Sockets IPv4 orientados a conexión (TCP) y no orientados a conexión (UDP)

Tomás Fernández Pena
José Carlos Cabaleiro Domínguez

Grado en Ingeniería Informática

Universidade de Santiago de Compostela

Redes, 2º Curso GrEI

Índice

Índice

Sockets

Un **socket** es una estructura software en un nodo que sirve como punto final para enviar y recibir datos

- Establecen una interfaz entre la aplicación y la capa de transporte
- Analogía: servicio de correo postal
  - Aplicación: usuario que deposita la carta en un buzón
  - Capa de transporte: cartero que recoge la carta del buzón
  - Socket: el buzón
- Hay diferentes tipos de sockets: locales, TCP, UDP, raw

Para establecer una comunicación entre dos puntos necesitamos dos sockets, uno en cada punto de la misma

- Socket origen
- Socket destino

Parámetros de los sockets

- El número de socket: variable `int` identificador del socket
- El dominio de comunicación (`AF_LOCAL`, `AF_INET`, `AF_INET6`, etc.)
  - Indica la familia de la dirección a usar en la comunicación
- El tipo de socket (`SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_RAW`, etc.)
- El protocolo a usar en la comunicación (`IPPROTO_TCP`, `IPPROTO_UDP`, `IPPROTO_ICMP`, etc.)
  - Normalmente, para cada dominio/tipo hay un único protocolo (pe, `AF_INET`/`SOCK_STREAM` → `IPPROTO_TCP`)
  - Se puede usar 0 como “cualquier protocolo”
  - La lista de protocolos y su número correspondiente se puede ver en el fichero `/etc/protocols` o en https://www.iana.org/assignments/protocol-numbers/
- La dirección del socket: se introduce a través de la estructura `struct sockaddr`

Sockets de red

Trabajaremos con sockets con las siguientes especificaciones:

- Dominio: `AF_INET` o `AF_INET6`
- Tipo: `SOCK_STREAM` (orientado a conexión) o `SOCK_DGRAM` (no orientado a conexión)
- Protocolo: por defecto para cada par dominio/tipo (protocolo 0)

Es necesario ligar (*bind*) cada socket a una dirección IP y un puerto:

- Dirección IP del ordenador origen y puerto del proceso origen
- Dirección IP del ordenador destino y puerto del proceso destino
- Valores encapsulados en una `struct sockaddr` (definida en `sys/socket.h` y `netinet/in.h`)

Estructura `struct sockaddr`

Un par dirección IP/puerto se encapsula en una estructura de tipo `struct sockaddr`

- Esta estructura se usa en la especificación de *sockets* y en otras funciones

La `struct sockaddr` es una estructura genérica

- Se usan versiones particulares de la misma según el dominio de conexión (p. ej., `AF_INET` o `AF_INET6`)
- Se hace un *cast* de esas versiones a la genérica cuando se usan en funciones
  - Evita tener que usar funciones distintas para cada tipo

Estructuras particulares para IPv4 y IPv6

- Direcciones IPv4

struct sockaddr_in	
sa_family_t sin_family	dominio (AF_INET)
struct in_addr sin_addr	la dirección IPv4
uint16_t sin_port	el número de puerto

Tamaño = `sizeof(struct sockaddr_in)`

- Direcciones IPv6

struct sockaddr_in6	
sa_family_t sin6_family	dominio (AF_INET6)
struct in6_addr sin6_addr	la dirección IPv6
uint32_t sin6_flowinfo	campo no usado
uint16_t sin6_port	el número de puerto

Tamaño = `sizeof(struct sockaddr_in6)`

Sockets

- Sockets orientados a conexión (TCP)
  - Dos programas con roles (estructura) diferentes
    - Cliente: solicita conexión
    - Servidor: acepta la conexión
    - Después se transmiten datos
- Sockets no orientados a conexión o sin conexión (UDP)
  - Dos programas con roles (estructura) similares
    - Extremo 1: envía y/o recibe datos
    - Extremo 2: recibe y/o envía datos

Principales funciones relacionadas con los sockets

- `int socket()` crear un socket
- `int bind()` asignar dirección a un socket
- `int listen()` marcar un socket como pasivo (para poder atender peticiones de clientes)
- `int connect()` en clientes, para solicitar la conexión con un servidor
- `int accept()` en servidores, para aceptar la conexión de clientes
- `int send()/recv()` enviar/recibir datos entre clientes y servidores
- `int sendto()/recvfrom()` enviar/recibir datos cuando no existe conexión establecida
- `int close` cerrar un socket

Todas estas funciones actualizan la variable `errno`

- función `perror()` para mostrar el error

Función `socket()`: creación de un socket

```
int socket(int domain, int style, int protocol)
```

- Parámetros:
  - `domain` entero que debe valer `AF_INET` para direcciones IPv4 o `AF_INET6` para IPv6
  - `style` usaremos `SOCK_STREAM` (orientado a conexión, TCP) o `SOCK_DGRAM` (no orientado a conexión, UDP)
  - `protocol` por defecto para cada par dominio/tipo (protocolo 0)
- Valor devuelto: entero identificador del socket en caso de éxito, -1 en caso de error
- Necesario incluir `sys/socket.h` y `sys/types.h`

Función `bind()`: asignación de dirección a un socket

```
int bind(int socket, struct sockaddr *addr,
         socklen_t length)
```

- Parámetros:
  - `socket` entero identificador del socket
  - `addr` puntero a un `struct sockaddr` con la dirección que se quiere asignar al socket
    - Puede ser IPv4 o IPv6, pero hay que convertirla al tipo genérico (`struct sockaddr *`) en la llamada a la función
    - En el caso del servidor debe ponerse `INADDR_ANY` para que pueda aceptar conexiones a través de cualquiera de las interfaces del mismo[1]
  - `length` tamaño de la estructura addr
- Valor devuelto: 0 en caso de éxito, -1 en caso de error

[1] Si se pone localhost, solo aceptará conexiones a la interfaz de lazo de vuelta

Función `listen()`: indicación de ponerse a la escucha

```
int listen(int socket, unsigned int n)
```

- Se usa en servidores:
  - Marcar el socket como pasivo, es decir, podrá escuchar posibles conexiones de clientes
    $\Longrightarrow$ **socket servidor**
- Parámetros:
  - `socket` entero identificador del socket
  - `n` número de peticiones de clientes que pueden estar en cola a la espera de ser atendidas
- Valor devuelto: 0 en caso de éxito, -1 en caso de error

Función `connect()`: solicitud de conexión

```
int connect(int socket, struct sockaddr *addr,
            socklen_t length)
```

- Se usa en clientes:
  - Solicitar la conexión con un servidor
- Parámetros: los mismos que la función `bind()`
  - `socket` entero identificador del socket
  - `addr` puntero a un `struct sockaddr` con la dirección del socket del servidor
    - Puede ser IPv4 o IPv6, pero hay que convertirla al tipo genérico (`struct sockaddr *`) en la llamada a la función
  - `length` tamaño de la estructura `addr`
- La función espera a que el servidor responda[2], pero si el servidor no se está ejecutando devuelve un error
- Valor devuelto: 0 en caso de conexión, -1 en caso de error

[2] Modificando las opciones de creación del socket podría hacerse no bloqueante

Función `accept()`: aceptar la conexión

```
int accept(int socket, struct sockaddr *addr,
           socklen_t *length_ptr)
```

- Se usa en servidores:
  - Atender las peticiones de clientes
- Parámetros:
  - `socket` entero identificador del socket de servidor
  - `addr` salida que es un puntero a un `struct sockaddr` con la dirección del cliente que se ha conectado
  - `length_ptr` puntero que es parámetro de entrada indicando el tamaño de la estructura `addr` y de salida con el espacio real consumido
- La función queda a la espera[3] y cuando se acepta la conexión, devuelve un nuevo socket $\Longrightarrow$ **socket de conexión**
- Valor devuelto: entero identificador del socket de conexión, -1 en caso de error

[3] Modificando las opciones del socket podría hacerse no bloqueante

Función `send()`: envío de datos

```
ssize_t send(int socket, void *buffer,
             size_t size, int flags)
```

- Envío de datos entre clientes y servidores
- Parámetros:
  - `socket` entero identificador del socket
  - `buffer` puntero a los datos a enviar
  - `size` número de bytes a enviar
  - `flags` opciones del envío. Por defecto 0
- Valor devuelto[4]: número de bytes transmitidos (no implica que se reciban sin errores), -1 en caso de error

[4] El tipo `ssize_t` es como `size_t`, pero además puede ser negativo

Función `recv()`: recepción de datos

```
ssize_t recv(int socket, void *buffer,
             size_t size, int flags)
```

- Recepción de datos entre clientes y servidores
- Parámetros:
  - `socket` entero identificador del socket
  - `buffer` puntero donde se recibirán los datos
  - `size` número máximo de bytes a recibir
  - `flags` opciones de la recepción[5]. Por defecto 0
- La función espera a que los datos lleguen[6] mientras el socket de conexión esté abierto
- Si se cierra el socket de conexión, sale devolviendo un 0
- Valor devuelto: número de bytes recibidos, -1 en caso de error

[5] Por ejemplo, no borrar datos o hacerla no bloqueante

[6] Salvo que se haya cambiado el comportamiento por defecto

Función `sendto()`: envío de datos sin conexión

```
ssize_t sendto(int socket, void *buffer, size_t size,
               int flags, struct sockaddr *addr,
               socklen_t length)
```

- Parámetros:
  - `socket` entero identificador del socket
  - `buffer` puntero a los datos a enviar
  - `size` número de bytes a enviar
  - `flags` opciones del envío. Por defecto 0
  - `addr` puntero a un `struct sockaddr` con la dirección a la que se quiere enviar
  - `length` tamaño de la estructura `addr`
- Valor devuelto: número de bytes transmitidos (no implica que se reciban sin errores), -1 en caso de error

Función `recvfrom()`: recepción de datos sin conexión

```
ssize_t recvfrom(int socket, void *buffer, size_t size,
                 int flags, struct sockaddr *addr,
                 socklen_t *length-ptr)
```

- Parámetros:
  - `socket` entero identificador del socket
  - `buffer` puntero donde se recibirán los datos
  - `size` número máximo de bytes a recibir
  - `flags` opciones de la recepción (igual que `recv()`). Por defecto 0
  - `addr` salida que es un puntero a un `struct sockaddr` con la dirección de la procedencia del paquete
  - `length_ptr` puntero que es parámetro de entrada indicando el tamaño de la estructura `addr` y de salida con el espacio real consumido[7]
- La función espera a que lleguen datos[8]
- Valor devuelto: número de bytes recibidos, -1 en caso de error

[7] Si `addr` es `NULL` y `length` es 0, indica que no interesa la procedencia

[8] Salvo que se haya cambiado el comportamiento por defecto

Resumen de las funciones de envío y recepción

Funciones de envío:

- `send()` para envíos orientados a conexión
- `sendto()` para envíos orientados a conexión y sin conexión
  - `send(sockfd, buf, len, flags);` equivale a `sendto(sockfd, buf, len, flags, NULL, 0);`

Funciones de recepción:

- `recv()` para recepciones orientados a conexión
- `recvfrom()` para recepciones orientados a conexión y sin conexión
  - `recv(sockfd, buf, len, flags);` equivale a `recvfrom(sockfd, buf, len, flags, NULL, 0);`

En la práctica

- `send()` y `recv()` para orientado a conexión (TCP)
- `sendto()` y `recvfrom()` para sin conexión (UDP)

Función `close()`: cierre del socket

```
int close(int socket)
```

- Cierra el socket
- Parámetros:
  - `socket` entero identificador del socket

Otras funciones relacionadas con los sockets

```
int shutdown(int socket, int how)
```

- Cierra el socket, pero permite ajustar las acciones: 0 cierra la recepción, 1 cierra la emisión y 2 cierra ambas[9]

```
int getsockname(int socket, struct sockaddr *addr,
                socklen_t *length-ptr)
```

- A partir de un socket, proporciona la dirección y su tamaño

```
int getpeername(int socket, struct sockaddr *addr,
                socklen_t *length-ptr)
```

- A partir de un socket, proporciona la dirección y su tamaño de quién está conectado

`getsockopt()`, `setsockopt()`, `fcntl()` e `ioctl()`

- Leer o modificar las opciones del socket, por ejemplo, el carácter no bloqueante

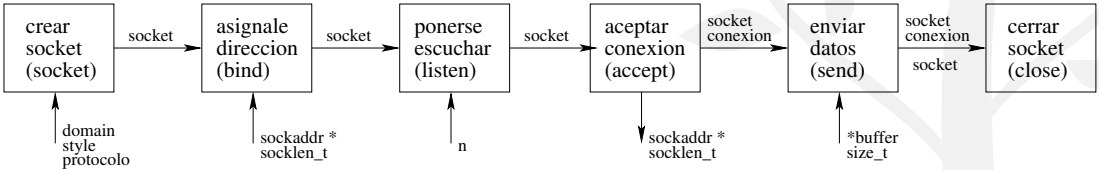
[9] Como la función `close()`

Índice

Sockets IPv4 orientados a conexión

Programa servidor y programa cliente básicos: el servidor envía un mensaje al cliente

- **Programa servidor**
  - Elegir un número de puerto[10] en el que escuchará peticiones de clientes
- Programa cliente
  - IP del servidor
  - Puerto que el servidor usa para escuchar peticiones
- **Esquema del programa servidor**



[10] Puerto mayor que `IPPORT_USERRESERVED`

Programa servidor TCP

- Incluir las cabeceras necesarias. Usar `man`
- Declarar variables:
  - Dos `int` para los sockets (socket de servidor y socket de conexión)
  - Un `struct sockaddr_in` para la dirección
  - Un `socklen_t` para el tamaño
  - Inicializar un string para el mensaje[11]

[11] Con `strlen` se obtiene la longitud del mensaje sin el terminador de string

Programa servidor TCP

Crear el socket con la función `socket()`

- `domain` AF_INET (IPv4)
- `style` SOCK_STREAM (orientado a conexión)
- `protocol` 0 (valor por defecto)
- Devuelve el `int` que identifica el **socket de servidor**
- Comprobar que se ha creado satisfactoriamente: algo como

```
if(sockserv < 0){
    perror("No se pudo crear el socket");
    exit(EXIT_FAILURE);
}
```

Programa servidor TCP

Asignar dirección y puerto al socket con la función `bind()`

- En la estructura `sockaddr_in` se indican:
  - AF_INET
  - Una de las IPs locales del servidor (del interfaz por el que se van a atender las peticiones) en formato binario
    - Para aceptar peticiones que lleguen por cualquier interfaz usad `INADDR_ANY`
    - Esa macro está en orden de host: hay que cambiarla
      `ipportserv.sin_addr.s_addr=htonl(INADDR_ANY)`
  - El puerto **en orden de red**
- Convertirla a `struct sockaddr *` en la llamada a la función y comprobar si da error

```
if(bind(sockserv, (struct sockaddr *) &ipportserv, sizeof(struct sockaddr_in)) < 0) {
    perror("No se pudo asignar direccion");
    exit(EXIT_FAILURE);
}
```

Programa servidor TCP

Marcar el socket como pasivo (para que pueda **escuchar** peticiones) con la función `listen()`

- Un socket de tipo pasivo se usa para atender solicitudes remotas de conexión
- Número máximo de solicitudes en la cola de espera: cualquier valor > 0
- Comprobar que no se ha producido error

Programa servidor TCP

Aceptar la conexión con la función `accept()`

- Parámetros similares que en la función `bind()`, pero ahora:
  - La estructura `addr` **es de salida**: dirección y puerto del cliente
  - El tamaño de la dirección es de entrada y de salida: inicializarla a `sizeof(struct sockaddr_in)` y pasar el puntero
- Devuelve un número de socket (**socket de conexión**), que se usará con las funciones de envío y recepción
- Se queda esperando a que un cliente solicite conexión
- Comprobar que se ha creado satisfactoriamente: algo como

```
if((sockcon = accept(sockserv, (struct sockaddr *) &ipportcli, &tamano)) < 0) {
    perror("No se pudo aceptar la conexion");
    exit(EXIT_FAILURE);
}
```

- Mostrar la **IP y el puerto** de quién se ha conectado

Programa servidor TCP

Enviar[12] un mensaje al cliente con la función `send()`

- Parámetros:
  - El socket de conexión
  - El mensaje
  - El tamaño del mensaje
  - `flags` el valor por defecto, 0
- Comprobar que se ha enviado correctamente
- Imprimir el **número de bytes enviados**

Cerrar los sockets con la función `close()`

- Parámetro: el identificador del socket que se desea cerrar

[12] También se podría recibir un mensaje del cliente con la función `recv`

Programa servidor TCP

Comprobar su funcionamiento

- El programa se bloquea esperando una conexión
- En un terminal se escribe `telnet IP puerto`
  - `IP` es la IP donde se ejecuta el servidor
  - `puerto` es el puerto elegido
- Se recibirá el mensaje del servidor y el programa terminará

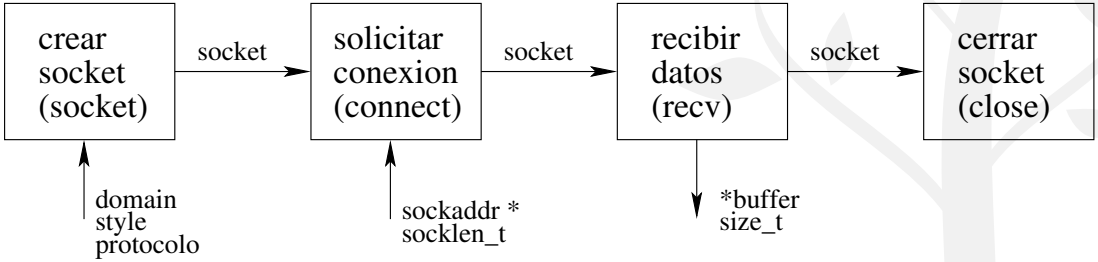
Programa servidor TCP

Mejora: permitir que el servidor atienda múltiples conexiones de clientes (secuencialmente) con un esquema del tipo:

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

Sockets IPv4 orientados a conexión

- Programa servidor
  - Elegir un número de puerto en el que escuchará peticiones de clientes
- **Programa cliente**
  - IP del servidor
  - Puerto que el servidor usa para escuchar peticiones
- **Esquema del programa cliente**



Programa cliente TCP

Cabeceras y variables

- Las mismas cabeceras que en el servidor
- Declarar un `int` para el socket, un `struct sockaddr_in` para la dirección, un `socklen_t` para el tamaño y declarar un string para recibir el mensaje[13]

Crea el socket con la función `socket()`

- Del mismo modo que en el servidor

Inicializa la estructura `sockaddr_in` con la dirección y puerto del servidor

- La IP es la dirección donde se ejecuta el servidor

```
inet_pton(AF_INET,IP_text,&direccion.sin_addr);
```

- Si se ejecuta en el mismo ordenador, `IP_text` es la del lazo de vuelta, 127.0.0.1

Solicita la conexión con la función `connect()`

- Mismos argumentos que la función `bind()` del servidor

[13] Asegurarse que tiene espacio suficiente para el mensaje

Programa cliente TCP

Recibe[14] el mensaje del servidor con la función `recv()`

- Parámetros:
  - El socket
  - El mensaje
  - El tamaño máximo del mensaje
  - `flags` el valor por defecto, 0
- Comprobar que se ha enviado correctamente
- Imprimir el mensaje y el número de bytes recibidos[15]

Cierra el socket con la función `close()`

[14] También se podría enviar un mensaje al servidor con la función `send`

[15] Lo que devuelve la función `recv()`, no el tamaño del string

Programa cliente TCP

Comprueba su funcionamiento

- Ejecutar el programa servidor[16]
- En otro terminal ejecutar el cliente
  - Si todo es correcto, se verá el mensaje del servidor y terminará

[16] También se puede simular un servidor con el programa `nc -lk puerto`

Programa cliente TCP

Prueba final: probar la conexión entre dos ordenadores diferentes

- La prueba debe hacerse entre dos ordenadores del aula (la eduroam bloquea las conexiones entre portátiles)
- Abre un terminal remoto usando:
  `ssh [usuario@]ip_ordenador` o `ssh [usuario@]nombre_ordenador`
  - Si el usuario es el mismo, no es necesario especificarlo
- Para copiar un fichero de un ordenador a otro se usa:
  `scp fichero [usuario@]ip_ordenador:dir_destino`
  donde `dir_destino` es el directorio en el ordenador remoto en el que se copia el fichero
- Ejemplo: copiar servidor.c en el directorio home del usuario en el PC con IP 172.25.45.53
  `scp servidor.c 172.25.45.53:~`
  (la ~ indica el directorio home del usuario)

Índice

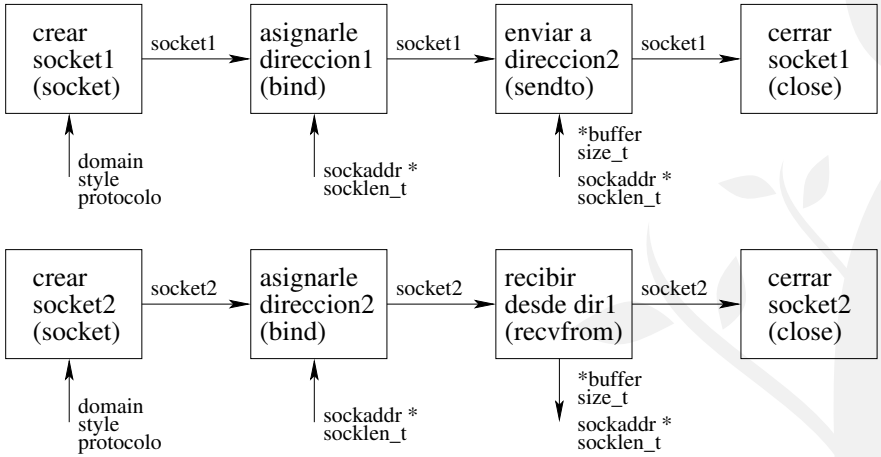
Sockets IPv4 sin conexión

El Programa 1 envía un mensaje al Programa 2

- Programa 2, que recibe datos
  - Elegir un puerto[17] en el que recibirá datos (puerto propio 2)
- Programa 1, que envía datos
  - Elegir un puerto que usará para enviar datos (puerto propio 1)
  - IP del equipo donde se ejecuta el programa 2
  - Puerto en el que el programa 2 recibirá datos (puerto remoto 1, que coincidirá con puerto propio 2)

[17] Puerto mayor que `IPPORT_USERRESERVED`

Sockets IPv4 sin conexión



Programas de envío/recepción UDP

- Los dos programas siguen el mismo esquema
- Incluir las cabeceras: las mismas que en el caso de TCP
- Declarar un `int` para el socket, dos `struct sockaddr_in`, una para la dirección del socket propio y otra para el remoto, un `socklen_t` para el tamaño y un string para el mensaje
- **Crear el socket** con la función `socket()`
  - `domain` AF_INET (IPv4)
  - `style` SOCK_DGRAM (sin conexión)
  - `protocol` 0 (valor por defecto)
  - Devuelve el `int` que identifica el socket
  - Comprobar que se ha creado satisfactoriamente

Programas de envío/recepción UDP

- Inicializar las estructuras `struct sockaddr_in`
- Programa que envía
  - Una con la IP y puerto propios (`INADDR_ANY` y el puerto elegido)
  - La otra con la IP[18] y puerto remotos (a dónde enviar datos)
- Programa que recibe
  - Una con la IP y puerto propios (`INADDR_ANY` y el puerto elegido)
  - La otra se sobreescribirá con la IP y puerto remotos, que los obtendrá del mensaje que reciba[19]
- **Asignar dirección al socket** con la función `bind()`
  - Al socket solo se le asigna la dirección local
  - La dirección remota se usa en las funciones de transmisión de datos
  - Comprobar que se ha creado satisfactoriamente

[18] Si se usa 127.0.0.1, los puertos tienen que ser distintos

[19] El tamaño sí se debe inicializar de todas formas

Programa de envío UDP

Enviar el mensaje con la función `sendto()`

- Parámetros:
  - El número socket
  - El mensaje
  - El tamaño del mensaje
  - `flags` el valor por defecto, 0
  - La estructura con la dirección del socket destino
  - El tamaño de la dirección
- Comprobar que se ha enviado correctamente
- Imprimir el **número de bytes enviados**

Cerrar el socket con la función `close()`

Programa de recepción UDP

Recibir el mensaje con la función `recvfrom()`

- Parámetros:
  - El número socket
  - El mensaje
  - El tamaño máximo del mensaje
  - `flags` el valor por defecto, 0
  - La estructura donde obtendremos la dirección del socket destino
  - El tamaño de la dirección[20]
- Comprobar que se ha recibido correctamente
- Imprimir el **número de bytes recibidos**
- Imprimir la **IP y el puerto** de quién ha enviado los datos

Cerrar el socket con la función `close()`

[20] Es un parámetro de entrada y de salida, es decir, debe estar indicado

Programas de envío/recepción UDP

Comprobar su funcionamiento

- En primer lugar se ejecuta el programa que recibe
- En otro terminal se ejecuta el que envía
  - Si todo es correcto, el programa que recibe imprimirá el mensaje y terminará
- El programa que envía se puede comprobar
  - En un terminal se ejecuta `nc -lku puertorecepción`
  - En otro terminal el programa que envía
  - En el primer terminal saldrá el mensaje enviado
- El programa que recibe se puede comprobar
  - En un terminal se ejecuta el programa que recibe
  - En otro terminal se ejecuta `echo "mensaje"| nc -u -q1 localhost puertoenvío`
  - En el primer terminal saldrá el mensaje enviado

Hacer que ambos programas puedan ejecutarse en distintos ordenadores, como en TCP, usando los argumentos del `main()` para introducir las direcciones y puertos necesarios