



# Sockets

Fernández Perdomo, Enrique  
Horat Flotats, David J.

# Índice

## ■ 1. Introducción

- Historia
- Definición
  - TCP
- API
  - Funciones
  - Estructuras
- Propiedades
- Tipos

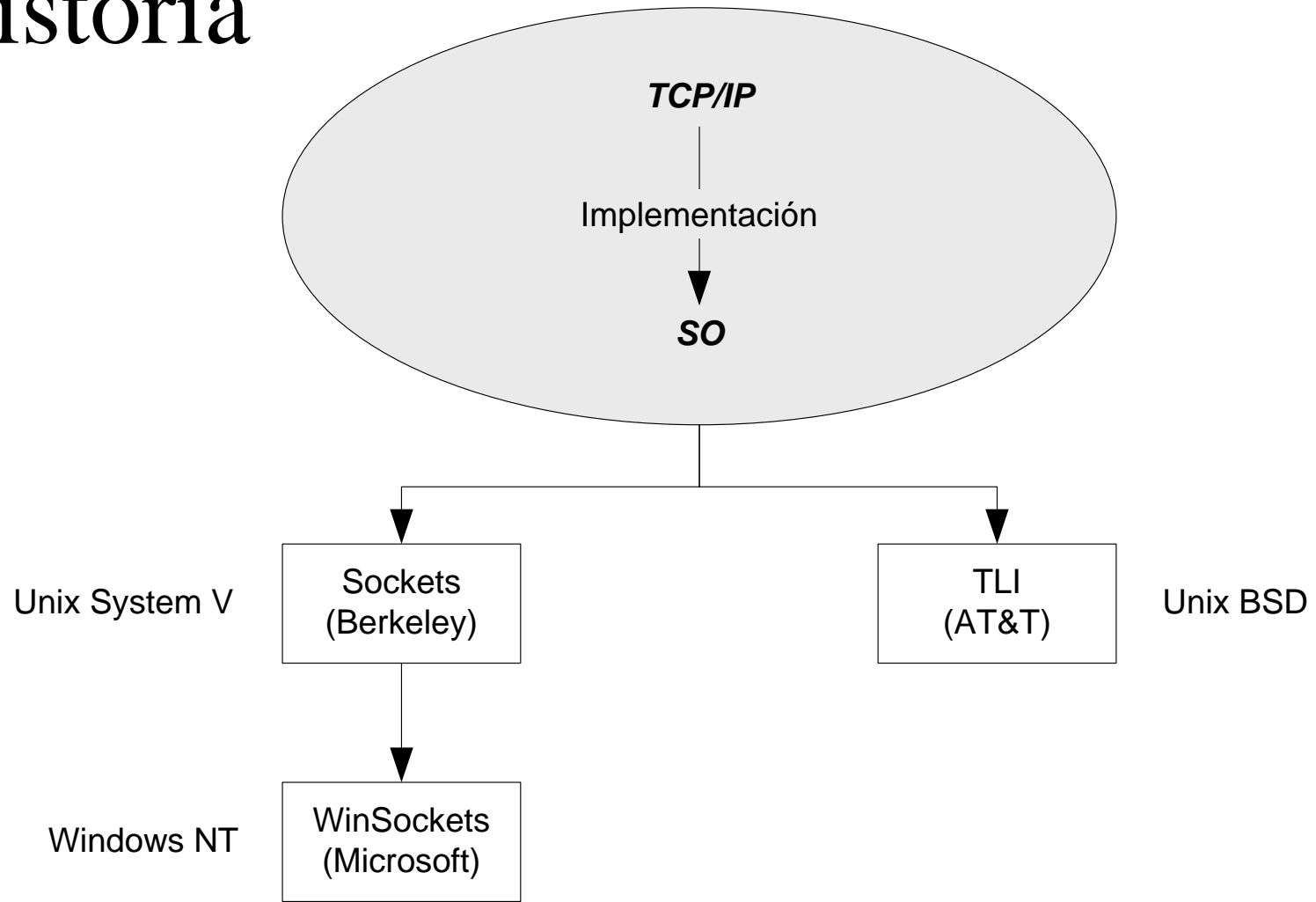
## ■ 2. Código

- Punto de entrada (***sys\_socketcall***)
- Funciones principales
- Funciones auxiliares



# 1. Introducción

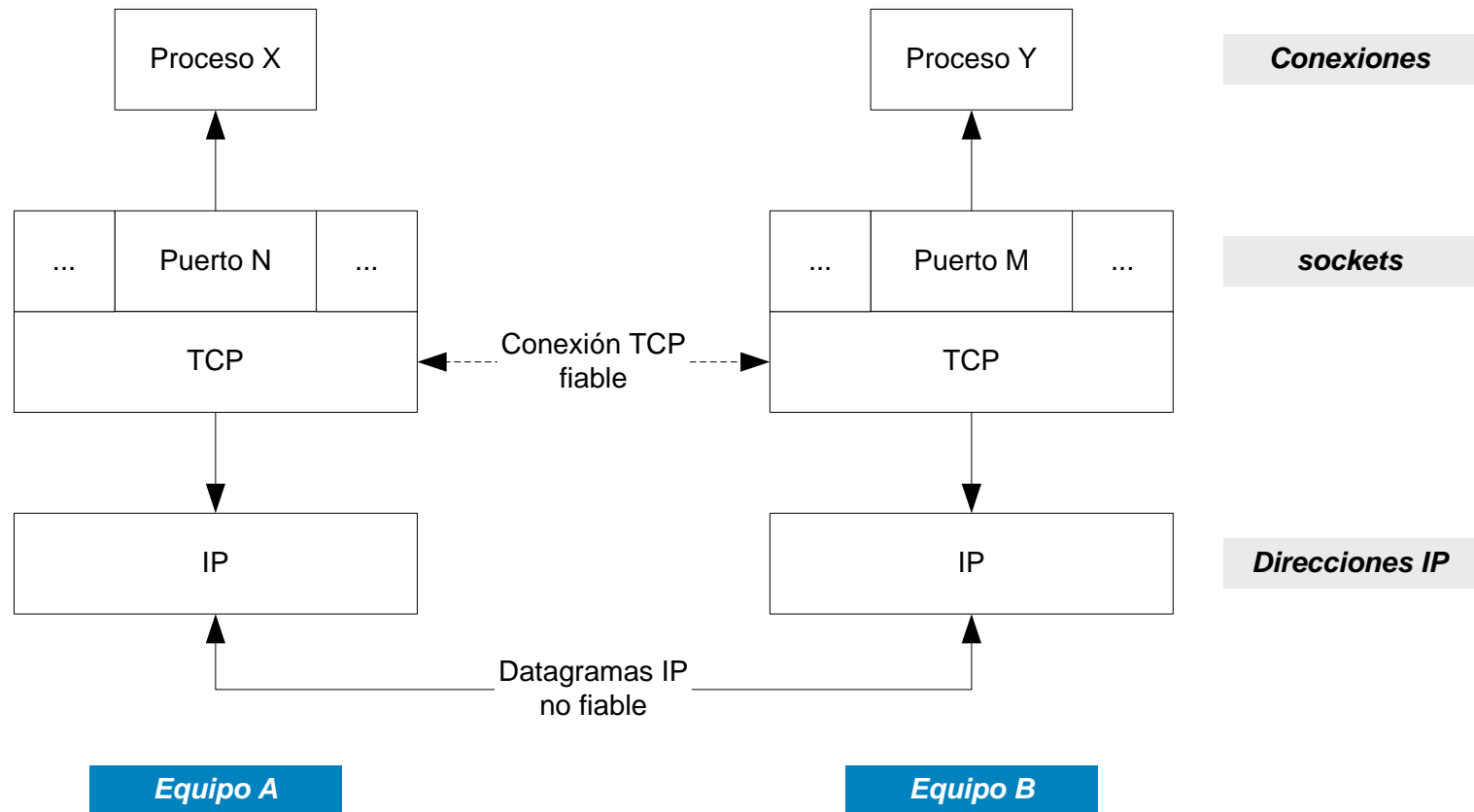
# Historia



# Definición

- Punto de comunicación
- Identificado por un descriptor
  - Igual al de entrada / salida
- TCP-IP
  - Origen (ip, puerto)
  - Destino (ip, puerto)

# Definición



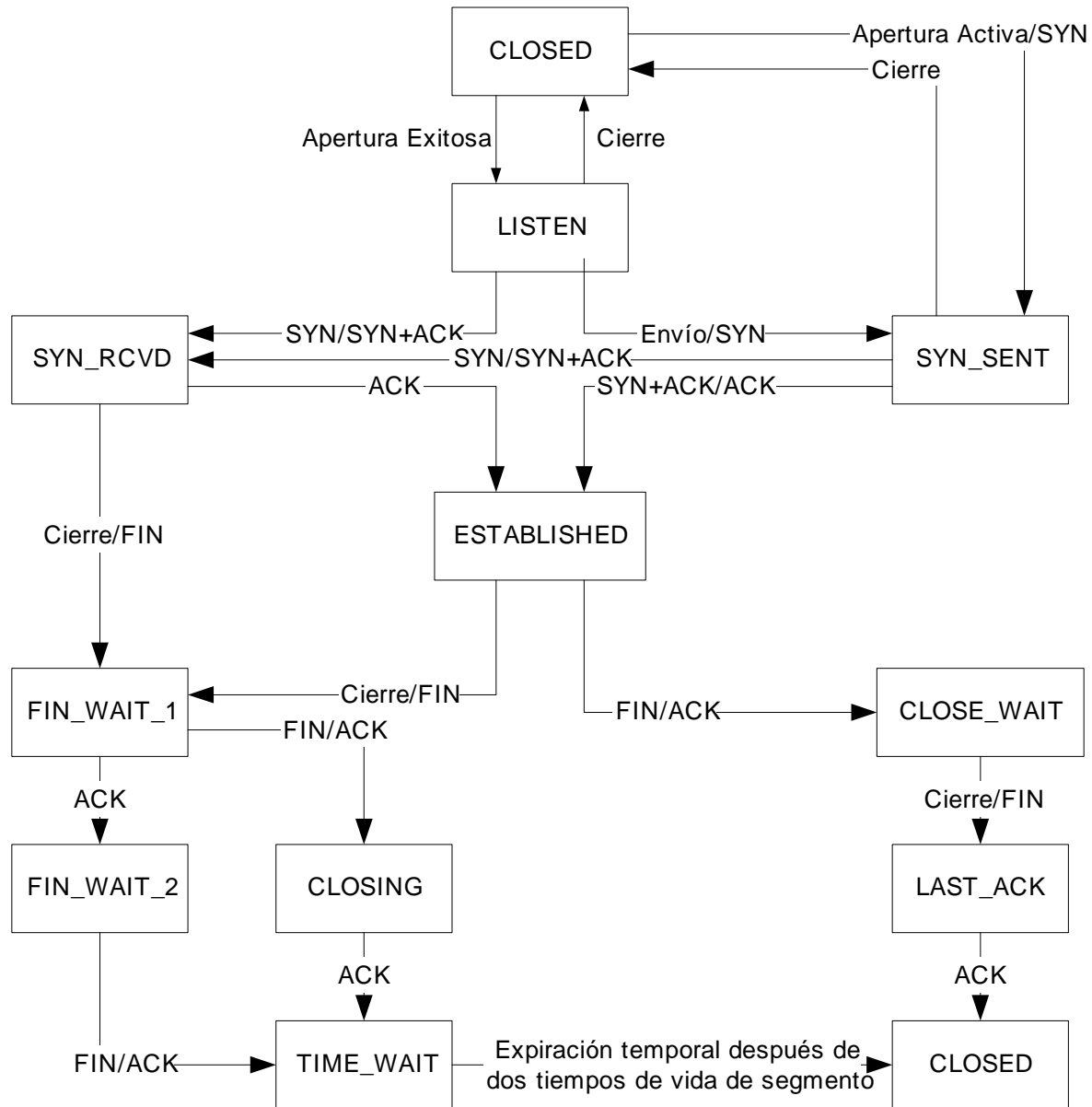
# Definición

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

# TCP

- Orientado a rstras
- Buffer
- Conexión por Circuito Virtual
- Conexión Full-duplex

# TCP



# API (Funciones)

- SOCKET (Creación)
- BIND (Unión Socket-Dirección)
- LISTEN (Escucha)
- CONNECT (Conexión cliente -> Servidor)
- SENDTO (Enviar)
- RECVFROM (Recibir)
- CLOSE (Cerrar)



# API (Estructuras)

Podemos identificar tres formatos de direcciones fundamentales:

- Estructura Genérica
- Dominio UNIX (AF\_UNIX)
- Dominio Internet (AF\_INET)

# Estructura Genérica

- Estructura por defecto
- Será reemplazada por la definida para el dominio concreto de comunicación.

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

# Dominio UNIX (AF\_UNIX)

- Sockets locales al sistema en que se definen.
- Comunicación interna entre procesos.
- Estructura 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)

- Direcciones de sockets con la estructura `sockaddr_in`.
  - Designa servicio sobre máquina particular.

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

# Dominio Internet (AF\_INET)

- El primer campo indica la familia del protocolo.
  - Valdrá AF\_INET para protocolos de Internet.
- El segundo campo (puerto) puede indicarse de dos formas:
  - Número no nulo o superior a IPPORT\_RESERVED
  - Simbólica: para servicios estándar
    - Ej. IPPORT\_TCP o IPPORT\_TELNET

```
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 */
};
```

# Dominio Internet (AF\_INET)

- El tercer campo indica la dirección de Internet. Puede tener dos formas:
  - Dirección de Internet como IP o Nombre de Dominio
  - El valor INADDR\_ANY. Se usa en pasarelas, que disponen de varias direcciones diferentes.
- El cuarto campo sirve para que coincidan en tamaño:
  - Esta estructura, con
  - La estructura de dirección genérica sockaddr
    - Máximo: 14 octetos de dirección

```
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 (I)

- Propiedades de la comunicación en las que se implican los sockets.
- Las propiedades son:
  1. Fiabilidad de la Transmisión.
    - No se pierden los datos transmitidos.
  2. Conservación del Orden de los Datos.
    - Datos llegan en el orden en que se emitieron.
  3. No Duplicación de los Datos.
    - El Dato sólo llega una vez.

# Propiedades (II)

- Las propiedades son:
- 4. Comunicación en modo conectado.
  - Conexión establecida antes de iniciar comunicación  
→ Emisión desde un extremo destinada al otro (implícitamente).
- 5. Conservación de los límites de los mensajes.
  - Límites de mensajes emitidos pueden encontrarse en destino.
- 6. Envío de Mensajes “urgentes”.
  - Datos fuera de flujo o fuera de banda
    - Enviar datos fuera de flujo normal → Accesible de inmediato.

# Tipos

- Existen tres tipos fundamentales:
  1. SOCK\_DGRAM
  2. SOCK\_STREAM
  3. SOCK\_RAW

# SOCK\_DGRAM

- Comunicación en modo no conectado.
- Envío de datagramas de tamaño limitado.
- Conserva 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

- Acceso a protocolos de bajo nivel.
- En Internet, el protocolo subyacente es IP.
- Uso reservado al Superusuario.
- Permite implantar nuevos protocolos.



## 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***.

# Estructura *sock\_common*

- *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***.

```
181 struct sock {  
186     struct sock_common    __sk_common;  
...  
267 };
```

# Estructura **sock**

- **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

```
181 struct sock {  
...  
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];  
...  
267 };
```

# Estructura **sock**

- **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

```
181 struct sock {  
...  
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;  
...  
267 };
```

# Estructura **sock**

- **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.

```
181 struct sock {
...
212     int                sk_sndbuf;
217     unsigned char     sk_no_largesend;
218     int                sk_route_caps;
226     struct {
227         struct sk_buff *head;
228         struct sk_buff *tail;
229     } sk_backlog;
...
267 };
```

# Estructura **sock**

- **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

```
181 struct sock {  
...  
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;  
...  
267 };
```

# Estructura **sock**

- **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 {  
...  
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***

- Llamada al sistema para todas las llamadas al sistema de sockets.
- En el parámetro ***call*** se indica cuál y se pasan sus argumentos en ***args***.

# *sys\_socketcall*

```
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 *)a1, a[2]); break;
1923         case SYS_CONNECT:
1924             err = sys_connect(a0, (struct sockaddr __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 *)a1, (int __user *)a[2]); break;
1932         case SYS_GETSOCKNAME:
1933             err = sys_getsockname(a0,(struct sockaddr __user *)a1, (int __user *)a[2]);
break;
```

# sys\_socketcall

```
1935     case SYS_GETPEERNAME:
1936         err = sys_getpeername(a0, (struct sockaddr __user *)a1, (int __user *)a[2]);
           break;
1938     case SYS_SOCKETPAIR:
1939         err = sys_socketpair(a0,a1, a[2], (int __user *)a[3]); break;
1941     case SYS_SEND:
1942         err = sys_send(a0, (void __user *)a1, a[2], a[3]); break;
1944     case SYS_SENDDTO:
1945         err = sys_sendto(a0,(void __user *)a1, a[2], a[3],
1946             (struct sockaddr __user *)a[4], a[5]); break;
1948     case SYS_RECV:
1949         err = sys_recv(a0, (void __user *)a1, a[2], a[3]); break;
1951     case SYS_RECVFROM:
1952         err = sys_recvfrom(a0, (void __user *)a1, a[2], a[3],
1953             (struct sockaddr __user *)a[4], (int __user *)a[5]); break;
1955     case SYS_SHUTDOWN:
1956         err = sys_shutdown(a0,a1); break;
```

# *sys\_socketcall*

```
1958         case SYS_SETSOCKOPT:
1959             err = sys_setsockopt(a0, a1, a[2], (char __user *)a[3], a[4]); break;
1961         case SYS_GETSOCKOPT:
1962             err = sys_getsockopt(a0, a1, a[2], (char __user *)a[3], (int __user *)a[4]);
            break;
1964         case SYS_SENDMSG:
1965             err = sys_sendmsg(a0, (struct msghdr __user *) a1, a[2]); break;
1967         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*)

- `long sys_socket(int family, int type, int protocol)`
  - Se le pasa la familia, el tipo y el protocolo que se va a usar
  - Devuelve un descriptor de socket que será el que se use con las demás funciones

# Creación (**sys\_socket**)

```
1182 asmlinkage long sys_socket(int family, int type, int protocol) {
1184     int retval; struct socket *sock;
1186     /* Creamos el socket */
1187     retval = sock_create(family, type, protocol, &sock);
1188     if (retval < 0) goto out;
1191     retval = sock_map_fd(sock); /* Lo mapeamos (valor usado) */
1192     if (retval < 0) goto out_release;
1195 out:
1196     /* Puede haber otro descriptor. No es un problema del kernel */
1197     return retval;
1199 out_release:
1200     sock_release(sock);
1201     return retval;
1202 }
```

# Creación (*sys\_socket*)

```
105 struct socket {
106     socket_state      state;
107     unsigned long     flags;
108     struct proto_ops  *ops;
109     struct fasync_struct *fasync_list;
110     struct file       *file;
111     struct sock       *sk;
112     wait_queue_head_t wait;
113     short             type;
114     unsigned char     passcred;
115 };
```

# Creación (**sys\_socket**)

- 94 \* *struct socket* - Socket general BSD
- 95 \* *@state* - estado (%SS\_CONNECTED, etc)
- 96 \* *@flags* - opciones (%SOCK\_ASYNC\_NOSPACE, etc)
- 97 \* *@ops* - operaciones específicas del protocolo
- 98 \* *@fasync\_list* - Lista asíncrona de despertado
- 99 \* *@file* - Puntero de identificador de fichero
- 100 \* *@sk* - representación interna del protocolo
- 101 \* *@wait* - cola de espera para muchos usos
- 102 \* *@type* - tipo de socket (%SOCK\_STREAM, etc)
- 103 \* *@passcred* - credenciales (usados en los sockets UNIX)

# Creación (*sys\_socket*)

```
1172 int sock_create(int family, int type, int protocol, struct socket **res)
1173 {
1174     return __sock_create(family, type, protocol, res, 0);
1175 }
1176
1177 int sock_create_kern(int family, int type, int protocol, struct socket
    **res)
1178 {
1179     return __sock_create(family, type, protocol, res, 1);
1180 }
```

# Creación (**sys\_socket**)

```
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 >= NPROTO) 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     }
```

# Creación (*sys\_socket*)

```
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;
```

# Creación (**sys\_socket**)

```
1140     err = -EAFNOSUPPORT; /* Comprobamos que los propietarios de los módulos */
1141     if (!try_module_get(net_families[family]->owner))
1142         goto out_release;
1144     if ((err = net_families[family]->create(sock, protocol)) < 0) /* Creamos el socket
según la familia */
1145         goto out_module_put;
1150     if (!try_module_get(sock->ops->owner)) {
1151         sock->ops = NULL; goto out_module_put; }
1158     module_put(net_families[family]->owner);
1159     *res = sock;
1160     security_socket_post_create(sock, family, type, protocol, kern);
1162 out:
1163     net_family_read_unlock(); return err;
1165 out_module_put:
1166     module_put(net_families[family]->owner);
1167 out_release:
1168     sock_release(sock); goto out; }
```

# Unión socket-dirección (*sys\_bind*)

- Enlazar un socket con una dirección
- La mayor parte del enlace es responsabilidad del protocolo
- Lo primero que se hace es mover la dirección del socket al núcleo, ya que el enlace se hace allí
- Antes comprobamos que la dirección es correcta y no hay problemas

# Unión socket-dirección (*sys\_bind*)

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

# Escucha (*sys\_listen*)

- Permitimos que el protocolo sea quien realice lo necesario para la escucha
- Marcamos el socket como listo para escucha

# Escucha (*sys\_listen*)

```
1306 int sysctl_somaxconn = SOMAXCONN;
1308 asmlinkage long sys_listen(int fd, int backlog){
1310     struct socket *sock; int err;
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*)

- Creamos un nuevo socket que será el que se enlace con el cliente
- Creamos la conexión con el cliente
- Devolvemos el nuevo identificador de socket

# Acepta una conexión (*sys\_accept*)

```
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;
```

# Acepta una conexión (*sys\_accept*)

```
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     }
```

# Acepta una conexión (***sys\_accept***)

```
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*)

- Conecta un socket con la dirección de un servidor
- Movemos la dirección del espacio de usuario al espacio de kernel una vez comprobada, ya que allí es donde se realiza la conexión
- La conexión depende del protocolo

# Conecta a un servidor (*sys\_connect*)

```
1412 asmlinkage long sys_connect(int fd, struct sockaddr __user *useraddr, 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(useraddr, 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*)

- Obtiene la dirección local de un socket
- Mueve la dirección obtenida al espacio de usuario

# Obtener la dirección local (*sys\_getsockname*)

```
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*)

- Obtiene la dirección remota de un socket
- Mueve la dirección obtenida al espacio de usuario

# Obtener la dirección remota (*sys\_getpeername*)

```
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 }
```

# Crea un par de sockets conectados (***sys\_socketpair***)

- Crea 2 sockets
- Conecta ambos sockets
- Asigna los descriptores a los espacios de usuario
- Devuelve el error (0 en caso de éxito)

# Crea un par de sockets conectados (***sys\_socketpair***)

```
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;
```

# Crea un par de sockets conectados (***sys\_socketpair***)

```
1228      /* Obtenemos los descriptors (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 descriptors 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;
```

# Crea un par de sockets conectados (***sys\_socketpair***)

```
1251     /* En caso de error, cerramos los descriptores y devolvemos el error */
1252     sys_close(fd2);
1253     sys_close(fd1);
1254     return err;
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 a una dirección dada.
- Dicha dirección se pasa al espacio del kernel.
  - Si dicha dirección es nula es porque se ha hecho un *sys\_send*.
- Se chequea que el área de datos del espacio de usuario se puede leer antes de invocar el protocolo.

# Envío de Mensajes (*sys\_sendto*)

```
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     }
```

# Envío de Mensajes (*sys\_sendto*)

// 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 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*

```
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*)

```
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**)

```
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 paquete de un socket y opcionalmente almacena la dirección del remitente.
- Se verifica que los buffers pueden escribirse y si es necesario se mueve la dirección del remitente del espacio del kernel al del usuario.

# Recepción de Mensajes (*sys\_recvfrom*)

```
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 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*

```
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*)

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

```
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.

# Fijar Opciones (*sys\_setsockopt*)

```
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\_setopt*)

```
// Pensado para todos los protocolos (genérico, a nivel de socket)
192 int sock_setopt(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.

# Tomar Opciones (*sys\_getsockopt*)

```
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*)

```
// 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*)

```
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*)

- Pasa de un descriptor de fichero a su slot de socket.
- Parámetros:
  - fd: descriptor de fichero
  - err: puntero a código de error devuelto
- El descriptor de fichero pasado se protege con un cerrojo y el se retorna el socket.
- 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.

# Obtener Socket (*sockfd\_lookup*)

```
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 }
```



# FIN

Fernández Perdomo, Enrique  
Horat Flotats, David J.