5. Parámetros y funciones de los sockets

5.1. Parámetros de los sockets

Los siguientes tipos de datos estan definidos en `sys/socket.h` y en `sys/un.h` (para los sockets locales) o `netinet/in.h` (para los sockets IPv4 e IPv6).

1. El número de socket (*socket*) es una variable de tipo `int` devuelta por el socket en el momento de su creación y que se utiliza para referenciar a dicho socket a través del programa.

2. El dominio (*domain*) es una variable de tipo `int` que indica el dominio de comunicaciones que usa el socket: `PF_LOCAL`, `PF_UNIX` o `PF_FILE` (los tres son sinónimos y se usan para comunicaciones a través de ficheros temporales), `PF_INET` (direcciones IPv4) y `PF_INET6` (direcciones IPv6).

3. El estilo (*style*) es una variable de tipo `int` que puede valer: `SOCK_STREAM` (para comunicaciones orientadas a conexión como TCP) y `SOCK_DGRAM` (sin conexión como UDP), `SOCK_RAW` (comunicaciones a bajo nivel), etc.

4. La dirección del socket se introduce en una función mediante la estructura genérica `struct sockaddr`. En realidad hay tres formatos distintos de direcciones, pero en la llamada a la función se convierten a esta estructura genérica (así no hay que usar funciones distintas para cada tipo de socket):

- Dirección de un socket local (transmisiones mediante ficheros temporales).

`struct sockaddr_un`	
`short int sun_family`	el formato de la dirección, en este caso `AF_LOCAL`
`char sun_path[108]`	el nombre del fichero temporal

  El tamaño de esta dirección se computa mediante la macro `int SUN_LEN (struct sockaddr_un * ptr)`.

- Dirección de un socket IPv4.

`struct sockaddr_in`	
`sa_family_t sin_family`	aquí se almacena el valor `AF_INET`
`struct in_addr sin_addr`	la dirección IPv4 (ver el apartado)
`uint16_t sin_port`	el número de puerto (ver el apartado)

  Su tamaño se computa con: `sizeof(struct sockaddr_in)`.

- Dirección de un socket IPv6.

`struct sockaddr_in6`	
`sa_family_t sin6_family`	aquí se almacena el valor `AF_INET6`
`struct in6 _addr sin6_addr`	la dirección IPv6 (ver el apartado)
`uint32_t sin6_flowinfo`	esta componente de momento no se usa
`uint16_t sin6_port`	el número de puerto (ver el apartado)

  Su tamaño se computa con: `sizeof(struct sockaddr_in6)`.

5.2. Funciones relacionadas con los sockets

En caso de error las funciones de sockets actualizan la variable `int errno`, cuyo valor puede imprimirse con la función `perror(const char *s)`.

1. La función `int socket (int domain, int style, int protocol)` crea un socket especificando el dominio (`PF_LOCAL` para ficheros, `PF_INET` para IPv4 o `PF_INET6` para IPv6), el estilo (`SOCK_STREAM` orientado a conexión o `SOCK_DGRAM` sin conexión) y el protocolo real que se usa (por defecto, 0, que elige el más apropiado para el tipo de socket).

2. La función `int bind (int socket, struct sockaddr *addr, socklen_t length)` se utiliza para asignar una dirección al socket. La dirección `addr` puede ser de tipo `struct sockaddr_un`, `struct sockaddr_in` o `struct sockaddr_in6` y siempre ha de convertirse al tipo genérico `(struct sockaddr *)` en la llamada a la función. `length` es el tamaño de esta dirección. La función devuelve 0 en caso de éxito y -1 en caso de error.

3. La función `int listen (int socket, unsigned int n)` se utiliza en servidores para indicar al socket que debe ponerse a la escucha de posibles transmisiones por parte de clientes, esto es, hace servidor al socket. El parámetro `n` indica el número de peticiones de clientes que pueden estar en la cola a la espera de ser atendidas. Devuelve 0 en éxito y -1 en fallo.

4. La función `int connect (int socket, struct sockaddr *addr, socklen_t length)` se utiliza en clientes para solicitar una conexión al servidor. Los parámetros son los mismos que para la función `bind`. La función espera a que el servidor responda[1], pero si el servidor no se está ejecutando devuelve un error. Devuelve 0 en caso de conexión y -1 en caso de error.

5. La función `int accept (int socket, struct sockaddr *addr, socklen_t *length_ptr)` se utiliza en los servidores para atender a peticiones de clientes. La dirección `*addr` es una salida que indica quién se ha conectado y `*length_ptr` es, en este caso, un puntero que funciona primero como entrada indicando el tamaño de la dirección y después como salida proporcionando el espacio real consumido. Esta función queda a la espera[1] y cuando se acepta una conexión, la función devuelve un socket nuevo para hacer las transmisiones. Este socket de conexión es el que se debe usar en las funciones de recepción y envío. Si algo falla, la función devuelve -1.

6. La función `int send (int socket, void *buffer, size_t size, int flags)` se utiliza para hacer envíos de datos desde clientes o servidores.

[1] Modificando las opciones de socket podría hacerse no bloqueante, ver el punto 14.

Es similar a la función `write` pero en `flags` podemos poner ciertas opciones (por defecto, 0). La función devuelve el número de bytes transmitidos (lo que no quiere decir que se recibirán sin errores) o -1 en caso de fallo.

7. La función `int recv (int socket, void *buffer, size_t size, int flags)` se usa para recepción de datos en clientes o servidores. Es similar a la función `read` pero en `flags` podemos poner 0 o ciertas opciones (por ejemplo, no borrar los datos o `MSG_DONTWAIT` para hacerlo no bloqueante). El parámetro `size` especifica el número máximo de bytes a leer. La función espera a que los datos lleguen[1] mientras el socket de conexión está abierto. La función devuelve el número de bytes recibidos o -1 en caso de fallo o se cierre el socket.

8. La función `int sendto (int socket, void *buffer, size_t size, int flags, struct sockaddr *addr, socklen_t length)` se utiliza para transmitir datos cuando no existe conexión. Transmite los datos de `buffer` a través de `socket` a la dirección destino especificada por los parámetros `addr` y `length`. El parámetro `size` es el número de bytes a transmitir. Los `flags` y el valor devuelto son los mismos que para `send`.

9. La función `int recvfrom (int socket, void *buffer, size_t size, int flags, struct sockaddr *addr, socklen_t *length-ptr)` se utiliza para recibir datos cuando no existe conexión. El parámetro `size` especifica el número máximo de bytes a leer. Si el paquete es mayor que `size` bytes el resto del paquete se descarta y no hay forma de recuperarlo. Los parámetros `addr` y `length-ptr`, que debe estar inicializado, proporcionan la dirección de la que procede el paquete (para sockets locales estos campos no tienen sentido, puesto que no se puede obtener la dirección de tal socket). También se pueden especificar punteros nulos a `addr` y a `length-ptr` si no estamos interesados en esa información. Los flags y el valor devuelto son los mismos que para `recv`.

10. La función `int close (int socket)` cierra el socket, como un fichero.

11. La función `int shutdown (int socket, int how)` al igual que `close` cierra el socket, pero permite ajustar las acciones: 0 cierra la recepción, 1 cierra la emisión (y ya no espera a los asentimientos) y 2 cierra ambas (como `close`). La función devuelve 0 en caso de éxito y -1 en caso de fallo.

12. La función `int getsockname (int socket, struct sockaddr *addr, socklen_t *length-ptr)` toma como entrada un número de socket y proporciona su dirección y tamaño. Devuelve 0 en éxito y -1 en fallo.

13. La función `int getpeername (int socket, struct sockaddr *addr, socklen_t *length-ptr)` toma como entrada un número de socket y pro-

porciona la dirección y tamaño de la dirección de quién está conectado a dicho socket. La función devuelve 0 en caso de éxito y -1 en caso de error.

14. Las funciones `getsockopt`, `setsockopt`, `fcntl` e `ioctl` (ver el manual) se utilizan para leer o modificar las opciones del socket. Por ejemplo, el socket podría hacerse no bloqueante, esto es, que las funciones retornen inmediatamente sin esperar la respuesta.

15. La función `int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res)` combina la funcionalidad provista por las funciones `getservbyname` y `getservbyport` en una única interfaz. Dado un nodo (un nombre de host o una IP) y un servicio (puede ser `NULL`), la función `getaddrinfo` crea en `res` una lista enlazada de estructuras de dirección (IPv4 o IPv6) de socket (definidas en `sys/types.h`, `sys/socket.h` y `netdb.h`) que pueden ser utilizadas por las llamadas a las funciones `bind` y `connect` para crear sockets. La función devuelve 0 en caso de éxito y -1 en caso de error.

    La estructura `addrinfo` usada por esta función contiene los siguientes miembros:

`struct addrinfo`	
`int *ai_flags`	indicadores
`int ai_family`	`AF_INET` o `AF_INET6`
`int ai_socktype`	`SOCK_STREAM` o `SOCK_DGRAM`
`int ai_protocol`	protocolo
`size_t ai_addrlen`	longitud de la dirección
`struct sockaddr *ai_addr`	dirección
`char *ai_canonname`	nombre canónico
`struct addrinfo *ai_next`	siguiente elemento de la lista

    El parámetro `hints` especifica el tipo de socket o protocolo preferido. Un valor de NULL en `hints` especifica que se acepta cualquier dirección de red o protocolo. Si este parámetro es distinto de `NULL`, la estructura `res` apunta a una estructura `addrinfo` cuyos miembros `ai_family`, `ai_socktype` y `ai_protocol` especifican el tipo de socket preferido. Los miembros en el parámetro `hints` que no se usen deben contener o bien 0 o un puntero `NULL`.

16. La función `void freeaddrinfo(struct addrinfo *res)` libera la estructura `res`.

Ejercicio:

1. Escribir un programa que tome como entrada un nombre de host y que muestre en pantalla el nombre y la dirección de la información obtenida

por la función `getaddrinfo`. En el parámetro `hints` especificar sockets orientados a conexión, IPv4 o IPv6 y que nos muestre el nombre canónico, tal y como se muestra a continuación.

```
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_CANONNAME;
```

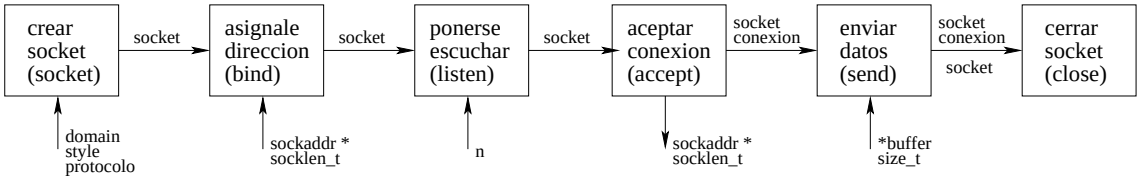
Tener en cuenta que es una lista enlazada y que las direcciones pueden ser IPv4 o IPv6. Probar con *www.tumangaonline.com* o *outlook.office365.com*. Comprobar con `dig nombre` para ver las IPv4 y con `dig nombre aaaa` para las IPv6.

6. Sockets IPv4 orientados a conexión

La comunicación mediante sockets de tipo IPv4 transfiere datos a través de la red. Vamos a utilizar el protocolo TCP, es decir una comunicación orientada a conexión. Aquí tendremos que eligir un número de puerto (que será el mismo en servidor y cliente) y al cliente tendremos que indicarle la dirección IP del servidor (pues uno y otro pueden estar en diferentes ordenadores).

6.1. Programa servidor

El esquema de funcionamiento se muestra en la siguiente figura. El servidor atenderá la petición de cualquier ordenador que se dirija al puerto fijado.



Escribir el programa servidor siguiendo las siguientes instrucciones.

1. Incluir las cabeceras: `stdio.h`, `string.h`, `stdlib.h`, `unistd.h`, `netinet/in.h`, `arpa/inet.h`, `sys/types.h` y `sys/socket.h`.

2. Declarar dos variables `int` para sockets, una `struct sockaddr_in` para la dirección, una `socklen_t` para el tamaño de la dirección y un array de `char` para contener el mensaje a transmitir.

3. Insertar el mensaje en su variable asegurándose que termina con el carácter '\0' para facilitar su impresión. La longitud del mensaje se puede obtener con la función `strlen` (hay que sumarle 1 para incluir el fin de cadena).

4. Crear el socket con la función `socket`. Poner como parámetros, para *domain* `PF_INET` (IPv4), para *style* `SOCK_STREAM` (orientado a conexión), y para *protocol* se puede poner 0. La función devuelve una salida de tipo `int`, el socket servidor.

   Es recomendable comprobar que el socket se ha creado satisfactoriamente, verificando su salida con una sentencia como la siguiente:

   `if(socket< 0) { perror("No se pudo crear socket"); exit(-1); }`.

   La función `perror` (declarada en *stdio.h*) está especialmente diseñada para imprimir mensajes de error.

5. A continuación hay que asignar la dirección al socket. En los campos de la estructura `sockaddr_in` ponemos `AF_INET`, el puerto (que hay que convertir

del orden de host al de la red con `htons(puerto)` y nuestra dirección IP (ponemos cualquiera con `direccion.sin_addr.s_addr=htonl(INADDR_ANY)`). El tamaño se computa con `tamaño=sizeof(struct sockaddr_in)`.

La función que asigna la dirección al socket es `bind`, que toma como argumentos el número de socket, el puntero a la dirección (que hay que convertir al tipo (`struct sockaddr *`) escribiendo esto en la llamada a la función) y el tamaño de la dirección.

De nuevo es recomendable comprobar que no se ha producido ningún error, escribiendo algo como esto:

```
if (bind (socket, (struct sockaddr *) &direccion, tamaño) < 0)
{ perror ("No se pudo asignar direccion"); exit(-1); }
```

6. Ahora toca hacer que el servidor se ponga a escuchar las conexiones. Esto lo hacemos con la función `listen`. Los argumentos son el número del socket y una variable que indica el número de solicitudes que pueden estar en la cola de espera (poner cualquier valor mayor que cero). Como siempre, conviene comprobar que no se ha producido ningún error al llamar a la función.

7. Cuando el servidor tenga una solicitud de conexión por parte de un cliente, queremos que la acepte; esto lo hace la función `accept`. Sus parámetros son los mismos que los de la función `bind`, pero en este caso la dirección es una salida (debemos reservarle un espacio si es necesario) y el tamaño de la dirección es un puntero y una entrada/salida (que hay que iniciar al tamaño de la dirección) que proporcionan información sobre el cliente. Aquí podemos reusar sin problemas la dirección que hemos utilizado en las anteriores funciones.

   Importante: la función devuelve un segundo número de socket (el socket de conexión), que asignaremos a la segunda variable entera declarada y que utilizaremos posteriormente en las funciones de envío y recepción.

   El programa se quedará esperando hasta que un cliente solicite una conexión. Como siempre, si la salida de la función es menor que cero, imprimir un mensaje de error y salir. En caso de éxito, **imprimir** con `printf` un mensaje indicando **la dirección IP y el puerto** de quién se ha conectado.

8. Ahora le enviamos al cliente un mensaje de bienvenida con la función `send` cuyos parámetros son el socket de conexión, el mensaje, el tamaño del mensaje y como `flags` podemos poner 0. De nuevo, comprobar el caso de que se produzca un error; en caso contrario, imprimir el número de bytes enviados.

9. Por último, cerrar los dos sockets con la función `close`, cuyo único argumento es el número de socket. Y el programa está yá listo para ser ejecutado.

10. Cuando ejecutemos el programa servidor, este se quedará esperando a la petición de un cliente. Podemos probarlo usando el comando *telnet* en la forma *telnet IP puerto*, donde *IP* es la IP del ordenador en la que hemos ejecutado el servidor y *puerto* es el puerto que le hemos asignado. El servidor nos enviará el mensaje y el programa se acabará.

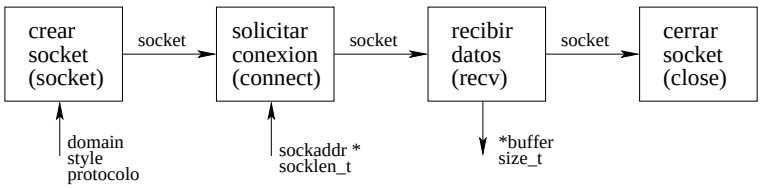
11. Si el programa se interrumpe por cualquier motivo, el socket puede quedar bloqueado durante algunos minutos, hasta que el sistema operativo se de cuenta de que el puerto está bloqueado y lo libere. Para evitar esto, primero deberá matarse al cliente y después al servidor. Otra alternativa es utilizar en cada prueba un número de puerto distinto. Para ello es muy útil usar los argumentos del `main` para especificar el puerto.

12. Hacer que el programa servidor pueda atender a múltiples conexiones cliente (secuencialmente) insertando un lazo del tipo:

    ```
    while(1) { accept(); send(); close(s_conexion); }
    ```

6.2. Programa cliente

El esquema de funcionamiento se muestra en la siguiente figura. El programa cliente necesita la dirección IP y el puerto del servidor que atiende el servicio.



Escribir el programa cliente siguiendo las siguientes instrucciones.

1. Los puntos 1 y 2 del cliente son similares a los del servidor, aunque aquí sólo necesitamos una variable de socket. Además, no necesitamos del punto 3, pero hay que reservar memoria para el mensaje que vamos a recibir.

2. A continuación creamos el socket, lo hacemos con la función `socket` de la misma forma que en el servidor.

3. Una vez creado el socket hay conectarlo al servidor. Previamente tenemos que iniciar la estructura `sockaddr_in` y la variable que almacena su tamaño. Aquí la dirección IPv4 será la dirección del ordenador en el que se ejecuta el servidor. Si la ponemos en el formato textual de números y puntos no necesitamos hacer conversión de bytes. Por ejemplo, si estamos en el mismo ordenador que el servidor, podemos elegir una de las tres formas alternativas:

a) `direccion.sin_addr.s_addr=inet_addr("127.0.0.1");`
b) `inet_aton("127.0.0.1",&direccion.sin_addr);`
c) `inet_pton(AF_INET,"127.0.0.1",&direccion.sin_addr);`.

Ahora tenemos que llamar a la función `connect`. Sus argumentos y salidas son similares a los de `bind` para el programa servidor.

4. Ya estamos listos para recibir el mensaje que nos va a mandar el servidor. Lo recogemos con la función `recv`, cuyos argumentos son los mismos que los de la función `send` del programa servidor. Aquí el número del socket es el único que hay definido y el tamaño del mensaje no se conoce a priori, por lo que aquí hay que especificar el máximo tamaño reservado. Mostrar en pantalla el mensaje recibido y el número de bytes recibidos.

5. Por último, cerramos el socket con la función `close`.

6. Ejecutar primero el servidor en una ventana y luego el cliente en otra ventana. Si todo va bien el servidor nos responderá y veremos en la pantalla del cliente el mensaje transmitido. En este punto, el cliente finalizará. También se puede probar ejecutando en un terminal la orden `nc -lk puerto` (servidor escuchando en ese puerto) y el cliente en otro y escribir un mensaje en el primero.

7. Hacer que el programa cliente se pueda conectar con el servidor ejecutándose en otro ordenador, es decir, cambiando en el punto 3 la IP del lazo de vuelta por la IP del servidor. Es una buena opción usar los argumentos del `main` para introducir el puerto y la IP del servidor. Para conectarse a otro equipo, se usa `ssh ip_ordenador` o `ssh nombre_ordenador`.

6.3. Ejercicios

Mantener las versiones de los programas correspondientes a los diferentes ejercicios.

1. Comprobar qué ocurre si en el cliente hay una sentencia `recv`, pero en el servidor no hay una `send` correspondiente. Para ver el comportamiento, poner en el servidor un `sleep` antes de cerrar el socket de conexión e inicializar el string donde se recibe el mensaje.

2. Comprobar también el caso contrario: que el servidor tenga `send` pero que el cliente no tenga `recv`. Como en el caso anterior, inicializar el string donde se recibe el mensaje.

3. Comprobar qué ocurre si el servidor sale con `exit(0)` sin cerrar los sockets y el cliente intenta todavía seguir leyendo o escribiendo en el socket. Para

probarlo, poner un `sleep` en el cliente para asegurarse de que lee del socket cuando el servidor ya salió.

4. Comprobar si se pueden transmitir dos mensajes con sendas funciones `send` y recibir después de un tiempo prudencial ambos mensajes con una única sentencia `recv`. Imprimir el **número de bytes recibidos**.

5. Comprobar qué ocurre si la función `recv` lee menos datos que los que le envió la sentencia `send`. ¿Podrán recuperarse los datos restantes con una nueva sentencia `recv`? ¿Para qué puede servir la sentencia siguiente?

   `while( (n=recv(socket,mensaje,tamaño_mensaje,0)) >0 ) { etc }`.

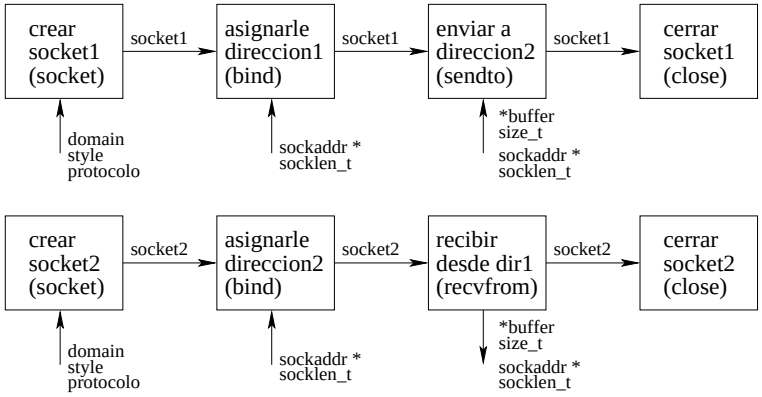
6. *Devolución de un eco.* Modificar los programas servidor y cliente iniciales, tal que:

   a) el cliente transmita al servidor lo que el usuario escriba por teclado,
   b) el servidor se lo devuelva al cliente pasado a mayúsculas,
   c) el cliente lo muestre en pantalla,
   d) de vuelta al paso a)

   El servidor puede atender a los sucesivos ecos de un mismo cliente con `while((n=recv(...))>0)`. La función que convierte un carácter a mayúsculas es `toupper`. Asegurarse que los mensajes terminen con el carácter '\0'. Recordar que el servidor debe ser capaz de atender a múltiples clientes secuencialmente, como se indica en el punto 12 de la descripción del programa servidor.

7. Sockets IPv4 sin conexión

En este apartado vamos a utilizar el protocolo UDP, es decir, que la comunicación será sin conexión. Los esquemas de funcionamiento de ambos lados de la comunicación se muestran en la siguiente figura. Como puede verse, no hay conexión de ningún tipo y en cada mensaje transmitido hay que indicar la dirección del ordenador destino.



Escribir los dos programas siguiendo las siguientes instrucciones.

1. Incluir las cabeceras `stdio.h`, `string.h`, `stdlib.h`, `unistd.h`, `netinet/in.h`, `arpa/inet.h`, `sys/types.h` y `sys/socket.h`.

2. Para cada uno de los programas nos llega con declarar una sola variable socket de tipo `int`, pero tenemos que declarar dos estructuras `sockaddr_in` para las direcciones, una para la dirección del socket propio y otra para indicar el socket remoto, además de una variable de tipo `socklen_t` para el tamaño de las estructuras y un array de `char` que contendrá el mensaje.

3. Rellenar las dos direcciones `sockaddr_in` con los datos propios y remotos (IP, puerto y familia) en ambos programas (en el lado que va a recibir realmente sólo es necesario rellenar la dirección propia). Podemos poner cualquier dirección IP propia con `htonl(INADDR_ANY)`. Si el lado remoto es el mismo ordenador, ponemos la IP remota con `inet_addr("127.0.0.1")`, teniendo en cuenta que, en este caso, los números de puerto han de ser distintos. Es una buena opción permitir cambiar los puertos en la línea de comandos, usando los argumentos del main. En segundo lugar, obtener el tamaño de las estructuras de dirección. Por último, insertar el mensaje a enviar en su correspondiente variable.

4. Crear un socket en cada programa con la función `socket`. Poner como argumentos, para *domain* `PF_INET` (IPv4), para *style* `SOCK_DGRAM` (sin conexión), y para *protocol* se puede poner 0.

5. Asignar al socket la dirección `sockaddr_in` que va a utilizar con la función `bind`. Sólo se usa un socket y una llamada a la función `bind` por programa. Uno de los lados ha de utilizar una de las direcciones y, el otro, la otra. Verlo en la figura del principio.

6. El primero de los programas le va a enviar al segundo el mensaje. La función de envío es `sendto` y sus argumentos son el número de socket, el mensaje, el tamaño del mensaje, los `flags` (que podemos poner 0) y la estructura y tamaño de la dirección del socket destino (del segundo programa). La salida es el número de bytes enviados por la función (mostrarla en pantalla). Se puede probar este programa ejecutando en un terminal la orden `nc -lku puertorecepción` y después en otro terminal el programa que envía.

7. El segundo de los programas recibirá el mensaje usando la función `recvfrom` que tiene como parámetros el número de socket, el mensaje, el tamaño del mensaje (aquí hay que poner el tamaño del array usado como almacenamiento puesto que no sabemos a priori el tamaño que tendrá el mensaje que vamos a recibir) y los `flags` (que podemos poner 0). La función nos proporciona la estructura y tamaño de la dirección del socket origen (primer programa), siempre que le hayamos reservado un espacio e indicado un tamaño. Si no estuviésemos interesados en esta información, podríamos poner en ambos NULL. La salida es el número de bytes recibidos por la función. Se puede probar este programa ejecutándolo en un terminal y después en otro terminal la orden `echo "mensaje"| nc -u -q1 localhost puertoenvío`.

   Mostrar en pantalla la dirección IP y el puerto del emisor, el número de bytes recibidos y el mensaje.

8. Por último, cerrar cada socket con la función `close`.

9. Lanzamos a ejecutar primero el programa 2 (el de recepción) y luego el 1 (el de envío). El programa 2 mostrará en pantalla el mensaje y ambos programas saldrán.

10. Hacer que ambos programas se puedan ejecutar en distinto ordenador, como se hizo en el caso de TCP.

7.1. Ejercicios

Mantener las versiones de los programas correspondientes a los diferentes ejercicios.

1. Comprobar qué ocurre si la función `recvfrom` lee menos datos que los que le envió la sentencia `sendto`. ¿Podrán recuperarse los datos restantes con una nueva sentencia `recvfrom`?

2. *Otros tipos de datos.* Modificar los programas para que en vez de transmitir cadenas de texto, se transmita un array de números de tipo float.

3. *Servicio de eco.* Convertir los programas en un servidor de eco y en un cliente de eco, de forma que:

   a) el cliente transmita al servidor lo que el usuario escriba por teclado,
   b) el servidor se lo devuelva al cliente pasado a mayúsculas,
   c) el cliente lo muestre en pantalla,
   d) de vuelta al paso a)

   Hacer que el servidor pueda atender a múltiples clientes insertando un lazo:

   ```
   while(1) { recvfrom(); sendto(); }
   ```

   Comprobar que se pueden atender varios clientes simultáneamente.