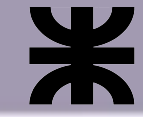


Socket

Un socket es creado usando la llamada al sistema socket()

```
fd = socket(domain, type, protocol);
```

El dominio de comunicación nos dice dónde se encuentran los procesos que se van a intercomunicar.

Dominio	Comunicación realizada	Comunicación entre aplicaciones	Estructura de Direcciones
AF_UNIX	Dentro de Kernel	En el mismo host	sockaddr_un
AF_INET	Vía IPv4	En host conectados por IPv4	sockaddr_in
AF_INET6	Vía IPv6	En host conectados por IPv6	sockaddr_in6

AF_UNIX: comunicación entre aplicaciones en el mismo host.
AF_INET: comunicación entre aplicaciones que se ejecutan en los hosts conectados a través de una red de con IPv4.
AF_INET6: comunicación entre aplicaciones que se ejecutan en los hosts conectados a través de una red de con IPv6.

Hay dos tipos Socket (soportados en UNIX y los dominios de internet):

- Los **Sockets Stream** (SOCK_STREAM) proporcionan un canal de comunicación confiable, bidireccional, byte-stream.

- Los **Socket Datagrama** (SOCK_DGRAM) proporcionan una comunicación de mensajes poco confiable, sin conexión.

Sockets activos y pasivos

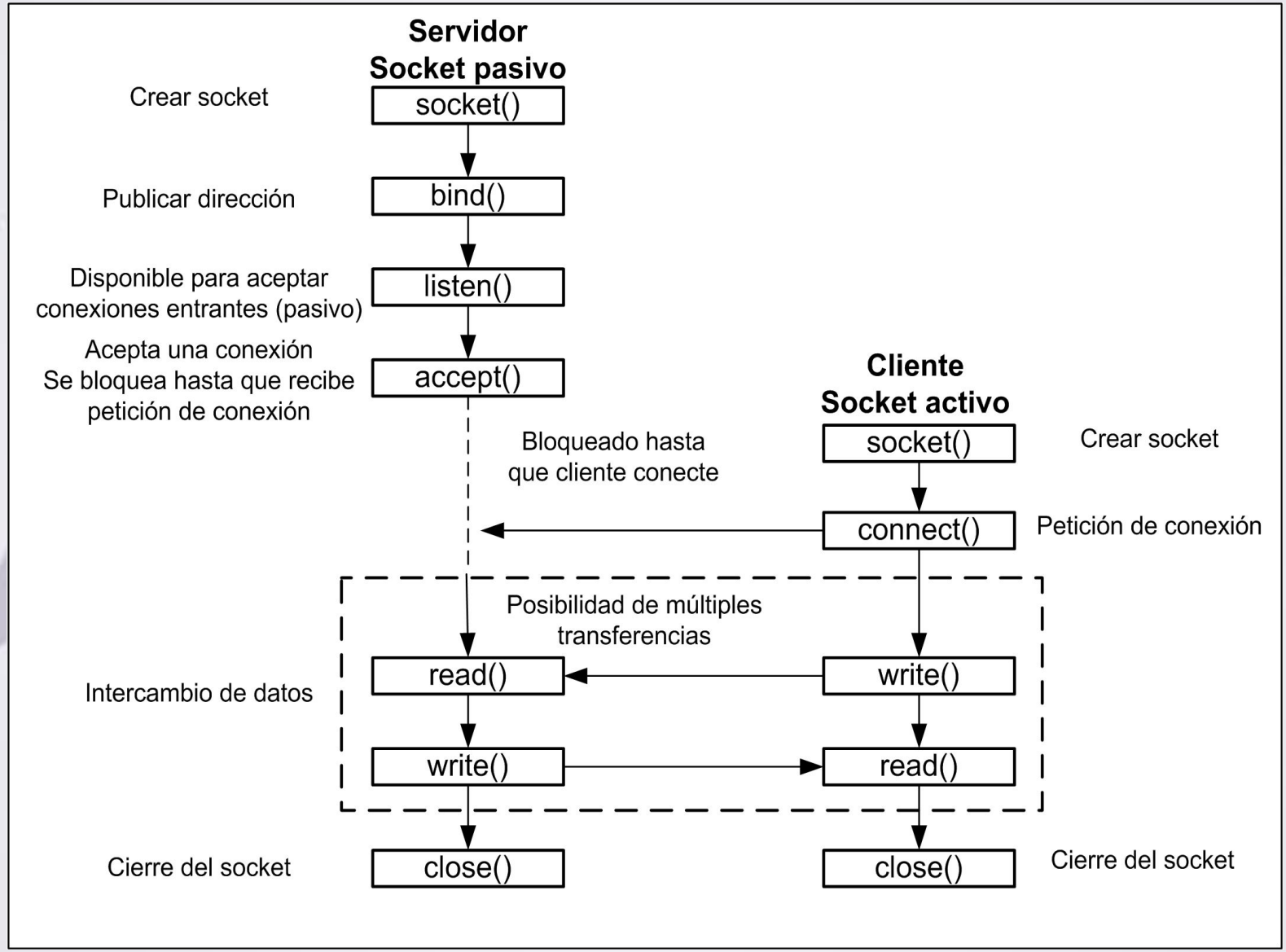
Los sockets stream a menudo se distinguen por ser activos o pasivos:

- Por defecto, un socket creado usando **`socket()`** es activo. Un socket activo puede conectarse a otro socket pasivo. Esto se conoce como la realización de una sesión abierta.

- Un socket pasivo (también llamado un socket de escucha) es uno que ha sido marcado para permitir conexiones entrantes. Aceptar una conexión de entrada se conoce como realizar una apertura pasiva.

Llamadas a sistemas de socket

- **`socket()`** crea un nuevo socket.
- **`bind()`** asocia un socket a una dirección. Por lo general, un servidor utiliza esta llamada para enlazar su socket a una dirección conocida, de modo que los clientes pueden localizar el socket.
- **`listen()`** permite a un socket stream aceptar conexiones entrantes de otros sockets.
- **`accept()`** acepta conexiones entrantes en un socket stream de escucha.
- **`connect()`** establece un sistema de conexión con otro socket.


Servidor
Socket pasivo
Crear socket
socket()
Publicar dirección
bind()
Disponible para aceptar conexiones entrantes (pasivo)
listen()
Acepta una conexión
Se bloquea hasta que recibe petición de conexión
accept()
Cliente
Socket activo
Bloqueado hasta que cliente conecte
socket()
Crear socket
connect()
Petición de conexión
Posibilidad de múltiples transferencias
Intercambio de datos
read()
write()
write()
read()
Cierre del socket
close()
close()
Cierre del socket

Funcionamiento de sockets stream

El funcionamiento de sockets stream se puede explicar por analogía con el sistema telefónico:

1. Creación del socket:

La llamada al sistema **`socket()`** crea un socket, una conexión, lo cual es el equivalente de la instalación de un teléfono.

Para que dos aplicaciones se comuniquen, cada una de ellas debe crear un socket.

Funcionamiento de sockets stream

2. Conexión:
La comunicación a través de un socket stream es análogo a una llamada telefónica. Una aplicación debe conectar su socket al socket de otra aplicación antes de que la comunicación pueda tener lugar.

1. Una aplicación llama a **`bind()`** a fin de vincular al socket a una dirección conocida, y luego llama a **`listen()`** para notificar al sistema operativo de su voluntad de aceptar conexiones entrantes. Este paso es análogo a tener un número de teléfono conocido.
2. La otra aplicación establece la conexión llamando a **`connect()`**, especificando la dirección del socket al que queremos conectar. Esto es análogo a marcar el número de teléfono de alguien.
3. La aplicación que llama a **`listen()`** entonces acepta la conexión mediante **`accept()`**. Esto es análogo a levantar el teléfono cuando suene. Si el **`accept()`** se realiza antes que la aplicación del cliente llame a **`connect()`**, entonces el **`accept()`** bloquea al proceso. Esto es análogo a esperar la llamada.

Funcionamiento de sockets stream

3. Comunicación:

Una vez que una conexión ha sido establecida, se pueden transmitir datos en ambas direcciones entre las aplicaciones. Esto es análogo a una conversación telefónica de dos vías. La conexión continúa hasta que uno de ellos cierra la conexión utilizando **`close()`**.

Creación de un Socket

Un socket se crea con la llamada a sistema **`socket()`**, la misma devuelve un descriptor de archivo que se utiliza para referirse al socket en llamadas posteriores del sistema.

```
#include <sys/socket.h>
int sockfd = socket(int domain, int type, int protocol);
```

Devuelve un descriptor de fichero en caso de éxito, ó -1 en caso de error.

El argumento de **`domain`** especifica el dominio de la comunicación.

El argumento **`type`** especifica el tipo de socket. Se especificar como SOCK_STREAM, para crear un socket stream, o SOCK_DGRAM, para crear un socket datagrama.

El argumento **`protocol`** se especifica siempre como 0 (protocolo por defecto para la familia elegida).

Vinculación de un socket a una dirección: *bind()*

La llamada al sistema **bind()** permite asociar una dirección a un socket existente. Generalmente es una dirección fija bien conocida, así los cliente pueden comunicarse con ese servidor.

```
#include <sys/socket.h>
int bind(int sockfd,const struct sockaddr *addr, socklen_t addrlen);
```

Devuelve 0 si tuvo éxito, o –1 en caso de error.

El argumento **sockaddr** es un puntero a una estructura que especifica la dirección a la que este socket se va a enlazar. El argumento **addrlen** especifica el tamaño de la estructura.

La familia de protocolos TCP/IP, usa una estructura sockaddr:

```
struct_sockaddr_in {
    char sin_len          //longitud total de la dirección
    char sin_family   //Familia de la dirección (AF_INET)
    short sin_port    //número de puerto de protocolo
   struct in_addr sin_addr   //dirección IP del host
}
```

Escuchar las conexiones entrantes: *listen()*

La llamada al sistema **`listen()`** marca al socket stream (cuyo descriptor es **`sockfd`**) como **pasivo**. El socket posteriormente se utilizará para aceptar conexiones de otros sockets.

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

Devuelve 0 en caso de éxito, o –1 si hay error.

El argumento **`backlog`**, nos permite limitar el número de conexiones pendientes del servidor. El kernel debe registrar alguna información acerca de cada solicitud de conexión pendiente, de modo que una posterior llamada **`accept()`** se pueda procesar.

Aceptar una conexión: *accept()*

La llamada al sistema **`accept()`** acepta conexiones entrantes en un socket stream de escucha, socket cuyo descriptor es sockfd.

La llamada bloquea al proceso hasta que se recibe una solicitud de conexión. Si hay clientes en espera el SO entregará el siguiente cliente de la cola de espera.

```
#include <sys/socket.h>
int accept(int sockfd,struct sockaddr *addr,socklen_t *addrlen);
```

Devuelve descriptor en caso de éxito, o -1 en caso de error.

Conectar *connect()*

La llamada de sistema **`connect()`** conecta un socket activo (cuyo descriptor es sockfd) al socket de escucha (pasivo) cuya dirección se especifica por addr y addrlen.

```
#include <sys/socket.h>
connect(int sockfd,const struct sockaddr *serv_addr,socklen_t addrlen);
```

Devuelve 0 en caso de éxito, o –1 si hay error.

Se debe especificar la dirección IP y el número de puerto (servicio) del servidor al que se desea conectar. Los argumentos **`serv_addr`** y **`addrlen`** se especifican en la misma forma que los argumentos correspondientes a **`bind()`**.

Si **`connect()`** falla y queremos volver a intentar la conexión, el método de hacerlo es cerrar el socket, crear un nuevo socket, y volver a intentar la conexión.

Terminación: *close()*

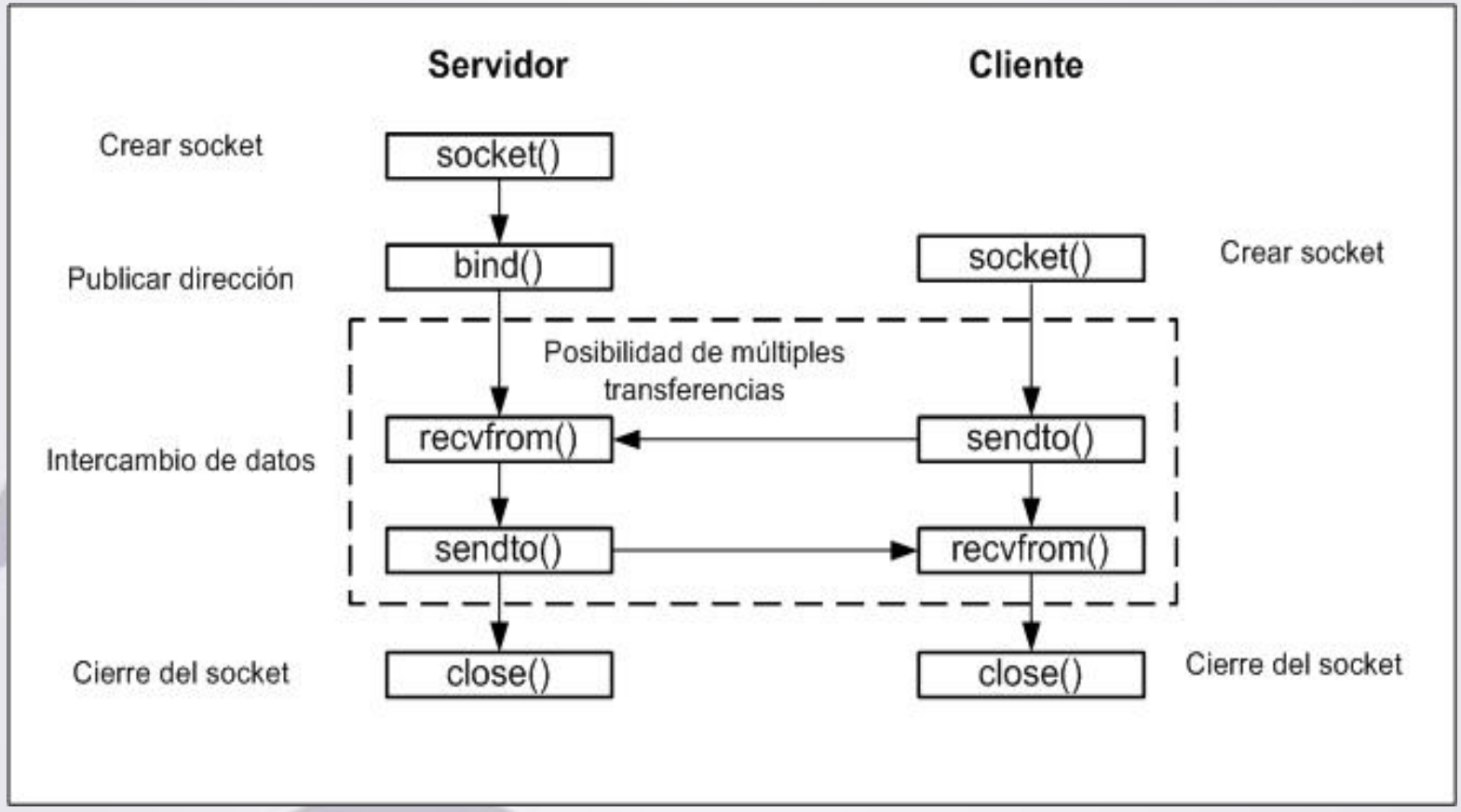
La manera usual de terminar una conexión socket stream es llamar a **`close()`**.

```
#include <sys/socket.h>
close(int sockfd);
```

Devuelve -1 en caso de error.

El argumento de la llamada **`close()`** es el descriptor del socket.

Sockets Datagram



Sockets Datagram

El funcionamiento de los socket datagramas es análogo con el sistema postal:

1. La llamada **`socket()`** crea un socket, es el equivalente a la creación de un buzón de correo. Cada aplicación que desea enviar o recibir datagramas crea un socket.
2. Con el fin de permitir que otra aplicación pueda enviar datagramas, una aplicación utiliza **`bind()`** para enlazar su socket a una dirección conocida. Típicamente, un servidor se une a su socket a una dirección conocida, y un cliente inicia la comunicación mediante el envío de un datagrama a esa dirección.
3. Para enviar un datagrama, una aplicación llama a **`sendto()`**, que tiene como uno de sus argumentos la dirección del socket a enviar. Esto es análogo a poner la dirección del destinatario en una carta y enviarla.
4. Con el fin de recibir un datagrama, una aplicación llama **`recvfrom()`**. Esta llamada nos permite obtener la dirección del remitente, por lo que podemos enviar una respuesta si lo deseamos. Como la analogía, una carta tiene la dirección y nombre del remitente.
5. Cuando el socket ya no es necesario, la aplicación lo cierra mediante **`close()`**.

Intercambio de datagramas: recvfrom() y sendto()

La llamada al sistema **recvfrom()** recibe datagramas en un socket datagrama.

```
#include <sys/socket.h>
ssize_t recvfrom(int sockfd, void *buffer, size_t length, int
flags,struct sockaddr *src_addr, socklen_t *addrlen);
```

Devuelve el número de bytes recibidos, 0 en EOF, o -1 en caso de error.

El valor de retorno y los primeros tres argumentos para esta llamada del sistema son los mismos que para **read()**.

El cuarto argumento, **flags**, es una máscara de bits de control.

Los argumentos **src_addr** y **addrlen** devuelven la dirección del socket remoto. El argumento **src_addr** es un puntero a una estructura de dirección correspondiente al dominio de la comunicación.

Intercambio de datagramas: recvfrom() y sendto()

La llamada al sistema ***sendto()*** envía datagramas en un socket datagrama

```
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *buffer, size_t length,
int flags,const struct sockaddr *dest_addr, socklen_t addrlen)
```

Devuelve el número de bytes enviados, o -1 en caso de error.

El valor de retorno y los primeros tres argumentos para esta llamada del sistema son los mismos que para ***write()***.

El cuarto argumento, flags, es una máscara de bits de control.

Los argumentos **dest_addr** y **addrlen** especifican la dirección el socket al cual se envía el datagrama. Estos argumentos se emplean en la misma manera que en la llamada ***connect()***.

Localhost

Localhost: es el nombre reservado en los host para hacerse referencia a sí mismo.

Cuando una aplicación necesita acceder a varios recursos en una red, se necesita saber la dirección del host remoto, sin embargo, si el archivo o recurso al que necesita tener acceso está en el mismo host donde está corriendo la aplicación, no necesita averiguar su nombre, ya que este se denomina como localhost y su dirección siempre es la misma.

El nombre localhost es traducido como la dirección IP de loopback 127.0.0.1 en IPv4. Cualquier dirección en la red 127.0.0.0/8 puede ser designada como la dirección loopback pero 127.0.0.1 es la opción habitual.

Si hacemos hacemos un comando ping desde una consola veremos:

```
ping localhost

Haciendo ping a ana [127.0.0.1] con 32 bytes de datos:

Respuesta desde 127.0.0.1: bytes=32 tiempo<1m TTL=128
Respuesta desde 127.0.0.1: bytes=32 tiempo<1m TTL=128
Respuesta desde 127.0.0.1: bytes=32 tiempo<1m TTL=128
Respuesta desde 127.0.0.1: bytes=32 tiempo<1m TTL=128

Estad¡sticas de ping para 127.0.0.1:
    Paquetes: enviados = 4, recibidos = 4, perdidos = 0
    (0% perdidos),
Tiempos aproximados de ida y vuelta en milisegundos:
    M¡nimo = 0ms, Maximo = 0ms, Media = 0ms
```

Los sockets son un método de comunicación entre procesos, que permiten el intercambio de datos entre aplicaciones, ya sea en el mismo host o en host diferentes conectadas por una red.

El manejo de sockets se realiza a nivel de capa de aplicación.

En un típico escenario de cliente-servidor, las aplicaciones se comunican usando sockets de la siguiente manera:

- Cada aplicación crea un socket. Un socket es una entidad que permite la comunicación y ambas aplicaciones requieren uno.
- El servidor se une a su socket con una dirección conocida (nombre) de modo que los clientes puedan localizarlo.

Bibliografía

Kerrisk, Michael. The linux programming Interface. 2011. Capítulos 56, 57, 58, 59.