# REDES DE COMPUTADORAS

## Modelo de Interacción Cliente-Servidor

# Paradigma Cliente – Servidor

- Es la base conceptual para prácticamente todas las aplicaciones distribuídas.
- Un programa inicia la interacción a lo cual otro programa responde.
  - Nota:  Las aplicaciones “peer-to-peer” usan el paradigma cliente-servidor internamente.

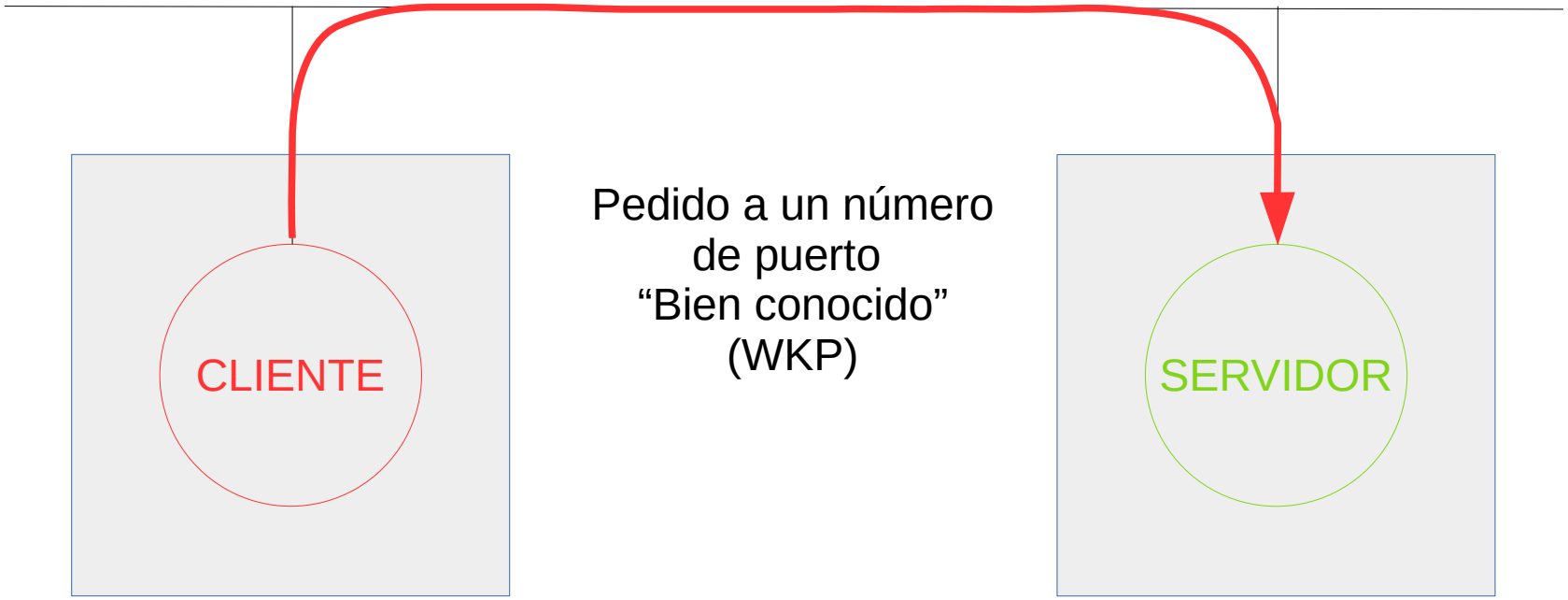
# Definiciones

- Cliente
  - Cualquier programa de aplicación puede ser cliente
  - Contacta a un servidor
  - Elabora y envía un pedido (request)
  - Espera una respuesta
- Servidor
  - Habitualmente un programa especializado que ofrece un servicio
  - Espera por pedidos
  - Calcula o elabora una respuesta
  - Envía la respuesta

# Persistencia del Servidor

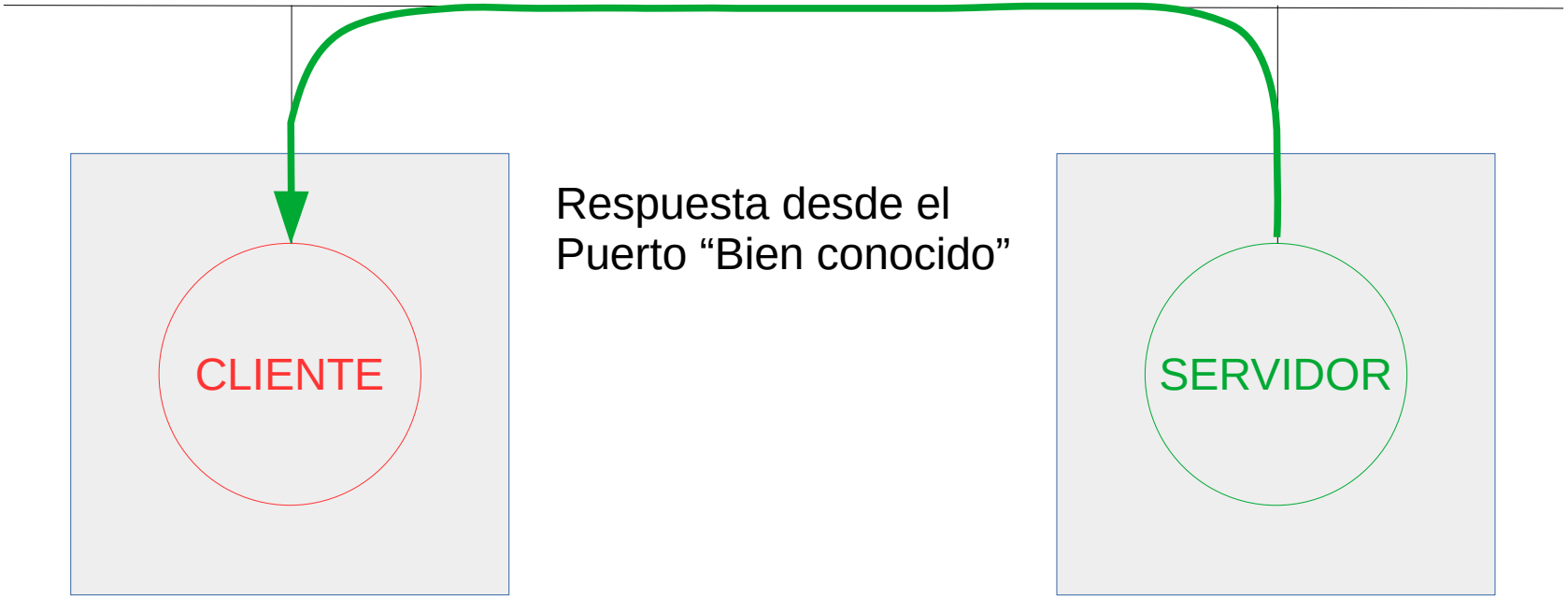
- Un servidor, comienza a ejecutarse antes de que comience la interacción y (habitualmente) continúa aceptando pedidos y contestando sin terminar su ejecución.
- Un cliente es un programa que elabora y envía un pedido y espera una respuesta; (habitualmente) termina luego de usar el servidor un número finito de veces.

# Paradigma Cliente-Servidor no concurrente




Cliente Envía Pedido desde Port Number de Cliente hacia WKP del Servidor

# Paradigma Cliente-Servidor no concurrente




Servidor contesta desde WKP del Servidor hacia el puerto del cliente desde el que se originó el pedido.

# Uso de Puertos de Protocolo

- Un servidor espera por pedidos en un puerto “Bien conocido” que ha sido reservado para el servicio que ofrece.
- Un Cliente utiliza un puerto arbitrario, no usado y no reservado para iniciar la comunicación

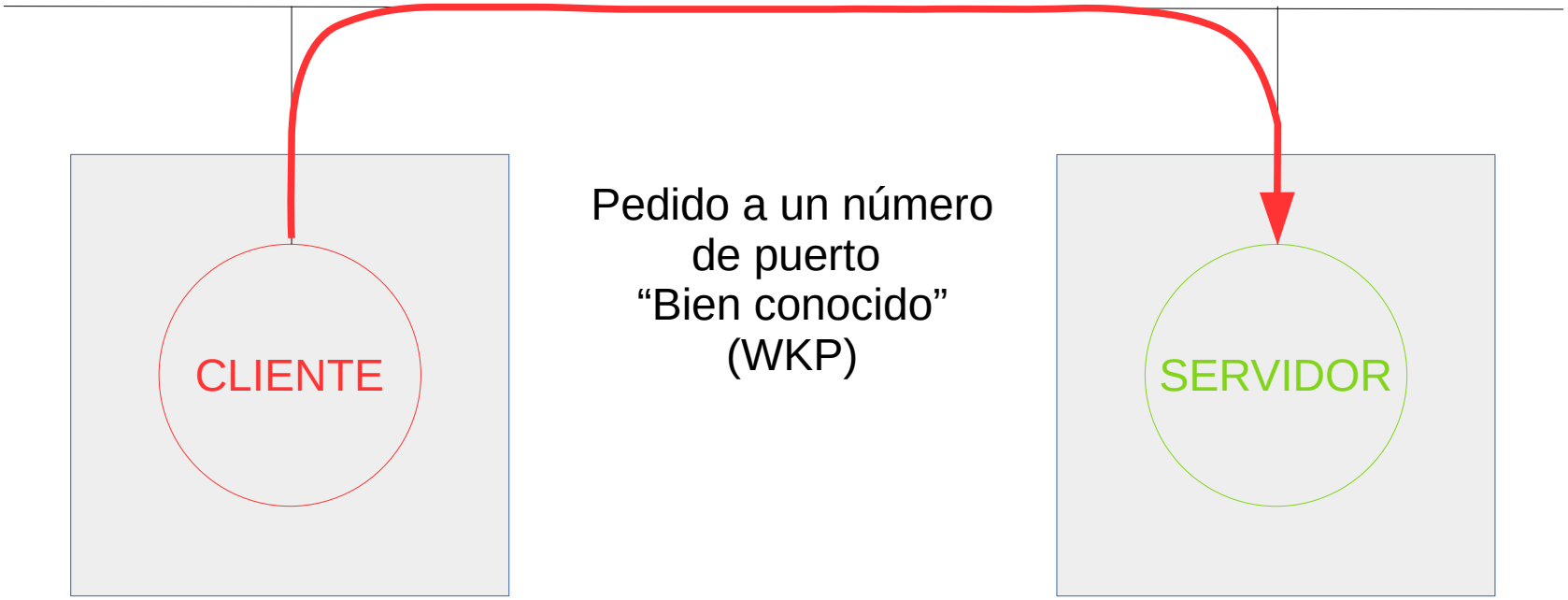
# Lado Cliente

- Cualquier aplicación puede convertirse o actuar como cliente.
- Debe saber como acceder al servidor, para eso necesita:
  - Dirección IP del Servidor
  - Numero de puerto del protocolo que usa el Servidor.
- Casi siempre es fácil de implementar

# Lado Servidor

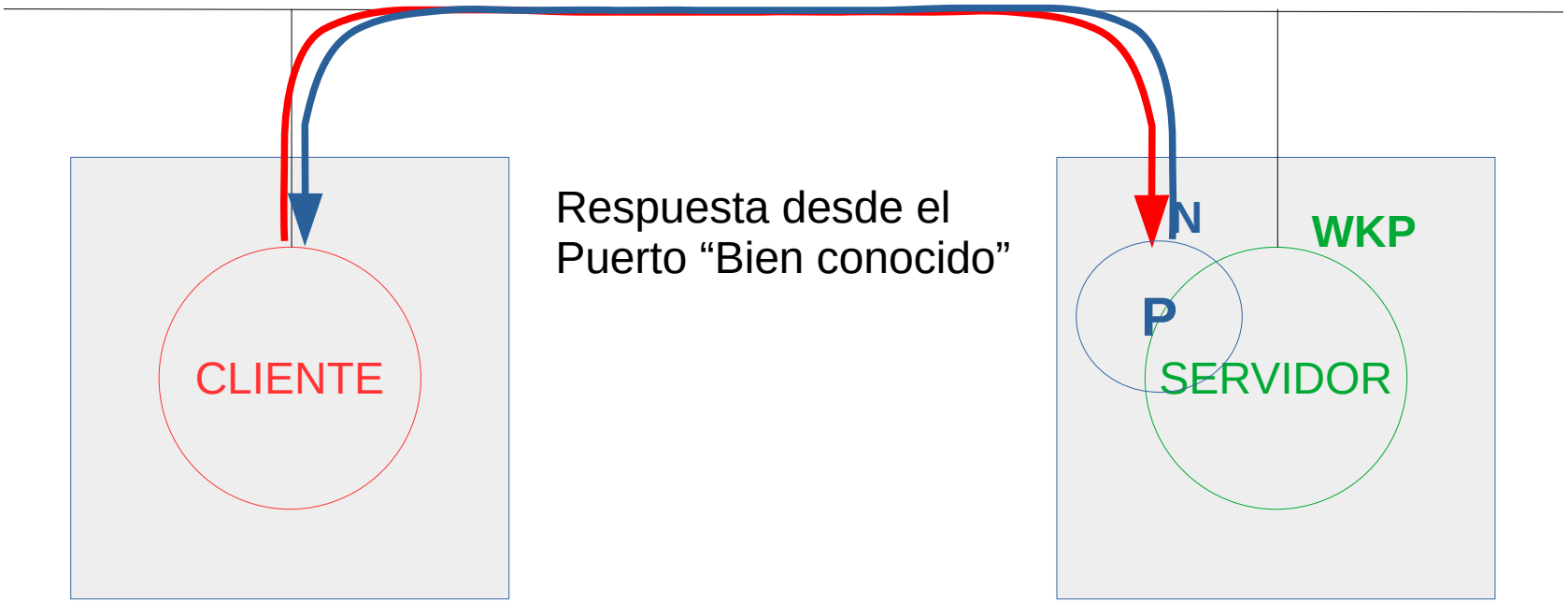
- Encuentra la ubicación del cliente a partir del pedido recibido
- Puede ser implementado con un programa de aplicación o puede venir incluido en el sistema operativo
- Comienza su ejecución antes de recibir pedidos.
- Debe asegurarse que el cliente esté autorizado a usarlo
- Debe hacer cumplir  las reglas de defensa y protección

# Algoritmo de Servidor Concurrente




Cliente Envía Pedido desde Port Number de Cliente hacia WKP del Servidor

# Algoritmo de Servidor Concurrente




Servidor crea un nuevo puerto **N** y un nuevo proceso **P** para interactuar con el cliente.

Mientras tanto, sigue “escuchando” en WKP posibles pedidos de otros clientes con el proceso original.

# Algoritmo de Servidor Concurrente


Cuando el cliente termina de interactuar, el servidor termina el proceso y cierra el puerto creados para atender al cliente.
El proceso original sigue “escuchando” en WKP posibles pedidos de nuevos clientes.

# Algoritmo de Servidor Concurrente

- Abrir un puerto Bien conocido  (well-known port)
- Esperar por el próximo pedido de cliente
- Crear un nuevo  puerto para el cliente
- Crear un hilo o proceso para manejar el pedido del cliente y asociarlo con el nuevo puerto.
- Volver a esperar el proximo pedido

# Complejidad de los Servidores

- Los Servidores son comúnmente mas difíciles de implementar que los clientes ya que aunque pueden ser realizados como programas de aplicación, deben verificar y hacer cumplir todas las políticas de seguridad y acceso del sistema en el cual son ejecutados y deben protegerse a si mismos de todos los posibles errores.

# Resumen

- El modelo Cliente Servidor, es la base para las aplicaciones distribuídas.
- Un Servidor es un programa especializado que ofrece un servicio.
- Hacer cumplir las politicas de seguridad y acceso, puede hacer que aumente la complejidad del programa servidor
- Otro aumento de la complejidad  puede producirse si se desea que el programa se proteja a si mismo de posibles errores
- Cualquier aplicación puede convertirse en cliente contactando un servidor y enviando un pedido.
- La mayoría de los servidores son concurrentes.

# REDES DE COMPUTADORAS
## Socket API

# Protocolos de Red

- El software que implementa los protocolos de red, está incluido en el SO
- Las aplicaciones acceden a utilizar los protocolos a través de una Interfaz de Programación de  Aplicaciones (Application Program Interface – API)

# Sockets API

- Los estándares TCP/IP describen la funcionalidad necesaria, pero no dan detalles de como se accede a ella.
- Cada SO puede definir su propia API
- En la práctica la Interfaz de Sockets (Socket Interface) se convirtió en el estándar de facto para la utilización de redes.
- Fue definida por la universidad de California en Berkeley como parte de la versión BSD UNIX.
- Microsoft la adoptó y modificó para Windows (Windows Sockets)

# Características de la API de Sockets

- En UNIX, cada recurso puede accederse siguiendo el paradigma *open – read – write – close.*
- La API de sockets también puede accederse de esta manera.
- Usa la abstracción de UNIX denominada *descriptor*

  1) Se crea un socket y se asocia a un *descriptor* entero

  2) Se llama a un conjunto de instrucciones que especifican los detalles del socket, usando el descriptor como argumento.
- Una vez que se estableció el *socket* se utilizan `read()` y `write()` o funciones equivalentes para transferir datos
- Cuando se finaliza se usa *close()* para liberar los recursos

# Creación de un socket

```
#include <sys/types.h>
#include <sys/socket.h>
int descriptor;
descriptor = socket(int pf, int type, int protocol);
```

- pf: familia de protocolos: AF_INET, AF_INET6, UNIX, etc.
- type: tipo de datos SOCK_DGRAM, SOCK_STREAM, SOCK_RAW, etc
- protocol: numero de protocolo (header IP). En la mayoría de las implementaciones, se usa 0 y el SO elije el protocolo adecuado. ( p. ej UDP para SOCK_DGRAM o TCP para SOCK_STREAM)

# Utilización del socket para UDP – TCP

- En el momento de creación se decide si el socket se va a utilizar para UDP (SOCK_DGRAM) o para TCP (SOCK_STREAM).
- Además debe indicarse si se va a utilizar como servidor o como cliente.
- Si se utiliza como servidor, debe asociarse a un PORT al cual deben dirigirse los pedidos de los clientes.
- Para eso se utiliza la función ***bind()***

# Establecer PORT del socket ( UDP y TCP)

- Dirección local (asociar un socket a un port)
- **`int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`**
  - Necesario para servidores, opcional para clientes
- **`sockfd:`** Descriptor de Socket
- **`struct sockaddr *addr:`** Estructura con descripción de la dirección local, esta estructura se define genericamente y luego se adapta a las distintas familias de protocolos ( AF_INET, AF_UNIX, etc.).
  - ```
    struct sockaddr {
                sa_family_t sa_family; /*(unsigned short)*/
                char        sa_data[14];
            };
    ```
- **`socklen_t addrlen:`** Tamaño de la estructura **sockaddr.** Se suele usar **`sizeof(sockaddr)`**
    ```

# Diagrama de estructuras Sockaddr

**Struct sockaddr**

| 0 | 16 | 31 |
|---|---|---|
| Familia de Direcciones (2) | Dirección, octetos 0 – 1 | |
| Dirección, octetos 2 – 5 | | |
| Dirección, octetos 6 – 9 | | |
| Dirección, octetos 10 – 13 | | |

**Para TCP/IP se usa Struct sockaddr_in**

| 0 | 16 | 31 |
|---|---|---|
| Familia de Direcciones (2) | Port (2) | |
| Dirección IP (4) | | |

# struct sockaddr_in

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* Port en el orden de bits de la red */
    struct in_addr sin_addr;   /* IP address */
    };

struct in_addr {
    uint32_t s_addr;      /* IP address en el orden de bits de la red */
    };
```

# UDP SERVER: *recvfrom()*

- El server UDP queda a la escucha de datagramas usando:

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

  - **sockfd:** descriptor del socket
  - **buf:** buffer donde se reciben los datos
  - **len:** tamaño del buffer
  - **flags:** opciones
  - **src_addr:** estructura con la dirección del que envió los datos
  - **addrlen:** tamaño de src_addr
- La función devuelve la cantidad de bytes efectivamente recibidos.

# UDP CLIENTE: *sendto()*

- Un cliente UDP, solamente necesita abrir un socket y luego enviar datagramas utilizando ***sendto()***

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

- ***buf*** y ***len*** indican el buffer donde se encuentran los datos a enviar y su tamaño respectivamente.
- ***flags*** permite ajustar opciones
- ***dest_addr*** y ***addrlen*** dan la información del destino de los datos .( IP y PORT)
- La función devuelve la cantidad de bytes enviados
- Si el cliente espera una respuesta del servidor, puede llamar a las funciones ***recvfrom()*** y ***sendto()*** en forma intercalada, El serviidor debe proceder de la misma manera

# TCP SERVER: *listen()*

- Como TCP establece una conexión virtual, incluye más complejidad que UDP.
- Además de ***bind()***, TCP usa ***listen()*** para indicar que el socket espera pedidos de conexión.

  ***listen(socket, qlength)***

- ***socket:*** descriptor del socket
- ***qlength:*** cantidad de conexiones pendientes que pueden encolarse, suele ser un número pequeño (5 – 10)

  La función ***listen()*** pone el socket en modo pasivo, es decir que le permite recibir pedidos de conexión que son encolados en la cola de tamaño ***qlength.***

# TCP SERVER: *accept()*

- Para procesar los pedidos de conexión entrantes se utiliza la función ***accept()***

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

  - ***sockfd: descriptor del socket***
  - ***addr: estructura con la dirección del que envió los datos***
  - ***addrlen: tamaño de addr***

- ***accept()*** extrae el pedido de conexión entrante de la cola y crea un nuevo socket para su posterior procesamiento, devuelve un descriptor del nuevo socket y deja liberado el socket original para atender nuevos pedidos de conexión.
- Si no hay pedidos de conexión en la cola creada por ***bind()***, la función es bloqueante

# TCP CLIENTE *connect( )*

- Para establecer una conexión TCP con un servidor que posea un socket en estado LISTEN, se utiliza la función ***connect()***

  ***int connect(int sockfd, const struct sockaddr *addr,***

  ***socklen_t addrlen);***

- ***sockfd:*** socket del cliente a través del cual se producirá la conexión
- ***addr:*** estructura con la información de dirección del servidor.
- ***addrlen;*** tamaño de la estructura ***addr***

# TCP CLIENTE y SERVIDOR *recv( )*

- Para recibir información a través de la conexión TCP, se utiliza la función ***recv().***
- ***ssize_t recv(int sockfd, void *buf, size_t len, int flags);***
- ***sockfd:*** socket del cliente a través del cual se recibirán los datos.
- ***buf y len*** son el buffer donde se recibirán los datos y su tamaño respectivamente.
- ***flags:*** opciones.
- La función pueden utilizarla tanto el servidor como el cliente, ya que la conexión TCP es *bidireccional* y *full duplex*
- La función devuelve el número de bytes efectivamente recibidos.

# TCP CLIENTE y SERVIDOR send( )

- Para enviar información a través de la conexión TCP, se utiliza la función ***send().***

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

- ***sockfd:*** socket del cliente/servidor a través del cual se enviarán los datos.
- ***buf y len*** son el buffer que contiene los datos a enviar y su longitud respectivamente.
- ***flags:*** opciones.
- La función pueden utilizarla tanto el servidor como el cliente, ya que la conexión TCP es *bidireccional* y *full duplex*
- La función devuelve el número de bytes efectivamente enviados.

# Cierre del socket: *close() (TCP y UDP)*

- El cierre del socket y terminación de la conexión en el caso de TCP se produce utilizando la función ***close().***
- ***int close( sockfd)***
- ***sockfd:*** socket del cliente/servidor que se quiere cerrar.
- La función devuelve 0 si todo salió bien y -1 en caso de error.