibe y envía actualizaciones a la tabla de enrutamiento de IPv6.

NETLINK_IP6_FW

Para recibir los paquetes que no pasaron las comprobaciones del cortafuegos de IPv6 (actualmente no implementado).

NETLINK_TAPBASE...NETLINK_TAPBASE+15

Son las **instancias del dispositivo ethertap**. El dispositivo ethertap es **un pseudodispositivo de tunel de red que permite simular un manejador ethernet desde el espacio de usuario**.

NETLINK_SKIP

Reservado para ENskip.

NETLINK_USERSOCK

Reservado para futuros protocolos en el espacio de usuario.

Los mensajes netlink consisten en un flujo de bytes con una o más cabeceras **nlmsghdr** y sus cargas útiles asociadas. Para los mensajes multipartes, la primera cabecera y las siguientes tienen activa la opción **NLM_F_MULTI**, excepto la última cabecera, que tiene el tipo **NLMSG_DONE**. El flujo de bytes sólo debería ser accedido con las macros estándares **NLMSG_***. Vea **netlink(3)**.

Netlink no es un protocolo fiable. Intenta hacerlo lo mejor que puede para entregar un mensaje en su destino (o destinos), pero **puede perder mensajes cuando no hay suficiente memoria** o se produce cualquier otro error. **Para una transferencia fiable, el emisor** puede **solicitar un reconocimiento del receptor activando** la opción **NLM_F_ACK**. Un **reconocimiento es un paquete NLMSG_ERROR cuyo campo de error vale 0**. La propia aplicación debe generar reconocimientos para los mensajes recibidos. El núcleo intenta enviar un mensaje **NLMSG_ERROR** para cualquier paquete que falle. Un proceso de usuario también debería seguir estas convenciones.

Cada familia netlink tiene un conjunto de 32 grupos multidestino (multicast). Cuando se llama a bind(2) sobre el conector, se debe

configurar el campo **nl_groups** de **sockaddr_nl** como una máscara de bits de los grupos que se desea escuchar. El valor por omisión para este campo es cero, lo que significa que no se recibirán multidestinos.

Un conector puede enviar un mensaje con varios destinos a cualquiera de los grupos multidestino, asignando a **nl_groups** una máscara de bits de los grupos a los que desea enviar cuando llama a **sendmsg(2)** o hace un **connect(2)**. Sólo los usuarios con un identificador de usuario efectivo 0 o la capacidad **CAP_NET_ADMIN** pueden enviar a o escuchar de un grupo netlink multidestino. Cualquier respuesta a un mensaje recibido por un grupo multidestino se debe enviar de regreso al pid emisor y al grupo multidestino.

```
struct nlmsghdr
{
    __u32   nlmsg_len;
    /* Longitud del mensaje incluyendo la cabecera */

    __u16   nlmsg_type;      /* Contenido del mensaje */
    __u16   nlmsg_flags;     /* Opciones adicionales */
    __u32   nlmsg_seq;       /* Número de secuencia */
    __u32   nlmsg_pid;       /* PID del proceso emisor */
};

struct nlmsgerr
{
    int     error;
    /* número de error negativo o 0 para reconocimientos */

    struct nlmsghdr msg;
    /* cabecera del mensaje que provocó el error */
};
```

Después de cada **nlmsghdr** viene la carga útil. **nlmsg_type** puede ser uno de los tipos de mensajes estándares:

NLMSG_NOOP

Se va a ignorar el mensaje.

NLMSG_ERROR

El mensaje indica un error y la carga útil contiene una estructura **nlmsgerr**.

NLMSG_DONE

El mensaje termina un mensaje multiparte.

Una familia netlink normalmente especifica más tipos de mensajes. Vea las páginas de manual adecuadas para ello. Por ejemplo, **rtnetlink(7)**

para **NETLINK_ROUTE**

Bits de opciones estándares en nlmsg_flags

NLM_F_REQUEST	Poner en todos los mensajes de solicitud
NLM_F_MULTI	El mensaje es parte de un mensaje multiparte terminado mediante NLMSG_DONE
NLM_F_ACK	Responder con un reconocimiento en caso de éxito
NLM_F_ECHO	Hacer eco de esta solicitud

Bits de opciones adicionales para peticiones GET

NLM_F_ROOT	Devolver la tabla completa en lugar de una única entrada.
NLM_F_MATCH	Todavía no implementado.
NLM_F_ATOMIC	Devolver una copia instantánea atómica de la tabla.
NLM_F_DUMP	Todavía no documentado.

Bits de opciones adicionales para peticiones NEW

NLM_F_REPLACE	Reemplazar un objeto existente.
NLM_F_EXCL	No reemplazar si el objeto ya existe.
NLM_F_CREATE	Crear un objeto sin que ya existe.
NLM_F_APPEND	Añadir al final de la lista de objetos.

FORMATOS DE LAS DIRECCIONES

La estructura **sockaddr_nl** describe un **cliente netlink en el espacio de usuario o en el núcleo**. Una **sockaddr_nl** puede ser o bien **unidestino** (**envío a un único igual**) o bien un **envío a grupos netlink (nl_groups distinto de 0)**.

```
struct sockaddr_nl
{
    sa_family_t          nl_family;        /* AF_NETLINK */
    unsigned short        nl_pad;           /* cero */
    pid_t                nl_pid;           /* PID del proceso */

    __u32                 nl_groups;
    /* Máscara de grupos multidirección */
};
```

nl_pid es el **PID del netlink en el espacio de usuario o 0 si el destino está en el núcleo**.

nl_groups es una **máscara de bits** con **cada bit representando a un número de grupo netlink**.

FALLOS

Esta página de manual no está completa.

NOTAS

Normalmente es **mejor usar netlink mediante libnetlink que mediante la interfaz de bajo nivel del núcleo.**

VERSIONES

La interfaz de conectores netlink es una nueva característica de la versión 2.2 de Linux.

La versión 2.0 de Linux soportaba una interfaz netlink más primitiva basada en dispositivos (que todavía está disponible por compatibilidad). Esta interfaz obsoleta no se describe aquí.

VÉASE TAMBIÉN

cmsg(3), rtnetlink(7), netlink(3).

ftp://ftp.inr.ac.ru/ip-routing/iproute2* para libnetlink

Página man de Linux

27 abril 1999

NETLINK(7)

Ahora veremos la página de manual: **man 7 x25**

X25(4)

Manual del Programador de Linux

X25(4)

NOMBRE

x25, PF_X25 - interfaz del protocolo ITU-T X.25 / ISO-8208.

SINOPSIS

```
#include <sys/socket.h>
#include <linux/x25.h>
```

```
x25_socket = socket(PF_X25, SOCK_SEQPACKET, 0);
```

DESCRIPCIÓN

Los **conectores (sockets) X25** proporcionan una **interfaz al protocolo de la capa de paquetes X.25**.

Éstos **permiten a las aplicaciones comunicarse a través de una red pública de datos X.25 según establece la recomendación X.25 de la International Telecommunication Union (X.25 modo DTE-DCE).**

Los conectores X25 **también se pueden usar para comunicarse sin una red X.25 intermedia (X.25 modo DTE-DTE)** como **se describe en ISO-8208.**

Se conservan los límites de los mensajes (una operación **read(2)** de un conector recuperará la misma cantidad de datos que la salida producida con la correspondiente operación **write(2)** en el conector asociado del otro extremo). Cuando es necesario, el núcleo se preocupa de segmentar y reensamblar los mensajes largos mediante el bit M de X.25. No existe un límite superior fijo para el tamaño de mensaje. Sin embargo, el reensamblaje de un mensaje largo puede fallar si se produce una escasez temporal de recursos del

sistema o cuando se ponen de manifiesto otras limitaciones (como la memoria de los conectores o los límites de los tamaños de los buffers). En tal caso, se reiniciará la conexión X.25.

DIRECCIONES DE LOS CONECTORES

La familia de direcciones de los conectores **AF_X25** usa la estructura **struct sockaddr_x25** para representar las direcciones de red tal y como se definen en la recomendación **X.121 del ITU-T**.

```
struct sockaddr_x25 {
    sa_family_t   sx25_family;    /* debe ser AF_X25 */
    x25_address sx25_addr;     /* Dirección X.121 */
};
```

sx25_addr contiene un array de caracteres, **x25_addr[]**, que se interpreta como una cadena terminada en un carácter nulo.

sx25_addr.x25_addr[] consiste de hasta 15 caracteres ASCII (sin contar el 0 del final) que forman la dirección X.121. Sólo se permiten los caracteres del '0' al '9'.

OPCIONES DE LOS CONECTORES

Las siguientes opciones específicas de los conectores X.25 se pueden configurar usando **setsockopt(2)** y se pueden leer con **getsockopt(2)**, asignándole al parámetro de nivel el valor **SOL_X25**.

X25_QBITINCL

Controla si el usuario puede acceder al bit Q de X.25 (Qualified Data Bit, bit de datos acreditados). Se espera un argumento entero.

Si es 0 (valor por defecto), nunca se activa el bit Q para los paquetes de salida y se ignora el bit Q de los paquetes de entrada.

Si es 1, se añade un primer byte adicional a cada mensaje leído de o escrito en el conector.

Para los datos leídos del conector, un primer byte 0 indica que no estaba activo el bit Q de los paquetes de datos de entrada correspondientes. Un primer byte con valor 1 indica que estaba activo el bit Q de los paquetes de datos de entrada correspondientes. Si el primer byte de los datos escritos en el conector es 1, se activa el bit Q de los paquetes de datos de salida correspondientes. Si el primer byte es 0, no se activará el bit Q.

FALLOS

Bastantes, ya que la implementación X.25 PLP es **CONFIG_EXPERIMENTAL**.

Esta página de manual está incompleta.

Todavía no existe un fichero cabecera específico para el

programador de aplicaciones. Necesita incluir el fichero cabecera <linux/x25.h> del núcleo. CONFIG_EXPERIMENTAL también implica que las versiones futuras de la interfaz podrían no ser compatibles a nivel binario.

Los eventos N-Reset de X.25 todavía no se propagan al proceso de usuario. Por eso, si se produce un reinicio, se podrían perder datos sin darse cuenta.

**VÉASE TAMBIÉN
socket(7), socket(2).**

**Jonathan Simon Naylor: "Reanálisis y reimplementación de X.25."
La URL es: <ftp://ftp.pspt.fi/pub/ham/linux/ax25/x25doc.tgz>**

VERSIONES

La familia de protocolo **PF_X25** es una nueva característica de la versión 2.2 de Linux.

Página man de Linux 1 diciembre 1998 X25(4)

Ahora leeremos la página de manual: **man 7 ddp**

DDP(7) Manual del Programador de Linux DDP(7)

NOMBRE

ddp - Implementación de Linux del protocolo AppleTalk

SINOPSIS

```
#include <sys/socket.h>
#include <netatalk/at.h>
```

```
ddp_socket = socket(PF_APPLETALK, SOCK_DGRAM, 0);
raw_socket = socket(PF_APPLETALK, SOCK_RAW, protocol);
```

DESCRIPCIÓN

Linux implementa el **protocolo Appletalk** descrito en **Inside Appletalk**. **Únicamente la capa DDP y el AARP se encuentran dentro del núcleo.** Están **diseñados para ser usados mediante las librerías netatalk** del protocolo. Esta **página documenta** la interfaz para aquellos que desean o necesitan **usar la capa DDP directamente**.

La **comunicación entre Appletalk y el programa de usuario funciona usando una interfaz de conectores compatible con BSD**. Para más información sobre conectores, vea **socket(7)**.

Un conector Appletalk se crea llamando a la función **socket(2)** y pasando **PF_APPLETALK** como familia del conector. Los tipos de conectores válidos son **SOCK_DGRAM** para abrir un conector ddp o **SOCK_RAW** para abrir un conector raw (directo).

protocol es el protocolo Appletalk a ser recibido o enviado.

Para **SOCK_RAW** debe especificar **ATPROTO_DDP**.

Los conectores directos sólo pueden ser abiertos por un proceso cuyo identificador de usuario efectivo sea 0 o por un proceso que posea la capacidad **CAP_NET_RAW**.

FORMATO DE LAS DIRECCIONES

La dirección de un conector Appletalk se define como la combinación de un número de red, un número de nodo y un número de puerto.

```
struct at_addr {  
    u_short      s_net;  
    u_char       s_node;  
};  
  
struct sockaddr_atalk {  
    sa_family_t   sat_family;      /* familia de direcciones */  
    u_char        sat_port;        /* port */  
    struct at_addr sat_addr;       /* red/nodo */  
};
```

A **sat_family** siempre se le asigna el valor **AF_APPLETALK**.

sat_port contiene el puerto. Los números de puerto por debajo de 129 se conocen como **puertos reservados**.

Sólo los procesos con identificador de usuario efectivo 0 o con la capacidad **CAP_NET_BIND_SERVICE** pueden enlazar estos conectores mediante **bind(2)**.

sat_addr es la dirección del anfitrión (host). El miembro **net** de **struct at_addr** contiene la red del anfitrión expresada en el formato "orden de red" de los bytes. El valor **AT_ANYNET** es un comodín y también implica "esta red". El miembro **node** de **struct at_addr** contiene el número de nodo del anfitrión. El valor **AT_ANYNODE** es un comodín y también implica "este nodo". El valor de **ATADDR_BCAST** es la dirección de enlace de difusión local.

OPCIONES DE LOS CONECTORES

No se soportan opciones de conector específicas del protocolo.

SYSCTLs

IP soporta una interfaz **sysctl** para configurar algunos parámetros AppleTalk globales. Se puede acceder a las sysctls leyendo o escribiendo los ficheros del directorio **/proc/sys/net/atalk** o con la interfaz **sysctl(2)**.

aarp-expiry-time

El intervalo de tiempo (en segundos) antes de que una entrada de la cache AARP expire.

aarp-resolve-time

El intervalo de tiempo (en segundos) antes de que se resuelva una entrada de la cache AARP.

aarp-retransmit-limit

El número de retransmisiones de una entrada AARP antes de que el nodo sea declarado muerto.

aarp-tick-time

La frecuencia del cronómetro (en segundos) para el cronómetro que controla el AARP.

Los valores por defecto coinciden con la especificación y nunca debe ser necesario el cambiarlos.

IOCTLs

Todas las ioctl's descritas en socket(7) se aplican también a ddp.

NOTAS

Tenga cuidado con la opción SO_BROADCAST (no es privilegiada en Linux). Es fácil sobrecargar la red sin darse cuenta enviando a la direcciones de difusión.

VERSIONES

Appletalk está soportado a partir de la versión 2.0 de Linux. La interfaz sysctl es nueva en la versión 2.2 de Linux.

ERRORES

ENOTCONN

La operación sólo está definida en un conector conectado pero el conector no está conectado.

EINVAL Se ha pasado un argumento inválido.

EMSGSIZE

El datagrama es mayor que la MTU de DDP.

EACCES El usuario ha intentado ejecutar una operación sin los permisos necesarios. Estos incluyen el enviar a una dirección de difusión sin haber activado la opción de difusión e intentar el enlace a un puerto reservado sin un identificador de usuario efectivo 0 y sin CAP_NET_BIND_SERVICE.

EADDRINUSE

Se ha intentado el enlace a una dirección ya en uso.

ENOMEM y ENOBUFS

No hay suficiente memoria disponible.

ENOPROTOOPT y EOPNOTSUPP

Se han pasado opciones de conector inválidas.

EPERM El usuario no tiene permiso para establecer una prioridad más alta, hacer un cambio a la configuración o enviar señales al proceso o grupo solicitado.

EADDRNOTAVAIL

Se ha solicitado una interfaz inexistente o la dirección fuente solicitada no es local.

EAGAIN La operación se bloquearía sobre un conector bloqueante.

ESOCKTNOSUPPORT

El conector está sin configurar o se ha solicitado un tipo de conector desconocido.

EISCONN

Se ha llamado a connect(2) sobre un conector ya conectado.

EALREADY

Ya se está realizando una operación de conexión sobre un conector no bloqueante.

ECONNABORTED

Se ha cerrado la conexión durante un accept(2).

EPIPE La conexión ha sido cerrada o cancelada por el otro extremo.

ENOENT Se ha llamado a SIOCGSTAMP sobre un conector en donde no han llegado paquetes.

EHOSTUNREACH

No existe una entrada en la tabla de enrutamiento que coincida con la dirección de destino.

ENODEV El dispositivo de red no está disponible o es incapaz de enviar IP.

ENOPKG No se ha configurado un subsistema del núcleo.

COMPATIBILIDAD

La interfaz básica de conectores AppleTalk es compatible con netatalk en

los sistemas derivados de BSD. Muchos sistemas BSD fallan al comprobar **SO_BROADCAST** cuando se envían tramas de difusión. Esto puede conducir a problemas de compatibilidad.

El modo de **conector directo es único de Linux y existe para soportar más fácilmente el paquete alternativo CAP y las herramientas de monitorización de AppleTalk.**

FALLOS

Hay **demasiados valores de error inconsistentes**.

Las **iocntl usadas para configurar las tablas de enrutamiento, dispositivos, tablas AARP y otros dispositivos no se han descrito todavía**.

VÉASE TAMBIÉN

sendmsg(2), recvmsg(2), socket(7)

Página man de Linux

1 mayo 1999

DDP(7)

Ahora nos disponemos a leer: **man 7 packet**

PACKET(7)

Manual del Programador de Linux

PACKET(7)

NOMBRE

packet, PF_PACKET - Interfaz de paquetes a nivel de dispositivo.

SINOPSIS

```
#include <sys/socket.h>
#include <features.h> /* para el número de versión de glibc */
#if __GLIBC__ >= 2 && __GLIBC_MINOR__ >= 1
    #include <netpacket/packet.h>
    #include <net/ethernet.h> /* los protocolos de nivel 2 */
#else
    #include <asm/types.h>
    #include <linux/if_packet.h>
    #include <linux/if_ether.h> /* los protocolos de nivel 2 */
#endif
```

```
packet_socket = socket(PF_PACKET, int socket_type, int protocol);
```

DESCRIPCIÓN

Los **conectores de paquetes (packet sockets)** se usan para **recibir o enviar paquetes directos (raw) en el nivel del manejador de dispositivo (Nivel 2 de OSI)**. Permiten al usuario **implementar módulos de protocolo en el espacio de usuario por encima de la capa física**.

socket_type es o bien **SOCK_RAW** para paquetes directos incluyendo la cabecera del nivel de enlace o bien **SOCK_DGRAM** para paquetes preparados con la cabecera del nivel de enlace eliminada. La información de la cabecera del nivel de enlace está disponible en un formato común en una estructura **sockaddr_ll**.

protocol es el protocolo IEEE 802.3 con los bytes en orden de red. Vea el fichero cabecera <linux/if_ether.h> para una lista de los protocolos permitidos. Cuando se asigna a **protocol** el valor **htons(ETH_P_ALL)**, se reciben todos los protocolos. Todos los paquetes de entrada con el tipo de protocolo indicado se pasarán al conector de paquetes antes de que sean pasados a los protocolos implementados dentro del núcleo.

Sólo los procesos con uid efectivo 0 o la capacidad CAP_NET_RAW pueden abrir conectores de paquetes.

Los paquetes **SOCK_RAW** se pasan a y desde el manejador de dispositivo sin ningún cambio en los datos del paquete. Cuando se recibe un paquete, la dirección todavía se analiza y se pasa en una estructura de dirección **sockaddr_ll** estándar. Cuando se transmite un paquete, el buffer proporcionado por el usuario debería contener la cabecera de la capa física. A continuación, ese paquete se encola sin modificar en la tarjeta de red de la interfaz definida por la dirección de destino. Algunos manejadores ('drivers') de dispositivo siempre añaden otras cabeceras. **SOCK_RAW** es similar pero no compatible con el obsoleto **SOCK_PACKET** de la versión 2.0 de Linux.

SOCK_DGRAM opera en un nivel ligeramente superior. Se elimina la cabecera física antes de que el paquete se pase al usuario. Los paquetes enviados a través de un conector de paquetes **SOCK_DGRAM** obtienen una cabecera adecuada de la capa física según la información de la dirección de destino **sockaddr_ll**, antes de ser encolados.

Por defecto, todos los paquetes del tipo de protocolo especificado se pasan a un conector de paquetes. Para obtener sólo los paquetes de una interfaz específica, use **bind(2)** especificando una dirección en una estructura **struct sockaddr_ll** para enlazar el conector de paquetes a una interfaz.

Sólo se usan para propósitos de enlace los campos de dirección **sll_protocol** y **sll_ifindex**.

La operación **connect(2)** no está soportada en conectores de paquetes.

TIPOS DE DIRECCIONES

sockaddr_ll es una dirección de la capa física independiente del dispositivo.

```
struct sockaddr_ll
{
    unsigned short sll_family; /* Siempre es AF_PACKET */
    unsigned short sll_protocol; /* Protocolo de la capa física */
    int sll_ifindex; /* Número de la interfaz */
    unsigned short sll_hatype; /* Tipo de cabecera */
    unsigned char sll_pkttype; /* Tipo de paquete */
    unsigned char sll_halen; /* Longitud de la dirección */
    unsigned char sll_addr[8]; /* Dirección de la capa física */
};
```

sll_protocol es el **tipo del protocolo ethernet** estándar dado **en orden de red definido en** el fichero cabecera **linux/if_ether.h**.

sll_ifindex es el **índice de la interfaz** (vea **netdevice(2)**). Un **0 concuerda con cualquier interfaz (sólo legal para enlazar)**.

sll_hatype es un **tipo ARP** de los **definidos en** el fichero cabecera **linux/if_arp.h**.

sll_pkttype contiene el **tipo del paquete**. Los **tipos válidos son**

PACKET_HOST para un **paquete aplicado al anfitrión (host) local**,

PACKET_BROADCAST para un **paquete de difusión de la capa física**,

PACKET_MULTICAST para un **paquete enviado a una dirección multidestino de la capa física**,

PACKET_OTHERHOST para un **paquete destinado a otros anfitriones** que ha sido **capturado por el manejador del dispositivo en modo promiscuo** y

PACKET_OUTGOING para un **paquete originado desde el anfitrión local** que es **devuelto de regreso a un conector de paquetes**.

Estos **tipos** sólo tienen sentido **para recibir**.

sll_addr y **sll_halen** contienen la **dirección de la capa física** (por ejemplo, **IEEE 802.3**) y su **longitud**. La interpretación exacta depende del dispositivo.

OPCIONES DE LOS CONECTORES

Los **conectores de paquetes sólo** se pueden **usar para configurar el envío multidestino de la capa física y el modo promiscuo**. Esto funciona llamando a `setsockopt(2)` con `SOL_PACKET`, para un **conector de paquetes**, y una de las opciones `PACKET_ADD_MEMBERSHIP` para añadir un enlace o `PACKET_DROP_MEMBERSHIP` para eliminarlo. Ambas esperan una estructura `packet_mreq` como argumento:

```
struct packet_mreq
{
    int             mr_ifindex;      /* índice de la interfaz */
    unsigned short mr_type;        /* acción */
    unsigned short mr_alen;        /* longitud de la dirección */

    unsigned char  mr_address[8];   /* dirección de la capa física */
};
```

mr_ifindex contiene el **índice de la interfaz cuyo estado debe cambiarse**.

El parámetro **mr_type** indica la **acción a realizar**.

PACKET_MR_PROMISC **habilita la recepción de todos los paquetes sobre un medio compartido** (conocido normalmente como ``**modo promiscuo**''),

PACKET_MR_MULTICAST **enlaza el conector al grupo multidestino de la capa física** indicado en **mr_address** y **mr_alen**, y

PACKET_MR_ALLMULTI **configura el conector para recibir todos los paquetes multidestino que lleguen a la interfaz.**

Además, se pueden usar las **ioctls** tradicionales, **SIOCSIFFLAGS**, **SIOCADDMULTI** y **SIOCDELMULTI**, para el mismo propósito.

IOCTLs

SIOCGSTAMP se puede usar **para recibir la marca de tiempo** del último paquete recibido. El **argumento es** una estructura **struct timeval**.

Además, **todas las ioctl**s estándares **definidas en netdevice(7)** y **socket(7)** **son válidas** en los conectores de paquetes.

MANEJO DE ERRORES

Los **conectores de paquetes no manejan otros errores que los ocurridos al pasar el paquete al manejador del dispositivo. No poseen el concepto de error pendiente.**

COMPATIBILIDAD

En la versión 2.0 de Linux, la única forma de obtener un conector de paquetes era llamando a **socket(PF_INET, SOCK_PACKET, protocol)**. Esto **todavía está soportado pero se desaprueba fuertemente**.

La principal **diferencia** entre los dos métodos **es que SOCK_PACKET**, para especificar una interfaz, **usa la antigua struct sockaddr_pkt que no proporciona independencia de la capa física**.

```
struct sockaddr_pkt
{
    unsigned short      spkt_family;
    unsigned char       spkt_device[14];
    unsigned short      spkt_protocol;
};
```

spkt_family contiene el **tipo del dispositivo**,

spkt_protocol es el **tipo del protocolo IEEE 802.3** de los definidos en **<sys/if_ether.h>** y

spkt_device es el nombre del dispositivo dado como una cadena terminada en un nulo, por ejemplo, **eth0**.

Esta **estructura** está **obsoleta y no debería usarse** en código nuevo.

NOTAS

Se sugiere que los **programas transportables usen PF_PACKET a través de pcap(3)**, aunque **esto sólo cubre un subconjunto de las características de PF_PACKET**.

Los **conectores** de paquetes **SOCK_DGRAM no intentan crear o analizar la cabecera LLC IEEE 802.2 para una trama IEEE 802.3. Cuando se especifica ETH_P_802_3 como protocolo para enviar, el núcleo crea la trama 802.3 y rellena el campo de longitud**. El **usuario** tiene que **proporcionar la cabecera LLC para obtener un paquete totalmente conforme**. Los **paquetes 802.3 de entrada no son multiplexados en los campos DSAP/SSAP** del protocolo. En su lugar, **se entregan al usuario como protocolo ETH_P_802_2 con la cabecera LLC añadida**. Por tanto, **es imposible enlazar con ETH_P_802_3. Enlace** en su lugar **con ETH_P_802_2 y haga usted mismo la multiplexación del protocolo**. Para **enviar por omisión se utiliza la encapsulación** estándar **Ethernet DIX con el dato del protocolo lleno**.

Los **conectores de paquetes no están sujetos a las cadenas de entrada ni de salida del cortafuegos.**

ERRORES

ENETDOWN

La **interfaz no está activa.**

ENOTCONN

No se ha pasado una **dirección de interfaz.**

ENODEV Nombre de dispositivo o índice de interfaz, especificados en la dirección de interfaz, **desconocidos**.

EMSGSIZE

El **paquete es más grande que la MTU de la interfaz.**

ENOBUFS

No **hay suficiente memoria** para colocar el paquete.

EFAULT El usuario ha pasado una **dirección de memoria inválida.**

EINVAL Argumento inválido.

ENXIO La dirección de interfaz contiene un índice de interfaz ilegal.

EPERM El **usuario no tiene privilegios** suficientes para llevar a cabo esta operación.

EADDRNOTAVAIL

Se ha pasado una **dirección desconocida de grupo multidestino.**

ENOENT No se ha recibido ningún paquete.

Además, el manejador de bajo nivel puede **generar otros errores.**

VERSIONES

PF_PACKET es una nueva característica de la versión 2.2 de Linux. Las primeras versiones de Linux sólo soportaban **SOCK_PACKET**.

FALLOS

glibc 2.1 no posee una **macro "define"** para **SOL_PACKET**. La **solución sugerida** es usar

```
#ifndef SOL_PACKET  
#define SOL_PACKET 263  
#endif
```

Esto **se soluciona en versiones posteriores de glibc**. Este problema tampoco se produce en sistemas **libc5**.

El tratamiento del **IEEE 802.2/803.3 LLC** se podría considerar un fallo.

No se han documentado los filtros de los conectores.

CREDITOS

Esta página de manual fue escrita por **Andi Kleen** con la **ayuda de Matthew Wilcox**.

Alexey Kuznetsov implementó la característica PF_PACKET de la versión 2.2 de Linux basándose en el código de Alan Cox y otros.

VÉASE TAMBIÉN

ip(7), socket(7), socket(2), raw(7), pcap(3).

RFC 894 para la Encapsulación Ethernet del Estandar IP.

RFC 1700 para la Encapsulación IP del IEEE 802.3.

El fichero cabecera **linux/if_ether.h** para los **protocolos de la capa física**.

Página man de Linux

29 abril 1999

PACKET(7)

Seguimos con la lectura de la página de manual en línea de comandos:

man 7 netdevice

NETDEVICE(7)

Manual del Programador de Linux

NETDEVICE(7)

NOMBRE

netdevice - Acceso de bajo nivel a los dispositivos de red de Linux.

SINOPSIS

```
#include <sys/ioctl.h>
#include <net/if.h>
```

DESCRIPCIÓN

Esta página de manual describe la **interfaz de conectores** que se usa para **configurar los dispositivos de red**.

Linux soporta algunas **ioctls** estándares **para configurar los dispositivos de red**. Se **pueden usar** sobre **cualquier descriptor de fichero de un conector sin tener en cuenta la familia o el tipo**. Se pasa una **estructura ifreq**:

```

struct ifreq
{
    char      ifr_name[IFNAMSIZ]; /* Nombre de la interfaz */
    union {
        struct sockaddr     ifr_addr;
        struct sockaddr     ifr_dstaddr;
        struct sockaddr     ifr_broadaddr;
        struct sockaddr     ifr_netmask;
        struct sockaddr     ifr_hwaddr;
        short               ifr_flags;
        int                ifr_ifindex;
        int                ifr_metric;
        int                ifr_mtu;
        struct ifmap         ifr_map;
        char               ifr_slave[IFNAMSIZ];
        char               ifr_newname[IFNAMSIZ];
        char *              ifr_data;
    };
}

struct ifconf
{
    int      ifc_len;           /* tamaño del buffer */
    union {
        char *            ifc_buf;   /* dirección del buffer */
        struct ifreq *    ifc_req;   /* array de estructuras */
    };
}

```

Normalmente, el usuario **especifica a qué dispositivo** va a afectar **asignando a ifr_name el nombre de la interfaz**. Todos los otros miembros de la estructura pueden compartir memoria.

IOCTLs

Si se marca una **ioctl** como **privilegiada** entonces su uso **requiere un identificador de usuario efectivo 0 o la capacidad CAP_NET_ADMIN**. Si éste **no es el caso se devuelve EPERM**.

SIOCGIFNAME

Dado un **ifr_ifindex**, **devuelve el nombre de la interfaz en ifr_name**. Ésta es la única ioctl que devuelve su resultado en **ifr_name**.

SIOCGIFINDEX

Devuelve el índice de interfaz de la interfaz en ifr_ifindex.

SIOCGIFFLAGS, SIOCSIFFLAGS

Obtiene o establece la palabra de banderas activas del dispositivo. **ifr_flags** contiene una **máscara de bits** de los **siguientes valores**:

<i>Significado de las banderas</i>	
IFF_UP	La interfaz está funcionando.
IFF_BROADCAST	Dirección de difusión válida asignada.
IFF_DEBUG	Bandera de depuración interna.
IFF_LOOPBACK	Ésta es una interfaz loopback.
IFF_POINTOPOINT	La interfaz es un enlace punto a punto.
IFF_RUNNING	Recursos necesarios reservados.
IFF_NOARP	Sin protocolo ARP, la dirección de destino de Nivel 2 no está configurada.
IFF_PROMISC	La interfaz se encuentra en modo promiscuo.
IFF_NOTRAILERS	Evitar el uso de terminadores.
IFF_ALLMULTI	Recibir todos los paquetes multidestino.
IFF_MASTER	Interfaz maestra de un grupo de balanceo de carga.
IFF_SLAVE	Interfaz esclava de un grupo de balanceo de carga.
IFF_MULTICAST	La interfaz soporta multidestino.
IFF_PORTSEL	La interfaz es capaz de seleccionar el tipo de medio mediante ifmap.
IFF_AUTOMEDIA	Autoselección de medios activa.
IFF_DYNAMIC	Las direcciones se pierden cuando la interfaz se desactiva.

La configuración de la palabra de banderas activas es una operación privilegiada pero cualquier proceso puede leerla.

SIOCGIFMETRIC, SIOCSIFMETRIC

Obtiene o establece la métrica del dispositivo usando `ifr_metric`. Todavía no implementado.

Asigna un 0 a `ifr_metric` cuando se intenta leer y devuelve EOPNOTSUPP cuando se intenta asignarle un valor.

SIOCGIFMTU, SIOCSIFMTU

Obtiene o establece la MTU (unidad de transferencia máxima) del dispositivo usando `ifr_mtu`.

La configuración de la MTU es una **operación privilegiada**. **Configurar la MTU con valores demasiado pequeños puede provocar un fallo del núcleo.**

SIOCGIFHWADDR, SIOCSIFHWADDR

Obtiene o establece la dirección hardware del dispositivo usando `ifr_hwaddr`. La configuración de la dirección hardware es una **operación privilegiada**.

SIOCSIFHWBROADCAST

Establece la dirección de difusión hardware del dispositivo a partir de `ifr_hwaddr`. Es una **operación privilegiada**.

SIOCGIFMAP, SIOCSIFMAP

Obtiene o establece los **parámetros hardware de la interfaz** usando **ifr_map**. La configuración de los parámetros es una **operación privilegiada**.

```
struct ifmap
{
    unsigned long      mem_start;
    unsigned long      mem_end;
    unsigned short     base_addr;
    unsigned char      irq;
    unsigned char      dma;
    unsigned char      port;
};
```

La **interpretación de la estructura ifmap depende del manejador del dispositivo y de la arquitectura**.

SIOCADDMULTI, SIOCDELMULTI

Añade una **dirección** a o **borra** una **dirección de los filtros multidestino de la capa de enlace de la interfaz** usando **ifr_hwaddr**. Estas **operaciones** son **privilegiadas**. Si quiere una alternativa, vea también **packet(7)**.

SIOCGIFTXQLEN, SIOCSIFTXQLEN

Obtiene o establece la **longitud de la cola de transmisión** de un dispositivo usando **ifr_qlen**.

La configuración de la longitud de la cola de transmisión es una **operación privilegiada**.

SIOCSIFNAME

Cambia el nombre de la interfaz indicada en **ifr_ifindex** a **ifr_newname**. Es una **operación privilegiada**.

SIOCGIFCONF

Devuelve una **lista de direcciones de interfaces** (capa de transporte). Actualmente, esto sólo significa **direcciones de la familia AF_INET** por compatibilidad. El usuario pasa a la **ioctl** una **estructura ifconf como argumento**. Contiene un **puntero a un array de estructuras ifreq** en **ifc_req** y **sus longitudes en bytes** en **ifc_len**. El **núcleo rellena los ifreqs con todas las direcciones de las interfaces de Nivel 3** actuales que están funcionando:

ifr_name contiene el **nombre de la interfaz (eth0:1 etc.)**,

ifr_addr la **dirección**.

El núcleo regresa con la longitud real en ifc_len. Si es igual a la longitud original, el usuario debe asumir que se desbordó y debe reintentarlo con un buffer mayor. Cuando no se produce ningún error, la ioctl devuelve 0. En otro caso, -1. El desbordamiento no es un error.

La mayoría de los protocolos soportan sus propias ioctls para configurar las opciones de la interfaz específicas del protocolo. Vea las páginas de manual de los protocolos para una descripción más amplia. Para la configuración de direcciones IP, **vea ip(7)**.

Además, **algunos dispositivos soportan ioctls privadas**. Éstas no se describen aquí.

NOTA

Si lo vemos de forma estricta, **SIOCGIFCONF** es **específica de IP** y **pertece a ip(7)**.

Los **nombres de interfaces que no tiene dirección o que no tienen la opción IFF_RUNNING** activa, se pueden **encontrar** a través de **/proc/net/dev**.

VÉASE TAMBIÉN

ip(7), proc(7)

Página man de Linux

2 mayo 1999

NETDEVICE(7)

Nos dispondremos a ver la página de manual en línea de comandos:

man 7 rtnetlink

RTNETLINK(7)

Manual del Programador de Linux

RTNETLINK(7)

NOMBRE

rtnetlink, NETLINK_ROUTE - Conector de enrutamiento IPv4 de Linux.

SINOPSIS

```
#include <asm/types.h>
#include <linux/netlink.h>
#include <linux/rtnetlink.h>
#include <sys/socket.h>

rtnetlink_socket = socket( PF_NETLINK,
                           int socket_type,
                           NETLINK_ROUTE
                         );
```

DESCRIPCIÓN

Rtnetlink permite **leer** y **alterar** las **tablas de enrutamiento del núcleo**. Se usa dentro del núcleo para **comunicar entre sí** varios **subsistemas** (aunque este uso no se documenta aquí) y para la **comunicación con programas en el espacio de usuario**. Las **rutas de red**, las **direcciones IP**, los **parámetros de enlace**, las **configuraciones de vecinos**, las **disciplinas de encolamiento**, las **clases de intercambio** y los **clasificadores de paquetes** pueden **controlarse a través de conectores NETLINK_ROUTE**. Todo esto se basa en **mensajes netlink**. Vea **netlink(7)** para más información.

ATRIBUTOS DE ENRUTAMIENTO

Algunos mensajes **rtnetlink** poseen atributos opcionales después de la cabecera inicial:

```
struct rtattr
{
    unsigned short rta_len;          /* Longitud de la opción */
    unsigned short rta_type;         /* Tipo de opción */
    /* Los datos vienen a continuación */
};
```

Estos **atributos sólo** deberían **manipularse usando las macros RTA_*** o la librería **libnetlink**. Vea **rtnetlink(3)**.

MENSAJES

Rtnetlink está formado por los siguientes tipos de mensajes (además de los mensajes netlink estándares):

RTM_NEWRINK, RTM_DELLINK, RTM_GETLINK

Crea, borra u obtiene información de una **interfaz de red** específica. Estos mensajes contienen una estructura **ifinfomsg** seguida por una serie de **estructuras rtattr**.

```
struct ifinfomsg
{
    unsigned char      ifi_family; /* AF_UNSPEC */
    unsigned char      ifi_pad;    /* Sin usar */
    unsigned short     ifi_type;   /* Tipo del dispositivo */
    int                ifi_index;  /* Índice de la interfaz */
    unsigned int       ifi_flags;  /* Opciones del dispositivo */
    unsigned int       ifi_change; /* Máscara de cambios */
};
```

ifi_flags contiene las **opciones del dispositivo**. Vea **netdevice(7)**.

ifi_index es el **índice de la interfaz**.

ifi_change se reserva para uso futuro y siempre debe valer **0xFFFFFFFF**.

<i>rta_type</i>	<i>Atributos de enrutamiento</i>	<i>Tipo de valor</i>	<i>Descripción</i>
IFLA_UNSPEC	-		sin especificar.
IFLA_ADDRESS	dirección hardware		dirección del Nivel 2 de la interfaz
IFLA_BROADCAST	dirección hardware		dirección de difusión del Nivel 2
IFLA_IFNAME	cadena terminada en cero		nombre del dispositivo
IFLA_MTU	entero sin signo		MTU del dispositivo
IFLA_LINK	entero		tipo de enlace
IFLA_QDISC	cadena terminada en cero		disciplina de encolamiento
IFLA_STATS	estructura net_device_stats		estadísticas de la interfaz

RTM_NEWWADDR, RTM_DELADDR, RTM_GETADDR

añade, elimina o recibe información de una dirección IP asociada con una interfaz. En Linux 2.2 una interfaz puede acarrear varias direcciones IP. Esto reemplaza el concepto de dispositivo alias de la versión 2.0. En Linux 2.2 estos mensajes soportan direcciones IPv4 e IPv6.

Contienen una estructura ifaddrmsg seguida opcionalmente por atributos de enrutamiento rtaddr.

struct ifaddrmsg

```
{
    unsigned char ifa_family;      /* Tipo de dirección */
    unsigned char ifa_prefixlen;   /* Longitud del prefijo de
                                    la dirección */
    unsigned char ifa_flags;       /* Opciones de la dirección */
    unsigned char ifa_scope;       /* Ámbito de la dirección */
    int           ifa_index;       /* Índice de la interfaz */
};
```

ifa_family es el **tipo de la familia de direcciones** (actualmente **AF_INET** o **AF_INET6**),

ifa_prefixlen es la **longitud de la máscara de la dirección si ésta está definida para la familia** (como ocurre con IPv4),

ifa_scope es el **ámbito de la dirección**,

ifa_index es el **índice de la interfaz** con la que la dirección está asociada.

ifa_flags es una **palabra de opciones** que contiene

IFA_F_SECONDARY para una **dirección secundaria (antiguo interfaz alias)**,

IFA_F_PERMANENT para una **dirección permanente** configurada por el usuario, y otras opciones sin documentar.

<i>Atributos</i>	Tipo de valor	Descripción
rta_type	-	sin especificar.
IFA_UNSPEC	dirección de protocolo	dirección de la interfaz
IFA_ADDRESS	dirección de protocolo	dirección local
IFA_LOCAL	cadena terminada en cero	nombre de la interfaz
IFA_LABEL	dirección de protocolo	dirección de difusión
IFA_BROADCAST	dirección de protocolo	dirección de "cualquier destino"
IFA_ANYCAST	dirección de protocolo	información de la dirección
IFA_CACHEINFO	estructura ifa_cacheinfo	

RTM_NEWRROUTE, RTM_DELROUTE, RTM_GETROUTE

Crea, borra o recibe información de una ruta de red. Estos mensajes **contienen** una estructura **rtmsg** con una secuencia opcional a continuación de estructuras **rtattr**.

```
struct rtmsg
{
    unsigned char rtm_family; /* Familia de direcciones de la ruta */
    unsigned char rtm_dst_len; /* Longitud del destino */
    unsigned char rtm_src_len; /* Longitud del origen */
    unsigned char rtm_tos; /* Filtro TOS */

    unsigned char rtm_table; /* Identificador de la tabla de
                                enrutamiento */
    unsigned char rtm_protocol; /* Protocolo de enrutamiento. Ver
                                más abajo */
    unsigned char rtm_scope; /* Ver más abajo */
    unsigned char rtm_type; /* Ver más abajo */

    unsigned int rtm_flags;
};
```

<i>rtm_type</i>	<i>Tipo de ruta</i>
RTN_UNSPEC	ruta desconocida
RTN_UNICAST	una pasarela (gateway) o ruta directa
RTN_LOCAL	una ruta de la interfaz local
RTN_BROADCAST	una ruta de difusión local (enviada en una difusión)
RTN_ANYCAST	una ruta de difusión local (enviada en un unidestino)
RTN_MULTICAST	una ruta multidestino
RTN_BLACKHOLE	una ruta para la pérdida de paquetes
RTN_UNREACHABLE	un destino inalcanzable
RTN_PROHIBIT	una ruta de rechazo de paquetes
RTN_THROW	continuar la búsqueda de rutas en otra tabla
RTN_NAT	una regla de traducción de direcciones de red
RTN_XRESOLVE	remitir a una entidad de resolución externa (no implementado)

<i>rtm_protocol</i>	<i>Ruta original</i>
RTPROT_UNSPEC	desconocido
RTPROT_REDIRECT	por una redirección ICMP (no usado actualmente)
RTPROT_KERNEL	por el núcleo
RTPROT_BOOT	durante el arranque
RTPROT_STATIC	por el administrador

Los valores mayores que **RTPROT_STATIC** no son interpretados por el núcleo, sólo son para información de usuario. Se pueden usar para identificar la fuente de una información de enrutamiento o para distinguir entre varios demonios de enrutamiento. Vea **<linux/rtnetlink.h>** para los identificadores de los demonios de enrutamiento que ya están asignados.

rtm_scope es la distancia al destino:

RT_SCOPE_UNIVERSE	ruta global
RT_SCOPE_SITE	ruta interior en el sistema autónomo local
RT_SCOPE_LINK	ruta en este enlace
RT_SCOPE_HOST	ruta en el anfitrión local
RT_SCOPE_NOWHERE	el destino no existe

Los valores entre **RT_SCOPE_UNIVERSE** y **RT_SCOPE_SITE** están disponibles para el usuario.

rtm_flags tiene los siguientes significados:

RTM_F_NOTIFY	si la ruta cambia, informar al usuario mediante rtnetlink
RTM_F_CLONED	la ruta es un duplicado de otra ruta
RTM_F_EQUALIZE	un equalizador multidestino (no implementado todavía)

rtm_table especifica la **tabla de enrutamiento**

RT_TABLE_UNSPEC	una tabla de enrutamiento sin especificar
RT_TABLE_DEFAULT	la tabla por defecto
RT_TABLE_MAIN	la tabla principal
RT_TABLE_LOCAL	la tabla local

El usuario puede asignar valores arbitrarios entre **RT_TABLE_UNSPEC** y **RT_TABLE_DEFAULT**.

rtt_type	<i>Atributos</i>		Descripción
	Tipo de valor		
RTA_UNSPEC	-		ignorado
RTA_DST	dirección de protocolo		dirección de destino de la ruta
RTA_SRC	dirección de protocolo		dirección de origen de la ruta
RTA_IIF	entero		índice de la interfaz de entrada
RTA_OIF	entero		índice de la interfaz de salida
RTA_GATEWAY	dirección de protocolo		la pasarela (gateway) de la ruta
RTA_PRIORITY	entero		prioridad de la ruta
RTA_PREFSRC			
RTA_METRICS	entero		métrica de la ruta
RTA_MULTIPATH			
RTA_PROTOINFO			
RTA_FLOW			
RTA_CACHEINFO			

¡Esta tabla está incompleta!

RTM_NEIGH, RTM_DELNEIGH, RTM_GETNEIGH

añade, borra o recibe información de una entrada de la tabla de vecinos (por ejemplo, una entrada ARP). El **mensaje contiene una estructura ndmsg**.

```
struct ndmsg
{
    unsigned char      ndm_family;
    unsigned char      ndm_pad1;
    unsigned short     ndm_pad2;
    int                ndm_ifindex; /* Índice de la interfaz */
    __u16              ndm_state;  /* Estado */
    __u8               ndm_flags;   /* Opciones */
    __u8               ndm_type;
};
```

```
struct nda_cacheinfo
{
    __u32      ndm_confirmed;
    __u32      ndm_used;
    __u32      ndm_updated;
    __u32      ndm_refcnt;
};
```

ndm_state es una **máscara de bits** de los siguientes estados:

NUD_INCOMPLETE	una entrada de la cache que se está resolviendo actualmente
NUD_REACHABLE	una entrada de la cache que ya se ha confirmado como operativa
NUD_STALE	una entrada de la cache caduca
NUD_DELAY	una entrada que espera a un cronómetro
NUD_PROBE	una entrada de la cache que se está sondeando de nuevo actualmente
NUD_FAILED	una entrada de la cache inválida
NUD_NOARP	un dispositivo sin cache de destinos
NUD_PERMANENT	una entrada estática

Los **ndm_flags** válidos son:

NTF_PROXY	una entrada proxy arp
NTF_ROUTER	un enrutador IPv6

Hay que documentar mejor los miembros de la estructura.

La **estructura rtaddr** tiene los siguientes **significados** para el **campo rta_type**:

NDA_UNSPEC	tipo desconocido
NDA_DST	una dirección de destino de la capa de red de la cache de vecinos
NDA_LLADDR	una dirección de la capa de enlace de la cache de vecinos
NDA_CACHEINFO	estadísticas de la cache

Si el campo **rta_type** es **NDA_CACHEINFO**, a continuación viene una cabecera **struct nda_cacheinfo**.

RTM_NEWRULE, RTM_DELRULE, RTM_GETRULE

añade, borra o recupera una **regla de enrutamiento**. Lleva asociada una **struct rtmsg**.

RTM_NEWQDISC, RTM_DELQDISC, RTM_GETQDISC

añade, borra u obtiene una **disciplina de encolamiento**. El mensaje contiene una **struct tcmsg** que puede ir seguida por una serie

de **atributos**.

struct tcmsg	<i>Atributos</i>	<i>Tipo de valor</i>	<i>Descripción</i>
{			
unsigned char tcm_family;			
unsigned char tcm_pad1;			
unsigned short tcm_pad2;			
int tcm_ifindex; /* Índice de la interfaz */			
__u32 tcm_handle; /* Descriptor qdisc */			
__u32 tcm_parent; /* Qdisc del padre*/			
__u32	tcm_info;		
}			
rta_type			
TCA_UNSPEC	-		sin especificar
TCA_KIND		cadena terminada en cero	nombre de la disciplina de encolamiento
TCA_OPTIONS		secuencia de bytes	opciones específicas de Qdisc que vienen a continuación
TCA_STATS		estructura tc_stats	estadísticas qdisc
TCA_XSTATS		específico de qdisc	estadísticas específicas del módulo
TCA_RATE		estructura tc_estimator	límite de la tasa

Además, se permiten **otros atributos** diferentes **específicos del módulo qdisc**. Para más información, **vea los ficheros cabecera adecuados**.

RTM_NEWTCLASS, RTM_DELTCLASS, RTM_GETTCLASS

Añade, borra u obtiene una clase de intercambio. Estos mensajes contienen una **struct tcmsg** como la descrita anteriormente.

RTM_NEWTFILTER, RTM_DELTFILTER, RTM_GETTFILTER

Añade, borra o recibe información de una filtro de tráfico. Estos mensajes contienen una **struct tcmsg** como la descrita anteriormente.

VERSIONES

rtnetlink es una nueva característica de la versión 2.2 de Linux.

FALLOS

Esta **página de manual** es **deficiente** y está **incompleta**.

VÉASE TAMBIÉN

netlink(7), cmsg(3), ip(7), rtnetlink(3)

NETLINK(3)

Manual del Programador de Linux

NETLINK(3)

NOMBRE

netlink - macros netlink

SINOPSIS

```
#include <asm/types.h>
#include <linux/netlink.h>
int NLMSG_ALIGN(size_t len);
int NLMSG_LENGTH(size_t len);
int NLMSG_SPACE(size_t len);
void *NLMSG_DATA(struct nlmsghdr *nlh);
struct nlmsghdr *NLMSG_NEXT(struct nlmsghdr *nlh, int len);
int NLMSG_OK(struct nlmsghdr *nlh, int len);
int NLMSG_PAYLOAD(struct nlmsghdr *nlh, int len);
```

DESCRIPCIÓN

netlink.h define varias macros estándares para **acceder o crear un datagrama netlink**. En esencia son **similares a las macros definidas en cmsg(3)** para los datos auxiliares. Se **debería acceder al buffer pasado a y desde un conector netlink usando únicamente estas macros**.

NLMSG_ALIGN

Redondea la longitud de un mensaje netlink hasta alinearlo adecuadamente.

NLMSG_LENGTH

Toma como **argumento la longitud del contenido útil** y **devuelve la longitud alineada** para almacenarlo en el **campo nlmsg_len de nlmsghdr**.

NLMSG_SPACE

Devuelve el número de bytes que ocuparía un **mensaje netlink** con un **contenido útil de la longitud pasada**.

NLMSG_DATA

Devuelve un puntero al contenido útil asociado con el nlmsghdr pasado.

NLMSG_NEXT

Obtiene el **siguiente nlmsghdr** en un **mensaje multiparte**. El invocador debe comprobar si el **nlmsghdr** actual no tenía activa la opción **NLMSG_DONE** (esta función no devuelve NULL al final). El **parámetro longitud** es un **valor izquierdo** (lvalue) que contiene la **longitud restante del buffer del mensaje**. Esta macro lo **decrementa en la longitud de la cabecera del mensaje**.

NLMSG_OK

Devuelve verdadero si el mensaje netlink no está truncado y es correcto para ser analizado.

NLMSG_PAYLOAD

Devuelve la longitud del contenido útil asociado con **nlmsghdr**.

NOTAS

Normalmente es mejor usar netlink a través de libnetlink que mediante la interfaz de bajo nivel del núcleo.

VÉASE TAMBIÉN

netlink(7)

ftp://ftp.inr.ac.ru/ip-routing/iproute2* para libnetlink

Página man de Linux

14 mayo 1999

NETLINK(3)

Continuamos con la lectura de:

man 3 rtinetlink

RTNETLINK(3)

Manual del Programador de Linux

RTNETLINK(3)

NOMBRE

rtinetlink - Macros para manipular mensajes rtinetlink

SINOPSIS

```
#include <asm/types.h>
#include <linux/netlink.h>
#include <linux/rtinetlink.h>
#include <sys/socket.h>

rtinetlink_socket = socket(PF_NETLINK,
                           int socket_type,
                           NETLINK_ROUTE);
int RTA_OK(struct rtattr *rta, int rtabuflen);
void *RTA_DATA(struct rtattr *rta);
unsigned int RTA_PAYLOAD(struct rtattr *rta);
struct rtattr *RTA_NEXT( struct rtattr *rta,
                        unsigned int rtabuflen);
unsigned int RTA_LENGTH(unsigned int length);
unsigned int RTA_SPACE(unsigned int length);
```

DESCRIPCIÓN

Todos los mensajes **rtinetlink(7)** están formados por una cabecera de mensaje **netlink(7)** y atributos añadidos. Los atributos sólo deberían ser manipulados usando las macros suministradas aquí.

RTA_OK(rta, attrlen)	devuelve verdadero si rta apunta a un atributo de enrutamiento válido.
attrlen	es la longitud actual del buffer de atributos . Cuando es falso debe asumir que no hay más atributos en el mensaje, aunque attrlen no sea cero.
RTA_DATA(rta)	devuelve un puntero al principio de los datos de este atributo.
RTA_PAYLOAD(rta)	devuelve la longitud de los datos de este atributo.
RTA_NEXT(rta, attrlen)	obtiene el siguiente atributo después de rta. Al llamar a esta macro se actualizará attrlen . Debería usar RTA_OK para comprobar la validez del puntero devuelto.
RTA_LENGTH(len)	devuelve la longitud que se necesita para len bytes de datos más la cabecera.
RTA_SPACE(len)	devuelve la cantidad de espacio que se necesitarán en el mensaje con len bytes de datos.

EJEMPLO

Crear un **mensaje rtinetlink** para **configurar la MTU** de un dispositivo.

```

struct
{
    struct nlmsghdr      nh;
    struct ifinfomsg    if;
    char                  attrbuf[512];
} req;

struct rtattr * rta;
unsigned int mtu = 1000;
int rtinetlink_sk = socket(          PF_NETLINK,
                                SOCK_DGRAM,
                                NETLINK_ROUTE);

memset(&req, 0, sizeof(req));
req.nh.nlmsg_len = NLMSG_LENGTH(sizeof(struct ifinfomsg));
req.nh.nlmsg_flags = NLM_F_REQUEST;
req.nh.nlmsg_type = RTNL_NEWRULE;
req.ififi_family = AF_UNSPEC;

```

```
req.if.ifi_index = INTERFACE_INDEX;
req.if.ifi_change = 0xffffffff; /* ???*/
rta = (struct rtattr*)((char *)&req) +
NLMSG_ALIGN(n->nlmsg_len)
);
rta->rta_type = IFLA_MTU;
rta->rta_len = sizeof(unsigned int);
req.n.nlmsg_len = NLMSG_ALIGN(
req.n.nlmsg_len) +
RTA_LENGTH(sizeof(mtu))
);
memcpy(RTA_DATA(rta), &mtu, sizeof (mtu));
send(rtinetlink_sk, &req, req.n.nlmsg_len);
```

FALLOS

Esta página de manual es escasa e incompleta.

VÉASE TAMBIÉN

[rtinetlink\(7\)](#), [netlink\(7\)](#), [netlink\(3\)](#)

Página man de Linux

14 mayo 1999

[RTNETLINK\(3\)](#)

A continuación veremos:

man 5 protocols

PROTOCOLS(5) Manual del Programador de Linux PROTOCOLS(5)

NOMBRE

protocols - el fichero de definición de protocolos

DESCRIPCIÓN

Éste es un **fichero ASCII plano** que **describe** los distintos **protocolos DARPA para Internet** que están **disponibles en el subsistema TCP/IP**. Se debería consultar este fichero en vez de usar los números de los ficheros de cabecera ARPA, o, peor aún, adivinarlos. Estos **números se incluyen en el campo de protocolo de cualquier cabecera IP**.

Este **fichero no se debe modificar** porque los cambios pueden producir paquetes IP incorrectos. Los **números y nombres de los protocolos se definen en el DDN Network Information Center**.

Cada línea tiene el siguiente **formato**:

protocolo número alias ...

donde los **campos se delimitan por espacios o TABs**. Las líneas **vacías y las que comienzan con un '#'** no son tenidas en cuenta.

Las **descripciones de los campos** son las siguientes:

protocolo

el **nombre nativo del protocolo**. Por ejemplo: ip, tcp o udp.

número

el **número oficial para este protocolo** tal como irá en la cabecera IP.

alias

alias o **nombres alternativos opcionales para este protocolo**.

Este **fichero se puede distribuir en una red usando servicios de nombre como Yellow Pages/NIS o BIND/Hesiod**.

FICHEROS

/etc/protocols

El **fichero de definición de protocolos**.

VÉASE TAMBIÉN

getprotoent(3)

Guía del Servicio NIS (Páginas Amarillas)

Guía del Servicio BIND/Hesiod

Linux

18 Octubre 1995

PROTOCOLS(5)

A continuación examinaremos la página de manual en línea de comandos:

man 3 getprotoent

GETPROTOENT(3) Manual del Programador de Linux GETPROTOENT(3)

NOMBRE

**getprotoent, getprotobynumber, setprotoent,
endprotoent - obtienen una entrada del fichero de protocolos**

SINOPSIS

#include <netdb.h>

struct protoent *getprotoent(void);

struct protoent *getprotobynumber(const char *nombre);

struct protoent *getprotobynumber(int proto);

void setprotoent(int dejaloabierto);

void endprotoent(void);

DESCRIPCIÓN

La función **getprotoent()** lee la siguiente línea del fichero **/etc/protocols** y devuelve una estructura **protoent** que contiene los campos de que consta la línea. El fichero **/etc/protocols** se abre si es necesario.

La función **getprotobynumber()** devuelve una estructura **protoent** para la línea de **/etc/protocols** que concuerde con el nombre de protocolo nombre.

La función **getprotobynumber()** devuelve una estructura **protoent** para la línea que concuerde con el número de protocolo proto.

La función **setprotoent()** abre y rebobina el fichero **/etc/protocols**. Si dejaloabierto es verdad (1), entonces el fichero no se cerrará entre llamadas a **getprotobyname()** o a **getprotobynumber()**.

La función **endprotoent()** cierra **/etc/protocols**.

La **estructura protoent** se define en **<netdb.h>** así:

```
struct protoent {  
    char *p_name;      /* nombre oficial de protocolo */  
    char **p_aliases; /* lista de sinónimos */  
    int p_proto;       /* número de protocolo */  
}
```

Los **miembros de la estructura protoent** son:

p_name El **nombre oficial del protocolo**.

p_aliases Una **lista terminada en cero de nombres alternos** para el protocolo.

p_proto El **número del protocolo**.

VALOR DEVUELTO

Las funciones **getprotoent()**, **getprotobyname()** y **getprotobynumber()** devuelven la estructura **protoent**, o un puntero **NULL** si ocurre un error o si se llega al final del fichero.

FICHEROS

/etc/protocols

fichero con los datos de protocolos

CONFORME A

BSD 4.3

VÉASE TAMBIÉN

getservent(3), getnetent(3), protocols(5)

BSD

29 Enero 1998

GETPROTOENT(3)

Ahora fijaremos nuestra atención en:

man 3 getservent

GETSERVENT(3) Manual del Programador de Linux GETSERVENT(3)

NOMBRE

**getservent, getservbyname, getservbyport, setservent,
endservent - obtener valores de servicios**

SINOPSIS

```
#include <netdb.h>

struct servent *getservent(void);

struct servent *getservbyname(const char *name, const char *proto);

struct servent *getservbyport(int port, const char *proto);

void setservent(int stayopen);

void endservent(void);
```

DESCRIPCIÓN

La función **getservent()** lee la siguiente línea del fichero **/etc/services** y devuelve una estructura **servent** que contiene en sus campos los **campos de la línea**. Si es necesario, se abre el fichero **/etc/services**.

La función **getservbyname()** devuelve una estructura **servent** conteniendo los **campos de la línea** de **/etc/services** que contiene el servicio **name** y usa el protocolo **proto**.

La función **getservbyport()** devuelve una estructura de tipo **servent** con los **datos de la línea** que contiene el puerto **port** (con los **bytes en el orden de red**) y usa el protocolo **proto**.

La función **setservent()** abre y reinicia el fichero **/etc/services**. Si **stayopen** es **verdadero (1)**, entonces no se cerrará el fichero entre llamadas a las funciones **getservbyname()** y **getservbyport()**.

La función **endservent()** cierra el fichero **/etc/services**.

La **estructura servent** está **definida en <netdb.h>** como sigue:

```
struct servent {
    char    *s_name;          /* nombre oficial del servicio */
    char    **s_aliases;       /* lista de alias */
    int     s_port;           /* numero de puerto */
    char    *s_proto;          /* protocolo a usar */
}
```

Los **miembros de la estructura servent** son:

s_name El **nombre oficial del servicio**.

s_aliases Una **lista terminada en cero de nombres alternativos** para el servicio.

s_port El **numero de puerto para el servicio, con sus bytes en el orden de red**.

s_proto El **nombre del protocolo a usar con este servicio**.

VALOR DEVUELTO

Las funciones **getservent()**, **getservbyname()** y **getservbyport()** **devuelve** una **estructura de tipo servent**, o un puntero **NUL** si ha **ocurrido un error** o se ha alcanzado el final del fichero.

FICHEROS

/etc/services Fichero de base de datos de servicios

ANALISTA UNIVERSITARIO DE SISTEMAS

SISTEMAS OPERATIVOS – Prof. Macchi Guido

108/

CONFORME A

BSD 4.3

VÉASE TAMBIÉN

getprotoent(3), getnetent(3), services(5)

BSD

22-Abril-1996

GETSERVENT(3)

Ahora nos dispondremos a leer :

man 3 getnetent

GETNETENT(3) Manual del Programador de Linux GETNETENT(3)

NOMBRE

getnetent, getnetbyname, getnetbyaddr, setnetent, endnetent - obtienen una entrada del fichero de redes

SINOPSIS

```
#include <netdb.h>

struct netent *getnetent(void);

struct netent *getnetbyname(const char *nombre);

struct netent *getnetbyaddr(long red, int tipo);

void setnetent(int dejaloabierto);

void endnetent(void);
```

DESCRIPCIÓN

La función **getnetent()** lee la línea siguiente del fichero **/etc/networks** y devuelve una estructura **netent** que contiene los **campos** descompuestos correspondientes a la línea. El fichero **/etc/networks** se abre si es necesario.

La función **getnetbyname()** devuelve una estructura **netent** correspondiente a la línea de **/etc/networks** que concuerde con el nombre de red **nombre**.

La función **getnetbyaddr()** devuelve una estructura **netent** para la línea que concuerde con el número de red **net** de tipo **tipo**.

La función **setnetent()** abre y rebobina el fichero **/etc/networks**. Si **dejaloabierto** es verdadero (1), entonces el fichero no se cerrará entre llamadas a **getnetbyname()** y **getnetbyaddr()**.

La función **endnetent()** cierra **/etc/networks**.

La **estructura netent** se define en **<netdb.h>** como sigue:

```
struct netent {
    char      *n_name;      /* nombre oficial de red */
    char      **n_aliases;   /* lista de sinónimos */
    int       n_addrtype;   /* tipo de dirección de red */
    unsigned long int n_net; /* número de red */
}
```

Los **miembros** de la **estructura netent** son:

n_name El **nombre oficial de la red**.

n_aliases Una **lista terminada en cero de nombres alternativos** para la red.

n_addrtype	El tipo del número de red; siempre es AF_INET.
n_net	El número de red en orden de byte del ordenador ('`host'').

VALOR DEVUELTO

Las funciones **getnetent()**, **getnetbyname()** y **getnetbyaddr()** devuelven la estructura **netent**, o un puntero **NULL** si ocurre un error o se llega al final del fichero.

FICHEROS

/etc/networks	fichero de datos de redes
----------------------	---------------------------

CONFORME A

BSD 4.3
RFC 1101

VÉASE TAMBIÉN

getprotoent(3), getservent(3), networks(5)

BSD

24 Julio 1993

GETNETENT(3)

Seguimos con la página de manual en línea de comandos:
man 5 services

SERVICES(5) Manual del Programador de Linux SERVICES(5)

NOMBRE

services - Lista de servicios de red de Internet

DESCRIPCIÓN

services es un fichero ASCII que proporciona una correspondencia entre nombres textuales cómodos para los servicios de internet y sus correspondientes números de puerto y tipos de protocolo subyacentes. Todo programa de red debería mirar este fichero para obtener el número de puerto (y protocolo) para su servicio.

Las funciones **getservent(3)**, **getservbyname(3)**, **getservbyport(3)**, **setservent(3)**, y **endservent(3)** de la biblioteca de C, permiten consultar este fichero desde un programa.

Los números de puerto son asignados por la IANA (Internet Assigned Numbers Authority: Autoridad para la Asignación de Números de Internet), y su política actual es la de asignar tanto los protocolos TCP y UDP cuando se asigna un número de puerto. Por tanto, la mayoría de las entradas tendrán dos entradas, incluso para los servicios que son exclusivos de TCP.

Los **números de puerto por debajo de 1024** (los así llamados "puertos de baja numeración") sólo pueden ser **enlazados por el superusuario** (ver **bind(2)**, **tcp(7)**, y **udp(7)**.) Esto es así para que los clientes que se conecten a los puertos de baja numeración puedan **confiar en que el servicio ejecutándose en el puerto es la implementación estándar y no un servicio trámoso ejecutado por un usuario** de la máquina. Los números de puerto bien conocidos especificados por la IANA se localizan normalmente en este espacio exclusivo del superusuario.

La presencia de una entrada para un servicio en el fichero **services** no significa, necesariamente, que el servicio se está ejecutando actualmente en la máquina. Vea **inetd.conf(5)** para la configuración de los servicios ofrecidos de Internet. Dese cuenta que **no todos los servicios de red son iniciados por inetd(8)**, por lo que **no aparecerán en inetd.conf(5)**. En particular, los servidores de noticias (NNTP) y de correo (SMTP) frecuentemente se inician desde los guiones de arranque del sistema.

La localización del fichero **services** viene especificada por **_PATH_SERVICES** en **/usr/include/netdb.h**.

Normalmente, el valor asignado es **/etc/services**.

Cada línea describe un servicio, y tiene el formato:

service-name port/protocol [aliases ...]

donde:

service-name

es el **nombre amigable por el que el servicio es conocido** y buscado. Distingue entre mayúsculas y minúsculas. Normalmente, el programa cliente se especifica tras service-name.

port es el **número de puerto (en decimal) usado por este servicio**.

protocol es el **tipo de protocolo** usado. Este **campo debe coincidir con una entrada del fichero protocols(5)**. Los **valores típicos incluyen tcp y udp**.

aliases es una lista separada, opcionalmente, por espacios o tabuladores de otros nombres para este servicio (pero consulte más abajo la sección ERRORES). Nuevamente, **se distingue entre mayúsculas y minúsculas**.

Se pueden usar o bien espacios o bien tabuladores para separar los campos.

Los **comentarios comienzan con un '#' y terminan con un final de línea**. Las líneas en blanco se saltan.

service-name deben comenzar en la primera columna del fichero, ya que no se eliminan los espacios iniciales. service-names **puede ser cualquier secuencia de caracteres imprimibles, excepto espacios y tabuladores**, aunque se debe hacer una **selección conservativa de caracteres para minimizar problemas de interoperabilidad**. Es decir, los caracteres a-z, 0-9 y el guión (-) deben ser una elección sensata.

Las líneas que no coincidan con este formato no deberían estar presentes en el fichero. (Actualmente, **getservent(3)**, **getservbyname(3)** y **getservbyport(3)** las saltan silenciosamente. Sin embargo, no debería fiarse de este comportamiento.)

Como característica de compatibilidad hacia atrás, la barra inclinada (/) entre el número de puerto (port) y el nombre del protocolo (protocol) puede ser, de hecho, o bien una barra inclinada o bien una coma (,). El uso de la coma en instalaciones modernas se desprecia.

Este fichero se podría distribuir a través de una red usando un servicio de nombres de red como Yellow Pages/NIS o BIND/Hesiod.

Un ejemplo. El **fichero services podría tener el siguiente aspecto:**

netstat	15/tcp	
qotd	17/tcp	quote
msp	18/tcp	# message send protocol
msp	18/udp	# message send protocol
chargen	19/tcp	ttytst source
chargen	19/udp	ttytst source
ftp	21/tcp	
# 22 - unassigned		
telnet	23/tcp	

ERRORES

Hay un **máximo de 35 alias**, debido a la forma en que está escrito el código de **getservent(3)**.

Las **líneas con una longitud superior a BUFSIZ** (actualmente, **1024**) **caracteres serán ignoradas por getservent(3), getservbyname(3), y getservbyport(3)**. Sin embargo, esto también provocará que la siguiente línea sea analizada incorrectamente.

FICHEROS

/etc/services

La lista de servicios de red de Internet.

**/usr/include/netdb.h
Definición de _PATH_SERVICES**

VÉASE TAMBIÉN

**getservent(3), getservbyname(3), getservbyport(3), setservent(3),
endservent(3), protocols(5), listen(2), inetd.conf(5), inetd(8).**

RFC de Números Asignados, más recientemente **RFC 1700,**
(AKA STD0002)

Guide to Yellow Pages Service

Guide to BIND/Hesiod Service

Linux 11 Ene 1996 SERVICES(5)

Ahora seguiremos con :

man 5 networks

NETWORKS(5) Administración del Sistema Linux NETWORKS(5)

NOMBRE

networks – información de los nombres de redes

DESCRIPCIÓN

El archivo **/etc/networks** es un **archivo de texto plano ASCII** que **describe las redes conocidas DARPA y los nombres simbólicos para estas redes.**

Cada línea representa **una red** y **tiene la siguiente sintaxis:**

nombre número alias

donde los **campos son delimitados por espacios o tabuladores**. Las **líneas en blanco son ignoradas**. Si una **línea comienza con “#” la parte restante de la misma es ignorada**.

La **descripción de los campos** es la siguiente:

nombre El **nombre simbólico para la red**

número El **número oficial para la red en notación decimal**. La terminación “.0” puede omitirse.

alias Los **nombres opcionales para la red**.

Este **archivo es leído por** las utilidades **route o netstat**. **Sólo son soportadas las redes de Clase A, B o C, las redes particionadas (por ejemplo red/26 o red/28) no son soportadas.**

ARCHIVOS

/etc/networks

El archivo de definición de las redes.

VÉASE TAMBIÉN

**getnetbyaddr(3), getnetbyname(3), getnetent(3), route(8),
netstat(8)**

GNU/Linux

2001-12-22

NETWORKS(5)

Ahora veremos la página del manual en línea de comandos:

man 8 inetd

INETD(8) BSD System Manager's Manual INETD(8)

NOMBRE

inetd - internet ``super-server''

SINOPSIS

inetd [-d] [-q queuelength] [fichero de configuración]

DESCRIPCIÓN

Inetd debería ejecutarse en el **arranque mediante /etc/rc.local** (**véase rc(8)**). A partir de ese momento está **a la escucha de conexiones en cierto conector (socket) de internet**. Cuando **encuentra** una **conexión** en uno de sus conectores, **decide** a **qué servicio** de conexión **corresponde**, y **llama a un programa para atender la solicitud**. Cuando este **programa termina**, **continúa a la escucha en el conector** (salvo en algún caso que se describirá más adelante). Esencialmente, **inetd permite ejecutar un demonio para llamar a otros muchos, reduciendo la carga del sistema**.

La **opciones disponibles** para inetd son:

-d Activa la depuración.

-q longitudcola

Asigna el valor indicado al **tamaño de la cola de escucha del conector**. Por defecto es **128**.

En ejecución, **inetd** lee su información de configuración de un **fichero de configuración**, que por defecto es **/etc/inetd.conf**. Tiene que **haber una entrada para cada campo del fichero de configuración**, con entradas para cada campo **separadas por tab o espacios**. Los **comentarios** se distinguen por **un ``#'' al principio de la línea**. Tiene que haber una entrada para cada campo.

Los campos del fichero de configuración son de la siguiente forma:

**nombre de servicio
tipo de conector
protocolo
wait/nowait[.max]
usuario[.grupo]
programa servidor
argumentos del programa servidor**

Para especificar un **servicio basado en Sun-RPC** la entrada debería contener estos campos.

**nombre servicio/versión
tipo de conector
rpc/protocolo
wait/nowait[.max]
usuario[.grupo]
programa servidor
argumentos del programa servidor**

La entrada **nombre** de servicio es el **nombre de un servicio válido** del fichero **/etc/services**. Para **servicio ``internos''** (discutidos después), el **nombre de servicio tiene que ser el nombre oficial del servicio** (esto es, la primera entrada de **/etc/services**). Cuando se usa para especificar un **servicio basado en Sun-RPC**, este campo es un **nombre de servicio RPC válido** del fichero **/etc/rpc**. La parte a la derecha de ``/'' es el **número de versión RPC**. Esto puede ser simplemente un **argumento numérico o un rango de versiones**. Un rango está acotado por las versiones menor y mayor - ``rusers/1-3''.

El **tipo de conector (tipo de socket)** debería ser ``stream'', ``dgram'', ``raw'', ``rdm'', o ``seqpacket'', dependiendo de si el conector es un **flujo, datagrama, en bruto, mensaje entregado fiable o conector de paquetes secuenciados**.

El **protocolo** tiene que ser **un protocolo válido** como los dados en **/etc/protocols**. Pueden ser ejemplos ``tcp'' o ``udp''. Los **servicios basados en Rpc se especifican con el tipo de servicio ``rpc/tcp'' o ``rpc/udp''**.

La entrada **wait/nowait** es aplicable a conectores de **datagrama sólo** (los otros conectores deberían tener una entrada ``nowait'' en este espacio). Si un **servidor de datagrama conecta a su par, liberando el conector, así inetd puede recibir posteriores mensajes en el conector, esto se dice que es un servidor ``multi-hilo'' y debería usar la entrada ``nowait''**.

Para los **servidores de datagramas que procesan** todos los **datagramas entrantes por un conector y al fin y al cabo desconecta**, el servidor se dice que es ``hilo simple'' y debería usar una entrada ``wait''. Comsat(8) (biff(1)) y talkd(8) son ambos ejemplos del último tipo de servidor de datagramas.

Tftpd(8) es una excepción; es un **servidor de datagrama** que establece **pseudoconexiones**. Este se debe indicar como ``wait'' con el fin de evitar una carrera; el **servidor lee el primer paquete, crea un nuevo coonector**, entonces **se desdobra (forks(2)) y sale(exit(2)) para permitir que inetd verifique nuevas solicitudes de servicio para activar nuevos servidores**.

El sufijo opcional ``max'' (separado de ``wait'' o ``nowait'' por un punto) especifica el **máximo numero de instancias del servidor que se pueden activar desde inetd en un intervalo de 60 segundos**. Cuando se omite ``max'' toma el valor por defecto de 40.

La entrada **usuario** debería contener el **nombre de usuario bajo el que ejecutaría el servidor**. Esto **permite que a los servidores se les dé menos permisos que los que posee root**. Se puede especificar un **nombre de grupo opcional añadiendo un punto al nombre de usuario seguido por el nombre de grupo**.

ANALISTA UNIVERSITARIO DE SISTEMAS

SISTEMAS OPERATIVOS – Prof. Macchi Guido

116/

Esto permite a los servidores ejecutarse con un identificador de grupo (primario) diferente al especificado en el fichero /etc/passwd. Si se especifica un grupo y el usuario no es root, se asignan los grupos suplementarios asociados con ese usuario.

La entrada **programa** servidor debería contener la **ruta completa del programa que se ejecutará por inetd cuando encuentre una solicitud en su conector**. Si inetd proporciona este servicio internamente, esta entrada debería ser ``internal''.

Los **argumentos del programa servidor** serán **como son** normalmente los argumentos (de un programa en C), empezando con argv[0], que es el **nombre del programa**. Si proporciona este servicio internamente, la palabra ``internal'' debería estar en el lugar de esta entrada.

Inetd proporciona varios servicios ``triviales'' internamente usando rutinas con él mismo. ``echo'', ``discard'', ``chargen'' (generador de caracteres), ``daytime'' (fecha-hora en formato legible), y ``time'' (fecha-hora formato de máquina, en el formato del número de segundos desde medianoche de 1 de enero de 1900). Todos estos servicios están basados en tcp. Para **detalles** de estos servicios, **consulte el RFC** adecuada del Network Information Center.

Inetd relee su fichero de configuración cuando recibe la señal de colgar **SIGHUP**. Se pueden añadir servicios, borrarlos o modificarlos cuando se lee el fichero de configuración. Inetd crea el fichero **/var/run/inetd.pid** que contiene su identificador de proceso.

VÉASE TAMBIÉN

comsat(8), fingerd(8), ftpd(8), rexecd(8), rlogind(8), rshd(8), telnetd(8), tftpd(8)

HISTORIA

El comando **inetd** apareció en **4.3BSD**. El soporte para servicios basados en Sun-RPC se ha servido del modelo proporcionado por **SunOS 4.1**.

Linux NetKit 0.09

23 Noviembre 1996

Linux NetKit 0.09

Ahora continuaremos con la página de manual en línea de comandos:

man 3 cmsg

CMSG(3)

Manual del Programador de Linux

CMSG(3)

NOMBRE

CMSG_ALIGN, CMSG_SPACE, CMSG_NXTHDR, CMSG_FIRSTHDR - Acceso a datos auxiliares.

SINOPSIS

#include <sys/socket.h>

```
struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *msgh);
struct cmsghdr *CMSG_NXTHDR(struct msghdr *msgh,
                           struct cmsghdr *cmsg);
size_t CMSG_ALIGN(size_t length);
size_t CMSG_SPACE(size_t length);
size_t CMSG_LEN(size_t length);
void *CMSG_DATA(struct cmsghdr *cmsg);

struct cmsghdr {
    socklen_t cmsg_len;      /* cantidad de bytes de datos,
                               incluyendo la cabecera */
    int       cmsg_level;    /* protocolo originario */
    int       cmsg_type;     /* tipo específico del protocolo */
    /* seguido de unsigned char           cmsg_data[]; */
};
```

DESCRIPCIÓN

Estas **macros** se usan para crear y acceder a mensajes de control (también llamados datos auxiliares) que no son parte del contenido útil de un conector. Esta información de control puede incluir la interfaz en la

que se **ha recibido el paquete**, diferentes **campos de cabecera usados raramente**, una **descripción de error** ampliada, un **conjunto de descriptores de fichero o credenciales de Unix**. Por ejemplo, los mensajes de control **se pueden usar para enviar campos de cabecera adicionales** tales como **opciones IP**. Los **datos auxiliares se envían llamando a sendmsg(2)** y **se reciben llamando a recvmsg(2)**. Vea sus páginas de manual para más información.

Los **datos auxiliares** son una **secuencia de estructuras struct cmsghdr con datos añadidos**. Sólo se debería **acceder** a esta secuencia **usando las macros descriptas** en esta página de manual y **nunca directamente**. Vea las páginas de manual específicas del protocolo para conocer los tipos de mensajes de control disponibles. El **tamaño máximo permitido del buffer auxiliar por conector** se **puede configurar con la sysctl net.core.optmem_max**.

Vea **socket(7)**.

CMSG_FIRSTHDR	devuelve un puntero a la primera cmsghdr en el buffer de datos auxiliares asociado con la msghdr pasada.
CMSG_NXTHDR	devuelve la siguiente cmsghdr válida después de la cmsghdr pasada. Devuelve NULL cuando no queda suficiente espacio en el buffer.
CMSG_ALIGN	dada una longitud, la devuelve incluyendo la alineación necesaria. Ésta es una expresión constante .
CMSG_SPACE	devuelve la cantidad de bytes que ocupa un elemento auxiliar cuyo contenido útil tiene la longitud de datos pasada. Ésta es una expresión constante .
CMSG_DATA	devuelve un puntero a la porción de datos de una cmsghdr.
CMSG_LEN	devuelve el valor a almacenar en el miembro cmsg_len de la estructura cmsghdr teniendo en cuenta cualquier alineación necesaria. Toma como argumento la longitud de los datos . Ésta es una expresión constante .

Para **crear datos auxiliares**, inicialice primero el **miembro msg_controllen de la estructura msghdr con el tamaño del buffer de mensajes de control**. Use **CMSG_FIRSTHDR** sobre **msghdr** para **obtener el primer mensaje de control** y **CMSG_NEXTHDR** para **obtener los**

siguientes. En cada mensaje de control, inicialice `cmsg_len` (con `CMSG_LEN`), los otros campos cabecera de `cmsghdr` y la parte de datos usando `CMSG_DATA`. Finalmente, debería asignar al campo `msg_controllen` de `msghdr` la suma de los `CMSG_SPACE` de las longitudes de todos los mensajes de control del buffer. Para más información sobre `msghdr`, vea `recvmsg(2)`.

Cuando el buffer de mensajes de control es demasiado pequeño para almacenar todos los mensajes, se activa la bandera `MSG_CTRUNC` en el miembro `msg_flags` de `msghdr`.

EJEMPLO

Este código busca la opción `IP_TTL` en un buffer auxiliar recibido:

```
struct msghdr msgh;
struct cmsghdr *cmsg;
int *ttlptr;
int received_ttl;

/* Recibir los datos auxiliares en msgh */
for (cmsg = CMSG_FIRSTHDR(&msgh);
     cmsg != NULL;
     cmsg = CMSG_NXTHDR(&msgh,cmsg)) {
    if (cmsg->cmsg_level == SOL_IP
        && cmsg->cmsg_type == IP_TTL) {
        ttlptr = (int *) CMSG_DATA(cmsg);
        received_ttl = *ttlptr;
        break;
    }
}
if (cmsg == NULL) {
    /* Error: IP_TTL no habilitada o buffer pequeño o
     * error de E/S.
     */
}
```

El siguiente código pasa un **vector de descriptores de ficheros** mediante un conector Unix usando **SCM_RIGHTS**:

```
struct msghdr msg = {0};  
struct cmsghdr *cmsg;  
int myfds[NUM_FD];  
/* Los descriptores de fichero a pasar. */  
char buf[CMSG_SPACE(sizeof myfds)];  
/* buffer de datos auxiliares */  
int *fdptr;  
msg.msg_control = buf;  
msg.msg_controllen = sizeof buf;  
cmsg = CMSG_FIRSTHDR(&msg);  
cmsg->cmsg_level = SOL_SOCKET;  
cmsg->cmsg_type = SCM_RIGHTS;  
cmsg->cmsg_len = CMSG_LEN(sizeof(int) * NUM_FD);  
/* Inicializar el contenido útil: */  
fdptr = (int *)CMSG_DATA(cmsg);  
memcpy(fdptr, myfds, NUM_FD * sizeof(int));  
/* Sumar la longitud de todos los mensajes de  
control en el buffer: */  
msg.msg_controllen = cmsg->cmsg_len;
```

NOTAS

Para transportabilidad, sólo se debería acceder a los datos auxiliares usando las macros descritas aquí. **CMSG_ALIGN** es una extensión de Linux y no debería usarse en programas transportables.

En Linux, **CMSG_LEN**, **CMSG_DATA** y **CMSG_ALIGN** son expresiones constantes (suponiendo que su argumento sea constante). Esto se podría usar para declarar el tamaño de variables globales pero, sin embargo, podría no ser transportable.

CONFORME A

El modelo de datos auxiliares sigue el borrador **POSIX.1003.1g**, **4.4BSD-Lite**, la API avanzada de IPv6 descrita en **RFC2292** y the Single Unix specification v2. **CMSG_ALIGN** es una extensión de Linux.

VÉASE TAMBIÉN

sendmsg(2), **recvmsg(2)**

RFC 2292

Página man de Linux

2 octubre 1998

CMSG(3)

Nos disponemos ahora a leer la página de manual en línea:

man 2 sendmsg

SEND(2)

Manual del Programador de Linux

SEND(2)

NOMBRE

send, sendto, sendmsg - envía un mensaje de un conector (socket)

SINTAXIS

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int send(int s, const void *msg, size_t len, int flags);
int sendto(int s,
           const void *msg,
           size_t len,
           int flags,
           const struct sockaddr *to,
           socklen_t tolen);
int sendmsg(int s, const struct msghdr *msg, int flags);
```

DESCRIPCIÓN

Send, sendto y sendmsg son utilizados para transmitir un mensaje a otro conector. Send solo puede ser usado cuando un conector está en un estado connected mientras sendto y sendmsg pueden ser utilizados en cualquier momento.

La dirección de destino viene dada por to con tolen especificando su tamaño. La longitud del mensaje viene dada por len. Si el mensaje es demasiado largo para pasar automáticamente a través del protocolo inferior, se devuelve el error EMSGSIZE y el mensaje no es transmitido.

La llamada send lleva implícita el que no se indiquen los posibles errores en la entrega. Los errores detectados localmente se indican devolviendo un valor -1.

Cuando el mensaje no cabe en el buffer de envío del conector, send se bloquea, a no ser que el conector se haya colocado en el modo de E/S no bloqueante. En el modo no bloqueante devolvería EAGAIN en este caso. Se puede utilizar la llamada select(2) para determinar cuando es posible enviar más información.

El parámetro flags es una palabra de opciones y puede contener las siguientes opciones:

MSG_OOB

Enviar datos fuera de orden (out-of-band) en conectores que soportan esta noción (p.ej. SOCK_STREAM); el protocolo subyacente también debe soportar datos fuera de orden.

MSG_DONTROUTE

No usar un ``gateway'' para enviar el paquete, enviar sólo a los ordenadores que se encuentren en redes conectadas directamente. Normalmente, esto sólo lo utilizan los programas de diagnóstico y enrutamiento. Esta **opción** sólo está **definida para familias de protocolos que enrutan**. Los **conectores de paquetes no enrutan**.

MSG_DONTWAIT

Habilitar el funcionamiento no bloqueante. Si la operación se bloqueara, se devolvería EAGAIN (esto también se puede habilitar usando la bandera O_NONBLOCK con la operación F_SETFL de fcntl(2)).

MSG_NOSIGNAL

Solicitar el no enviar SIGPIPE en caso de error en conectores orientados a conexión cuando el otro extremo rompa la conexión. Todavía se devuelve el error EPIPE.

Vea recv(2) para una descripción de la estructura msghdr. Puede enviar información de control usando los miembros msg_control y msg_controllen. La longitud máxima del buffer de control que el núcleo puede procesar por conector está limitada por la sysctl net.core.optmem_max. Vea socket(7).

VALOR DEVUELTO

Las llamadas devuelven el **numero de caracteres enviados, o -1 si ha ocurrido un error**.

ERRORES

Estos son **algunos errores estándares generados por la capa de conectores**. Los módulos de los protocolos subyacentes pueden generar y devolver errores adicionales. Vea sus páginas de manual respectivas.

EBADF Se ha especificado un descriptor no válido.

ENOTSOCK

El argumento s no es un conector.

EFAULT Se ha especificado como parámetro una dirección incorrecta del espacio de usuario. tro.

EMSGSIZE

El conector requiere que este **mensaje sea enviado automáticamente, y el tamaño** del mensaje a ser enviado **lo hace imposible**.

EAGAIN o EWOULDBLOCK

El conector está marcado como no bloqueante y la operación solicitada lo bloquearía.

ENOBUFS

La **cola de salida de la interface de red está llena**. Esto generalmente **indica** que el **interfaz ha parado de enviar**, pero **puede ser causado por una congestión temporal**. (Esto **no puede ocurrir en Linux**, los **paquetes** simplemente **se suprimen silenciosamente cuando la cola de un dispositivo se desborda**.)

EINTR Se ha producido una **señal**.

ENOMEM No hay memoria disponible.

EINVAL Se ha pasado un **argumento inválido**.

EPIPE Se ha desconectado el **extremo local en un conector orientado a conexión (TCP)**. En este caso el **proceso** también **recibirá** una señal **SIGPIPE** a menos que se **active** la **opción MSG_NOSIGNAL**.

CONFORME A

4.4BSD, SVr4, borrador POSIX 1003.1g (estas llamadas a función aparecieron en 4.2BSD).

NOTA

Los **prototipos** indicados más arriba **siguen 'the Single Unix Specification'**, ya que **glibc2 también lo hace**; el argumento flags era `int' en BSD 4.* pero `unsigned int' en libc4 y libc5; el argumento len era `int' en BSD 4.* y libc4 pero `size_t' en libc5; el argumento tolen era `int' en BSD 4.*, libc4 y libc5. Vea también **accept(2)**.

VÉASE TAMBIÉN

fcntl(2), recv(2), select(2), getsockopt(2), sendfile(2), socket(2), write(2), socket(7), ip(7), tcp(7), udp(7)

Página man de Linux

julio 1999

SEND(2)

A continuación veremos:

man 2 recvmsg

RECV(2)

Manual del Programador de Linux

RECV(2)

NOMBRE

recv, recvfrom, recvmsg - reciben un mensaje desde un conector

SINOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int recv(int s, void *buf, size_t lon, int flags);
```

```
int recvfrom( int s,  
              void *buf,  
              size_t lon,  
              int flags,  
              struct sockaddr *desde,  
              socklen_t *londesde);  
  
int recvmsg(int s, struct msghdr *msg, int flags);
```

DESCRIPCIÓN

Las llamadas **recvfrom** y **recvmsg** se emplean para **recibir mensajes desde un conector** ("socket"), y pueden utilizarse para **recibir datos de un conector sea orientado a conexión o no**.

Si "desde" no es NULL y el **conector no es orientado a conexión**, la dirección fuente del mensaje se llena. El argumento "londesde" es un **parámetro pasado por referencia**, inicializado al **tamaño del búfer asociado con desde**, y **modificado cuando la función regresa para indicar el tamaño real de la dirección guardada ahí**.

La llamada a **recv** se utiliza normalmente sólo en un **conector conectado** (vea **connect(2)**) y es **idéntica a recvfrom con** un parámetro "desde" con valor NULL.

Las tres rutinas **devuelven la longitud del mensaje cuando terminan bien**. Si un **mensaje es demasiado largo** como para caber en el **búfer suministrado**, los **bytes que sobran pueden descartarse dependiendo del tipo de conector del que se reciba el mensaje** (vea **socket(2)**).

Si no hay mensajes disponibles en el conector, las llamadas de recepción esperan que llegue un mensaje, a menos que el **conector sea no bloqueante** (vea **fcntl(2)**) en cuyo caso se **devuelve el valor -1** y la variable externa **errno toma el valor EAGAIN**. Las llamadas de recepción devuelven normalmente **cualquier dato disponible, hasta la cantidad pedida, en vez de esperar la recepción** de la cantidad pedida completa.

Las **llamadas select(2)** o **poll(2)** pueden emplearse para determinar cuándo llegan más datos.

El argumento **flags** de una llamada a **recv** se forma aplicando el operador de bits O-lógico a uno o más de los valores siguientes:

MSG_OOB

Esta opción **pide la recepción de datos fuera-de-banda que no se recibirían en el flujo de datos normal**. Algunos protocolos ponen **datos despachados con prontitud en la cabeza de la cola de datos normales**, y así, esta opción no puede emplearse con tales protocolos.

MSG_PEEK

Esta opción hace que la operación de **recepción devuelva datos del principio de la cola de recepción sin quitarlos de allí**. Así, **una próxima llamada de recepción devolverá los mismos datos**.

MSG_WAITALL

Esta opción hace que la **operación se bloquee hasta que se satisfaga la petición completamente**. Sin embargo, la llamada **puede aún devolver menos datos de los pedidos si se captura una señal, si ocurre un error o una desconexión, o si los próximos datos que se van a recibir son de un tipo diferente del que se ha devuelto**.

MSG_NOSIGNAL

Esta opción **desactiva el que se produzca una señal SIGPIPE sobre los conectores orientados a conexión (TCP) cuando el otro extremo desaparece**.

MSG_ERRQUEUE

Esta opción **indica que los errores encolados deben recibirse desde la cola de errores de conectores**. El error se pasa en un mensaje auxiliar con **un tipo dependiente del protocolo (para IPv4 éste es IP_RECVERR)**. El usuario debe proporcionar un buffer de tamaño suficiente.

Vea **cmsg(3)** para obtener más información sobre mensajes auxiliares.

El error se suministra en una estructura **sock_extended_err**:

```
#define SO_EE_ORIGIN_NONE      0
#define SO_EE_ORIGIN_LOCAL      1
#define SO_EE_ORIGIN_ICMP        2
#define SO_EE_ORIGIN_ICMP6       3

struct sock_extended_err
{
    u_int32_t    ee_errno;          /* número de error */
    u_int8_t     ee_origin;         /* origen del error */
    u_int8_t     ee_type;           /* tipo */
    u_int8_t     ee_code;           /* código */
    u_int8_t     ee_pad;
    u_int32_t    ee_info;           /* información adicional */
    u_int32_t    ee_data;           /* otros datos */
    /* Pueden ir más datos a continuación */
};

struct sockaddr *SOCK_EE_OFFENDER(struct sock_extended_err *);
```

ee_errno	contiene el número errno del error encolado .
ee_origin	es el código del origen en donde se ha originado el error . Los otros campos son específicos del protocolo . La macro SOCK_EE_OFFENDER devuelve un puntero a la dirección del objeto de red desde donde se ha originado el error dando un puntero al mensaje auxiliar. Si esta dirección se desconoce , el miembro sa_family de sockaddr contiene AF_UNSPEC y los otros campos de sockaddr quedan indefinidos . El contenido útil del paquete que ha producido el error se pasa como datos normales .

Para los errores locales no se pasa ninguna dirección (esto se puede comprobar con el miembro **cmsg_len** de **cmsghdr**). Para los errores recibidos, se asigna **MSG_ERRQUEUE** a **msghdr**. Después de que se haya pasado un error, el error de conector pendiente se regenera basándose en el siguiente error encolado y se pasará en la siguiente operación de conectores.

La llamada **recvmsg** utiliza una **estructura msghdr** para minimizar el número de parámetros suministrados directamente. Esta **estructura tiene** la forma siguiente, según se define en **<sys/socket.h>**:

```
struct msghdr {  
    void * msg_name;          /* dirección opcional */  
    socklen_t msg_namelen;   /* tamaño de la dirección */  
    struct iovec * msg iov;  /* vector dispersar/agrupar */  
    size_t msg iovlen;       /* nº de elementos en msg iov */  
    void * msg control;     /* datos auxiliares, ver más abajo */  
    socklen_t msg controllen; /* long buffer datos auxiliares */  
    int msg flags;           /* opciones en mensaje recibido */  
};
```

Aquí **msg_name** y **msg_namelen** especifican la dirección de destino si el conector está desconectado; **msg_name** puede darse como un puntero nulo si no se desean o requieren nombres. Los campos **msg iov** y **msg iovlen** describen localizaciones dispersar/agrupar, como se discute en **readv(2)**. El campo **msg_control**, que tiene de longitud **msg_controllen**, apunta a un búfer para otros mensajes relacionados con control de protocolo o para datos auxiliares diversos. Cuando se llama a **recvmsg**, **msg_controllen** debe contener la longitud del buffer disponible en **msg_control**; a la vuelta de una llamada con éxito contendrá la longitud de la secuencia de mensajes de control.

Los **mensajes son de la forma:**

```
struct cmsghdr {  
    socklen_t cmsg_len;      /* Nº de byte de datos, incluye cab. */  
    int       cmsg_level;    /* protocolo originante */  
    int       cmsg_type;     /* tipo específico del protocolo */  
    /* seguido por */  
    u_char   cmsg_data[]; */  
};
```

Los **datos auxiliares sólo deberían ser accedidos mediante las macros definidas en cmsg(3).**

Como ejemplo, **Linux usa este mecanismo de datos auxiliares para pasar errores ampliados, opciones IP o descriptores de fichero mediante conectores Unix.**

El campo **msg_flags** toma un **valor al regresar dependiendo del mensaje recibido**. **MSG_EOR** indica **fin-de-registro**; los **datos devueltos completaron un registro** (generalmente **empleado con conectores** del tipo **SOCK_SEQPACKET**). **MSG_TRUNC** indica que la **porción trasera de un datagrama ha sido descartada** porque el **datagrama era más grande que el búfer suministrado**. **MSG_CTRUNC** indica que **algún dato de control ha sido descartado debido a la falta de espacio en el búfer para datos auxiliares**. **MSG_OOB** se devuelve para indicar que **se han recibido datos despachados prontamente o fuera-de-banda**.

MSG_ERRQUEUE indica que **no se ha recibido ningún dato sino un error ampliado de la cola de errores de conectores**.

VALOR DEVUELTO

Estas llamadas **devuelven el número de bytes recibidos, o bien -1 en caso de** que ocurriera un **error**.

ERRORES

Estos son **algunos errores estándares generados por la capa de conectores**. Los modulos de los protocolos subyacentes pueden generar y devolver errores adicionales. Consulte sus páginas de manual.

EBADF El **argumento s** es un **descriptor inválido**.

ENOTCONN

El **conector está asociado con un protocolo orientado a la conexión y no ha sido conectado** (vea **connect(2)** y **accept(2)**).

ENOTSOCK

El **argumento s no se refiere a un conector**.

EAGAIN El **conector está marcado como no-bloqueante, y la**

operación de recepción produciría un bloqueo, o se ha puesto un límite de tiempo en la recepción, que ha expirado antes de que se recibieran datos.

EINTR La recepción ha sido interrumpida por la llegada de una señal antes de que hubiera algún dato disponible.

EFAULT El puntero a búfer de recepción (o punteros) apunta afuera del espacio de direcciones del proceso.

EINVAL Se ha pasado un argumento inválido.

CONFORME A

4.4BSD (estas funciones aparecieron por primera vez en 4.2BSD).

NOTA

Los prototipos dados anteriormente siguen a glibc2. The Single Unix Specification coincide en todo excepto en que el tipo de los valores devueltos es `ssize_t' (mientras que BSD 4.* , libc4 y libc5 tienen `int'). El argumento flags es un `int' en BSD 4.* pero es un `unsigned int' en libc4 y libc5. El argumento lon es un `int' en BSD 4.* pero es un `size_t' en libc4 y libc5. El argumento londesde es un `int' en BSD 4.* , libc4 y libc5. El actual `socklen_t' fue inventado por POSIX.

Vea también **accept(2)**.

VÉASE TAMBIÉN

fcntl(2), read(2), select(2), getsockopt(2), socket(2), cmsg(3)

Página man de Linux

abril 1999

RECV(2)

Ahora continuaremos con la lectura de: **man 2 getsockopt**
que es la misma que: **man 2 setsockopt**

GETSOCKOPT(2) Manual del Programador de Linux GETSOCKOPT(2)

NOMBRE

getsockopt, setsockopt - obtiene y pone opciones en conectores (sockets)

SINOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int getsockopt( int s, int nivel, int nomopc, void *valopc,
                socklen_t *lonopc);
```

```
int setsockopt( int s, int nivel, int nomopc, const void *valopc,  
                socklen_t lonopc);
```

DESCRIPCIÓN

getsockopt y **setsockopt** manipulan las **opciones asociadas a un conector**. Éstas **pueden existir en múltiples niveles de protocolo**; **siempre están presentes en el nivel más alto de conector**.

Al manipular opciones de conector, **deben especificarse el nivel en el que reside la opción, y su nombre**.

Para manipular **opciones en el nivel de conector**, nivel se especifica como **SOL_SOCKET**. Para manipular opciones a **cualquier otro nivel**, se **suministra el número de protocolo del apropiado que controle la opción**. Por ejemplo, para indicar que una opción ha de ser interpretada por el protocolo TCP, nivel debe ponerse como el número de protocolo de TCP; vea **getprotoent(3)**.

Los parámetros ``**valopc**'' y ``**lonopc**'' se emplean **para acceder a valores de opciones de setsockopt**. Para **getsockopt** identifican a un búfer en el que **se pondrá el valor para la opción pedida (u opciones)**.

Para **getsockopt**, ``**lonopc**'' es un parámetro por referencia, que **contiene inicialmente el tamaño del búfer apuntado por ``optval'', y que se modifica al acabar la función para contener el tamaño real del valor devuelto**. Si no se va a suministrar o devolver un **valor de opción**, ``**valopc**'' puede ser **NULL**.

``**nomopc**'' y **cualesquiera opciones especificadas se pasan sin interpretar al módulo de protocolo apropiado para su interpretación**. El fichero de cabecera **<sys/socket.h>** **contiene definiciones para opciones de nivel de conector**, descritas más abajo. Las **opciones a otros niveles de protocolo varían en formato y nombre**; consulte las páginas apropiadas de la sección 4 del Manual.

La **mayoría de las opciones de nivel-conector utilizan un parámetro int para ``valopc''**. Para **setsockopt**, el parámetro debe ser distinto de cero para permitir una opción booleana, o cero si la opción va a ser deshabilitada.

Para una descripción de las opciones disponibles para conectores vea **socket(7)** y las páginas de manual del protocolo apropiado.

VALOR DEVUELTO

Se **devuelve cero en caso de éxito**. En caso de error se devuelve **-1** y **errno toma un valor apropiado**.

ERRORES

EBADF El argumento **s** no es un descriptor válido.

ENOTSOCK

El argumento **s** es un fichero, no un conector.

ENOPROTOOPT

La opción es desconocida al nivel indicado.

EFAULT La dirección apuntada por ``valopc'' no está en un sitio válido del espacio de direcciones del proceso. Para **getsockopt**, este error puede también ser devuelto si ``lonopc'' no está en un sitio válido del espacio de direcciones del proceso.

CONFORME A

SVr4, 4.4BSD (estas primitivas aparecieron por primera vez en 4.2BSD). **SVr4** documenta los **códigos de error adicionales ENOMEM** y **ENOSR**, pero **no documenta** las opciones **SO_SNDLOWAT**, **SO_RCVLOWAT**, **SO_SNDFTIMEO** ni **SO_RCVTIMEO**

NOTA

El quinto argumento de **getsockopt** y **setsockopt** es en realidad un entero [\[*\]](#) (y esto es lo que tienen **BSD 4.***, **libc4** y **libc5**). Cierta confusión en **POSIX** dio como resultado el actual **socklen_t**. El estándar propuesto todavía no ha sido adoptado pero glibc2 ya lo sigue y también tiene **socklen_t** [\[*\]](#). Vea también [accept\(2\)](#).

FALLOS

Algunas de las opciones de conector deberían ser manejadas a niveles más bajos del sistema.

VÉASE TAMBIÉN

ioctl(2), **socket(2)**, **getprotoent(3)**, **protocols(5)**, **socket(7)**, **unix(7)**, **tcp(7)**

Página man de Linux

24 mayo 1999

GETSOCKOPT(2)

Continuamos con la página de manual en línea de comandos:
man 2 getsockname

GETSOCKNAME(2) Manual del Programador de Linux GETSOCKNAME(2)

NOMBRE

getsockname - obtener nombre de conexión

SINOPSIS

#include <sys/socket.h>

int getsockname(int s, struct sockaddr * name, socklen_t * namelen)

DESCRIPCIÓN

getsockname devuelve el nombre actual para la conexión indicada. El parámetro ``namelen'' debe ser inicializado para indicar la cantidad de espacio apuntado por ``name''. La devolución contiene el tamaño actual del nombre devuelto (en bytes).

VALOR DEVUELTO

Si es correcto, devuelve un cero. Si hay error, devuelve -1, y se asigna a ``errno'' un valor apropiado.

ERRORES

EBADF El argumento s no es un descriptor válido.

ENOTSOCK

El argumento s es un fichero, no una conexión.

ENOBUFS

No había suficientes recursos disponibles en el sistema para llevar a cabo la operación.

EFAULT El parametro ``name'' apunta a una memoria que no está dentro de una zona válida del espacio de direcciones del proceso.

CONFORME A

SVr4, 4.4BSD (la función getsockname apareció en 4.2BSD). SVr4 documenta dos códigos de error adicionales, ENOMEM y ENOSR.

NOTA

El tercer argumento de getsockname es en realidad un entero (y esto es lo que tienen BSD 4.* , libc4 y libc5). Cierta confusión en **POSIX** dió como resultado el actual **socklen_t**. El estándar propuesto todavía no ha sido adoptado pero glibc2 ya lo sigue y también tiene **socklen_t**. Vea también **accept(2)**.

VÉASE TAMBIÉN

bind(2), socket(2)

Página de Manual BSD

24 julio 1993

GETSOCKNAME(2)

Seguimos con la página de manual:

man 2 socketpair

SOCKETPAIR(2) Manual del Programador de Linux SOCKETPAIR(2)

NOMBRE

socketpair - crea un par de conectores conectados

SINOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int socketpair(int d, int tipo, int protocolo, int sv[2]);
```

DESCRIPCIÓN

La llamada a **crea** una **pareja sin nombre de conectores ('sockets', 'enchufes) en el dominio especificado por 'd'**, del tipo especificado por ``tipo'', y empleando opcionalmente el protocolo especificado por ``protocolo''. Los descriptores utilizados para referenciar los **nuevos conectores se devuelven en sv[0] y sv[1]**. Los dos conectores son indistinguibles.

VALOR DEVUELTO

En caso de éxito, se devuelve cero. En caso de error, se devuelve -1 y se pone en ``errno'' un valor apropiado.

ERRORES

EMFILE Demasiados descriptores están en uso por este proceso.

EAFNOSUPPORT

La familia de direcciones especificada **no está admitida** en esta máquina.

EPROTONOSUPPORT

El protocolo especificado no está admitido en esta máquina.

EOPNOSUPPORT

El protocolo especificado no admite la creación de parejas de conectores.

EFAULT La dirección ``sv'' no especifica una parte válida del espacio de direcciones del proceso.

CONFORME A

4.4BSD (la llamada al sistema socketpair apareció en 4.2BSD). Generalmente transportable a o desde sistemas no BSD que admitan clónicos de la capa de conectores BSD (incluyendo variantes System V).

VÉASE TAMBIÉN

read(2), write(2), pipe(2)

BSD

3 febrero 1998

SOCKETPAIR(2)

Ahora nos dispondremos a leer la página de manual en línea:

man 2 socketcall

IPC(2)

Manual del Programador de Linux

IPC(2)

NOMBRE

socketcall - llamadas al sistema relativas a zócalos

SINOPSIS

int socketcall(int llamada, unsigned long *args);

DESCRIPCIÓN

socketcall es un punto de entrada al núcleo común para las llamadas al sistema relativas a zócalos.

El argumento ``llamada'' determina a qué función de zócalos llamar. El parámetro ``args'' apunta a un bloque que contiene los argumentos reales, que se pasan tal cual a la función apropiada.

Los programas de usuario deberían llamar a las funciones apropiadas por sus nombres usuales. Solamente los implementadores de la biblioteca estándar y los buenos programadores del núcleo necesitan conocer la existencia de socketcall.

CONFORME A

Esta llamada es específica de Linux, y no debería emplearse en programas pretendidamente transportables.

VÉASE TAMBIÉN

accept(2), bind(2), connect(2), getpeername(2), getsockname(2), getsockopt(2), listen(2), recv(2), recvfrom(2), send(2), sendto(2), setsockopt(2), shutdown(2), socket(2), socketpair(2)

Linux 1.2.4

17 Febrero 1998

IPC(2)

Continuamos con la página del manual:

man 2 getpeername

GETPEERNAME(2) Manual del Programador de Linux GETPEERNAME(2)

NOMBRE

getpeername - obtiene el nombre de la pareja conectada

SINOPSIS

#include <sys/socket.h>

int getpeername(int s, struct sockaddr *nombre, socklen_t *longinom);

DESCRIPCIÓN

getpeername devuelve el nombre de la pareja conectada al zócalo ``s''. El parámetro ``longinom'' debería inicializarse de forma que indicara la cantidad de espacio a la que apuntará nombre. Al regresar

la función, contendrá el tamaño real del nombre devuelto (en bytes). El nombre se trunca si el búfer provisto es demasiado pequeño.

VALOR DEVUELTO

En caso de éxito, se devuelve cero. En caso de error, se devuelve -1 y se pone en ``errno'' un valor apropiado.

ERRORES

EBADF El argumento ``s'' no es un descriptor válido.

ENOTSOCK

El argumento ``s'' es un fichero, no un zócalo.

ENOTCONN

El zócalo no está conectado.

ENOBUFS

No había en el sistema suficientes recursos como para efectuarse la operación.

EFAULT El parámetro ``nombre'' apunta a memoria que no está en una zona válida del espacio de direcciones del proceso.

CONFORME A

SVr4, 4.4BSD (la llamada al sistema getpeername apareció por vez 1^a en 4.2BSD).

NOTA

El tercer argumento de getpeername es en realidad un entero (y esto es lo que tienen BSD 4.* , libc4 y libc5). Cierta confusión en POSIX dio como resultado el actual **socklen_t**. El estándar propuesto todavía no ha sido adoptado pero glibc2 ya lo sigue y también tiene **socklen_t**. Vea también accept(2).

VÉASE TAMBIÉN

accept(2), bind(2), getsockname(2)

BSD

30 julio 1993

GETPEERNAME(2)

Ahora continuaremos con la página de manual:

man 2 sendfile

SENDFILE(2)

Manual del Programador de Linux

SENDFILE(2)

NOMBRE

sendfile - transfiere datos entre descriptores de fichero

SINOPSIS

```
#include <unistd.h>
```

```
ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count)
```

DESCRIPCIÓN

Esta llamada **copia datos entre un descriptor de fichero y otro**. Cualquiera de los dos **descriptores** de fichero o los dos **pueden referirse a un conector (socket)**. ``**in_fd**'' debe ser un **descriptor de fichero abierto para lectura** y ``**out_fd**'' debe ser un **descriptor abierto para escritura**. ``**offset**'' es un **puntero a una variable que contiene la posición del puntero del fichero de entrada desde la que sendfile(2) empezará a leer datos**. Cuando sendfile regrese, a esta variable se le asignará la posición del byte siguiente al último byte que se ha leído. **count** es la cantidad de bytes a copiar entre los descriptores de fichero.

NOTAS

sendfile no modifica el puntero de fichero actual de ``in_fd'', pero sí lo hace para ``out_fd''.

Si planea usar sendfile **para enviar ficheros a un conector TCP**, pero necesita **enviar algunos datos de cabecera delante de los contenidos** del fichero, por favor **vea la opción TCP_CORK en tcp(7)** para minimizar el **número de paquetes y ajustar el rendimiento**.

VALOR DEVUELTO

Si la transferencia ha tenido éxito, se devuelve el número de bytes escritos en ``out_fd''. En caso de error, se devuelve -1 y se asigna a ``errno'' un valor apropiado.

ERRORES

EBADF El fichero de entrada no ha sido abierto para lectura o el fichero de salida no ha sido abierto para escritura.

EINVAL Descriptor inválido o bloqueado.

ENOMEM No hay memoria suficiente para leer de ``in_fd''.

EIO Se ha producido un error indeterminado al leer de ``in_fd''.

VERSIONES

sendfile es una nueva característica de la versión 2.2 de Linux.

Otros Unix normalmente implementan sendfile con otras semánticas y prototipos. No debería usarse en programas transportables.

VÉASE TAMBIÉN
socket(2), open(2)

Página man de Linux

1 diciembre 1998

SENDFILE(2)

Nos disponemos a leer la página de manual: **man 2 readv**
Que es la misma que: **man 2 writev**

READV(2)

Manual del Programador Linux

READV(2)

NOMBRE

readv, writev - lee o escribe un vector

SINOPSIS

#include <sys/uio.h>

int readv(int fd, const struct iovec * vector, int count);

int writev(int fd, const struct iovec * vector, int count);

```
struct iovec {
    _ptr_t    iov_base;      /* Comienzo de Direcciones. */
    size_t     iov_len;       /* Longitud en bytes. */
};
```

DESCRIPCIÓN

readv lee datos desde el descriptor de fichero ``fd'' y pone los resultados en la zona de memoria descrita por ``vector''. El número de bloques de memoria se especifica en ``count''. Los bloques se llenan en el orden indicado. Funciona igual que **read** salvo que los datos son puestos en ``vector'' en lugar de en una zona contigua de memoria.

writev escribe datos al descriptor de fichero ``fd'' y desde la zona de memoria descrita por **vector**. El número de bloques de memoria se especifica en ``count''. Los bloques son usados en el orden indicado.

Funciona igual que **write** excepto que los datos son tomados desde **vector** en lugar de una zona contigua de memoria.

VALOR DEVUELTO

En caso de éxito **readv** devuelve el número de bytes leídos. En caso de éxito **writev** devuelve el número de bytes escritos. En caso de error, se devuelve **-1** y se asigna a ``errno'' un valor adecuado.

ERRORES

EINVAL Se ha dado un argumento inválido. Por ejemplo ``count'' podría ser mayor que **MAX_IOVEC** o cero. ``fd'' podría estar unido a un objeto inadecuado para lectura (para **readv**) o escritura (para **writev**).

EFAULT "Fallo de segmentación". Probablemente ``vector'' o alguno de los punteros ``iov_base'' apuntan a una zona de memoria que no está correctamente reservada.

EBADF El descriptor de fichero ``fd'' no es válido.

EINTR La llamada ha sido interrumpida por una señal antes de que algún dato fuese leído/escrito.

EAGAIN Se ha seleccionado E/S no bloqueante usando O_NONBLOCK y no había datos disponibles inmediatamente para ser leídos. (O el descriptor de fichero ``fd'' apunta a un objeto que está bloqueado.)

EISDIR ``fd'' hace referencia a un directorio.

EOPNOTSUP

``fd'' hace referencia a un socket o dispositivo que no soporta lectura/escritura.

ENOMEM No hay suficiente memoria del núcleo disponible.

Podrían ocurrir otros errores, dependiendo del objeto conectado a ``fd''.

CONFORME A

4.4BSD (las funciones **readv** y **writev** aparecieron por primera vez en BSD 4.2) y Unix98. La libc5 de Linux usa **size_t** como el tipo del parámetro ``count'', lo cual es lógico pero no estándar.

VÉASE TAMBIÉN

read(2), write(2), fprintf(3), fscanf(3)

Linux 2.2.0-pre8

20 enero 1999

READV(2)

Ahora continuamos con la lectura de la página de manual: **man 2 fcntl**
FCNTL(2) **Manual del Programador de Linux** **FCNTL(2)**

NOMBRE

fcntl - manipula el descriptor de fichero

SINOPSIS

```
#include <unistd.h>
#include <fcntl.h>
```

```
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock * lock);
```

DESCRIPCIÓN

fcntl realiza una de las diversas y variadas operaciones sobre ``fd''. La operación en cuestión se determina mediante ``cmd'':

F_DUPFD Busca el descriptor de fichero disponible de menor número mayor o igual que ``arg'' y lo convierte en una copia de ``fd''. Esto es diferente en **dup2(2)** que usa exactamente el descriptor especificado.

Los descriptores antiguo y nuevo pueden usarse indistintamente. Ambos comparten **candados (locks)**, indicadores de posición de ficheros y **banderas (flags)**; por ejemplo, si la posición del fichero se modifica usando **Iseek(2)** en uno de los descriptores, la posición del otro resulta modificada simultáneamente.

Sin embargo, los dos descriptores no comparten la **bandera close-on-exec "cerrar-al-ejecutar"**. La **bandera close-on-exec de la copia está desactivada**, significando que **no se cerrará en ejecución**.

En caso de éxito, se devuelve el nuevo descriptor.

F_GETFD Lee la bandera close-on-exec. Si el bit **FD_CLOEXEC** es 0, el **fichero permanecerá abierto durante exec(3)**, en caso contrario se cerrará el fichero.

F_SETFD Asigna el valor de la bandera close-on-exec al valor especificado por el bit **FD_CLOEXEC** de ``arg''.

F_GETFL Lee las banderas del descriptor (todas las banderas, según hayan sido asignadas por **open(2)**, serán devueltas).

F_SETFL Asigna las banderas del descriptor al valor asignado por ``arg''. **Sólo O_APPEND, O_NONBLOCK y O_ASYNC pueden asignarse; las otras banderas no se ven afectadas.**

Las banderas se comparten entre copias (hechas con **dup(2)**, **fork(2)**, etc.) del mismo descriptor de fichero.

Las banderas y su semántica están descritas en **open(2)**.

F_GETLK, F_SETLK y F_SETLKW se utilizan para gestionar **candados de ficheros discretionarios (discretionary file locks)**. El tercer argumento ``lock'' es un puntero a una **struct flock** (que puede ser sobrescrita por esta llamada).

F_GETLK

Devuelve la estructura **flock** que nos impide obtener el **candado**, o establece el campo ``l_type'' del candado a **F_UNLCK** si no hay obstrucción.

F_SETLK

El **candado está cerrado** (cuando ``I_type'' es F_RDLCK o F_WRLCK) o **abierto** (cuando es F_UNLCK).

Si el **candado** está **tomado por alguien más**, esta llamada **devuelve -1 y pone en ``errno'' el código de error EACCES o EAGAIN.**

F_SETLKW

Como F_SETLK, pero **en vez de devolver un error esperamos que el candado se abra**. Si se recibe una señal a capturar mientras fcntl está esperando, se interrumpe y (después de que el manejador de la señal haya terminado) regresa inmediatamente (devolviendo -1 y asignado a ``errno'' el valor EINTR).

F_GETOWN, F_SETOWN, F_GETSIG y F_SETSIG se utilizan para gestionar las señales de disponibilidad de E/S:

F_GETOWN

Obtiene el **ID de proceso o el grupo de procesos que actualmente recibe las señales SIGIO y SIGURG** para los eventos **sobre** el descriptor de fichero ``fd''.

Los **grupos de procesos se devuelven como valores negativos**.

F_SETOWN

Establece el **ID de proceso o el grupo de procesos que recibirá las señales SIGIO y SIGURG para los eventos sobre el descriptor de fichero ``fd''**.

Los **grupos de procesos** se especifican mediante **valores negativos**. (Se puede usar **F_SETSIG** para **especificar** una **señal diferente a SIGIO**).

Si activa la **bandera de estado O_ASYNC** sobre un descriptor de fichero (tanto si proporciona esta bandera con la llamada open(2) como si usa la orden **F_SETFL** de fcntl), **se enviará una señal SIGIO cuando sea posible la entrada o la salida sobre ese descriptor de fichero**.

El **proceso o el grupo de procesos que recibirá la señal se puede seleccionar usando** la orden **F_SETOWN** de la función fcntl. Si el descriptor de fichero **es un enchufe (socket)**, esto **también seleccionará al recipiente de las señales SIGURG que se entregan cuando llegan datos fuera de orden (out-of-band, OOB) sobre el enchufe**. (SIGURG se **envía** en cualquier **situación en la que select(2) informaría que el enchufe tiene una "condición excepcional"**). Si el descriptor de fichero **corresponde a un dispositivo de terminal**, entonces **las señales SIGIO se envían al grupo de procesos en primer plano de la terminal**.

F_GETSIG

Obtiene la señal enviada cuando la entrada o la salida son posibles. Un valor **cero** significa que se envía SIGIO. Cualquier **otro valor** (incluyendo SIGIO) es la señal enviada en su lugar y en este caso se dispone de información adicional para el manejador de señal si se instala con SA_SIGINFO.

F_SETSIG

Establece la señal enviada cuando la entrada o la salida son posibles. Un valor **cero** significa enviar la señal por defecto SIGIO. Cualquier **otro valor** (incluyendo SIGIO) es la señal a enviar en su lugar y en este caso se dispone de información adicional para el manejador de señal si se instala con SA_SIGINFO.

Usando F_SETSIG con un **valor distinto de cero** y asignando SA_SIGINFO para el manejador de señal (vea `sigaction(2)`), se pasa información extra sobre los eventos de E/S al manejador en la estructura `siginfo_t`. Si el campo ```si_code`'' indica que la fuente es SI_SIGIO, el campo ```si_fd`'' proporciona el descriptor de fichero asociado con el evento. En caso contrario, no se indican qué descriptores de ficheros hay pendientes y, para determinar qué descriptores de fichero están disponibles para E/S, debería usar los mecanismos usuales (select(2), poll(2), read(2) con O_NONBLOCK activo, etc.).

Seleccionando una **señal de tiempo real POSIX.1b** (**valor >= SIGRTMIN**), se pueden encolar varios eventos de E/S usando los mismos números de señal. (El encolamiento depende de la memoria disponible). Se dispone de información extra si se asigna SA_SIGINFO al manejador de señal, como antes.

Usando estos mecanismos, **un programa puede implementar E/S totalmente asíncrona, sin usar select(2) ni poll(2) la mayor parte del tiempo.**

El uso de O_ASYNC, F_GETOWN y F_SETOWN es específico de Linux y BSD. F_GETSIG y F_SETSIG son específicos de Linux. POSIX posee E/S asíncrona y la estructura aio_sigevent para conseguir cosas similares; estas también están disponibles en Linux como parte de la biblioteca de C de GNU (GNU C Library, Glibc).

VALOR DEVUELTO

Para una llamada con éxito, el valor devuelto depende de la operación:

F_DUPFD El nuevo descriptor.

F_GETFD Valor de la bandera.

F_GETFL Valor de las banderas.

F_GETOWN Valor del propietario del descriptor.

F_GETSIG Valor de la señal enviada cuando la lectura o la escritura son posibles o cero para el comportamiento tradicional con SIGIO.

Para cualquier otra orden
Cero.

En caso de error el valor devuelto es -1, y se pone un valor apropiado en ``errno''.

ERRORES

EACCES La operación está prohibida por candados mantenidos por otros procesos.

EAGAIN La operación está prohibida porque el fichero ha sido asociado a memoria por otro proceso.

EDEADLK Se ha detectado que el comando **F_SETLKW** especificado provocaría un interbloqueo.

EFAULT ``lock'' está fuera de su espacio de direcciones accesible.

EBADF ``fd'' no es un descriptor de fichero abierto.

EINTR El comando **F_SETLKW** ha sido interrumpido por una señal. Para **F_GETLK** y **F_SETLK**, la orden fue interrumpida por una señal antes de que el candado fuera comprobado o adquirido. Es más probable al poner un candado a un fichero remoto (por ejemplo, un candado sobre NFS) pero algunas veces puede ocurrir localmente.

EINVAL Para **F_DUPFD**, ``arg'' es negativo o mayor que el valor máximo permitido. Para **F_SETSIG**, ``arg'' no es un número de señal permitido.

EMFILE Para **F_DUPFD**, el proceso ya ha llegado al número máximo de descriptores de ficheros abiertos.

ENOLCK Demasiados candados de segmento abiertos, la tabla de candados está llena o ha fallado un protocolo de candados remoto (por ejemplo, un candado sobre NFS).

EPERM Se ha intentado limpiar la bandera **O_APPEND** sobre un fichero que tiene activo el atributo de 'sólo añadir' (append-only).

NOTAS

Los **errores devueltos por dup2 son distintos de aquéllos dados por F_DUPFD.**

CONFORME A

SVID, AT&T, POSIX, X/OPEN, BSD 4.3. Sólo las operaciones **F_DUPFD, F_GETFD, F_SETFD, F_GETFL, F_SETFL, F_GETLK, F_SETLK y F_SETLKW** se especifican en **POSIX.1**. **F_GETOWN y F_SETOWN** son **BSD-ismos no aceptados en SVr4**; **F_GETSIG y F_SETSIG** son específicos de Linux. Las banderas legales para **F_GETFL/F_SETFL** son aquéllas que acepta **open(2)** y varían entre estos sistemas; **O_APPEND, O_NONBLOCK, O_RDONLY y O_RDWR** son las que se mencionan en **POSIX.1**. SVr4 admite algunas **otras opciones y banderas** no documentadas aquí.

SVr4 documenta las **condiciones de error adicionales EIO, ENOLINK y EOVERRLOW**.

VÉASE TAMBIÉN

open(2), socket(2), dup2(2), flock(2).

Linux

12 julio 1999

FCNTL(2)

Seguimos con la página de manual:

man 2 poll

POLL(2)

Manual del Programador de Linux

POLL(2)

NOMBRE

poll - espera un evento en un descriptor de fichero

SINOPSIS

#include <sys/poll.h>

int poll(struct pollfd *ufds, unsigned int nfds, int timeout);

DESCRIPCIÓN

poll es una **variación de select**. Especifica un **vector de ``nfds'' estructuras del tipo**

```
struct pollfd {
    int fd;          /* Descriptor de fichero */
    short events;    /* Eventos solicitados */
    short revents;   /* Eventos ocurridos */
};
```

y un tiempo límite ``timeout'' en milisegundos. Un **valor negativo** indica un tiempo **infinito**. El campo **``fd''** contiene el descriptor de fichero de un fichero abierto. El campo **``events''** es un **parámetro de entrada**, una **máscara de bits** que **especifica los eventos en los que la aplicación** está **interesada**.

El campo ``**revents**'' es un **parámetro de salida, completado por el núcleo con los eventos que han ocurrido realmente**, bien del tipo solicitado **o bien** de uno de los tipos **POLLERR, POLLHUP o POLLNVAL**. (**Estos tres bits carecen de significado en el campo ``events'' y se pondrán a 1 en el campo ``revents'' en el momento en que la condición correspondiente sea cierta**). Si no se ha producido ninguno de los eventos solicitados (y ningún error) para ninguno de los descriptores de fichero, el **núcleo espera durante ``timeout'' milisegundos a que se produzca uno de estos eventos**. Los **bits posibles en estas máscaras están definidos en <sys/poll.h>**:

```
#define POLLIN 0x0001 /* Hay datos que leer */
#define POLLPRI 0x0002 /* Hay datos urgentes que leer */
#define POLLOUT 0x0004 /* La escritura ahora será no bloqueante */
#define POLLERR 0x0008 /* Condición de error */
#define POLLHUP 0x0010 /* Colgado */
#define POLLNVAL 0x0020 /* Petición inválida: fd no está abierto */
```

En **<asm/poll.h>** también se definen los valores **POLLRDNORM, POLLRDBAND, POLLWRNORM, POLLWRBAND** y **POLLMSG**.

VALOR DEVUELTO

En caso de éxito, se devuelve un número positivo que indica el número de estructuras cuyo campo ``**revents**'' tiene un valor distinto de cero (en otras palabras, aquellos descriptores para los que se ha producido un evento o un error). Un valor 0 indica que se ha cumplido el tiempo límite (timeout) de la llamada y que no se ha seleccionado ningún descriptor de fichero. En caso de error, se devuelve -1 y se asigna a ``**errno**'' un valor apropiado.

ERRORES

ENOMEM No hay espacio disponible para ubicar la tabla de descriptores del fichero.

EFAULT El vector pasado como argumento no está ubicado en el espacio de direcciones del programa invocador.

EINTR Se ha producido una señal antes de cualquier evento.

CONFORME A XPG4-UNIX.

DISPONIBILIDAD

La llamada al sistema **poll()** se introdujo en la versión 2.1.23 de Linux. La función de biblioteca **poll()** se introdujo en la versión 5.4.28 de libc (y emula la llamada al sistema poll mediante select si su núcleo no tiene dicha llamada al sistema).

**VÉASE TAMBIÉN
select(2)**

Linux 2.1.23

7 Diciembre 1997

POLL(2)

A continuación veremos la página de manual: **man 2 select**

SELECT(2) Manual del Programador de Linux SELECT(2)

NOMBRE

select, FD_CLR, FD_ISSET, FD_SET, FD_ZERO - multiplexación de E/S síncrona

SINOPSIS

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
```

```
int select( int n,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout);

FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

DESCRIPCIÓN

select espera a que una serie de descriptores de ficheros cambien su estado.

Se miran tres conjuntos independientes de descriptores.

Aquéllos listados en

``**readfds**'' serán **observados para ver si hay caracteres que llegan a estar disponibles para lectura,**

aquéllos en

``**writefds**'' serán **observados para ver si es correcto escribir inmediatamente en ellos,**

y aquéllos en

``**exceptfds**'' serán **observados para ver si ocurren excepciones.**

En caso de éxito, los conjuntos se modifican en marcha para indicar qué descriptores cambiaron realmente su estado.

Se proporcionan **cuatro macros para manipular los conjuntos.**

FD_ZERO	limpiará un conjunto.
FD_SET y FD_CLR	añaden o borran un descriptor dado a o de un conjunto.
FD_ISSET	mira a ver si un descriptor es parte del conjunto; esto es útil después de que select regrese.

``n'' es el descriptor con el número más alto en cualquiera de los tres conjuntos, más 1.

``timeout'' es un límite superior de la cantidad de tiempo transcurrida antes de que select regrese. Puede ser cero, causando que select(2) regrese inmediatamente. Si **``timeout''** es NULL (no hay tiempo de espera), select puede bloquear indefinidamente.

VALOR DEVUELTO

En caso de éxito, select(2) devuelve el número de descriptores contenidos en los conjuntos de descriptores, que puede ser cero si el tiempo de espera expira antes de que ocurra algo interesante. En caso de error, se devuelve -1, y se pone un valor apropiado en **``errno''**; los conjuntos y **``timeout''** estarán indefinidos, así que no confíe en sus contenidos tras un error.

ERRORES

EBADF Se ha dado un descriptor de fichero inválido en uno de los conjuntos.

EINTR Se ha capturado una señal no bloqueante.

EINVAL **``n''** es negativo.

ENOMEM select no ha sido capaz de reservar memoria para las tablas internas.

OBSERVACIONES

Hay algún código por ahí que llama a select con los tres conjuntos vacíos, **``n''** cero, y un **``timeout''** distinto de cero como una forma transportable y curiosa de dormir con una precisión por debajo del segundo.

En Linux, ``timeout'' se modifica para reflejar la cantidad de tiempo no dormido; la mayoría de otras implementaciones no hacen esto. Esto produce problemas cuando el código de Linux que lee **``timeout''** se transporta a otros sistemas operativos, y cuando se

transporta a Linux código que reutiliza una struct timeval para varias selects en un bucle sin reinicializarla. Considere que timeout está indefinido después de que select regrese.

EJEMPLO

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int
main(void)
{
    fd_set rfds;
    struct timeval tv;
    int valret;

/* Mirar stdin (descriptor de fichero 0) para ver si tiene entrada */
    FD_ZERO(&rfds);
    FD_SET(0, &rfds);

/* Esperar hasta 5 s */
    tv.tv_sec = 5;
    tv.tv_usec = 0;

    valret = select(1, &rfds, NULL, NULL, &tv);
    /* ¡No confiar ahora en el valor de tv! */

    if (valret)
        printf("Los datos ya están disponibles.\n");
        /* FD_ISSET(0, &rfds) será verdadero */
    else
        printf("Ningún dato en 5 segundos.\n");

    return(0);
}
```

CONFORME A

4.4BSD (la función `select(2)` apareció por primera vez en 4.2BSD). Generalmente es transportable a o desde sistemas no-BSD que admitan clones de la capa de zócalos de BSD (incluyendo variantes System V). Sin embargo, observe que la variante System V normalmente pone la variable de espera antes de salir, pero la variante BSD no.

VÉASE TAMBIÉN

`accept(2), connect(2), poll(2), read(2), recv(2), send(2), write(2)`

Linux 1.2 11 febrero 1996 SELECT(2)

Fijaremos nuestra atención en la página de manual: **man 2 flock**

FLOCK(2) Manual del Programador de Linux FLOCK(2)

NOMBRE

flock - impone o elimina un candado de recomendación en un fichero abierto.

SINOPSIS

#include <sys/file.h>

int flock(int fd, int operation)

DESCRIPCIÓN

Impone o elimina un candado de recomendación (advisory lock) en un fichero abierto. El fichero está **especificado por ``fd''**. Las operaciones válidas son:

LOCK_SH	Candado compartido. Más de un proceso puede tener un candado compartido para un fichero en un momento dado.
LOCK_EX	Candado exclusivo. Solamente un proceso puede tener un candado exclusivo para un fichero en un momento dado.
LOCK_UN	Desbloqueo.
LOCK_NB	No bloquear cuando se cierre el candado. Puede especificarse (mediante la operación OR) junto con otra de las operaciones.

Un fichero no puede tener simultáneamente candados compartido y exclusivo.

Es un fichero el que está encadenado (por ejemplo, el **nodo-i**), **no el descriptor de fichero.** Por tanto, **dup(2)** y **fork(2)** **no crean múltiples casos de un candado.**

VALOR DEVUELTO

En caso de éxito, cero. En caso de error, -1 , y se pone en ``errno'' un código apropiado.

ERRORES

EWOULDBLOCK

El fichero está encadenado y la bandera **LOCK_NB** ha sido elegida.

CONFORME A

4.4BSD (la llamada al sistema **flock(2)** apareció por primera vez en **4.2BSD**).

NOTAS

flock(2) no impone candados en ficheros sobre NFS. Use fcntl(2) en su lugar: funcionará sobre NFS, dada una versión suficientemente reciente de Linux y un servidor que soporte candados.

Los candados de flock(2) y fcntl(2) tienen semánticas diferentes con respecto a los procesos creados con fork(2) y con respecto a dup(2).

VÉASE TAMBIÉN

open(2), close(2), dup(2), execve(2), fcntl(2), fork(2). También están **locks.txt** y **mandatory.txt** en **/usr/src/linux/Documentation**.

Linux

11 Diciembre 1998

FLOCK(2)

Ahora fijaremos nuestra atención en la página de manual: **man 2 ioctl**

IOCTL(2)

Manual del Programador de Linux

IOCTL(2)

NOMBRE

ioctl - controlar dispositivo

SINOPSIS

#include <sys/ioctl.h>

int ioctl(int d, int peticion, ...)

[El "tercer" argumento es tradicionalmente **char *argp**, y así se le llamará de aquí en adelante.]

DESCRIPCIÓN

La función **ioctl** manipula los parámetros subyacentes de ficheros especiales. En particular, muchas características operacionales de los ficheros especiales de caracteres (verbigracia, es decir, por ejemplo las terminales) pueden controlarse con llamadas a ioctl. El argumento **``d'' debe ser un descriptor de fichero abierto.**

Una petición de ioctl tiene codificada en sí misma si el argumento es un parámetro de entrada o de salida, y el tamaño del argumento **``argp'' en bytes**. En el fichero de cabecera **<sys/ioctl.h>** se definen **macros** empleadas al especificar una petición de ioctl.

VALOR DEVUELTO

En caso de éxito, se devuelve cero. En caso de error, se devuelve -1 y se pone en ``errno'' un valor apropiado.

ERRORES

EBADF ``d'' no es un descriptor válido.

EFAULT ``argp'' referencia a una zona de memoria inaccesible.

ENOTTY ``d'' no está asociado con un dispositivo especial de caracteres.

ENOTTY La petición especificada no se aplica a la clase de objeto que referencia el descriptor ``d''.

EINVAL Petición o ``argp'' no es válido.

CONFORME A

Ningún estándar en particular. Los argumentos, valores devueltos y semántica de ioctl(2) varían según el controlador de dispositivo en cuestión (la llamada se usa como un recogedor para las operaciones que no encajen claramente en el modelo de flujos de E/S de Unix). Vea ioctl_list(2) para una lista de muchas de las llamadas conocidas a ioctl. La función ioctl apareció por primera vez en Unix de AT&T Versión 7.

VÉASE TAMBIÉN

execve(2), fcntl(2), mt(4), sd(4), tty(4)

BSD

22 octubre 1996

IOCTL(2)

A continuación veremos la página de manual:

man 2 bind

BIND(2)

Manual del Programador de Linux

BIND(2)

NOMBRE

bind - enlaza un nombre a un conector (socket)

SINOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

DESCRIPCIÓN

bind da al conector ``sockfd'' la dirección local ``my_addr''. ``my_addr'' tiene una longitud de ``addrlen bytes''. Tradicionalmente, esto se conoce como "asignar un nombre a un

conector". Cuando un conector se crea con socket(2), existe en un espacio de nombres (familia de direcciones) pero carece de nombre.

Normalmente, es necesario asignar una dirección local usando bind a un conector SOCK_STREAM (orientado a conexión – TCP) antes de que éste pueda recibir conexiones (vea accept(2)).

NOTAS

Las reglas usadas en el enlace de nombres varían entre familias de direcciones. Consulte las entradas de manual de la Sección 7 para obtener una información más detallada.

Para

AF_INET vea ip(7),

para

AF_UNIX vea unix(7),

para

AF_APPLETALK vea ddp(7),

para

AF_PACKET vea packet(7),

para

AF_X25 vea x25(7)

y para

AF_NETLINK vea netlink(7).

VALOR DEVUELTO

Se devuelve 0 en caso de éxito. En caso de error, se devuelve -1 y a ``errno" se le asigna un valor apropiado.

ERRORES

EBADF ``sockfd" no es un descriptor válido.

EINVAL El conector ya está enlazado a una dirección. Esto puede cambiar en el futuro: véase linux/unix/sock.c para más detalles.

EACCES La dirección está protegida y el usuario no es el superusuario.

Los siguientes errores son específicos a los conectores del dominio UNIX (AF_UNIX):

EINVAL La dirección ``addr_len" es incorrecta o el conector no pertenecía a la familia AF_UNIX.

EROFS El nodo-i del conector reside en un sistema de ficheros de 'sólo lectura'.

EFAULT ``my_addr" señala fuera del espacio de direcciones accesible por el usuario.

ENAMETOOLONG

``my_addr'' es demasiado larga.

ENOENT El fichero no existe.

ENOMEM No hay suficiente memoria disponible en el núcleo.

ENOTDIR

Un componente del camino no es un directorio.

EACCES El permiso de búsqueda ha sido denegado en uno de los componentes del camino.

ELOOP Se han encontrado demasiados enlaces simbólicos al resolver ``my_addr''.

FALLOS

No están descritas las opciones de proxy transparente.

CONFORME A

SVr4, 4.4BSD (la función **bind** apareció por primera vez en BSD 4.2).

SVr4 documenta condiciones generales de error adicionales:

EADDRNOTAVAIL, **EADDRINUSE** y **ENOSR**, y condiciones de error específicas del dominio UNIX adicionales: **EIO**, **EISDIR** y **EROFS**.

NOTA

El tercer argumento de **bind** es en realidad un entero (y esto es lo que tienen BSD 4.* , libc4 y libc5). Cierta confusión en POSIX dio como resultado el actual **socklen_t**. El estándar propuesto todavía no ha sido adoptado pero glibc2 ya lo sigue y tiene también **socklen_t**. Véa también **accept(2)**.

VÉASE TAMBIÉN

accept(2), **connect(2)**, **listen(2)**, **socket(2)**, **getsockname(2)**, **ip(7)**, **socket(7)**

Linux 2.2

3 octubre 1998

BIND(2)

Continuamos con la página de manual:

man 2 listen

LISTEN(2)

Manual del Programador de Linux

LISTEN(2)

NOMBRE

listen - espera conexiones en un conector (socket)

SINOPSIS

```
#include <sys/socket.h>
int listen(int s, int backlog);
```

DESCRIPCIÓN

Para aceptar conexiones,

- primero** se **crea un conector con socket(2)**,
- luego** se **especifica con listen(2) el deseo de aceptar conexiones entrantes y un límite de la cola para dichas conexiones**,
- y por último** las **conexiones son aceptadas mediante accept(2)**.

La llamada listen(2) se aplica solamente a conectores de tipo SOCK_STREAM o SOCK_SEQPACKET.

El parámetro ``backlog'' define la **longitud máxima** a la que puede llegar la **cola de conexiones pendientes**. Si una **petición de conexión llega estando la cola llena**, el **cliente puede recibir un error** con una indicación de **ECONNREFUSED** o, si el protocolo subyacente **acepta retransmisiones**, la **petición puede no ser tenida en cuenta, de forma que un reintento tenga éxito**.

NOTAS

El **comportamiento del parámetro ``backlog'' sobre conectores TCP ha cambiado con la versión 2.2 de Linux**. Ahora indica la **longitud de la cola para conectores establecidos completamente que esperan ser aceptados**, en lugar del número de peticiones de conexión incompletas. La **longitud máxima de la cola para conectores incompletos se puede configurar con la**

sysctl tcp_max_syn_backlog.

Cuando los "syncookies" están activos, no existe una longitud máxima lógica y la configuración de esta sysctl se ignora. Vea **tcp(7)** para más información.

VALOR DEVUELTO

En **caso de éxito, se devuelve cero**. En **caso de error, se devuelve -1 y se pone en ``errno'' un valor apropiado**.

ERRORES

EBADF El **argumento ``s'' no es un descriptor válido.**

ENOTSOCK

El argumento ``s'' no es un conector.

EOPNOTSUPP

El conector no es de un tipo que admite la operación listen(2).

CONFORME A

Single Unix, 4.4BSD, borrador POSIX 1003.1g. La llamada a función listen apareció por 1^a vez en 4.2BSD.

FALLOS

Si el conector es de tipo AF_INET y el argumento ``backlog'' es mayor que la constante SOMAXCONN (128 en 2.0 y 2.2), se trunca silenciosamente a SOMAXCONN. Para aplicaciones transportables, no confíe en este valor puesto que BSD (y algunos sistemas derivados de BSD) limitan ``backlog'' a 5.

VÉASE TAMBIÉN

accept(2), connect(2), socket(2)

BSD 22 octubre 1996 LISTEN(2)

Seguimos con la página de manual:

man 2 accept

ACCEPT(2) Manual del programador de Linux ACCEPT(2)

NOMBRE

accept - acepta una conexión sobre un conector (socket).

SINOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

DESCRIPCIÓN

La función accept se usa con conectores orientados a conexión (SOCK_STREAM, SOCK_SEQPACKET y SOCK_RDM).

Extrae la primera petición de conexión de la cola de conexiones pendientes,

le asocia un nuevo conector con, prácticamente, las misma propiedades que ``s'' y

reserva un nuevo descriptor de fichero para el conector, el cuál es el valor devuelto por la llamada.

El conector original ``s'' no se ve afectado por esta llamada. Dese cuenta que cualquier opción por descriptor de fichero (cualquiera que se pueda establecer con F_SETFL de fcntl(2), como ``no bloqueante'' o ``estado asincrónico'') no se hereda en un accept.

El argumento ``**s**'' es un conector que

ha sido creado con socket(2),

ligado a una dirección local con bind(2) y

que se encuentra a la escucha tras un listen(2).

El argumento ``**addr**'' es un **puntero a una estructura sockaddr**. Esta estructura **se rellena con la dirección de la entidad con la que se conecta, tal y como la conoce la capa de comunicaciones**. El **formato exacto de la dirección** pasada en el parámetro ``**addr**'' viene **determinado por la familia del conector** (vea **socket(2)** y las páginas de manual del protocolo correspondiente).

El argumento ``**addrlen**'' es un **parámetro de entrada/salida: al efectuar la llamada debe contener el tamaño de la estructura apuntada por ``addr''; a la salida, contendrá la longitud real (en bytes) de la dirección devuelta. Cuando ``addr'' es NULL nada se rellena.**

Si no hay conexiones pendientes en la cola y el conector no está marcado como "no bloqueante", accept(2) bloqueará al invocador hasta que se presente una conexión. Si el conector está marcado como no bloqueante y no hay conexiones pendientes en la cola, accept(2) devolverá EAGAIN.

Para ser informado de las conexiones entrantes que se produzca en un conector, puede usar select(2) o poll(2). Se producirá un evento de lectura en el intento de una nueva conexión y entonces puede llamar a accept(2) para obtener un conector para esa conexión. Alternativamente, puede configurar el conector para que provoque una señal SIGIO cuando se produzca actividad en el conector; vea socket(7) para más detalles.

Para determinados protocolos que necesitan una confirmación explícita, tales como DECNet, accept(2) se puede interpretar como una función que, simplemente, desencola la siguiente petición de conexión sin que ello implique la confirmación. Se sobreentiende la confirmación cuando se produce una lectura o escritura normal sobre el nuevo descriptor de fichero, y el rechazo puede ser de igual manera implícito cerrando el nuevo conector. Actualmente, sólo DECNet tiene esta semántica en Linux.

NOTAS

Puede que no siempre haya una conexión esperando después de que se produzca una señal SIGIO, o después de que select(2) o poll(2) devuelvan un evento de lectura, debido a que la conexión podría haber

sido eliminada por un error asíncrono de red u otro hilo antes de que se llame a `accept(2)`. Si esto ocurre entonces la llamada se bloqueará esperando a que llegue la siguiente conexión. Para asegurarse de que `accept(2)` nunca se bloquee, es necesario que el conector ``s'' pasado tenga la opción O_NONBLOCK activa (vea `socket(7)`).

VALOR DEVUELTO

La llamada devuelve -1 ante un error. Si tiene éxito, devuelve un entero no negativo que es el descriptor del conector aceptado.

MANEJO DE ERRORES

La llamada `accept(2)` de Linux pasa los errores de red ya pendientes sobre el nuevo conector como un código de error de `accept(2)`. Este comportamiento difiere de otras construcciones de conectores BSD.

Para un funcionamiento fiable, la aplicación debe detectar los errores de red definidos por el protocolo tras una llamada a `accept(2)` y tratarlos como EAGAIN reintentando la operación. En el caso de TCP/IP estos son ENETDOWN, EPROTO, ENOPROTOOPT, EHOSTDOWN, ENONET, EHOSTUNREACH, EOPNOTSUPP y ENETUNREACH.

ERRORES

EAGAIN o EWOULDBLOCK

El conector está marcado como no-bloqueante y no hay conexiones que aceptar.

EBADF El descriptor es inválido.

ENOTSOCK

El descriptor referencia a un fichero, no a un conector.

EOPNOTSUPP

El conector referenciado no es del tipo SOCK_STREAM.

EFAULT El parámetro ``addr'' no se encuentra en una zona accesible para escritura por el usuario.

EPERM Las reglas del cortafuegos prohiben la conexión.

ENOBUFS, ENOMEM

No hay suficiente memoria disponible. Esto normalmente significa que la asignación de memoria está limitada por los límites del buffer de conectores, no por la memoria del sistema, aunque esto no es 100% coherente.

Además, se pueden devolver otros errores de red para el nuevo conector y que se encuentren definidos en el protocolo. Diferentes núcleos de Linux pueden devolver otros errores diferentes como EMFILE, EINVAL, ENOSR, ENOBUFS, EPERM, ECONNABORTED,

ESOCKTNOSUPPORT, EPROTONOSUPPORT, ETIMEDOUT, ERESTARTSYS.

CONFORME A

SVr4, 4.4BSD (la función **accept(2)** apareció por primera vez en **BSD 4.2**). La página de manual de BSD documenta cinco posibles respuestas de error (EBADF, ENOTSOCK, EOPNOTSUPP, EWOULDBLOCK, EFAULT).

SUSv2 documenta los errores EAGAIN, EBADF, ECONNABORTED, EFAULT, EINTR, EINVAL, EMFILE, ENFILE, ENOBUFS, ENOMEM, ENOSR, ENOTSOCK, EOPNOTSUPP, EPROTO, EWOULDBLOCK.

NOTA

El tercer argumento de **accept(2)** se declaró originalmente como un `int *` (y así está en **libc4** y **libc5** y en otros muchos sistemas como **BSD 4.***, **SunOS 4**, **SGI**); el estándar propuesto **POSIX 1003.1g** quiso cambiarlo a `size_t *` y así está en **SunOS 5**. Más tarde, los borradores **POSIX** tenían `socklen_t *` y así lo tomaron the **Single Unix Specification** y **glibc2**.

Citando a Linus Torvalds:

“Cualquier biblioteca razonable debe hacer que "socklen_t" sea del mismo tamaño que int. Cualquier otra cosa destroza todo lo de la capa de conectores BSD. POSIX inicialmente estableció el tipo a size_t y, de hecho, yo (y es de suponer que otros aunque, obviamente, no demasiados) nos quejamos a gritos. El ser de tipo size_t es completamente desastroso, precisamente porque, por ejemplo, size_t muy rara vez es del mismo tamaño que "int" en arquitecturas de 64 bit. Y tiene que ser del mismo tamaño que "int" porque así está en la interfaz de conectores BSD. De cualquier modo, los de POSIX finalmente tuvieron una idea y crearon "socklen_t". Para empezar, no deberían haberlo tocado pero, una vez que lo hicieron, pensaron que debían tener un tipo con nombre propio por alguna insonable razón (probablemente alguien no quería desprestigiarse por haber cometido la estupidez original por lo que, simplemente, renombraron su metedura de pata de forma silenciosa)”.

VÉASE TAMBIÉN

bind(2), connect(2), listen(2), select(2), socket(2)

CONNECT(2) Manual del Programador de Linux CONNECT(2)

NOMBRE

connect - inicia una conexión en un conector (socket)

SINOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int connect(int sockfd,
            const struct sockaddr *serv_addr,
            socklen_t addrlen);
```

DESCRIPCIÓN

El descriptor de fichero ``sockfd'' debe referenciar a un conector.

Si el conector es del tipo

SOCK_DGRAM entonces la dirección ``serv_addr'' es la dirección a la que por defecto se envían los datagramas y la única dirección de la que se reciben datagramas.

Si el conector es del tipo

SOCK_STREAM o SOCK_SEQPACKET, esta llamada intenta hacer una conexión a otro conector. El otro conector está especificado por ``serv_addr'', la cual es una dirección (de longitud ``addrlen'') en el espacio de comunicaciones del conector. Cada espacio de comunicaciones interpreta el parámetro ``serv_addr'' a su manera.

Generalmente, los

conectores de protocolos orientados a conexión (TCP) pueden conectarse con éxito mediante connect(2) una vez solamente;

los conectores de protocolos no orientados a conexión pueden usar connect(2) múltiples veces para cambiar sus asociaciones. Los conectores de protocolos no orientados a conexión **pueden disolver la asociación conectándose a una dirección en la que al miembro ``sa_family'' de sockaddr se le ha asignado el valor AF_UNSPEC.**

VALOR DEVUELTO

Si conexión o enlace tiene éxito, se devuelve **0**. En caso de error, se devuelve **-1**, y se asigna a la variable ``**errno**'' un valor apropiado.

ERRORES

Los siguientes sólo son errores generales de conector. Puede haber otros códigos de error específicos del dominio.

EBADF El descriptor del fichero no es un índice válido de la tabla de descriptores.

EFAULT La estructura de dirección del conector está fuera del espacio de direcciones del usuario.

ENOTSOCK

El descriptor del fichero no está asociado con un conector.

EISCONN

El conector ya está conectado.

ECONNREFUSED

No hay nadie escuchando en la dirección remota.

ETIMEDOUT

Finalizó el plazo de tiempo mientras se intentaba la conexión. El servidor puede estar demasiado ocupado para aceptar nuevas conexiones. Dese cuenta que para conectores IP el plazo de tiempo puede ser muy largo cuando se han habilitado los "syncookies" en el servidor.

ENETUNREACH

Red inaccesible.

EADDRINUSE

La dirección local ya está en uso.

EINPROGRESS

El conector es no bloqueante y la conexión no puede completarse inmediatamente. Es posible usar select(2) o poll(2) para completarla seleccionando el conector para escritura. Después que select(2) indique que la escritura es posible, use getsockopt(2) para leer la opción SO_ERROR al nivel SOL_SOCKET para determinar si connect(2) se completó con éxito (BSO_ERROR será cero) o sin éxito (BSO_ERROR será uno de los códigos de error usuales listados aquí, explicando la razón del fallo).

EALREADY

El conector es no bloqueante y todavía no se ha terminado un intento de conexión anterior.

EAGAIN No hay más puertos locales libres o las entradas en la cache de enrutamiento son insuficientes.

Para **PF_INET** vea la

sysctl net.ipv4.ip_local_port_range

en **ip(7)** para ver cómo incrementar el número de puertos locales.

EAFNOSUPPORT

La dirección pasada no tiene la familia de direcciones correcta en su campo ``sa_family''.

EACCES, EPERM

El usuario **ha intentado conectarse a una dirección de difusión (broadcast) sin que el conector tenga activa la opción de difusión, o la petición de conexión ha fallado debido a una regla del cortafuegos local.**

CONFORME A

SVr4, 4.4BSD (la función **connect(2)** apareció por primera vez **en BSD 4.2**). **SVr4** documenta adicionalmente los códigos de error generales EADDRNOTAVAIL, EINVAL, EAFNOSUPPORT, EALREADY, EINTR, EPROTOTYPE y ENOSR. También documenta muchas condiciones de error adicionales que no se describen aquí.

NOTA

El **tercer argumento de connect es en realidad un entero (y esto es lo que tienen BSD 4.* , libc4 y libc5)**. Cierta confusión en POSIX dio como resultado el actual **socklen_t**. El estándar propuesto todavía no ha sido adoptado pero glibc2 ya lo sigue y también tiene **socklen_t**. Vea también **accept(2)**.

FALLOS (BUGS)

Desconectar un conector llamando a connect con una dirección AF_UNSPEC no se ha implementado todavía.

VÉASE TAMBIÉN

accept(2), bind(2), listen(2), socket(2), getsockname(2)

Linux 2.2

3 octubre 1998

CONNECT(2)

A continuación seguiremos leyendo la página de manual:

man 3 byteorder

BYTEORDER(3) Manual del Programador de Linux BYTEORDER(3)

NOMBRE

htonl, htons, ntohs, ntohl, ntohs - convierten valores cuyos bytes se encuentran en orden de host a valores cuyos bytes se encuentran en orden de red y viceversa

SINOPSIS

```
#include <netinet/in.h>
```

```
unsigned long int htonl(unsigned long int hostlong);
```

```
unsigned short int htons(unsigned short int hostshort);
```

```
unsigned long int ntohl(unsigned long int netlong);
```

```
unsigned short int ntohs(unsigned short int netshort);
```

DESCRIPCIÓN

La función **htonl()** convierte el entero largo **hostlong** desde el orden de bytes del host al de la red.

La función **htons()** convierte el entero corto **hostshort** desde el orden de bytes del host al de la red.

La función **ntohl()** convierte el entero largo **netlong** desde el orden de bytes de la red al del host.

La función **ntohs()** convierte el entero corto **netshort** desde el orden de bytes de la red al del host.

En los i80x86 en el orden de bytes del host está primero el byte menos significativo (LSB), mientras que el orden de bytes de la red, tal como se usa en Internet, tiene primero el byte más significativo (MSB).

**CONFORME A
BSD 4.3**

**VÉASE TAMBIÉN
gethostbyname(3), getservent(3)**

BSD

15 Abril 1993

BYTEORDER(3)

Ahora fijaremos nuestra atención en la página de manual:

man 3 inet

INET(3)

Manual del Programador de Linux

INET(3)

NOMBRE

**inet_aton, inet_addr, inet_network, inet_ntoa, inet_makeaddr,
inet_lnaof, inet_netof - Rutinas de manipulación de direcciones de
Internet**

SINOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton(const char *cp, struct in_addr *inp);

unsigned long int inet_addr(const char *cp);

unsigned long int inet_network(const char *cp);

char *inet_ntoa(struct in_addr in);

struct in_addr inet_makeaddr(int net, int host);

unsigned long int inet_lnaof(struct in_addr in);

unsigned long int inet_netof(struct in_addr in);
```

DESCRIPCIÓN

inet_aton() convierte la dirección de Internet ``cp'' desde la notación estándar de números y puntos a la representación binaria, y la guarda en la estructura a la que apunte ``inp''. **inet_aton** devuelve no-cero si la dirección es válida, cero si no.

La función **inet_addr()** convierte la dirección de Internet ``cp'' desde la notación de números y puntos a la de datos binarios en orden de bytes del ordenador local. Si la entrada no es válida, devuelve **INADDR_NONE** (usualmente **-1**). Ésta es una interfaz obsoleta a **inet_aton**, descrita anteriormente; es obsoleta porque **-1** es una dirección válida (**255.255.255.255**), e **inet_aton** proporciona una manera más clara para indicar que ha ocurrido un error.

La función **inet_network()** extrae el número de red en orden de bytes de red desde la dirección ``cp'' a la notación de números y puntos. Si la entrada es inválida, devuelve **-1**.

La función **inet_ntoa()** convierte la dirección de Internet ``in'' dada en orden de bytes de red a una cadena de caracteres en la notación estándar de números y puntos. La cadena se devuelve en un búfer reservado estáticamente, que será sobreescrito en siguientes llamadas.

La función **inet_makeaddr()** construye una dirección de Internet en orden de bytes de red combinando el número de red ``net'' con la dirección local ``host'' en la red ``net'', ambas en orden de bytes de ordenador local.

La función **inet_lnaof()** devuelve la parte del ordenador local de la dirección de Internet ``in''. La dirección de ordenador local se devuelve en orden de bytes de ordenador local.

La función **inet_netof()** devuelve la parte de número de red de la dirección de Internet ``in''. El número de red se devuelve en orden de bytes de ordenador local.

La estructura **in_addr**, empleada en **inet_ntoa()**, **inet_makeaddr()**, **inet_lnaof()** e **inet_netof()** se define en **<netinet/in.h>** como:

```
struct in_addr {  
    unsigned long int s_addr;  
}
```

Observe que

en el i80x86

el orden de bytes de ordenador es: primero el Byte Menos Significativo (LSB),

mientras que

el orden de bytes de red es, como se usa en la Internet, el Byte Más Significativo (MSB) primero.

**CONFORME A
BSD 4.3**

**VÉASE TAMBIÉN
gethostbyname(3), getnetent(3), hosts(5), networks(5)**

BSD

3 Septiembre 1995

INET(3)

A continuación proseguiremos con las notas de clase.

Clase 12-09-2006: Programa de chat

PAR CLIENTE-SERVIDOR TCP NO BLOQUEANTE (select(2)):

Haremos un **chat** usando el **PROTOCOLO TCP/IP** y el **SUBPROTOCOLO TCP**. Será un **PAR CLIENTE – SERVIDOR** que permitirá enviar/recibir texto entre los 2 procesos.

El problema es que **el papel del**

CLIENTE es **iniciar el comienzo de la sesión**

SERVIDOR es **esperar el comienzo de la sesión**

Pero

**UNA VEZ QUE LA SESIÓN SE ESTABLECIÓ
ESTO SE VUELVE SIMÉTRICO,**

otro

**PROBLEMA ES QUIÉN DE AMBOS HACE EL PRIMER ENVÍO
(EL OTRO DEBE RECIBIR).**

Esto

SE COMPLICA PUÉS LAS LECTURAS (read(2)) SON BLOQUEANTES

UN POSIBLE ESCENARIO:

CLIENTE

SERVIDOR

write(2)

read(2)

**bloquea hasta que el cliente
escriba,
no puede escribir hasta no
salir del read(2)**

Entonces LAS POSIBLES SOLUCIONES:

- a) “Muerto el perro se acabó la rabia!” Podemos usar **read(2) NO BLOQUEANTE**. Se puede hacer con **ioctl(2)** usando el atributo **O_NDELAY**, no se recomienda pues baja mucho la eficiencia.
- b) Un **read(2) ¿siempre es bloqueante?**
Pues, siempre que no haya nada que leer. Entonces **debemos hacer read(2) cuando haya algo que leer.** Algo así como:
if(“hay algo que leer”) read()

Clase 12-09-2006: Programa de chat

PAR CLIENTE-SERVIDOR TCP NO BLOQUEANTE (select(2)):

¿Cómo implementamos esto?

b.a) “conectando una bola de cristal al PL”, o

b.b) Usando select(2)

Apareció en BSD (AT&T usó poll(2)), permite quedar “en escucha” sobre varios canales de E/S a la vez.

Es una de las primitivas más complicadas y su prototipo es:

```
int select(      int maxfdpmas1,      /* descriptor de fichero con
                           el número más alto en
                           cualesquiera de los tres
                           conjuntos, incrementado en
                           uno */
                  fd_set * in,        /* conjunto de descriptores de
                           ficheros usados para lectura */
                  fd_set * out,       /* conjunto de descriptores de
                           ficheros usados para escritura */
                  fd_set * excepciones,  /* conjunto de
                           descriptores de
                           ficheros usados para
                           excepciones */
                  struct timeval * tv); /* Limite de tiempo, que
                           transcurre antes de que
                           select(2) regrese */
```

Tiene asociadas las siguientes macros para manipular los conjuntos de descriptores de ficheros:

FD_ZERO(fd_set *)	/* Limpia un conjunto */
FD_SET(int fd, fd_set *)	/* añade descriptor ``fd'' al conjunto */
FD_CLEAR(int fd, fd_set *)	/* borra descriptor ``fd'' del conjunto */
FD_ISSET(int fd, fd_set *)	/* verifica si descriptor ``fd'' está en el conjunto */

Ahora veamos cómo sería el código fuente del SERVIDOR TCP NO BLOQUEANTE:

Clase 12-09-2006: Programa de chat

PAR CLIENTE-SERVIDOR TCP NO BLOQUEANTE (select(2)):

```
/* Archivo: servidor-tcp-no-bloqueante-v2.c
 *      Abre un socket TCP y espera la conexión. Una vez establecida,
 *      verifica si existen paquetes de entrada por el socket ó por STDIN.
 *      Si se reciben paquetes por el socket, los envía a STDOUT.
 *      Si se reciben paquetes por STDIN, los envía por el socket.
 */
```

/* ARCHIVOS DE CABECERA */

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<sys/time.h>
#include<sys/select.h>
```

/* DEFINICIONES */

```
#define PORT 5000  
#define MAX(x,y) ((x)>(y) ? (x) : (y))  
#define SIZE 1024  
#define TIME 3600
```

/* SINONIMOS */

```
typedef struct sockaddr *sad;
```

/* FUNCIONES */

```
void error(char *s){  
    perror(s);  
    exit(-1);  
}
```

```
/* FUNCION PRINCIPAL MAIN */
```

```
int main(){
```

struct sockaddr_in sin,

char li

```
fd_set in, in_orig;
struct timeval tv;
```

```
if((sock=socket(PF
```

```
error("socket")
```

```
if((sock=socket(PF_INET, SOCK_STREAM,0))<0) //sock es el que escucha  
    error("socket"); //sock1 es el que acepta y recibe
```

Clase 12-09-2006: Programa de chat

PAR CLIENTE-SERVIDOR TCP NO BLOQUEANTE (select(2)):

```
/* Continua: Archivo: servidor-tcp-no-bloqueante-v2.c */
sin.sin_family=AF_INET; // Familia de direcciones de sock
sin.sin_port=htons(PORT); // Puerto, con bytes en orden de red, para sock
sin.sin_addr.s_addr=INADDR_ANY; //dirección de internet, con bytes en
                                // orden de red, para sock

if(bind(sock, (sad)&sin, sizeof sin)<0) //publica dirección sin de sock
    error("bind");
if(listen(sock,5)<0) //pone en escucha a sock, 5 es tamaño de cola espera
    error("listen");

largo=sizeof sin1;

if((sock1=accept(sock,(sad)&sin1,&largo))<0) //por sock1 acepta
                                                //conexiones TCP
    error("accept");                      //desde dirección remota sin1, mediante
                                            //dirección local sin de sock
// lo que se lea de STDIN mediante dirección sin de sock, se escribirá en sock1
// lo que se lea por sock1 desde dirección remota sin1, se escribirá en STDOUT

/*tiene select*/
FD_ZERO(&in_orig); //Limpia el conjunto de descriptores de ficheros in_orig
FD_SET(0, &in_orig); //añade STDIN al conjunto in_orig
FD_SET(sock1, &in_orig); //añade sock1 al conjunto in_orig

/*tiene 1 hora*/
tv.tv_sec=TIME; //tiempo hasta que select(2) retorne: 3600 segundos
tv.tv_usec=0;
for(;){
    memcpy(&in, &in_orig, sizeof in); // copia conjunto in_orig en in
    if((cto=select(MAX(0,sock1)+1,&in,NULL,NULL,&tv))<0) // observa
        error("select");      // si hay algo para leer en el conjunto in
    if(cto==0)    //si tiempo de espera de select(2) termina ==> error
        error("timeout");

    /* averiguamos donde hay algo para leer*/
    if(FD_ISSET(0,&in)){ //si hay para leer desde STDIN

        fgets(linea, SIZE, stdin); // lee hasta 1024 caracteres de STDIN,
                                    //los pone en linea

        if( write(sock1, linea, strlen(linea)) < 0 ) // escribe contenido de
                                                    //linea en sock1
            error("write");
    }
}
```

**Clase 12-09-2006: Programa de chat
PAR CLIENTE-SERVIDOR TCP NO BLOQUEANTE (select(2)):**

```
/* Continua: Archivo: servidor-tcp-no-bloqueante-v2.c */

if(FD_ISSET(sock1,&in)){ // si hay para leer desde sock1 ``remoto``
    if( (cto=read(sock1,linea,1024)) < 0 ) // lee hasta 1024
        //caracteres de
        error("read"); // sock1, los pone en linea
    else if( cto == 0 ) // si lectura devuelve 0 ==> parar ejecucion
        break;

    linea[cto]=0; // marcar fin del buffer con ``0``

    /* Imprime en pantalla dirección de internet del cliente
       desde donde vienen datos */
    printf("\nDe la direccion[ %s ] : puerto[ %d ] --- llega el mensaje:\n",
           inet_ntoa(sin1.sin_addr),
           ntohs(sin1.sin_port));
    printf("%s \n",linea);
}
}

close(sock1);
close(sock);
return 0;
}
/* Fin Archivo: servidor-tcp-no-bloqueante-v2.c */
```

Que se compila mediante:

```
gcc -Wall      servidor-tcp-no-bloqueante-v2.c
      -o          servidor-tcp-no-bloqueante-v2.out
```

Y se ejecuta mediante:

```
./servidor-tcp-no-bloqueante-v2.out
```

Ahora veamos como sería el código fuente del
CLIENTE TCP NO BLOQUEANTE:

```
/* Archivo: cliente-tcp-no-bloqueante-v2.c
 * Abre un socket TCP e inicia una conexión TCP hacia un servidor
 * TCP remoto. Una vez conectado, verifica si existen paquetes de
 * entrada por el socket ó por STDIN.
 * Si se reciben paquetes por el socket, los envía a STDOUT.
 * Si se reciben paquetes por STDIN, los envía por el socket.
 **/
```

**Clase 12-09-2006: Programa de chat
PAR CLIENTE-SERVIDOR TCP NO BLOQUEANTE (select(2)):**

```
/* Continua: cliente-tcp-no-bloqueante-v2.c */

/* ARCHIVOS DE CABECERA */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<sys/time.h>
#include<sys/select.h>
#include<string.h>

/* DEFINICIONES */
#define PORT 5000
#define MAX(x,y) ((x)>(y) ? (x) : (y))
#define SIZE 1024
#define TIME 3600

/* SINONIMOS */
typedef struct sockaddr *sad;

/* FUNCIONES */
void error(char *s){
    perror(s);
    exit(-1);
}

/* FUNCION PRINCIPAL MAIN */
int main(int argc, char **argv){
    int sock, cto; //por sock va a iniciar conexion TCP
                    //a servidor TCP remoto

    if(argc < 2){
        fprintf(stderr,"usa: %s ipaddr \n", argv[0]);
        exit(-1);
    }

    struct sockaddr_in sin; // direccion de internet para sock
    char linea[SIZE]; // ``buffer'' de 1024 caracteres (un arreglo)
    fd_set in, in_orig; // conjuntos de descriptores de ficheros
    struct timeval tv; // tiempo limite que debe transcurrir antes que
                      // select(2) retorne
```

Clase 12-09-2006: Programa de chat

PAR CLIENTE-SERVIDOR TCP NO BLOQUEANTE (select(2)):

```
/* Continua: cliente-tcp-no-bloqueante-v2.c */
if((sock = socket(PF_INET, SOCK_STREAM,0))<0) // abre sock
    error("socket");

/* Continua: cliente-tcp-no-bloqueante-v2.c */
sin.sin_family=AF_INET;    // familia de direcciones a la que pertenece sin,
                           // para sock
sin.sin_port=htons(PORT); // puerto de sin, para sock

inet_aton(argv[1],&sin.sin_addr);
//direccion de internet de sin, para sock (del servidor remoto)
/* transforma argv[1] (la direccion IP pasada como argumento al programa)
 * desde formato ``ascii'' (puntero a cadena de caracteres)
 * al formato ``network'' (guarda en la ``struct in_addr`` ``sin.sin_addr`` )
 * que a su vez es miembro de la ``struct sockaddr_in`` ``sin``.
 * Se trata de la direccion IP del servidor al que se quiere conectar.
 */
if( connect(sock, (sad)&sin, sizeof sin)<0) //sock inicia la conexion TCP
    error("conect");                      // hacia direccion remota sin.sin_addr

/*tiene select*/
FD_ZERO(&in_orig); //Limpia el conjunto de descriptores de fichero ``in_orig``
FD_SET(0, &in_orig); // añade al conjunto ``in_orig`` el descriptor STDIN
FD_SET(sock,&in_orig); // añade al conjunto ``in_orig`` el descriptor sock

/*tiene 1 hora*/
tv.tv_sec=TIME;//tiempo disponible hasta que select(2) regrese: 3600 segs.
tv.tv_usec=0;

for(;;){
    memcpy(&in, &in_orig, sizeof in); //copia conjunto in_orig en in
    if((cto = select(MAX(0,sock)+1,&in,NULL,NULL,&tv))<0) //observa si hay
        error("select");                                //algo en el conjunto in
    if(cto==0)    //si el tiempo de espera para que select(2) retorne expira
        error("timeout");    // error

    /* averiguamos donde hay algo para leer*/
    if(FD_ISSET(0,&in)){ //si hay para leer desde STDIN
        fgets(linea, SIZE, stdin); //lee hasta 1024 caracteres de STDIN y los
                                   //pone en linea
        write(sock, linea, strlen(linea)); //escribir contenido de linea en sock
    }
}
```

**Clase 12-09-2006: Programa de chat
PAR CLIENTE-SERVIDOR TCP NO BLOQUEANTE (select(2)):**

```
/* Continua: cliente-tcp-no-bloqueante-v2.c */
if(FD_ISSET(sock,&in)){ //Si hay para leer desde sock
    cto = read(sock, linea, SIZE); //lee hasta 1024 caracteres de sock y los
                                    //pone en linea
    if(cto < 0)
        error("read"); //si lectura devuelve < 0 ==> error al leer
    else if(cto==0) break;//si lectura devuelve 0 ==> parar la ejecucion

    linea[cto]=0; //marcar el final del buffer con ``0``

/* Imprime en pantalla la direccion del servidor desde donde vienen datos */

    printf("\nDe la direccion[ %s ] : puerto [ %d ] --- llega el mensaje:\n",
           inet_ntoa(sin.sin_addr),
           ntohs(sin.sin_port));
    //Imprime en pantalla lo que vino desde sock1 mediante buffer linea */
    printf("%s \n",linea);
}
}
close(sock);
return 0;
}

/* Fin Archivo: cliente-tcp-no-bloqueante-v2.c */
```

Que **se compila** mediante:

```
gcc -Wall     cliente-tcp-no-bloqueante-v2.c
      -o       cliente-tcp-no-bloqueante-v2.out
```

Y **se ejecuta** de acuerdo a la siguiente sintaxis:

```
./cliente-tcp-no-bloqueante-v2.out IPADDRESS
```

Por ejemplo puede realizarse una prueba referenciando a la dirección IP de la propia computadora, invocándolo de la siguiente forma:

```
./cliente-tcp-no-bloqueante-v2.out 127.0.0.1
```

Para comprender mejor ambos códigos fuente, necesitamos ver algunas páginas del manual en línea de comandos.

Veamos entonces la página de manual:

```
man 3 inet_ntop
```

inet_ntop(3)

Linux Programmer's Manual

inet_ntop(3)

NOMBRE

inet_ntop – Analiza sintácticamente las estructuras de una dirección de red

SINTAXIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
```

```
const char *inet_ntop(
    int af,
    const void *src,
    char *dst,
    socklen_t cnt
);
```

DESCRIPCIÓN

Esta función **convierte la estructura de una dirección de red ``src'' que pertenece a la familia de direcciones ``af'' en una cadena de caracteres**, la cuál es **copiada a un buffer** de caracteres ``dst'' que posee ``cnt'' bytes de longitud.

inet_ntop(3) extiende a la función inet_ntoa(3) para dar soporte a múltiples familias de direcciones, inet_ntoa(3) ahora se considera obsoleta en favor de **inet_ntop(3)**. Las siguientes familias de direcciones son actualmente soportadas:

AF_INET

``src'' apunta a una struct **in_addr** (con formato de bytes en orden de red) que es convertida a una dirección de red IPv4 en el formato de notación numérica con puntos, "ddd.ddd.ddd.ddd". El buffer ``dst'' debe tener al menos **INET_ADDRSTRLEN** bytes de largo.

AF_INET6

``src'' apunta a una struct **in6_addr** (con formato de bytes en orden de red) que es convertida a una representación en el mas apropiado formato de dirección de red **IPv6** para esta dirección. El buffer ``dst'' debe tener al menos **INET6_ADDRSTRLEN** bytes de longitud.

VALOR DEVUELTO

inet_ntop devuelve un puntero a ``dst'' no nulo. Se devuelve **NULL** si ocurrió un error, seteando la variable **errno** con **EAFNOSUPPORT** si ``af'' no es una familia de direcciones válida, o asignando a **errno ENOSPC** si la cadena de caracteres convertida excediera el tamaño de ``dst'' dado por el argumento ``cnt''.

CONFORME A

POSIX 1003.1-2001. Note que el **RFC 2553 define** un **prototipo donde** el último **parámetro ``cnt`` es del tipo size_t**. Muchos sistemas siguen el RFC 2553. **Glibc 2.0 y 2.1 tienen size_t pero la versión 2.2 tiene socklen_t.**

VÉASE TAMBIÉN
inet_pton(3)

FALLOS

AF_INET6 convierte las direcciones IPv4 mapeadas a IPv6 en un formato IPv6.

Página Man de Linux

2000-12-18

inet_ntop(3)

También veamos la página de manual:

man 3 inet_pton

inet_pton(3)

Linux Programmer's Manual

inet_pton(3)

NOMBRE

inet_pton – Crea una estructura de dirección de red a partir de una cadena de caracteres

SINTAXIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
```

```
int inet_pton(int af, const char *src, void *dst);
```

DESCRIPCIÓN

Esta función **convierte la cadena de caracteres ``src`` en una estructura de dirección de red perteneciente a la familia de direcciones ``af``**, entonces **copia esta estructura a ``dst``**.

inet_pton(3) extiende a la función **inet_addr(3)** para dar soporte a múltiples familias de direcciones, **inet_addr(3)** ahora se considera **obsoleta en favor de inet_pton(3)**. Las siguientes familias de direcciones están soportadas actualmente:

AF_INET

``src`` apunta a una cadena de caracteres que contiene una dirección IPv4 en el formato de notación numérica con puntos, "ddd.ddd.ddd.ddd". La dirección es convertida a una struct **in_addr** y copiada a ``dst``, que debe tener sizeof(struct **in_addr**) bytes de longitud.

AF_INET6

``src'' apunta a una cadena de caracteres que contiene una dirección de red IPv6 en algún formato de dirección IPv6 permitido. La dirección es convertida a una struct in6_addr y copiada a ``dst'', que debe tener sizeof(struct in6_addr) bytes de longitud.

Ciertos formatos hexadecimales y octales heredados de **AF_INET** no están soportados por inet_pton, quien los rechaza.

VALOR DEVUELTO

inet_pton devuelve un **valor negativo** y pone **EAFNOSUPPORT** en **errno** si ``af'' no contiene una familia de direcciones válida. Devuelve 0 si ``src'' no contiene una cadena de caracteres que represente una dirección de red válida en la familia de direcciones especificada. Devuelve un **valor positivo** si la dirección de red fue convertida satisfactoriamente.

VÉASE TAMBIÉN
inet_ntop(3)

FALLOS

AF_INET6 no reconoce direcciones IPv4. Un mapeo explícito de una dirección IPv4 a una dirección IPv6 debe suministrarse en ``src'' en su lugar.

Página del Manual de Linux

2000-12-18

inet_pton(3)

Otra página del manual que debemos observar es :

man 3 fgets

GETS(3)

Manual del Programador de Linux

GETS(3)

NOMBRE

fgetc, fgets, getc, getchar, gets, ungetc - entrada de caracteres y cadenas de ellos

SINOPSIS

#include <stdio.h>

```
int fgetc(FILE *flujo);
char *fgets(char *s, int tam, FILE *flujo);
int getc(FILE *flujo);
int getchar(void);
char *gets(char *s);
int ungetc(int c, FILE *flujo);
```

DESCRIPCIÓN

fgetc() lee el siguiente carácter de ``flujo'' y lo devuelve como un **unsigned char modelado a un int**, o **EOF** al llegar al final del flujo o en caso de error.

getc() es equivalente a **fgetc()** excepto en el hecho de que puede estar implementado como una macro que evalúe flujo más de una vez.

getchar() es equivalente a **getc(stdin)**.

gets() lee una línea de stdin y la guarda en el búfer al que apunta ``s'' hasta que se encuentre bien un carácter terminador **nueva-línea**, bien **EOF**, al cual reemplaza con '\0'. No se hace ninguna comprobación de desbordamiento del búfer (vea FALLOS más abajo).

fgets() lee como mucho **uno menos de ``tam''** caracteres del ``flujo'' y los guarda en el búfer al que apunte ``s''. La lectura se para tras un **EOF** o una **nueva-línea**. Si se lee una **nueva-línea**, se guarda en el búfer. Tras el último carácter en el búfer se guarda un '\0'.

ungetc() pone ``c'' de vuelta en ``flujo'', **modelado a unsigned char**, donde queda disponible para una posterior operación de lectura.

Los caracteres puestos en el ``flujo'' serán devueltos en orden inverso; sólo se garantiza que se pueda devolver al flujo un carácter.

Las llamadas a las funciones descritas aquí pueden mezclarse unas con otras y con llamadas a otras funciones de entrada de la biblioteca **stdio** para el mismo flujo de entrada.

VALOR DEVUELTO

fgetc(), **getc()** y **getchar()** devuelven el carácter leído como un **unsigned char modelado a un int** o **EOF** al llegar al final de la entrada o en caso de error.

gets() y **fgets()** devuelven ``s'' en caso de éxito, y **NULL** en caso de error o cuando se llegue al final del fichero mientras que no se haya leído ningún carácter.

ungetc() devuelve ``c'' en caso de éxito, o **EOF** en caso de error.

CONFORME A

ANSI - C, POSIX.1

FALLOS

Puesto que es imposible saber, sin conocer de antemano los datos, cuántos caracteres va a leer **gets()**, y puesto que **gets()** continuará guardando caracteres una vez alcanzado el final del búfer, su empleo es extremadamente peligroso. Muchas veces ha sido utilizado

para comprometer la seguridad de un sistema. **En su lugar emplee fgets() siempre que pueda.**

No es recomendable mezclar llamadas a las funciones de entrada de la biblioteca stdio con llamadas de bajo nivel a read() para el descriptor de fichero asociado con el flujo de entrada; los resultados serán indefinidos y probablemente no los deseados.

VÉASE TAMBIÉN

read(2), write(2), fopen(3), fread(3), scanf(3), puts(3), fseek(3), ferror(3)

GNU

3 Febrero 1998

GETS(3)

**Clase 03-10-2006: PAR CLIENTE SERVIDOR TCP NO BLOQUEANTE:
OPCIONES DE LOS SOCKETS:**

La implementación del protocolo TCP/IP permite setear/resetear opciones de los sockets mediante:

```
int setsockopt(  
    int sock,           // conector al que se aplica la opción  
    int level,          // nivel en que se aplica la opción  
    int option,         // opción  
    void * value,  
    size_t s_value  
);
```

Ver (más arriba) **man 2 setsockopt** para más detalles.

Veremos un ejemplo de aplicación de opciones a los sockets. Agregaremos una modificación a los código fuente del **SERVIDOR TCP NO BLOQUEANTE**. **Permitiremos que más de un proceso cliente pueda usar el mismo puerto del servidor simultáneamente.**

De esta manera, **ya no se observará el mensaje de error:**

``Address already in use''

Sólo necesitamos agregar al código del servidor TCP NO BLOQUEANTE: una variable:

const int yes = 1;

la función setsockopt(2), aplicada antes de la función bind(2):

setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof yes)

Significa que al conector ``**sock**'' se le aplica la opción ``**SO_REUSEADDR**'' perteneciente al nivel de protocolo ``**SOL_SOCKET**'' con el valor de ``**yes**''.

**Clase 03-10-2006: PAR CLIENTE SERVIDOR TCP NO BLOQUEANTE:
OPCIONES DE LOS SOCKETS:**

Veamos entonces como queda el código fuente del
SERVIDOR TCP NO BLOQUEANTE CON PUERTO REUTILIZABLE:

```
/* Archivo: servidor-tcp-no-bloqueante-con-puerto-reutilizable.c
 * Abre un socket TCP y espera la conexión.
 * Con setsockopt(2) aplica al conector ``sock'' la opción
 * ``SO_REUSEADDR'' del nivel ``SOL_SOCKET'' con el valor
 * de la variable ``yes'' para permitir que más de una instancia
 * del proceso cliente pueda conectarse al mismo puerto en forma
 * simultánea.
 * La comunicación bidireccional sólo se logra entre un par
 * cliente servidor, el resto de los clientes pueden conectarse
 * pero no podrán enviar ni recibir mensajes.
 * Una vez establecida la conexión, verifica si existen paquetes
 * de entrada por el socket ó por STDIN.
 * Si se reciben paquetes por el socket, los envía a STDOUT.
 * Si se reciben paquetes por STDIN, los envía por el socket.
 */
```

/* ARCHIVOS DE CABECERA */

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<sys/time.h>
#include<sys/select.h>
```

/* DEFINICIONES */

```
#define PORT 5000
#define MAX(x,y) ((x)>(y) ? (x) : (y))
#define SIZE 1024
#define TIME 3600
```

/* SINONIMOS */

```
typedef struct sockaddr *sad;
```

/* FUNCIONES */

```
void error(char *s){
    perror(s);
    exit(-1);
}
```

**Clase 03-10-2006: PAR CLIENTE SERVIDOR TCP NO BLOQUEANTE:
OPCIONES DE LOS SOCKETS:**

```
/* Continua: servidor-tcp-no-bloqueante-con-puerto-reutilizable.c */

/* FUNCION PRINCIPAL MAIN */
int main(){
    const int yes = 1; //para setsockopt: SO_REUSEADDR
    int sock, sock1, cto, largo;
    struct sockaddr_in sin, sin1;
    char linea[SIZE];
    fd_set in, in_orig;
    struct timeval tv;

    if((sock=socket(PF_INET, SOCK_STREAM,0))<0) //sock es el que escucha
        error("socket");                      //sock1 es el que acepta y recibe

    sin.sin_family=AF_INET; // Familia de direcciones de sock
    sin.sin_port=htons(PORT); // Puerto, con bytes en orden de red, para sock
    sin.sin_addr.s_addr=INADDR_ANY; //dirección de internet, con bytes en
                                    // orden de red, para sock

    /* Aplica al conector ``sock`` la opción ``SO_REUSEADDR``
     * del nivel ``SOL_SOCKET`` con el valor de ``yes``
     */
    if(setsockopt(sock,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof yes) < 0)
        error("setsockopt: sock");

    if(bind(sock, (sad)&sin, sizeof sin) < 0) //publica dirección sin de sock
        error("bind");
    if(listen(sock, 5) < 0 ) //pone en escucha a sock, 5 es tamaño de cola
    espera
        error("listen");

    largo = sizeof sin1;

    /*por ``sock1`` acepta conexiones TCP desde dirección remota ``sin1``,
     * mediante dirección local ``sin`` de ``sock``
     */
    if((sock1 = accept(sock, (sad)&sin1, &largo)) < 0)
        error("accept");

    //lo que se lea de STDIN mediante dirección sin de sock, se escribirá en sock1
    // lo que se lea por sock1 desde dirección remota sin1, se escribirá en STDOUT
```

**Clase 03-10-2006: PAR CLIENTE SERVIDOR TCP NO BLOQUEANTE:
OPCIONES DE LOS SOCKETS:**

/* Continua: **servidor-tcp-no-bloqueante-con-puerto-reutilizable.c** */

```
/*tiene select*/
FD_ZERO(&in_orig); //Limpia el conjunto de descriptores de ficheros in_orig
FD_SET(0, &in_orig); //añade STDIN al conjunto in_orig
FD_SET(sock1, &in_orig); //añade sock1 al conjunto in_orig

/*tiene 1 hora*/
tv.tv_sec = TIME; //tiempo hasta que select(2) retorne: 3600 segundos
tv.tv_usec=0;

for(;;{
    memcpy(&in, &in_orig, sizeof in); // copia conjunto in_orig en in
    if((cto=select(MAX(0,sock1)+1,&in,NULL,NULL,&tv))<0) // observa
        error("select"); // si hay algo para leer en el conjunto in
    if(cto==0) //si tiempo de espera de select(2) termina ==> error
        error("timeout");

    /* averiguamos donde hay algo para leer*/
    if(FD_ISSET(0,&in)){ //si hay para leer desde STDIN

        fgets(linea, SIZE, stdin); // lee hasta 1024 caracteres de STDIN,
                                // los pone en linea

        if( write(sock1, linea, strlen(linea)) < 0 ) // escribe contenido de
            // linea en sock1
            error("write");
    }

    if(FD_ISSET(sock1,&in)){ // si hay para leer desde sock1 ``remoto``
        if( (cto=read(sock1,linea,1024)) < 0 ) // lee hasta 1024
            // caracteres de
            error("read"); // sock1, los pone en linea
        else if( cto == 0 ) // si lectura devuelve 0 ==> parar ejecucion
            break;
        linea[cto]=0; // marcar fin del buffer con ``0``

        /* Imprime en pantalla dirección de internet del cliente desde
         donde vienen datos */
        printf("\nDe la direccion[ %s ] : puerto[ %d ] --- llega el mensaje:\n",
               inet_ntoa(sin1.sin_addr),
               ntohs(sin1.sin_port));
        printf("%s \n",linea);
    }
}
```

**Clase 03-10-2006: PAR CLIENTE SERVIDOR TCP NO BLOQUEANTE:
OPCIONES DE LOS SOCKETS:**

/* Continua: **servidor-tcp-no-bloqueante-con-puerto-reutilizable.c** */

```
close(sock1);
close(sock);
return 0;
}
/* Fin Archivo: servidor-tcp-no-bloqueante-con-puerto-reutilizable.c */
```

Se compila mediante:

```
gcc -Wall      servidor-tcp-no-bloqueante-con-puerto-reutilizable.c
-o           servidor-tcp-no-bloqueante-con-puerto-reutilizable.out
```

Se ejecuta mediante:

servidor-tcp-no-bloqueante-con-puerto-reutilizable.out

Puede, por ejemplo, invocar a 3 instancias de los procesos clientes vistos con anterioridad:

- cliente 1: **./cliente-tcp-no-bloqueante-v2.out 127.0.0.1**
- cliente 2: **./cliente-tcp-no-bloqueante-v2.out 127.0.0.1**
- cliente 3: **./cliente-tcp-no-bloqueante-v2.out 127.0.0.1**

Los 3 procesos clientes se han podido conectar evitando el mensaje “Address already in use”,

pero sólo el primer proceso cliente logra establecer una sesión y una comunicación bidireccional hacia el proceso servidor.

Los procesos clientes restantes no pueden recibir ni enviar mensajes.

Nos dispondremos a reimplementar ahora estos códigos fuentes, pero haciendo uso del **PROTOCOLO IP – SUBPROTOCOLO UDP
(NO ORIENTADO A CONEXIÓN, QUE UTILIZA DATAGRAMAS)**

Nos dedicaremos a implementar un **PAR CLIENTE-SERVIDOR UDP**:

Veamos entonces como sería el código fuente del:

servidor-udp.c

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP:

```
/* Archivo: servidor-udp.c
 * Abre un socket UDP (NO ORIENTADO A CONEXIÓN).
 * Falta verificar si existen paquetes de entrada por el
 * socket ó por STDIN (select(2)).
 * Deberá ver el orden de envíos/recepción de mensajes
 * hacia/desde el servidor udp, es decir el usuario
 * debe sincronizar los envíos/recepciones.
 */
```

/* ARCHIVOS DE CABECERA */

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<sys/time.h>
#include<sys/select.h>
```

/* DEFINICIONES */

```
#define PORT 5000
#define MAX(x,y) ((x)>(y) ? (x) : (y))
#define SIZE 1024
#define TIME 3600
```

/* SINONIMOS*/

```
typedef struct sockaddr *sad;
/* FUNCIONES */
void error(char *s){
    perror(s);
    exit(-1);
}
```

/* FUNCION PRINCIPAL MAIN */

```
int main(){
    int sockio, largo, recibidos;
    struct sockaddr_in sinio;
    char linea[SIZE];

    //abre socket UDP
    if((sockio=socket(PF_INET, SOCK_DGRAM, 0)) < 0 )
        error("socket");
```

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP:

```
/* Continua: servidor-udp.c */
// Familia de direcciones de sockio
sinio.sin_family = AF_INET;

// Puerto, con bytes en orden de red, para sockio
sinio.sin_port = htons(PORT);

//dirección de internet, con bytes en orden de red, para sockio
sinio.sin_addr.s_addr = INADDR_ANY;

//publica dirección-puerto sinio de sockio
if( bind(sockio, (sad)&sinio, sizeof sinio) < 0 )
    error("bind");

largo = sizeof sinio;

for(;;){

    // lee hasta 1024 caracteres de sockio, los pone en linea
if((recibidos = recvfrom(sockio, linea, SIZE, 0, (sad)&sinio, &largo )) < 0)
    error("recvfrom");

    // marcar fin del buffer con ``0``
linea[recibidos]=0;

    /* Imprime en pantalla dirección de internet del cliente desde
     * donde vienen datos */
printf("\nDe la direccion[ %s ] : puerto[ %d ] --- llega el mensaje:\n",
       inet_ntoa(sinio.sin_addr), ntohs(sinio.sin_port));
printf("%s \n",linea);

    //lee hasta 1024 caracteres de STDIN, los pone en linea
fgets(linea, SIZE, stdin);

    //envía contenido de linea en sockio
if(sendto(sockio, linea, sizeof linea, 0, (sad)&sinio, largo ) < 0)
    error("sendto");
}

//cierra el conector sockio
close(sockio);
return 0;

}
/* Fin Archivo: servidor-udp.c */
```

Que **se compila** mediante: **gcc -Wall servidor-udp.c -o servidor-udp.out**

Y **se ejecuta** mediante: **./servidor-udp.out**

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP:

Debemos continuar con el código fuente del:

cliente-udp.c

```
/* Archivo: cliente-udp.c
 *      Abre un socket UDP (NO ORIENTADO A CONEXIÓN).
 *      Falta verificar si existen paquetes DATAGRAMAS de
 *      entrada por el socket ó por STDIN (select(2)).
 *      Deberá ver el orden de envíos/recepción de mensajes
 *      hacia/desde el servidor udp, es decir el usuario
 *      debe sincronizar los envíos/recepciones.
 **/
```

/* ARCHIVOS DE CABECERA */

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<sys/time.h>
#include<sys/select.h>
#include<string.h>
```

/* DEFINICIONES */

```
#define PORT 5000
#define MAX(x,y) ((x)>(y) ? (x) : (y))
#define SIZE 1024
#define TIME 3600
```

/* SINONIMOS */

```
typedef struct sockaddr *sad;
/* FUNCIONES */
void error(char *s){
    perror(s);
    exit(-1);
}
```

/* FUNCION PRINCIPAL MAIN */

```
int main(int argc, char **argv){
    if(argc < 2){
        fprintf(stderr,"usa: %s ipaddr \n", argv[0]);
        exit(-1);
    }
    //por sockio va a enviar-recibir paquetes datagramas
    int sockio, largo, recibidos;
```

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP:

```
/* continua: cliente-udp.c */

struct sockaddr_in sini, sino; // direcciones-puertos para sockio
// sini para recibir-leer
// sino para enviar-escribir

char linea[SIZE]; // buffer de 1024 caracteres (un arreglo)

// abre conector UDP
if((sockio = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
    error("socket");

//familia de direcciones a la que pertenece sino, para sockio
sino.sin_family = AF_INET;

// puerto de sino, para sockio
sino.sin_port = htons(PORT);

/* POSIX TO NETWORK, ver man 3 inet_pton transforma argv[1]
 * (la direccion IP pasada como argumento al programa)
 * desde formato ascii (puntero a cadena de caracteres)
 * al formato network
 * (guarda en la struct in_addr sino.sin_addr )
 * que a su vez es miembro de la struct sockaddr_in sino.
 * Se trata de la direccion IP del servidor al que quiere enviar
 * datagramas, perteneciente a la familia de direcciones AF_INET
 */
inet_pton(AF_INET, argv[1], &sino.sin_addr);

for(;){

    //leer hasta 1024 caracteres de STDIN y ponerlos en linea
    fgets(linea, SIZE, stdin);

    //escribir contenido de linea en sockio
    if(sendto(sockio, linea, sizeof linea, 0, (sad)&sino, sizeof sino) < 0 )
        error("sendto");
    largo = sizeof sini;
    if((recibidos = recvfrom(sockio, linea, SIZE, 0, (sad)&sino, &largo)) < 0)
        error("recvfrom"); //si recepcion devuelve < 0 ==> error al leer
    linea[recibidos] = 0; //marcar el final del buffer con ``0``
    /* Imprime en pantalla la direccion del servidor
     * desde donde vienen datos */
    printf("\nDe la direccion[ %s ] : puerto [ %d ] --- llega el mensaje:\n",
           inet_ntoa(sino.sin_addr),
           ntohs(sino.sin_port));
```

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP:

```
/* continua: cliente-udp.c */  
  
    //Imprime en pantalla lo que vino desde sockio mediante buffer linea  
    printf("%s \n",linea);  
}  
close(sockio); //cierra el conector sockio  
return 0;  
}  
/* Fin Archivo: cliente-udp.c */
```

Que **se compila** mediante: **gcc -Wall cliente-udp.c -o cliente-udp.out**

Y **se ejecuta respetando la sintaxis:** **cliente-udp.out IP_ADDRESS**

Por ejemplo **un ciclo de uso del PAR CLIENTE – SERVIDOR UDP puede ser:**

1º) Ejecutar el: **./servidor-udp.out**

2º) Ejecutar el: **./cliente-udp.out 127.0.0.1**

-----> 3º) Desde el Cliente: **enviar mensaje**

| 4º) Desde el Servidor: **responder mensaje**

----- 5º) Volver al punto 3º

6º) Terminar el proceso cliente/servidor mediante CTRL+C

Nota: si no se sigue el orden de envíos de mensajes, estos se irán encolando, y serán enviados inmediatamente después de que el cliente/servidor haga un envío al servidor/cliente.

Como este **COMPORTAMIENTO de nuestro PAR CLIENTE – SERVIDOR UDP ES INDESEABLE**, adicionaremos algunas **modificaciones para lograr un funcionamiento similar al PAR CLIENTE – SERVIDOR TCP**.

Por lo tanto prosigamos con la implementación de un **PAR CLIENTE – SERVIDOR UDP que utilizará select(2) para saber si hay algo que leer desde el respectivo socket o desde su respectiva STDIN para cada uno de estos procesos.**

Veamos el código fuente del

servidor-udp-con-select.c

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP CON SELECT:

```
/* Archivo: servidor-udp-con-select.c
 * Abre un socket UDP (NO ORIENTADO A CONEXIÓN).
 * Verifica con select(2) si existen datagramas de entrada
 * por el socket ó por STDIN.
 * Si se reciben paquetes por el socket, los envía a STDOUT.
 * Si se reciben paquetes por STDIN, los envía por el socket.
 */
```

/* ARCHIVOS DE CABECERA */

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<sys/time.h>
#include<sys/select.h>
```

/* DEFINICIONES */

```
#define PORT 5000
#define MAX(x,y) ((x)>(y) ? (x) : (y))
#define SIZE 1024
#define TIME 3600
```

/* SINONIMOS*/

```
typedef struct sockaddr *sad;
```

/* FUNCIONES */

```
void error(char *s){
    perror(s);
    exit(-1);
}
```

/* FUNCION PRINCIPAL MAIN */

```
int main(){
```

```
    int sockio, cuanto, largo, recibidos;
```

```
    // dirección-puerto para sockio
    struct sockaddr_in sinio;
```

```
    char linea[SIZE]; //buffer de 1024 caracteres
```

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP CON SELECT:

/* Continua: servidor-udp-con-select.c */

```
// conjunto de descriptores de ficheros: in e in_orig
fd_set in, in_orig;

/* estructura de datos necesaria para utilizar select(2)
 * en la que se almacena el tiempo límite ántes de que select(2) retorne */
struct timeval tv;

if((sockio=socket(PF_INET, SOCK_DGRAM, 0)) < 0 )
    error("socket");

// Familia de direcciones de sockio
sinio.sin_family = AF_INET;

// Puerto, con bytes en orden de red, para sockio
sinio.sin_port = htons(PORT);

/*dirección de internet, con bytes en orden de red,
 * para sockio */
sinio.sin_addr.s_addr = INADDR_ANY;

//publica dirección-puerto sinio, del conector sockio
if( bind(sockio, (sad)&sinio, sizeof sinio) < 0 )
    error("bind");

/*Lo que lea de STDIN mediante dirección sinio.sin_addr.s_addr
 *de sockio, se escribirá en sockio */

/*Lo que lea por sockio desde dirección remota
 *sinio.sin_addr, se escribirá en STDOUT */

/*tiene select*/
//Limpia el conjunto de descriptores de ficheros in_orig
FD_ZERO(&in_orig);

//añade STDIN al conjunto in_orig
FD_SET(0, &in_orig);

//añade sockio al conjunto in_orig
FD_SET(sockio, &in_orig);

/*tiene 1 hora*/
//tiempo hasta que select(2) retorne: 3600 segundos
tv.tv_sec=TIME;
tv.tv_usec=0;
```

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP CON SELECT:

```
/* Continua: servidor-udp-con-select.c */

for(;;){
    // copia conjunto in_orig en in
    memcpy(&in, &in_orig, sizeof in);

    /* espera a ver si se reciben datagramas por STDIN
     * o por sockio */
    if( (cuanto = select( MAX(0,sockio)+1, &in, NULL, NULL, &tv) ) < 0 )
        error("select: error");
    // si el tiempo de select(2) termina ==> error timeout
    if(cuanto == 0)
        error("select: timeout");

    largo = sizeof sinio;

    /* averigua donde hay algo para leer*/

    // Si hay para leer desde el conector sockio
    if(FD_ISSET(sockio, &in)){
        //recibe hasta 1024 caracteres de sockio y los pone en linea
        if((recibidos=recvfrom(sockio, linea, SIZE, 0, (sad)&sinio, &largo )) < 0)
            error("recvfrom");
        else if(recibidos == 0) //si lectura devuelve 0 -> parar ejecucion
            break;

        // marca el fin del buffer con '0'
        linea[recibidos]=0;

        /* Imprime en pantalla dirección de internet del cliente
         * desde donde vienen datos */
        printf("\nDe la direccion[ %s ] : puerto[ %d ] --- llega el mensaje:\n",
               inet_ntoa(sinio.sin_addr),
               ntohs(sinio.sin_port));

        // imprime el mensaje recibido
        printf("%s \n",linea);
    }
    //si hay para leer desde STDIN
    if(FD_ISSET(0,&in)){
        //lee hasta 1024 caracteres de STDIN, los pone en linea
        fgets(linea, SIZE, stdin);
```

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP CON SELECT:

/* Continua: servidor-udp-con-select.c */

```
//envía contenido de linea en sockio
if(sendto(sockio, linea, sizeof linea, 0, (sad)&sinio, largo) < 0 )
    error("sendto");
}
close(sockio); //cierra el conector sockio
return 0;
}
/* Fin Archivo: servidor-udp-con-select.c */
```

Que se compila mediante:

gcc -Wall servidor-udp-con-select.c -o servidor-udp-con-select.out

Y se ejecuta con: **./servidor-udp-con-select.out**

Ahora debemos ver el código del

cliente-udp-con-select.c

```
/* Archivo: cliente-udp-con-select.c
* Abre un socket UDP (NO ORIENTADO A CONEXIÓN).
* Verifica con select(2) si existen paquetes DATAGRAMAS de
* entrada por el socket ó por STDIN.
*
* Si se reciben paquetes por STDIN, los envía por el socket.
* (sendto(2))
*
* Si se reciben paquetes por el socket, los envía a STDOUT.
* (recvfrom(2))
**/
```

/* ARCHIVOS DE CABECERA */

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<sys/time.h>
#include<sys/select.h>
#include<string.h>
```

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP CON SELECT:

```
/* Continua: cliente-udp-con-select.c */

/* DEFINICIONES */
#define PORT 5000
#define MAX(x,y) ((x)>(y) ? (x) : (y))
#define SIZE 1024
#define TIME 3600

/* SINONIMOS */
typedef struct sockaddr *sad;

/* FUNCIONES */
void error(char *s){
    perror(s);
    exit(-1);
}

/* FUNCION PRINCIPAL MAIN */
int main(int argc, char **argv){
    if(argc < 2){
        fprintf(stderr,"usa: %s ipaddr \n", argv[0]);
        exit(-1);
    }
    //por sockio va a enviar-recibir paquetes datagramas
    int sockio, cuanto, largo, recibidos;

    //direcciones-puertos para sockio
    struct sockaddr_in sini, sino;
    // sini para recibir-leer
    // sino para enviar-escribir

    char linea[SIZE]; //buffer de 1024 caracteres (un arreglo)
fd_set in, in_orig;//conjuntos de descriptores de ficheros
struct timeval tv; //tiempo limite hasta que select(2) retorne

    // abre conector UDP
    if((sockio = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
        error("socket");
    //familia de direcciones a la que pertenece sino, para sockio
    sino.sin_family = AF_INET;
    sino.sin_port = htons(PORT); //puerto de sino, para sockio
```

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP CON SELECT:

/* Continua: cliente-udp-con-select.c */

```
/* POSIX TO NETWORK, ver man 3 inet_pton : transforma
 * argv[1] (la direccion IP pasada como argumento al programa)
 * desde formato ascii (puntero a cadena de caracteres)
 * al formato network
 * guarda en la struct in_addr sino.sin_addr que a su vez
 * es miembro de la struct sockaddr_in sino. Se trata de la
 * direccion IP del servidor al que quiere enviar datagramas,
 * pertenece a la familia de direcciones ``AF_INET``
 */
inet_pton(AF_INET, argv[1], &sino.sin_addr);

/*tiene select*/
//Limpia el conjunto de descriptores de fichero in_orig
FD_ZERO(&in_orig);

//añade al conjunto in_orig el descriptor STDIN
FD_SET(0, &in_orig);

//añade al conjunto in_orig el descriptor sockio
FD_SET(sockio, &in_orig);

/*tiene 1 hora*/
//tiempo disponible hasta que select(2) regrese: 3600 segundos
tv.tv_sec=TIME;
tv.tv_usec=0;
for(;;{
    // copia contenido del conjunto in_orig en in
    memcpy(&in, &in_orig, sizeof in);

    //ve si hay algún datagrama en el conjunto in
    if((cuanto = select(MAX(0,sockio)+1, &in, NULL, NULL, &tv)) < 0)
        error("select: error");

    //si el tiempo de espera de select(2) expira ==> timeout
    if(cuanto == 0)
        error("select: timeout");

    /*averiguamos donde hay algo para leer*/
```

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP CON SELECT:

/* Continua: cliente-udp-con-select.c */

```
//si hay para leer desde STDIN
if(FD_ISSET(0,&in)){
    //lee hasta 1024 caracteres de STDIN y pone en linea
    fgets(linea, SIZE, stdin);

    //envía contenido de linea en sockio
    if(sendto(sockio, linea, sizeof linea, 0, (sad)&sino, sizeof sino) < 0)
        error("sendto");
}

largo = sizeof sini;

//si hay para leer desde sockio
if(FD_ISSET(sockio, &in)){
    /*recibe hasta 1024 bytes desde sockio
     * mediante dirección-puerto sini
     * y guarda en linea */
    if((recibidos = recvfrom(sockio, linea, SIZE, 0, (sad)&sini, &largo)) < 0)
        error("recvfrom");
    else if(recibidos == 0)
        break; //si lectura devuelve 0 ==> parar la ejecucion

    //marcar el final del buffer con '0'
    linea[recibidos] = 0;

    /* Imprime en pantalla la dirección del servidor
     * desde donde vienen datos */
    printf("\nDe la direccion[ %s ] : puerto [ %d ] --- llega el mensaje:\n",
           inet_ntoa(sini.sin_addr),
           ntohs(sini.sin_port));

    //Imprime en pantalla datos recibidos
    printf("%s \n",linea);
}
close(sockio); // cierra el conector sockio
return 0;
}
/* Fin Archivo: cliente-udp-con-select.c */
```

Que se compila mediante:

gcc -Wall cliente-udp-con-select.c -o cliente-udp-con-select.out
Y se ejecuta respetando la sintaxis: **./cliente-udp-con-select.c IP_ADDRESS**

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP CON SELECT:

Por ejemplo **un ciclo de uso del**

PAR CLIENTE – SERVIDOR UDP CON SELECT puede ser:

1º) Iniciar el Servidor: `./servidor-udp-con-select.out`

2º) Iniciar el cliente: `./cliente-udp-con-select.out 127.0.0.1`

OPCIONAL: Puede iniciar otros procesos clientes de la misma forma.

3º) Desde cliente x enviar mensaje al servidor e inmediatamente desde servidor responder al cliente x.

OPCIONAL: Si hubieran otros clientes iniciados, siempre que un cliente x envía, los demás clientes deben esperar a que el servidor le responda, así se mantienen las comunicaciones mediante turnos. Si no se respeta, el servidor al responder siempre envía la respuesta al último cliente desde el cuál llegó el mensaje.

Cuando un cliente x envía un mensaje, solo el servidor lo recibe, y cuando el servidor responde, solo recibe el mensaje el cliente x, mientras que los otros procesos clientes no reciben ningún mensaje.

4º) Cerrar todos los procesos clientes con CTRL+C y el proceso servidor también de la misma manera.

Una mejora que podemos agregar a este **PAR CLIENTE – SERVIDOR UDP CON SELECT** se logra mediante **fcntl(2)** con su comando **F_SETFL** asignando el valor **O_NONBLOCK** para aplicarlo al conector del cliente y del servidor, así obtendriamos un comportamiento similar al anterior pero con la característica de funcionar en modo NO BLOQUEANTE.

Veamos entonces como sería el código fuente del:

`servidor-udp-con-select-fcntl.c`

```
/* Archivo: servidor-udp-con-select-fcntl.c
 * Abre un socket UDP (NO ORIENTADO A CONEXIÓN).
 * Mediante fcntl(2) usa la operación F_SETFL
 * para asignar a la bandera del descriptor sockio
 * el valor 'O_NONBLOCK'.
 * Verifica con select(2) si existen datagramas de entrada
 * por el socket ó por STDIN.
 * Si se reciben paquetes por el socket, los envía a STDOUT.
 * Si se reciben paquetes por STDIN, los envía por el socket.
 */
```

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP CON SELECT Y Fcntl:

```
/* Continua: servidor-udp-con-select-fcntl.c */
/* ARCHIVOS DE CABECERA */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<sys/time.h>
#include<sys/select.h>
#include <fcntl.h>
/* DEFINICIONES */
#define PORT 5000
#define MAX(x,y) ((x)>(y) ? (x) : (y))
#define SIZE 1024
#define TIME 3600
/* SINONIMOS*/
typedef struct sockaddr *sad;
/* FUNCIONES */
void error(char *s){
    perror(s);
    exit(-1);
}
/* FUNCION PRINCIPAL MAIN */
int main(){
    int sockio, cuanto, largo, recibidos;
    struct sockaddr_in sinio;
    char linea[SIZE];
    fd_set in, in_orig;
    struct timeval tv;
    //abre un conector UDP sockio
    if((sockio=socket(PF_INET, SOCK_DGRAM, 0)) < 0 )
        error("socket");
    // Familia de direcciones de sockio
    sinio.sin_family = AF_INET;
    // Puerto, con bytes en orden de red, para sockio
    sinio.sin_port = htons(PORT);
    /*dirección de internet, con bytes en orden de red,
     * para sockio */
    sinio.sin_addr.s_addr = INADDR_ANY;

    // Asigna al conector sockio la bandera 'O_NONBLOCK'
    if( fcntl(sockio, F_SETFL, O_NONBLOCK) < 0)
        error("fcntl");
```

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP CON SELECT Y Fcntl:

/* Continua: servidor-udp-con-select-fcntl.c */

```
//publica dirección-puerto sinio, del conector sockio
if( bind(sockio, (sad)&sinio, sizeof sinio) < 0 )
    error("bind");

/*Lo que lea de STDIN mediante dirección sinio.sin_addr
 *de sockio, se escribirá en sockio */

/*Lo que lea por sockio desde dirección remota
 *sinio.sin_addr, se escribirá en STDOUT */

/*tiene select*/
//Limpia el conjunto de descriptores de ficheros in_orig
FD_ZERO(&in_orig);
//añade STDIN al conjunto in_orig
FD_SET(0, &in_orig);
//añade sockio al conjunto in_orig
FD_SET(sockio, &in_orig);

/*tiene 1 hora*/
//tiempo hasta que select(2) retorne: 3600 segundos
tv.tv_sec=TIME;
tv.tv_usec=0;
for(;;){
    // copia conjunto in_orig en in
    memcpy(&in, &in_orig, sizeof in);

    /* espera a ver si se reciben datagramas por STDIN
     * o por sockio */
    if( (cuanto = select( MAX(0,sockio)+1, &in, NULL, NULL, &tv) ) < 0 )
        error("select: error");
    // si el tiempo de select(2) termina -> error timeout
    if(cuanto == 0)
        error("select: timeout");

    largo = sizeof sinio;

    /* averigua donde hay algo para leer*/
    // Si hay para leer desde el conector sockio
    if(FD_ISSET(sockio, &in)){
```

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP CON SELECT Y Fcntl:

/* Continua: servidor-udp-con-select-fcntl.c */

```
//recibe hasta 1024 caracteres de sockio y los pone en linea
if((recibidos=recvfrom(sockio, linea, SIZE, 0, (sad)&sinio, &largo )) < 0)
    error("recvfrom");
else if(recibidos == 0) //si lectura devuelve 0 -> parar ejecucion
    break;

// marca el fin del buffer con '0'
linea[recibidos]=0;

/* Imprime en pantalla dirección de internet del cliente
 * desde donde vienen datos */
printf("\nDe la direccion[ %s ] : puerto[ %d ] --- llega el mensaje:\n",
       inet_ntoa(sinio.sin_addr),
       ntohs(sinio.sin_port));

// imprime el mensaje recibido
printf("%s \n", linea);
}

//si hay para leer desde STDIN
if(FD_ISSET(0,&in)){
    //lee hasta 1024 caracteres de STDIN, los pone en linea
    fgets(linea, SIZE, stdin);

    //envía contenido de linea en sockio
    if(sendto(sockio, linea, sizeof linea, 0, (sad)&sinio, largo ) < 0 )
        error("sendto");
    }
close(sockio); //cierra el conector
return 0;
}
/* Fin Archivo: servidor-udp-con-select-fcntl.c */
```

Que se compila mediante:

```
gcc -Wall servidor-udp-con-select-fcntl.c
-o servidor-udp-con-select-fcntl.out
```

Y se ejecuta mediante: **./servidor-udp-con-select-fcntl.out**

Ahora veamos el código fuente del:

cliente-udp-con-select-fcntl.c

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP CON SELECT Y Fcntl:

```
/* Archivo: cliente-udp-con-select-fcntl.c
 * Abre un socket UDP (NO ORIENTADO A CONEXIÓN).
 * Mediante fcntl(2) usa la operación F_SETFL
 * para a signar a la bandera del descriptor sockio
 * el valor 'O_NONBLOCK'.
 * Verifica con select(2) si existen DATAGRAMAS de
 * entrada por el socket ó por STDIN.
 *
 * Si se reciben paquetes por STDIN, los envía por el socket.
 * (sendto(2))
 *
 * Si se reciben paquetes por el socket, los envía a STDOUT.
 * (recvfrom(2))
 */
/* ARCHIVOS DE CABECERA */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<sys/time.h>
#include<sys/select.h>
#include<string.h>
#include<fcntl.h>
/* DEFINICIONES */
#define PORT 5000
#define MAX(x,y) ((x)>(y) ? (x) : (y))
#define SIZE 1024
#define TIME 3600
/* SINONIMOS */
typedef struct sockaddr *sad;
/* FUNCIONES */
void error(char *s){
    perror(s);
    exit(-1);
}
/* FUNCION PRINCIPAL MAIN */
int main(int argc, char **argv){
    if(argc < 2){
        fprintf(stderr,"usa: %s ipaddr \n", argv[0]);
        exit(-1);
    }
    int sockio, cuanto, largo, recibidos;
    //por sockio va a enviar-recibir paquetes datagramas
```

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP CON SELECT Y Fcntl:

```
/* Continua: cliente-udp-con-select-fcntl.c */

//direcciones-puerto de internet para sockio
struct sockaddr_in sini, sino;
// sini para recibir-leer
// sino para enviar-escribir

char linea[SIZE]; //buffer de 1024 caracteres (un arreglo)
fd_set in, in_orig;//conjuntos de descriptores de ficheros
struct timeval tv; //tiempo limite hasta que select(2) retorne
// abre conector UDP
if((sockio = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
    error("socket");
//familia de direcciones a la que pertenece sino, para sockio
sino.sin_family = AF_INET;
sino.sin_port = htons(PORT); //puerto de sino, para sockio
inet_pton(AF_INET, argv[1], &sino.sin_addr);
/* POSIX TO NETWORK, ver man 3 inet_pton : transforma
 * argv[1] (la direccion IP pasada como argumento al programa)
 * desde formato ascii (puntero a cadena de caracteres)
 * al formato network
 * guarda en la struct in_addr sino.sin_addr que a su vez
 * es miembro de la struct sockaddr_in sino. Se trata de la
 * direccion IP del servidor al que quiere enviar datagramas,
 * pertenece a la familia de direcciones ``AF_INET``
 */
/* Asigna al conector sockio mediante la operación F_SETFL
 * el valor O_NONBLOCK con fcntl(2)
 */
if( fcntl(sockio, F_SETFL, O_NONBLOCK) < 0)
    error("fcntl");

/*tiene select*/
//Limpia el conjunto de descriptores de fichero in_orig
FD_ZERO(&in_orig);
//añade al conjunto in_orig el descriptor STDIN
FD_SET(0, &in_orig);
//añade al conjunto in_orig el descriptor sockio
FD_SET(sockio, &in_orig);

/*tiene 1 hora*/
//tiempo disponible hasta que select(2) regrese: 3600 segundos
tv.tv_sec=TIME;
tv.tv_usec=0;
```

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP CON SELECT Y Fcntl:

```
/* Continua: cliente-udp-con-select-fcntl.c */

for(;;){
    // copia contenido del conjunto in_orig en in
    memcpy(&in, &in_orig, sizeof in);

    //ve si hay algún datagrama en el conjunto in
    if((cuanto = select(MAX(0,sockio)+1, &in, NULL, NULL, &tv)) < 0)
        error("select: error");

    //si el tiempo de espera de select(2) expira ==> timeout
    if(cuanto == 0)
        error("select: timeout");

    /*averiguamos donde hay algo para leer*/
    //si hay para leer desde STDIN
    if(FD_ISSET(0,&in)){
        //lee hasta 1024 caracteres de STDIN y pone en linea
        fgets(linea, SIZE, stdin);

        //envía contenido de linea en sockio
        if(sendto(sockio, linea, sizeof linea, 0, (sad)&sino, sizeof sino) < 0)
            error("sendto");
    }

    largo = sizeof sini;

    //si hay para leer desde sockio
    if(FD_ISSET(sockio, &in)){
        /* recibe hasta 1024 bytes desde sockio
         * mediante dirección-puerto sini y guarda en linea */
        if((recibidos = recvfrom(sockio, linea, SIZE, 0, (sad)&sini, &largo)) < 0)
            error("recvfrom");
        else if(recibidos == 0)
            break; //si lectura devuelve 0 ==> parar la ejecucion
        //marcar el final del buffer con '0'
        linea[recibidos] = 0;

        /* Imprime en pantalla la dirección del servidor
         * desde donde vienen datos */
        printf("\nDe la direccion[ %s ] : puerto [ %d ] --- llega el mensaje:\n",
              inet_ntoa(sini.sin_addr),
              ntohs(sini.sin_port));
    }
}
```

Clase 03-10-2006: PAR CLIENTE SERVIDOR UDP CON SELECT Y Fcntl:

```
/* Continua: cliente-udp-con-select-fcntl.c */
```

```
    //Imprime en pantalla datos recibidos
    printf("%s \n",linea);
}
}
close(sockio); //cierra el conector
return 0;
}
/* Fin Archivo: cliente-udp-con-select-fcntl.c */
```

Que se compila mediante:

```
gcc -Wall cliente-udp-con-select-fcntl.c
-o cliente-udp-con-select-fcntl.out
```

Y se ejecuta respetando la sintaxis:

```
./cliente-udp-con-select-fcntl.out IP_ADDRESS
```

Un ciclo de uso, de ejemplo, de este

PAR CLIENTE – SERVIDOR UDP CON SELECT Y FCNTL

puede ser como el siguiente:

- 1º) Ejecuta el Servidor: ./servidor-udp-con-select-fcntl.out
- 2º) Ejecuta un cliente: ./cliente-udp-con-select-fcntl.out 127.0.0.1
- > 3º) Cliente envía mensaje
- | 4º) Servidor responde mensaje
- 5º) Opcionalmente continuar la comunicación Servidor-Cliente
- > 6º) Opcionalmente ejecutar otro Cliente (llamaremos **Cliente X**):
| ./cliente-udp-con-select-fcntl.out 127.0.0.1
- | ---> 7º) **Cliente X** envía mensaje al Servidor, tomando el control de la
| | comunicación bidireccional mediante datagramas.
- | | 8º) Servidor responde al Cliente X (y el datagrama es enviado sólo
| | a ese **Cliente X**, mientras que los otros clientes no reciben
| | mensaje alguno.
- | | 9º) Proseguir la comunicación Servidor-**Cliente X**
- 10º) Opcionalmente ejecutar otro Cliente
- 11º) Cuando desee terminar, finalice todos los procesos Clientes y el
proceso Servidor mediante **CTRL+C**

Pero **este par CLIENTE -SERVIDOR UDP CON SELECT Y FCNTL** todavía posee una **falla que se puede evitar**, el mensaje de error:
“Address already in use”.

Para evitarlo, vamos a usar **setsockopt(2)** aplicando la **opción del nivel 'SOL_SOCKET'** con el **valor 'SO_REUSEADDR'** a **los conectores del CLIENTE y del SERVIDOR**.

Clase 03-10-2006:
PAR CLIENTE SERVIDOR UDP CON SELECT, Fcntl Y Setsockopt:

```
/* Archivo: cliente-udp-con-select-fcntl-setsockopt.c
*
* Abre un socket UDP (NO ORIENTADO A CONEXIÓN).
*
* Mediante fcntl(2) usa la operación 'F_SETFL'
* para asignar a la bandera del descriptor 'sockio'
* el valor 'O_NONBLOCK'.
*
* Mediante setsockopt(2) aplica la opción del nivel
* 'SOL_SOCKET' con el valor 'SO_REUSEADDR' al
* conector 'sockio'
*
* Verifica con select(2) si existen DATAGRAMAS de
* entrada por el socket ó por STDIN.
*
* Si se reciben paquetes por STDIN, los
* envía por el socket (sendto(2)).
*
* Si se reciben paquetes por el socket, los
* envía a STDOUT (recvfrom(2)).
*/
/* ARCHIVOS DE CABECERA */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<sys/time.h>
#include<sys/select.h>
#include<string.h>
#include<fcntl.h>
/* DEFINICIONES */
#define PORT 5000
#define MAX(x,y) ((x)>(y) ? (x) : (y))
#define SIZE 1024
#define TIME 3600
/* SINONIMOS */
typedef struct sockaddr *sad;
/* FUNCIONES */
void error(char *s){
    perror(s);
    exit(-1);
}
```

Clase 03-10-2006:

PAR CLIENTE SERVIDOR UDP CON SELECT, FCBNTL Y SETSOCKOPT:

```
/* Continua: cliente-udp-con-select-fcbntl-setsockopt.c */
/* FUNCION PRINCIPAL MAIN */
int main(int argc, char **argv){
    if(argc < 2){
        fprintf(stderr,"usa: %s ipaddr \n", argv[0]);
        exit(-1);
    }
    const int yes = 1;
    int sockio, cuanto, largo, recibidos;
    // Por 'sockio' va a enviar-recibir paquetes datagramas

    // Direcciones-puertos de internet para 'sockio'
    struct sockaddr_in sini, sino;
    // 'sini' para recibir-leer
    // 'sino' para enviar-escribir

    // Buffer de 1024 caracteres (un arreglo)
    char linea[SIZE];
    // Conjuntos de descriptores de ficheros
    fd_set in, in_orig;
    // Tiempo limite hasta que select(2) retorne
    struct timeval tv;
    // abre conector UDP 'sockio'
    if((sockio = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
        error("socket");
    /* Familia de direcciones a la que pertenece
     * 'sino', para 'sockio' */
    sino.sin_family = AF_INET;
    // Puerto de 'sino', para 'sockio'
    sino.sin_port = htons(PORT);
    inet_pton(AF_INET, argv[1], &sino.sin_addr);
    /* POSIX TO NETWORK, ver man 3 inet_pton :
     * transforma argv[1]
     * (la direccion IP pasada como argumento)
     * desde formato ascii
     * (puntero a cadena de caracteres)
     * al formato network, y guarda en la
     * 'struct in_addr sino.sin_addr' que a su vez
     * es miembro de la 'struct sockaddr_in sino' .
     * Se trata de la direccion IP del servidor
     * al que quiere enviar datagramas, pertenece
     * a la familia de direcciones 'AF_INET'
    */
}
```

**Clase 03-10-2006:
PAR CLIENTE SERVIDOR UDP CON SELECT, Fcntl Y Setsockopt:**

```
/* Continua: cliente-udp-con-select-fcntl-setsockopt.c */

/* Asigna al conector 'sockio' mediante la
 * operación 'F_SETFL' el valor 'O_NONBLOCK'
 * con fcntl(2). */
if( fcntl(sockio, F_SETFL, O_NONBLOCK) < 0)
    error("fcntl");

/* Mediante setsockopt(2) aplica la opción del nivel
 * 'SOL_SOCKET' con el valor 'SO_REUSEADDR' al
 * conector 'sockio' */
if(setsockopt(sockio,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof yes)<0)
    error("setsockopt");

/* Tiene select(2) */
// Limpia el conjunto de descriptores de fichero 'in_orig'
FD_ZERO(&in_orig);
// Añade al conjunto 'in_orig' el descriptor STDIN
FD_SET(0, &in_orig);
// Añade al conjunto 'in_orig' el descriptor 'sockio'
FD_SET(sockio, &in_orig);

/* Tiene 1 hora */
// Tiempo disponible hasta que select(2) regrese: 3600 segundos
tv.tv_sec=TIME;
tv.tv_usec=0;
for(;;){
    // Copia contenido del conjunto 'in_orig' en 'in'
    memcpy(&in, &in_orig, sizeof in);

    // Ve si hay algún datagrama en el conjunto 'in'
    if((cuanto = select(MAX(0,sockio)+1, &in, NULL, NULL, &tv)) < 0)
        error("select: error");

    // Si el tiempo de espera de select(2) expira ==> timeout
    if(cuanto == 0)
        error("select: timeout");
```

Clase 03-10-2006:
PAR CLIENTE SERVIDOR UDP CON SELECT, Fcntl Y Setsockopt:

```
/* Continua: cliente-udp-con-select-fcntl-setsockopt.c */
/* Averiguamos donde hay algo para leer */

    // Si hay para leer desde STDIN
    if(FD_ISSET(0,&in)){
        /* Lee hasta 1024 caracteres de STDIN y
         * pone en linea */
        fgets(linea, SIZE, stdin);
        // Envía contenido de linea en 'sockio'
        if(sendto(sockio, linea, sizeof linea, 0, (sad)&sino, sizeof sino) < 0)
            error("sendto");
    }

    largo = sizeof sini;

    // Si hay para leer desde 'sockio'
    if(FD_ISSET(sockio, &in)){
        /* Recibe hasta 1024 bytes desde 'sockio'
         * mediante dirección-puerto 'sini' y
         * guarda en 'linea' */
        if((recibidos = recvfrom(sockio, linea, SIZE, 0, (sad)&sini, &largo)) < 0)
            error("recvfrom");
        else if(recibidos == 0)
            break; // Si recepción devuelve 0 ==> parar la ejecucion
        // Marcar el final del buffer con '0'
        linea[recibidos] = 0;
        /* Imprime en pantalla la dirección del servidor
         * desde donde vienen datos */
        printf("\nDe la direccion[ %s ] : puerto [ %d ] --- llega el mensaje:\n",
               inet_ntoa(sini.sin_addr),
               ntohs(sini.sin_port));
        // Imprime en pantalla datos recibidos
        printf("%s \n",linea);
    }
}

close(sockio); // Cierra el conector UDP
return 0; // Finalización exitosa
}
/* Fin Archivo: cliente-udp-con-select-fcntl-setsockopt.c */
```

Que **se compila** mediante:

```
gcc -Wall     cliente-udp-con-select-fcntl-setsockopt.c
      -o       cliente-udp-con-select-fcntl-setsockopt.out
```

Y **se ejecuta** de acuerdo a la sintaxis:

```
./cliente-udp-con-select-fcntl-setsockopt.out IP_ADDRESS
```

**Clase 03-10-2006:
PAR CLIENTE SERVIDOR UDP CON SELECT, Fcntl Y Setsockopt:**

```
/* Archivo: servidor-udp-con-select-fcntl-setsockopt.c
*
* Abre un socket UDP 'sockio' (NO ORIENTADO A CONEXIÓN).
*
* Mediante fcntl(2) usa la operación 'F_SETFL'
* para asignar a la bandera del descriptor 'sockio'
* el valor 'O_NONBLOCK'.
*
* Mediante setsockopt(2) aplica la opción de nivel
* 'SOL_SOCKET' con el valor 'SO_REUSEADDR' al
* conector 'sockio'.
*
* Verifica con select(2) si existen datagramas de entrada
* por el socket ó por STDIN.
*
* Si se reciben paquetes por el socket, los
* envía a STDOUT (recvfrom(2)).
*
* Si se reciben paquetes por STDIN, los
* envía por el socket (sendto(2)).
*/
/* ARCHIVOS DE CABECERA */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<sys/time.h>
#include<sys/select.h>
#include <fcntl.h>
/* DEFINICIONES */
#define PORT 5000
#define MAX(x,y) ((x)>(y) ? (x) : (y))
#define SIZE 1024
#define TIME 3600
/* SINONIMOS*/
typedef struct sockaddr *sad;
/* FUNCIONES */
void error(char *s){
    perror(s);
    exit(-1);
}
```

**Clase 03-10-2006:
PAR CLIENTE SERVIDOR UDP CON SELECT, Fcntl Y Setsockopt:**

/* Continua: servidor-udp-con-select-fcntl-setsockopt.c */

/* FUNCION PRINCIPAL MAIN */

```
int main(){
```

```
// Para setsockopt(2): SO_REUSEADDR
```

```
const int yes = 1;
```

```
int sockio, cuanto, largo, recibidos;
```

```
// Dirección-puerto 'sinio' para 'sockio'
```

```
struct sockaddr_in sinio;
```

```
char linea[SIZE];
```

```
fd_set in, in_orig;
```

```
struct timeval tv;
```

```
// Abre un conector UDP 'sockio'
```

```
if((sockio = socket(PF_INET, SOCK_DGRAM, 0)) < 0 )  
    error("socket");
```

```
// Familia de direcciones de 'sinio', para 'sockio'
```

```
sinio.sin_family = AF_INET;
```

```
/* Puerto, con bytes en orden de red, de 'sinio'
```

```
* para 'sockio' */
```

```
sinio.sin_port = htons(PORT);
```

```
/* Dirección de internet, con bytes en orden de red,
```

```
* de 'sinio', para 'sockio' */
```

```
sinio.sin_addr.s_addr = INADDR_ANY;
```

```
// Asigna al conector 'sockio' la bandera 'O_NONBLOCK'
```

```
if( fcntl(sockio, F_SETFL, O_NONBLOCK) < 0)
```

```
    error("fcntl");
```

```
/* Mediante setsockopt(2) aplica la opción de nivel
```

```
* 'SOL_SOCKET' con el valor 'SO_REUSEADDR' al
```

```
* conector 'sockio' */
```

```
if(setsockopt(sockio,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof yes)<0)  
    error("setsockopt");
```

```
/* Publica la dirección-puerto 'sinio',
```

```
* del conector 'sockio' */
```

```
if( bind(sockio, (sad)&sinio, sizeof sinio) < 0 )  
    error("bind");
```

```
/* Tiene select(2) */
```

**Clase 03-10-2006:
PAR CLIENTE SERVIDOR UDP CON SELECT, Fcntl Y Setsockopt:**

/* Continua: servidor-udp-con-select-fcntl-setsockopt.c */

```
/* Limpia el conjunto de descriptores
 * de ficheros 'in_orig' */
FD_ZERO(&in_orig);
// Añade STDIN al conjunto 'in_orig'
FD_SET(0, &in_orig);
//añade 'sockio' al conjunto 'in_orig'
FD_SET(sockio, &in_orig);
/* Tiene 1 hora */
// Tiempo hasta que select(2) retorne: 3600 segundos
tv.tv_sec=TIME;
tv.tv_usec=0;
for(;;){
    // Copia conjunto 'in_orig' en 'in'
    memcpy(&in, &in_orig, sizeof in);
    /* Espera a ver si se reciben datagramas por STDIN
     * o por 'sockio' */
    if( (cuanto = select( MAX(0,sockio)+1, &in, NULL, NULL, &tv ) ) < 0 )
        error("select: error");
    // Si el tiempo de select(2) termina -> error timeout
    if(cuanto == 0)
        error("select: timeout");
    largo = sizeof sinio;
    /* Averigua donde hay algo para leer*/
    // Si hay para leer desde el conector 'sockio'
    if(FD_ISSET(sockio, &in)){
        /* Recibe hasta 1024 caracteres de 'sockio'
         * y los pone en 'linea' */
        if((recibidos = recvfrom(sockio, linea, SIZE, 0, (sad)&sinio, &largo )) < 0)
            error("recvfrom");
        else if(recibidos == 0) // Si recepción devuelve 0
            break;          // parar la ejecución
        // Marca el fin del buffer con '0'
        linea[recibidos]=0;
        /* Imprime en pantalla dirección de internet
         * del cliente desde donde vienen datos */
        printf("\nDe la direccion[ %s ] : puerto[ %d ] --- llega el mensaje:\n",
               inet_ntoa(sinio.sin_addr),
               ntohs(sinio.sin_port));
        // Imprime el mensaje recibido
        printf("%s \n",linea);
    }
}
```

Clase 03-10-2006:
PAR CLIENTE SERVIDOR UDP CON SELECT, FCNTL Y SETSOCKOPT:

/* Continua: servidor-udp-con-select-fcntl-setsockopt.c */

```
// Si hay para leer desde STDIN
if(FD_ISSET(0,&in)){
    /* Lee hasta 1024 caracteres de STDIN,
     * los pone en 'linea' */
    fgets(linea, SIZE, stdin);
    // Envía contenido de 'linea' en 'sockio'
    if(sendto(sockio, linea, sizeof linea, 0, (sad)&sinio, largo) < 0 )
        error("sendto");
}
close(sockio); // Cierra el conector UDP 'sockio'
return 0; // Finalización exitosa
}
/* Fin Archivo: servidor-udp-con-select-fcntl-setsockopt.c */
```

Que **se compila** mediante:

```
gcc -Wall servidor-udp-con-select-fcntl-setsockopt.c
      -o servidor-udp-con-select-fcntl-setsockopt.out
```

Y **se ejecuta** mediante:

```
./servidor-udp-con-select-fcntl-setsockopt.out
```

El ciclo de uso de este PAR CLIENTE – SERVIDOR UDP CON SELECT, FCNTL Y SETSOCKOPT es similar a los vistos anteriormente.

NOTA: Si intentan hacer un **PAR CLIENTE-SERVIDOR TCP CON SELECT, FCNTL Y SETSOCKOPT**, verán que podrán compilarlo pero que su funcionamiento es algo extraño, y esto se debe a la aplicación de la función **fcntl(2)** sobre los **CONECTORES TCP**.

Ahora en más comenzaremos a tratar un nuevo tema llamado

REMOTE PROCEDURE CALL , mejor conocido como 'RPC':

Surge en **1980** durante el **desarrollo del Sistema Operativo MACH (PRIMER MICRO-NÚCLEO) (R. Rachid et al.)**.

En **1982**, **SUN presenta** una técnica similar y la denomina **SUNRPC**.

Sigue un model similar al MODELO CLIENTE – SERVIDOR, que se lo conoce como:

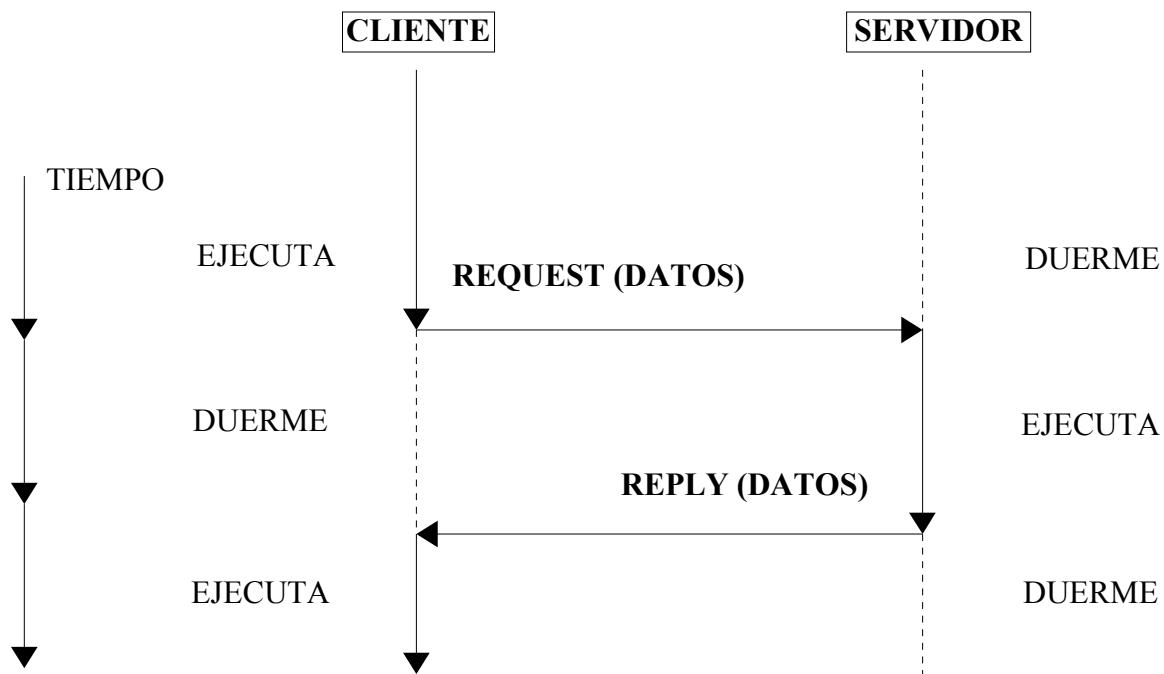
EL MODELO REQUEST – REPLY

que en español sería algo así como:

EL MODELO SOLICITUD – RESPUESTA

Clase 03-10-2006: REMOTE PROCEDURE CALL – RPC:

Este es **modelo REQUEST – REPLY** es **síncrono**, pues veamos:



Al mismo tiempo que uno de los procesos “pasa a ejecutarse”, el otro “pasa a dormir”.

Este **esquema** es **muy propio de una función “toma datos” y “entrega datos”**.

Antes de continuar veremos que dice “WIKIPEDIA” con respecto a RPC:

RPC

De Wikipedia, la enciclopedia libre

Saltar a [navegación](#), [búsqueda](#)

Para el [Estado socialista de China continental](#), véase [República Popular China](#).

El **RPC** (del [inglés](#) *Remote Procedure Call, Llamada a Procedimiento Remoto*) es un [protocolo](#) que permite a un programa de ordenador ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambos. El protocolo es un gran avance sobre los [sockets](#) usados hasta el momento. De esta manera el programador no tenía que estar pendiente de las comunicaciones, estando éstas encapsuladas dentro de las RPC.

Clase 03-10-2006: REMOTE PROCEDURE CALL – RPC:

Las RPC son muy utilizadas dentro del [paradigma cliente-servidor](#). Siendo el cliente el que inicia el proceso solicitando al servidor que ejecute cierto procedimiento o función y enviando éste de vuelta el resultado de dicha operación al cliente.

Hay distintos tipos de RPC, muchos de ellos estandarizados como pueden ser el RPC de Sun ([RFC 1057](#)), Distributed Computing Environment ([DCE](#)), [DCOM](#) de [Microsoft](#). No siendo compatibles entre sí. La mayoría de ellos utilizan un lenguaje de descripción de interfaz ([IDL](#)) que define los métodos exportados por el servidor.

Hoy en día se está utilizando el [XML](#) como lenguaje para definir el IDL y el [HTTP](#) como protocolo de red. Dando lugar a lo que se conoce como [servicios web](#). Ejemplos de éstos pueden ser [SOAP](#) o [XML-RPC](#).

[editar] Enlaces relacionados

- [Sun Microsystems](#) (en inglés)
- [Soap](#) (en inglés)
- [RFC 1057](#) (en inglés)

Ahora veamos esta otra fuente:

<http://ditec.um.es/laso/docs/tut-tcip/3376c410.html>

4.10 RPC("Remote Procure Call")

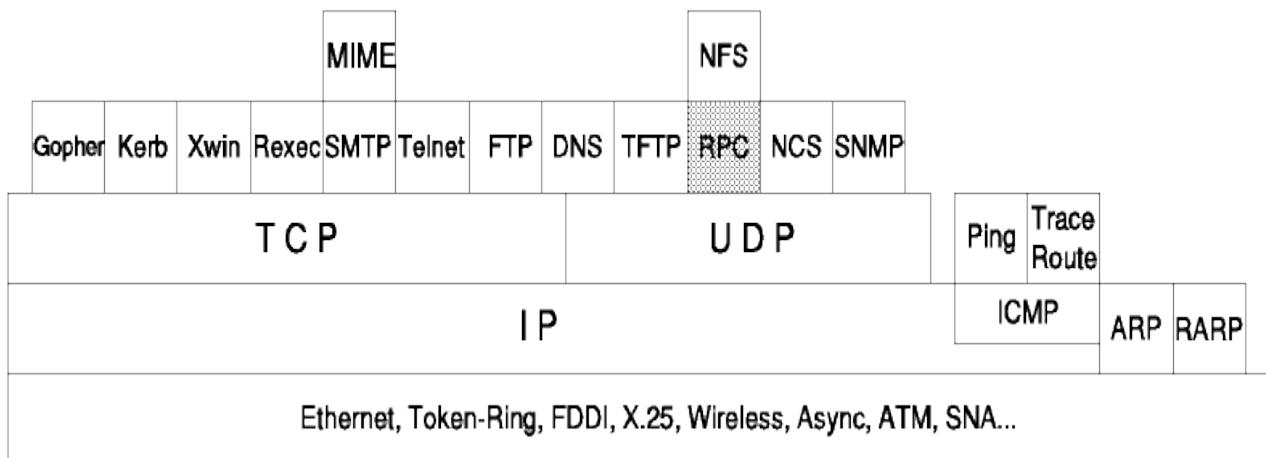


Figure: RPC("Remote Procure Call")

El RPC de Sun es un *protocolo propuesto como estándar*. Su status es *electivo*.

RCP es un estándar desarrollado por Sun Microsystems y usado por muchos distribuidores de sistemas UNIX. La especificación actual de UNIX se halla en el *RFC 1057 - RPC ("Remote Procure Call")*: *especificación de protocolo de la versión 2*.

El RPC es una interfaz de programación de aplicación(API) disponible para el desarrollo de aplicaciones distribuidas. Permite que los programas llamen a subrutinas que se ejecutan en un

Clase 03-10-2006: REMOTE PROCEDURE CALL – RPC:

sistema remoto. El programa llamador, denominado (llamado *client*) envía una *mensaje de llamada* al proceso *proceso servidor* y espera por un *mensaje de respuesta*. La llamada incluye los parámetros del procedimiento y la respuesta los resultados.

El RPC de Sun consta de las siguientes partes:

- *RPCGEN*: Un compilador que toma la definición de la interfaz de un procedimiento remoto, y genera los "stubs" del cliente y del servidor.
- *XDR ("eXternal Data Representation")*: Una forma estándar de codificar datos de modo para que sean transportables entre distintos sistemas. Impone una ordenación big - endian de los bytes y el tamaño mínimo de cualquier campo ha de ser 32 bits. Esto significa que tanto el cliente como el servidor han de realizar algún tipo de traducción.
- Una librería "run-time".

4.10.1 Concepto de RPC

El concepto de RPC se puede simplificar del modo siguiente:

- El proceso llamador envía un mensaje de llamada y espera por la respuesta.
- En el lado del servidor un proceso permanece dormido a la espera de mensajes de llamada. Cuando llega una llamada, el proceso servidor extrae los parámetros del procedimiento, calcula los resultados y los devuelve en un mensaje de respuesta.

Ver [Figura - RPC](#) muestra un modelo concepcual de RPC.

Este es sólo un posible modelo, ya que el protocolo RPC de Sun no impone restricciones específicas en el modelo de concurrencia. En el modelo anterior, el proceso llamador se bloquea hasta que se recibe un mensaje de respuesta. Otros modelos son igualmente posibles; por ejemplo, el llamador puede continuar su ejecución mientras espera una respuesta, o el servidor puede despachar una tarea separada para cada llamada que reciba de modo que quede libre para recibir otros mensajes.

Las llamadas a procedimientos remotos difieren de las llamadas a procedimientos locales en los siguientes aspectos:

- Uso de variables globales ya que el servidor no tiene acceso al espacio de memoria del llamador.
- El rendimiento puede verse afectado por los tiempos de transmisión.
- Puede ser necesaria la autentificación del usuario.
- Se debe conocer la dirección del servidor.

Clase 03-10-2006: REMOTE PROCEDURE CALL – RPC:

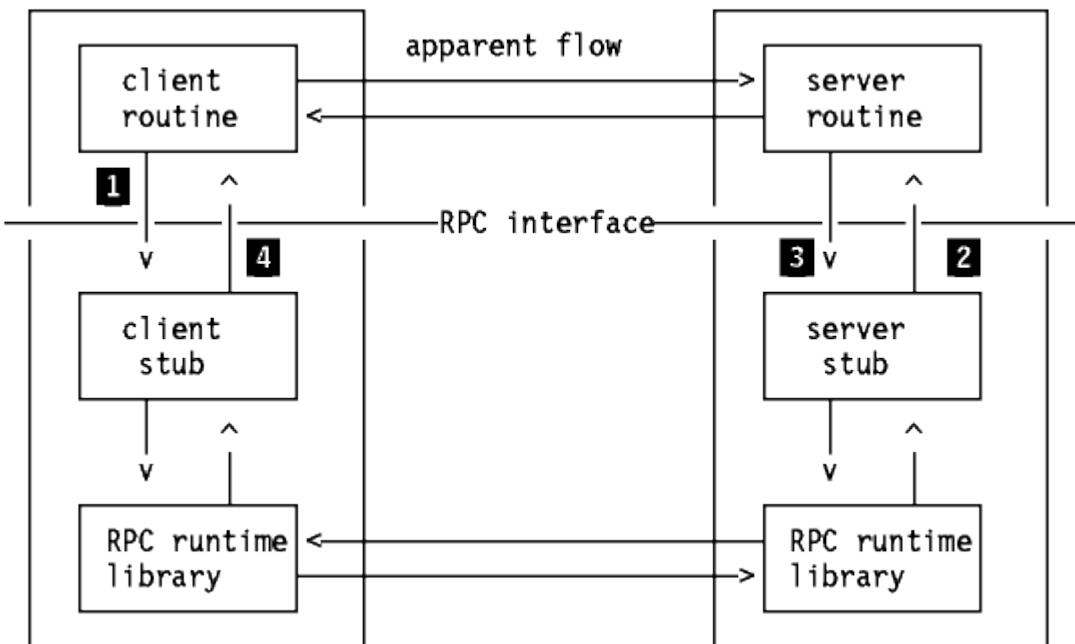


Figure: RPC - Modelo de llamadas a procedimientos remotos

4.10.1.1 Transporte

El protocolo RPC se puede implementar sobre cualquier protocolo de transporte. En el caso de TCP/IP, puede usar tanto TCP como UDP como capa de transporte. El tipo de *transporte* es un parámetro del comando *RPCGEN*. En caso de que se use UDP, recuérdese que no proporciona fiabilidad, por lo que dependerá del programa llamador el garantizarla (usando tiempos límite y retransmisiones, implementadas normalmente en rutinas de librería e RPC). Cabe señalar que incluso con TCP, el programa llamador sigue necesitando una rutina para el tiempo límite con el fin de tratar situaciones excepcionales, como por ejemplo la caída del servidor.

Los mensajes de llamada y respuesta se formatean al estándar XDR.

4.10.1.2 Mensaje de llamada de RPC

El mensaje de llamada de RPC consta de varios campos:

- Números de programa y de procedimiento

Cada llamada contiene tres campos (enteros sin signo):

- Número del programa remoto
- Número de versión del programa remoto
- Número del procedimiento remoto

que identifican únicamente al procedimiento e ejecutar. El número de programa remoto identifica un grupo funcional de procedimientos, por ejemplo, un sistema de ficheros, que incluiría procedimientos individuales como "leer" y "escribir". Estos procedimientos individuales se identifican con un número de procedimiento único dentro del programa remoto. A medida que el programa remoto evoluciona, a cada versión se le asigna un número de versión.

Cada programa remoto está conectado a un puerto de. El número de este puerto se

Clase 03-10-2006: REMOTE PROCEDURE CALL – RPC:

puede elegir libremente, exceptuando los puertos reservados para "servicios bien conocidos". Es evidente que el llamador tendrá que conocer el número de puerto usado por el programa remoto.

Números de programas asignados:

00000000 - 1FFFFFFF
definidos por Sun
20000000 - 3FFFFFFF
definidos por el usuario
40000000 - 5FFFFFFF
de transición(números temporales)
60000000 - FFFFFFFF
reservados

● Campos de autentificación

Existen dos campos, *credenciales* y *verificador*, para la autentificación del llamador al servicio. Depende del servidor el usar esta información para la autentificación del usuario. Además, cada implementación es libre de elegir entre los varios protocolos de autentificación que están soportados. Algunos de ellos son:

- Autentificación nula.
- Autentificación UNIX. Los llamadores de un procedimiento remoto se pueden identificar de igual modo que en el sistema UNIX.
- Autentificación DES. Además del identificador de usuario, al servidor se le envía un campo correspondiente a un sello de tiempo. Este sello de tiempo es la hora actual, cifrada con una llave conocida sólo para el servidor y el llamador (basado en el concepto de *llave secreta* y *llave pública* de DES).
- Parámetros de los procedimientos.

Los datos(parámetros) pasados al procedimiento remoto.

4.10.1.3 Mensaje de respuesta de RPC

Existen diversas respuestas, dependiendo del tipo de acción a tomar:

- SUCCESS: los resultados del procedimiento se devuelven al cliente.
- RPC_MISMATCH: el servidor está ejecutando una versión de RPC distinta de la del llamador.
- AUTH_ERROR: autentificación de usuario fallida.
- PROG_MISMATCH: el programa no está disponible, la versión solicitada no existe o el procedimiento no está disponible.

Para un descripción detallada de los mensajes de llamada y respuesta, ver el *RFC 1057 - RPC: ("Remote Procedure Call")*: especificación de protocolo de la versión 2, que contiene además las definiciones de tipos (typedef) para los mensajes en el lenguaje XDR.

4.10.1.4 Portmap o Portmapper(Mapeador de puertos)

Como se indica más arriba, el llamador tiene que conocer el número de puerto exacto usado por un programa RPC concreto para ser capaz de enviarle un mensaje. Portmap es una aplicación del

Clase 03-10-2006: REMOTE PROCEDURE CALL – RPC:

servidor que mapea el número de programa y de versión al puerto usado por ese programa. Debido a que portmap tiene asignado el número de puerto reservado (*servicio bien conocido*) 111, todo lo que tiene que hacer el llamador es preguntarle al servicio Portmap en el host remoto por el puerto usado por el programa servidor. Ver [Figura - Portmap](#).

Portmap sólo tiene conocimiento de los programas de su host(sólo programas RPC en el host local).

Con el fin de que Portmap adquiera conocimiento del RPC, cada programa RPC debería registrarse con el Portmap local cuando este arranca. También debería anular su registro cuando el Portmap finaliza su ejecución.

Normalmente, la aplicación llamadora contacta con el Portmap en el host de destino para obtener el número de puerto correcto de un programa remoto determinado, y luego envía el mensaje de llamada a ese puerto. Existe una variante consistente en que el llamador envía también los parámetros del procedimiento al Portmap y este a su vez se encarga de invocarlo.

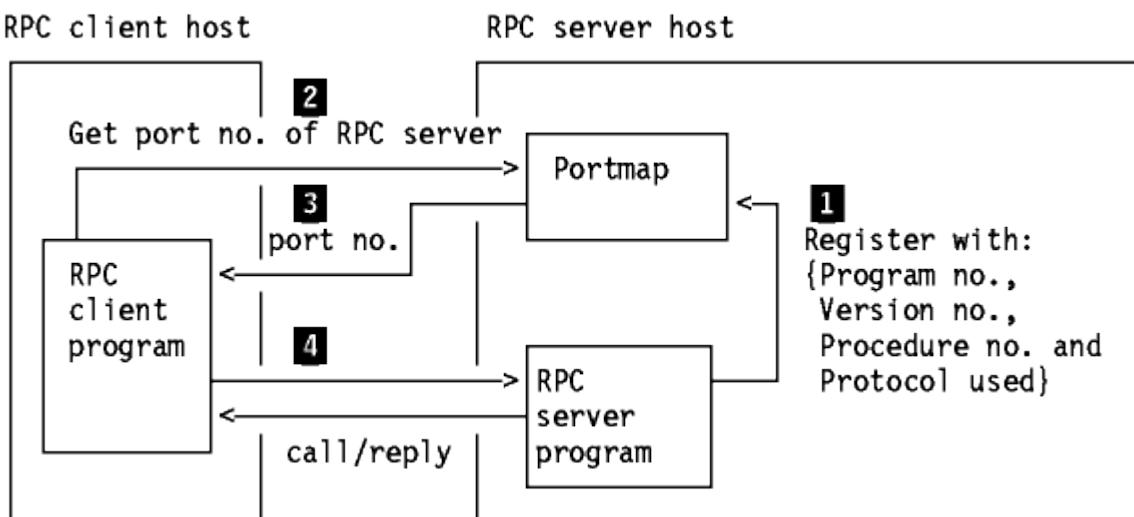


Figura: Portmap - Informa al llamador de que número de puerto ocupa un programa en su host.

4.10.1.5 RPCGEN

RPCGEN es una herramienta que genera código en C para el protocolo RPC. La entrada de RPCGEN es un fichero escrito en un lenguaje similar a C, conocido como lenguaje RPC. Asumiendo que se usa un fichero de entrada llamado *proto.x*, RPCGEN produce los siguientes ficheros de salida:

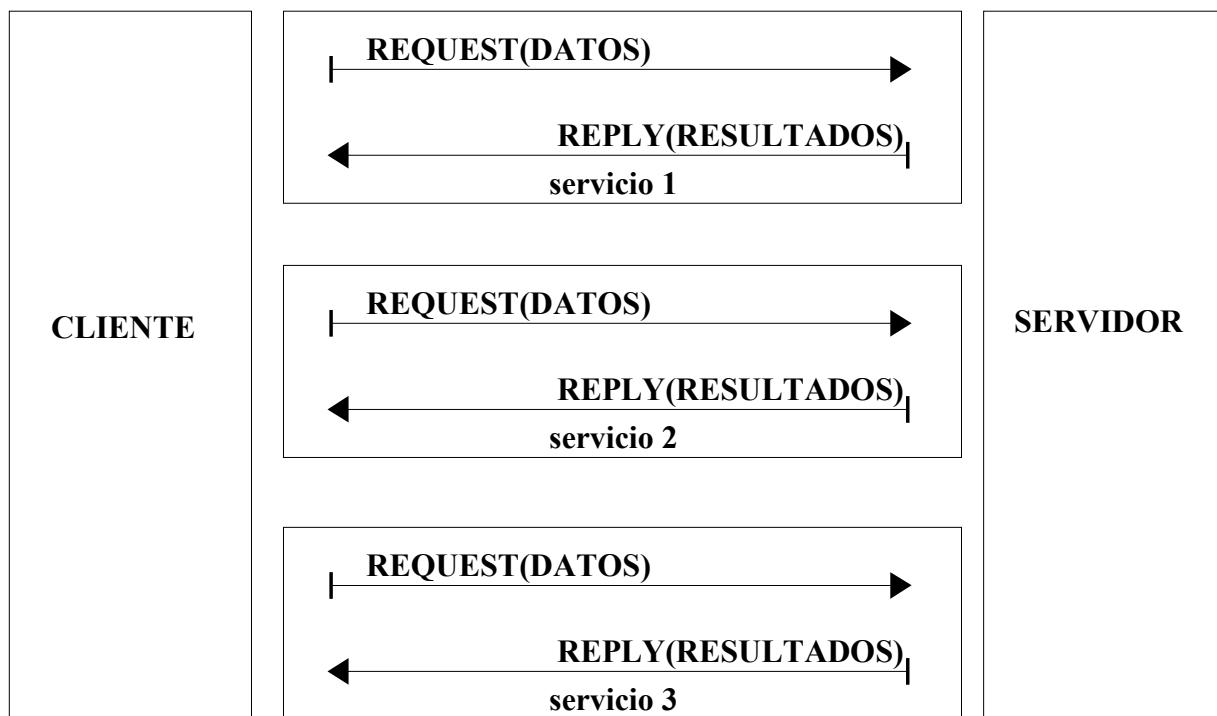
- Un fichero cabecera llamado *proto.h* que contiene definiciones comunes de constantes y macros.
- El código fuente del "stub" del cliente, *protoc.c*
- El código fuente del "stub" del servidor, *protos.c*
- El fichero fuente de rutinas XDR, *protox.c*

Clase 03-10-2006: REMOTE PROCEDURE CALL – RPC:

Concentrémosnos en la **similitud encontrada con una función matemática**. Podemos interpretar a cada **PAR SOLICITUD – RESPUESTA como partes integrantes de un SERVICIO**.

Cada **SERVICIO** estará entonces **formado por un PAR SOLICITUD(DATOS) – RESPUESTA(RESULTADOS)**.

Veamos el siguiente esquema:



Ahora **si vemos a los**

SERVICIOS OFRECIDOS POR EL PROCESO SERVIDOR

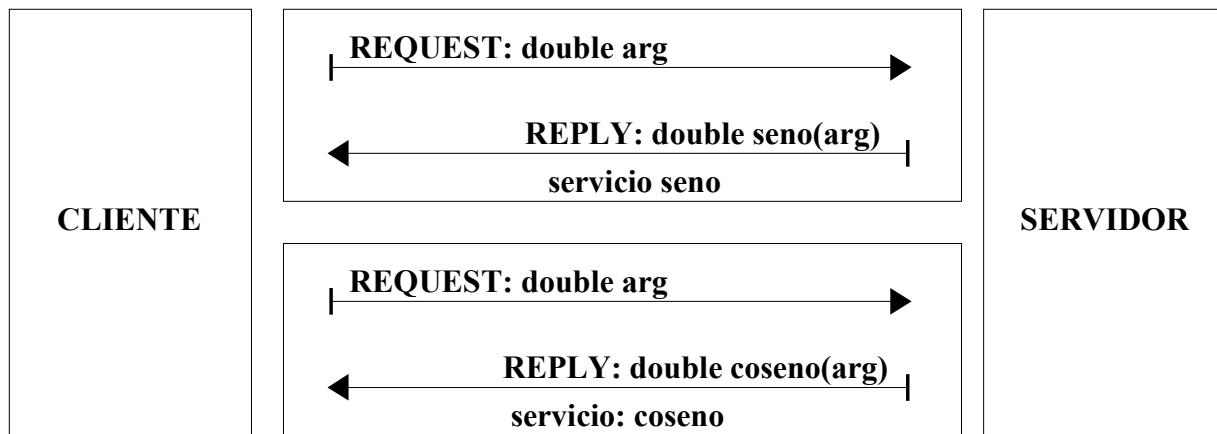
como **FUNCIONES** ya sean matemáticas, o mejor aún, implementaciones de estas mismas en un lenguaje de programación,
entonces

podemos **DESCRIBIR AL SERVIDOR USANDO FUNCIONES**, al menos desde el punto de vista de la comunicación.

Comencemos entonces a materializar todas estas ideas mediante **un ejemplo** concreto. Tratemos de describir un **PROCESO SERVIDOR DE FUNCIONES TRIGONOMÉTRICAS (por ahora, seno y coseno)**.

Clase 03-10-2006: REMOTE PROCEDURE CALL – RPC:

Un **esquema del PAR DE PROCESOS CLIENTE – SERVIDOR** correspondiente a este ejemplo, puede ser el siguiente:



En **SUNRPC** describimos esto así, en un archivo llamado **trig.x** escrito en un **lenguaje de descripción de la interfaz (IDL)** que define los métodos exportados por el SERVIDOR:

```
program TRIG {  
version VTRIG {  
    double seno(double)=1;  
    double coseno(double)=2;  
}=1;  
}=0x80000001;
```

Este archivo se puede editar **con cualquier editor**, como por ejemplo el **vi**.

Lo que sigue es, utilizar el comando **rpcgen**, un compilador para el protocolo **RPC**, de la siguiente manera:

```
rpcgen -N -a trig.x
```

Este comando **creará los siguientes archivos nuevos, en el directorio de trabajo actual:**

Makefile.trig	// archivo para la utilidad de compilación: make
trig_client.c	// código fuente en C para el CLIENTE
trig_clnt.c	// código fuente en C del stub del CLIENTE
trig.h	// Archivo de cabecera
trig_server.c	// código fuente en C para el SERVIDOR
trig_svc.c	// código fuente en C del stub del SERVIDOR

Nosotros vamos a modificar el archivo: trig_server.c para adecuarlo a nuestro modelo.

Clase 03-10-2006: REMOTE PROCEDURE CALL – RPC:

En el archivo: **trig_server.c**

las siguientes líneas “en negrita” son las que se deben añadir:

```
#include<math.h>
double *
seno_1_svc(double * arg1, struct svc_req * rqstp )
{
    static double result;
result = sin(arg1);
    return &result;
}
double *
coseno_1_svc(double * arg1, struct svc_req * rqstp)
{
    static double result;
result = cos(arg1);
    return &result;
}
```

Luego **guardamos los cambios y compilamos de la siguiente manera:**

```
gcc -Wall trig_server.c trig_svc.c -lmc -o trig_server.out
```

Ya tenemos listo el SERVIDOR, ahora debemos proseguir con el CLIENTE.

En el archivo: **trig_client.c**

las siguientes líneas “en negrita” son las que se deben añadir:

```
/*
 * 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 "trig.h"
#include<stdlib.h> //añadido
```

Clase 03-10-2006: REMOTE PROCEDURE CALL – RPC:

```
/* Continua:      trig_client.c */
void
trig_1(char *host, double d)
{
    CLIENT *clnt;
    double *result_1;
    double seno_1_arg1 = d;
    double *result_2;
    double coseno_1_arg1 = d;

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

    result_1 = seno_1(seno_1_arg1, clnt);
    if (result_1 == (double *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    printf("seno(%f) = %f \n", d, *result_1);
    result_2 = coseno_1(coseno_1_arg1, clnt);
    if (result_2 == (double *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    printf("coseno(%f) = %f \n", d, *result_2);

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

Clase 03-10-2006: REMOTE PROCEDURE CALL – RPC:

```
/* Continua:      trig_client.c */
int
main (int argc, char *argv[])
{
    char *host;
    /* argv[0] <--> nombre_archivo_ejecutable */
    if (argc < 2) {
        printf ("Usage: %s server_host angulo\n", argv[0]);
        exit (1);
    }
    host = argv[1];    /* argv[1] <--> servidor */
    trig_1 (host, atof(argv[2])); /* argv[2] <--> angulo */
return 0;
}
```

Luego **guardamos los cambios y compilamos de la siguiente manera:**

```
gcc -Wall trig_client.c trig_clnt.c -lm -o trig_client.out
```

Solo nos resta ejecutar este PAR CLIENTE – SERVIDOR RPC.

Por ejemplo:

Podemos iniciar el PROCESO SERVIDOR RPC de la siguiente manera:

```
./trig_server.out&
```

Mediante el “&” se le indica que se EJECUTE EN SEGUNDO PLANO.

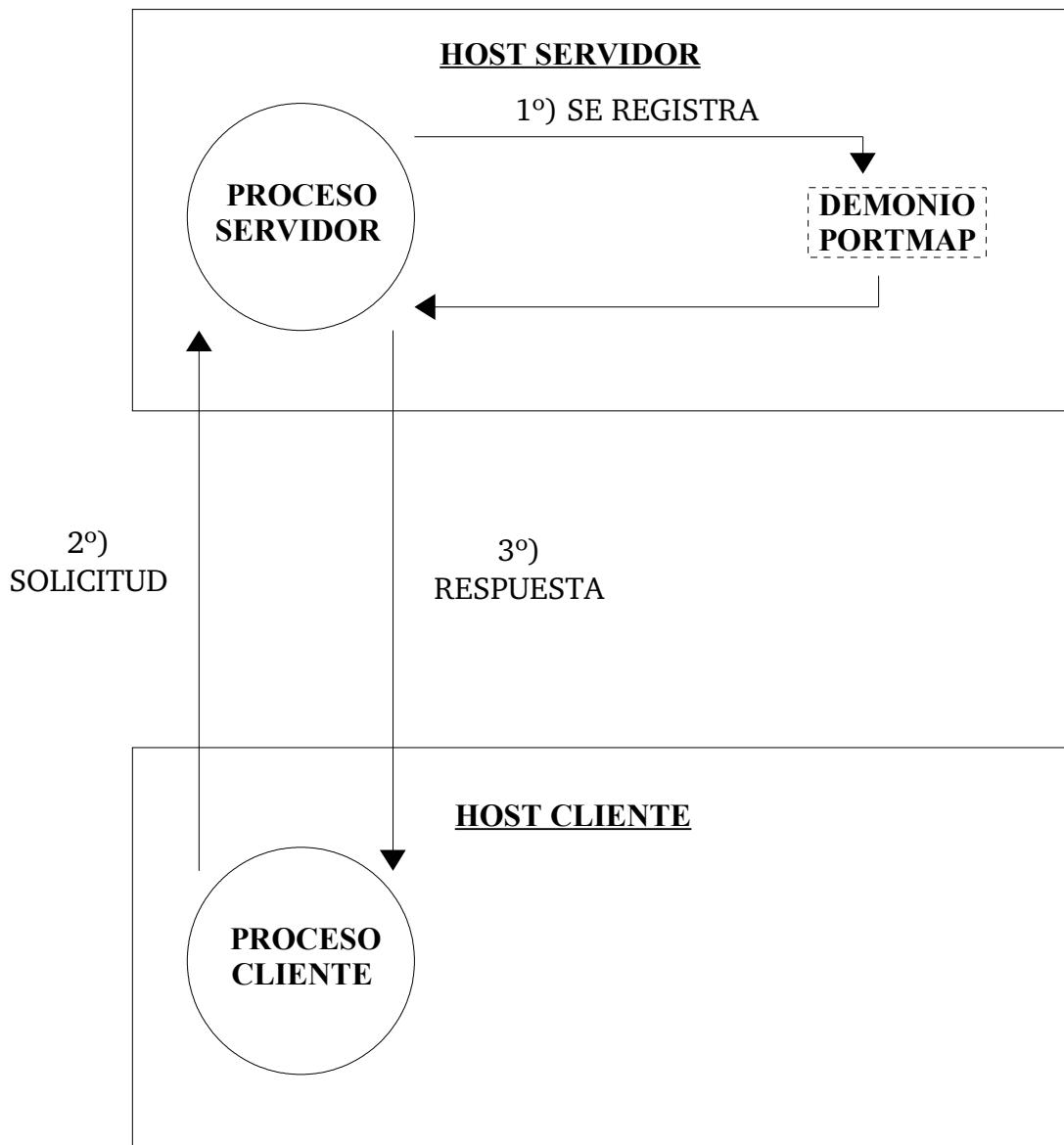
Ahora podemos iniciar varias instancias consecutivas, una después de que termina la otra, mediante:

```
for i in 0, 0.1, 0.2, 0.3; do
    ./trig_client.out localhost $i ;
done;
```

Así ejecuta la siguiente serie de llamadas sucesivas:

```
./trig_client.out localhost 0
./trig_client.out localhost 0.1
./trig_client.out localhost 0.2
./trig_client.out localhost 0.3
```

Clase 03-10-2006: REMOTE PROCEDURE CALL – RPC:
¿Qué ocurre cuando se ejecuta un PAR CLIENTE – SERVIDOR RPC?



Vemos que **usar el SERVICIO REMOTO es como usar UNA FUNCIÓN DE C “COMÚN”**. No parece que estuvieramos comunicándonos con un protocolo TCP/IP, aunque en realidad así es.

Pero...

¿Hasta dónde podemos estirar esta analogía?

Pues podremos usar hasta PUNTEROS, ESTRUCTURAS y UNIONES.

Clase 10-10-2006: REMOTE PROCEDURE CALL – RPC:

El lenguaje que se usa para RPC es parecido a C.

Veamos el siguiente ejemplo:

```
/* Archivo: q.x --- SINTAXIS INCORRECTA */
program Q {
    version VQ {
        double * f(int * );
    }=1;
} =0x20000001;
```

Al compilarlo mediante:

```
rpcgen -N -a q.x
```

Se obtiene el siguiente mensaje de error:

```
ehv80@hostneo:/cliente-servidor-RPC-Q$ rpcgen -N -a q.x
double * f(int * );
^^^^^^^^^^^^^^^^^
q.x, line 3: expected 'identifier'
```

Esto se debe a que **en RPC no se pueden usar tipos compuestos**, como por ejemplo: **int *** , es decir, “**puntero a entero**”

Entonces **para evitar este mensaje de error** debemos **utilizar las sentencias para crear SINÓNIMOS**, los **typedef**, como se muestra a continuación:

```
/* Archivo: q.x --- SINTAXIS CORRECTA */
typedef double * PDouble;
typedef int * PInt;
program Q {
    version VQ {
        PDouble f(PInt )=1;
    }=1;
} =0x20000001;
```

Luego al compilarlo mediante: **rpcgen -N -a q.x**

Se obtienen, en el mismo directorio de trabajo, **los siguientes archivos**:

Makefile.q	//para la utilidad de compilación make
q_client.c	//código fuente en C para el CLIENTE RPC
q_cInt.c	//código fuente en C para el stub del CLIENTE RPC
q.h	//Archivo de cabecera

Clase 10-10-2006: REMOTE PROCEDURE CALL – RPC:

q_server.c //código fuente en C para el SERVIDOR RPC
q_svc.c //código fuente en C del stub del SERVIDOR RPC
q_xdr.c //código fuente en C con rutinas para la
//representación externa de datos
//Es un "Marshaller", un serializador

De entre ellos **editamos** el archivo: **q_server.c**

A continuación se muestra **en negrita** el código fuente agregado.

```
/*
 * 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 "q.h"

PDouble *
f_1_svc(PInt arg1, struct svc_req *rqstp)
{
    static PDouble result;

    /* inserted server code here */
    static double d;
    d = *arg1;
    d += 0.5;
    result = &d;

    return &result;
}
```

Lo compilamos mediante:

```
gcc -Wall q_server.c q_svc.c q_xdr.c -o q_server
```

Clase 10-10-2006: REMOTE PROCEDURE CALL – RPC:

Con lo cual se obtiene el siguiente mensaje de advertencia:

**q_xdr.c: En la función `xdr_PDouble':
q_xdr.c:11: aviso: unused variable `buf'
q_xdr.c: En la función `xdr_PInt':
q_xdr.c:21: aviso: unused variable `buf'**

Ahora editamos: **q_client.c**

En el siguiente código **se muestra en negrita las modificaciones añadidas.**

```
/*
 * 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 "q.h"
void
q_1(char *host)
{
    CLIENT *clnt;
    PDouble *result_1;
    PInt f_1_arg1;
    /* código adicionado */
    static int i;
    i = 47;
    f_1_arg1 = &i;
#ifndef DEBUG
    clnt = clnt_create (host, Q, VQ, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif /* DEBUG */
```

Clase 10-10-2006: REMOTE PROCEDURE CALL – RPC:

```
/* Continua:    q_client.c */
    result_1 = f_1(f_1_arg1, clnt);
    if (result_1 == (PDouble *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    printf("llega %f \n", **result_1); /* agregado */
#ifndef DEBUG
    clnt_destroy (clnt);
#endif /* DEBUG */
}
int
main (int argc, char *argv[])
{
    char *host;

    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    q_1 (host);
    exit (0);
}
```

Lo compilamos con:

```
gcc -Wall q_client.c q_clnt.c q_xdr.c -o q_client
```

Con lo cual se obtiene el siguiente mensaje de advertencia:

```
q_xdr.c: En la función `xdr_PDouble':
q_xdr.c:11: aviso: unused variable `buf'
q_xdr.c: En la función `xdr_PInt':
q_xdr.c:21: aviso: unused variable `buf'
```

Luego ejecutamos el servidor en segundo plano: ./q_server&

Y el cliente: ./q_client localhost

Clase 10-10-2006: REMOTE PROCEDURE CALL – RPC:

Veamos un ejemplo un poco mas dificil. Vamos a implementar un **PAR CLIENTE SERVIDOR RPC que manipule la ESTRUCTURA DE DATOS conocida como ARBOL BINARIO.**

Editamos el archivo: **tree.x**

```
/* tree.x */

typedef struct Nodo * PNodo;

struct Nodo{
    int valor;
    PNodo izq;
    PNodo der;
};

program TREE{
    version VTREE{
        void pone(int)=1;
        PNodo arbol()=2;
        =1;
    }=0x20000001;
}
```

Lo **compilamos** mediante: **rpcgen -N -a tree.x**

Así se generan los siguiente archivos:

Makefile.tree	//archivo para la utilidad de compilación make
tree_client.c	//Código en C del CLIENTE RPC
tree_clnt.c	//Código en C del stub del CLIENTE RPC
tree.h	//Archivo de cabecera
tree_server.c	//Código en C del SERVIDOR RPC
tree_svc.c	//Código en C del stub del SERVIDOR RPC
tree_xdr.c	//código en C con rutinas para la //representación externa de datos

De entre ellos **editamos** el archivo: **tree_server.c**

En negrita se muestra el código en C añadido.

Clase 10-10-2006: REMOTE PROCEDURE CALL – RPC:

```
/*      Archivo: tree_server.c
 * 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 "tree.h"
static PNodo raiz = NULL;
PNodo inserta( int arg, PNodo n)
{
    if( n == NULL )
    {
        n = (PNodo)malloc(sizeof( *n ));
        n->valor = arg;
        n->izq = n->der = NULL;
    }
    else if(arg < n->valor) n->izq = inserta(arg, n->izq);
    else if(arg > n->valor) n->der = inserta(arg, n->der);
    return n;
}
void *
pone_1_svc(int arg1, struct svc_req *rqstp)
{
    static char * result;
    raiz = inserta(arg1, raiz);
    return (void *) &result;
}
PNodo *
arbol_1_svc(struct svc_req *rqstp)
{
    static PNodo result;
    result = raiz;
    return &result;
} /* FIN:      tree_server.c */
```

Clase 10-10-2006: REMOTE PROCEDURE CALL – RPC:

Lo compilamos mediante:

```
gcc -Wall tree_server.c tree_svc.c tree_xdr.c -o tree_server
```

Obteniéndose:

```
tree_xdr.c: En la función `xdr_PNodo':  
tree_xdr.c:11: aviso: unused variable `buf'  
tree_xdr.c: En la función `xdr_Nodo':  
tree_xdr.c:21: aviso: unused variable `buf'
```

Continuamos ahora **editando** el archivo: **tree_client.c**

En negrita se muestran las modificaciones hechas.

Archivo: tree_client.c

```
/*  
 * 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 "tree.h"  
  
void recorre(PNodo n)  
{  
    if(n!=NULL)  
    {  
        recorre(n->izq);  
        printf("%d ", n->valor);  
        recorre(n->der);  
    }  
}
```

Clase 10-10-2006: REMOTE PROCEDURE CALL – RPC:

```
/* Continua: tree_client.c */
void
tree_1(char *host)
{
    CLIENT *clnt;
    void *result_1;
    int pone_1_arg1;
    PNodo *result_2;

#ifndef DEBUG
    clnt = clnt_create (host, TREE, VTREE, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif /* DEBUG */
    int i;
    for(i=0 ; i<1000 ; i++ )
    {
        pone_1_arg1 = 1+(int) (1000.0*rand()/(RAND_MAX+1.0));
        result_1 = pone_1(pone_1_arg1, clnt);
        if (result_1 == (void *) NULL) {
            clnt_perror (clnt, "call failed");
        }
    }
    result_2 = arbol_1(clnt);
    if (result_2 == (PNodo *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    recorre(*result_2);

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

Clase 10-10-2006: REMOTE PROCEDURE CALL – RPC:

```
/* Continua: tree_client.c */
int
main (int argc, char *argv[])
{
    char *host;
    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    tree_1 (host);
    exit (0);
}
/* FIN: tree_client.c */
```

Lo compilamos mediante:

```
gcc -Wall tree_client.c tree_clnt.c tree_xdr.c -o tree_client
```

Obteniéndose:

```
tree_xdr.c: En la función `xdr_PNodo':
tree_xdr.c:11: aviso: unused variable `buf'
tree_xdr.c: En la función `xdr_Nodo':
tree_xdr.c:21: aviso: unused variable `buf'
```

Ahora vamos a **ejecutar este PAR CLIENTE – SERVIDOR RPC** mediante:

- **./tree_server&** Para **iniciar el SERVIDOR en segundo plano**
- **./tree_client localhost** Para **iniciar el CLIENTE**

Para **analizar lo que sucede al ejecutar este PAR DE PROCESOS CLIENTE – SERVIDOR RPC**, veamos la **SALIDA ESTANDAR del PROCESO CLIENTE**:

Clase 10-10-2006: REMOTE PROCEDURE CALL – RPC:

Una ejecución arroja los siguientes números en dicha pantalla:

2 3 4 5 6 7 10 14 15 17 18 19 21 24 29 30 33 35 36 37 39 40 41 44 45
48 49 50 51 52 53 54 55 57 59 63 64 65 69 70 71 73 74 75 77 79 82
8587 88 92 93 94 96 97 98 99 100 101 102 103 104 105 106 108 109
110 112 113 114 115 120 122 123 126 127 130 132 134 135 137 138
139 140 142 143 144 148 149 152 153 155 156 157 158 161 163 164
165 166 167 168 169 170 173 174 176 177 178 179 181 182 183 184
185 187 188 189 190 191 192 193 194 196 198 199 200 201 203 204
205 206 208 211 212 215 216 219 220 222 223 224 225 226 227 228
229 230 231 232 233 234 239 240 241 243 245 247 248 249 251 257
258 259 261 262 263 265 266 267 270 271 273 274 275 277 278 281
282 283 284 285 288 289 291 292 293 294 295 297 298 302 305 306
308 309 312 313 314 316 317 322 324 325 326 327 328 329 330 331
332 333 334 335 336 337 339 342 344 345 348 349 351 353 355 357
358 360 361 362 363 364 365 367 369 374 375 376 378 379 381 382
383 384 385 386 388 392 394 395 399 400 401 402 403 406 407 411
413 414 417 427 428 432 433 434 435 437 438 439 440 441 443 444
447 448 450 451 453 457 458 459 461 462 463 464 466 467 468 470
471 472 474 476 477 478 480 481 482 483 484 486 488 490 493 494
495 496 497 498 499 503 504 507 508 511 513 514 515 516 517 518
519 521 522 523 524 525 526 527 528 529 530 531 532 533 537 539
540 547 548 549 552 553 554 555 556 557 558 561 562 564 567 568
574 577 578 579 580 583 584 585 588 589 590 591 592 593 594 595
596 597 598 599 600 601 602 604 605 607 608 609 610 612 613 615
618 619 620 622 623 624 625 626 627 628 629 630 631 634 636 638
639 640 641 642 643 645 646 647 648 649 650 651 652 654 655 656
657 658 659 660 662 664 665 667 668 669 672 674 675 676 677 678
680 682 684 685 686 687 688 693 696 698 699 700 701 704 707 708
709 712 713 715 718 719 720 721 722 725 728 730 731 733 736 737
738 739 741 744 746 747 748 750 753 754 756 758 760 761 762 764
765 767 768 769 770 771 772 774 775 776 778 779 781 783 784 786
787 788 790 794 795 796 798 799 800 802 804 805 806 808 809 810
813 814 815 816 819 820 821 822 823 827 828 829 830 831 832 833
834 837 838 839 840 841 844 845 846 848 849 850 851 853 858 859
862 865 866 868 869 870 871 874 875 878 879 880 881 882 883 884
885 886 888 889 890 891 892 893 894 898 900 901 902 903 904 905
907 909 910 911 912 913 914 916 917 919 920 921 922 923 924 925
926 927 928 931 932 933 934 935 936 937 938 940 944 945 946 947
948 949 950 951 953 957 959 960 962 964 966 968 971 972 973 980
982 984 985 987 993 995 996 998 999 1000

Esto significa que el **CLIENTE RPC** se encarga de generar **1000 números enteros pseudo-aleatorios** (ver **man 3 rand**) que son “insertados”, de a uno por vez y en forma ordenada, en el **ARBOL BINARIO** que mantiene el **SERVIDOR RPC**. Luego el **CLIENTE RPC** muestra todos los nodos del **ARBOL BINARIO** que reside en el **SERVIDOR RPC**.

Clase 17-10-2006: REMOTE PROCEDURE CALL – RPC:

Ahora resolvamos el siguiente ejercicio:

Hacer un programa “cubilete” para un solo dado. Debe ofrecer un servicio, que sin argumentos, retorne un entero entre uno y seis.

Sugerencia: usar random(3)

Comencemos **editando el archivo**, en lenguaje IDL para RPC: **cub.x**

```
/*      cub.x
 * CUBILETE DE UN DADO.
 * GENERA UN NÚMERO ENTRE UNO Y SEIS.
 */
```

```
program CUB {
    version VCUB{
        int tirada()=1;
        }=1;
    }=0x20000002;
/* FIN: cub.x */
```

Debemos **procesarlo mediante: rpcgen -N -a cub.x**

Obteniéndose los siguientes archivos:

```
cub_client.c //Código C del CLIENTE RPC
cub_clnt.c   //Código C del STUB del CLIENTE RPC (NO EDITAR)
cub.h         //Archivo de CABECERA
cub_server.c //Código C del SERVIDOR RPC
cub_svc.c    //Código C del STUB del SERVIDOR RPC (NO EDITAR)
Makefile.cub //Archivo para la utilidad de compilación make
```

Clase 17-10-2006: REMOTE PROCEDURE CALL – RPC:

A continuación **editamos el archivo: cub_server.c**

Se muestra en negrita las modificaciones realizadas.

```
/* Archivo: cub_server.c
 * 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 "cub.h"
int *
tirada_1_svc(struct svc_req *rqstp)
{
    static int result;
    /* agregado */
    result = ((random() % 6) + 1);
    return &result;
}
/* FIN del archivo: cub_server.c */
```

Luego editamos el archivo: **cub_client.c**

En negrita se denotan las modificaciones hechas.

```
/* Archivo: cub_client.c
 * 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 "cub.h"
```

Clase 17-10-2006: REMOTE PROCEDURE CALL – RPC:

```
void                      /* Continua: cub_client.c */
cub_1(char *host)
{
    CLIENT *clnt;
    int *result_1;

#ifndef DEBUG
    clnt = clnt_create (host, CUB, VCUB, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif /* DEBUG */
    result_1 = tirada_1(clnt);
    if (result_1 == (int *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    printf("RESULTADO DE LA TIRADA DE UN DADO: %d\n",
           *result_1);

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

int main (int argc, char *argv[])
{
    char *host;
    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    cub_1 (host);
    exit (0);
}      /* Fin del archivo: cub_client.c */
```

Clase 17-10-2006: REMOTE PROCEDURE CALL – RPC:

Ahora **solo nos falta compilar:**

- EL **SERVIDOR RPC** mediante:

```
gcc -Wall cub_server.c cub_svc.c -o cub_server
```

- EL **CLIENTE RPC** haciendo:

```
gcc -Wall cub_client.c cub_clnt.c -o cub_client
```

Entonces **ahora podemos ejecutarlo** de la siguiente manera, **por ejemplo, queremos hacer 10 tiradas consecutivas del dado:**

- **Iniciamos el PROCESO SERVIDOR RPC, en segundo plano con:**

```
./cub_server&
```

- **Iniciamos 10 veces sucesivas, el PROCESO CLIENTE RPC, uno después que finaliza el otro, mediante:**

```
for x in 1, 2, 3, 4, 5, 6, 7, 8, 9, 10;  
do ./cub_client localhost;  
done;
```

- **Traemos a primer plano el SERVIDOR RPC con:** fg

- **FINALIZAMOS EL SERVIDOR RPC mediante:** CTRL+C

Ahora vamos a complicar un poco más este sencillo ejercicio:

**En lugar de tirar un dado a la vez queremos
tirar 5 dados a la vez con nuestro cubilete**

Debemos notar que **nuestro servicio ahora deberá manejar un conjunto de 5 enteros**, en lugar de uno solo, es decir, **tenemos que manejar:**

ARREGLOS EN EL LENGUAJE IDL PARA RPC
para lograr nuestro cubilete con varios datos.

Clase 17-10-2006: REMOTE PROCEDURE CALL – RPC:

Debemos recordar que se deben definir mediante **typedef**, pues el **IDL de RPC no soporta tipos de datos compuestos como por ejemplo: int ***, en español, “**punteros a enteros**”.

En RPC hay tres clases de arreglos:

a) arreglos fijos: por ejemplo **typedef int dados[5];**

b) arreglos variables de hasta 'n' elementos:

por ejemplo, arreglo de hasta 5 enteros **typedef int dados<5>;**

c) arreglos variables con una cantidad arbitraria de elementos:

por ejemplo **typedef int dados<>;**

El rpcgen(1):

- **usa un lenguaje de descripción del servicio** que se llama **IDL** (**Interface Description Language**).
- **emplea el PREPROCESADOR DE C, por eso entonces podemos utilizar** **#include** y **#define**

Vamos a **editar un nuevo archivo:** **cub-varios-dados.x**

```
/*      cub-varios-dados.x
 * CUBILETE DE VARIOS DADOS.
 * GENERA VARIOS NÚMEROS ENTRE UNO Y SEIS, EN CADA
 * TIRADA.          */
%#include <stdlib.h>
/*Se incluye en todos los archivos '.c' y '.h' que genere rpcgen(1)*/
typedef int dados<5>;
/* arreglo de hasta 5 enteros */
program CUBVARIOSDADOS {
    version VCUBVARIOSDADOS{
        dados tirada()=1;
        }=1;
        }=0x20000003;
/* Fin del archivo: cub-varios-dados.x */
```

Clase 17-10-2006: REMOTE PROCEDURE CALL – RPC:

Compilamos esta descripción en IDL de RPC con:

rpcgen -N -a cub-varios-dados.x

Obteniéndose los siguientes archivos:

cub-varios-dados_client.c //Código C del CLIENTE RPC

cub-varios-dados_clnt.c //Código C del STUB del CLIENTE RPC
//(NO EDITAR)

cub-varios-dados.h //Archivo de CABECERA

cub-varios-dados_server.c //Código C del SERVIDOR RPC

cub-varios-dados_svc.c //Código C del STUB del SERVIDOR RPC
//(NO EDITAR)

cub-varios-dados_xdr.c //Código C con la REPRESENTACIÓN
//EXTERNA DE LOS DATOS

Makefile.cub-varios-dados //Archivo para la utilidad de compilación make

Si observamos el archivo:
encontraremos que:

cub-varios-dados.h

- el **arreglo que se definió en el archivo: cub-varios-dados.x**
como

typedef int dados<5>;

- aparece en el archivo:
como
- typedef struct {**
- | | |
|-----------------------------|------------------------------------|
| u_int dados_len; | /* cantidad de elementos */ |
| int *dados_val; | /* base del arreglo */ |
- } dados;**

Ahora **vamos a modificar el archivo:** **cub-varios-dados_server.c**

Vea que en negrita se denotan las modificaciones.

Clase 17-10-2006: REMOTE PROCEDURE CALL – RPC:

```
/* Archivo: cub-varios-dados_server.c
 * 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 "cub-varios-dados.h"
#include <stdlib.h>
dados *
tirada_1_svc(struct svc_req *rqstp)
{
    static dados result;
    int i;
    static int d[5];
    for( i=0 ; i < 5 ; i++ )
        d[i] = ( (random() % 6) + 1 );
    result.dados_val = d; /* BASE DEL ARREGLO */
    result.dados_len = 5; /* LONGITUD DEL ARREGLO */
    return &result;
}
/* Fin del archivo: cub-varios-dados_server.c */
```

Nos falta modificar el archivo:

cub-varios-dados_client.c

Se muestran en negrita las modificaciones realizadas.

Clase 17-10-2006: REMOTE PROCEDURE CALL – RPC:

```
/* Archivo: cub-varios-dados_client.c
 * 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 "cub-varios-dados.h"
#include <stdlib.h>
void
cubvariosdados_1(char *host)
{
    CLIENT *clnt;
    dados *result_1;
#ifndef DEBUG
    clnt = clnt_create (host, CUBVARIOSDADOS, VCUBVARIOSDADOS, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif /* DEBUG */
    result_1 = tirada_1(clnt);
    if (result_1 == (dados *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    /* agregado */
    int i;
    /* int a[5]; DABA ERROR TIPOS INCOMPATIBLES */
    int * a;
    for( i=0 ; i<5 ; i++){
        a = (result_1->dados_val); /* BASE DEL ARREGLO */
        printf("TIRADA DEL DADO Nº %d VALE: %d\n", (i+1), a[i]);
    }
#endif /* DEBUG */
    clnt_destroy (clnt);
#endif /* DEBUG */
```

Clase 17-10-2006: REMOTE PROCEDURE CALL – RPC:

```
/* Continua: cub-varios-dados_client.c */
int
main (int argc, char *argv[]) udp      0      0 *:32791          *:*
4683/cub-varios-dad
{
    char *host;
    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    cubvariosdados_1 (host);
exit (0);
}
/* Fin del archivo: cub-varios-dados_client.c */
```

Solo nos resta compilar el PAR CLIENTE – SERVIDOR RPC:

- al SERVIDOR RPC mediante:

```
gcc
-Wall  cub-varios-dados_server.c  cub-varios-dados_svc.c
           cub-varios-dados_xdr.c
-o  cub-varios-dados_server
```

- al CLIENTE RPC con:

```
gcc
-Wall  cub-varios-dados_client.c  cub-varios-dados_clnt.c
           cub-varios-dados_xdr.c
-o  cub-varios-dados_client
```

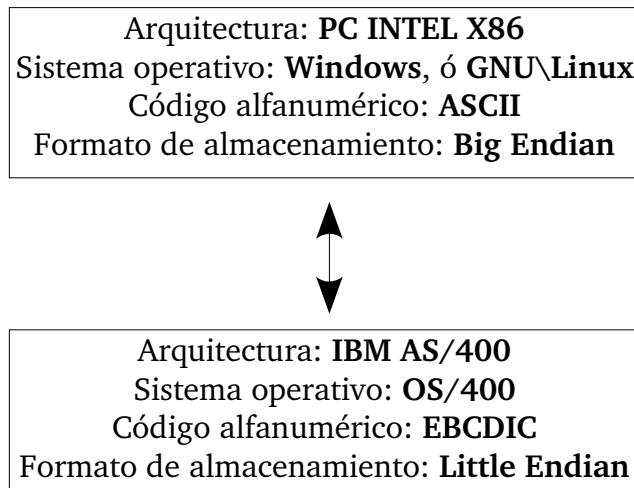
Luego podemos ejecutar:

- al SERVIDOR RPC, en segundo plano: ./cub-varios-dados_server&
- al CLIENTE RPC: ./cub-varios-dados_client localhost

Clase 31-10-2006: REMOTE PROCEDURE CALL – RPC:

RPC canoniza los datos para enviarlos y los descanoniza al recibirlos.

Esto **permite** lograr una **transmisión coherente de datos entre distintas arquitecturas con distintos sistemas operativos**, como se muestra en el siguiente esquema:



Esto es muy bueno, hasta que debemos enviar algo sin canonizar, como por ejemplo, una imagen. Para esto, se puede usar un arreglo que ``opaque`` y haga creer que no está canonizado.

Veamos ahora cómo declarar: **UNIONES EN RPC**

No son como las UNIONES DE C, sino que se parecen, en su espíritu, a los VARIANTS RECORDS de PASCAL.

Por ejemplo:

```
type R = record
    case b : bool of
        true : i : integer
        false : r : real
    end;
```

En C se puede simular este tipo así:

```
struct R{
    int b;      /* tag o etiqueta */
    Union{
        int i;
        double r;
    } u;
};
```

Clase 31-10-2006: REMOTE PROCEDURE CALL – RPC:

Las UNIONES EN RPC sirven para indicar errores. Por ejemplo:

Union U

```
switch(int error){  
    case 0: opaque peli[  
    default;  
};
```

Veamos un ejemplo. Vamos a editar **la Descripción de la Interfaz del Servicio RPC** en el archivo: **union.x**

```
/* Archivo: union.x */  
/* Ejemplo de uso de UNIONES EN RPC */  
  
union U switch(int flag){  
    case 1: int i;  
    case 2: double d;  
    default: void;  
};  
  
program UNION {  
    version VUNION {  
        U funcion(int)=1;  
        }=1;  
        }=0x20000004;  
/* Fin del archivo: union.x */
```

Lo procesamos con: **rpcgen -N -a union.x**

y obtenemos los siguientes archivos:

Makefile.union //Archivo para la utilidad de compilación make
union_client.c //Código en C del CLIENTE RPC
union_clnt.c //Código en C del STUB del CLIENTE RPC
union.h //Archivo de CABECERA
union_server.c //Código en C del SERVIDOR RPC
union_svc.c //Código en C del STUB del SERVIDOR RPC
union_xdr.c //Código en C para la DEFINICIÓN de lo DATOS
//EXTERNOS, o sea donde se canonizan los datos

Clase 31-10-2006: REMOTE PROCEDURE CALL – RPC:

Observemos que en el archivo de cabecera union.h aparece:

```
struct U {  
    int flag;  
    union {  
        int i;  
        double d;  
    } U_u;  
};  
typedef struct U U;
```

que luce muy similar a los VARIANTS RECORDS de PASCAL como habíamos dicho antes.

Editamos entonces el archivo del SERVIDOR RPC en C: union_server.c

Se muestran en negrita las modificaciones realizadas.

```
/* Archivo: union_server.c  
 * 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 "union.h"  
U *  
funcion_1_svc(int arg1, struct svc_req *rqstp)  
{  
    static U result;  
    /* EN CASO DE QUE RESTO DE DIVISION ENTERA POR 3  
     * SEA: 0 ==> ASIGNA: result.flag = 1  
     * SEA: 1 ==> ASIGNA: result.flag = 2  
     * SEA: 2 ==> ASIGNA: result.flag = 0  
    */
```

Clase 31-10-2006: REMOTE PROCEDURE CALL – RPC:

```
/* Continua: union_server.c */
switch(arg1 % 3){

    case 0:
        result.U_u.i = 10;
        result.flag = 1; /* ? */
        break;

    case 1:
        result.U_u.d = 3.14;
        result.flag = 2; /* ? */
        break;

    case 2:
        result.flag = 0; /* ? */
        break;

}
return &result;
}

/*Fin del archivo: union_server.c */
```

Seguimos con la **edición del archivo correspondiente al CLIENTE RPC:**
union_client.c

Se denotan en negrita las modificaciones hechas.

```
/* Archivo: union_client.c
 * 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 "union.h"
```

Clase 31-10-2006: REMOTE PROCEDURE CALL – RPC:

```
/* Continua: union_client.c */
void
union_1(char *host)
{
    CLIENT *clnt;
    U *result_1;
    int funcion_1_arg1;

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

for(funcion_1_arg1=0 ; funcion_1_arg1 < 10 ; funcion_1_arg1++){
    result_1 = funcion_1(funcion_1_arg1, clnt);
    if (result_1 == (U *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    switch( result_1->flag ){
        /* case 0: */
        case 1:
            printf("Un entero! vale: %d \n", result_1->U_u.i);
            break;
        /* case 1: */
        case 2:
            printf("Un doble! vale: %f \n", result_1->U_u.d);
            break;
        default:
            printf("Auuch!!! ESTO NO ES NADA!!! \n");
            break;
    }
}
```

Clase 31-10-2006: REMOTE PROCEDURE CALL – RPC:

/* Continua: union_client.c */

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

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

    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    union_1 (host);
exit (0);
}
/* Fin del archivo: union_client.c */
```

Nos falta compilar:

- el SERVIDOR RPC mediante:

gcc -Wall union_server.c union_svc.c union_xdr.c -o union_server

- el CLIENTE RPC mediante:

gcc -Wall union_client.c union_clnt.c union_xdr.c -o union_client

Y para ejecutar:

- el SERVIDOR RPC, en segundo plano: **./union_server&**

- el CLIENTE RPC, en la propia computadora: **./union_client localhost**

Clase 31-10-2006: REMOTE PROCEDURE CALL – RPC:

Continuemos ahora con un nuevo ejemplo, en el cuál

“trataremos de hacer un CLIENTE RPC pero SIN USAR RPC”.

La **descripción del servicio RPC en lenguaje IDL** se muestra a continuación en el archivo: **qq.x**

```
/* Archivo: qq.x */
/* DESCRIPCIÓN DE UN SERVICIO RPC EN IDL */

program QQ{
    version VQQ{
        double f(int) = 1;
        }=11;
    }=0x30000003;
/* Fin del archivo: qq.x */
```

Lo procesamos mediante: **rpcgen -N -a qq.x**

Y obtenemos los archivos:

Makefile.qq	//Archivo para la utilidad de compilación make
qq_client.c	//Código en C del CLIENTE RPC
qq_clnt.c	//Código en C del STUB del CLIENTE RPC
qq.h	//Archivo de CABECERA
qq_server.c	//Código en C del SERVIDOR RPC
qq_svc.c	//Código en C del STUB del SERVIDOR RPC

Editamos el archivo del SERVIDOR RPC: qq_server.c

En negrita se muestran las modificaciones.

```
/* Archivo: qq_server.c
 * 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 "qq.h"
```

Clase 31-10-2006: REMOTE PROCEDURE CALL – RPC:

```
/* Continua: qq_server.c */
double *
f_11_svc(int arg1, struct svc_req *rqstp)
{
    static double result;
    result = arg1 + (1.0 * arg1)/100;
    return &result;
}
/* Fin del archivo: qq_server.c */
```

Lo compilamos mediante:

```
gcc -Wall qq_server.c qq_svc.c -o qq_server
```

Entonces eliminamos el código en C del CLIENTE RPC y del STUB del CLIENTE RPC mediante:

```
rm qq_client.c qq_clnt.c
```

Y ahora nos dedicamos a hacer nuestro propio CLIENTE RPC rápido:

```
/* Archivo: qq_client.c
 * NUESTRO PROPIO CLIENTE RPC
 * PARA EL SERVIDOR RPC: qq_server.c */

/* ARCHIVOS DE CABECERA */
#include<stdio.h>
#include<rpc/rpc.h>
#include<rpc/xdr.h>

/* DEFINICIONES */
#define NPROG 0x30000003
#define NVERS 11
#define NFUNC 1
```

Clase 31-10-2006: REMOTE PROCEDURE CALL – RPC:

```
/* Continua: qq_client.c */
/* FUNCION PRINCIPAL MAIN */
int main()
{
    int i;
    double d;
    for(i=0 ; i<10 ; i++){
        callrpc(
            "localhost",
            NPROG,
            NVERS,
            NFUNC,
            xdr_int,      /* CANONIZA A UN DATO DE TIPO int */
            &i,
            xdr_double,
            /* CANONIZA A UN DATO DE TIPO double */
            &d
        );
        printf("Para %d viene %f \n", i , d );
    }
    return 0;
}
/* Archivo: qq_client.c */
```

Lo compilamos mediante: **gcc -Wall qq_client.c -o qq_client**

Entonces:

- ejecutamos el SERVIDOR RPC, en segundo plano mediante :
./qq_server&
- ejecutamos nuestro CLIENTE RPC con:
./qq_client

Clase 14-11-2006: REMOTE PROCEDURE CALL – RPC:

Apartándose del modelo REQUEST/REPLY:

REQUEST/REPLY es un modelo secuencial, pero a veces el paralelismo es necesario.

Es una **tentación usar PThreads**.

Es una **MALA idea usar: las funciones generadas por rpcgen(1) porque no son reentrantes.**

Una posible solución es **no usar threads sino fork**. Listo! Ponemos un **fork(2)** en nuestras funciones y el programa ``revienta''.

En realidad hay que meter mano en los archvivos ***_svc.c** (que dicen ``**DO NOT EDIT**'').

Para ver dónde debemos tocar y si dá resultado, **vamos a generar el PAR CLIENTE – SERVIDOR para la siguiente descripción del servicio RPC: delay.x**

```
/* Archivo: delay.x */  
/* DESCRIPCIÓN DE SERVICIO RPC EN LENGUAJE IDL */
```

```
program DELAY{  
    version VDELAY{  
        void demora(void)=1;  
        }=1;  
        }=0x20000002;  
    /* Fin del archivo: delay.x */
```

Lo procesamos con: **rpcgen -N -a delay.x**

Que genera los siguientes archivos:

```
delay_client.c //Código en C del CLIENTE RPC  
delay_clnt.c //Código en C del STUB del CLIENTE RPC  
delay.h //Archivo de CABECERA  
delay_server.c //Código en C del SERVIDOR RPC  
delay_svc.c //Código en C del STUB del SERVIDOR RPC  
Makefile.delay //Archivo para la utilidad de compilación make
```

Clase 14-11-2006: REMOTE PROCEDURE CALL – RPC:

Editamos el archivo: delay_server.c

En negrita están las modificaciones.

```
/* Archivo: delay_server.c
 * 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 "delay.h"
#include<unistd.h>
void *
demora_1_svc(struct svc_req *rqstp)
{
    static char * result;
sleep(10);
    return (void *) &result;
}
/*Fin del archivo: delay_server.c */
```

Lo compilamos con:

```
gcc -Wall delay_server.c delay_svc.c -o delay_server
```

Sin realizar cambios en el CLIENTE RPC lo compilamos:

```
gcc -Wall delay_client.c delay_clnt.c -o delay_client
```

Ahora ejecutamos el SERVIDOR RPC en segundo plano con:

```
./delay_server&
```

Y desde 5 terminales virtuales diferentes ejecutamos 5 instancias distintas del CLIENTE RPC, lo más rápido posible, mediante:

```
./delay_client localhost
```

Observará que el SERVIDOR RPC atiende a dos CLIENTES RPC mientras que en los 3 CLIENTES RPC restantes se observa el mensaje:

```
``call failed: RPC: Timed out``
```

Clase 14-11-2006: REMOTE PROCEDURE CALL – RPC:

Podemos paralelizar al SERVIDOR RPC para ello debemos editar el archivo:

delay_svc.c

```
/* Archivo: delay_svc.c
 * Please do not edit this file.
 * It was generated using rpcgen.
 */
```

```
#include "delay.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

static void *
_demora_1 (void *argp, struct svc_req *rqstp)
{
    return (demora_1_svc(rqstp));
}

static void
delay_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
    union {
        int fill;
    } argument;
```

Clase 14-11-2006: REMOTE PROCEDURE CALL – RPC:

```
/* Continua: delay_svc.c */
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 demora:
_xdr_argument = (xdrproc_t) xdr_void;
_xdr_result = (xdrproc_t) xdr_void;
local = (char *(*)(char *, struct svc_req *)) _demora_1;
break;
default:
    svcerr_noproc (transp);
    return;
}
memset ((char *)&argument, 0, sizeof (argument));
if (!svc_getargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)){
    svcerr_decode (transp);
    return;
}
result = (*local)((char *)&argument, rqstp);
if (result != NULL && !svc_sendreply(transp, (xdrproc_t) _xdr_result, result)) {
    svcerr_systemerr (transp);
}
if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
    fprintf (stderr, "%s", "unable to free arguments");
    exit (1);
}
return;
}
```

Clase 14-11-2006: REMOTE PROCEDURE CALL – RPC:

```
/* Continua: delay_svc.c */
int main (int argc, char **argv)
{
    register SVCXPRT *transp;

    pmap_unset (DELAY, VDELAY);

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL) {
        fprintf (stderr, "%s", "cannot create udp service.");
        exit(1);
    }
    if (!svc_register(transp, DELAY, VDELAY, delay_1, IPPROTO_UDP)) {
        fprintf (stderr, "%s", "unable to register (DELAY, VDELAY, udp).");
        exit(1);
    }

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL) {
        fprintf (stderr, "%s", "cannot create tcp service.");
        exit(1);
    }
    if (!svc_register(transp, DELAY, VDELAY, delay_1, IPPROTO_TCP)) {
        fprintf (stderr, "%s", "unable to register (DELAY, VDELAY, tcp).");
        exit(1);
    }

    svc_run ();
    fprintf (stderr, "%s", "svc_run returned");
    exit (1);
    /* NOTREACHED */
}

/* Fin del archivo: delay_svc.c */
```

Clase 14-11-2006: REMOTE PROCEDURE CALL – RPC:

En negrita se muestra la sección de código que se debería modificar.
Es la parte más importante del STUB (o SKELETON) del SERVIDOR RPC.
En ella se hace uso de un 'switch' sobre un miembro de la estructura de datos que se muestra a continuación EN NEGRITA:

DEL ARCHIVO: /usr/include/rpc/svc.h

...

```
/*
 * Service request
 */
struct svc_req {
    rpcprog_t rq_prog;          /* Número del PROCESO SERVIDOR*/
    rpcvers_t rq_vers;          /* VERSIÓN DEL PROCESO SERVIDOR */
rpcproc_t rq_proc;          /* SOLICITUD QUE SE PIDE */
    struct opaque_auth rq_cred; /* credenciales crudas de la conexión*/
    caddr_t rq_clntcred;        /* credenciales preparadas para "solo lectura" */
    SVCXPRT *rq_xprt;           /* asa del transporte asociado al PROCESOSERVIDOR */
};
```

Entonces “SEGÚN SEA LA SOLICITUD QUE SE PIDE...” se realiza la acción correspondiente a cada 'case':.

Hasta aquí son las NOTAS DE CLASE.

Ahora vamos a ver los ENUNCIADOS del TRABAJO FINAL QUE HAY QUE PRESENTAR:

- PRIMER TRABAJO PRIMER:

USANDO SOCKETS

CON EL PROTOCOLO: IP Y

EL SUB-PROTOCOLO: UDP (USER DATAGRAM PROTOCOL)

Realizar un PORTFORWARDER QUE RESPETE LA SIGUIENTE FORMA DE INVOCACIÓN:

./port-forwarder-udp port_in port_out ipaddres_out

- SEGUNDO TRABAJO SEGUNDO:

USANDO RPC con la siguiente DESCRIPCIÓN DE LA INTERFACE DEL SERVICIO (IDL - SunRPC/XDR)

```
typedef string cadena<>;
/* arreglo de cadena de caracteres de longitudarbitraria */

program RELAY{
    version VRELAY{
        void manda(cadena, cadena)=1;
        cadena retira(cadena)=2;
    }=1;
}=0x20000009;
```

Comencemos con el “**PRIMER TRABAJO PRIMER**”

Si leyeron hasta acá, entonces tienen el conocimiento suficiente para usar la API de sockets que nos ofrece el sistema GNU/Linux, o bien siguen los siguientes pasos que son los que yo seguí:

VAMOS A HACER **UN PAR CLIENTE – SERVIDOR** que usa el **PROTOCOLO IP con el SUBPROTOCOLO UDP** y opera en **MODO NO BLOQUEANTE** gracias a la utilización de las funciones **select(2)** y **fcntl(2)**.

1º) Editamos el archivo: cliente-udp-con-select-fcntl-setsockopt.c

Que lo encuentran en la página 198 de este documento.

Lo compilan como se detalla.

2º) Editan el archivo: servidor-udp-con-select-fcntl-setsockopt.c

Que lo encuentran en la página 202 de este documento.

Lo compilan como se detalla.

3º) Editan el archivo: portforwarder-udp-select-fcntl-setsockopt.c

PRIMER TRABAJO PRIMER:

/* Archivo: portforwarder-udp-select-fcntl-setsockopt.c

* Toma como **argumentos del programa** para línea de comandos:

- * - PUERTO DE ENTRADA
- * - PUERTO DE SALIDA
- * - IP DE DESTINO

*

* **Abre un socket UDP ('sockio', NO ORIENTADO A CONEXIÓN).**

*

* **Se utilizan dos estructuras sockaddr_in:**

*

- * - 'sini' tiene como puerto al '**PUERTO DE ENTRADA'**
y como dirección a '**INADDR_ANY'**
(IP local de la propia computadora)

*

- * - 'sino' tiene como puerto al '**PUERTO DE SALIDA'**
y como dirección a '**IP DE DESTINO'**

*

* Mediante **fcntl(2)** usa la **operación 'F_SETFL'** para **asignar a la bandera del descriptor sockio el valor 'O_NONBLOCK'** para funcionar en **MODO NO BLOQUEANTE.**

*

* Mediante **setsockopt(2)** aplica al **conector 'sockio'** la **opción del nivel 'SOL_SOCKET'** con el valor '**SO_REUSEADDR**' para evitar mensaje '**'Address already in use'**

* **Verifica con select(2):**

*

* **Si existen datagramas de entrada los recibe por el conector 'sockio' y la dirección-puerto 'sini', entonces los envía mediante el socket 'sockio' y la dirección-puerto 'sino' a la 'IP DE DESTINO'**

*/

PRIMER TRABAJO PRIMER:

/* Continua: portforwarder-udp-select-fcntl-setsockopt.c */

/* ARCHIVOS DE CABECERA */

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<sys/time.h>
#include<sys/select.h>
#include<fcntl.h>
```

/* DEFINICIONES */

```
#define MAX(x,y) ((x)>(y) ? (x) : (y))
#define SIZE 1024
#define TIME 3600
```

/* SINONIMOS*/

```
typedef struct sockaddr *sad;
```

/* FUNCIONES */

```
void error(char *s){
    perror(s);
    exit(-1);
}
```

/* FUNCION PRINCIPAL MAIN */

```
int main(int argc, char ** argv){
    if(argc < 4 || argc > 4){
        fprintf(stderr, "Usa: %s port_in port_out ipaddr_destiny \n",argv[0]);
        exit(-1);
}
```

PRIMER TRABAJO PRIMER:

/* Continua: portforwarder-udp-select-fcntl-setsockopt.c */

const int yes = 1; //Para setsockopt(2)

int sockio, cuanto, largo, recibidos;

struct sockaddr_in sini, sino;

// ver: man 3 byteorder

unsigned short int port_in, port_out;

if(argv[1])

{

 port_in = (unsigned short int)atoi(argv[1]);

 printf("\nport_in: %d \n", port_in);

}

else

 error("error: port_in");

if(argv[2])

{

 port_out = (unsigned short int)atoi(argv[2]);

 printf("port_out: %d \n", port_out);

}

else

 error("error: port_out");

if(argv[3])

{

 // Familia de direcciones para 'sino'

 sino.sin_family = AF_INET;

 // Puerto para 'sino'

 sino.sin_port = htons(port_out);

 // Dirección IP para 'sino'

PRIMER TRABAJO PRIMER:

/* Continua: portforwarder-udp-select-fcntl-setsockopt.c */

```
inet_pton(AF_INET, argv[3], &sino.sin_addr);
/* POSIX TO NETWORK, ver man 3 inet_pton :
 * transforma argv[3]
 * (la direccion IP pasada como argumento)
 * desde formato ascii
 * (puntero a cadena de caracteres)
 * al formato network, guarda en la
 * struct in_addr sino.sin_addr que a su
 * vez es miembro de la
 * struct sockaddr_in sino .
 * Se trata de la direccion IP del host
 * al que quiere enviar datagramas,
 * y pertenece a la familia de direcciones
 * 'AF_INET'
 */
}

else
    error("error: ipaddr_destiny bad address");

//Buffer de 1024 bytes (caracteres)
char linea[SIZE];

/* Conjuntos de descriptores de ficheros 'in'
 * e 'in_orig' que supervisa select(2) */
fd_set in, in_orig;

/* Estructura que guarda el tiempo límite
 * hasta que retorne select(2) */
struct timeval tv;
```

PRIMER TRABAJO PRIMER:

```
/* Continua: portforwarder-udp-select-fcntl-setsockopt.c */
    //abre un conector UDP 'sockio'
    if((sockio = socket(PF_INET, SOCK_DGRAM, 0)) < 0 )
        error("socket UDP");

    /* Familia de direcciones de 'sockio'
     * con dirección-puerto 'sini' */
    sini.sin_family = AF_INET;
    /* Puerto, con bytes en orden de red,
     * para 'sockio' con dirección-puerto 'sini' */
    sini.sin_port = htons(port_in);
    /* Dirección de internet, con bytes en orden de red,
     * para 'sockio' */
    sini.sin_addr.s_addr = INADDR_ANY;

    // Asigna al conector 'sockio' la bandera 'O_NONBLOCK'
    if( fcntl(sockio, F_SETFL, O_NONBLOCK) < 0 )
        error("fcntl");

    /* Mediante setsockopt(2) aplica al conector 'sockio'
     * la opción del nivel 'SOL_SOCKET' con el valor
     * 'SO_REUSEADDR' para evitar mensaje
     * 'Address already in use' */
    if( setsockopt(sockio,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof yes)<0)
        error("setsockopt");

    //publica dirección-puerto 'sini', del conector 'sockio'
    if( bind(sockio, (sad)&sini, sizeof sini) < 0 )
        error("bind");
```

PRIMER TRABAJO PRIMER:

```
/* Continua: portforwarder-udp-select-fcntl-setsockopt.c */
printf("\nESCUCHANDO EN DIRECCIÓN LOCAL[ %s ] : PUERTO LOCAL[ %d ]\n",
       inet_ntoa(sini.sin_addr),
       ntohs(sini.sin_port));

/* Tiene select(2) */

// Limpia el conjunto de descriptores de ficheros 'in_orig'
FD_ZERO(&in_orig);
// Añade 'sockio' al conjunto de descriptores 'in_orig'
FD_SET(sockio, &in_orig);

/* Tiene 1 hora */
// Tiempo hasta que select(2) retorne: 3600 segundos
tv.tv_sec = TIME;
tv.tv_usec = 0;
for(;){
    // copia conjunto 'in_orig' en 'in'
    memcpy(&in, &in_orig, sizeof in);

    /* Espera a ver si se reciben datagramas por 'sockio'
     * con dirección-puerto 'sini'*/
    if((cuanto = select( MAX(0,sockio)+1, &in, NULL, NULL, &tv ) ) < 0 )
        error("select: error");
    // Si el tiempo de select(2) termina -> error timeout
    if(cuanto == 0)
        error("select: timeout");

    largo = sizeof sini;
```

PRIMER TRABAJO PRIMER:

```
/* Continua: portforwarder-udp-select-fcntl-setsockopt.c */

    /* Si hay para leer desde el conector 'sockio' con
     * dirección-puerto 'sini' */

    if(FD_ISSET(sockio, &in)) {
        /* Recibe hasta 1024 caracteres de 'sockio'
         * y los pone en el buffer 'linea' */

        if((recibidos = recvfrom(sockio, linea, SIZE, 0, (sad)&sini, &largo )) < 0)
            error("recvfrom");
        else if(recibidos == 0) // Si recepción devuelve 0 -> parar
            break;
        // Marca el fin del buffer con '0' (CARACTER NULO)
        // linea[recibidos]=0;
        /* Imprime en pantalla un aviso de que llega un datagrama
         * a la dirección-puerto local 'sini' */
        printf("\nA LA DIRECCIÓN LOCAL[ %s ] : PUERTO LOCAL[ %d ]\n",
LLEGA DATAGRAMA\n",
                inet_ntoa(sini.sin_addr),
                ntohs(sini.sin_port));
        // Imprime el mensaje recibido
        // printf("%s \n",linea);
        /* Envía contenido de 'linea' mediante 'sockio' con
         * la dirección-puerto 'sino' a la 'IP DE DESTINO' */
        if(sendto(sockio, linea, sizeof linea, 0, (sad)&sino, largo ) < 0 )
            error("sendto");

        /* Imprime en pantalla un aviso de que se envía un datagrama desde la
         * dirección local 'sini.sin_addr' por el puerto local 'sino.sin_port'
         * ('port_out') hacia la dirección remota 'sino.sin_addr' ('IP DE DESTINO')
        */
        printf("\nDATAGRAMA ENVIADO DESDE LA DIRECCIÓN LOCAL[ %s ] : PUERTO LOCAL[ %d ] A LA DIRECCIÓN REMOTA [ %s ] \n",
                inet_ntoa(sini.sin_addr),
                ntohs(sino.sin_port),
                inet_ntoa(sino.sin_addr));
    }

} //CIERRA EL for
```

PRIMER TRABAJO PRIMER:

```
/* Continua: portforwarder-udp-select-fcntl-setsockopt.c */
    close(sockio); // Cierra el conector UDP 'sockio'
    return 0; // Finalización exitosa
}
/* Fin Archivo: portforwarder-udp-select-fcntl-setsockopt.c */
```

Solo falta compilarlo con:

```
gcc -Wall      portforwarder-udp-select-fcntl-setsockopt.c
      -o          portforwarder-udp-select-fcntl-setsockopt
```

Y Para ejecutarlo deben respetar la siguiente forma de invocación:

```
./portforwarder-udp-select-fcntl-setsockopt port_in port_out ipdest
```

Eso es todo lo que necesitan para hacer el PRIMER TRABAJO PRIMER.

Ahora pasemos al SEGUNDO TRABAJO SEGUNDO:

1º) Editan el archivo que contiene la DESCRIPCIÓN DE LA INTERFACE DEL SERVICIO RPC (IDL – SunRPC – XDR) denominado: relay.x

```
* Archivo: relay.x                                */
/* Descripción de la Interface (en IDL) del Servicio RPC      */
/* RELAY RPC QUE NO CONSIDERA CADUCIDAD DE MENSAJES           */
/* SINTAXIS:                                                 */
/*     void         manda( cadena_ip ,   cadena_mensaje )*/
/*     cadena_mensaje retira( cadena_ip )                   */
typedef string cadena<>;
/* arreglo de cadena de caracteres de longitud arbitraria */
program RELAY{
    version VRELAY{
        void manda(cadena, cadena)=1;
        cadena retira(cadena)=2;
    }=1;
}=0x20000009;
/* Archivo: relay.x                                */
```

SEGUNDO TRABAJO SEGUNDO

2º) Procesan este archivo con el compilador de lenguaje RPC:

rpcgen -N -a ./relay.x

De este modo obtienen los siguientes archivos:

Makefile.relay //Para la utilidad de compilación make
relay_client.c //Código en C del CLIENTE RPC
relay_clnt.c //Código en C del STUB del CLIENTE RPC
relay.h //Archivo de CABECERA
relay_server.c //Código en C del SERVIDOR RPC
relay_svc.c //Código en C del SKELETON del SERVIDOR RPC
relay_xdr.c /* Código en C con funciones XDR que

*** CODIFICAN/DECODIFICAN los datos en el FORMATO DE**

*** REPRESENTACIÓN EXTERNA DE LOS DATOS**

*** (eXternal Data Representation) */**

3º) Modifican el archivo: relay.h para que quede como el siguiente:

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

```
#ifndef _RELAY_H_RPCGEN
#define _RELAY_H_RPCGEN
```

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

```
#ifdef __cplusplus
extern "C" {
#endif
```

```
typedef char *cadena;
```

SEGUNDO TRABAJO SEGUNDO

```
/* Continua: relay.h */
struct manda_1_argument {
    cadena arg1;
    cadena arg2;
};

typedef struct manda_1_argument manda_1_argument;

#define RELAY 0x20000009
#define VRELAY 1
#define SIZE 1024           /* <.. agregado */

#if defined(__STDC__) || defined(__cplusplus)
#define manda 1
extern void * manda_1(cadena , cadena , CLIENT *);
extern void * manda_1_svc(cadena , cadena , struct svc_req *);
#define retira 2
extern cadena * retira_1(cadena , CLIENT *);
extern cadena * retira_1_svc(cadena , struct svc_req *);
extern int relay_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);

#else /* K&R C */
#define manda 1
extern void * manda_1();
extern void * manda_1_svc();
#define retira 2
extern cadena * retira_1();
extern cadena * retira_1_svc();
extern int relay_1_freeresult ();
#endif /* K&R C */
```

SEGUNDO TRABAJO SEGUNDO

/* Continua: relay.h */

/* the xdr functions */

```
#if defined(__STDC__) || defined(__cplusplus)
extern bool_t xdr_cadena (XDR *, cadena*);
extern bool_t xdr_manda_1_argument (XDR *, manda_1_argument*);
```

#else /* K&R C */

```
extern bool_t xdr_cadena ();
extern bool_t xdr_manda_1_argument ();
```

#endif /* K&R C */

```
#ifdef __cplusplus
}
#endif
```

#endif /* !_RELAY_H_RPCGEN */

/* FIN relay.h */

4º) Modifican el archivo relay_server.c para que quede como se muestra a continuación:

/* relay_server.c

```
* 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 "relay.h"

char msg[SIZE]; /* <.. agregado */

SEGUNDO TRABAJO SEGUNDO

```
/* Continua: relay_server.c */
void *
manda_1_svc(cadena arg1, cadena arg2, struct svc_req *rqstp)
{           /*HOST*/    /*MSG*/    /*SOLICITUD RPC*/
    static char * result;

    fprintf(stdout,"[manda_1_svc]...\t"); /* <.. agregado */
    /*msg = arg2; TIPOS INCOMPATIBLES      * <.. agregado */
    strcpy(msg, arg2);                  /* <.. agregado */

    return (void *) &result;
}

cadena *
retira_1_svc(cadena arg1, struct svc_req *rqstp)
{           /*HOST*/    /*SOLICITUD RPC*/
    static cadena result;

    fprintf(stdout, "...[retira_1_svc]\n"); /* <.. agregado */
    if( msg != NULL )                      /* <.. agregado */
        result = msg;                     /* <.. agregado */
    else                                /* <.. agregado */
    {                                     /* <.. agregado */
        fprintf(stderr, "ERROR: BAD MESSAGE\n");/* <.. agregado */
        exit(4);                         /* <.. agregado */
    }                                     /* <.. agregado */
    return &result;
}
/* FIN: relay_server.c */
```

5º) Modifican el relay_svc.c para que quede como:

SEGUNDO TRABAJO SEGUNDO

```
/*      relay_svc.c
 * Please do not edit this file.
 * It was generated using rpcgen.
 */
#include "relay.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

static void *
_manda_1 (manda_1_argument *argp, struct svc_req *rqstp)
{           /* HOST Y MSG */          /* SOLICITUD RPC */

    /* VER EN relay_server.c
     * FUNCIÓN: manda_1_svc */
    return (manda_1_svc(argp->arg1, argp->arg2, rqstp));
}           /*HOST*/   /*MSG*/

static cadena *
_retira_1 (cadena *argp, struct svc_req *rqstp)
{           /*HOST*/   /* SOLICITUD RPC */
    /* VER EN relay_server.c
     * FUNCIÓN: retira_1_svc */
    return (retira_1_svc(*argp, rqstp));
}           /*HOST*/
```

SEGUNDO TRABAJO SEGUNDO

```
/* Continua:      relay_svc.c      */
static void
relay_1(struct svc_req *rqstp, register SVCXPRT *transp)
{   /* SOLICITUD RPC */ /*MANIPULADOR DEL TRANSPORTE*/
    /*DEL LADO DEL SERVIDOR */
union {
    manda_1_argument manda_1_arg; /*SERVER_HOST Y MSG*/
    cadena retira_1_arg;          /*DESTINATION_HOST*/
} argument;
char *result;

/* ARGUMENTOS Y VALOR RETORNO CANONIZADOS
 * EN FORMATO DE REPRESENTACIÓN EXTERNA DE DATOS
 *
 * VER: /usr/include/rpc/
 */
xdrproc_t _xdr_argument, _xdr_result;
/*HOST Y MSG*/ /* MSG */
/* Y */
/* HOST */

char *(*local)(char *, struct svc_req *);
/* local: PUNTERO A UNA FUNCIÓN LOCAL, PERTENECIENTE AL
 * SERVIDOR.
 * ARGUMENTOS QUE TOMA ESA FUNCIÓN:
 * - char *
 * - struct svc_req *
 * VALOR DEVUELTO POR ESA FUNCIÓN:
 * - char *
 *
 * VER ARCHIVO: relay_server.c
 * SIMILITUD CON: retira_1_svc
 */
```

SEGUNDO TRABAJO SEGUNDO

/* Continua: relay_svc.c */

```
/* SOLICITUD DE SERVICIO RPC
 * VER: /usr/include/rpc/svc.h ESTÁ DECLARADA LA
 * struct svc_req {
 *     rpcprog_t rq_prog;           // service program number
 *     rpcvers_t rq_vers;          // service protocol version
 *     rpcproc_t rq_proc;          // the desired procedure
 *     struct opaque_auth rq_cred; // raw creds from the wire
 *     caddr_t rq_clntcred;        // read only cooked cred
 *     SVCXPRT *rq_xprt;           // associated transport
 * };
 */
/*PROCEDIMIENTO*/
switch ( rqstp->rq_proc ) { /*SEGÚN SEA EL PROCEDIMIENTO PEDIDO*/
case NULLPROC:
```

```
    /* LLAMADA A RUTINA DE ATENCIÓN PARA ENVIAR
     * RESULTADOS DE UNA LLAMADA A PROCEDIMIENTO
     * REMOTO */
    (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
    return;
```

/* CASO EN QUE EL PROCEDIMIENTO SOLICITADO ES: manda */

case manda:

```
    _xdr_argument = (xdrproc_t) xdr_manda_1_argument;
    /*VER: relay_xdr.c */
    _xdr_result = (xdrproc_t) xdr_void;
```

```
    local = (char * (*)(char *, struct svc_req *)) _manda_1;
    /* VER MAS ARRIBA */
    /* ASIGNA AL PUNTERO A FUNCION 'local' LA DIRECCIÓN
     * DE LA FUNCIÓN '_manda_1' APLICÁNDOLE UN CAST PARA
     * LOGRAR CONCORDANCIA EN SUS PROTOTIPOS */
    break;
```

SEGUNDO TRABAJO SEGUNDO

```
/* Continua:      relay_svc.c      */
/* CASO EN QUE EL PROCEDIMIENTO SOLICITADO ES: retira */
case retira:
    _xdr_argument = (xdrproc_t) xdr_cadena; /* VER: relay_xdr.c */
    _xdr_result = (xdrproc_t) xdr_cadena; /* VER: relay_xdr.c */

    local = (char * (*)(char *, struct svc_req *)) _retira_1;
    /* ASIGNA AL PUNTERO A FUNCION 'local' LA DIRECCIÓN
     * DE LA FUNCIÓN '_retira_1' APlicandole un CAST PARA
     * LOGRAR CONCORDANCIA EN SUS PROTOTIPOS
    */
    break;

/* CASO EN QUE EL PROCEDIMIENTO SOLICITADO ES: CUALQUIER
   OTRA COSA */
default:
    /* LLAMADA A UNA RUTINA DE ATENCIÓN DE SERVICIOS QUE
     * NO IMPLANTA EL NÚMERO DE PROCEDIMIENTO QUE
     * SOLICITA EL INVOCADOR
    */
    svcerr_noproc (transp);
    return;
}

/*PONE 'sizeof (argument)' BYTES CON EL VALOR '0'
 * EN LA ZONA DE MEMORIA '(char *)&argument'
 */
memset ((char *)&argument, 0, sizeof (argument));

/*Una macro que decodifica los argumentos de una petición RPC
 * asociada con la asa de transporte de un servicio RPC 'transp'.
 * El parámetro '(caddr_t) &argument' es la dirección donde se
 * colocarán los argumentos. '(xdrproc_t) _xdr_argument' es la
 * rutina XDR usada para decodificar los argumentos. Esta rutina
 * devuelve 1 si la decodificación tiene éxito y cero en caso
 * contrario.*/

```

SEGUNDO TRABAJO SEGUNDO

```
/* Continua: relay_svc.c */
if (!svc_getargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)){
    svcerr_decode (transp);
    return;
}

/*INVOCA A LA FUNCIÓN '*local' APUNTADA POR 'local' */
result = (*local)( (char *)&argument , rqstp );

if (result != NULL && !svc_sendreply(transp, (xdrproc_t) _xdr_result,
result)) {
    svcerr_systemerr (transp);
}

/* svc_freeargs: Macro que libera cualquier dato reservado por el
 * sistema RPC/XDR cuando decodificó los argumentos a un
 * procedimiento de servicio usando svc_getargs().*/
if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument, (caddr_t)
&argument)) {
    fprintf (stderr, "%s", "unable to free arguments");
    exit (1);
}
return;

}// FIN de función: relay_1
int
main (int argc, char **argv)
{
    register SVCXPRT *transp;

    pmap_unset (RELAY, VRELAY);

/*CREA SERVIDOR RPC USANDO PROTOCOLO IP-SUBPROTOCOLO UDP */
    transp = svcudp_create(RPC_ANYSOCK);
```

SEGUNDO TRABAJO SEGUNDO

```
/* Continua:      relay_svc.c      */
if (transp == NULL) {
    fprintf (stderr, "%s", "cannot create udp service.");
    exit(1);
}

/* svc_register:
 *
 * Asocia 'RELAY' y 'VRELAY' con
 * el procedimiento de atención de servicio, 'relay_1'.
 *
 * Se establece una correspondencia entre la terna
 * '[RELAY, VRELAY, IPPROTO_UDP]' y 'transp->xp_port' con el
 * servicio portmap local.
 *
 * La rutina svc_register() devuelve uno en caso de éxito
 * y cero en caso contrario.
 */
if (!svc_register(transp, RELAY, VRELAY, relay_1, IPPROTO_UDP)) {
    fprintf (stderr, "%s", "unable to register (RELAY, VRELAY, udp).");
    exit(1);
}

/*CREA SERVIDOR RPC USANDO PROTOCOLO IP-SUBPROTOCOLO TCP */
transp = svctcp_create(RPC_ANYSOCK, 0, 0);
if (transp == NULL) {
    fprintf (stderr, "%s", "cannot create tcp service.");
    exit(1);
}
if (!svc_register(transp, RELAY, VRELAY, relay_1, IPPROTO_TCP)) {
    fprintf (stderr, "%s", "unable to register (RELAY, VRELAY, tcp).");
    exit(1);
}
```

SEGUNDO TRABAJO SEGUNDO

```
/* Continua:      relay_svc.c      */
/* ESPERA POR PETICIONES RPC. CUANDO RECIBE UNA LA
 * ATIENDE MEDIANTE LA RUTINA 'relay_1'
 */
svc_run ();
fprintf (stderr, "%s", "svc_run returned");
//SI svc_run RETORNA HAY ERROR
exit (1);
/* NOTREACHED */
}
/* FIN relay_svc.c      */
```

6º) Modifican el archivo: relay_clnt.c para que luzca como:

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

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

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

void *
manda_1(cadena arg1, cadena arg2, CLIENT *clnt)
{      /*HOST*/      /*MSG*/
    manda_1_argument arg;//ARGUMENTOS A CANONIZAR
    static char clnt_res;
```

```
    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    arg.arg1 = arg1;      //HOST
    arg.arg2 = arg2;      //MSG
```

SEGUNDO TRABAJO SEGUNDO

```
/* Continua: relay_clnt.c      */
/* LLAMADA A PROCEDIMIENTO REMOTO: manda
 * VER ARCHIVO: relay_svc.c
 */
if (clnt_call (clnt, manda, (xdrproc_t) xdr_manda_1_argument, (caddr_t)
&arg,
    (xdrproc_t) xdr_void, (caddr_t) &clnt_res,
    TIMEOUT) != RPC_SUCCESS) {
    return (NULL);
}
fprintf(stdout, "[manda_1]···\t"); /* <.. agregado */
return ((void *)&clnt_res);
}

cadena *
retira_1(cadena arg1, CLIENT *clnt)
{
    /*HOST*/
    static cadena clnt_res;

    memset((char *)&clnt_res, 0, sizeof(clnt_res));

/* LLAMADA A PROCEDIMIENTO REMOTO: retira
 * VER ARCHIVO: relay_svc.c
 */
if (clnt_call (clnt, retira,
    (xdrproc_t) xdr_cadena, (caddr_t) &arg1,
    (xdrproc_t) xdr_cadena, (caddr_t) &clnt_res,
    TIMEOUT) != RPC_SUCCESS) {
    return (NULL);
}
fprintf(stdout,"···[retira_1]\n"); /* <.. agregado */
return (&clnt_res);
/*&MSG*/
}/* FIN relay_clnt.c */
```

SEGUNDO TRABAJO SEGUNDO

7º) Modifican el archivo: relay_client.c para que quede como:

```
/*      relay_client.c
 * 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 "relay.h"

/* (3) AGREGADOS LOS ARGUMENTOS: 'msg' y 'destinationhost' */
void
relay_1(char * serverhost , char * destinationhost, char * msg)
{
    CLIENT *cInt;
    void *result_1;
    cadena manda_1_arg1 = serverhost;      //SERVER_HOST
    cadena manda_1_arg2 = msg;                //MSG_ENVIAR
    cadena *result_2;                          //MSG_RECIBIR
    cadena retira_1_arg1 = destinationhost; //DESTINATION_HOST

#ifndef DEBUG
    //CREA CLIENTE RPC
    cInt = clnt_create (serverhost, RELAY, VRELAY, "udp");
    if (cInt == NULL) {
        clnt_pcreateerror (serverhost);
        exit (1);
    }
#endif /* DEBUG */
```

SEGUNDO TRABAJO SEGUNDO

```
/* Continua: relay_client.c */
//LLAMADA A PROCEDIMIENTO REMOTO: manda_1
//VER ARCHIVO: relay_clnt.c
result_1 = manda_1(manda_1_arg1, manda_1_arg2, clnt);
if (result_1 == (void *) NULL) {
    clnt_perror (clnt, "call failed");
}

//LLAMADA A PROCEDIMIENTO REMOTO: retira_1
//VER ARCHIVO: relay_clnt.c
result_2 = retira_1(retira_1_arg1, clnt);
if (result_2 == (cadena *) NULL) {
    clnt_perror (clnt, "call failed");
}
fprintf(stdout, "MENSAJE: \n %s \n", *result_2 ); /*<-- agregado */

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

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

    if (argc < 3 || argc > 3) {
        printf ("usage: %s server_host destination_host \n", argv[0]);
        exit (1);
    }

    char * serverhost;
    char * destinationhost;
/* 2da IMPLEMENTACIÓN: MSG_ENVIAR LO LEE DE STDIN
 * LO GUARDA EN UN BUFFER TEMPORAL Y LO ENVÍA */
}
```

SEGUNDO TRABAJO SEGUNDO

```
/* Continua: relay_client.c */
char msg[SIZE];

if( argv[1] != NULL )
    serverhost = argv[1];
else
{
    fprintf(stderr, "ERROR: BAD SERVER_HOST!!!\n");
    exit(2);
}
if( argv[2] != NULL )
    destinationhost = argv[1];
else
{
    fprintf(stderr, "ERROR: BAD DESTINATION_HOST!!!\n");
    exit(2);
}

fprintf(stdout, "INTRODUZCA EL MENSAJE A ENVIAR:\n");
if( fgets(msg, SIZE, stdin) == NULL )
{
    fprintf(stderr, "ERROR: BAD MESSAGE!!!\n");
    exit(2);
}
relay_1 (serverhost, destinationhost, msg);
/* (1)AGREGADO: ARGUMENTO 'msg' */
/* (2)AGREGADO: ARGUMENTO 'destinationhost' */
exit (0);
}
/* FIN relay_client.c */
```

8º) Observen el archivo relay_xdr.c

SEGUNDO TRABAJO SEGUNDO

```
/* relay_xdr.c
```

```
* Please do not edit this file.
```

```
* It was generated using rpcgen.
```

```
*/
```

```
#include "relay.h"
```

/* FUNCIÓN DE CANONIZACIÓN DE DATOS.

*** *xdr_cadena:***

*** TOMA UN 'cadena *' Y LO TRANSFORMA EN**

*** 'XDR *'.**

*** DEVUELVE:**

*** TRUE EN CASO DE EXITO**

*** FALSE SI HUBO ERROR**

```
**/
```

```
bool_t
```

```
xdr_cadena (XDR *xdrs, cadena *objp)
```

```
{
```

```
    register int32_t *buf;
```

/* FUNCIÓN DE CANONIZACIÓN DE DATOS: *xdr_string*

*** VER ARCHIVO: /usr/include/rpc/xdr.h**

```
*
```

*** OPERADOR UNARIO DE COMPLEMENTO A UNO: ~**

```
*/
```

```
    if (!xdr_string (xdrs, objp, ~0))
```

```
        return FALSE;
```

```
    return TRUE;
```

```
}
```

SEGUNDO TRABAJO SEGUNDO

/* Continua: relay_xdr.c */

```
/* FUNICÓN DE CANONIZACIÓN DE DATOS.  
 * xdr_manda_1_argument:  
 * TOMA UN 'manda_1_argument' Y LO TRANSFORMA EN  
 * 'XDR'.  
 * DEVUELVE:  
 *   TRUE  EN CASO DE EXITO  
 *   FALSE SI HUBO ERROR  
 */  
bool_t  
xdr_manda_1_argument (XDR *xdrs, manda_1_argument *objp)  
{  
    if (!xdr_cadena (xdrs, &objp->arg1))  
        return FALSE;  
    if (!xdr_cadena (xdrs, &objp->arg2))  
        return FALSE;  
    return TRUE;  
}  
/* FIN relay_xdr.c */
```

9º) Compilan el SERVIDOR RPC MEDIANTE:

gcc -Wall relay_server.c relay_svc.c relay_xdr.c -o relay_server

10º) Compilan el CLIENTE RPC MEDIANTE:

gcc -Wall relay_client.c relay_clnt.c relay_xdr.c -o relay_client

11º) Ejecutan el SERVIDOR RPC MEDIANTE:

./relay_server

12º) Ejecutan el CLIENTE RPC MEDIANTE:

./relay_client RPC_SERVER_HOST RPC_DESTINATION_HOST