Redes de Computadores
Práctica 2: Interfaz *socket* y programación en red sobre TCP/IP

Natalia Ayuso Escuer · Juan Segarra Flor · Jesús Alastruey Benedé

1. Objetivos

Introducción al modelo cliente-servidor, abstracción *socket* y estructuras de direcciones. Programación de una aplicación sencilla TCP. Uso de *netcat* como herramienta de depuración.

2. El modelo cliente-servidor

El modelo más utilizado para el desarrollo de aplicaciones en red es el cliente-servidor:

a) El proceso servidor es un programa en ejecución que está a la espera de que algún cliente requiera sus servicios.

b) Un proceso cliente, en ejecución en el mismo o en otro computador de la red, envía una petición de servicio hacia el servidor.

c) El servidor recibe la petición, responde al cliente y queda de nuevo a la espera de nuevas peticiones.

Un ejemplo de aplicación que sigue este modelo el de cliente-servidor web. El servidor web espera que un cliente (navegador) le solicite una página web, como puede verse en la figura 1.

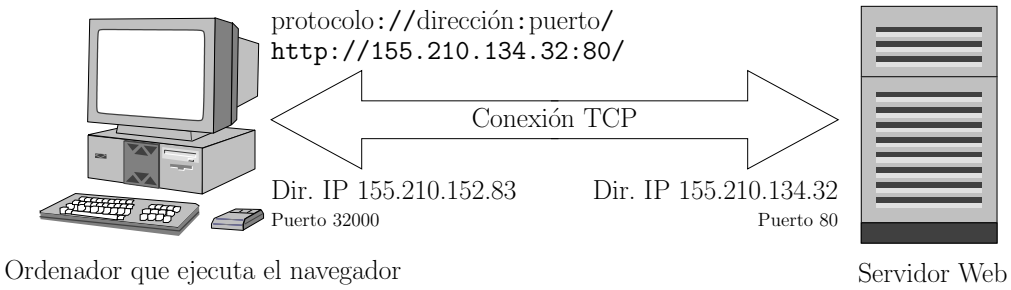

Figura 1: Conexión de un cliente a un servidor web

Para que la comunicación entre cliente y servidor sea posible, es necesaria la participación de una serie de protocolos de red. En nuestro caso nos vamos a centrar en la pila de protocolos TCP/IP. Por ejemplo, los clientes y servidores web utilizan el protolo TCP (*Transport Control Protocol*), que a su vez utiliza el protocolo IP (*Internet Protocol*).

Cliente y servidor son normalmente procesos de usuario, mientras que los protocolos de transporte y red están implementados en el kernel del sistema operativo.

El nivel de transporte se encarga de comunicar dos procesos concretos, cada uno en un extremo de la comunicación que es identificado mediante un número de puerto. En esta capa se sitúan los protocolos TCP (*Transport Control Protocol*) y UDP (*User Datagram Protocol*). Dependiendo del tipo de transmisión deseada, algunas aplicaciones usan TCP, UDP o ambos. En esta práctica trabajaremos con TCP. En la siguiente práctica implementaremos una aplicación cliente-servidor sobre UDP.

3. Introducción a los *sockets*

La implementación de clientes y servidores TCP se basa en una abstracción llamada *socket*. Un socket (enchufe) representa uno de los extremos de una conexión bidireccional entre dos procesos. Cuando lo necesita, un proceso pide al sistema operativo la creación de un socket. El sistema devuelve un «descriptor» que el proceso usará para referirse al nuevo socket. Dependiendo de las características deseadas para la comunicación (en general TCP o UDP) se solicitará el tipo de socket correspondiente. Los dos extremos de la comunicación deben tener *el mismo tipo de socket*.

Para poder crear un socket, hay que especificar tipo de comunicación, direcciones y número de puerto locales.

3.1. Tipos de socket

En la pila de protocolos TCP/IP en general se usan dos tipos de socket:

`SOCK_STREAM`: proporciona una transmisión bidireccional continua y fiable (los datos se reciben ordenados, sin errores, sin pérdidas y sin duplicados) de bytes *con conexión* mediante el protocolo TCP (*Transport Control Protocol*).

`SOCK_DGRAM`: proporciona una transmisión bidireccional no fiable, de longitud máxima prefijada, *sin conexión* mediante el protocolo UDP (*User Datagram Protocol*).

3.2. Direcciones de red

Una dirección de red identifica un interfaz de red de una máquina. En la familia de protocolos de Internet las direcciones IPv4 son de 32 bits, mientras que las IPv6 son de 128 bits.

3.3. Números de puerto

Los puertos se identifican por un número entero sin signo de 16 bits (rango de 0 a 65535). Los puertos 0 a 1023 están reservados para los servicios «bien conocidos» (*well-known ports*), y requieren privilegios para su uso. Por ejemplo, el puerto 80 está reservado para el servicio web (protocolo HTTP). En la práctica anterior ya viste que puedes consultar el puerto que ocupa cada servicio en el archivo `/etc/services`.

3.4. Cliente-servidor TCP

Para una comunicación *con* conexión (TCP), hay que seguir una serie de pasos, tal y como se muestra en la figura 2.

El servidor creará inicialmente un extremo de la conexión pidiendo un socket (`socket`) y asociándolo a una dirección local (`bind`). En TCP/IP una dirección local es la dirección IP de la máquina más un número de puerto sobre un protocolo de transporte (TCP o UDP). El servidor puede en algún momento recibir varias peticiones de conexión simultáneas por lo que se debe especificar el número máximo de conexiones en espera (`listen`). A continuación, si hay alguna conexión pendiente la atenderá (`accept`), en caso contrario, se quedará bloqueado a la espera de alguna conexión entrante.

Por su parte, el cliente también debe crear un socket. *Se desaconseja* el uso de `bind` en el cliente, puesto que el cliente puede usar cualquier puerto libre y no es necesario especificar uno concreto. Una vez creado el socket, lanzará una petición de conexión al servidor (`connect`). Si el servidor está disponible, es decir, si ha ejecutado `accept` y no hay peticiones anteriores en cola, inmediatamente se desbloquean tanto cliente como servidor. En la parte del servidor, `accept` habrá devuelto un nuevo identificador del socket que está conectado con el cliente. El identificador del socket original sigue sirviendo para atender nuevas peticiones de conexión a medida que se vayan realizando llamadas `accept`. Cliente y servidor se intercambiarán datos mediante `send` y `recv` pudiendo finalmente cerrar la conexión mediante la llamada `shutdown` o `close`. En los sistemas Unix/Linux también es posible usar las llamadas `read` y `write` para leer y escribir de un socket.

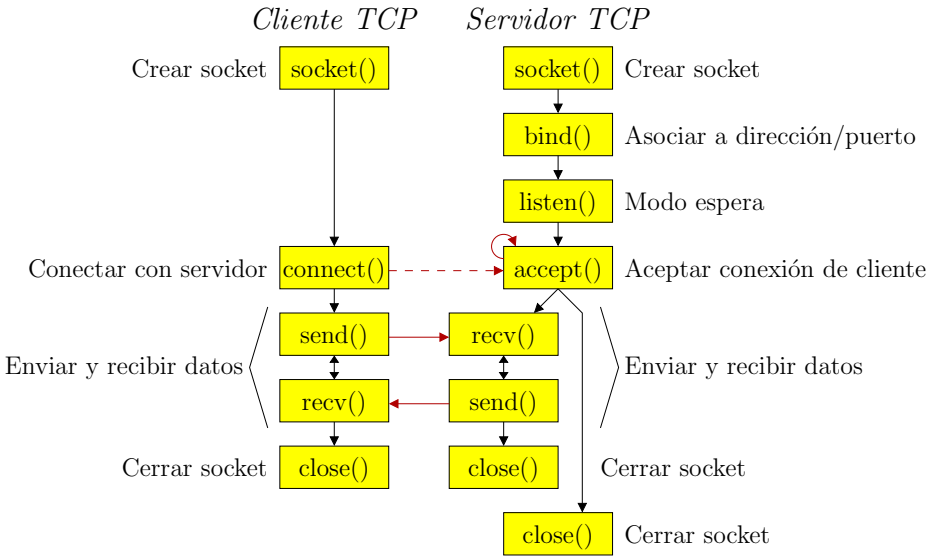

Figura 2: Llamadas al sistema para sockets en un protocolo orientado a conexión.

3.5. Implementación de un cliente TCP sencillo

Vamos a considerar un ejemplo para ilustrar algunos de los conceptos que se han presentado. La figura 3 muestra la implementación de un cliente TCP. Este cliente establece una conexión con el puerto especificado de un sistema identificado por su dirección IP y muestra por salida estándar la cadena de texto enviada por el servidor. A continuación se describen las acciones principales relacionadas con la interfaz socket.

Crear un socket TCP. En la línea 12, la función `socket` crea un socket de la familia Internet `AF_INET` de tipo `SOCK_STREAM`, es decir, un socket TCP. Esta función devuelve un número entero que se utilizará para identificar al socket en futuras llamadas a funciones.

Especificar dirección IP y puerto del servidor. En las líneas 17-19 se rellena la variable `servaddr`, que es una estructura de direcciones IPv4 (tipo `struct sockaddr_in`). En concreto, se especifica familia Internet `AF_INET`, y la dirección IP y puerto que se han pasado como parámetros por línea de comandos (`argv[1] y argv[2]`). La dirección IP y puerto en esta estructura deben almacenarse en un formato establecido, por eso se llama a las funciones `inet_pton` (*presentation to numeric*) y `htons` (*host to network short*) para realizar las conversiones apropiadas.

Establecer conexión con el servidor. En la línea 23, la función `connect` establece una conexión TCP con el proceso del servidor especificado como parámetro (`servaddr`). Esta función devuelve un número entero que se utilizará para identificar al socket en futuras llamadas a funciones.

Leer mensaje del servidor. En la línea 28, la función `read` recibe el mensaje del servidor.

Terminar el proceso. La función `exit` en la línea 34 finaliza la ejecución del proceso. Unix/Linux cierra los descriptores de fichero abiertos de un proceso que termina, por lo que el socket se cierra.

3.6. Implementación de un servidor TCP

La figura 4 muestra la implementación de un servidor TCP. Este servidor recibe una petición de conexión, la acepta y envía al cliente una cadena de texto donde indica el número de conexiones TCP que ha aceptado. A continuación se describen las acciones principales relacionadas con la interfaz socket.

```
1  int main(int argc, char **argv)
2  {
3      int sockfd, n;
4      char recvline[MAXLINE + 1];
5      struct sockaddr_in  servaddr = { 0 };
6
7      if (argc != 3)
8      {
9          printf("usage: %s <IPaddress> <puerto>\n", argv[0]);
10         exit(0);
11     }
12     if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
13     {
14         printf("socket error\n");
15         exit(0);
16     }
17     servaddr.sin_family = AF_INET;
18     servaddr.sin_port   = htons(atoi(argv[2]));
19     if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0) {
20         printf("inet_pton error for %s", argv[1]);
21         exit(0);
22     }
23     if (connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0)
24     {
25         printf("connect error");
26         exit(0);
27     }
28     n = read(sockfd, recvline, MAXLINE);
29     if (n > 0)
30     {
31         recvline[n] = 0;     /* null terminate */
32         printf("%s\n", recvline);
33     }
34     exit(0);
35 }
```

Figura 3: Cliente TCP (cabeceras y definición de constantes omitidas).

Crear un socket TCP. El proceso de creación del socket en la línea 13 es idéntico al realizado por el cliente.

Asociar puerto a socket. En las líneas 14-17 se asocia el socket creado al puerto local que va a prestar el servicio. Para ello, primero se rellena la variable `servaddr` (estructura de direcciones IPv4, de tipo `struct sockaddr_in`). En concreto, se especifica familia Internet `AF_INET`, la dirección IP por la que se admitirán conexiones (`INADDR_ANY`, cualquiera de los interfaces de red) y el puerto local, pasado como parámetro desde la línea de comandos (`argv[1]`).

Convertir socket en socket a la escucha. En la línea 18 la función `listen` convierte el socket en un socket a la escucha, es decir, ya preparado para aceptar conexiones entrantes. El segundo parámetro (`LISTENQ`) especifica el máximo número de conexiones pendientes que el kernel puede encolar.

Aceptar conexión del cliente. Normalmente, un proceso servidor pasa a la cola de bloqueados tras la llamada a la función `accept` (línea 21), esperando a que llegue una petición de conexión de un cliente.

```
1  int main(int argc, char **argv)
2  {
3      int listenfd, connfd;
4      struct sockaddr_in  servaddr =  { 0 };
5      char buff[MAXLINE];
6      int ncons = 0;
7
8      if (argc != 2)
9      {
10         printf("usage: %s <puerto>\n", argv[0]);
11         exit(0);
12     }
13     listenfd = socket(AF_INET, SOCK_STREAM, 0);
14     servaddr.sin_family      = AF_INET;
15     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
16     servaddr.sin_port        = htons(atoi(argv[1]));
17     bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
18     listen(listenfd, LISTENQ);
19     for ( ; ; )
20     {
21         connfd = accept(listenfd, (struct sockaddr *) NULL, NULL);
22         ncons++;
23         snprintf(buff, sizeof(buff), "Número de conexiones aceptadas por este
               servidor: %d\n", ncons);
24         write(connfd, buff, strlen(buff));
25         close(connfd);
26     }
27 }
```

Figura 4: Servidor TCP (cabeceras y definición de constantes omitidas).

Cuando se acepta dicha petición, `accept` devuelve un nuevo descriptor de fichero que se usará para la comunicación con el cliente. Se crea un nuevo descriptor para cada cliente que se conecta al servidor.

Enviar mensaje al cliente. En la línea 24, se envía el mensaje al cliente mediante la función `write`.

Cerrar la conexión. La función `close` en la línea 25 cierra la conexión con el cliente.

3.7. Ejecución de cliente y servidor TCP

Descarga los códigos, disponibles en *Moodle*. Se pueden descomprimir y desempaquetar con las siguientes órdenes[1]:

```
$ gzip -d p2.tar.gz
$ tar xvf p2.tar
```

Compila cliente y servidor:

```
$ gcc -o tcpsrv tcpsrv.c
$ gcc -o tcpcli tcpcli.c
```

A continuación, lanza el servidor en un puerto mayor que el 1024:

```
$ ./tcpsrv 12345
```

Y lanza el cliente, indicando la dirección local y el puerto donde escucha el servidor:

```
$ ./tcpcli 127.0.0.1 12345
```

[1]El símbolo dolar representa la secuencia de caracteres que muestra el intérprete de comandos (shell) para indicar que está a la espera de órdenes.

Puedes también probar a conectarte a servidores de tus compañeros:

```
$ ./tcpcli 155.210.154.x 12345
```

4. Estructuras de direcciones

La implementación de aplicaciones independientes del protocolo no es trivial, ya que cada protocolo tiene un formato de direcciones distinto. Por ejemplo, las direcciones IPv4 son de 32 bits, mientras que las IPv6 son de 128 bits. A continuación se describen las principales estructuras de datos del API de sockets. Tienes más detalles en el capítulo 3 de la *Guía de programación en red utilizando sockets*[2] disponible en Moodle. Su conocimiento permitirá desarrollar aplicaciones que funcionen con distintos protocolos.

```
1  struct addrinfo {
2      int              ai_flags;      // AI_PASSIVE, AI_CANONNAME, etc.
3      int              ai_family;     // AF_INET, AF_INET6, AF_UNSPEC
4      int              ai_socktype;   // SOCK_STREAM, SOCK_DGRAM
5      int              ai_protocol;   // use 0 for "any"
6      size_t           ai_addrlen;    // size of ai_addr in bytes
7      struct sockaddr  *ai_addr;      // struct sockaddr_in or _in6
8      char             *ai_canonname; // full canonical hostname
9      struct addrinfo  *ai_next;      // linked list, next node
10 };
```

Figura 5: Estructura de (lista de) direcciones `addrinfo`

La estructura `addrinfo` (figura 5) contiene una lista de direcciones. Cada una de estas direcciones, que puede corresponder a un protocolo distinto y por tanto tener detalles específicos, se almacena en el campo `ai_addr`. Es decir, el campo `ai_addr`, que formalmente es del tipo genérico `struct sockaddr` será en realidad de un tipo específico dependiendo de la familia de direcciones que corresponda. Las dos familias de direcciones que vamos a estudiar son las de Internet: `struct sockaddr_in` para IPv4 (figura 6, direcciones de 32 bits) y `struct sockaddr_in6` para IPv6 (figura 7, direcciones de 128 bits).

```
1  struct sockaddr_in {
2      short int          sin_family;    // Address family, AF_INET
3      unsigned short int sin_port;      // Port number
4      struct in_addr     sin_addr;      // Internet address
5      unsigned char      sin_zero[8];   // Same size as struct sockaddr
6  };
```

Figura 6: Estructura de dirección `sockaddr_in` (IPv4)

```
1  struct sockaddr_in6 {
2      u_int16_t       sin6_family;   // address family, AF_INET6
3      u_int16_t       sin6_port;     // port number, Network Byte Order
4      u_int32_t       sin6_flowinfo; // IPv6 flow information
5      struct in6_addr sin6_addr;     // IPv6 address
6      u_int32_t       sin6_scope_id; // Scope ID
7  };
```

Figura 7: Estructura de dirección `sockaddr_in6` (IPv6)

Como en general no se conoce la familia de direcciones que usa el equipo remoto, todas las estructuras de direcciones usan los primeros 16 bits para indicar la familia a la que pertenecen. Así, se puede leer esa

[2] Beej's Guide to Network Programming Using Internet Sockets

información independientemente del tipo de dirección y después interpretar el resto de la estructura de acuerdo a la familia de direcciones indicada. Como la estructura genérica `struct sockaddr` puede estar definida con un tamaño en el que no quepa una dirección IPv6, existe también la estructura genérica `sockaddr_storage`, que cumple la misma función pero con un tamaño mayor. En el caso de `struct sockaddr_storage`, los primeros 16 bits corresponden a un campo llamado `ss_family`. Así, se suele declarar una estructura `sockaddr_storage` y después usarla mediante interpretación explícita de tipos (*type casting*) como la estructura que interese.

4.1. Implementación de `migetaddrinfo()`

Para facilitar la tarea de construir la estructura de direcciones se usa la función `getaddrinfo`. A esta función se le pasan como parámetros el nombre o dirección IP y el servicio o número de puerto de un equipo, y proporciona la estructura de direcciones con la información necesaria para crear un socket. Además, en la llamada se especifica con tanto detalle como se desee el tipo de dirección que se desee obtener.

En esta parte de la práctica hay que completar el código del apartado 7.1 (`migetaddrinfo.c`) y 7.2 (`comun.c`), que realiza una llamada a `getaddrinfo` e imprime la estructura de direcciones obtenida. Te será muy útil la información sobre la función `getaddrinfo`, que puedes encontrar en el manual (`man getaddrinfo`) y en la sección 5.1 de la guía de programación en red utilizando sockets, disponible en Moodle.

Para completar el código, se recomienda seguir los siguientes pasos:

a) Completa los huecos numerados del código en el apartado 7.1

b) Descarga el código de Moodle y realiza los cambios anteriores

c) Compila el código en un equipo del laboratorio y corrige posibles errores y avisos (*warnings*) de compilación

En una máquina x86, puedes compilar el código con `make`:

```
$ make migetaddrinfo
```

o manualmente:

```
$ gcc -Wall -Wextra -o migetaddrinfo migetaddrinfo.c comun.c
```

La opción de compilación utilizada (`-Wall`, *warnings: all*) es recomendable para que muestre todos los avisos, incluso los más triviales.

En `hendrix`, la orden para compilar con el `Makefile` es:

```
$ gmake a=1 migetaddrinfo
```

y la opción manual:

```
$ gcc -Wall -o migetaddrinfo migetaddrinfo.c comun.c -lsocket -lnsl
```

En ambos casos se especifica al compilador que vamos a usar las bibliotecas `socket` y `nsl`.

La compilación con `make` guarda los binarios en el directorio `bin`.

Una vez el programa funcione correctamente, contesta a las siguientes preguntas:

1. ¿Qué *flag* hay que especificar en las pistas (*hints*) de la llamada a *getaddrinfo()* cuando vamos a solicitar una estructura de direcciones para lanzar un servidor?

2. Lanza `./migetaddrinfo www.unizar.es 80` ¿Cuál es su dirección IP? Verifica que la salida del programa muestra el puerto 80 en formato local.

3. Desde tu equipo local, ejecuta `./migetaddrinfo moodle.unizar.es https` ¿Cuál es su dirección IP? ¿Coincide el servicio https con el número de puerto que aparece en `/etc/services`?

4. ¿Qué ocurre al lanzar el programa especificando la dirección IP anterior y el número de puerto anterior?

5. Ejecuta ahora desde `hendrix`: `migetaddrinfo moodle.unizar.es https` ¿Qué sucede? ¿Está definido el servicio https en `hendrix` en `/etc/services`?

6. Lanza `./migetaddrinfo www.v6.facebook.com http` ¿En qué se diferencia la respuesta con respecto a los casos anteriores?

7. Ejecuta ahora `./migetaddrinfo hendrix-ssh.cps.unizar.es ssh` ¿Qué puedes deducir?

8. Lanza ahora `./migetaddrinfo www.google.com http` ¿Qué puedes deducir?

9. Ejecuta ahora `./migetaddrinfo http` y observa que la dirección obtenida no es válida. En la versión en inglés de la Wikipedia[3] aparecen 5 posibles usos para esta dirección. ¿Cuál de ellos corresponde a este caso?

4.2. Preguntas de comprensión

Responde a las siguientes preguntas. No olvides que puedes consultar cualquier detalle de las llamadas en el manual (e.g. `man socket`).

10. Observa los parámetros que necesita la llamada `socket` e indica a qué campos del `struct addrinfo` corresponden.

11. Observa los parámetros que necesita la llamada `connect` e indica a qué campos del `struct addrinfo` corresponden.

12. ¿Es necesario que el servidor esté bloqueado esperando conexión de un cliente para poder crear el socket con la llamada `socket` en el cliente? ¿Y para iniciar la conexión con la llamada `connect`?

13. La llamada `bind` asocia el socket con un puerto y es necesaria en el servidor. ¿Por qué?

14. En el servidor, la llamada `accept` devuelve un descriptor de socket, con lo que en ese punto del programa disponemos de dos descriptores de socket: el devuelto por `accept` y el devuelto por `socket`. ¿Cuál de los dos utilizaremos en las llamadas `send` y `recv` del servidor?

15. Teniendo en cuenta los dos descriptores de socket anteriores, ¿cuál es la diferencia entre usar la función `close` con cada uno de ellos?

4.3. Herramienta `netcat`

Como ya vimos en la práctica anterior, Netcat es una herramienta para conectar transmisiones con la entrada/salida estándar. Es decir, lo que se introduce por la entrada estándar (teclado) es transmitido hacia donde se le indique, y lo que recibe se muestra en la salida estándar (pantalla). El comando puede funcionar como cliente y como servidor. Como servidor se quedará escuchando (*listen*) (`-l`) a la espera de conexiones entrantes en el puerto especificado (`-p numpuerto`), y como cliente iniciará una conexión con el servidor y puerto que le especifiquemos como parámetro. Por defecto, netcat realiza conexiones mediante el protocolo de la capa de transporte TCP, pero se le puede indicar que en su lugar use el protocolo de transporte UDP con `-u`. Hay varios detalles importantes a tener en cuenta. El primero de ellos es que, tanto en hendrix como en los equipos del laboratorio, la herramienta que vamos a usar se lanza con el comando `netcat` y *no* con el comando `nc` (en ciertos mensajes de ayuda se muestra incorrectamente que el comando es `nc`). En segundo lugar, para la práctica es recomendable usar siempre el parámetro `-v`, que hará que el comando muestre información adicional. Para obtener información más completa, lanza `netcat -h`.

16. Lanza el netcat en el equipo de prácticas como servidor en el puerto 32005 (`netcat -l -p 32005 -v`) y a continuación lanza en hendrix el netcat en modo cliente para que se conecte al servidor netcat de tu equipo (`netcat -v <direccionIP> 32005`). Recuerda que en una pregunta anterior has obtenido la dirección IP de tu equipo. Una vez esté establecida la conexión, escribe algo en cualquiera de los dos lados. ¿Qué sucede?

[3] `http://en.wikipedia.org/wiki/0.0.0.0`

Como hemos visto en el fichero `/etc/services`, cada servicio tiene asociado un número de puerto por defecto. Ciertos servicios requieren que el usuario se identifique, como por ejemplo el servicio *imap3*. Si ese puerto no está ocupado, en principio cualquier usuario podría lanzar el netcat en ese puerto, y con ello vería las contraseñas de cualquier otro usuario que intentara conectarse a ese servicio. Para evitar este problema, los puertos menores que 1024 están restringidos, y solo el administrador puede asociar procesos a esos puertos.

17. Prueba a lanzar el netcat como servidor en un puerto menor que 1024. ¿Qué error da?

18. En la misma máquina, lanza un servidor netcat que use TCP en cierto puerto (mayor que 1024) y al mismo tiempo (desde otra ventana) lanza otro servidor que también use TCP en el mismo puerto. No olvides poner la opción `-v`. ¿Es posible o da error?

19. Realiza el experimento anterior, pero usando TCP en uno de los servidores netcat y UDP en el otro. ¿Es posible o da error?

20. Teniendo en cuenta los resultados anteriores, ¿puede haber dos servidores usando el mismo protocolo de transporte (TCP o UDP) y el mismo puerto en la misma máquina (asumiendo que tiene una única dirección IP)? ¿Y si usan el mismo puerto pero uno usando TCP y otro usando UDP?

4.4. Implementación de programas cliente/servidor

En las secciones 7.3 y 7.4 se presenta una pareja de programas cliente/servidor *incompletos*, también disponibles en *Moodle*. El cliente lee la entrada estándar y se la envía al servidor. Éste cuenta el número de vocales y devuelve el resultado al cliente, que lo muestra por pantalla. Ten en cuenta que el carácter «fin de fichero» está asociado a la combinación de teclas Ctrl + d . Es decir, cuando se pulsa Ctrl + d se cierra el fichero de entrada y finaliza el envío de datos por parte del cliente.

En este apartado hay que completar los programas para que funcionen correctamente. Para ello, se propone seguir los siguientes pasos, empezando por el *cliente*:

a) Completa de forma esquemática los huecos numerados del código en la sección 7.3

b) Traslada el resultado anterior al código fuente

c) Compila el código y corrige los posibles errores *y avisos* de compilación

d) Verifica *parte* del funcionamiento con netcat: Lanza netcat como servidor en un puerto/servicio (e.g. 32000) de tu máquina local y a continuación lanza el cliente en el mismo equipo con los parámetros correspondientes. Lo que escribas en el cliente debería aparecer en el netcat, pero no podrás verificar la respuesta con el número de vocales.

e) Realiza otra vez la verificación anterior, pero ahora lanzando el cliente en *hendrix*. Recuerda que tendrás que recompilar el código en hendrix para generar un ejecutable para dicha máquina:
`gcc -Wall -o clientevocalesTCPhendrix clientevocalesTCP.c comun.c -lsocket -lnsl`

A continuación, haz lo mismo para el *servidor*:

a) Completa de forma esquemática los huecos numerados del código en la sección 7.4

b) Traslada el resultado anterior al código fuente

c) Compila el código y corrige los posibles errores *y avisos* de compilación

d) Verifica el funcionamiento lanzando el servidor en un puerto/servicio (e.g. 32000) de tu máquina local y a continuación el cliente en el mismo equipo con los parámetros correspondientes

e) Realiza otra verificación lanzando el servidor en el equipo local y el cliente en hendrix

f) Realiza otra verificación lanzando el cliente en el equipo local y el servidor en hendrix. Recuerda lo observado en la pregunta 7. Para compilar en hendrix:
`gcc -Wall -o servidorvocalesTCPhendrix servidorvocalesTCP.c comun.c -lsocket -lnsl`

Una de las capas vistas en clase de teoría ha sido la *capa de presentación* de datos. Esta capa se encarga de homogeneizar los datos transmitidos entre distintos equipos. Como esta capa no existe en la arquitectura TCP/IP, este trabajo ha de hacerlo la aplicación. Ya hemos visto ejemplos de ello en las funciones tipo *ntohl*, pero hay que verificar que cualquier dato sea interpretado correctamente. Por ejemplo, tanto en los equipos del laboratorio como en Hendrix, el texto se codifica mediante UTF-8.

21. Prueba a enviar desde el cliente al servidor de contar vocales los siguientes caracteres, cada uno en una línea distinta: línea sin ningún carácter, «a», «ñ», «€». ¿Cuántos bytes ocupa cada envío?

5. Evaluación de la práctica

El trabajo realizado en esta práctica forma parte de la nota de prácticas de laboratorio de la asignatura. La entrega se realizará vía *Moodle* mediante un cuestionario donde se rellenarán los huecos de los códigos proporcionados. Es muy recomendable que antes de realizar el cuestionario te asegures de que tus códigos compilan correctamente (sin *warnings*) y funcionan como deberían. En la siguiente sesión de prácticas *necesitarás usar estos códigos*. ¡Ten en cuenta la *fecha límite* del cuestionario!

6. ¿Sabías que...?

- La correspondencia puertos-protocolos se puede consultar en: `http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers`.
- Un API (Application Program Interface) es el conjunto de funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción. El interfaz de programación de aplicaciones de red original fue desarrollado para UNIX BSD (Universidad de Berkely). Para GNU/linux se denomina API de Sockets BSD o Sockets de Berkeley. Esta interfaz también se ha portado a Windows bajo el nombre Windows Sockets, abreviado como WinSock.
- TCP es uno de los protocolos fundamentales en Internet. Fue creado entre los años 1973 y 1974 por Vinton Cerf y Robert Kahn. Vinton Cerf fue investido Doctor Honoris Causa por la Universidad de Zaragoza en 2008 (ver detalles).

7. Códigos fuente a utilizar

Para esta práctica y las posteriores necesitarás los siguientes códigos, disponibles en *Moodle*. Una vez descargados, se pueden descomprimir y desempaquetar con las siguientes órdenes:

```
$ gunzip p2.tar.gz
$ tar xvf p2.tar
```

7.1. Código *migetaddrinfo* (`migetaddrinfo.c`)

```
1  // el preprocesador sustituye cada include por contenido del fichero referenciado
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <sys/types.h>
7  #include <sys/socket.h>
8  #include <arpa/inet.h>
9  #include <netinet/in.h>
10 #include <netdb.h>
11
12 #include "comun.h"
13
```

```
14 /* argc indica el número de argumentos que se han usado en el la línea de
       comandos.
15  * argv es un vector de cadenas de caracteres.
16  * argv[0] apunta al nombre del programa y así sucesivamente*/
17 int main(int argc, char * argv[])
18 {
19     char f_verbose = 1;  // flag, 1: imprimir información por pantalla
20     struct addrinfo* direccion; // puntero (no inicializado!) a estructura de
           dirección
21
22     // verificación del número de parámetros:
23     if ((argc != 2) && (argc != 3))
24     {
25         printf("Número de parámetros incorrecto \n");
26         printf("Uso: %s [servidor] <puerto/servicio>\n", argv[0]);
27         exit(1); // finaliza el programa indicando salida incorrecta (1)
28     }
29
30     if (argc == 2)
31     {
32         // devuelve la estructura de dirección del servicio solicitado asumiendo
               que vamos a actuar como servidor
33         direccion = obtener_struct_direccion(NULL, argv[1], f_verbose);
34     }
35     else // if (argc == 3)
36     {
37         // devuelve la estructura de dirección al equipo y servicio solicitado
38         direccion = obtener_struct_direccion(argv[1], argv[2], f_verbose);
39     }
40
41     // cuando ya no se necesite, hay que liberar la memoria dinámica obtenida en
           getaddrinfo() mediante freeaddrinfo()
42     if (f_verbose)
43     {
44         printf("Devolviendo al sistema la memoria usada por servinfo (ya no se va
               a usar)... ");
45         fflush(stdout);
46     }
47     freeaddrinfo(direccion);
48     if (f_verbose) printf("hecho\n");
49     direccion = NULL; // como ya no tenemos la memoria, dejamos de apuntarla para
           evitar acceder a ella por error
50
51     // finaliza el programa indicando salida correcta (0)
52     exit(0);
53 }
```

7.2. Código *comun* (comun.c)

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <sys/types.h>
6  #include <sys/socket.h>
7  #include <arpa/inet.h>
8  #include <netinet/in.h>
9  #include <netdb.h>
10
```

```
11 /*************************** printsockaddr ****************************/
12 /*
13  * Imprime una estructura sockaddr_in o sockaddr_in6 almacenada en
        sockaddr_storage
14  */
15 void printsockaddr(struct sockaddr_storage * saddr)
16 {
17     struct sockaddr_in  *saddr_ipv4; // puntero a estructura de dirección IPv4
18     // el compilador interpretará lo apuntado como estructura de dirección IPv4
19     struct sockaddr_in6 *saddr_ipv6; // puntero a estructura de dirección IPv6
20     // el compilador interpretará lo apuntado como estructura de dirección IPv6
21     void *addr; // puntero a dirección. Como puede ser tipo IPv4 o IPv6 no
           queremos que el compilador la interprete de alguna forma particular, por
           eso void
22     char ipstr[INET6_ADDRSTRLEN]; // string para la dirección en formato texto
23     int port; // para almacenar el número de puerto al analizar estructura
           devuelta
24
25     if (saddr == NULL)
26     {
27         printf("La dirección está vacía\n");
28     }
29     else
30     {
31         printf("\tFamilia de direcciones: ");
32         fflush(stdout);
33         if (saddr->ss_family == AF_INET6)
34         {   //IPv6
35             printf("IPv6\n");
36             // apuntamos a la estructura con saddr_ipv6 (el cast evita el warning
                   ),
37             // así podemos acceder al resto de campos a través de este puntero
                   sin más casts
38             saddr_ipv6 = (struct sockaddr_in6 *)saddr;
39             // apuntamos a donde está realmente la dirección dentro de la
                   estructura
40             addr = &(saddr_ipv6->sin6_addr);
41             // obtenemos el puerto, pasando del formato de red al formato local
42             port = ntohs(saddr_ipv6->sin6_port);
43         }
44         else if (saddr->ss_family == AF_INET)
45         {   //IPv4
46             printf("IPv4\n");
47             saddr_ipv4 = [ 1 ];
48             addr = [ 2 ];
49             port = [ 3 ];
50         }
51         else
52         {
53             fprintf(stderr, "familia desconocida\n");
54             exit(1);
55         }
56         // convierte la dirección ip a string
57         inet_ntop(saddr->ss_family, addr, ipstr, sizeof ipstr);
58         printf("\tDirección (interpretada según familia): %s\n", ipstr);
59         printf("\tPuerto (formato local): %d\n", port);
60     }
61 }
62
63
64 /****************** obtener_struct_direccion ********************/
```

```
65  /*
66   * Función que devuelve una estructura de direcciones rellenada
67   * con al menos una dirección que cumpla los parámetros especificados.
68   * El último parámetro controla la verbosidad del programa (información detallada
         por salida estándar)
69   */
70  struct addrinfo*
71  obtener_struct_direccion(char *dir_servidor, char *servicio, char f_verbose)
72  {
73      struct addrinfo hints,      // variable para especificar la solicitud
74                      *servinfo; // puntero para respuesta de getaddrinfo()
75      struct addrinfo *direccion; // puntero para recorrer la lista de direcciones
            de servinfo
76      int status;       // finalización correcta o no de la llamada getaddrinfo()
77      int numdir = 1;  // contador de estructuras de direcciones en la lista de
            direcciones de servinfo
78
79      // sobreescribimos con ceros la estructura para borrar cualquier dato que
            pueda malinterpretarse
80      memset(&hints, 0, sizeof hints);
81
82      // genera una estructura de dirección con especificaciones de la solicitud
83      if (f_verbose)
84      {
85          printf("1 - Especificando detalles de la estructura de direcciones a
                solicitar... \n");
86          fflush(stdout);
87      }
88
89      hints.ai_family = [ 4 ]; // opciones: AF_UNSPEC; IPv4: AF_INET; IPv6:
            AF_INET6; etc.
90
91      if (f_verbose)
92      {
93          printf("\tFamilia de direcciones/protocolos: ");
94          switch (hints.ai_family)
95          {
96              case AF_UNSPEC: printf("IPv4 e IPv6\n"); break;
97              case AF_INET:   printf("IPv4)\n"); break;
98              case AF_INET6:  printf("IPv6)\n"); break;
99              default:        printf("No IP (%d)\n", hints.ai_family); break;
100         }
101         fflush(stdout);
102     }
103
104     hints.ai_socktype = [ 5 ]; // especificar tipo de socket
105
106     if (f_verbose)
107     {
108         printf("\tTipo de comunicación: ");
109         switch (hints.ai_socktype)
110         {
111             case SOCK_STREAM: printf("flujo (TCP)\n"); break;
112             case SOCK_DGRAM:  printf("datagrama (UDP)\n"); break;
113             default:          printf("no convencional (%d)\n", hints.ai_socktype)
                    ; break;
114         }
115         fflush(stdout);
116     }
117
```

```
118     // flags específicos dependiendo de si queremos la dirección como cliente o
            como servidor
119     if (dir_servidor != NULL)
120     {
121         // si hemos especificado dir_servidor, es que somos el cliente y vamos a
                conectarnos con dir_servidor
122         if (f_verbose) printf("\tNombre/dirección del equipo: %s\n", dir_servidor
                );
123     }
124     else
125     {
126         // si no hemos especificado, es que vamos a ser el servidor
127         if (f_verbose) printf("\tNombre/dirección: equipo local\n");
128         hints.ai_flags = [ 6 ]; // especificar flag para que la IP se rellene
                con lo necesario para hacer bind
129     }
130     if (f_verbose) printf("\tServicio/puerto: %s\n", servicio);
131
132     // llamada a getaddrinfo() para obtener la estructura de direcciones
            solicitada
133     // getaddrinfo() pide memoria dinámica al SO, la rellena con la estructura de
             direcciones,
134     // y escribe en servinfo la dirección donde se encuentra dicha estructura.
135     // La memoria *dinámica* reservada por una función NO se libera al salir de
            ella.
136     // Para liberar esta memoria, usar freeaddrinfo()
137     if (f_verbose)
138     {
139         printf("2 - Solicitando la estructura de direcciones con getaddrinfo()...
                ");
140         fflush(stdout);
141     }
142     status = getaddrinfo(dir_servidor, servicio, &hints, &servinfo);
143     if (status != 0)
144     {
145         fprintf(stderr,"Error en la llamada getaddrinfo(): %s\n", gai_strerror(
                status));
146         exit(1);
147     }
148     if (f_verbose) printf("hecho\n");
149
150     // imprime la estructura de direcciones devuelta por getaddrinfo()
151     if (f_verbose)
152     {
153         printf("3 - Analizando estructura de direcciones devuelta... \n");
154         direccion = servinfo;
155         while (direccion != NULL)
156         {   // bucle que recorre la lista de direcciones
157             printf("    Dirección %d:\n", numdir);
158             printsockaddr((struct sockaddr_storage*) direccion->ai_addr);
159             // "avanzamos" a la siguiente estructura de direccion
160             direccion = direccion->ai_next;
161             numdir++;
162         }
163     }
164
165     // devuelve la estructura de direcciones devuelta por getaddrinfo()
166     return servinfo;
167 }
```

7.3. Código *cliente* de contar vocales (clientevocalesTCP.c)

```
1  // importación de funciones, constantes, etc.
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <sys/types.h>
7  #include <sys/socket.h>
8  #include <arpa/inet.h>
9  #include <netinet/in.h>
10 #include <netdb.h>
11
12 #include "comun.h"
13
14 // definición de constantes
15 #define MAX_BUFF_SIZE 1000 // tamaño máximo del buffer
16
17 // función que crea la conexión y se conecta al servidor
18 static int initsocket(struct addrinfo *servinfo, char f_verbose)
19 {
20     int sock;
21
22     printf("\nSe usará ÚNICAMENTE la primera dirección de la estructura\n");
23
24     // crea un extremo de la comunicación y devuelve un descriptor
25     if (f_verbose)
26     {
27         printf("Creando el socket (socket)... ");
28         fflush(stdout);
29     }
30     sock = socket( 7 , 8 , 9 );
31     if (sock < 0)
32     {
33         perror("Error en la llamada socket: No se pudo crear el socket");
34         /* muestra por pantalla el valor de la cadena suministrada por el
               programador, dos puntos y un mensaje de error que detalla la causa del
               error cometido */
35         exit(1);
36     }
37     if (f_verbose) printf("hecho\n");
38
39     // inicia una conexión en el socket:
40     if (f_verbose)
41     {
42         printf("Estableciendo la comunicación a través del socket (connect)... ")
               ;
43         fflush(stdout);
44     }
45     if (connect( 10 , 11 , 12 ) < 0)
46     {
47         perror("Error en la llamada connect: No se pudo conectar con el destino")
               ;
48         exit(1);
49     }
50     if (f_verbose) printf("hecho\n");
51
52     return sock;
53 }
54
```

```
55 /*************************** MAIN ******************************/
56 int main(int argc, char * argv[])
57 {
58     // declaración de variables propias del programa principal (locales a main)
59     char f_verbose = 1;  // flag, 1: imprimir información extra
60     const char fin = 4; // carácter ASCII end of transmission (EOT) para indicar
           fin de transmisión
61     struct addrinfo * servinfo; // puntero a estructura de dirección destino
62     int sock; // descriptor del socket
63     char msg[MAX_BUFF_SIZE]; // buffer donde almacenar datos para enviar
64     ssize_t len, // tamaño de datos leídos por la entrada estándar (size_t con
           signo)
65             sentbytes; // tamaño de datos enviados (size_t con signo)
66     uint32_t num; // variable donde anotar el número de vocales
67
68     // verificación del número de parámetros:
69     if (argc != 3)
70     {
71         printf("Número de parámetros incorrecto \n");
72         printf("Uso: %s servidor puerto/servicio\n", argv[0]);
73         exit(1); // finaliza el programa indicando salida incorrecta (1)
74     }
75
76     // obtiene estructura de direccion
77     servinfo = obtener_struct_direccion(argv[1], argv[2], f_verbose);
78
79     // crea un extremo de la comunicación con la primera de las direcciones de
           servinfo e inicia la conexión con el servidor. Devuelve el descriptor del
           socket
80     sock = initsocket(servinfo, f_verbose);
81
82     // cuando ya no se necesite, hay que liberar la memoria dinámica usada para
           la dirección
83     freeaddrinfo(servinfo);
84     servinfo = NULL; // como ya no tenemos la memoria, dejamos de apuntarla para
           evitar acceder a ella por error
85
86     // bucle que lee texto del teclado y lo envía al servidor
87     printf("\nTeclea el texto a enviar y pulsa <Enter>, o termina con <Ctrl+d>\n"
           );
88     while ((len = read(0, msg, MAX_BUFF_SIZE)) > 0)
89     {
90         // read lee del teclado hasta que se pulsa INTRO, almacena lo leído en
               msg y devuelve la longitud en bytes de los datos leídos
91         if (f_verbose) printf("  Leídos %zd bytes\n", len);
92
93         // envía datos al socket
94         if ((sentbytes = send( 13 , 14 , 15 , 0)) < 0)
95         {
96             perror("Error de escritura en el socket");
97             exit(1);
98         }
99         else
100        {
101            if (f_verbose) printf("  Enviados correctamente %zd bytes \n",
                   sentbytes);
102        }
103        // en caso de que el socket sea cerrado por el servidor,
104        // al llamar a send() se genera una señal SIGPIPE,
105        // que como en este código no se captura,
106        // hace que finalice el programa SIN mensaje de error
```

```
107         // Las señales se estudian en la asignatura Sistemas Operativos
108
109         printf("Teclea el texto a enviar y pulsa <Enter>, o termina con <Ctrl+d>\
                n");
110     }
111
112     // se envía una marca de finalización:
113     if (send( 16 , 17 , 18 , 0) < 0)
114     {
115         perror("Error de escritura en el socket");
116         exit(1);
117     }
118     if (f_verbose)
119     {
120       printf("Enviada correctamente la marca de finalización.\nEsperando
              respuesta del servidor...");
121       fflush(stdout);
122     }
123
124     // recibe del servidor el número de vocales recibidas:
125     if (recv( 19 , 20 , 21 , 0) < 0)
126     {
127         perror("Error de lectura en el socket");
128         exit(1);
129     }
130     printf(" hecho\nEl texto enviado contenía en total %d vocales\n", 22 );
131     // convierte el entero largo sin signo desde el orden de bytes de la red al
          del host
132
133     close( 23 ); // cierra la conexión del socket:
134     if (f_verbose) printf("Socket cerrado\n");
135
136     exit(0); // finaliza el programa indicando salida correcta (0)
137 }
```

7.4. Código *servidor* de contar vocales (servidorvocalesTCP.c)

```
1  // importación de funciones, constantes, variables, etc.
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <sys/socket.h>
7  #include <arpa/inet.h>
8  #include <netdb.h>
9  #include <netinet/in.h>
10
11 #include "comun.h"
12
13 // definición de constantes
14 #define EOT 4               // carácter ASCII end of transmision
15 #define BUFF_SIZE 1000      // tamaño del buffer
16
17
18 // función que crea la conexión y espera conexiones entrantes
19 static int establecer_servicio(struct addrinfo *servinfo, char f_verbose)
20 {
21     int sock;
22
```

```
23      printf("\nSe usará ÚNICAMENTE la primera dirección de la estructura\n");
24
25      // crea un extremo de la comunicación y devuelve un descriptor
26      if (f_verbose)
27      {
28          printf("Creando el socket (socket)... ");
29          fflush(stdout);
30      }
31      sock = socket( 24 , 25 , 26 );
32      if (sock < 0)
33      {
34          perror("Error en la llamada socket: No se pudo crear el socket");
35          /* muestra por pantalla el valor de la cadena suministrada por el
                programador, dos puntos y un mensaje de error que detalla la causa del
                 error cometido */
36          exit(1);
37      }
38      if (f_verbose) printf("hecho\n");
39
40      // asocia el socket con un puerto
41      if (f_verbose)
42      {
43          printf("Asociando socket a puerto (bind)... ");
44          fflush(stdout);
45      }
46      if (bind( 27 , 28 , 29 ) < 0)
47      {
48          perror("Error asociando el socket");
49          exit(1);
50      }
51      if (f_verbose) printf("hecho\n");
52
53      // espera conexiones en un socket
54      if (f_verbose)
55      {
56          printf("Permitiendo conexiones entrantes (listen)... ");
57          fflush(stdout);
58      }
59      listen( 30 , 5); // 5 es el número máximo de conexiones pendientes en
            algunos sistemas
60      if (f_verbose) printf("hecho\n");
61
62      return sock;
63  }
64
65  // función que cuenta vocales
66  static uint32_t countVowels(char msg[], size_t s)
67  {
68      uint32_t result = 0;
69      size_t i;
70      for (i = 0; i < s; i++)
71      {
72          if (msg[i] == 'a' || msg[i] == 'A' ||
73              msg[i] == 'e' || msg[i] == 'E' ||
74              msg[i] == 'i' || msg[i] == 'I' ||
75              msg[i] == 'o' || msg[i] == 'O' ||
76              msg[i] == 'u' || msg[i] == 'U') result++;
77      }
78      return result;
79  }
80
```

```
81  /*************************** MAIN *****************************/
82  int main(int argc, char * argv[])
83  {
84      // declaración de variables propias del programa principal (locales a main)
85      char f_verbose = 1; // flag, 1: imprimir información extra
86      struct addrinfo * servinfo; // dirección propia (servidor)
87      int sock, conn; // descriptores de socket
88      char msg[BUFF_SIZE]; // espacio para almacenar los datos recibidos
89      ssize_t readbytes; // numero de bytes recibidos
90      uint32_t num, netNum; // contador de vocales en formato local y de red
91      struct sockaddr_storage caddr; // dirección del cliente
92      socklen_t clen; // longitud de la dirección
93
94      // verificación del número de parámetros:
95      if (argc != 2)
96      {
97          printf("Número de parámetros incorrecto \n");
98          printf("Uso: %s puerto\n", argv[0]);
99          exit(1);
100     }
101
102     // obtiene estructura de direccion
103     servinfo = obtener_struct_direccion(NULL, argv[1], f_verbose);
104
105     // crea un extremo de la comunicación. Devuelve el descriptor del socket
106     sock = establecer_servicio(servinfo, f_verbose);
107
108     // cuando ya no se necesite, hay que liberar la memoria dinámica usada para
            la dirección
109     freeaddrinfo(servinfo);
110     servinfo = NULL; // como ya no tenemos la memoria, dejamos de apuntarla para
            evitar acceder a ella por error
111
112     // bucle infinito para atender conexiones una tras otra
113     while (1)
114     {
115         printf("\nEsperando conexión (pulsa <Ctrl+c> para finalizar la ejecución)
                ...\n");
116
117         // acepta la conexión
118         clen = sizeof caddr;
119         if ((conn = accept( 31 , (struct sockaddr *)&caddr, &clen)) < 0)
120         {
121             perror("Error al aceptar una nueva conexión");
122             exit(1);
123         }
124
125         // imprime la dirección obtenida
126         printf("Aceptada conexión con cliente:\n");
127         printsockaddr(&caddr);
128
129         // bucle de contar vocales hasta recibir marca de fin
130         num = 0;
131         do {
132             if ((readbytes = recv( 32 , 33 , BUFF_SIZE,0)) < 0)
133             {
134                 perror("Error de lectura en el socket");
135                 exit(1);
136             }
137             printf("Mensaje recibido: "); fflush(stdout);
```

```
138             write(1, msg, readbytes); // muestra en pantalla (salida estándar 1) el mensaje recibido
139             // evitamos usar printf por si lo recibido no es texto o no acaba con \0
140             num += countVowels(msg, readbytes);
141             printf("Vocales contadas hasta el momento: %d\n",num);
142
143         // condición de final: haber recibido 0 bytes o que el último carácter leído sea EOT
144         } while ((readbytes > 0) && (msg[readbytes - 1] != EOT));
145
146         printf("\nMarca de fin recibida\n");
147         printf("Contadas %d vocales\n", num);  // muestra las vocales recibidas
148         netNum = htonl(num);  // convierte el entero largo sin signo hostlong desde el orden de bytes del host al de la red
149         // envia al cliente el número de vocales recibidas:
150         if (send( 34 , &netNum,sizeof netNum,0) < 0)
151         {
152             perror("Error de escritura en el socket");
153             exit(1);
154         }
155         if (f_verbose) printf("Enviado el número de vocales contadas\n");
156
157         // cierra la conexión con el cliente
158         close( 35 );
159         if (f_verbose) printf("Cerrada la conexión con el cliente\n");
160     }
161     exit(0);
162 }
```