Sistemas Operativos I

IPC: Sockets

La clase pasada vimos dos mecanismos de comunicación entre procesos:

+ Señales, estas eran más que nada usadas por el sistema operativo y indicaciones de terminación de procesos
+ Pipes, que nos presentaban un canal de comunicación unidireccional.

Y además mencionamos bastante el concepto de File Descriptor como la forma de representar dispositivos de Lectura y Escritura del sistema operativo (descendientes de Unix).

En la clase de hoy vamos a comenzar a ver Sockets.

Un socket es un canal de comunicación entre procesos generalizado. Al igual que un pipe, un socket se representa como un File Descriptor. A diferencia de los pipes, los sockets permiten la comunicación entre procesos no relacionados (es decir, por ejemplo, no hace falta que sean Parent y Child) e incluso entre procesos que se ejecutan en diferentes computadoras en una misma red. Los sockets son el principal medio de comunicación con otras máquinas; telnet, rlogin, ftp, talk y otros programas de red conocidos utilizan sockets.

El sistema operativo le asigna un file descriptor a objetos que tienen asociado un dispositivo (o archivo) a los que se le pueden realizar operaciones de lectura y escritura. En el caso de los Sockets se busca independizar el objeto en sí del dispositivo que se usa como medio para la comunicación.
Por ejemplo, podemos crear un socket, y decidir el dispositivo al momento de enviar

mensajes, incluso usar el mismo socket para enviar mensajes a diferentes destinos por medio de distintos dispositivos.

También voy a mencionar el concepto de *puerto* y/o *dirección*. El puerto es un puerto dedicado en el sistema operativo transversal a todos los usuario/procesos que estén ejecutándose en el SO. Algunos de los conocidos son 8080, 21/22, etc… Los destinados de uso común son los del 49152 al 65535. Las direcciones son un concepto un poco más complejo y ya las veremos cuando llegue el momento.

Socket

Un socket es un **canal generalizado para la comunicación entre procesos**.

- Al igual que un pipe, un socket se representa como un File Descriptor.
- A diferencia de los pipes, los sockets **permiten la comunicación entre procesos no relacionados** (es decir, por ejemplo, no hace falta que sean Parent y Child) e incluso entre procesos que se ejecutan en diferentes computadoras en una misma red.
- Los sockets son el principal medio de comunicación con otras computadoras; telnet, rlogin, ftp, talk y otros programas de red conocidos utilizan sockets.

Creación de Sockets

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

int socket(int domain, int type, int protocol);
```

- **Dominio:** dominio que se utilizara (Local vs Red)
- **Tipo:** tipo de la conexión (Datagramas vs Stream)
- **Protocolo:** protocolo a utilizar (Normalmente, solo existe un único protocolo para admitir un tipo de socket particular dentro de una familia de protocolo determinada, en cuyo caso el protocolo se puede especificar como 0)

Al momento de crear un socket deberemos indicar cierta información pertinente a:

+ qué medio de comunicación utilizaremos, es decir, el medio físico
+ qué tipo de conexión será, orientada a la conexión o simplemente al envío de mensajes
+ dependiendo del Dominio y tipo de la conexión podríamos especificar el protocolo que utilizaremos

El protocolo especifica un protocolo particular que se utilizará con el socket. Normalmente, solo existe un único protocolo para admitir un tipo de socket particular dentro de una familia de protocolo determinada, en cuyo caso el protocolo se puede especificar como 0. Sin embargo, es posible que existan muchos protocolos, en cuyo caso se debe especificar un protocolo particular en este campo. El número de protocolo a utilizar es específico del "dominio de comunicación" en el que se realizará la comunicación; ver protocolos(5). Consulte getprotoent(3) sobre cómo asignar cadenas de nombre de protocolo a números de protocolo.

La lista de todos los protocolos existentes se puede encontrar en:
https://www.iana.org/assignments/protocol-numbers

Local para comunicar procesos dentro del mismo sistema podremos utilizar AF_UNIX o AF_LOCAL (sinónimo de AF_UNIX)

Por la red nos comunicaremos utilizando una conexión de tipo IPv4 o IPv6, las constantes que los identifican con AF_INET y AF_INET6
Para ver el resto de los posibles dominios ver socket(2)
‘man 2 socket’

Socket: Tipos de comunicación

- **UDP (User Datagram Protocol)**
  - **SOCK_DGRAM**: UDP es no confiable y orientado a sin conexión, es decir, no garantiza la entrega de los datagramas, sin embargo, esta propiedad de UDP es precisamente, la que hace tan interesantes los protocolos SNMP (Simple Network Management Protocol) por cargar poco la red y por su absoluta independencia del hardware lo que facilita el intercambio de información.
- **TCP (Transmission Control Protocol)**
  - **SOCK_STREAM**: TCP es confiable y orientado a conexión, esto es que garantiza que todos los paquetes lleguen correctamente y en orden.

UDP manda de a un mensaje a la vez.
TCP manda un stream de bytes

Socket Orientado a Datagramas

- El protocolo **UDP**, **SOCK_DGRAM**, se usa para enviar **paquetes individuales a una dirección dada de manera no confiable**. Cada vez que se escriben datos en un socket tipo SOCK_DGRAM, esos datos se convierten en un paquete. Dado que los sockets SOCK_DGRAM no trabajan con una conexión, se debe especificar la dirección del destinatario con cada paquete.
- La única garantía que ofrece el sistema sobre las solicitudes de transmisión de datos es que **hará todo lo posible para entregar cada paquete que se envíe**. Puede tener éxito con el 6º paquete después de fallar con el 4º y el 5º paquete; el 7º paquete puede llegar antes que el 6º y puede llegar una segunda vez después del 6º.

La conexión se piensa para el envío de un mensaje o datagrama, es decir, que no hay un canal de comunicación directo y estable durante toda la comunicación sino que el mensaje propiamente dicho es el que recorre un camino.
Enviar un mensaje en el mismo socket puede recorrer incluso otro camino.

La analogía con la vida real es el del sistema de cartas. Hay toda una conexión física real entre las diferentes oficinas de correo, que son todos los caminos posibles que pueden realizar las camionetas o repartidores de correo. Dicho de otra manera todas las oficinas de correo estás conectadas mediante calles. Ahora cuando uno envía una carta, en la carta pone los datos del destinatario y del remitente, es decir, que en la carta está toda la información para llegar al destino (o en caso de que sea necesario que se devuelva al remitente).
La carta comienza su viaje yendo de una oficina de correo a otra, recorriendo un camino que depende de las decisiones que tomen los diferentes repartidores de correo.
Si vuelvo a enviar una carta con los mismos datos, tanto de destinatario y remitente, no significa que se vuelva a tomar el mismo camino o se utilicen los mismos recursos, etc.

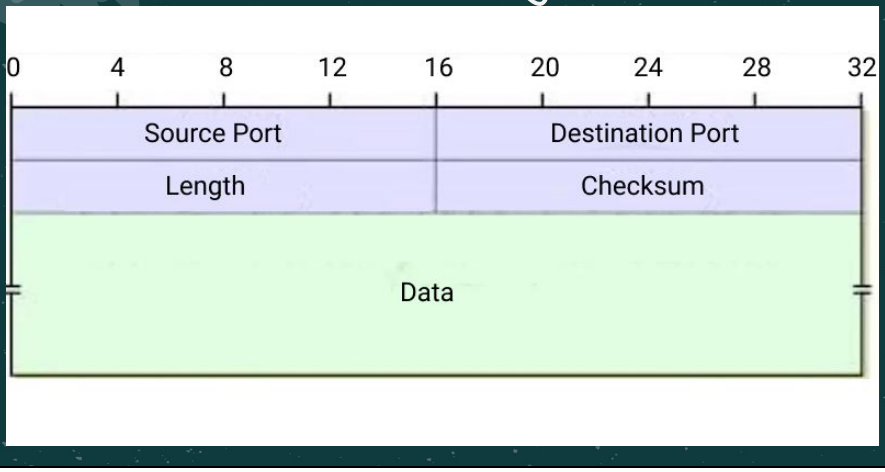
Volviendo un paso atrás, el **dominio** es lo que garantiza que existe un camino, es la calle, mientras que el socket orientado a datagramas nos indica que la información que necesitamos es la de destinatario y remitente.

Tiene algunas desventajas como ser que los mecanismos de error están por parte del

usuario del servicio, la carta se intenta enviar y si durante el recorrido se pierde no se hará nada al respecto (best effort basis), incluso por cuestiones de la red el datagrama o mensaje se puede repetir, una forma de obtener más garantías es la replicación de mensajes, etc.
Y no hay acuso de recibo por parte del destinatario a nivel de conexión, es decir, que si quieren tenerlo deberá ser implementado por parte del usuario del servicio.

Este tipo de conexiones son muy útiles por ejemplo para hacer streaming de video a múltiples usuarios, donde perder un par de fotogramas no importa tanto.

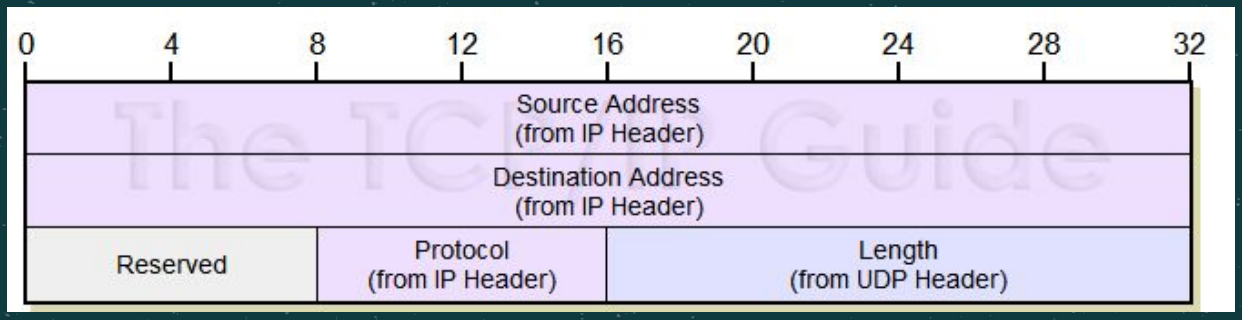
Socket Orientado a Datagramas



En comunicaciones esto es manejado por un protocolo denominado UDP (User Datagram Protocol).
Los ‘datagramas’ que se envían tienen la siguiente los siguientes datos:

+ [el puerto origen]
+ el puerto de destino
+ la longitud del paquete
+ [campo para verificar la integridad de los datos] (Por si algunos bits se pierden o se cambian durante el viaje del telegrama)

Socket Orientado a Datagramas

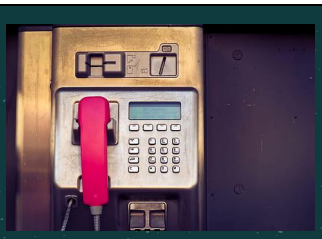


A esto le vamos a sumar un poco más de información para enviarlos por la red:

+ Direcciones de destino y remitente
+ Fragmento reservado, cuestiones de protocolo ip y la longitud del mensaje.

Esto es un pseudo-encabezado como para que vean como sería.

Socket de tipo Stream



Los sockets de tipo stream orientan la conexión a establecer un canal de comunicación entre dos procesos o hilos. Es decir, se busca establecer una conexión estable y luego comenzar con el envío de mensajes. Esto garantiza la conexión, el envío y recepción de los mensajes, como a su vez el orden en el que fueron enviados. En particular garantizan las siguientes propiedades:

- **Conexión Orientada a Stream:** Cuando se quieren transmitir grandes volúmenes de datos, estos datos se dividen en bytes (8-bits), y se crea una flujo (stream) de bytes. El flujo se recibe en el orden en que fue enviada. Al ser un flujo de bytes, no se conservan límites en los datos.
- **Conexión Virtual de Circuitos:** primero se establece una conexión estable. Luego, si la comunicación falla por alguna razón (por ejemplo, algún nodo de la red se cae), ambas computadoras detectan el fallo y lo comunican al proceso correspondiente.
- **Full Duplex:** la conexión es bidireccional.

Los sockets de tipo *stream* orientan la conexión a establecer un canal de comunicación entre dos procesos o hilos. Es decir, se busca establecer una conexión estable y luego comenzar con el envío de mensajes. Esto garantiza la conexión, el envío y recepción de los mensajes, como a su vez el orden en el que fueron enviados. En particular garantizan las siguientes propiedades:

+ **Conexión orientada a Stream:** Cuando se quieren transmitir grandes volúmenes de datos, estos datos se dividen en 8-bit octetos, llamados bytes, y se crea una cadena de bytes. La cadena se recibe en el orden en que fue enviada.
+ **Conexión Virtual de Circuitos:** Al momento de comenzar la transferencia, primero se establece una conexión estable en el dominio, y una vez que se garantiza la conexión se comienza la transmisión. Durante la transferencia, el protocolo de conexión garantiza que los mensajes son recibidos correctamente. Si la comunicación falla por alguna razón (por ejemplo, algún nodo de la red se cae), ambas computadoras detectan el fallo y lo comunican al proceso correspondiente.
+ **Conexión Full Duplex:** la conexión es bidireccional.

Un ejemplo sencillo es la comunicación telefónica. Una persona marca el número con quien quiere comunicarse, la central telefónica establece una conexión con la unidad telefónica destino, la persona acepta la conexión, se establece la conexión, y se puede hablar bi-dreccionalmente por el mismo medio.

Socket de tipo Stream

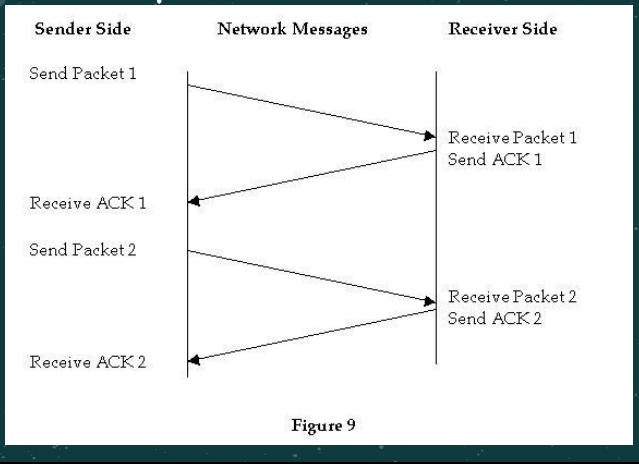


Figure 9

Esto se logra en base a un principio fundamental: *acuse de recibo con retransmisión*. Es decir, todo mensaje enviado por un socket de tipo stream va requerir que el destinatario confirme la recepción del mensaje, enviando de vuelta un mensaje de confirmación llamado **ACK**. El emisor mantendrá cuenta de los paquetes que envía y de los ACKs que recibe, y así saber si se perdieron paquetes o no.

Socket de tipo Stream

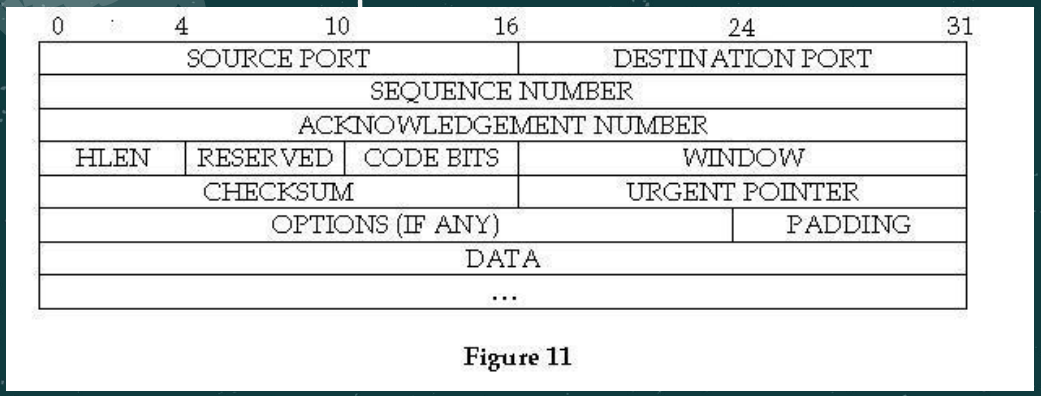
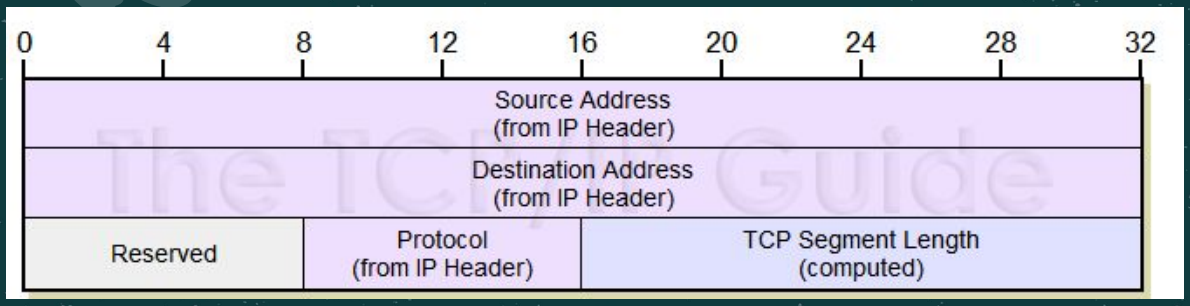


Figure 11

Los paquetes enviados por el protocolo de transmisión basado en stream es un poco más complejo, además de la información en UDP tenemos principalmente:

- El *SEQUENCE NUMBER* (número de secuencia) que indica la posición del segmento que se envía en el mensaje.
- El *ACKNOWLEDGMENT NUMBER* (número de ACK) indica el los fragmentos que fueron recibidos.
- *HLEN* (Longitud de la cabecera) longitud de la cabecera medida en múltiplos de 32-bit.

Socket de tipo Stream



Y al igual que UDP podemos agregarle la información requerida por el protocolo IP.

Socketpair

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

int socketpair(int domain, int type, int protocol, int sv[2]);
```

La llamada socketpair() crea un par de sockets sin nombre conectados en el dominio, tipo y usando el protocolo especificado.

- Los Files Descriptors que hacen referencia a los nuevos sockets se devuelven en sv[0] y sv[1]. Los dos endpoints son indistinguibles.
- Un socketpair es muy parecido a un pipe; la principal diferencia es que el socketpair es bidireccional, mientras que el pipe tiene un extremo de solo entrada y un extremo de solo salida.
- En cada proceso, como en el pipe debemos cerrar los extremos que no usemos.

Vamos a ver un ejemplo práctico y para esto usaremos una primitiva más sencilla que simula el comportamiento de lo que era un Pipe en este caso bidireccional.

socketpair() crea un par de sockets anónimos, generalmente sockets locales/unix, que solo son útiles para la comunicación entre un proceso Parent y un proceso Child o en otros casos donde los procesos que necesitan usarlos pueden heredar los descriptores de archivo de un ancestro común.

Si va a comunicarse entre procesos no relacionados (en el sentido de paternidad), debe usar la función socket(), bind() y connect() para crear un socket de escucha en un proceso y crear un socket de cliente para conectarse a él en el otro proceso.

Ejemplo con socketpair

Cree dos procesos utilizando fork() y que se comuniquen a través de un socketpair. Ambos deben enviar y recibir mensajes (string) y mostrarlos por pantalla.

Hasta ahora vimos que para crear un socket necesitamos definir primero

+ el dominio: que tipo de conexión física vamos a utilizar (Local o Red)
+ El tipo de conexión orientada a datagramas o a stream de datos
+ Un protocolo, que depende de los dos anteriores y que en general no vamos a especificarlo

Y esto nos generaba un socket aunque éste todavía no este **conectado**.
Para poder transmitir mensajes por el socket deberemos tener un socket **conectado**

Asignar nombres a sockets

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

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Para asignarle nombres a los sockets utilizamos entonces la función de `bind`, donde se les asigna un nombre.
Esto es para indicar a qué socket queremos enviar mensajes y para esto le tenemos que asignar un nombre.
Es lo que hace por nosotros la función `socketpair` donde en vez de asignarles un nombres le asigna File Descriptors, y crea una especie de sockets anónimos.

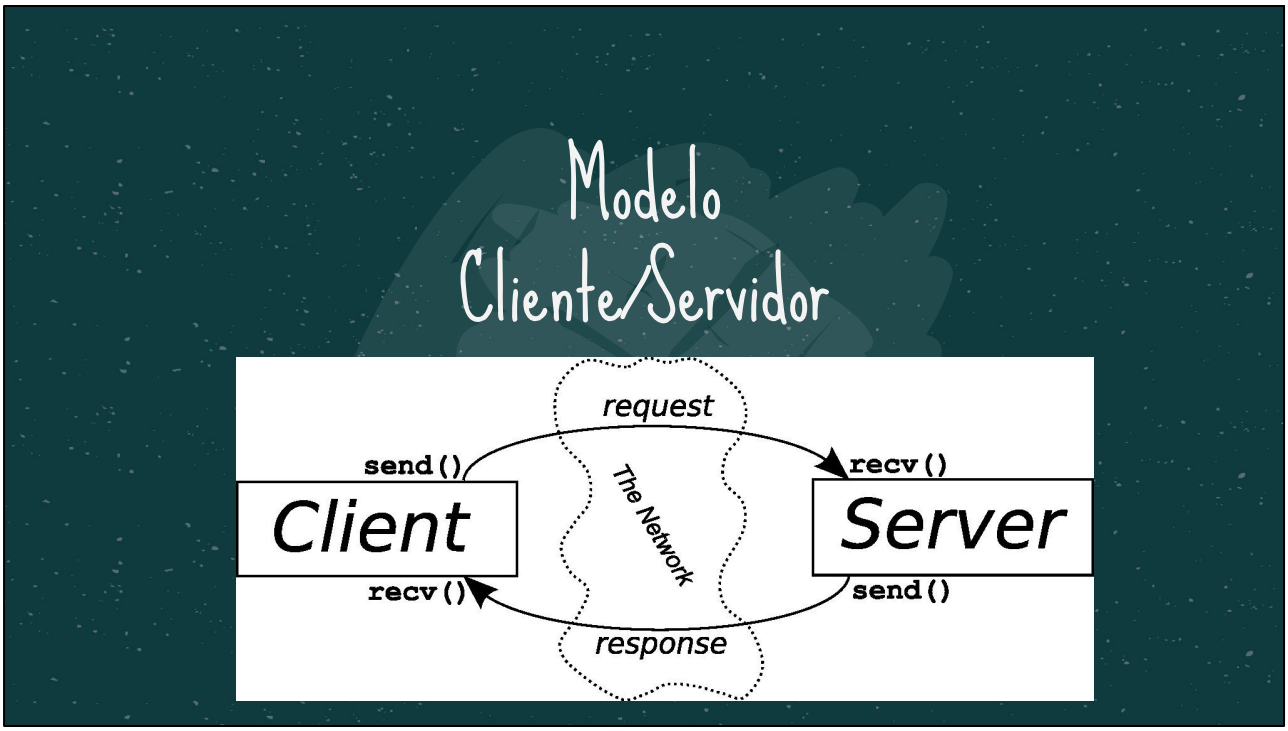
Conectar Sockets: Direcciones Locales vs de Red

```
struct sockaddr_un {
  sa_family_t sun_family;              /* AF_UNIX */
  char        sun_path[108];           /* Pathname */
};
```

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};

/* Internet address. */
 struct in_addr {
    uint32_t       s_addr;     /* address in network byte order */
};
```

Sin ninguna sorpresa, las direcciones locales son en realidad el nombre del archivo que se va a utilizar como dispositivo de conexión.



La idea consiste en repensar la forma en que programamos.
Vamos a pensar en programar el software asumiendo que hay un proceso ofreciendo un servicio, que vamos a llamar servidor y un proceso que accede a dicho servicio que llamaremos cliente.
Tiene varias ventajas como ser que varios clientes pueden pedir por diferentes servicios del servidor, y la lógica del programa está dividida en dos, aunque en general el servidor es quien mantiene la mayor lógica del programa mientras que los clientes presentan una interfaz a dicho servicio.

SEND y RECV

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

ssize_t send(int sockfd, const void *buf, size_t len, int flags);

ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t
addrlen);
```

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

ssize_t recv(int sockfd, void *buf, size_t len, int flags);

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen);
```

Dependiendo del dominio y tipo de socket las funciones tienen un comportamiento un tanto diferente.
Tanto send/recv trabajan sobre sockets conectados.
sendto/recvfrom permiten trabajar sobre sockets orientados al envío y recepción de mensajes o datagramas.

Ejemplo Servidor Echo con Datagramas

Para el ejemplo del Servidor de Echo vamos a tener 2 procesos:

- El Cliente enviará un mensaje al Servidor, y luego esperará la respuesta del mismo que mostrará en pantalla
- El Servidor está a la espera de que algún cliente le envíe un mensaje, y se lo responderá.

Para eso necesitamos dos sockets, uno para el cliente y uno para el servidor.

Ahora veamos cómo montar un simple servidor utilizando sockets orientados a la conexión.
Y para esto vamos a tener que implementar la infraestructura de hacer una conexión cuando antes simplemente intercambiamos mensajes en diferentes procesos.
Ahora el servidor en vez de estar a la espera de mensajes va a tener que estar a la espera de procesos que intenten conectarse, y aceptar dichas conexiones.

Entonces del lado del servidor introduciremos dos funciones: listen(2), accept(2):

+ La función bloquea el servidor a la espera de clientes que quieran iniciar una conexión
+ La función accept acepta y establece dichas conexiones.

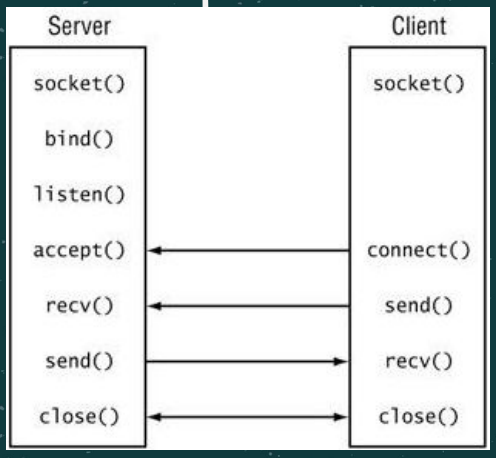
Mientras que del lado del cliente introducimos la función `connect` para conectar sockets.

Recuerden que las los sockets orientados a la conexión establece una conexión antes de empezar a enviar mensajes. Es bastante similar a `bind` pero en este caso espera que se pueda establecer una conexión entre el socket y lo que sea que esté en la dirección que apunta `addr`.

Vamos a hacer lo mismo que antes pero esta vez utilizaremos una conexión!

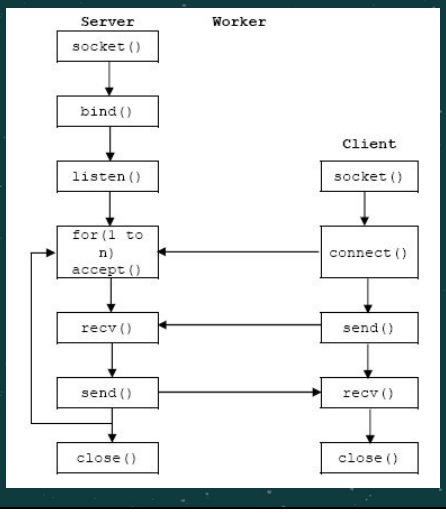
Ejemplo Servidor con AF_INET

Conexiones de tipo STREAM



En general el procedimiento será siempre el mismo, desde el lado del servidor se creará un socket, se le asignará una dirección, y se pondrá a la espera de conexiones.
Las conexiones las aceptará con `accept` y comenzará la comunicación con el cliente.
Desde el lado del cliente es simplemente intentar establecer la conexión.

Servidores



Aunque en realidad los servidores tienen un patrón más similar a este. Donde en realidad se quedan esperando a que diferentes clientes aparezcan.