

## UN POQUITO DE TEORÍA

El concepto de socket y de programación con sockets se desarrolló en los años ochenta en el entorno UNIX como la Interfaz de Sockets de Berkeley.

Un socket, por su traducción al Castellano, es un "**enchufe**", es decir, una conexión con otro ordenador que nos permitirá intercambiar datos. Por así decirlo, cuando quiere conectarse a una pagina web, el navegador, que es un programa, crea un socket, que contendrá información acerca del servidor, información suficiente para poder realizar una asociación, y de esta manera poder ejercer una conexión.

La definición y utilización de los sockets es clara:

‘Conectar dos ordenadores para que puedan comunicarse entre ellos de forma remota o local.’

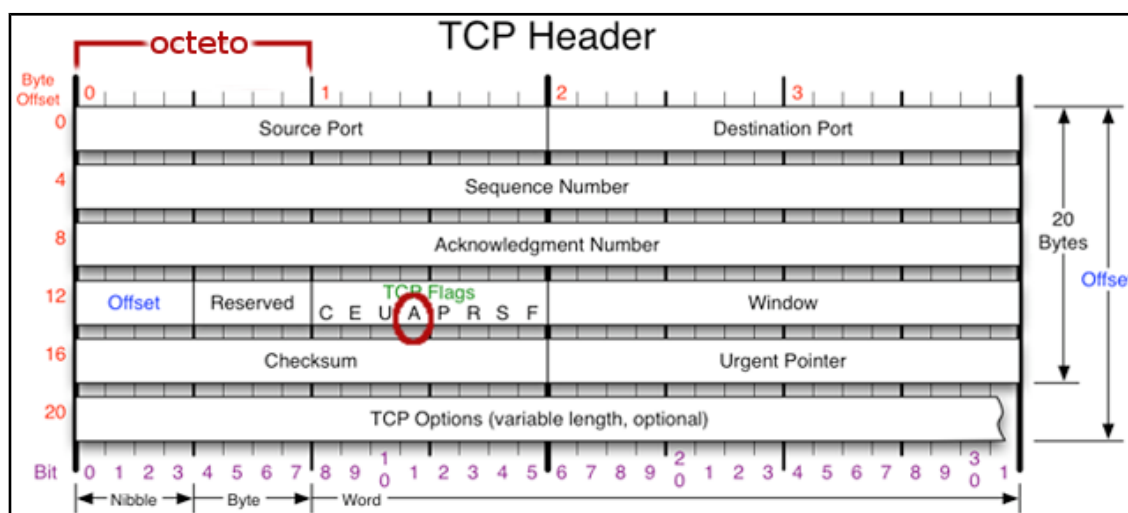


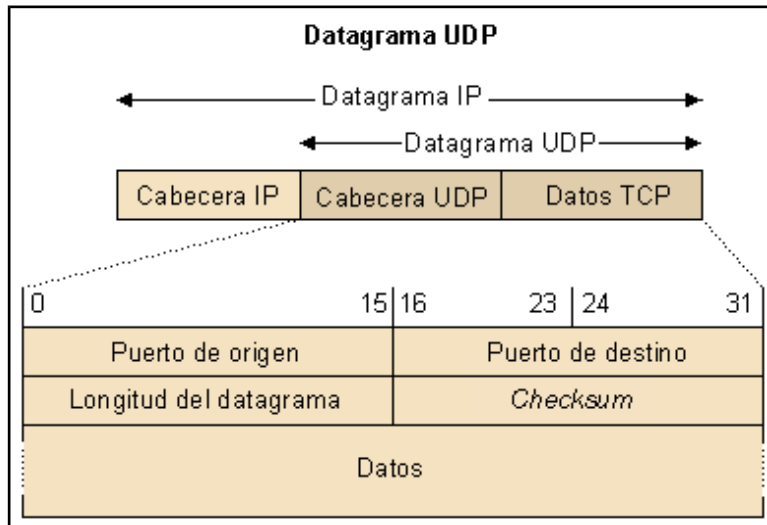
## UN POQUITO MÁS ...

Un socket permite la comunicación entre un proceso cliente y un proceso servidor y puede ser orientado a conexión o no orientado a conexión. Una vez que han entrado en comunicación los sockets, los dos computadores pueden intercambiar datos. Normalmente, los computadores con servidores basados en sockets mantienen abierto un puerto *TCP* o *UDP*, preparado para llamadas entrantes no planificadas.

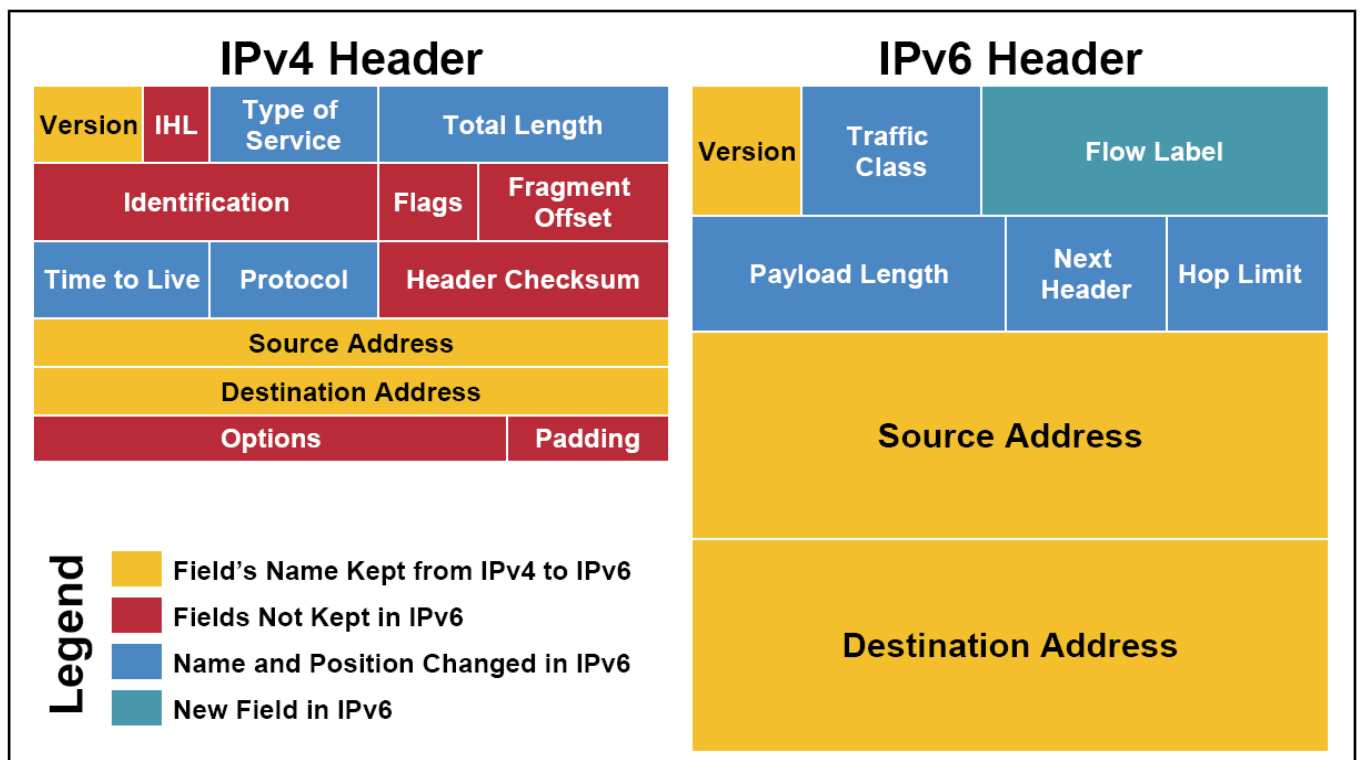
La **Berkeley Sockets Interface** es el API (Interfaz de Programación de Aplicaciones) estándar para el desarrollo de aplicaciones de red, en un amplio rango de sistemas operativos. **Windows Sockets** (WinSock) está basado en la especificación de Berkeley.

Recordar que cada cabecera **TCP** y **UDP** incluye los campos del puerto origen y puerto destino. Estos valores de puerto identifican a los respectivos usuarios (aplicaciones) de las dos entidades.





Además, cada cabecera IPv4 e IPv6 incluye los campos con las direcciones origen y destino, estas direcciones IP identifican a los respectivos sistemas.



La concatenación de un valor de puerto y una dirección IP forma un socket, que es único en Internet.

Correspondiéndose con los dos protocolos, el API de Sockets reconoce dos tipos de sockets: **sockets stream** y **sockets datagrama**.

***\*Los socket stream*** hacen uso de TCP y proporcionan una transferencia de datos fiable orientada a conexión. Por tanto, con sockets stream, se garantiza que todos los bloques de datos enviados entre un par de sockets llegan en el orden en que se enviaron.

***\*Los sockets datagrama***, hacen uso de UDP, que no proporciona las características orientadas a conexión de TCP. Por tanto, con sockets datagrama, ni se garantiza la entrega, ni que se mantenga el orden.

(Esto no significa que uno sea mejor o peor, no hay que caer en eso, significa que cada tipo de socket tiene su uso)

Hay un tercer tipo de sockets proporcionados por el API de Sockets: **sockets raw**. Los sockets raw permiten acceso directo a las capas más bajas del protocolo, tales como IP.

Y ahora... empecemos a darle caña!

## DISEÑO E IMPLEMENTACIÓN DE SOCKETS I

Con el fin de hacer más comprensible la implementación, la dividiremos en fases.

1. ***Configuración o puesta a punto del Socket:*** para crear nuestro socket haremos uso de la llamada a la función **socket()**. Dicha función posee 3 parámetros:
  - La familia del Protocolo: que dependerá del tipo de conexiones que utilicemos, existen dos que son las mas relevantes: **AF\_INET** para conexiones a Internet, y **AF\_UNIX** que es un tipo de conexión que se utiliza para conexiones internas de la máquina.
  - El Tipo: que dependerá del tipo de socket que deseamos utilizar, ya sea **socket stream** o **datagrama**. (para TCP o UDP)
  - El Protocolo: se diseñó para permitir nuevos protocolos de nivel de transporte en futuras implementaciones. Por defecto nosotros lo estableceremos a **0** para usar un sólo protocolo.

La función socket() devuelve un valor entero que identifica este socket, es similar a un **descriptor de fichero** UNIX.

2. ***Bindeado o asociación del socket a su estructura:*** pero antes de ello, debemos conocer la estructura del socket (la información para diferenciarlo de otros), que es la que se pasará a la función bind():

La estructura es: **struct sockaddr\_in**. Esa estructura contiene 4 elementos principales:

- **sin\_family** -> Protocolo para la conexión (AF\_INET o AF\_UNIX).
- **sin\_port** -> Puerto sobre el que aceptará las peticiones.
- **struct in\_addr sin\_addr** -> Estructura que contiene un elemento:
  - **unsigned long s\_addr** -> Elemento que refiere a la IP del Host.
- **unsigned char sin\_zero[8]** -> Relleno de ceros.

### *¿Por qué esta estructura?*

Porque en primer lugar, C necesita saber de que tipo de socket se trata (si es un stream socket o un datagrama), en segundo lugar necesita saber el puerto que pone a la escucha (y como va a ser del tipo **Ordenación de Bytes para Redes** con htons() ).

Accediendo al elemento de la estructura **in\_addr**, ira la IP del Host, en formato entero largo sin signo (**unsigned long**) que obtendremos con inet\_addr() . Y finalmente, rellenamos con ceros el elemento restante. Con **INADDR\_ANY** en el **server.sin\_addr.s\_addr** nos coloca la dirección IP automáticamente.

### *¿Y por qué no usar Sockaddr?*

La estructura sockaddr no se utiliza ya que es mucho mas sencillo manejar e introducir los datos necesarios con la estructura sockaddr\_in, que ya viene con sus elementos del struct organizados. El tamaño de ambas estructuras va a ser el mismo, ya sea de sockaddr y de sockaddr\_in:

- **Tamaño de sockaddr:** **2 bytes** del unsigned short, y **14** del array de char (*1 byte por cada char*) = **16 bytes**.
- **Tamaño de sockaddr\_in:** **2 bytes** del short int, **2** del unsigned short int, el struct es igual a la suma del tamaño de todos sus elementos, en este caso tenemos un unsigned long (IP) que son **4 bytes**, y por último un array de unsigned char de 8 elementos (a rellenar con 0), es decir, **8 bytes** = total de **16 bytes**.

Fíjate que si no rellenáramos los 8 bytes con ceros de la estructura sockaddr\_in, nos faltarían para llegar al tamaño de la estructura sockaddr. De esta forma, igualamos la cantidad de bytes de cada una de las estructuras.

<http://decsai.ugr.es/~jfv/ed1/c/cdrom/cap2/cap24.htm> (Tamaños en Bytes de cada variable)

Entonces, tenemos dos estructuras, del mismo tamaño, con los campos, en un principio, similares, ya que tenemos el primer elemento igual, que serían 2 bytes para el protocolo, y luego 14 bytes restantes para los datos adicionales.

Pero hay una diferencia, y es que **bind()** solicita un **struct sockaddr\*** es decir, un puntero a una estructura del tipo sockaddr, por ello no nos basta tan sólo con meter la dirección de memoria del struct sockaddr\_in (& sockaddr\_in), por el que queremos sustituir, ya que, si a pesar de que ambos contienen una dirección/posición de memoria, no apuntan al mismo elemento, es decir no es lo mismo un puntero a **x** que un puntero a **y**. Intentando hacer esta asignación nos daría un warning como una casa o puede que un error. Así que por ello hacemos esa **reconversión forzosa** o **cast**.

**Forzamos a convertir a un puntero de struct sockaddr a la dirección de memoria de la variable del struct sockaddr\_in** (esto último es lo que haces con el operador &). Quedando así:

**(struct sockaddr \*) &servidor;** (siendo **servidor** nuestra estructura sockaddr\_in)

Es necesario concretar una serie de **funciones de conversión** para establecer estos valores correctamente cuando rellenemos nuestra estructura del socket para llamar a bind():

- La función **gethostbyname()** nos permitirá, pasando le un **dns** o **nombre de dominio** (not DeNegacion de Servicio), obtener la dirección IP del host al que nos referimos. Pero como citan en las **páginas MAN**, esta función se encuentra obsoleta, en su lugar recomiendan usar: **getaddrinfo()**, que podéis consultar vosotros, aunque para la simple acción de resolver un dns, nosotros nos serviremos de gethostbyname.

A pesar de lo anterior, recalcar que *The gethostbyname() function returns a structure of type hostent for the given host name*, de donde podemos acceder al campo **char \*h\_name** para obtener el dns. Para que esta función se pueda utilizar se ha de añadir la librería **<netdb.h>** en las cabeceras.

- La función **gethostname()** (no confundir con la anterior), nos proporciona el nombre de la máquina local. Requiere de 2 parámetros: **una cadena** (ya que char\* es similar a char[]) {\*1}, y **el tamaño de dicha cadenas** decir, establecer un **sizeof()** de la cadena que guardará el nombre del equipo.

Una vez formada la estructura del socket, procedemos a su bindeado, para ello usaremos la función **bind()**, que posee 3 parámetros:

- Descriptor del Socket:** es el entero que nos devuelve la función socket(). Recordamos la frase de: **"En Linux todo es un archivo"**, teniendo esta en cuenta, 'al utilizar las funciones de E/S para ejecutar operaciones en un archivo, el archivo se identifica mediante la utilización de punteros. Al utilizar las funciones C del sistema de archivos integrado, el archivo se identifica especificando un descriptor de archivo. Un descriptor de archivo es un entero positivo que debe ser **exclusivo en cada trabajo**. El trabajo utiliza un descriptor de archivo para **identificar un archivo abierto** al realizar operaciones sobre el archivo`.

Fragmento extraído de [www.ibm.com](http://www.ibm.com).

Sabiendo lo anterior, podemos darnos cuenta que un descriptor hace referencia a un archivo abierto, aportándonos una serie de información de dicho archivo y permitiendo su manipulación. { \*2 }

- **Puntero a nuestra estructura del socket:** este es el caso que hemos citado y explicado anteriormente en la sección de *¿Y por qué no usar Sockaddr?* Para utilizar la re-conversión y referencia a nuestra estructura sockaddr\_in.
- **El tamaño de la estructura Sockaddr:** no es más que una medida de seguridad, se le pasa un sizeof() con la estructura sockaddr y pista.

La función bind() nos devolverá un **-1** en el caso de ocurrir algún **error**. Por eso se debe verificar el retorno.

\* Hay que recordar que **TODOS los puertos menores al 1024 SÓLO SE PODRÁN ASOCIAR CON PERMISOS DE SUPERUSUARIO**. Se podrá establecer un puerto, siempre que esté entre 1024 y 65535 (y siempre que no estén siendo usados por otros programas).

3. Una vez tenemos nuestro socket asignado a un puerto (resumen del funcionamiento de bind), procedemos a **ponerlo a la escucha**, es decir, utilizando **listen()**, que nos pedirá 2 parámetros:

- **Descriptor del Socket:** recordamos que es el entero que nos devuelve la función socket().
- **El BackLog{\*3}:** el backlog es el *número de conexiones no aceptadas que podemos tener en cola*. Por defecto se encuentra en `/proc/sys/net/ipv4/tcp_max_syn_backlog`. Se mantiene en cola hasta que no haces un accept, en ese momento, la conexión pasa a tener su propio descriptor de fichero y se sale del backlog.

De igual forma que bind(), si falla, nos retornará **-1**, así que ya sabéis que hacer. Recordar que la función bind() es opcional, sino se hace, se asignará un puerto por defecto.

En este momento, tenemos nuestro puerto a la escucha. Podríamos comprobarlo añadiendo un simple bucle infinito, y una vez compilado y ejecutado, haciendo un **sudo netstat -putona** (para las conexiones de la máquina) y obtendríamos que, entre otros puertos, el que hemos especificado nosotros, estaría en **LISTEN**.

En esta captura lo podemos ver claramente, junto con el puerto que hemos abierto (**10000**) y el estado **LISTEN** junto con el **PID** del proceso (de nuestro programa server) **6478**.

```
[zero@x11 server]$ sudo netstat -putona | grep server
Active Internet connections (servers and established)
tcp        0      0 0.0.0.0:10000        0.0.0.0:*          LISTEN      6478/./server      off (0.00/0/0)
[zero@x11 server]$ gnome-screenshot -a
```

4. Pero... si intentamos conectarnos a él, veremos que no podemos, porque sólo esta a la escucha, no hay ninguna forma de permitir que se establezca dicha conexión (esto daría **CONNECTED** en netstat) y empezar a recibir datos. Para ello necesitamos ver una función que es **accept()**, que precisa de 3 parámetros:

- **Descriptor del Socket:** el de siempre, el de toda la vida. Esto se debe a que podemos poner varios sockets a la escucha, y por tanto necesitamos especificarle cual, con su descriptor.
- **Puntero a nuestra estructura sockaddr:** ahora viene la gracia; los sockets se conectan con otros sockets, y por tanto, vamos a recibir la estructura de dicho socket en el accept. Recordamos que nosotros teníamos una estructura sockaddr\_in, que identificaban a nuestro socket, es decir, al **Socket Servidor**, pues ahora, necesitamos tener otra estructura de este tipo, que identificará al **Socket Cliente**. Y como es un puntero que contiene una dirección de memoria, pues ejecutamos el mismo paso que en bind().
- **Tamaño de la estructura sockaddr:** lo mismo de bind(), le pasamos el sizeof de sockaddr.

Y esta función también retorna **-1** en el caso de fallar, pero, si no lo hace, y la conexión es correcta, **retornará el descriptor que identificará a la conexión con nuestro cliente**, por ello hay que obtener dicho valor retornado y comprobarlo.

5. Y sobretodo no podemos olvidarnos de la función que nos permitirá **establecer la conexión desde el cliente al servidor**, y no me refiero a otra que **connect()**. Para esta función le pasaremos:

- **Descriptor del Socket:** el respectivo al programa del Cliente (ya que connect se ejecuta desde el código del Cliente).
- **Estructura del Servidor:** en esta le pasaremos, de la forma que hemos explicado antes, la estructura que albergará la información del socket del Servidor.
- **Tamaño de la estructura:** igual que hemos utilizado en bind().

Si falla, como en las anteriores, nos devolverá **-1**. En caso contrario, podríamos ver, ejecutando un netstat, algo parecido a la siguiente captura (el LISTEN porque hemos puesto 2 de Backlog):

```
[zero@x11 Client]$ sudo netstat -putona | grep server
Active Internet connections (servers and established)
tcp        0      0 0.0.0.0:10000        0.0.0.0:*            LISTEN     17898/./server      off (0.00/0/0)
tcp        0      0 127.0.0.1:10000      127.0.0.1:36112      ESTABLISHED 17898/./server      off (0.00/0/0)
[zero@x11 Client]$ gnome-screenshot -a
```

Una vez conectado un cliente, procederemos a mostrar las dos **funciones esenciales para comunicarnos**, que son: send() y recv().

6. La función **send()** se utiliza para **enviar un mensaje al cliente que se ha conectado / o al servidor al que nos hemos conectado**. Para ello se requieren:

- **Descriptor del Socket del Cliente**, es decir, el retornado por la función accept().
- **Puntero a void**: es decir, un puntero al dato que se quiera enviar. Por ello es void, porque puedes enviar un carácter, una cadena de caracteres, un entero, etc.
- **Tamaño**: en bytes de/del dato/datos que se van a enviar. Sizeof() a nuestro mensaje y pista.
- **Flag**: se establecerá a **0**, por defecto no vamos a utilizar ningún flag, sin embargo, pueden leerse cada uno de ello usando **man 2 send** . En el apartado de 'The flags argument'.

Al igual que todas las demás llamadas que hemos visto, send() devuelve **-1** en caso de error, o **el número de bytes enviados en caso de éxito**.

7. La función **recv()** la utilizaremos para **recibir datos del cliente que se ha conectado a nuestro socket / o para recibir datos del servidor al que nos hemos conectado**. Para ello la función requiere de los siguientes parámetros:

- **Descriptor del Socket del Cliente**.
- **Puntero a void**: como en send, solo que esta vez se utilizará como buffer del mensaje a recibir.
- **Tamaño** en bytes del mensaje a recibir.
- **Flag**: se establecerá a **0** como antes.

Análogamente a send(), recv() devuelve el **número de bytes leídos en el buffer**, o **-1** si se produjo un error.

8. Y finalmente, si se ha realizado lo anterior de forma correcta, **la forma de finalizar la comunicación es cerrando el descriptor de los sockets** (Cliente y Servidor), usando la función **close()**, con un único parámetro y es el *descriptor del socket que deseamos cerrar*.

Aprovecharé para destacar algo importante:

Una alternativa al uso de las funciones send() y recv(), son las funciones **write()** y **read()**, respectivamente. Pero, ya no sólo en el tema de sockets, sino que, estas funciones, se utilizan generalmente para leer cualquier tipo de fichero o archivo (cabe recordar la filosofía de Linux).



La diferencia de estas, con, por ejemplo, `fwrite()` y `fread()`, son varias:

-En primer lugar, hacer hincapié en que **no nos sirven para sockets `fread()` y `fwrite()`**, ya que estas **sólo sirven para leer y escribir en archivos abiertos con `fopen()`**, que retorna el tipo `FILE*`, (*'the standard C way of handling file IO'*). Por tanto, no puedes usar estas dos funciones con sockets, pero si sobre ficheros normales (los acostumbrados a manejar como pueden ser textos, imágenes, etc), ya que `fread()` y `fwrite()` pueden tratar con bits (al igual que `read` y `write`), la diferencia es a la hora de acceder al disco: `fread()` utiliza de un buffer intermedio, mediante el cual, realiza retiradas de bloques de tamaño 8KB al buffer, y luego te ira dando los segmentos que tu le vas pidiendo hasta que se le vacía el buffer y pide mas al disco.

<https://softwareengineering.stackexchange.com/questions/171315/optimal-buffer-size-for-fread-fwrite>

- En segundo lugar, te beneficia usar `read` para archivos grandes, ya que podrás utilizar un tamaño de bloque superior al que puede utilizar `fread()`, y por tanto cubrir más bloques por acceso a disco. Y de igual forma, con mismo tamaño de bloque, `fread()` siempre hará un acceso más, que no es más que el de pasarlo al buffer de donde tú iras solicitando pedazos.

La función **`read()`**, como podemos ver, posee 3 parámetros:

- **`int fildes`**: descriptor de archivo, aquél del que tanto hemos hablado.
- **`void *buf`**: un puntero al buffer donde almacenaremos el mensaje que leamos.
- **`size_t nbyte`**: número de bytes a leer del socket, recomendable que sea proporcional al tamaño del buffer de nuestro mensaje.

`read()` devuelve el **número de bytes leídos** del descriptor del archivo.

De forma muy similar a la anterior, la función **`write()`**, opera con los siguientes parámetros:

- **`int fildes`**: descriptor de archivo, aquél del que tanto hemos hablado.
- **`const void *buf`**: puntero a un constante, esto se debe a que **C precisa de la correctitud o inmutabilidad**, que hace referencia a la adecuada declaración de una variable. El mensaje no puede modificarse al tiempo de escribirlo. De ahí el `const` requerido.
- **`size_t nbyte`**: número de bytes a leer del socket, recomendable que sea proporcional al tamaño del buffer de nuestro mensaje.

`write()` devuelve el **número de bytes escritos** sobre el descriptor del archivo.

## **REFERENCIAS:**

<http://es.tldp.org/Tutoriales/PROG-SOCKETS/prog-sockets.html>

[https://es.wikibooks.org/wiki/Programaci%C3%B3n\\_en\\_C/Sockets](https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C/Sockets)

<http://wiki.elhacker.net/programacion/cc/articulos/introduccion-a-los-sockets-en-ansi-c>

<http://informatica.uv.es/iiguia/R/apuntes/laboratorio/Usos.pdf>

Para concluir, espero que os haya servido mucho y gracias a Fran por ayudarme con la recopilación y estudio. Y al compañero L por la paciencia infinita que hay que tener con este servidor. Os deseo lo mejor, By **Secury**

*Hasta la próxima!*