
Índice

1. Introducción	3
Historia.....	3
Definición.....	4
Los sockets orientados a connexion (TCP).....	4
API (Funciones)	5
API (Estructuras).....	6
Estructura Genérica.....	6
Dominio UNIX (AF_UNIX).....	6
Dominio Internet (AF_INET)	7
Propiedades	8
Tipos.....	8
SOCK_DGRAM	8
SOCK_STREAM.....	9
SOCK_RAW.....	9
2. Código.....	10
Estructura <i>sock_common</i>	10
Estructura <i>sock</i>	10
Llamada del sistema <i>sys_socketcall</i>	12
Creación (<i>sys_socket</i>)	14
Unión socket-dirección (<i>sys_bind</i>)	16
Escucha (<i>sys_listen</i>)	16
Acepta una conexión (<i>sys_accept</i>)	17
Conecta a un servidor (<i>sys_connect</i>)	18
Obtener la dirección local (<i>sys_getsockname</i>)	19
Obtener la dirección remota (<i>sys_getpeername</i>)	19
Crea un par de sockets conectados (<i>sys_socketpair</i>)	20
Envío de Mensajes (<i>sys_sendto</i>)	21
Envío de Mensajes (<i>sys_send</i>)	21
Interfaz BSD para <i>sendmsg</i>	22

Enviar Mensaje (<i>sock_sendmsg</i>)	22
Enviar Mensaje (<i>__sock_sendmsg</i>)	23
Recepción de Mensajes (<i>sys_recvfrom</i>)	23
Recepción de Mensajes (<i>sys_recv</i>)	24
Interfaz BSD para <i>recvmsg</i>	24
Recibir Mensaje (<i>sock_recvmsg</i>)	24
Enviar Mensaje (<i>__sock_recvmsg</i>)	25
Fijar Opciones (<i>sys_setsockopt</i>)	25
Fijar Opciones (<i>sock_setsockopt</i>)	26
Tomar Opciones (<i>sys_getsockopt</i>)	26
Tomar Opciones (<i>sock_getsockopt</i>)	27
Cerrar (<i>sys_shutdown</i>)	27
Funciones Auxiliares.....	28
Obtener Socket (<i>sockfd_lookup</i>)	28

Autores:

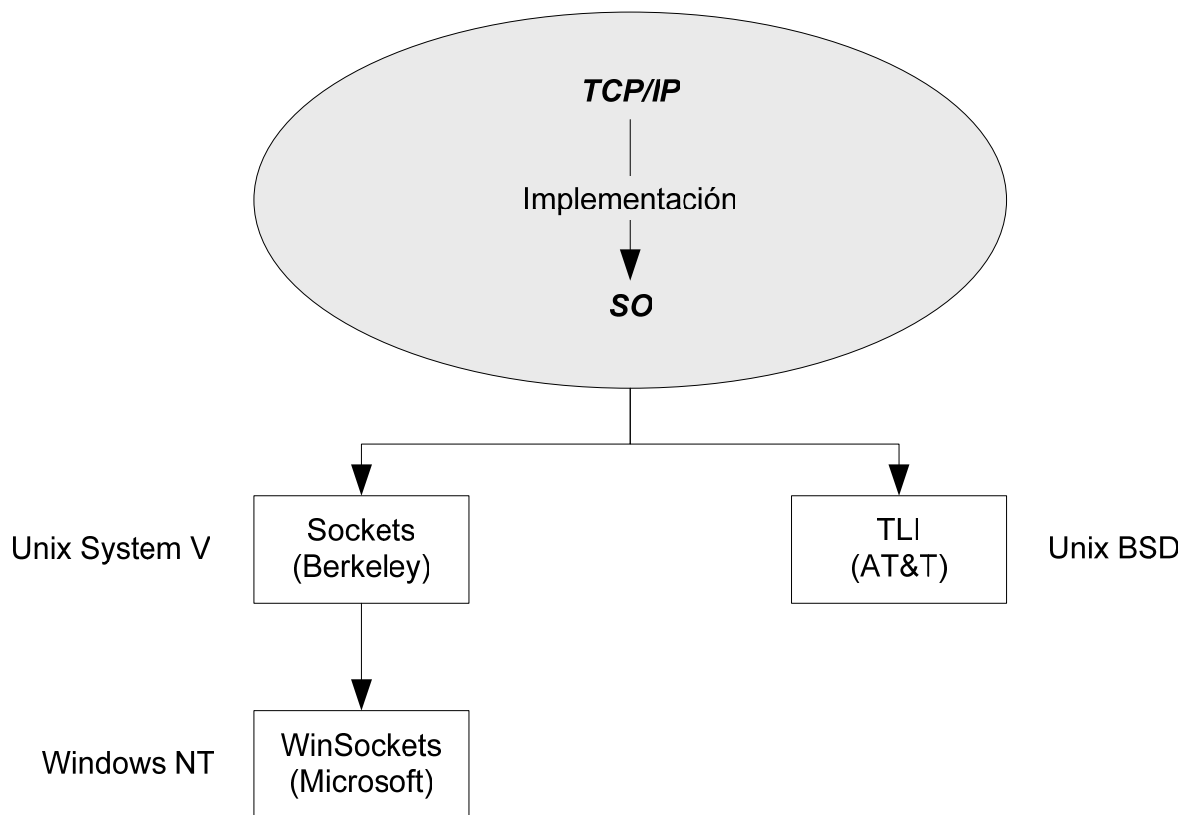
Fernández Perdomo, Enrique

Horat Flotats, David J.

1. Introducción

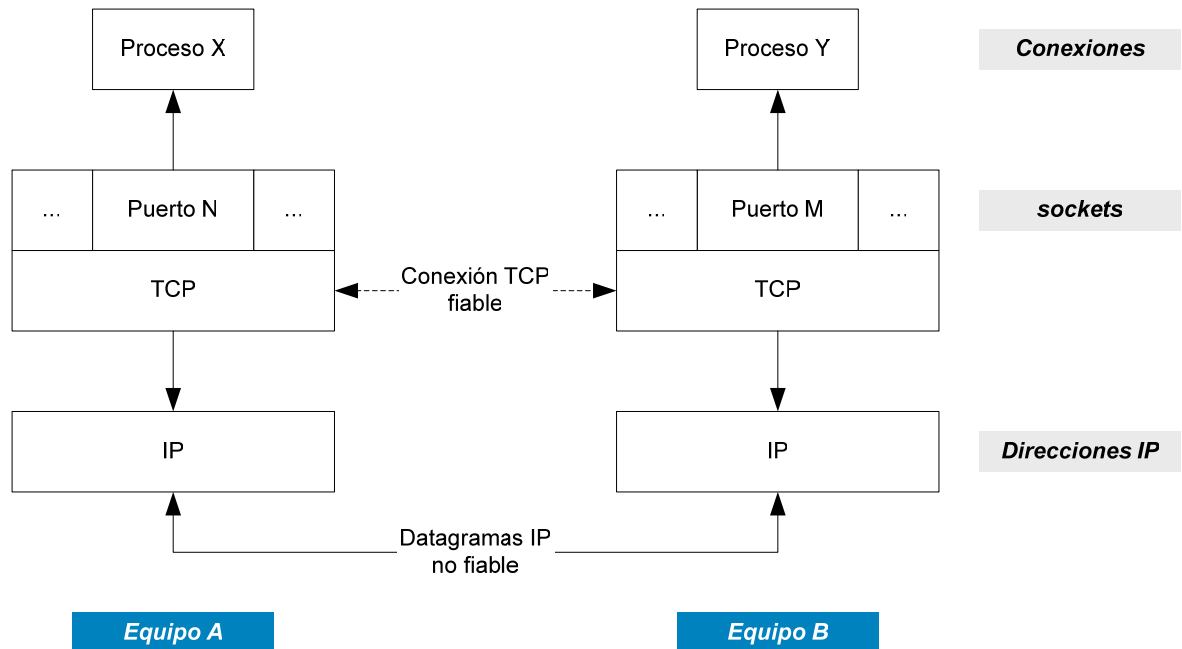
Historia

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

Un socket es un punto de comunicación, identificado por un descriptor. Dicho descriptor es igual al de entrada / salida estándar. 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).

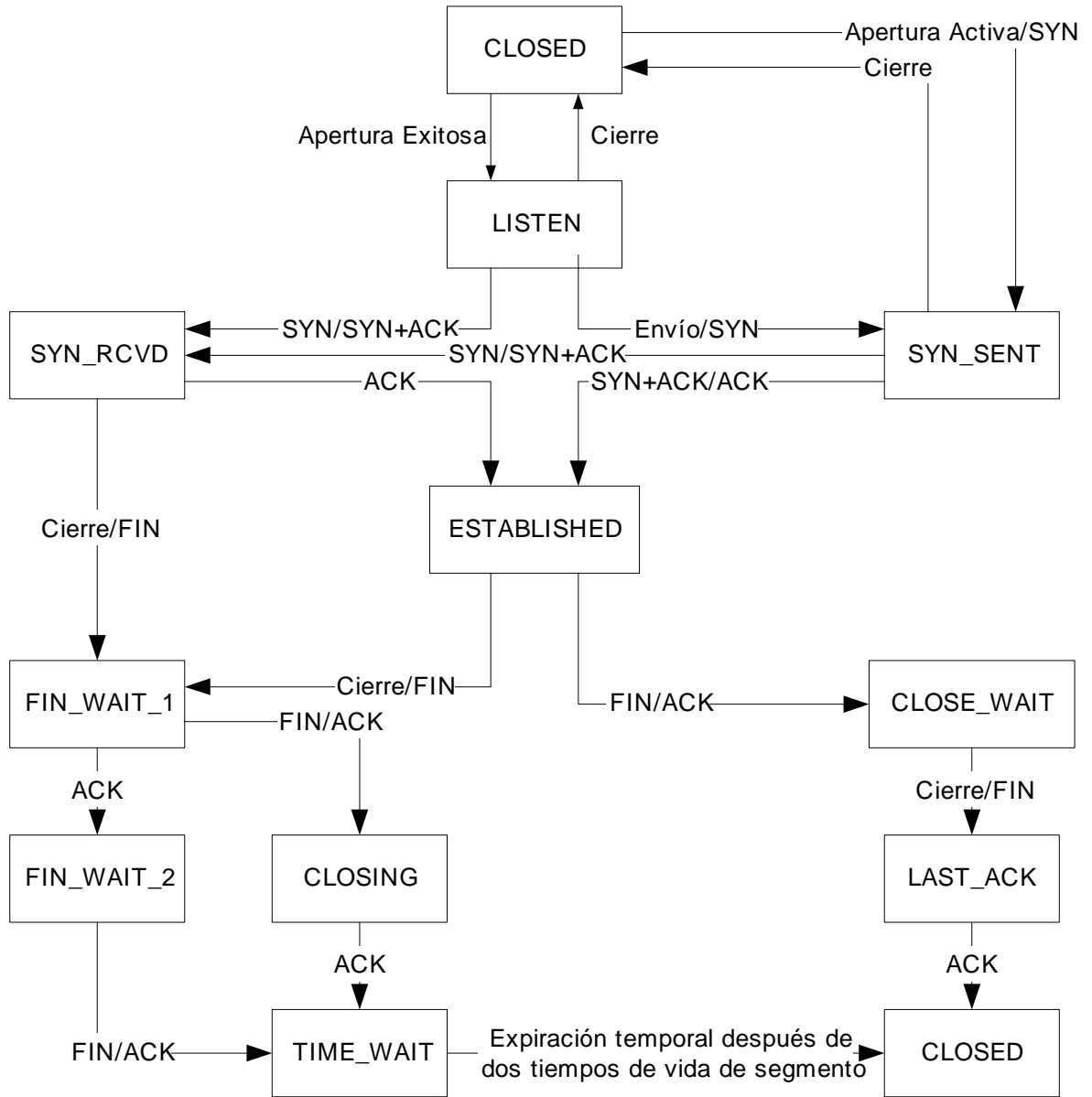


Existen dos tipos de sockets:

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

Los sockets orientados a conexión (TCP)

Este tipo de sockets están orientado a ristas (stream). Manejan un buffer para el envío y recepción de datos. Permite la conexión por Circuito Virtual y en modo Full-duplex.



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.

```
struct sockaddr {  
    u_short sa_family; /* familia de dirección */  
    char sa_data[14]; /* 14 octetos de dirección (máximo) */  
};
```

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. Esta estructura está predefinida en `<sys/un.h>`.

```
struct sockaddr_un{
    short sun_family;    /* dominio UNIX: AF_UNIX */
    char  sun_data[108]; /* referencia de dirección */
};
```

Dominio Internet (AF_INET)

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

```
struct in_addr{
    u_long s_addr;
};
```

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 ejemplo: `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.

```
struct sockaddr_in{
    short  sin_family;    /* familia de dirección: AF_INET */
    u_short sin_port;    /* el número de puerto */
};
```

```
struct in_addr sin_addr; /* la dirección Internet */
char    sin_zero[8];     /* un campo de 8 ceros */
};
```

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.

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

```
105 struct sock_common {
106     unsigned short      skc_family;
107     volatile unsigned char skc_state;
108     unsigned char       skc_reuse;
109     int                  skc_bound_dev_if;
110     struct hlist_node    skc_node;
111     struct hlist_node    skc_bind_node;
112     atomic_t             skc_refcnt;
113 };
```

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 una 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

```

181 struct sock {
186     struct sock_common    __sk_common;
195     unsigned char        sk_shutdown;
198     socket_lock_t        sk_lock;
199     int                   sk_rcvbuf;
200     wait_queue_head_t     *sk_sleep;
201     struct dst_entry      *sk_dst_cache;
202     rwlock_t              sk_dst_lock;
203     struct xfrm_policy    *sk_policy[2];
204     atomic_t              sk_rmem_alloc;
205     struct sk_buff_head   sk_receive_queue;
206     atomic_t              sk_wmem_alloc;
207     struct sk_buff_head   sk_write_queue;
209     int                   sk_wmem_queued;
210     int                   sk_forward_alloc;
211     unsigned int          sk_allocation;
212     int                   sk_sndbuf;
217     unsigned char        sk_no_large_send;
218     int                   sk_route_caps;
226     struct {
227         struct sk_buff *head;
228         struct sk_buff *tail;
229     } sk_backlog;
232     struct proto          *sk_prot;
233     int                   sk_err;
237     __u32                 sk_priority;
238     unsigned short        sk_type;
239     unsigned char         sk_localroute;
240     unsigned char         sk_protocol;
245     struct sk_filter      *sk_filter;
248     struct timer_list     sk_timer;
249     struct timeval        sk_stamp;
250     struct socket         *sk_socket;
251     void                  *sk_user_data;
252     struct module         *sk_owner;
253     struct page           *sk_sndmsg_page;
254     __u32                 sk_sndmsg_off;
267 };

```

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.

```

1899 asmlinkage long sys_socketcall(int call, unsigned long __user *args)
1900 {
1915     switch(call)
1916     {
1917         case SYS_SOCKET:
1918             err = sys_socket(a0,a1,a[2]); break;
1920         case SYS_BIND:
1921             err = sys_bind(a0,(struct sockaddr __user
1922 *)a1, a[2]); break;
1923         case SYS_CONNECT:
1924             err = sys_connect(a0, (struct sockaddr
1925 __user *)a1, a[2]); break;
1926         case SYS_LISTEN:
1927             err = sys_listen(a0,a1); break;
1929         case SYS_ACCEPT:
1930             err = sys_accept(a0,(struct sockaddr __user
1931 *)a1, (int __user *)a[2]); break;
1932         case SYS_GETSOCKNAME:
1933             err = sys_getsockname(a0,(struct sockaddr
1934 __user *)a1, (int __user *)a[2]); break;
1935         case SYS_GETPEERNAME:
1936             err = sys_getpeername(a0, (struct sockaddr
1937 __user *)a1, (int __user *)a[2]); break;
1938         case SYS_SOCKETPAIR:
1939             err = sys_socketpair(a0,a1, a[2], (int
1940 __user *)a[3]); break;
1941         case SYS_SEND:
1942             err = sys_send(a0, (void __user *)a1, a[2],
1943 a[3]); break;
1944         case SYS_SENDTO:
1945             err = sys_sendto(a0,(void __user *)a1, a[2],
1946 a[3],
1947             (struct sockaddr __user
1948 *)a[4], a[5]); break;
1949         case SYS_RECV:
1950             err = sys_recv(a0, (void __user *)a1, a[2],
1951 a[3]); break;
1952         case SYS_RECVFROM:
1953             err = sys_recvfrom(a0, (void __user *)a1,
1954 a[2], a[3],
1955             (struct sockaddr __user
1956 *)a[4], (int __user *)a[5]); break;
1957         case SYS_SHUTDOWN:
1958             err = sys_shutdown(a0,a1); break;
1959         case SYS_SETSOCKOPT:
1960             err = sys_setsockopt(a0, a1, a[2], (char
1961 __user *)a[3], a[4]); break;
1962         case SYS_GETSOCKOPT:
1963             err = sys_getsockopt(a0, a1, a[2], (char
1964 __user *)a[3], (int __user *)a[4]); break;
1965         case SYS_SENDMSG:
1966             err = sys_sendmsg(a0, (struct msghdr __user
1967 *) a1, a[2]); break;
1968         case SYS_RECVMSG:

```

```

1968                                     err = sys_recvmsg(a0, (struct msghdr __user
*) a1, a[2]); break;
1970                                     default:
1971                                     err = -EINVAL;
1973                                     }
1974                                     return err;
1975 }

```

Creación (**sys_socket**)

Para crear un socket llamamos a la función `sys_socket`, a la cual le pasamos la familia, el tipo y el protocolo que deseamos usar. Esta función nos devuelve el descriptor del socket, que es el que usaremos con las otras funciones.

```

1182 asmlinkage long sys_socket(int family, int type, int protocol)
1183 {
1184     int retval;
1185     struct socket *sock;
1186
1187     retval = sock_create(family, type, protocol, &sock);
1188     if (retval < 0)
1189         goto out;
1190
1191     retval = sock_map_fd(sock);
1192     if (retval < 0)
1193         goto out_release;
1194
1195 out:
1196     /* Ya hay otro descriptor. No es un problema del kernel */
1197     return retval;
1198
1199 out_release:
1200     sock_release(sock);
1201     return retval;
1202 }

1172 int sock_create(int family, int type, int protocol, struct socket
**res)
1173 {
1174     return __sock_create(family, type, protocol, res, 0);
1175 }

1071 static int __sock_create(int family, int type, int protocol, struct
socket **res, int kern) {

```

```

1073     int err; struct socket *sock;
1077     /* Comprobación de que los parámetros son correctos */
1079     if (family < 0 || family >= IPPROTO) return -EAFNOSUPPORT;
1081     if (type < 0 || type >= SOCK_MAX) return -EINVAL;
1083     /* Compatibilidad hacia atrás */
1089     if (family == PF_INET && type == SOCK_PACKET) {
1090         static int warned;
1091         if (!warned) {
1092             warned = 1;
1093             printk(KERN_INFO "%s uses obsolete
(PF_INET,SOCK_PACKET)\n", current->comm);
1094         }
1095         family = PF_PACKET;
1096     }
1097     /* Creamos el socket */
1098     err = security_socket_create(family, type, protocol, kern);
1099     if (err) return err;
1101     /* Obtenemos el cerrojo */
1115     net_family_read_lock();
1116     if (net_families[family] == NULL) {
1117         err = -EAFNOSUPPORT;
1118         goto out;
1119     }
1121     /* Pedimos un descriptor de sockets al sistema */
1127     if (!(sock = sock_alloc())) {
1128         printk(KERN_WARNING "socket: no more sockets\n");
1129         err = -ENFILE;
1131         goto out;
1132     }
1134     sock->type = type;
1135
1136     /* Comprobamos que los propietarios de los módulos de la familia son correctos */
1140     err = -EAFNOSUPPORT;
1141     if (!try_module_get(net_families[family]->owner))
1142         goto out_release;
1143
1144     if ((err = net_families[family]->create(sock, protocol)) <
0) /* Creamos el socket según la familia */
1145         goto out_module_put;
1146
1150     if (!try_module_get(sock->ops->owner)) {
1151         sock->ops = NULL;
1152         goto out_module_put;
1153     }
1158     module_put(net_families[family]->owner);
1159     *res = sock;
1160     security_socket_post_create(sock, family, type, protocol,
kern);
1161
1162 out:
1163     net_family_read_unlock();
1164     return err;
1165 out_module_put:
1166     module_put(net_families[family]->owner);
1167 out_release:
1168     sock_release(sock);

```

```

1169         goto out;
1170     }

```

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

Una vez creador, deberemos especificar qué dirección usaremos con el 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.

```

1278 asmlinkage long sys_bind(int fd, struct sockaddr __user *umyaddr,
1279 int addrlen)
1279 {
1280     struct socket *sock;
1281     char address[MAX_SOCKET_ADDR];
1282     int err;
1283
1284     if((sock = sockfd_lookup(fd,&err))!=NULL)
1285     {
1286
1287         if((err=move_addr_to_kernel(umyaddr,addrlen,address))>=0) {
1288             err = security_socket_bind(sock, (struct
1289             sockaddr *)address, addrlen);
1290             if (err) {
1291                 sockfd_put(sock);
1292                 return err;
1293             }
1294             err = sock->ops->bind(sock, (struct sockaddr
1295             *)address, addrlen);
1296             sockfd_put(sock);
1297         }
1298     }
1299     return err;
1300 }

```

Escucha (***sys_listen***)

En esta función, dejamos que sea el protocolo el que realice lo necesario para la escucha y lo marcamos como listo para escuchar.

```

1306 int sysctl_somaxconn = SOMAXCONN;
1308 asmlinkage long sys_listen(int fd, int backlog){
1310     struct socket *sock; int err;
1312
1313     if ((sock = sockfd_lookup(fd, &err)) != NULL) {
1314         if ((unsigned) backlog > sysctl_somaxconn)
1315             backlog = sysctl_somaxconn;
1317         err = security_socket_listen(sock, backlog);
1318         if (err) {
1319             sockfd_put(sock);
1320             return err;
1321         }
1323         err=sock->ops->listen(sock, backlog);
1324         sockfd_put(sock);
1325     }
1326     return err;
1327 }

```

Acepta una conexión (**sys_accept**)

Una vez recibimos una petición de conexión, porque anteriormente hemos estado escuchando, tendremos que aceptar dicha conexión. Para ello creamos un nuevo socket que será el que se enlace con el cliente y devolvemos su identificador.

```

1342 asmlinkage long sys_accept(int fd, struct sockaddr __user
*upeer_sockaddr, int __user *upeer_addrlen){
1344     struct socket *sock, *newsock; int err, len; char
address[MAX_SOCKET_ADDR];
1347     /* Buscamos el socket */
1348     sock = sockfd_lookup(fd, &err);
1349     if (!sock) goto out;
1351     /* Creamos un nuevo socket */
1352     err = -ENFILE;
1353     if (!(newsock = sock_alloc())) goto out_put;
1355     newsock->type = sock->type;
1357     newsock->ops = sock->ops;
1359     err = security_socket_accept(sock, newsock);
1360     if (err) goto out_release;
1362
1363     /* No hace falta probar el módulo, ya que el socket
1364     * ya tiene el módulo del protocolo (sock->ops->owner) */
1367     __module_get(newsock->ops->owner);
1368
1369     err = sock->ops->accept(sock, newsock, sock->file->f_flags);
1370     if (err < 0) goto out_release;
1373     if (upeer_sockaddr) {
1374         if(newsock->ops->getname(newsock, (struct sockaddr
*)address, &len, 2)<0){
1375             err = -ECONNABORTED; goto out_release;

```

```

1377         }
1378         err = move_addr_to_user(address, len,
upeer_sockaddr, upeer_addrlen);
1379         if (err < 0) goto out_release;
1381     }
1382
1383     /* Pedimos un descriptor al SO */
1385     if ((err = sock_map_fd(newsock)) < 0) goto out_release;
1387
1388     security_socket_post_accept(sock, newsock);
1389
1390 out_put:
1391     sockfd_put(sock);
1392 out:
1393     return err;
1394 out_release:
1395     sock_release(newsock);
1396     goto out_put;
1397 }

```

Conecta a un servidor (*sys_connect*)

Si somos clientes, queremos conectar con un servidor, para lo cual usaremos esta función. Básicamente movemos la dirección del espacio de usuario al espacio del núcleo una vez comprobada ya que es allí donde se realiza la conexión. La conexión depende del protocolo, por lo que será este el encargado de casi toda su gestión.

```

1412 asmlinkage long sys_connect(int fd, struct sockaddr __user
*servaddr, int addrlen){
1414     struct socket *sock; char address[MAX_SOCKET_ADDR]; int err;
1417
1418     sock = sockfd_lookup(fd, &err);
1419     if (!sock) goto out;
1421     err = move_addr_to_kernel(servaddr, addrlen, address);
1422     if (err < 0) goto out_put;
1424
1425     err = security_socket_connect(sock, (struct sockaddr
*)address, addrlen);
1426     if (err) goto out_put;
1428
1429     err = sock->ops->connect(sock, (struct sockaddr *) address,
addrlen, sock->file->f_flags);
1431 out_put:
1432     sockfd_put(sock);
1433 out:
1434     return err;
1435 }

```

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.

```
1442 asmlinkage long sys_getsockname(int fd, struct sockaddr __user
*usockaddr, int __user *usockaddr_len){
1444     struct socket *sock; char address[MAX_SOCKET_ADDR]; int len,
err;
1447
1448     sock = sockfd_lookup(fd, &err);
1449     if (!sock) goto out;
1452     err = security_socket_getsockname(sock);
1453     if (err) goto out_put;
1456     err = sock->ops->getname(sock, (struct sockaddr *)address,
&len, 0);
1457     if (err) goto out_put;
1459     err = move_addr_to_user(address, len, usockaddr,
usockaddr_len);
1461 out_put:
1462     sockfd_put(sock);
1463 out:
1464     return err;
1465 }
```

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.

```
1472 asmlinkage long sys_getpeername(int fd, struct sockaddr __user
*usockaddr, int __user *usockaddr_len){
1474     struct socket *sock; char address[MAX_SOCKET_ADDR]; int len,
err;
1477
1478     if ((sock = sockfd_lookup(fd, &err))!=NULL)
1479     {
1480         err = security_socket_getpeername(sock);
1481         if (err) {sockfd_put(sock); return err;}
1486         err = sock->ops->getname(sock, (struct sockaddr
*)address, &len, 1);
1487         if (!err) err=move_addr_to_user(address, len,
usockaddr, usockaddr_len);
1489         sockfd_put(sock);
```

```

1490     }
1491     return err;
1492 }

```

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

Crearemos dos sockets y los conectamos entre ellos. Asignamos los descriptores creados a los espacios de usuario especificados. Devolvemos el número de error o 0 en caso de éxito.

```

1208 asmlinkage long sys_socketpair(int family, int type, int protocol,
int __user *usockvec){
1210     struct socket *sock1, *sock2; int fd1, fd2, err;
1212
1213     /* Creamos los 2 sockets */
1218     err = sock_create(family, type, protocol, &sock1);
1219     if (err < 0) goto out;
1222     err = sock_create(family, type, protocol, &sock2);
1223     if (err < 0) goto out_release_1;
1224     /* Conectamos ambos sockets */
1226     err = sock1->ops->socketpair(sock1, sock2);
1227     if (err < 0) goto out_release_both;
1228     /* Obtenemos los descriptores (fd1 y fd2) de ambos sockets */
1230     fd1 = fd2 = -1;
1232     err = sock_map_fd(sock1);
1233     if (err < 0) goto out_release_both;
1235     fd1 = err;
1237     err = sock_map_fd(sock2);
1238     if (err < 0) goto out_close_1;
1240     fd2 = err;
1245     /* Asignamos los descriptores a los usuarios y salimos si no hay error */
1246     err = put_user(fd1, &usockvec[0]);
1247     if (!err) err = put_user(fd2, &usockvec[1]);
1249     if (!err) return 0;
1251     /* En caso de error, cerramos los descriptores y devolvemos el error */
1252     sys_close(fd2);
1253     sys_close(fd1);
1254     return err;
1255
1256 out_close_1:
1257     sock_release(sock2);
1258     sys_close(fd1);
1259     return err;
1261 out_release_both:
1262     sock_release(sock2);
1263 out_release_1:
1264     sock_release(sock1);
1265 out:
1266     return err;

```

1267 }

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.

```

1500 asmlinkage long sys_sendto(int fd, void __user * buff, size_t len,
unsigned flags,
1501         struct sockaddr __user *addr, int
addr_len)
1502 {
        // PASO 1: Obtener el socket
1509     sock = sockfd_lookup(fd, &err);
        // PASO 2: Construir mensaje
1514     msg.msg_name=NULL;           // Por defecto es un send (no hay
destinatario)
1519     msg.msg_namelen=0;           // Por defecto es un send (no hay
destinatario)
1520     if(addr) // Si hay destinatario
1521     {
1522         err = move_addr_to_kernel(addr, addr_len, address);
1525         msg.msg_name=address;
1526         msg.msg_namelen=addr_len;
1527     }
// PASO 3: Enviar mensaje por el socket
1531     err = sock_sendmsg(sock, &msg, len);
1533 out_put:
1534     sockfd_put(sock); // Libera socket usado localmente (sock)
1535 out:
1536     return err;
1537 }

```

Envío de Mensajes (**sys_send**)

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

```

1543 asmlinkage long sys_send(int fd, void __user * buff, size_t len,
unsigned flags)
1544 {
1545     return sys_sendto(fd, buff, len, flags, NULL, 0);
1546 }

```

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*.

```

1693 asmlinkage long sys_sendmsg(int fd, struct msghdr __user *msg,
unsigned flags)
1694 {
    // PASO 1: Obtener el socket
1711     sock = sockfd_lookup(fd, &err);
1715     // PASO 2: Chequear que msg_sys es válido
1729     // PASO 3: Mover los datos de la dirección al espacio del kernel
    // PASO 3.1: Mueve los datos de la dirección (cabeceras cmsghdr del msg_sys)
    al espacio del kernel
1744     err = cmsghdr_from_user_compat_to_kern(&msg_sys,
ctl, sizeof(ctl));
    // PASO 3.2: Se copia el msg_sys al espacio del kernel
1761     if (copy_from_user(ctl_buf, (void __user *)
msg_sys.msg_control, ctl_len))
1762         goto out_freectl;
1764     }
    // PASO 4: Enviar mensaje
1769     err = sock_sendmsg(sock, &msg_sys, total_len);
    // Finalización
1781 }

```

Enviar Mensaje (*sock_sendmsg*)

Primera función en las llamadas para envío de mensajes.

```

548 int sock_sendmsg(struct socket *sock, struct msghdr *msg, size_t
size)
549 {
554     init_sync_kiobc(&iocb, NULL);
555     iocb.private = &siocb;
556     ret = __sock_sendmsg(&iocb, sock, msg, size); // Enviar mensaje
    // Controlar error
559     return ret;
560 }

```

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.

```

530 static inline int __sock_sendmsg(struct kiocb *iocb, struct socket
*sock,
531                               struct msghdr *msg, size_t size)
532 {
    // Envío con operación segura
541     err = security_socket_sendmsg(sock, msg, size);
    // Envío del mensaje dependiente del protocolo.
545     return sock->ops->sendmsg(iocb, sock, msg, size);
546 }

```

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.

```

1554 asmlinkage long sys_recvfrom(int fd, void __user * ubuf, size_t
size, unsigned flags,
1555                               struct sockaddr __user *addr, int
__user *addr_len)
1556 {
    // PASO 1: Obtener el socket
1563     sock = sockfd_lookup(fd, &err);
    // PASO 2: Preparar mensaje para su recepción
1573     msg.msg_name=address;
    // PASO 3: Recibir el mensaje
1577     err=sock_recvmsg(sock, &msg, size, flags);
1579     if(err >= 0 && addr != NULL) // Si hay remitente
1580     {
1581         err2=move_addr_to_user(address, msg.msg_namelen,
addr, addr_len);
1584     }
1585     sockfd_put(sock);
1586 out:
1587     return err;
1588 }

```

Recepción de Mensajes (*sys_recv*)

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

```
1594 asmlinkage long sys_recv(int fd, void __user * ubuf, size_t size,
unsigned flags)
1595 {
1596     return sys_recvfrom(fd, ubuf, size, flags, NULL, NULL);
1597 }
```

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*.

```
1787 asmlinkage long sys_recvmsg(int fd, struct msghdr __user *msg,
unsigned int flags)
1788 {
    // PASO 1: Obtener el socket
1811     sock = sockfd_lookup(fd, &err);
1819     // PASO 2: Chequear que msg_sys es válido
1832     // PASO 3: Salvar las direcciones modo usuario
    // PASO 4: Recibir mensaje
1850     err = sock_recvmsg(sock, &msg_sys, total_len, flags);
    // PASO 5: Copiar el nombre del socket (msg_sys.msg_name) en el espacio
de direcciones del usuario
1855     if (uaddr != NULL) {
1856         err = move_addr_to_user(addr, msg_sys.msg_namelen,
uaddr, uaddr_len);
1859     }
    // PASO 6: Poner datos del mensaje en el espacio de usuario (__put_user)
1860     err = __put_user(msg_sys.msg_flags, COMPAT_FLAGS(msg));
    // Finalización
1880 }
```

Recibir Mensaje (*sock_recvmsg*)

Primera función en las llamadas para envío de mensajes.

```
599 int sock_recvmsg(struct socket *sock, struct msghdr *msg,
600                 size_t size, int flags)
601 {
```

```

606     init_sync_kiobc(&iocb, NULL);
607     iocb.private = &siocb;
608     ret = __sock_recvmsg(&iocb, sock, msg, size, flags); // Recibir
mensaje
        // Controlar error
559     return ret;
560 }

```

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.

```

580 static inline int __sock_recvmsg(struct kiocb *iocb, struct socket
*sock,
581                                struct msghdr *msg, size_t size, int
flags)
582 {
        // Recepción con operación segura
592     err = security_socket_recvmsg(sock, msg, size, flags);
        // Recepción del mensaje dependiente del protocolo.
596     return sock->ops->recvmsg(iocb, sock, msg, size, flags);
546 }

```

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.

```

1604 asmlinkage long sys_setsockopt(int fd, int level, int optname, char
__user *optval, int optlen)
1605 {
        // PASO 1: Obtener el socket
1612     if ((sock = sockfd_lookup(fd, &err))!=NULL)
1613     {
1614         err =
security_socket_setsockopt(sock, level, optname);
        // PASO 2: Fijar la opción
1620         if (level == SOL_SOCKET)
1621         err=sock_setsockopt(sock, level, optname, optval, optlen);
1622         else
1623             err=sock->ops->setsockopt(sock, level,
optname, optval, optlen);
1624             sockfd_put(sock);
1625     }
1626     return err;

```

```
1627 }
```

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*.

```
// Pensado para todos los protocolos (genérico, a nivel de socket)
192 int sock_setsockopt(struct socket *sock, int level, int optname,
193                    char __user *optval, int optlen)
194 {
224     switch(optname)
225     {
// Tratar cada opción de forma genérica
para todos los protocolos
437     }
438     release_sock(sk);
439     return ret;
440 }
```

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.

```
1634 asmlinkage long sys_getsockopt(int fd, int level, int optname, char
__user *optval, int __user *optlen)
1635 {
// PASO 1: Obtener el socket
1639     if ((sock = sockfd_lookup(fd, &err)) != NULL)
1640     {
1641         err = security_socket_getsockopt(sock, level,
1642                                         optname);
// PASO 2: Tomar la opción
1648         if (level == SOL_SOCKET)
1649             err = sock_getsockopt(sock, level, optname, optval, optlen);
1650         else
1651             err = sock->ops->getsockopt(sock, level,
optname, optval, optlen);
1652         sockfd_put(sock);
1653     }
1654     return err;
1655 }
```

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***.

```
// Pensado para todos los protocolos (genérico, a nivel de socket)
443 int sock_getsockopt(struct socket *sock, int level, int optname,
444                    char __user *optval, int __user *optlen)
445 {
224     switch(optname)
225     {
// Tratar cada opción de forma genérica
para todos los protocolos
437     }
// Pasar valor de la opción al espacio de usuario
598     if (copy_to_user(optval, &v, len))
599         return -EFAULT;
600 lenout:
601     if (put_user(len, optlen))
602         return -EFAULT;
603     return 0;
604 }
```

Cerrar (***sys_shutdown***)

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

```
1662 asmlinkage long sys_shutdown(int fd, int how)
1663 {
// PASO 1: Obtener el socket
1667     if ((sock = sockfd_lookup(fd, &err)) != NULL)
1668     {
1669         err = security_socket_shutdown(sock, how);
// PASO 2: Cerrar el socket de forma dependiente del protocolo
1675         err = sock->ops->shutdown(sock, how);
1676         sockfd_put(sock);
1677     }
1678     return err;
1679 }
```

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 sobrescribe 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.

```
427 struct socket *sockfd_lookup(int fd, int *err)
428 {
433     if (!(file = fget(fd))) // Tomar el fichero
434     {
435         *err = -EBADF;
436         return NULL;
437     }
439     inode = file->f_dentry->d_inode;
440     if (!inode->i_sock || !(sock = SOCKET_I(inode))) // Chequear que
es un socket
441     {
442         *err = -ENOTSOCK;
443         fput(file);
444         return NULL;
445     }
451     return sock;
452 }
```