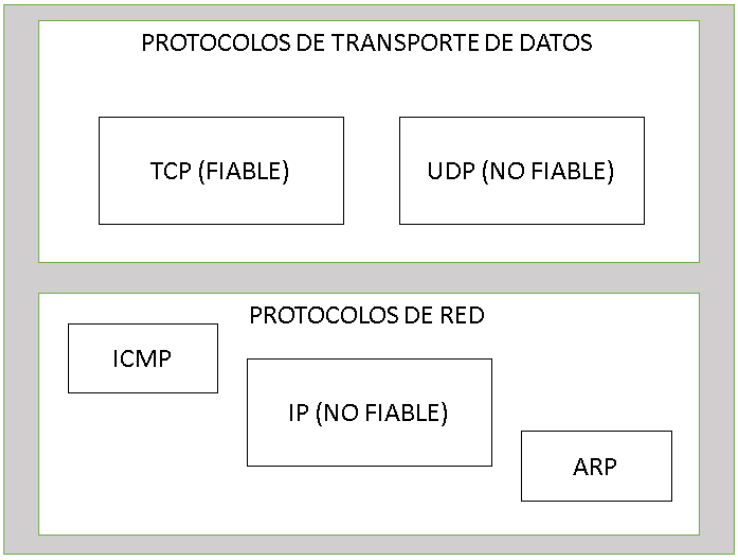
COMUNICACIÓN ENTRE PROCESOS (3ª PARTE): TUBERÍAS BIDIRECCIONALES Y FIABLES POR INTERNET

En esta ocasión vamos a utilizar los sockets de una manera totalmente fiable, puesto que vamos a decirle al sistema operativo que, en vez de utilizar el protocolo de transporte de datos UDP, utilice el denominado TCP (transmission control protocol).

Antes de pasar a la acción, hagamos un breve recordatorio de los protocolos implementados en el kernel:

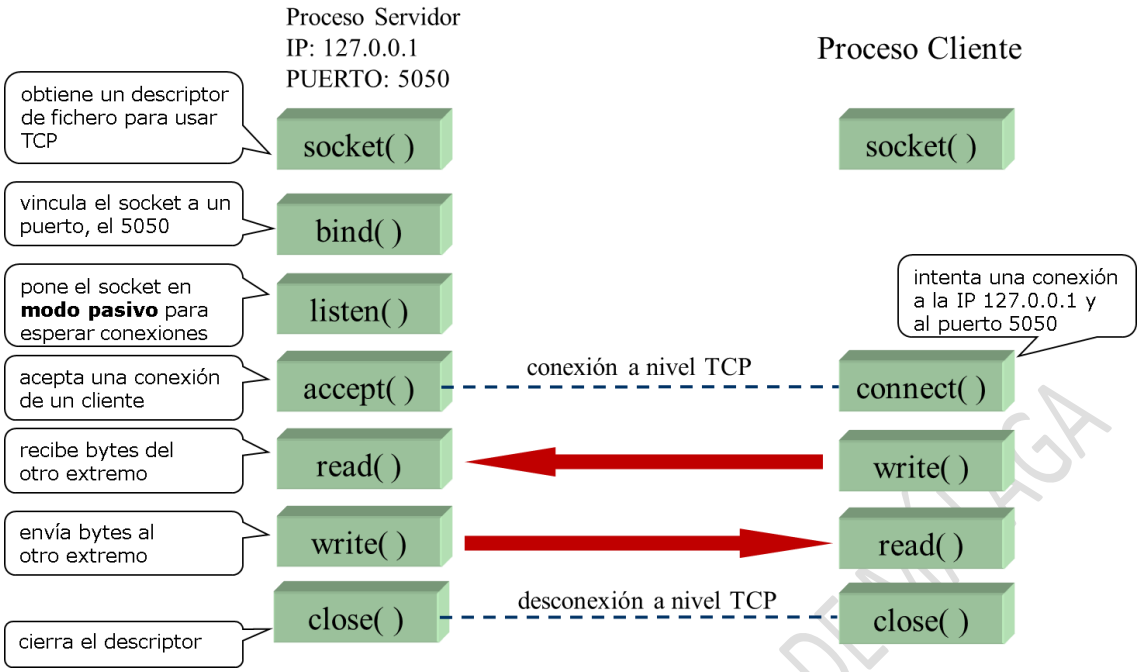


Los datos de los procesos clientes y servidores viajan dentro de mensajes TCP o UDP, que se intercambian los sistemas operativos de cada ordenador. Cuando se selecciona UDP, estamos eligiendo que el transporte de datos se haga sin ninguna fiabilidad. Sin embargo, en muchas ocasiones interesa asegurar que los datos llegan ordenados al destino, sin pérdidas ni duplicados posibles. Así que, en este caso, nuestras aplicaciones deben utilizar TCP.

¿Y cómo consigue un protocolo como TCP hacer lo que no consigue el pobre UDP? Pues a través de varias características, que veremos en el tema siguiente. Quizá la más interesante, y que nos afecta a la hora de montar nuestros programas con sockets TCP es que, para que el protocolo funcione de forma fiable, se necesita OBLIGATORIAMENTE establecer una conexión entre el cliente y el servidor.

Y la otra diferencia más apreciable es que UDP entregaba mensajes completos, pero TCP entrega bytes ordenados. Eso quiere decir los dos canales del kernel asociados al socket TCP (tanto el de transmisión como el de recepción) se comportan como TUBERÍAS. Sí, sí, como estas leyendo. Se comportan de forma idéntica a las pipes o las fifos ya vistas en cuanto a la escritura y lectura de bytes. Así que tenemos casi todo el trabajo ya hecho (y lo repasaremos más adelante).

¡No hacen falta más preámbulos! Te enseño el esquema de llamadas al sistema de un servidor y un cliente TCP, y a programar:



En la figura de arriba puedes ver que hay puntos muy similares a cuando aprendimos a usar sockets con UDP. Por lo pronto, vemos que se vuelven a utilizar socket(), bind() ¡y close()!.

¿Y qué pasa con las nuevas llamadas al sistema? Son muy sencillas:

listen() y accept() sirven para decirle al sistema operativo que este proceso va a actuar como servidor, y que va a esperar hasta que aparezca un cliente solicitando conectarse con él.

connect() es la llamada al sistema que utiliza un proceso cliente para solicitar una conexión a un servidor, y va a bloquearse hasta ser aceptado por él.

Si te das cuenta, accept() y connect() actúan como puntos de sincronización entre los procesos servidor y cliente. Ni el servidor ni el cliente pueden continuar su ejecución hasta que la conexión se haya establecido entre ambos. A partir, de ahí, pueden transmitirse bytes ¡y lo harán con las ya míticas llamadas al sistema read() y write()! Perdón, he querido decir:

-read(): si no conocemos a priori el tamaño de los bytes a leer de la tubería de recepción

-readn(): si conocemos a priori el tamaño de los bytes a leer

-writen(): siempre mejor que write(), para asegurarnos de dejar en la tubería de transmisión del kernel todos los bytes deseados (luego ya el sistema operativo los mandará al destino usando TCP).

NOTA IMPORTANTE: Supongo que quedó claro en la sesión anterior de UDP, pero para que estos dos procesos se comuniquen a través de TCP, también el servidor se debe ejecutar ANTES del cliente, obviamente.

PROGRAMACIÓN DE UN SERVIDOR TCP

Lo primero que vamos a hacer es crear un socket de Internet pero, en esta ocasión, no se solicita al kernel un servicio no fiable de datagramas, sino un servicio que proporcione un flujo (stream) de bytes fiable en el segundo argumento a la llamada a socket():

```
int sd = socket(PF_INET, SOCK_STREAM, 0);
/*comprueba que ha ido bien*/
```

Al igual que con UDP, poner el tercer parámetro a cero indica que se quiere utilizar aquí el protocolo por defecto que tenga el kernel, que en este caso es TCP.

Una vez que tengamos el descriptor asociado al socket, al que vamos a llamar a partir de ahora **socket de conexiones**, el servidor se tiene que realizar la asociación a un puerto, de forma obligatoria, puesto que los clientes van a intentar conectar con él en la IP del ordenador en el que se esté ejecutando y en el puerto vinculado.

Para esto, hacemos exactamente las mismas operaciones que en UDP: rellenar una variable de tipo struct sockaddr_in con el puerto con el que queremos vincular al socket de conexiones, y luego llamar a bind():

```
1  struct sockaddr_in vinculo;
2  memset(&vinculo, 0, sizeof(vinculo));
3  vinculo.sin_family      = AF_INET;
4  vinculo.sin_port        = htons(5050); //puerto en big endian
5  vinculo.sin_addr.s_addr = INADDR_ANY;
6
7  int resultado = bind(sd, (struct sockaddr *)&vinculo, sizeof(vinculo));
8  if(resultado < 0){
9    perror("Error en bind");
10   exit(1);
11 }
```

Siempre es buen momento para recordar que el puerto es una variable entera sin signo que ocupa dos bytes, así que al viajar al otro extremo, debes ponerla siempre en formato de red (big endian) con esa pedazo de función **htons()**.

¡Ojo! ¡No te equivoques! No utilices nunca un htonl() para pasar a formato de red un puerto, porque entonces se estarían dando la vuelta a cuatro bytes en vez de a dos y se armaría un lío. No es que no compile, es que no te has vinculado al 5050 y los clientes que intentan llegar hasta el servidor por ese puerto no se van a poder conectar.

A continuación, hay que decirle al sistema operativo que este socket va a ser el que actúe como servidor. Para ello, lo pones en modo TCP pasivo, es decir, que a partir de ahora este socket va a esperarse a recibir solicitudes de conexiones de clientes de forma pasiva (los clientes usan su socket de forma activa, iniciando ellos la conexión). Para indicar que el socket va a pasar a actuar como servidor, se utiliza la llamada al sistema listen() (man 2 listen):

```
12 int resultado = listen(sd, 10);
13 /*comprueba que ha ido bien*/
```

Si hay algún error, listen() devuelve -1. Este error se produce, por ejemplo, si el socket se había creado como UDP con SOCK_DGRAM, y el sistema operativo no puede poner en modo pasivo un socket UDP (¡es que no se esperan conexiones en UDP!).

El segundo parámetro indica el número de clientes que pueden estar como máximo en cola esperando a que se les acepte una conexión. Efectivamente, puede haber varios clientes intentando acceder al servicio, y su solicitud de conexión se encola hasta que el servidor la acepte. Si un cliente, al intentar conectarse, se encuentra con la cola de espera del servidor llena, entonces directamente aborta el intento (lo veremos en el código del cliente). ¿Y qué valor ponemos nosotros? Pues el que quieras. Normalmente, los kernel modernos de Linux aceptan hasta un máximo de 4096 clientes posibles en cola (si tú pones un valor mayor, simplemente se ignora y se considera ese máximo). Lo único que aquí podría tenerse en cuenta es que, a mayor tamaño permitido de cola, más probabilidad de que un cliente espere más a ser atendido si le toca estar al final de esa cola. A lo mejor, es preferible que se encuentre una cola pequeña llena y aborte el intento rápidamente, dejando en manos del programador si se reintenta de nuevo la conexión, o no. En cualquier caso, para nuestra asignatura no es algo importante.

Y, por fin, llegamos a la llamada al sistema encargada de bloquear al proceso hasta que aparezca un cliente solicitando la conexión. Esta llamada al sistema ES LA MÁS IMPORTANTE.

```
14  struct sockaddr_in dir_cliente;
15  socklen_t longitud_dir = sizeof(dir_cliente);
16  int csd = accept(sd, (struct sockaddr *)&dir_cliente, &longitud_dir);
17  if(csd < 0) {
18    perror("error en accept");
19    exit(1);
20  }
```

Si accept() tiene algún fallo, o llega una señal y lo interrumpe (en modo System V), devuelve -1. Si todo va bien, significa que está conectado con un cliente, identificado por su IP y el puerto, que se devuelven por referencia en la dirección de socket que se pasó como segundo argumento. El tercer argumento es el tamaño en bytes de dicha dirección, y tiene que estar inicializado antes de llamar a accept(). Por supuesto, vuelvo a recordar que estas líneas de código son propensas a COPIAR Y PEGAR, y no se trata de memorizar nada, solo de entenderlo.

Aquí es donde tienes que prestar especial atención:

Si todo ha ido bien, **accept() devuelve un nuevo descriptor de socket**, distinto al socket que hasta ahora ha servido para aceptar conexiones con un cliente. Ese lo seguiremos manteniendo, y luego volveremos a él.

En el ejemplo, *csd* es el nuevo descriptor de socket que da acceso a las dos tuberías donde se encolan los bytes a transmitir y recibidos del cliente conectado (recuerda que así se permite la comunicación BIDIRECCIONAL). Para diferenciarlo del socket de conexión, lo vamos a llamar **socket de datos**, porque es el único que sirve para hacer lecturas y escrituras.

De hecho, si intentas hacer un write() o un read() sobre el socket de conexiones, las llamadas al sistema van a fallar ¡ahí no están las tuberías! Realmente, es un fallo de concepto bastante grave.

A partir de ahora, ya estamos preparados para leer y escribir del socket de datos. Depende de la especificación del sistema que estemos programando, el servidor será el primero en mandarle bytes al cliente, o quizá sea el cliente el que envíe primero, y el servidor los reciba esperando en su read() correspondiente.

Imagina que queremos implementar un servicio de eco. En este caso, el servidor espera un mensaje del cliente y, simplemente, lo retransmite.

```
21  int leidos;
22  char buffer[2048];
23  do{
24    leidos = read(csd, buffer, 2048);
25    if(leidos > 0)
26      int escritos = writen(csd, buffer, leidos);
27      /*comprueba que todo ha ido bien*/
28  }while(leidos > 0);
```

¿Reconoces este código? ¡Es el bucle de la asignatura! Lees y envías de vuelta en bucle, mientras el canal esté abierto en el otro extremo (si lo cierran, read() se desbloquea y la variable *leidos* vale cero). Como los sockets son bidireccionales, aquí estamos usando el mismo descriptor para leer y para escribir (ya sabemos que, internamente, hay dos tuberías separadas para transmitir y recibir).

Las reglas de lectura y escritura de las tuberías se aplican aquí normalmente. Si la tubería del kernel donde están los datos recibidos del cliente está vacía, read() es bloqueante.

Si la tubería donde se almacenan los bytes a transmitir por el sistema operativo está llena, write() es bloqueante. También se puede dar el caso de que no se consigan escribir todos los bytes que se necesitan, así que me he permitido reemplazar write() por nuestro milagroso writen().

Llegados a este punto te habrás dado cuenta de la diferencia fundamental de un envío UDP (con sendto()) y este envío con write()/writen(). ¡Aquí no tenemos que poner la IP y el puerto del destinatario al que van a enviarse los datos! Efectivamente, como se estableció una conexión, siempre se va a estar dialogando con el mismo cliente hasta el final.

Por fin, cuando terminemos de dar el servicio con el cliente, hacemos un cierre del socket de datos, liberando los todos recursos del sistema operativo y cerrando la conexión TCP con ese cliente:

```
close(csd);
```

Fíjate que no he cerrado el socket de conexiones, solo el de datos. Dejo abierto el de conexiones si voy a volver en bucle a aceptar una nueva conexión con otro cliente, y así mi servidor puede presumir de dar un servicio permanente e ininterrumpido. Esto es lo normal en software de comunicaciones.

Pero, atiende:

Antes de terminar el servidor, da igual si lo haces con Ctrl-C, acuérdate de cerrar el socket de conexiones:

```
close(sd);
```

Si se te olvida y no lo haces, te puede pasar que, si intentas ejecutar de nuevo el servidor, **te falle la llamada al sistema bind()**. El error habitual que sale al usar perror() es (lo digo en español) que EL PUERTO YA ESTÁ SIENDO USADO. Está claro que el fallo es por culpa de intentar

hacer la vinculación del socket con el puerto, porque ahora dice que está siendo usado ¿pero por quién? Si la ejecución de nuestro servidor anterior ya había terminado, ¡ya tendría que estar libre de nuevo!

Resulta que, si no se hizo un close(sd) explícito en nuestro programa, el sistema operativo Unix/Linux suele mantener la conexión abierta dos minutos más. Una solución, en las prácticas, es cambiar el puerto sobre el que se hace el bind() y seleccionar otro libre, si no se quiere esperar todo ese tiempo para volver a ejecutar el servidor. En el ejemplo, se podría usar el puerto 5051, por decir alguno. Y el cliente se debe conectar a ese nuevo puerto, obviamente.

ARQUITECTURAS DE SERVIDORES

En esta sección, te presento los tres diseños habituales de servidor según la forma en que se atiende a los clientes:

SERVIDORES ITERATIVOS. Es el diseño básico, que permite dar un servicio completo a un cliente antes de aceptar una nueva conexión:

```
/*arquitectura de un servidor iterativo*/
while(!fin){
  int csd = accept(sd, ...);
  /* read/write usando csd hasta dar el servicio completo */
  close(csd);
}
/*fin puede ponerlo a 1 una manejadora de señal, por ejemplo*/
close(sd);
```

Sin embargo, como sucede en cualquier comercio del mundo real, cuando hay que esperar mucho en cola, los clientes suelen tener una mala experiencia. Es más desesperante aun cuando un cliente requiere de un servicio considerado de larga duración (como la transferencia de un fichero o ficheros grandes) y los demás están esperando.

SERVIDORES CONCURRENTES. Por fortuna para nosotros, existe una solución a este problema: podemos generar un hijo que atienda a cada cliente por separado, de tal manera que su experiencia mejora al minimizar el tiempo que se tarda en que se acepten las conexiones. Mira este esquema:

```
/*arquitectura de un servidor concurrente*/
signal(SIGCHLD, SIG_IGN); /*ya no hace falta hacer wait(0)*/
while(!fin){
  int csd = accept(sd, ...);
  pid_t  pid = fork();
  /*chequea que todo ha ido bien*/
  if(pid ==0){ /*zona exclusiva del hijo*/
    /* read/write usando csd hasta dar el servicio completo */
    close(csd);
    exit(0);
  }
}
/*fin puede ponerlo a 1 una manejadora de señal, por ejemplo*/
close(sd);
```

Nota que **el bloqueo en accept() se debe hacer ANTES del fork()**. Es decir, una vez que se acepta la conexión con un cliente, el servicio completo lo realiza un nuevo hijo. Mientras tanto, de forma concurrente, el padre vuelve a esperar a un nuevo cliente, y así vuelta a empezar.

El ejemplo de arquitectura de servidor concurrente esbozado arriba no es muy realista. Un servidor no puede aceptar de forma concurrente decenas de miles de clientes simultáneos (que implican clonarse en miles de hijos para poder servirlos) sin consumir muchos recursos de la máquina y terminar degradando el funcionamiento de todos los programas y del propio sistema operativo. Lo más habitual es tener un máximo de clientes simultáneos: solo se aceptan nuevas conexiones mientras no se supere el umbral, que puede venir indicado justo al arrancar el servidor (en un archivo de configuración, como argumento al ejecutar el programa, etc.)

SERVIDORES QUE ATIENDEN A MÁS DE UN CLIENTE GRACIAS A UN REACTOR

Esta es otra variante, que vamos a poder practicar también en los ejercicios de laboratorio. Se trata de utilizar select() para estar atentos, a la vez, tanto del socket de conexiones como de sockets de datos para leer bytes de clientes.

Recuerda la idea detrás del reactor: todos los descriptores bloqueantes se le pasan al sistema operativo para que haga un sondeo hasta que alguno esté listo.

En el caso del descriptor de conexiones del servidor (después de hacer listen()), lo metemos en un fd_set para que select() nos avise cuando esté listo para lectura. Si esto sucede, entonces es que hay una conexión solicitada por un cliente, y se puede hacer accept() sin peligro de dejar bloqueado el proceso.

Normalmente, el nuevo descriptor de socket que devuelve accept(), el que permite realizar las recepciones y envios de datos con el cliente, ¡también se mete en el fd_set que se pasará al select() en la siguiente iteración! Te voy a poner los pasos necesarios para utilizar bien esta estrategia:

/*arquitectura de un servidor que usa select() -BOCETO-*/

1) Se hace socket(), bind() y listen(). De esta forma tenemos un descriptor de socket TCP, denominado *sd*, funcionando como servidor, y que puede aceptar conexiones de clientes
2) Se mete *sd* en un conjunto fd_set
3) Se declara una variable almacén para introducir más adelante descriptores de datos de clientes conectados. Este almacén puede ser un vector o una lista (en C++) o un simple array en C. Al principio, está vacío (más adelante te doy más detalles sobre esto).
4) Comienza el bucle del select(), al que se pasa el fd_set anterior para que el proceso se bloquee hasta que alguno de sus descriptores está listo para LECTURA
5) Al salir de select() de forma exitosa, se comprueba con FD_ISSET si *sd* está listo para lectura. En este caso se ejecuta:
    - int csd = accept(sd, ...);
    - inserta en el almacén el valor csd (que identifica el socket de datos con un nuevo cliente)
    - inserta csd en el fd_set que se le va a pasar a select() en la próxima iteración
6) Recorre cada elemento del almacén (con un for o un while, por ejemplo), chequeando con FD_ISSET si el socket de datos almacenado en cada casilla está listo para lectura
    - En caso afirmativo, se ejecuta read() sobre ese descriptor, se procesan los datos recibidos y se atiende al cliente
    - Si read() devuelve cero, es que el cliente se ha desconectado, así que se quita ese descriptor del almacén y también se quita del fd_set que va a volver a chequear el select en la siguiente iteración

7) Se vuelve al bucle (paso 4)

Todo esto hay que afianzarlo implementándolo en C, como no podría ser de otra forma. Te recomiendo que intentes montar un servidor de eco con select() . Tu almacén puede ser un array como éste (seguro que en primer curso hiciste ejercicios similares):

int clientes_conectados[50];

Puedes inicializarlo con todas las casillas a -1, por ejemplo. Si te encuentras ese valor en una casilla, significa que está libre (el -1 no es un descriptor de fichero válido). Cuando te toque insertar un csd en el array, lo recorres buscando una casilla libre, y lo metes ahí. En caso de que te encuentres el array lleno al intentar insertar, significa que has alcanzado tu máximo de clientes soportado, por lo que no te quedará otra que hacer directamente:

close(csd); //cierra la conexión con el cliente, NO se le va a poder atender

Y, por supuesto, ya no metas csd en el fd_set que se le pasa a select(), porque ya no es un descriptor válido.

Técnicamente, podría decirse que tanto un servidor implementado usando fork() como uno implementado usando select() son capaces de atender a más de un cliente simultáneamente. Pero, recuerda:

"En el servidor basado en select(), mientras que estás atendiendo a un cliente, NO puedes ni aceptar nuevas conexiones, ni atender a otros clientes. Por lo tanto, úsalo con cabeza, SOLO en situaciones donde el servicio que se le da al cliente es de cortísima duración (un eco, el chat que veremos en la práctica, darle el día y la hora al cliente que necesita sincronizar el reloj de su sistema operativo, etc.)".

PROGRAMACIÓN DE UN CLIENTE TCP

Ya solo nos falta lo más fácil. Además de la llamada a socket para la reserva de TCP:

```
int sd = socket(PF_INET, SOCK_STREAM, 0);
/*chequear que ha ido bien*/
```

Se necesita realizar la conexión con el servidor. Para ello, se rellena una dirección de socket con la IP y el puerto donde está vinculado el proceso servidor, y se llama a connect() (man 2 connect):

```
1  struct sockaddr_in dir_servidor;
2  memset(&dir_servidor, 0, sizeof(dir_servidor));
3  dir_servidor.sin_family     = AF_INET;
4  dir_servidor.sin_port       = htons(5050); //puerto en big endian
5  dir_servidor.sin_addr.s_addr = inet_addr("127.0.0.1");
6
7  int resultado = connect(sd, (struct sockaddr *)&dir_servidor, sizeof(dir_servidor));
8  if(resultado < 0){
9    perror("Error en connect");
10   exit(1);
11 }
```

En la línea 5 he indicado que la IP es la de pruebas del propio ordenador, útil siempre que el servidor se esté ejecutando en la misma máquina que el cliente.

La llamada al sistema connect() de vuelve cero si todo ha ido bien, y -1 si no se pudo conectar con el servidor (por ejemplo, si no está ejecutándose, antes del cliente, en la IP y puerto indicados).

Fíjate que connect() no devuelve ningún nuevo descriptor de socket. El descriptor original *sd*, tras la conexión, es el que permite hacer las lecturas y escrituras de bytes con el servidor. En el caso del ejemplo de eco:

```
12  char mensaje[] = "hola";
13  int escritos = writen(sd, mensaje, strlen(mensaje));
14  /*comprueba que todo ha ido bien*/
15
16  char buffer[2048];
17  int leidos = read(sd, buffer, 2048);
18  /*comprueba que todo ha ido bien*/
19  writen(1, buffer, leidos);
20  /*comprueba que todo ha ido bien*/
21  close(sd);
```

FORMAS ALTERNATIVAS DE ENVÍO Y RECEPCIÓN DE BYTES

Para sockets TCP, es válido utilizar el tradicional write() o una variante que se llama send() (man 2 send), que tiene los tres primeros parámetros iguales a write(), más un cuarto que permite especificar ciertas opciones de envío. En nuestra asignatura, no vamos a utilizar ninguna opción de envío, y dejamos a cero este cuarto argumento. De esta forma, las dos llamadas son totalmente equivalentes:

```
send(sd, mensaje, strlen(mensaje), 0);
write(sd, mensaje, strlen(mensaje));
```

De igual manera, existe una variante de read() para poder especificar ciertas opciones de recepción en TCP. Dicha variante se llama recv() (man 2 recv). Nosotros no vamos a utilizar ninguna opción de recepción, y dejamos a cero este cuarto argumento. De esta forma, las dos llamadas son totalmente equivalentes:

```
recv(sd, buffer, 2048, 0);
read(sd, buffer, 2048);
```

Mi consejo es que sigamos trabajando con write() y read(), y así aprovechamos nuestras implementaciones caseras de writen() y readn() para que funcionen con descriptores de ficheros, pipes, fifos y, ahora, también de sockets TCP.

CONSIDERACIONES FINALES

1) En un servidor, ¿para qué querríamos saber la dirección IP y el puerto del cliente que se ha conectado con nosotros y que se nos devuelve en accept(), por referencia? Ya hemos visto que no necesitamos esta información para poder enviarle datos al cliente.

   La respuesta es que esa información se puede utilizar para ver si el cliente está en una lista negra, o si tiene algún privilegio en el servicio que se le va a dar, entre otros posibles usos.

   Si tenemos servidores que atienden a más de un cliente, y ves que la IP y puerto de cada uno te hará falta más adelante en el programa, no hace falta crear un array o una lista para conseguir guardar esta información por cada cliente. Hay una llamada al sistema denominada getpeername() (man 2 getpeername), que se utiliza sobre un socket de datos, para poder obtener la información de IP y puerto del extremo (peer) de la conexión. Ejemplo:

   /*imagina que estamos en la zona exclusiva de un hijo que atiende a un cliente identificado por su socket de datos csd/
   struct sockaddr_in dir_cliente;
   socklen_t longitud_dir = sizeof(dir_cliente);
   int resultado = getpeername(csd, (struct sockaddr *)&dir_cliente, &longitud_dir);
   /*si ha habido exito, ahora dir_cliente tiene la IP y el puerto del cliente al que estamos conectados*/

2) Para terminar, es importante volver a recalcar las diferencias entre un cliente y servidor que use UDP o TCP. En caso de que tengas libertad absoluta en tu diseño, tendrás que seleccionar, de entre los dos, el protocolo de transporte que mejor se adapte a los requisitos de tu sistema:

   a. Si necesitas fiabilidad absoluta: utiliza TCP sin ninguna duda

   b. Si necesitas el protocolo más rápido, por encima de otros aspectos: puedes utilizar UDP, ya que no establece una conexión entre las dos partes, ni comprobaciones extra para evitar desorden de datos, pérdidas o duplicados. Todo esto agiliza la transferencia de información, claro, pero a costa de la fiabilidad. Eso sí, puede que tus aplicaciones tengan que incorporar algún mecanismo de chequeo extra, si lo necesitasen (es decir, que lo tendrías que programar tú).

Y esto es todo, amigos. Sabemos lo fundamental para crear programas clientes y servidores para Internet. A partir de ahora, con muy poco esfuerzo más, serías capaz de abrir cualquier documento donde venga especificado el protocolo que ha de seguir un determinado servicio en Internet e implementarlo en C[1]. Por ejemplo, podrías hacer un cliente HTTP básico para recibir páginas web de un servidor; un servidor básico HTTP para que Firefox o Chrome te pidan una página web a ti; un cliente de correo POP3 para recibir tus correos de un servidor...

[1] Esos documentos son públicos, se llaman RFCs (Request For Comments). Tanto para el cliente como para el servidor, te dicen cómo construir cada mensaje y el momento en el que se tiene que enviar al otro extremo. De igual forma, también deja claro cómo interpretar cada uno de los mensajes recibidos, etc. Si te interesa seguir indagando a partir de aquí, echa un vistazo a https://www.rfc-editor.org/