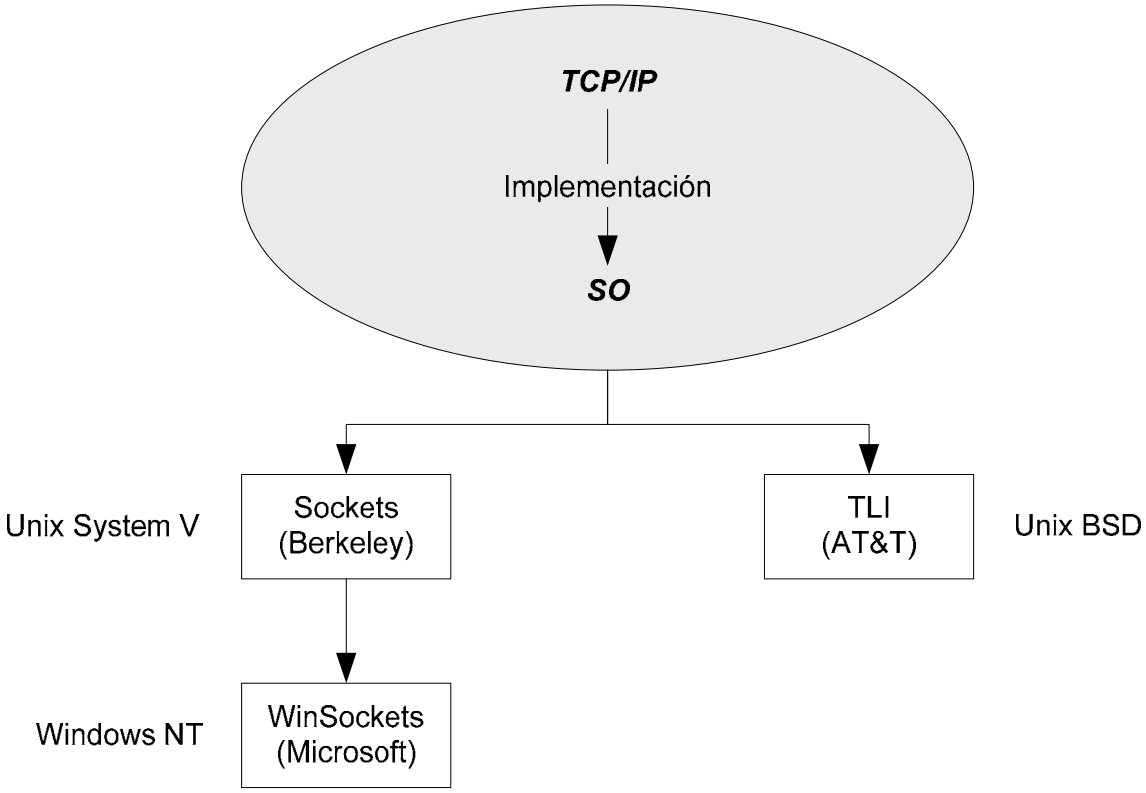
Índice

1. Introducción

Historia

Cuando se estandarizaron los protocolos TCP e IP, había que implementarlos en los diferentes sistemas operativos existentes en el mercado. Inicialmente surgieron dos implementaciones: Sockets (Berkeley) y TLI (AT&T). La que proliferó fue Sockets, en la que se basa Unix y Linux. Posteriormente, Windows lo tomó como base y lo extendió para crear sus WinSockets. En el caso de Linux, existen funciones para compatibilidad con los Sockets BSD.

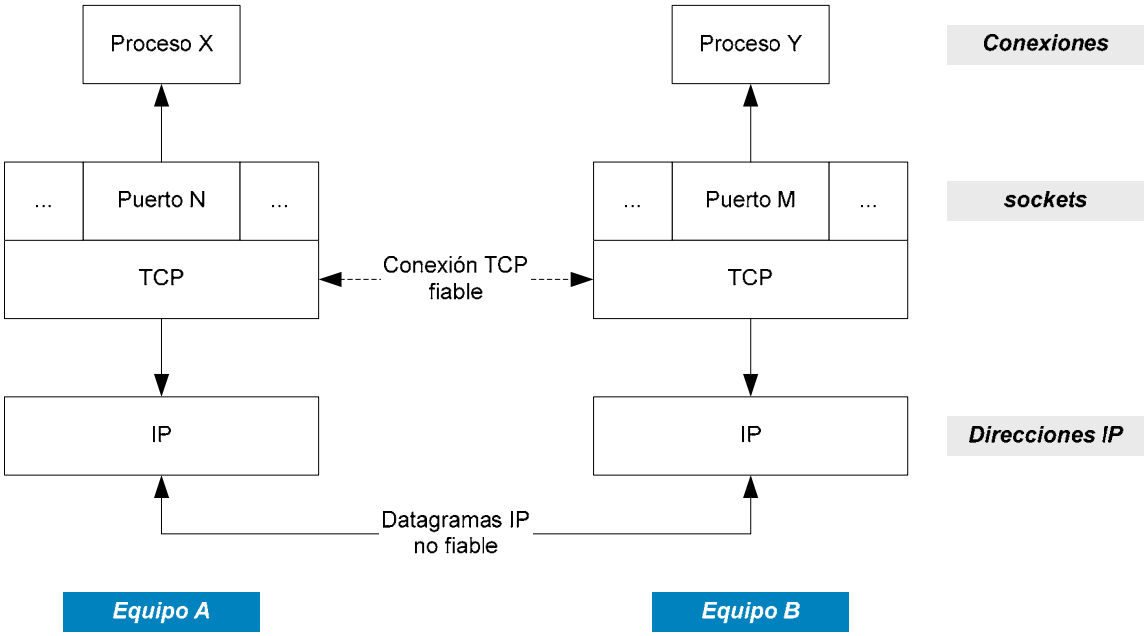


Definición

Socket designa un concepto abstracto por el cual dos programas (posiblemente situados en computadoras distintas) pueden intercambiarse cualquier flujo de datos, generalmente de manera fiable y ordenada.

Es un punto de comunicación, identificado por un descriptor. Dicho descriptor es igual al de entrada / salida estándar. Un *socket* queda definido por una dirección IP, un protocolo y un número de puerto.

En el caso concreto de TCP-IP, un socket se define por una dupla Origen – Destino. Tanto el origen como el destino vienen indicados por un par (ip, puerto).

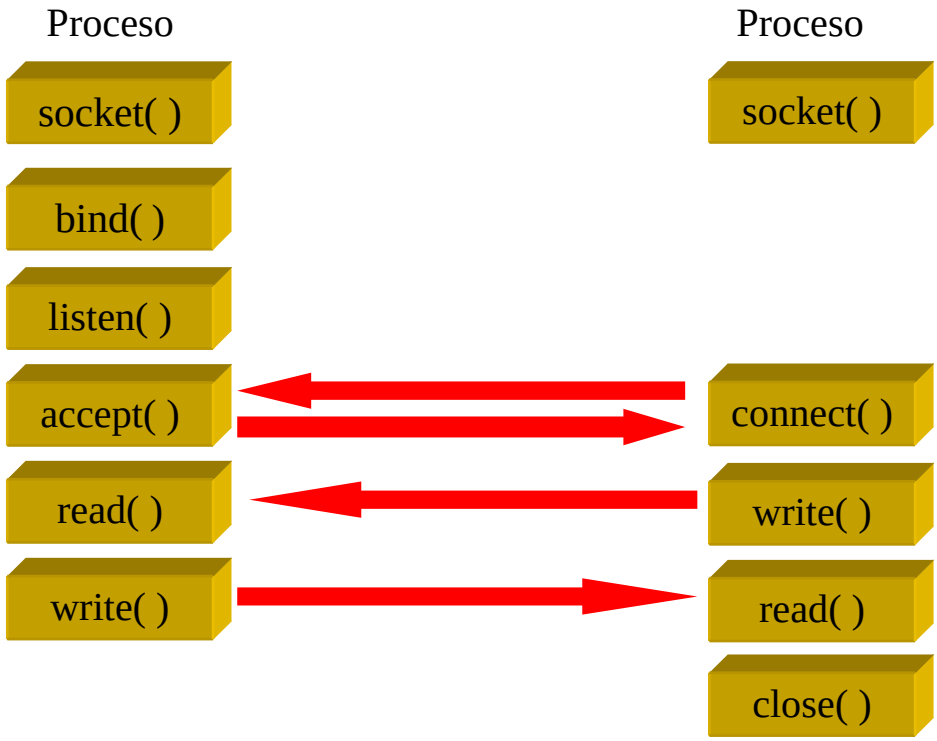


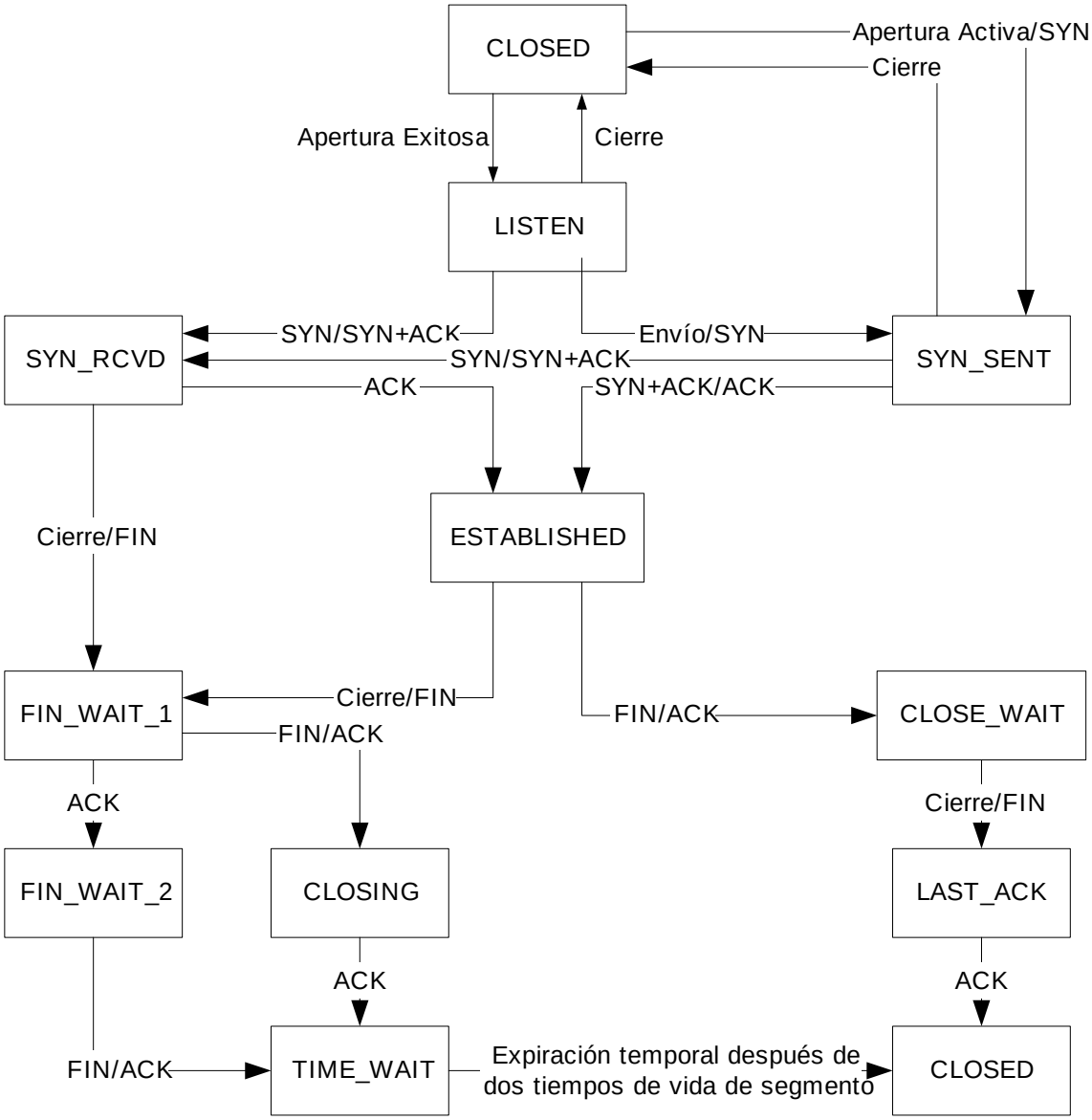
Existen dos tipos de sockets:

- Orientados a conexión (TCP)
  - Comunicaciones fiables
  - Circuito Virtual
- No orientados a conexión (UDP)
  - El programa de aplicación da la fiabilidad

Los sockets orientados a connexion (TCP)

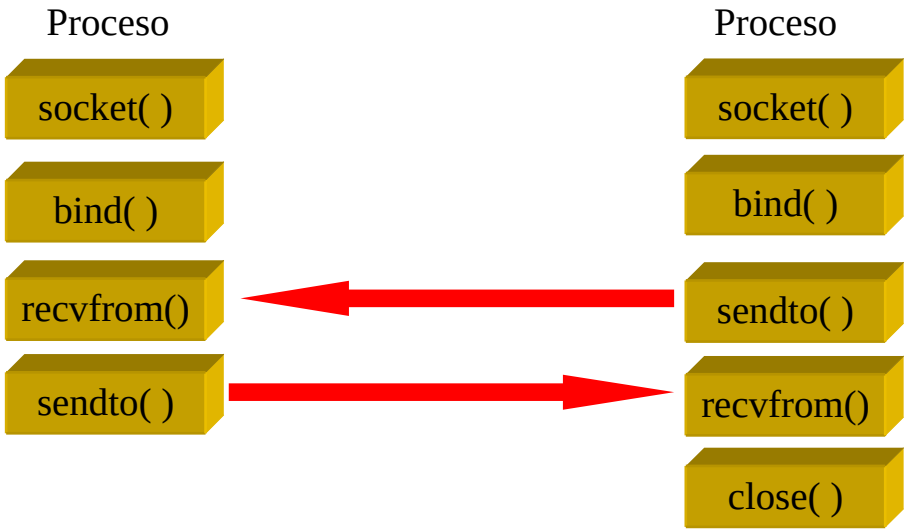
Este tipo de sockets están orientado a ristras (stream). Manejan un buffer para el envío y recepción de datos. Permite la conexión por Circuito Virtual (Se establece una conexión entre dos puntos antes del principio de la comunicación) y en modo Full-duplex (permite la comunicación bidireccional).




CLOSED
Apertura Activa/SYN
Cierre
Apertura Exitosa
Cierre
LISTEN
SYN_RCVD
SYN/SYN+ACK
Envío/SYN
SYN_SENT
SYN/SYN+ACK
ACK
SYN+ACK/ACK
Cierre/FIN
ESTABLISHED
FIN_WAIT_1
Cierre/FIN
FIN/ACK
CLOSE_WAIT
FIN/ACK
ACK
Cierre/FIN
FIN_WAIT_2
CLOSING
LAST_ACK
ACK
ACK
FIN/ACK
TIME_WAIT
Expiración temporal después de
dos tiempos de vida de segmento
CLOSED

Los sockets no orientados a connexion (UDP)

Las comunicaciones correspondientes tienen la propiedad de conservar los límites de los mensajes enviados. En el dominio Internet, el protocolo subyacente es el UDP. La transmisión es a nivel de paquetes, donde cada paquete puede seguir una ruta distinta, no garantizándose una recepción secuencial de la información.



API (Funciones)

Las funciones principales definidas en la API para la creación y manejo de sockets son:

1. SOCKET (Creación)
2. BIND (Unión Socket-Dirección)
3. LISTEN (Escucha)
4. CONNECT (Conexión cliente -> Servidor)
5. SENDTO (Enviar)
6. RECVFROM (Recibir)
7. CLOSE (Cerrar)

API (Estructuras)

Podemos identificar tres formatos de direcciones fundamentales:

1. Estructura Genérica
2. Dominio UNIX (AF_UNIX)
3. Dominio Internet (AF_INET)

Estos formatos condicionarán las estructuras que se usarán para representar el socket.

Estructura Genérica

Es la estructura empleada por defecto. Será reemplazada por la definida para el dominio concreto de comunicación, en el caso de que se defina una de éstas.

<incluye/linux/un.h>

```
 33typedef unsigned short  sa_family_t;
 39struct sockaddr {
 40  sa_family_t  sa_family;    /* familia de dirección, AF_xxx       */
 41 char          sa_data[14];  /* 14 bytes de dirección (máximo) */
 42};
```

Dominio UNIX (AF_UNIX)

Se trata de sockets locales al sistema en que se definen. Esta implementación es la que adopta UNIX, permitiendo la comunicación interna entre procesos.

<include/linux/un.h>

```
  4#define UNIX_PATH_MAX   108
  6 struct sockaddr_un {
  7       sa_family_t sun_family; /* dominio unix: AF_UNIX */
  8       char sun_path[UNIX_PATH_MAX];   /* pathname (referencia de dirección)*/
                                                                              9};
```

Dominio Internet (AF_INET)

Familia de sockets que se comunican mediante protocolos tales como TCP (Transmission Control Protocol) y UDP (User Datagram Protocol).

Hace uso de direcciones de sockets según el formato de la estructura sockaddr_in. Designa el servicio sobre una máquina particular.

<include/linux/in.h>

```
55/* Internet address. */
 56struct in_addr  {
 57      __be32  s_addr;
 58};
```

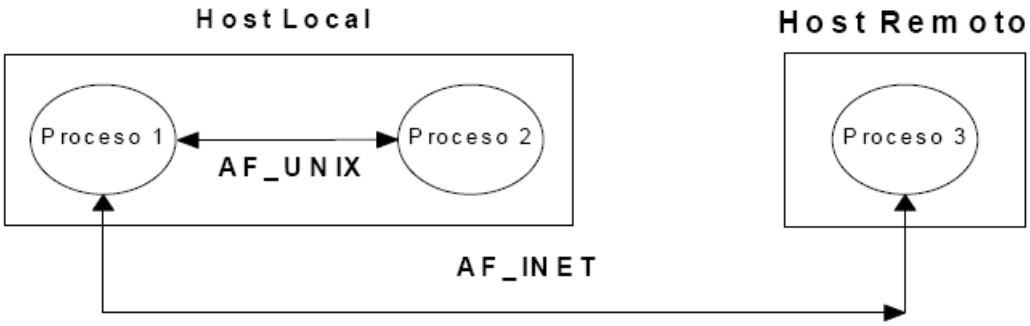
En la estructura del socket tiene los siguientes campos:

1. El primer campo de dicha estructura indica la familia del protocolo. AF_INET valdrá para protocolos de Internet.
2. El segundo campo (puerto) puede indicarse de dos formas:
    a. Número no nulo o superior a IPPORT_RESERVED
    b. Simbólica: para servicios estándar; por ejmplo: IPPORT_TCP o IPPORT_TELNET
3. El tercer campo indica la dirección de Internet. Puede tener dos formas:
    a. Dirección de Internet como IP o Nombre de Dominio
    b. El valor INADDR_ANY. Se usa en pasarelas, que disponen de varias direcciones diferentes.
4. El cuarto campo sirve para que coincidan en tamaño esta estructura con la estructura de dirección genérica sockaddr; el máximo es de 14 octetos de dirección, en principio.

<include/linux/in.h>

```
179/* Structure describing an Internet (IP) socket address. */
180#define __SOCK_SIZE__   16          /* sizeof(struct sockaddr)     */
181struct sockaddr_in {
```

```
182  sa_family_t         sin_family;    /* Address family          */
183  __be16              sin_port;      /* Port number             */
184  struct in_addr      sin_addr;      /* Internet address        */
185
186  /* Pad to size of `struct sockaddr'. Campo de ceros para alcanzar el tamaño de sockaddr */
187  unsigned char       __pad[__SOCK_SIZE__ - sizeof(short int) -
188                      sizeof(unsigned short int) - sizeof(struct in_addr)];
189};
```



Familias UNIX e INET

La diferencia entre los sockets de la familia UNIX y los de la familia Internet es que los primeros permiten la comunicación entre procesos que se están ejecutando en la misma máquina mientras que los segundos permiten la comunicación entre procesos que pueden estar ejecutándose en una misma máquina o en máquinas distintas.

Propiedades

Se trata de tener en cuenta las propiedades de la comunicación en las que se implican los sockets. Dichas propiedades son:

1. Fiabilidad de la Transmisión. No se pierden los datos transmitidos.
2. Conservación del Orden de los Datos. Los datos llegan en el orden en que se emitieron.

3. No Duplicación de los Datos. El Dato sólo llega una vez.
4. Comunicación en modo conectado. La conexión está establecida antes de iniciar la comunicación. De este modo, la emisión desde un extremo va destinada al otro (implícitamente).
5. Conservación de los límites de los mensajes. Los límites de mensajes emitidos pueden encontrarse o conocerse en el destino.
6. Envío de Mensajes “urgentes”. Permite el envío de datos fuera de flujo o fuera de banda. Al enviar datos fuera del flujo normal, son accesibles de inmediato.

Tipos

Existen tres tipos fundamentales:

1. SOCK_DGRAM
2. SOCK_STREAM
3. SOCK_RAW

SOCK_DGRAM

Se realiza la comunicación en modo no conectado, es decir, ya se ha establecido previamente la conexión. El envío de datagramas es de tamaño limitado. Se conservan los límites de los mensajes (Propiedad 5).

En Internet el protocolo subyacente es UDP.

SOCK_STREAM

Comunicaciones fiables en modo conectado (Propiedades 1 a la 4). Eventualmente autorizan mensajes fuera de banda (Propiedad 6).

En Internet, el protocolo subyacente es TCP.

SOCK_RAW

Permite el acceso a protocolos de bajo nivel, lo que permite definir el protocolo en sí, es decir, el proceso real de comunicación y sincronización.

En Internet, el protocolo subyacente es IP.

Su uso está reservado al Superusuario. Permite implantar o implementar nuevos protocolos, pues se tiene acceso a bajo nivel, de ahí el nombre de SOCK_RAW.

SOCK_SEQPACKET

Tiene las características del SOCK_STREAM pero además el tamaño de los mensajes es fijo.

2. Código

Estructura ***sock_common***

Esta es la capa de red mínima para la representación de sockets. Se incluirá en la cabecera de la estructura ***sock***. Sus campos son:

skc_family - familia de direcciones de red
skc_state - Estado de conexión
skc_bound_dev_if - número de dispositivos limitado si if != 0
skc_node - enlace main a hash para varias tablas de búsqueda de protocolos
skc_bind_node - enlace bind a hash para varias tablas de búsqueda de protocolos
skc_refcnt - contador de referencias

<include/net/sock.h>

```
 97/**
 98 *      struct sock_common - minimal network layer representation of sockets
 99 *      @skc_family: network address family
100 *      @skc_state: Connection state
101 *      @skc_reuse: %SO_REUSEADDR setting
102 *      @skc_bound_dev_if: bound device index if != 0
103 *      @skc_node: main hash linkage for various protocol lookup tables
104 *      @skc_bind_node: bind hash linkage for various protocol lookup tables
105 *      @skc_refcnt: reference count
106 *      @skc_hash: hash value used with various protocol lookup tables
107 *      @skc_prot: protocol handlers inside a network family
108 *
109 *      This is the minimal network layer representation of sockets, the header
110 *      for struct sock and struct inet_timewait_sock.
111 */
112struct sock_common {
```

```
113     unsigned short          skc_family;
114     volatile unsigned char  skc_state;
115     unsigned char           skc_reuse;
116     int                     skc_bound_dev_if;
117     struct hlist_node       skc_node;
118     struct hlist_node       skc_bind_node;
119     atomic_t                skc_refcnt;
120     unsigned int            skc_hash;
121     struct proto                    *skc_prot;
122};
```

Estructura ***sock***

Es la capa de red para la representación de sockets. Incluye la representación mínima, de ***sock_common***. Sus campos principales son:

sk_shutdown - máscara de %SEND_SHUTDOWN y/o %RCV_SHUTDOWN
sk_lock - sincronizador
sk_rcvbuf - tamaño de buffer de recepción en bytes
sk_sleep - cola de espera del socket
sk_dst_cache - cache de destino
sk_dst_lock - cache lock de destino
sk_policy - política de flujo
sk_rmem_alloc - bytes efectivos de cola de recepción
sk_receive_queue - paquetes entrantes
sk_wmem_alloc - bytes efectivos de cola de transmisión
sk_write_queue - cola de envío de paquetes
sk_wmem_queued - tamaño de cola persistente
sk_forward_alloc - espacio reservado hacia delante
sk_allocation - modo de reserva
sk_sndbuf - tamaño del buffer de envío en bytes

sk_no_largesend - enviar o no segmentos largos

sk_route_caps - prestaciones del enrutamiento (ej. %NETIF_F_TSO)

sk_backlog - siempre usado con el spinlock de cada socket tomado. La cola backlog es especial, pues siempre se usa con el spinlock de cada socket tomado y requiere un acceso de baja latencia. Por ello se implementa como un caso especial.

sk_prot - manejadores de protocolo dentro de una familia de red

sk_err - último error

sk_priority - configuración %SO_PRIORITY

sk_type - tipo de socket (%SOCK_STREAM, etc)

sk_localroute - sólo enrutamiento local, configuración %SO_DONTROUTE

sk_protocol - a que protocolo pertenece este socket en la familia de red

sk_filter - instrucciones de filtrado de socket

sk_timer - temporizador para limpiar el sock

sk_stamp - time stamp del último paquete recibido

sk_socket - Identidad y señales de e/s reportadas

sk_user_data - datos privados de la capa RPC

sk_owner - módulo propietario de este socket

sk_sndmsg_page - página cacheada para sendmsg

sk_sndmsg_off - desplazamiento cacheado para sendmsg

<include/net/sock.h>

```
183struct sock {
184      /*
185       * Now struct inet_timewait_sock also uses sock_common, so please just
186       * don't add nothing before this first member (__sk_common) --acme
187       */
188      struct sock_common      __sk_common;
189#define sk_family               __sk_common.skc_family
190#define sk_state                __sk_common.skc_state
191#define sk_reuse                __sk_common.skc_reuse
```

```
192#define sk_bound_dev_if        __sk_common.skc_bound_dev_if
193#define sk_node                __sk_common.skc_node
194#define sk_bind_node           __sk_common.skc_bind_node
195#define sk_refcnt                      __sk_common.skc_refcnt
196#define sk_hash                __sk_common.skc_hash
197#define sk_prot                __sk_common.skc_prot
198     unsigned char          sk_shutdown : 2,
199                            sk_no_check : 2,
200                            sk_userlocks : 4;
201     unsigned char          sk_protocol;
202     unsigned short         sk_type;
203     int                    sk_rcvbuf;
204     socket_lock_t          sk_lock;
205     /*
206      * The backlog queue is special, it is always used with
207      * the per-socket spinlock held and requires low latency
208      * access. Therefore we special case it's implementation.
209      */
210     struct {
211          struct sk_buff *head;
212          struct sk_buff *tail;
213     } sk_backlog;
214     wait_queue_head_t      *sk_sleep;
215     struct dst_entry       *sk_dst_cache;
216     struct xfrm_policy     *sk_policy[2];
217     rwlock_t               sk_dst_lock;
218     atomic_t               sk_rmem_alloc;
219     atomic_t               sk_wmem_alloc;
220     atomic_t               sk_omem_alloc;
221     int                    sk_sndbuf;
```

```
222     struct sk_buff_head     sk_receive_queue;
223     struct sk_buff_head     sk_write_queue;
224     struct sk_buff_head     sk_async_wait_queue;
225     int                     sk_wmem_queued;
226     int                     sk_forward_alloc;
227     gfp_t                   sk_allocation;
228     int                     sk_route_caps;
229     int                     sk_gso_type;
230     int                     sk_rcvlowat;
231     unsigned long           sk_flags;
232     unsigned long           sk_lingertime;
233     struct sk_buff_head     sk_error_queue;
234     struct proto            *sk_prot_creator;
235     rwlock_t                sk_callback_lock;
236     int                     sk_err,
237                             sk_err_soft;
238     unsigned short          sk_ack_backlog;
239     unsigned short          sk_max_ack_backlog;
240     __u32                   sk_priority;
241     struct ucred            sk_peercred;
242     long                    sk_rcvtimeo;
243     long                    sk_sndtimeo;
244     struct sk_filter        *sk_filter;
245     void                    *sk_protinfo;
246     struct timer_list       sk_timer;
247     ktime_t                 sk_stamp;
248     struct socket           *sk_socket;
249         void                                *sk_user_data;
250     struct page             *sk_sndmsg_page;
251     struct sk_buff          *sk_send_head;
```

```
252     __u32           sk_sndmsg_off;
253     int             sk_write_pending;
254         void                        *sk_security;
255     void            (*sk_state_change)(struct sock *sk);
256     void            (*sk_data_ready)(struct sock *sk, int bytes);
257     void            (*sk_write_space)(struct sock *sk);
258     void            (*sk_error_report)(struct sock *sk);
259     int             (*sk_backlog_rcv)(struct sock  *sk,
260                                       struct sk_buff *skb);
261     void            (*sk_destruct)(struct sock  *sk);
262};
```

Llamada del sistema ***sys_socketcall***

Cuando llamamos a una de las funciones de la API de sockets en Linux, lo que se realiza es una llamada al sistema tradicional, e internamente se llama a la función sys_socketcall, en cuyos parámetros se pasa el tipo de llamada que es (parámetro call) y sus argumentos (parámetro args). Es el punto de entrada al código de los sockets.

<net/socket.c>

```
1988/*
1989 *      System call vectors.
1990 *
1991 *      Argument checking cleaned up. Saved 20% in size.
1992 *  This function doesn't need to set the kernel lock because
1993 *  it is set by the callees.
1994 */
1995
1996asmlinkage long sys_socketcall(int call, unsigned long __user *args)
1997{
1998     unsigned long a[6];
```

```
1999    unsigned long a0, a1;
2000    int err;
2001
2002    if (call < 1 || call > SYS_RECVMSG)
2003            return -EINVAL;
2004
2005    /* copy_from_user should be SMP safe. */
2006    if (copy_from_user(a, args, nargs[call]))
2007            return -EFAULT;
2008
2009    err = audit_socketcall(nargs[call] / sizeof(unsigned long), a);
2010            if (err)
2011            return err;
2012
2013    a0 = a[0];
2014    a1 = a[1];
2015
2016    switch (call) {
2017    case SYS_SOCKET:
2018            err = sys_socket(a0, a1, a[2]);
2019            break;
2020    case SYS_BIND:
2021            err = sys_bind(a0, (struct sockaddr __user *)a1, a[2]);
2022            break;
2023    case SYS_CONNECT:
2024            err = sys_connect(a0, (struct sockaddr __user *)a1, a[2]);
2025            break;
2026    case SYS_LISTEN:
2027            err = sys_listen(a0, a1);
2028            break;
```

```
2029            case SYS_ACCEPT:
2030            err =
2031                sys_accept(a0, (struct sockaddr __user *)a1,
2032                        (int __user *)a[2]);
2033            break;
2034        case SYS_GETSOCKNAME:
2035            err =
2036                sys_getsockname(a0, (struct sockaddr __user *)a1,
2037                        (int __user *)a[2]);
2038                    break;
2039        case SYS_GETPEERNAME:
2040            err =
2041                sys_getpeername(a0, (struct sockaddr __user *)a1,
2042                        (int __user *)a[2]);
2043            break;
2044        case SYS_SOCKETPAIR:
2045            err = sys_socketpair(a0, a1, a[2], (int __user *)a[3]);
2046            break;
2047        case SYS_SEND:
2048            err = sys_send(a0, (void __user *)a1, a[2], a[3]);
2049            break;
2050        case SYS_SENDTO:
2051            err = sys_sendto(a0, (void __user *)a1, a[2], a[3],
2052                        (struct sockaddr __user *)a[4], a[5]);
2053            break;
2054        case SYS_RECV:
2055            err = sys_recv(a0, (void __user *)a1, a[2], a[3]);
2056            break;
2057        case SYS_RECVFROM:
2058            err = sys_recvfrom(a0, (void __user *)a1, a[2], a[3],
```

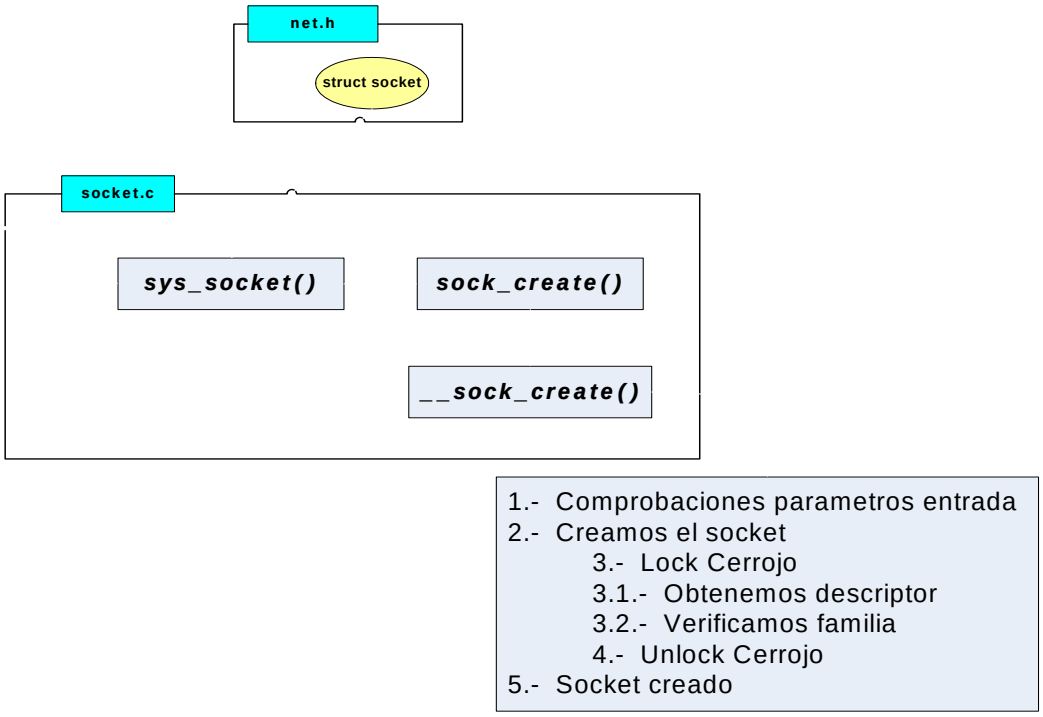
```
2059                    (struct sockaddr __user *)a[4],
2060                    (int __user *)a[5]);
2061            break;
2062          case SYS_SHUTDOWN:
2063            err = sys_shutdown(a0, a1);
2064                    break;
2065    case SYS_SETSOCKOPT:
2066           err = sys_setsockopt(a0, a1, a[2], (char __user *)a[3], a[4]);
2067            break;
2068    case SYS_GETSOCKOPT:
2069            err =
2070              sys_getsockopt(a0, a1, a[2], (char __user *)a[3],
2071                    (int __user *)a[4]);
2072                    break;
2073    case SYS_SENDMSG:
2074            err = sys_sendmsg(a0, (struct msghdr __user *)a1, a[2]);
2075            break;
2076    case SYS_RECVMSG:
2077            err = sys_recvmsg(a0, (struct msghdr __user *)a1, a[2]);
2078            break;
2079    default:
2080            err = -EINVAL;
2081                    break;
2082    }
2083    return err;
2084}
```

Creación (***sys_socket***)

Para crear un socket (sin nombre: no tiene asignada una dirección especifica) llamamos a la función sys_socket, a la cual le pasamos la familia, el tipo y el protocolo que deseamos usar.

Esta llamada, abre un canal bidireccional de comunicaciones. El efecto de socket es la creación de un punto terminal para conectarse a un canal y devuelve un descriptor, el cual, será empleado en llamadas posteriores cada vez que se quiera hacer uso del socket creado.

Esta función nos devuelve el descriptor del socket, que es el que usaremos con las otras funciones.



<include/linux/net.h>

```
/**
 * struct socket - general BSD socket
 * @state: socket state (%SS_CONNECTED, etc)
 * @flags: socket flags (%SOCK_ASYNC_NOSPACE, etc)
 * @ops: protocol specific socket operations
 * @fasync_list: Asynchronous wake up list
```

```
103 *  @file: File back pointer for gc
104 *  @sk: internal networking protocol agnostic socket representation
105 *  @wait: wait queue for several uses
106 *  @type: socket type (%SOCK_STREAM, etc)
107 */
108struct socket {
109     socket_state        state;
110     unsigned long       flags;
111     const struct proto_ops  *ops;
112     struct fasync_struct    *fasync_list;
113     struct file         *file;
114     struct sock         *sk;
115     wait_queue_head_t       wait;
116     short               type;
117};
```

<net/socket.c>

```
1197asmlinkage long sys_socket(int family, int type, int protocol)
1198{
1199     int retval;
1200     struct socket *sock;
1201
1202     retval = sock_create(family, type, protocol, &sock);
1203     if (retval < 0)
1204          goto out;
1205
1206     retval = sock_map_fd(sock);
1207     if (retval < 0)
1208          goto out_release;
1209
1210out:
```

```
1211        /* It may be already another descriptor 8) Not kernel problem. */
1212        return retval;
1213
1214out_release:
1215        sock_release(sock);
1216        return retval;
1217}


1187int sock_create(int family, int type, int protocol, struct socket **res)
1188{
1189        return __sock_create(family, type, protocol, res, 0);
1190}
1191


1074static int __sock_create(int family, int type, int protocol,
1075                         struct socket **res, int kern)
1076{
1077         int err;
1078        struct socket *sock;
1079        const struct net_proto_family *pf;
1080
1081        /*
1082         *      Check protocol is in range
1083         */
1084        if (family < 0 || family >= NPROTO)
1085                return -EAFNOSUPPORT;
1086        if (type < 0 || type >= SOCK_MAX)
1087                return -EINVAL;
1088
1089        /* Compatibility.
```

```
1090
1091        This uglymoron is moved from INET layer to here to avoid
1092        deadlock in module load.
1093       */
1094      if (family == PF_INET && type == SOCK_PACKET) {
1095            static int warned;
1096            if (!warned) {
1097                  warned = 1;
1098                  printk(KERN_INFO "%s uses obsolete (PF_INET,SOCK_PACKET)\n",
1099                        current->comm);
1100            }
1101            family = PF_PACKET;
1102      }
1103     /* Creamos el socket */
1104      err = security_socket_create(family, type, protocol, kern);
1105      if (err)
1106            return err;
1107
1108      /*
1109       *      Allocate the socket and allow the family to set things up. if
1110       *      the protocol is 0, the family is instructed to select an appropriate
1111       *      default.
1112       */
1113      sock = sock_alloc();
1114      if (!sock) {
1115            if (net_ratelimit())
1116                  printk(KERN_WARNING "socket: no more sockets\n");
1117            return -ENFILE; /* Not exactly a match, but its the
1118                        closest posix thing */
1119      }
```

```
1120
1121        sock->type = type;
1122
1123#if defined(CONFIG_KMOD)
1124        /* Attempt to load a protocol module if the find failed.
1125         *
1126         * 12/09/1996 Marcin: But! this makes REALLY only sense, if the user
1127         * requested real, full-featured networking support upon configuration.
1128         * Otherwise module support will break!
1129         */
1130        if (net_families[family] == NULL)
1131                request_module("net-pf-%d", family);
1132#endif
1133
1134        rcu_read_lock();
1135        pf = rcu_dereference(net_families[family]);
1136        err = -EAFNOSUPPORT;
1137        if (!pf)
1138                goto out_release;
1139
1140        /*
1141         * We will call the ->create function, that possibly is in a loadable
1142         * module, so we have to bump that loadable module refcnt first.
1143         */
1144        if (!try_module_get(pf->owner))
1145                goto out_release;
1146
1147        /* Now protected by module ref count */
1148        rcu_read_unlock();
1149
```

```
1150        err = pf->create(sock, protocol);
1151        if (err < 0)
1152                goto out_module_put;
1153
1154        /*
1155         * Now to bump the refcnt of the [loadable] module that owns this
1156         * socket at sock_release time we decrement its refcnt.
1157         */
1158        if (!try_module_get(sock->ops->owner))
1159                goto out_module_busy;
1160
1161        /*
1162         * Now that we're done with the ->create function, the [loadable]
1163         * module can have its refcnt decremented
1164         */
1165        module_put(pf->owner);
1166        err = security_socket_post_create(sock, family, type, protocol, kern);
1167        if (err)
1168                goto out_sock_release;
1169        *res = sock;
1170
1171        return 0;
1172
1173out_module_busy:
1174        err = -EAFNOSUPPORT;
1175out_module_put:
1176        sock->ops = NULL;
1177        module_put(pf->owner);
```

```
1178out_sock_release:
1179        sock_release(sock);
1180        return err;
1181
1182out_release:
1183        rcu_read_unlock();
1184        goto out_sock_release;
1185}
```

Unión socket-dirección (***sys_bind***)

Una vez creado, deberemos especificar qué dirección usaremos con el socket (lo que se conoce como dar nombre al socket). En nuestro ordenador pueden existir muchas direcciones de red: la local, la de las interfaces de red, … En esta función enlazaremos el socket con las direcciones de red que queramos. La mayor parte del enlace es responsabilidad del protocolo. Lo primero que haremos será mover la dirección del socket a la memoria del núcleo, ya que el enlace se realiza allí.

Bind hace que el socket cuyo descriptor es ***fd*** se una a la dirección de socket específica apuntada por ***umyaddr***, ***addrlen*** indica el tamaño de la dirección.

Hay tres usos posibles de bind:

1. *Un servidor registra su dirección bien conocida en el sistema. Tanto los servidores orientados a conexión como los no orientados a conexión han de llevar a cabo este paso antes de aceptar peticiones de clientes.*
2. *Un cliente puede registrar una dirección específica para sí mismo si lo desea.*
3. *Un cliente no orientado a conexión necesita asegurarse de que el sistema le asigna una dirección única en el sistema, de forma que el otro extremo tiene una dirección de retorno válida donde mandar las respuestas. Esto es equivalente a asegurarnos de que un sobre tiene una dirección válida del remitente si esperamos obtener respuesta de la persona a la que le enviamos la carta.*

<net/socket.c>

```
1319asmlinkage long sys_bind(int fd, struct sockaddr __user *umyaddr, int addrlen)
1320{
1321        struct socket *sock;
1322        char address[MAX_SOCK_ADDR];
1323        int err, fput_needed;
1324
1325        sock = sockfd_lookup_light(fd, &err, &fput_needed);
1326        if (sock) {
1327                err = move_addr_to_kernel(umyaddr, addrlen,  address);
1328                if (err >= 0) {
1329                        err = security_socket_bind(sock,
1330                                                   (struct sockaddr *)address,
1331                                                    addrlen);
1332                                                       if (!err)
1333                                err = sock->ops->bind(sock,
1334                                                      (struct sockaddr *)
1335                                                      address, addrlen);
1336                }
1337                fput_light(sock->file, fput_needed);
1338        }
1339                return err;
1340}
```

Obtener la dirección local (***sys_getsockname***)

Sirve para obtener la dirección local de un socket. Para ello movemos la dirección obtenida al espacio de usuario.

<net/socket.c>

```
 /*
1499 *      Get the local address ('name') of a socket object. Move the obtained
```

```
1500 *      name to user space.
1501 */
1502
1503asmlinkage long sys_getsockname(int fd, struct sockaddr __user *usockaddr,
1504                         int __user *usockaddr_len)
1505{
1506         struct socket *sock;
1507         char address[MAX_SOCK_ADDR];
1508         int len, err, fput_needed;
1509
1510         sock = sockfd_lookup_light(fd, &err, &fput_needed);
1511         if (!sock)
1512                 goto out;
1513
1514         err = security_socket_getsockname(sock);
1515                 if (err)
1516                 goto out_put;
1517
1518         err = sock->ops->getname(sock, (struct sockaddr *)address, &len, 0);
1519                 if (err)
1520                 goto out_put;
1521         err = move_addr_to_user(address, len, usockaddr, usockaddr_len);
1522
1523out_put:
1524         fput_light(sock->file, fput_needed);
1525out:
1526         return err;
1527}
```

Obtener la dirección remota (***sys_getpeername***)

Sirve para obtener la dirección remota de un socket. Para ello movemos la dirección obtenida al espacio de usuario.

<net/socket.c>

```
 /*
1530 *      Get the remote address ('name') of a socket object. Move the obtained
1531 *     name to user space.
1532 */
1533
1534asmlinkage long sys_getpeername(int fd, struct sockaddr __user *usockaddr,
1535                         int __user *usockaddr_len)
1536{
1537      struct socket *sock;
1538      char address[MAX_SOCK_ADDR];
1539      int len, err, fput_needed;
1540
1541      sock = sockfd_lookup_light(fd, &err, &fput_needed);
1542      if (sock != NULL) {
1543            err = security_socket_getpeername(sock);
1544            if (err) {
1545                  fput_light(sock->file, fput_needed);
1546                  return err;
1547            }
1548
1549            err =
1550               sock->ops->getname(sock, (struct sockaddr *)address, &len,
1551                                  1);
1552            if (!err)
```

```
1553                    err = move_addr_to_user(address, len, usockaddr,
1554                                            usockaddr_len);
1555            fput_light(sock->file, fput_needed);
1556    }
1557    return err;
1558}
```

Escucha (***sys_listen***)

Esta llamada sólo es aplicable en sockets orientados a conexión. Es invocada por el programa servidor e indica que está disponible para recibir peticiones de conexión.

Listen habilita una cola asociada al socket cuyo descriptor es ***fd***, dicha cola se va a encargar de alojar peticiones de conexión procedentes de los procesos cliente y que están pendientes de ser atendidas. El entero ***backlog*** especifica la longitud de dicha cola, este límite es necesario para proteger el sistema de usar todos sus recursos para mantener pendientes peticiones de servicio para un servidor con problemas que nunca termina de atender a la actual petición. La cola es sobre todo importante en servidores de tipo interactivo. El valor que se suele usar para ***backlog*** es 5, que es el máximo permitido.

<net/socket.c>

```
 /*
1343 *      Perform a listen. Basically, we allow the protocol to do anything
1344 *      necessary for a listen, and if that works, we mark the socket as
1345 *      ready for listening.
1346 */
1347
1348int sysctl_somaxconn __read_mostly = SOMAXCONN;
1349
1350asmlinkage  long  sys_listen(int fd, int backlog)
1351{
1352     struct socket *sock;
```

```
1353    int err, fput_needed;
1354
1355    sock = sockfd_lookup_light(fd, &err, &fput_needed);
1356    if (sock) {
1357        if ((unsigned)backlog > sysctl_somaxconn)
1358            backlog = sysctl_somaxconn;
1359
1360                err = security_socket_listen(sock, backlog);
1361        if (!err)
1362                        err = sock->ops->listen(sock, backlog);
1363
1364        fput_light(sock->file, fput_needed);
1365    }
1366    return err;
1367}
```

Acepta una conexión (***sys_accept***)

Esta llamada es ejecutada por los procesos servidores y como su propio nombre sugiere, sirve para aceptar peticiones de conexión

Esta llamada se usa con sockets orientados a conexión (SOCK_STREAM), el argumento ***fd*** es el descriptor del socket que está a la escucha de peticiones de servicio normalmente en un puerto bien conocido. La llamada accept extrae la primera petición de conexión de la cola de peticiones pendientes que se crea con la llamada a listen, y crea para dicho servicio un nuevo socket. No obstante, si no hay peticiones de servicio pendientes de atención accept permanece bloqueada hasta que se produce alguna, (a menos que el socket tenga activo el modo de acceso no bloqueante).

El socket original (***fd***) permanece abierto y puede aceptar nuevas conexiones; sin embargo, el socket recién creado no puede usarse para aceptar nuevas conexiones. La

estructura addr, contendrá la dirección del socket remoto una vez que la llamada a accept se haya ejecutado con éxito

<net/socket.c>

```
/*
1370 *     For accept, we attempt to create a new socket, set up the link
1371 *     with the client, wake up the client, then return the new
1372 *     connected fd. We collect the address of the connector in kernel
1373 *     space and move it to user at the very end. This is unclean because
1374 *     we open the socket then return an error.
1375 *
1376 *     1003.1g adds the ability to recvmsg() to query connection pending
1377 *     status to recvmsg. We need to add that support in a way thats
1378 *     clean when we restucture accept also.
1379 */
1380
1381asmlinkage long sys_accept(int fd, struct sockaddr __user *upeer_sockaddr,
1382                 int __user *upeer_addrlen)
1383{
1384     struct socket *sock, *newsock;
1385     struct file *newfile;
1386     int err, len, newfd, fput_needed;
1387     char address[MAX_SOCK_ADDR];
1388
1389     sock = sockfd_lookup_light(fd, &err, &fput_needed);
1390     if (!sock)
1391          goto out;
1392
1393     err = -ENFILE;
1394     if (!(newsock = sock_alloc()))
```

```
1395            goto out_put;
1396
1397    newsock->type = sock->type;
1398    newsock->ops = sock->ops;
1399
1400    /*
1401     * We don't need try_module_get here, as the listening socket (sock)
1402     * has the protocol module (sock->ops->owner) held.
1403                */
1404    __module_get(newsock->ops->owner);
1405
1406              newfd = sock_alloc_fd(&newfile);
1407    if (unlikely(newfd < 0)) {
1408            err = newfd;
1409            sock_release(newsock);
1410            goto out_put;
1411    }
1412
1413    err = sock_attach_fd(newsock,  newfile);
1414    if (err < 0)
1415            goto out_fd_simple;
1416
1417    err = security_socket_accept(sock, newsock);
1418    if (err)
1419            goto out_fd;
1420
1421    err = sock->ops->accept(sock, newsock, sock->file->f_flags);
1422              if (err < 0)
1423            goto out_fd;
1424
```

```
1425        if (upeer_sockaddr) {
1426            if (newsock->ops->getname(newsock, (struct sockaddr *)address,
1427                                      &len, 2) < 0) {
1428                err = -ECONNABORTED;
1429                goto out_fd;
1430            }
1431            err = move_addr_to_user(address, len, upeer_sockaddr,
1432                                    upeer_addrlen);
1433            if (err < 0)
1434                goto out_fd;
1435        }
1436
1437        /* File flags are not inherited via accept() unlike another OSes. */
1438
1439        fd_install(newfd, newfile);
1440        err = newfd;
1441
1442        security_socket_post_accept(sock, newsock);
1443
1444out_put:
1445        fput_light(sock->file, fput_needed);
1446out:
1447        return err;
1448out_fd_simple:
1449        sock_release(newsock);
1450        put_filp(newfile);
1451        put_unused_fd(newfd);
1452        goto out_put;
1453out_fd:
1454        fput(newfile);
```

```
1455        put_unused_fd(newfd);
1456        goto out_put;
1457}
```

Conecta a un servidor (***sys_connect***)

Esta llamada es invocada por un proceso cliente que desea establecer una conexión con un proceso servidor a través de un socket. Básicamente movemos la dirección del espacio de usuario al espacio del núcleo una vez comprobada ya que es allí donde se realiza la conexión.

fd es el descriptor del socket en nuestra máquina local que nos va a dar acceso al canal mientras que ***uservaddr*** es un puntero a una estructura que contiene la dirección del socket remoto al que deseamos conectarnos. ***addrlen*** es el tamaño en bytes de la dirección.

Para los protocolos orientados a conexión, como es el caso de TCP, la llamada al sistema connect resulta en el establecimiento de una conexión entre el sistema local y el remoto. En este caso la llamada connect bloquea el proceso hasta que se efectúa la conexión o un código de error es devuelto en caso de que no sea posible conectar ambos procesos. *El cliente no tiene que llamar a bind antes de ejecutar connect.*

Un cliente no orientado a conexión puede usar también la llamada a connect, pero el resultado será diferente al que hemos descrito hasta ahora. Para un protocolo no orientado a conexión, lo que hace connect es almacenar la dirección del servidor, especificada por el proceso, en addr, de forma que el sistema sabe donde enviar cualquier dato que en el futuro el proceso desee escribir en el socket. Además sólo datagramas, podrán ser enviados por el socket. Una ventaja de usar connect en protocolos no orientados a conexión es que no necesitamos especificar la dirección destino en cada envío de datos.

<net/socket.c>

```
/*
1460 *      Attempt to connect to a socket with the server address.  The address
1461 *      is in user space so we verify it is OK and move it to kernel space.
```

```
1462 *
1463 *      For 1003.1g we need to add clean support for a bind to AF_UNSPEC to
1464 *      break bindings
1465 *
1466 *      NOTE: 1003.1g draft 6.3 is broken with respect to AX.25/NetROM and
1467 *      other SEQPACKET protocols that take time to connect() as it doesn't
1468 *      include the -EINPROGRESS status for such sockets.
1469 */
1470
1471asmlinkage long sys_connect(int fd, struct sockaddr __user *uservaddr,
1472                            int addrlen)
1473{
1474     struct socket *sock;
1475     char address[MAX_SOCK_ADDR];
1476     int err, fput_needed;
1477
1478     sock = sockfd_lookup_light(fd, &err, &fput_needed);
1479     if (!sock)
1480          goto out;
1481     err = move_addr_to_kernel(uservaddr, addrlen, address);
1482     if (err < 0)
1483          goto out_put;
1484
1485     err =
1486        security_socket_connect(sock, (struct sockaddr *)address, addrlen);
1487     if (err)
1488                    goto out_put;
1489
1490     err = sock->ops->connect(sock, (struct sockaddr *)address, addrlen,
1491                              sock->file->f_flags);
```

```
1492out_put:
1493        fput_light(sock->file,  fput_needed);
1494out:
1495        return err;
1496}
```

Crea un par de sockets conectados (***sys_socketpair***)

Creamos dos sockets y los conectamos entre ellos. Asignamos los descriptores creadors a los espacios de usuario especificados.

family, type y ***protocol*** tiene el mismo significado que en la definición de ***sys_socket()***

usockvec es un vector que contendrá los 2 descriptores de los sockets generados.

<net/socket.c>

```
/*
1220 *     Create a pair of connected sockets.
1221 */
1222
1223asmlinkage  long  sys_socketpair(int family, int type, int protocol,
1224                        int __user *usockvec)
1225{
1226        struct socket *sock1, *sock2;
1227        int fd1, fd2, err;
1228        struct file *newfile1, *newfile2;
1229
1230        /*
1231         * Obtain the first socket and check if the underlying protocol
1232         * supports the socketpair call.
1233         */
```

```
1234
1235    err = sock_create(family, type, protocol, &sock1);
1236    if (err < 0)
1237            goto out;
1238
1239              err = sock_create(family, type, protocol, &sock2);
1240    if (err < 0)
1241            goto out_release_1;
1242
1243    err = sock1->ops->socketpair(sock1, sock2);
1244    if (err < 0)
1245                            goto out_release_both;
1246
1247    fd1 = sock_alloc_fd(&newfile1);
1248    if (unlikely(fd1 < 0))
1249            goto out_release_both;
1250
1251              fd2 = sock_alloc_fd(&newfile2);
1252    if (unlikely(fd2 < 0)) {
1253                            put_filp(newfile1);
1254            put_unused_fd(fd1);
1255            goto out_release_both;
1256    }
1257
1258    err = sock_attach_fd(sock1,  newfile1);
1259    if (unlikely(err < 0)) {
1260            goto out_fd2;
1261    }
1262
1263    err = sock_attach_fd(sock2, newfile2);
```

```
1264        if (unlikely(err < 0)) {
1265                fput(newfile1);
1266                goto out_fd1;
1267        }
1268
1269        err = audit_fd_pair(fd1, fd2);
1270        if (err < 0) {
1271                fput(newfile1);
1272                fput(newfile2);
1273                goto out_fd;
1274                }
1275
1276        fd_install(fd1, newfile1);
1277        fd_install(fd2, newfile2);
1278        /* fd1 and fd2 may be already another descriptors.
1279         * Not kernel problem.
1280         */
1281
1282        err = put_user(fd1, &usockvec[0]);
1283        if (!err)
1284                err = put_user(fd2, &usockvec[1]);
1285                if (!err)
1286                return 0;
1287
1288        sys_close(fd2);
1289        sys_close(fd1);
1290        return err;
1291
1292out_release_both:
1293        sock_release(sock2);
```

```
1294out_release_1:
1295        sock_release(sock1);
1296out:
1297        return err;
1298
1299out_fd2:
1300        put_filp(newfile1);
1301                sock_release(sock1);
1302out_fd1:
1303        put_filp(newfile2);
1304        sock_release(sock2);
1305out_fd:
1306                put_unused_fd(fd1);
1307        put_unused_fd(fd2);
1308        goto out;
1309}
```

Envío de Mensajes

En todas las llamadas ***fd*** es un descriptor de socket en el que se escriben los datos, ***buff*** es un puntero al buffer donde están los datos que se desean enviar y ***len*** el número de bytes que hay en el buffer.

En **sockets orientados a conexión** sólo pueden usarse después de haber establecido conexión con una llamada a ***connect***. En **sockets no orientados a conexión**, SOCK_DGRAM, debe usarse la función ***sendto***, a menos que previamente se indique la dirección destino con una llamada a ***connect***. La dirección del socket destino está en el buffer apuntado por ***adra*** y su tamaño es ***addr_len***.

En sockets datagrama para los que no se ha definido una dirección mediante ***bind***, si se envía un mensaje con ***sendto***, el sistema va a elegir de forma automática una dirección

local, pero no hay garantías de que esa dirección siga siendo la misma en sucesivas llamadas a ***sendto***.

Flags tiene como valores posibles 0 y MSG_OOB (ver *"Recepcion de mensajes"*) para enviar mensajes fuera de banda. No obstante para la familia AF_UNIX no se pueden enviar mensajes fuera de banda.

Envío de Mensajes (***sys_sendto***)

Envía un datagrama (en realidad un mensaje) a una dirección dada. Dicha dirección se pasa al espacio del kernel. Puede ser que dicha dirección sea nula, en cuyo caso se ha hecho realmente un un ***sys_send***. Además, se chequea que el área de datos del espacio de usuario se puede leer antes de invocar el protocolo.

<net/socket.c>

```
1560/*
1561 *      Send a datagram to a given address. We move the address into kernel
1562 *      space and check the user space data area is readable before invoking
1563 *      the protocol.
1564 */
1565
1566asmlinkage long sys_sendto(int fd, void __user *buff, size_t len,
1567                                        unsigned flags, struct sockaddr __user *addr,
1568                int addr_len)
1569{
1570     struct socket *sock;
1571     char address[MAX_SOCK_ADDR];
1572     int err;
1573     struct msghdr msg;
1574     struct iovec  iov;
1575     int fput_needed;
1576     struct file *sock_file;
```

```
1577
1578        sock_file = fget_light(fd, &fput_needed);
1579                err = -EBADF;
1580        if (!sock_file)
1581                goto out;
1582
1583        sock = sock_from_file(sock_file, &err);
1584        if (!sock)
1585                goto out_put;
1586        iov.iov_base = buff;
1587        iov.iov_len = len;
1588        msg.msg_name  =  NULL;
1589        msg.msg_iov = &iov;
1590        msg . msg_iovlen = 1;
1591        msg.msg_control = NULL;
1592        msg.msg_controllen = 0;
1593        msg.msg_namelen = 0;
1594        if (addr) {
1595                err = move_addr_to_kernel(addr, addr_len, address);
1596                if (err < 0)
1597                        goto out_put;
1598                msg.msg_name = address;
1599                msg.msg_namelen = addr_len;
1600        }
1601        if (sock->file->f_flags & O_NONBLOCK)
1602                flags |= MSG_DONTWAIT;
1603        msg.msg_flags = flags;
1604        err = sock_sendmsg(sock, &msg, len);
1605
1606out_put:
```

```
1607        fput_light(sock_file, fput_needed);
1608out:
1609        return err;
1610}
```

Envío de Mensajes (*sys_send*)

Envía un datagrama (en realidad un mensaje) por un socket.

<net/socket.c>

```
   /*
1613 *      Send a datagram down a socket.
1614 */
1615
1616asmlinkage long sys_send(int fd, void __user *buff, size_t len, unsigned flags)
1617{
1618        return sys_sendto(fd, buff, len, flags, NULL, 0);
1619}
```

Interfaz BSD para *sendmsg*

La interfaz BSD es la implementación al estilo BSD para garantizar la compatibilidad con la misma. Como se trata de una interfaz, la implementación final será la misma que para el envío de mensajes al estilo UNIX con ***sys_sendto***; dicha implementación será común desde ***sock_sendmsg***.

<net/socket.c>

```
1776asmlinkage long sys_sendmsg(int fd, struct msghdr __user *msg, unsigned flags)
1777{
1778        struct compat_msghdr __user *msg_compat =
1779            (struct compat_msghdr __user *)msg;
1780        struct socket *sock;
1781        char address[MAX_SOCK_ADDR];
```

```
1782    struct iovec iovstack[UIO_FASTIOV], *iov = iovstack;
1783    unsigned char ctl[sizeof(struct cmsghdr) + 20]
1784        __attribute__ ((aligned(sizeof(__kernel_size_t))));
1785    /* 20 is size of ipv6_pktinfo */
1786    unsigned char *ctl_buf = ctl;
1787    struct msghdr msg_sys;
1788    int err, ctl_len, iov_size, total_len;
1789    int fput_needed;
1790
1791    err = -EFAULT;
1792    if (MSG_CMSG_COMPAT & flags) {
1793        if (get_compat_msghdr(&msg_sys, msg_compat))
1794            return -EFAULT;
1795    }
1796    else if (copy_from_user(&msg_sys, msg, sizeof(struct msghdr)))
1797        return -EFAULT;
1798   /* Obtener el socket*/
1799    sock = sockfd_lookup_light(fd, &err, &fput_needed);
1800    if (!sock)
1801        goto out;
1802
1803    /* do not move before msg_sys is valid */
1804    err = -EMSGSIZE;
1805    if (msg_sys.msg_iovlen > UIO_MAXIOV) /* Comprobar que msg_sys es válido*/
1806        goto out_put;
1807
1808    /* Check whether to allocate the iovec area */
1809    err = -ENOMEM;
1810    iov_size = msg_sys.msg_iovlen * sizeof(struct iovec);
1811    if (msg_sys.msg_iovlen > UIO_FASTIOV) {
```

```
1812            iov = sock_kmalloc(sock->sk, iov_size, GFP_KERNEL);
1813            if (!iov)
1814                    goto out_put;
1815    }
1816
1817    /* This will also move the address data into kernel space */
1818    if (MSG_CMSG_COMPAT & flags) {
1819        err = verify_compat_iovec(&msg_sys, iov,  address, VERIFY_READ);
1820    } else
1821            err = verify_iovec(&msg_sys, iov, address, VERIFY_READ);
1822    if (err < 0)
1823            goto out_freeiov;
1824    total_len = err;
1825
1826    err = -ENOBUFS;
1827
1828    if (msg_sys.msg_controllen > INT_MAX)
1829            goto out_freeiov;
1830    ctl_len = msg_sys.msg_controllen;
1831    if ((MSG_CMSG_COMPAT & flags) && ctl_len) {
1832            err =
1833              cmsghdr_from_user_compat_to_kern(&msg_sys, sock->sk, ctl,
1834                                               sizeof(ctl));
1835                      if (err)
1836                    goto out_freeiov;
1837            ctl_buf = msg_sys.msg_control;
1838            ctl_len = msg_sys.msg_controllen;
1839    } else if (ctl_len) {
1840            if (ctl_len > sizeof(ctl)) {
1841                ctl_buf = sock_kmalloc(sock->sk, ctl_len, GFP_KERNEL);
```

```
1842                    if (ctl_buf == NULL)
1843                                                    goto out_freeiov;
1844            }
1845            err = -EFAULT;
1846    /*
1847     * Careful! Before this, msg_sys.msg_control contains a user pointer.
1848     * Afterwards, it will be a kernel pointer. Thus the compiler-assisted
1849     * checking falls down on this.
1850     */
1851          if (copy_from_user(ctl_buf, (void __user *)msg_sys.msg_control,
1852                             ctl_len))
1853                  goto out_freectl;
1854            msg_sys.msg_control = ctl_buf;
1855      }
1856      msg_sys.msg_flags = flags;
1857
1858      if (sock->file->f_flags & O_NONBLOCK)
1859            msg_sys.msg_flags |= MSG_DONTWAIT;
1860      err = sock_sendmsg(sock, &msg_sys, total_len);
1861
1862out_freectl:
1863      if (ctl_buf != ctl)
1864            sock_kfree_s(sock->sk, ctl_buf, ctl_len);
1865out_freeiov:
1866      if (iov != iovstack)
1867            sock_kfree_s(sock->sk, iov, iov_size);
1868out_put:
1869      fput_light(sock->file, fput_needed);
1870out:
1871      return err;
```

```
1872}
```

Enviar Mensaje (***sock_sendmsg***)

Primera función en las llamadas para envío de mensajes.

<net/socket.c>

```
559int sock_sendmsg(struct socket *sock, struct msghdr *msg, size_t size)
560{
561     struct kiocb  iocb;
562     struct sock_iocb siocb;
563     int ret;
564
565     init_sync_kiocb(&iocb, NULL);
566     iocb.private = &siocb;
567     ret = __sock_sendmsg(&iocb, sock, msg, size); /* Enviar mensaje*/
568     if (-EIOCBQUEUED == ret)
569          ret = wait_on_sync_kiocb(&iocb);
570     return ret; /* Controlar el error*/
571}
```

Enviar Mensaje (***__sock_sendmsg***)

Segunda función en las llamadas para envío de mensajes. Desde ella se llama a la función concreta del protocolo usado para la comunicación.

<net/socket.c>

```
541static inline int __sock_sendmsg(struct kiocb *iocb, struct socket *sock,
542                    struct msghdr *msg, size_t size)
543{
544     struct sock_iocb *si = kiocb_to_siocb(iocb);
```

```
545     int err;
546
547     si->sock = sock;
548     si->scm = NULL;
549     si->msg = msg;
550             si->size = size;
551     /* Envio de operación segura*/
552     err = security_socket_sendmsg(sock, msg, size);
553     if (err)
554             return err;
555     /* Envio del mensaje dependiente del protocolo*/
556     return sock->ops->sendmsg(iocb, sock,  msg, size);
557}
```

Recepción de Mensajes

En todas las llamadas ***fd*** es el descriptor del socket del cual se leen los datos, ***ubuf/msg*** es un puntero al buffer donde se van a escribir los datos leídos y ***len*** el número máximo de bytes que se pueden escribir en el buffer (espacio reservado para memorizar el mensaje).

En **sockets orientados a conexión**, SOCK_STREAM, las llamadas pueden usarse sólo después de haber establecido una conexión previa llamada a ***connect***, mientras que en **sockets no orientados a conexión**, SOCK_DGRAM, las llamadas pueden usarse tanto si se ha establecido una conexión como sí no.

La llamada a ***recvfrom***, tiene como parámetro ***addr*** que, como se observa, es un puntero a una estructura de dirección de socket remoto. En sockets datagrama, en este parámetro se almacena la dirección desde la que se enviaron los datos mientras que para sockets stream, ***recvfrom*** actúa igual que ***recv*** y por tanto en ***addr*** no se devuelve la dirección origen de la información.

El argumento ***flags*** puede valer 0 o bien cualquiera de los siguientes bits:

- MSG_PEEK: Cualquier dato leído del socket es tratado como si la lectura no se hubiese llevado a cabo, por lo que la siguiente operación de lectura va a leer los mismos datos. Esta opción se puede usar para ojear si hay datos disponibles en el socket.
- MSG_OOB: Se utiliza para el envío de datos fuera de banda (Out Of Band). Son datos a los que se les da un carácter de urgencia por lo que tienen preferencia en el envío y en la recepción.

Recepción de Mensajes (*sys_recvfrom*)

Recibe un datagrama (en realidad un mensaje) de un socket y opcionalmente almacena la dirección del remitente. Previamente se comprueba que los buffers pueden escribirse y si es necesario se mueve la dirección del remitente del espacio del kernel al del usuario.

<net/socket.c>

```
/*
1622 *      Receive a frame from the socket and optionally record the address of the
1623 *      sender. We verify the buffers are writable and if needed move the
1624 *      sender address from kernel to user space.
1625 */
1626
1627asmlinkage long sys_recvfrom(int fd, void __user *ubuf, size_t size,
1628                             unsigned flags, struct sockaddr __user *addr,
1629                             int __user *addr_len)
1630{
1631     struct socket *sock;
1632     struct iovec iov;
1633     struct msghdr msg;
1634     char address[MAX_SOCK_ADDR];
1635     int err, err2;
```

```
1636    struct file *sock_file;
1637    int fput_needed;
1638
1639    sock_file = fget_light(fd, &fput_needed);
1640    err = -EBADF;
1641    if (!sock_file)
1642        goto out;
1643  /* Obtener el socket*/
1644    sock = sock_from_file(sock_file, &err);
1645    if (!sock)
1646        goto out_put;
1647   /* Preparar el mensaje para su recepción*/
1648    msg.msg_control = NULL;
1649    msg.msg_controllen = 0;
1650    msg.msg_iovlen = 1;
1651    msg.msg_iov = &iov;
1652    iov.iov_len = size;
1653    iov.iov_base = ubuf;
1654    msg.msg_name = address;
1655    msg.msg_namelen = MAX_SOCK_ADDR;
1656    if (sock->file->f_flags & O_NONBLOCK)
1657        flags |= MSG_DONTWAIT;
        /* Recibir el mensaje*/
1658    err = sock_recvmsg(sock, &msg, size, flags);
1659
1660    if (err >= 0 && addr != NULL) {
1661      err2 = move_addr_to_user(address, msg.msg_namelen, addr, addr_len);
1662        if (err2 < 0)
1663            err = err2;
1664    }
```

```
1665out_put:
1666        fput_light(sock_file, fput_needed);
1667out:
1668        return err;
1669}
```

Recepción de Mensajes (***sys_recv***)

Recibe un datagrama (en realidad un mensaje) de un socket.

<net/socket.c>

```
/*
1672 *      Receive a datagram from a socket.
1673 */
1674
1675asmlinkage long sys_recv(int fd, void __user *ubuf, size_t size,
1676                          unsigned flags)
1677{
1678        return sys_recvfrom(fd, ubuf, size, flags, NULL, NULL);
1679}
```

Interfaz BSD para ***recvmsg***

La interfaz BSD es la implementación al estilo BSD para garantizar la compatibilidad con la misma. Como se trata de una interfaz, la implementación final será la misma que para la recepción de mensajes al estilo UNIX con ***sys_recvto***; dicha implementación será común desde ***sock_recvmsg***.

<net/socket.c>

```
1878asmlinkage long sys_recvmsg(int fd, struct msghdr __user *msg,
```

```
1879                    unsigned int flags)
1880{
1881    struct compat_msghdr __user *msg_compat =
1882                (struct compat_msghdr __user *)msg;
1883    struct socket *sock;
1884          struct iovec iovstack[UIO_FASTIOV];
1885    struct iovec *iov = iovstack;
1886    struct msghdr msg_sys;
1887    unsigned long cmsg_ptr;
1888    int err, iov_size, total_len, len;
1889    int fput_needed;
1890
1891    /* kernel mode address */
1892    char addr[MAX_SOCK_ADDR];
1893
1894    /* user mode address pointers */
1895    struct sockaddr __user *uaddr;
1896    int __user *uaddr_len;
1897
1898    if (MSG_CMSG_COMPAT & flags) {
1899        if (get_compat_msghdr(&msg_sys, msg_compat))
1900            return -EFAULT;
1901    }
1902    else if (copy_from_user(&msg_sys, msg, sizeof(struct msghdr)))
1903        return -EFAULT;
1904   /* Obtener el socket*/
1905    sock = sockfd_lookup_light(fd, &err, &fput_needed);
1906    if (!sock)
1907        goto out;
1908
```

```
1909    err = -EMSGSIZE;
1910    if (msg_sys.msg_iovlen > UIO_MAXIOV) /* Chequear msg_sys es valido*/
1911            goto out_put;
1912
1913    /* Check whether to allocate the iovec area */
1914    err = -ENOMEM;
1915    iov_size = msg_sys.msg_iovlen * sizeof(struct iovec);
1916    if (msg_sys.msg_iovlen > UIO_FASTIOV) {
1917            iov = sock_kmalloc(sock->sk, iov_size, GFP_KERNEL);
1918            if (!iov)
1919                    goto out_put;
1920    }
/*
1923             *      Save the user-mode address (verify_iovec will change the
1924             *      kernel msghdr to use the kernel address space)
1925             */
1926
1927            uaddr = (void __user *)msg_sys.msg_name;
1928            uaddr_len = COMPAT_NAMELEN(msg);
1929            if (MSG_CMSG_COMPAT & flags) {
1930                    err = verify_compat_iovec(&msg_sys, iov, addr, VERIFY_WRITE);
1931            } else
1932                    err = verify_iovec(&msg_sys, iov, addr, VERIFY_WRITE);
1933            if (err < 0)
1934                    goto out_freeiov;
1935            total_len = err;
1936
1937            cmsg_ptr = (unsigned long)msg_sys.msg_control;
1938            msg_sys.msg_flags = flags & (MSG_CMSG_CLOEXEC|MSG_CMSG_COMPAT);
1939
1940            if (sock->file->f_flags & O_NONBLOCK)
1941                    flags |= MSG_DONTWAIT; /* Recivir mensaje*/
1942            err = sock_recvmsg(sock, &msg_sys, total_len, flags);
1943            if (err < 0)
1944                    goto out_freeiov;
1945            len = err;
1946  /* copiar el nombre del socket en el espacio de direcciones del usuario*/
1947     if (uaddr != NULL) {
1948                    err = move_addr_to_user(addr, msg_sys.msg_namelen, uaddr,
1949                                            uaddr_len);
1950                    if (err < 0)
1951                            goto out_freeiov;
1952            }
1953            err = __put_user((msg_sys.msg_flags & ~MSG_CMSG_COMPAT),
1954                             COMPAT_FLAGS(msg));
```

```
1955        if (err)
1956                goto out_freeiov;
/* Poner datos del mensaje en el espacio del usuario */
1957        if (MSG_CMSG_COMPAT & flags)
1958                err = __put_user((unsigned long)msg_sys.msg_control -
cmsg_ptr,
1959                                 &msg_compat->msg_controllen);
1960        else
1961                err = __put_user((unsigned long)msg_sys.msg_control -
cmsg_ptr,
1962                                 &msg->msg_controllen);
1963        if (err)
1964                goto out_freeiov;
1965        err = len;
1966
1967out_freeiov:
1968        if (iov != iovstack)
1969                sock_kfree_s(sock->sk, iov, iov_size);
1970out_put:
1971        fput_light(sock->file, fput_needed);
1972out:
1973        return err;
1974}
```

Recibir Mensaje (***sock_recvmsg***)

Primera función en las llamadas para envío de mensajes.

<net/socket.c>

```
641int sock_recvmsg(struct socket *sock, struct msghdr *msg,
642                  size_t size, int flags)
643{
644     struct kiocb iocb;
645     struct sock_iocb siocb;
646     int ret;
647
648          init_sync_kiocb(&iocb, NULL);
649     iocb.private = &siocb;  /* Recibir el mensaje*/
650     ret = __sock_recvmsg(&iocb, sock, msg, size, flags);
651     if (-EIOCBQUEUED == ret)
```

```
652             ret = wait_on_sync_kiocb(&iocb);
653         return ret; /* Controla el error*/
654}
```

Enviar Mensaje (__*sock_recvmsg*)

Segunda función en las llamadas para envío de mensajes. Desde ella se llama a la función concreta del protocolo usado para la comunicación.

<net/socket.c>

```
622static inline int __sock_recvmsg(struct kiocb *iocb, struct socket *sock,
623                         struct msghdr   *msg, size_t size, int flags)
624{
625         int err;
626         struct sock_iocb *si = kiocb_to_siocb(iocb);
627
628         si->sock = sock;
629         si->scm = NULL;
630         si->msg = msg;
631         si->size = size;
632         si->flags = flags;
633         /*Recepción con operación segurat*/
634         err = security_socket_recvmsg(sock, msg, size, flags);
635         if (err)
636                 return err;
637         /* Recepción del mensaje dependiente del protocolo*/
638         return sock->ops->recvmsg(iocb, sock, msg, size, flags);
639}
```

Fijar Opciones (*sys_setsockopt*)

Fija una opción del socket. Como no se saben las longitudes de las opciones se pasa el parámetro del modo de usuario para los protocolos, para ordenarlas.

<net/socket.c>

```
*
1682 *      Set a socket option. Because we don't know the option lengths we have
1683 *      to pass the user mode parameter for the protocols to sort out.
1684 */
1685
1686asmlinkage long sys_setsockopt(int fd, int level, int optname,
1687                                char __user *optval, int optlen)
1688{
1689     int err, fput_needed;
1690     struct socket *sock;
1691
1692     if (optlen < 0)
1693          return -EINVAL;
1694     /* Obtener el socket*/
1695     sock = sockfd_lookup_light(fd, &err, &fput_needed);
1696     if (sock != NULL) {
1697          err = security_socket_setsockopt(sock, level, optname);
1698          if (err)
1699               goto out_put;
1700    /* Fijar la opción*/
1701          if (level == SOL_SOCKET)
1702                              err =
1703                sock_setsockopt(sock, level, optname, optval,
1704                                optlen);
1705          else
```

```
1706                err =
1707                                        sock->ops->setsockopt(sock, level, optname, optval,
1708                                    optlen);
1709out_put:
1710            fput_light(sock->file, fput_needed);
1711        }
1712        return err;
1713}
```

Fijar Opciones (***sock_setsockopt***)

Implementación genérica para fijar opciones. Es la alternativa a las funciones específicas que defina cada protocolo, que se tiene en la estructura de operaciones ***sock→ops***.

<net/core/sock.c>

```
 425int sock_setsockopt(struct socket *sock, int level, int optname,
 426                    char __user *optval, int optlen)
 427{
 428        struct sock *sk=sock->sk;
 429        struct sk_filter *filter;
 430        int val;
 431        int valbool;
 432        struct linger ling;
 433        int ret = 0;
 434
 435        /*
 436         *      Options without arguments
 437         */
 438
 439#ifdef SO_DONTLINGER            /* Compatibility item... */
 440        if (optname == SO_DONTLINGER) {
 441                lock_sock(sk);
 442                sock_reset_flag(sk, SOCK_LINGER);
 443                release_sock(sk);
 444                return 0;
 445        }
 446#endif
 447
 448        if (optname == SO_BINDTODEVICE)
```

```
449                 return sock_bindtodevice(sk, optval, optlen);
450
451         if (optlen < sizeof(int))
452                 return -EINVAL;
453
454         if (get_user(val, (int __user *)optval))
455                 return -EFAULT;
456
457         valbool = val?1:0;
458
459         lock_sock(sk);
460
461         switch(optname) {
462         case SO_DEBUG:
463                 if (val && !capable(CAP_NET_ADMIN)) {
464                         ret = -EACCES;
465                 }
466                 else if (valbool)
467                         sock_set_flag(sk, SOCK_DBG);
468                 else
469                         sock_reset_flag(sk, SOCK_DBG);
470                 break;
471         case SO_REUSEADDR:
472                 sk->sk_reuse = valbool;
473                 break;
474         case SO_TYPE:
475         case SO_ERROR:
476                 ret = -ENOPROTOOPT;
477                 break;
478         case SO_DONTROUTE:
479                 if (valbool)
480                         sock_set_flag(sk, SOCK_LOCALROUTE);
481                 else
482                         sock_reset_flag(sk, SOCK_LOCALROUTE);
483                 break;
484         case SO_BROADCAST:
485                 sock_valbool_flag(sk, SOCK_BROADCAST, valbool);
486                 break;
487         case SO_SNDBUF:
488                 /* Don't error on this BSD doesn't and if you think
489                    about it this is right. Otherwise apps have to
490                    play 'guess the biggest size' games. RCVBUF/SNDBUF
491                    are treated in BSD as hints */
492
493                 if (val > sysctl_wmem_max)
494                         val = sysctl_wmem_max;
495set_sndbuf:
496                 sk->sk_userlocks |= SOCK_SNDBUF_LOCK;
497                 if ((val * 2) < SOCK_MIN_SNDBUF)
498                         sk->sk_sndbuf = SOCK_MIN_SNDBUF;
```

```
499                 else
500                         sk->sk_sndbuf = val * 2;
501
502                 /*
503                  *      Wake up sending tasks if we
504                  *      upped the value.
505                  */
506                 sk->sk_write_space(sk);
507                 break;
508
509         case SO_SNDBUFFORCE:
510                 if (!capable(CAP_NET_ADMIN)) {
511                         ret = -EPERM;
512                         break;
513                 }
514                 goto set_sndbuf;
515
516         case SO_RCVBUF:
517                 /* Don't error on this BSD doesn't and if you think
518                    about it this is right. Otherwise apps have to
519                    play 'guess the biggest size' games. RCVBUF/SNDBUF
520                    are treated in BSD as hints */
521
522                 if (val > sysctl_rmem_max)
523                         val = sysctl_rmem_max;
524set_rcvbuf:
525                 sk->sk_userlocks |= SOCK_RCVBUF_LOCK;
526                 /*
527                  * We double it on the way in to account for
528                  * "struct sk_buff" etc. overhead.   Applications
529                  * assume that the SO_RCVBUF setting they make will
530                  * allow that much actual data to be received on that
531                  * socket.
532                  *
533                  * Applications are unaware that "struct sk_buff" and
534                  * other overheads allocate from the receive buffer
535                  * during socket buffer allocation.
536                  *
537                  * And after considering the possible alternatives,
538                  * returning the value we actually used in getsockopt
539                  * is the most desirable behavior.
540                  */
541                 if ((val * 2) < SOCK_MIN_RCVBUF)
542                         sk->sk_rcvbuf = SOCK_MIN_RCVBUF;
543                 else
544                         sk->sk_rcvbuf = val * 2;
545                 break;
546
547         case SO_RCVBUFFORCE:
548                 if (!capable(CAP_NET_ADMIN)) {
```

```
549                             ret = -EPERM;
550                             break;
551                     }
552                     goto set_rcvbuf;
553
554             case SO_KEEPALIVE:
555#ifdef CONFIG_INET
556                     if (sk->sk_protocol == IPPROTO_TCP)
557                             tcp_set_keepalive(sk, valbool);
558#endif
559                     sock_valbool_flag(sk, SOCK_KEEPOPEN, valbool);
560                     break;
561
562             case SO_OOBINLINE:
563                     sock_valbool_flag(sk, SOCK_URGINLINE, valbool);
564                     break;
565
566             case SO_NO_CHECK:
567                     sk->sk_no_check = valbool;
568                     break;
569
570             case SO_PRIORITY:
571                     if ((val >= 0 && val <= 6) || capable(CAP_NET_ADMIN))
572                             sk->sk_priority = val;
573                     else
574                             ret = -EPERM;
575                     break;
576
577             case SO_LINGER:
578                     if (optlen < sizeof(ling)) {
579                             ret = -EINVAL;  /* 1003.1g */
580                             break;
581                     }
582                     if (copy_from_user(&ling,optval,sizeof(ling))) {
583                             ret = -EFAULT;
584                             break;
585                     }
586                     if (!ling.l_onoff)
587                             sock_reset_flag(sk, SOCK_LINGER);
588                     else {
589#if (BITS_PER_LONG == 32)
590                                     if ((unsigned int)ling.l_linger >=
MAX_SCHEDULE_TIMEOUT/HZ)
591                                             sk->sk_lingertime =
MAX_SCHEDULE_TIMEOUT;
592                             else
593#endif
594                                     sk->sk_lingertime = (unsigned
int)ling.l_linger * HZ;
595                             sock_set_flag(sk, SOCK_LINGER);
```

```
596                 }
597                 break;
598
599         case SO_BSDCOMPAT:
600                 sock_warn_obsolete_bsdism("setsockopt");
601                 break;
602
603         case SO_PASSCRED:
604                 if (valbool)
605                         set_bit(SOCK_PASSCRED, &sock->flags);
606                 else
607                         clear_bit(SOCK_PASSCRED, &sock->flags);
608                 break;
609
610         case SO_TIMESTAMP:
611         case SO_TIMESTAMPNS:
612                 if (valbool)  {
613                         if (optname == SO_TIMESTAMP)
614                                                 sock_reset_flag(sk,
SOCK_RCVTSTAMPNS);
615                         else
616                                 sock_set_flag(sk, SOCK_RCVTSTAMPNS);
617                         sock_set_flag(sk, SOCK_RCVTSTAMP);
618                         sock_enable_timestamp(sk);
619                 } else {
620                         sock_reset_flag(sk, SOCK_RCVTSTAMP);
621                         sock_reset_flag(sk, SOCK_RCVTSTAMPNS);
622                 }
623                 break;
624
625         case SO_RCVLOWAT:
626                 if (val < 0)
627                         val = INT_MAX;
628                 sk->sk_rcvlowat = val ? : 1;
629                 break;
630
631         case SO_RCVTIMEO:
632                         ret = sock_set_timeout(&sk->sk_rcvtimeo, optval,
optlen);
633                 break;
634
635         case SO_SNDTIMEO:
636                         ret = sock_set_timeout(&sk->sk_sndtimeo, optval,
optlen);
637                 break;
638
639         case SO_ATTACH_FILTER:
640                 ret = -EINVAL;
641                 if (optlen == sizeof(struct sock_fprog)) {
642                         struct sock_fprog fprog;
```

```
643
644                             ret = -EFAULT;
645                                 if (copy_from_user(&fprog, optval,
sizeof(fprog)))
646                                     break;
647
648                             ret = sk_attach_filter(&fprog, sk);
649                     }
650                     break;
651
652             case SO_DETACH_FILTER:
653                     rcu_read_lock_bh();
654                     filter = rcu_dereference(sk->sk_filter);
655                     if (filter) {
656                             rcu_assign_pointer(sk->sk_filter, NULL);
657                             sk_filter_release(sk, filter);
658                             rcu_read_unlock_bh();
659                             break;
660                     }
661                     rcu_read_unlock_bh();
662                     ret = -ENONET;
663                     break;
664
665             case SO_PASSSEC:
666                     if (valbool)
667                             set_bit(SOCK_PASSSEC, &sock->flags);
668                     else
669                             clear_bit(SOCK_PASSSEC, &sock->flags);
670                     break;
671
672                     /* We implement the SO_SNDLOWAT etc to
673                        not be settable (1003.1g 5.3) */
674             default:
675                     ret = -ENOPROTOOPT;
676                     break;
677             }
678             release_sock(sk);
679             return ret;
680}
```

Tomar Opciones (*sys_getsockopt*)

Obtener una opción del socket. Como no se saben las longitudes de las opciones se pasa el parámetro del modo de usuario para los protocolos, para ordenarlas.

<net/socket.c>

```
1715/*
1716 *     Get a socket option. Because we don't know the option lengths we have
1717 *     to pass a user mode parameter for the protocols to sort out.
1718 */
1719
1720asmlinkage long sys_getsockopt(int fd, int level, int optname,
1721                               char __user *optval, int __user *optlen)
1722{
1723      int err, fput_needed;
1724      struct socket *sock;
1725     /*Obtener el socket */
1726      sock = sockfd_lookup_light(fd, &err, &fput_needed);
1727              if (sock != NULL) {
1728           err = security_socket_getsockopt(sock, level, optname);
1729           if (err)
1730                goto out_put;
1731     /* Tomar la opción */
1732           if (level == SOL_SOCKET)
1733                err =
1734                  sock_getsockopt(sock, level, optname, optval,
1735                                  optlen);
1736                       else
1737                err =
1738                  sock->ops->getsockopt(sock,  level, optname, optval,
1739                                        optlen);
1740out_put:
1741           fput_light(sock->file, fput_needed);
1742      }
1743      return err;
1744}
```

Tomar Opciones (***sock_getsockopt***)

Implementación genérica para tomar opciones. Es la alternativa a las funciones específicas que defina cada protocolo, que se tiene en la estructura de operaciones ***sock→ops***.

<net/core/sock.c>

```
/*Pensado para todos los protocolos (genérico, a nivel de socket) */
683int sock_getsockopt(struct socket *sock, int level, int optname,
684                    char __user *optval, int __user  *optlen)
685{
686     struct sock *sk = sock->sk;
687
688     union {
689          int val;
690          struct linger  ling;
691          struct timeval tm;
692     } v;
693
694     unsigned int lv = sizeof(int);
695     int len;
696
697     if (get_user(len, optlen))
698          return -EFAULT;
699     if (len < 0)
700          return -EINVAL;
701     /* Tratar cada opción de forma genérica para todos los protocolos */
702     switch(optname) {
703     case SO_DEBUG:
```

```
704             v.val = sock_flag(sk, SOCK_DBG);
705             break;
706
707     case SO_DONTROUTE:
708             v.val = sock_flag(sk, SOCK_LOCALROUTE);
709             break;
710
711     case SO_BROADCAST:
712             v.val = !!sock_flag(sk, SOCK_BROADCAST);
713             break;
714
715     case SO_SNDBUF:
716             v.val = sk->sk_sndbuf;
717             break;
718
719               case SO_RCVBUF:
720             v.val = sk->sk_rcvbuf;
721                       break;
722
723     case SO_REUSEADDR:
724             v.val = sk->sk_reuse;
725             break;
726
727     case SO_KEEPALIVE:
728             v.val = !!sock_flag(sk, SOCK_KEEPOPEN);
729             break;
730
731     case SO_TYPE:
732             v.val = sk->sk_type;
733             break;
```

```
734
735     case SO_ERROR:
736             v.val = -sock_error(sk);
737             if (v.val==0)
738                     v.val = xchg(&sk->sk_err_soft, 0);
739             break;
740
741     case SO_OOBINLINE:
742             v.val = !!sock_flag(sk, SOCK_URGINLINE);
743             break;
744
745     case SO_NO_CHECK:
746             v.val = sk->sk_no_check;
747             break;
748
749     case SO_PRIORITY:
750             v.val = sk->sk_priority;
751             break;
752
753     case SO_LINGER:
754             lv              = sizeof(v.ling);
755             v.ling.l_onoff  = !!sock_flag(sk, SOCK_LINGER);
756             v.ling.l_linger = sk->sk_lingertime / HZ;
757             break;
758
759     case SO_BSDCOMPAT:
760             sock_warn_obsolete_bsdism("getsockopt");
761             break;
762
763     case SO_TIMESTAMP:
```

```
764             v.val = sock_flag(sk, SOCK_RCVTSTAMP) &&
765                             !sock_flag(sk, SOCK_RCVTSTAMPNS);
766             break;
767
768     case SO_TIMESTAMPNS:
769             v.val = sock_flag(sk, SOCK_RCVTSTAMPNS);
770                             break;
771
772     case SO_RCVTIMEO:
773             lv=sizeof(struct timeval);
774             if (sk->sk_rcvtimeo == MAX_SCHEDULE_TIMEOUT) {
775                     v.tm.tv_sec = 0;
776                     v.tm.tv_usec = 0;
777             } else {
778                     v.tm.tv_sec = sk->sk_rcvtimeo / HZ;
779                                             v.tm.tv_usec = ((sk->sk_rcvtimeo % HZ) * 1000000) / HZ;
780             }
781             break;
782
783     case SO_SNDTIMEO:
784             lv=sizeof(struct timeval);
785             if (sk->sk_sndtimeo == MAX_SCHEDULE_TIMEOUT) {
786                                             v.tm.tv_sec = 0;
787                     v.tm.tv_usec = 0;
788             } else {
789                     v.tm.tv_sec = sk->sk_sndtimeo / HZ;
790                 v.tm.tv_usec = ((sk->sk_sndtimeo % HZ) * 1000000) / HZ;
791             }
792             break;
793
```

```
794     case SO_RCVLOWAT:
795             v.val = sk->sk_rcvlowat;
796             break;
797
798     case SO_SNDLOWAT:
799             v.val=1;
800             break;
801
802     case SO_PASSCRED:
803             v.val  =  test_bit(SOCK_PASSCRED, &sock->flags) ? 1 : 0;
804             break;
805
806     case SO_PEERCRED:
807             if (len > sizeof(sk->sk_peercred))
808                     len = sizeof(sk->sk_peercred);
809             if (copy_to_user(optval, &sk->sk_peercred, len))
810                     return -EFAULT; /* Pasar el valor de la opción al espacio de usuario */
811             goto lenout;
812
813     case SO_PEERNAME:
814     {
815             char address[128];
816
817        if (sock->ops->getname(sock, (struct sockaddr *)address, &lv, 2))
818                     return -ENOTCONN;
819             if (lv < len)
820                                     return -EINVAL;
821             if (copy_to_user(optval, address, len))
822                     return -EFAULT; /* Pasar el valor de la opción al espacio de usuario */
823             goto lenout;
```

```
824     }
/* Dubious BSD thing... Probably nobody even uses it, but
827          * the UNIX standard wants it for whatever reason... -DaveM
828          */
829         case SO_ACCEPTCONN:
830                 v.val = sk->sk_state == TCP_LISTEN;
831                 break;
832
833         case SO_PASSSEC:
834                 v.val = test_bit(SOCK_PASSSEC, &sock->flags) ? 1 : 0;
835                 break;
836
837         case SO_PEERSEC:
838                  return security_socket_getpeersec_stream(sock, optval,
optlen, len);
839
840         default:
841                 return -ENOPROTOOPT;
842         }
843
844         if (len > lv)
845                 len = lv;
846         if (copy_to_user(optval, &v, len))
847                 return -EFAULT;
848lenout:
849         if (put_user(len, optlen))
850                 return -EFAULT;
851         return 0;
852}
```

Cerrar (***sys_shutdown***)

Cierra un socket, pasando al estado CLOSE. Equivale a cerrar el fichero que mantiene el socket.

La llamada al sistema ***shutdown*** proporciona gran control sobre una conexión full-duplex.

Si el argumento ***how*** es 0, ningún dato más puede ser recibido en el socket. Un valor de 1 causa que ninguna salida mas sea permitida en el socket. Un valor 2, provoca que tanto la salida como la entrada de datos en el socket sea desactivada. Es importante recordar que un socket es normalmente un canal de comunicación bidireccional. Los datos que viajan en una dirección son independientes de los datos que van en la otra dirección. Por esto es por

lo que ***shutdown***, permite cerrar cualquiera de las dos direcciones, independientemente de la otra.

<net/socket.c>

```
1746/*
1747 *      Shutdown a socket.
1748 */
1749
1750asmlinkage long sys_shutdown(int fd, int how)
1751{
1752        int err, fput_needed;
1753        struct socket *sock;
1754      /* Obtener el socket */
1755        sock = sockfd_lookup_light(fd, &err, &fput_needed);
1756        if (sock != NULL) {
1757                err = security_socket_shutdown(sock, how);
1758                if (!err) /* Cerrar el socket de forma dependiente del protocolo */
1759                        err = sock->ops->shutdown(sock, how);
1760                fput_light(sock->file, fput_needed);
1761        }
1762        return err;
1763}
```

Funciones Auxiliares

Obtener Socket (***sockfd_lookup***)

Permite el paso de un descriptor de fichero a su slot de socket, es decir, se obtiene el socket a partir del descriptor de fichero que realmente mantiene el socket. Los parámetros de esta función son:

fd: descriptor de fichero

err: puntero a código de error devuelto

El descriptor de fichero pasado se protege con un cerrojo y el socket se retorna finalmente. Si ocurre un error, el puntero err se sobreescribe con un código errno negativo y se devuelve NULL.

La función chequea descriptores no válidos y que no sean socket. Si todo va bien se devuelve un puntero al socket.

```
/**
421 *     sockfd_lookup  -      Go from a file number to its socket slot
422  *         @fd: file handle
423 *     @err: pointer to an error code return
424 *
425 *     The file handle passed in is locked and the socket it is bound
426 *     too is returned. If an error occurs the err pointer is overwritten
427 *     with a negative errno code and NULL is returned. The function checks
428 *     for both invalid handles and passing a handle which is not a socket.
429 *
430 *     On a success the socket object pointer is returned.
431 */
432
433struct socket *sockfd_lookup(int fd, int *err)
434{
435             struct  file *file;
436       struct socket *sock;
437
438       file = fget(fd);
439       if (!file) {
440             *err = -EBADF;
441             return NULL;
```

```
442     }
443
444     sock = sock_from_file(file, err);
445     if (!sock)
446             fput(file);
447     return sock;
448}
```