Universidad de Murcia

Facultad de Informática

Departamento de Ingeniería y Tecnología de Computadores

Área de Arquitectura y Tecnología de Computadores

PRÁCTICAS DE REDES

I.I./I.T.I. SISTEMAS/I.T.I. GESTIÓN

Boletín 1 – Programación con *Sockets*

SEPTIEMBRE DE 2009

Índice

1. Introducción **3**

2. Sockets **3**

3. Llamadas al sistema para el manejo de *sockets* **7**

4. Resumen **13**

5. Ejercicios propuestos **14**

6. Programas de ejemplo **14**

6.1. common.h 15
6.2. cliente_tcp.c 16
6.3. servidor_tcp.c 18
6.4. cliente_udp.c 20
6.5. servidor_udp.c 22
6.6. getmyip.c 24

Bibliografía **25**

1. Introducción

Este boletín es una guía para la primera sesión de prácticas. La documentación está dividida en una introducción teórica, un resumen de los conceptos fundamentales, varios ejercicios propuestos y algunos programas de ejemplo.

2. Sockets

Los ***sockets*** fueron la respuesta del UNIX de Berkeley (4.3BSD) a las comunicaciones entre procesos. Por su parte, System V proporcionaba un mecanismo parecido, llamado TLI (*Transport Layer Interface*), que pronto quedó desbancado por los *sockets* de Berkeley.

Los *sockets* ofrecen un mecanismo que permite comunicar procesos de una misma máquina o de distintas máquinas (a través de una **red de ordenadores**) utilizando un interfaz de llamadas al sistema muy parecido al utilizado con los ficheros normales o con las tuberías.

Desde el punto de vista del programador, los *sockets* ofrecen un canal de comunicación que permite comunicar dos procesos (llamados a veces *entidades pares*, véase figura 1). Las características de este canal de comunicación pueden ajustarse según las necesidades de los dos procesos. Así, por ejemplo, se pueden crear canales de comunicaciones limitados a un uso local (es decir, sólo comunicarán procesos en la misma máquina), canales que se comportan como un flujo continuo de bytes, o canales que responden a un uso petición/respuesta.

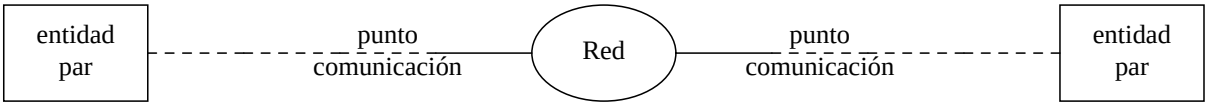

Figura 1: Los dos *sockets* en una comunicación entre entidades pares

Tipos de *sockets*

De forma genérica, los tipos de *sockets* se agrupan en familias. Una familia engloba a todos los *sockets* que comparten ciertas características, como protocolos de comunicación, convenios para formar direcciones de ordenadores, etc. Las dos familias más comunes (y las que se explican en este documento) son:

UNIX Familia de *sockets* restringida a conexiones locales (dentro de una misma máquina). Utilizan nombres de ficheros como direcciones, y, conceptualmente, son iguales a las tuberías, aunque ambos extremos son de lectura y escritura.

Internet Familia de *sockets* que permite la comunicación de dos procesos cualquiera en toda la red Internet.

Los *sockets*, además, pueden ser fundamentalmente de tres tipos:

Sockets *stream*. Los *sockets stream* están relacionados con el protocolo de transporte más famoso en Internet (la parte TCP de TCP/IP). TCP se encarga de que la comunicación se vea como un *stream* contínuo de bytes en el que la aplicación puede escribir y leer datos. TCP también asegura que los datos van a llegar en orden y que se va a mantener la conexión de una manera fiable durante todo el tiempo que dure la comunicación. Por tanto, es un protocolo orientado a la conexión, lo que significa que tiene una fase de establecimiento en donde se reservan los recursos necesarios para mantener la comunicación, una de comunicación, y otra de liberación de la conexión. Esto es un trabajo difícil, ya que TCP sólo dispone de los servicios de IP, que es no orientado a la conexión. El sistema operativo y los lenguajes de programación ofrecen estos servicios al programador como integrados en los ficheros estándar: escribir/leer en/de un *socket* es como escribir/leer en/de un fichero.

Sockets datagrama. Al contrario que los anteriores, la única garantía que ofrecen estos *sockets* es que todo paquete recibido está entero (o dicho de otro modo, los paquetes que no llegan enteros se descartan). La comunicación a través de los *sockets* datagrama se estructura en paquetes independientes que se envían al receptor sin ningún tipo de confirmación de que el mensaje llegó sin contratiempos. Los *sockets* datagrama utilizan el protocolo de Internet UDP. Como se puede observar, UDP es no orientado a la conexión. Las funciones provistas para manejar este tipo de comunicación incluyen funciones para formar el paquete que se transmitirá, para transmitirlo y recibirlo, y para desempaquetarlo. A primera vista, este tipo de comunicación puede parecer poco adecuado. Sin embargo, como veremos, es útil para cierto tipo de aplicaciones orientadas a mensajes. Su ventaja es la rapidez, al no tener que establecer conexión inicial alguna.

Sockets *raw*. Este tipo de *sockets* no son interpretados por el nivel de transporte, sino que son una ventana directa al nivel de red (IP). Su uso está restringido a cuestiones de administración del nivel de red, como el protocolo ICMP, que, entre otras cosas, ofrece la facilidad *ping*. No veremos aquí este tipo de *sockets*.

La elección de un tipo u otro dependerá, como siempre, de los requisitos de la aplicación.

Programación con *sockets*

La comunicación a través de *sockets* utiliza el paradigma cliente/servidor. En este modelo, ambos procesos que se comunican juegan papeles distintos en la comunicación. Uno de ellos, llamado **servidor** ofrece un conjunto de "servicios" a los que se puede acceder estableciendo una conexión con él. Los **clientes** se conectan a los servidores para hacer uso de los servicios que desean.

Nótese que el concepto de "servicio" es muy genérico (puede representar desde obtener una página web o la hora del día hasta insertar un documento en una base de datos). Los *sockets* soportan este concepto genérico debido, precisamente, a que nos ofrecen un canal de comunicación por el que el usuario del *socket* puede enviar virtualmente *cualquier* cosa.

En cuanto a los *sockets*, cada una de las entidades pares actuará asumiendo uno de los dos roles. Así, en una comunicación siempre habrá un servidor y un cliente. Como veremos a continuación, tanto el funcionamiento como el conjunto de pasos a seguir en servidor y en cliente son distintos: el servidor establece un punto de comunicación identificado convenientemente y espera a que los clientes requieran sus servicios. Los clientes deben conocer de antemano la "dirección" del servicio para así poder solicitarlo.

Independientemente del tipo, cada *socket* tiene un nombre. En el caso de la familia UNIX, este nombre corresponde a un nombre de fichero. En el caso de la familia Internet, el nombre depende de la red subyacente. La familia Internet se asienta sobre la pila de protocolos de red TCP/IP. La estructuración de la pila de protocolos TCP/IP requiere que en cada nivel, cada entidad tenga un nombre único (figura 2).

Conceptualmente, cada nivel se comunica con el mismo nivel del otro *host*. Las líneas punteadas indican esta comunicación. Los *sockets* son un servicio ofrecido por el nivel de transporte a las aplicaciones, el nivel superior. Por lo tanto, el nombre de un *socket* está compuesto por tres partes:

- La dirección de Red (IP): es un entero de 32 bits que se representa como cuatro números decimales separados por puntos, como en 155.54.204.49, o, a través del DNS (*Domain Name System*), un nombre que lo identifica, como por ejemplo, ditec.um.es.
- la dirección de Transporte: es un número de 16 bits que identifica los distintos servicios que están disponibles en una máquina. A este número se le llama *puerto*.
- y el protocolo de transporte utilizado: TCP si queremos utilizar *sockets stream*, UDP si queremos utilizar *sockets* datagrama, etc.

Existen una serie de puertos cuya numeración se ha estandarizado, estos se llaman *well-known ports* (puertos bien conocidos, números del 0 al 1023). Estos puertos son utilizados por las aplicaciones servidor para facilitar que los clientes encuentren los servicios que esperan. Por ejemplo, el servicio HTTP reside en el puerto 80 (decimal) a través de TCP. Dada una dirección IP de un *host* del que queremos acceder al servicio HTTP, basta con conectar un *socket* TCP al puerto 80 del *host*. Una manera sencilla de ver el conjunto de puertos bien conocidos es listar el fichero `/etc/services`.

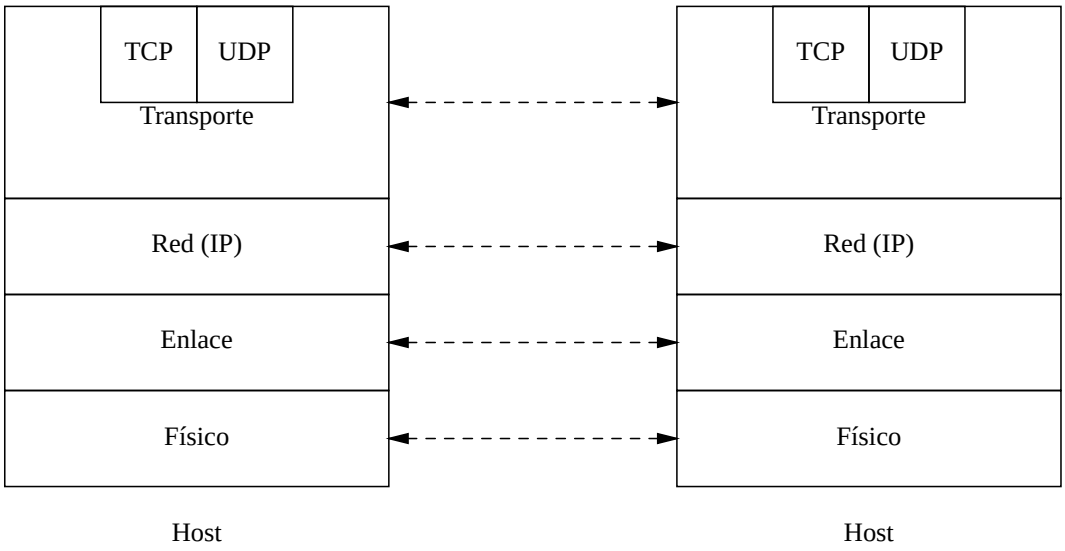

Figura 2: Pila de protocolos TCP/IP en dos *hosts* que se comunican.

Un escenario de comunicación utilizando *sockets*

El conjunto de llamadas al sistema que veremos en la siguiente sección incluye funciones específicas para ayudar a que una aplicación utilice sus *sockets* como cliente o como servidor. A continuación se muestra una interacción típica utilizando *sockets*: se establece un diálogo entre cliente y servidor en donde el primero pide al segundo cierta información. Los pasos a seguir son los siguientes (figura 3):

1. **Crear ambos *sockets* para la comunicación**. Ambos procesos (cliente y servidor) deben crear su *socket* correspondiente para poder establecer la comunicación. Para esto se utiliza la llamada al sistema `socket`.

2. **El servidor debe establecer el puerto de servicio**. Esto lo consigue el servidor llamando a la función `bind` que hace que éste tenga un nombre único en la red.

3. **Estar alerta de las conexiones de los clientes**. El servidor debe indicar, en el modo de *socket stream*, que su *socket* quedará alerta de las posibles conexiones de los clientes. Esto lo consigue con la función `listen`.

4. **Conectar con el servidor**. El cliente debe entonces, gracias a la función `connect`, conectar con el servidor, especificando la dirección del mismo.

5. **Aceptar la conexión**. El servidor ha quedado bloqueado por la función `accept` esperando conexiones de los clientes. Llegados a este punto, tenemos dos posibilidades:
   - que el servidor cree un nuevo proceso o hilo (*thread*) para que atienda la nueva conexión. Tras ello, el programa original volverá a ejecutar la función `accept` para esperar nuevas conexiones. En este caso se dice que tenemos un *servidor concurrente*.
   - que sea el propio servidor el que atienda la nueva conexión. En este caso se dice que tenemos un *servidor iterativo*.

6. **Comenzar la negociación**. Aquí es donde realmente se produce el intercambio de información entre cliente y servidor. Esta parte depende de la aplicación específica. Se utilizarán funciones del tipo `read` y `write`, como con los ficheros convencionales.

7. **Cerrar los *sockets***. Una vez que se ha terminado la comunicación, ambas partes cierran sus *sockets*, utilizando la función `close`. Nótese, sin embargo, que el *socket* original del servidor sigue todavía activo esperando nuevas conexiones.

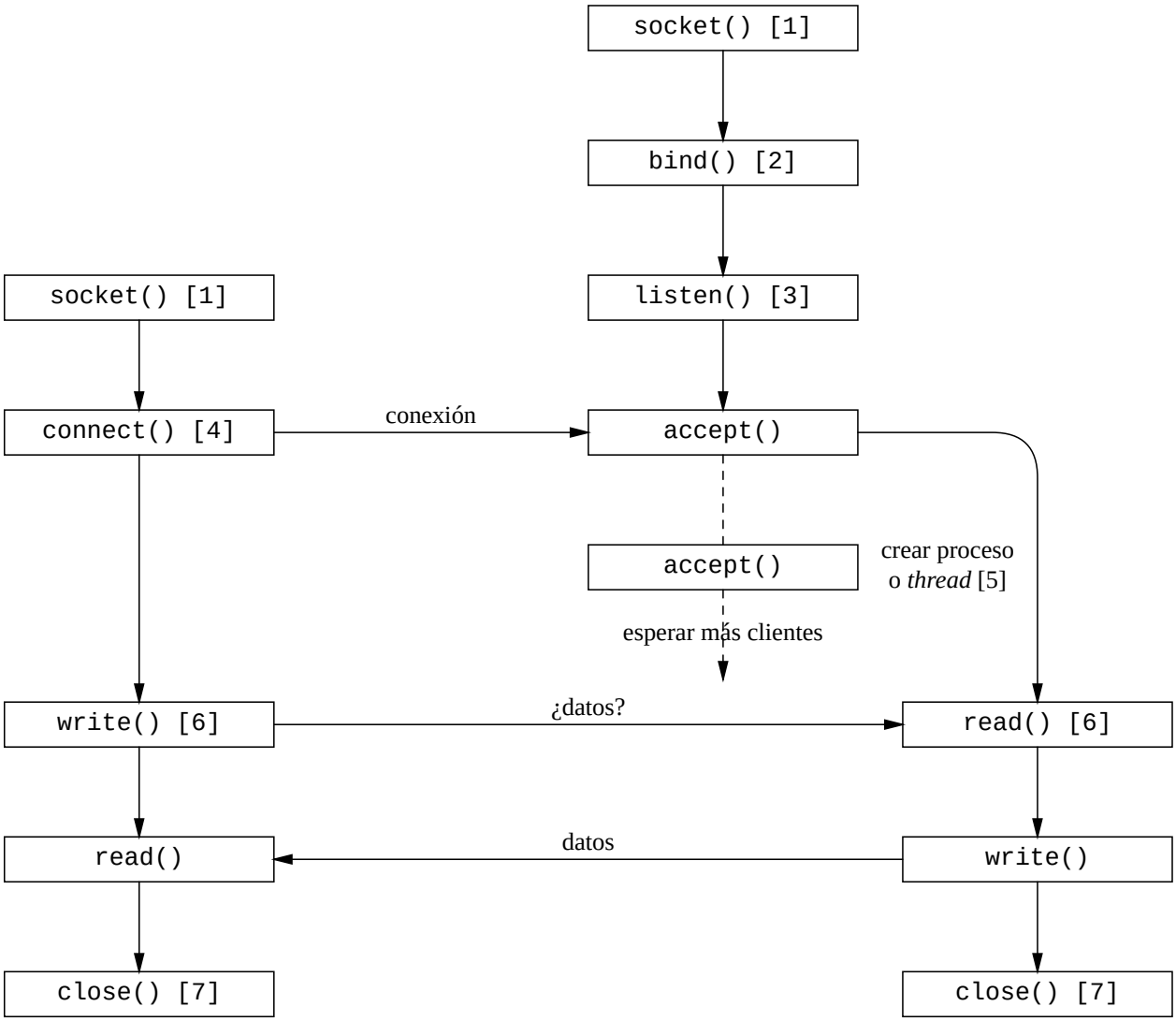

Figura 3: Una conversación utilizando *sockets stream* entre cliente (a la izquierda) y servidor concurrente (a la derecha).

Esta interacción corresponde a *sockets stream*. Los *sockets* datagrama son muy similares. Sin embargo, como no se tiene que realizar una conexión inicial, el servidor no tiene por qué ejecutar ni el `listen` ni el `accept`. El proceso se resume a continuación (figura 4):

1. **Crear ambos *sockets* para la comunicación**. Ambos procesos (cliente y servidor) deben crear su *socket* correspondiente para poder establecer la comunicación. Al igual que antes, para esto se utiliza la llamada al sistema `socket`.

2. **Tanto el servidor como el cliente eligen un puerto donde escuchar al otro**. Esto lo consiguen llamando a la función `bind`. Como la comunicación aquí no se realiza a través de una conexión, sino que cada comunicación es independiente, tanto el servidor como el cliente deben estar localizados en un puerto

específico. El servidor, por su parte, se liga con un puerto conocido por el cliente; el cliente debe especificar, con otra llamada a `bind`, un puerto arbitrario o bien 0 (en este caso el sistema lo elegirá), ya que el servidor, al recibir la petición del cliente, sabe en qué puerto y dirección IP está localizado el cliente.

3. **El cliente envía una petición y el servidor la recibe**. El cliente, a través de la llamada `sendto`, es capaz de enviar un mensaje arbitrario de petición al servidor, que a su vez será capaz de recogerlo a través de la llamada `recvfrom`. El servidor recibe de `recvfrom` la dirección IP y el puerto en donde está localizado el cliente.

4. **El servidor envía la respuesta y el cliente la recibe**. Una vez el servidor ha procesado la petición del cliente y ha generado alguna respuesta, la envía al cliente. Nótese que ahora los roles de cliente y servidor se invierten, ya que ahora el servidor utiliza la función que antes utilizaba el cliente y viceversa.

5. **El cliente cierra la comunicación**. Cuando el cliente ha recibido la respuesta (ha obtenido los datos o el servicio que deseaba), cierra unilateralmente la comunicación utilizando la llamada `close`. Nótese que esto sólo cierra la parte del cliente. El servidor puede seguir sirviendo a otros clientes.

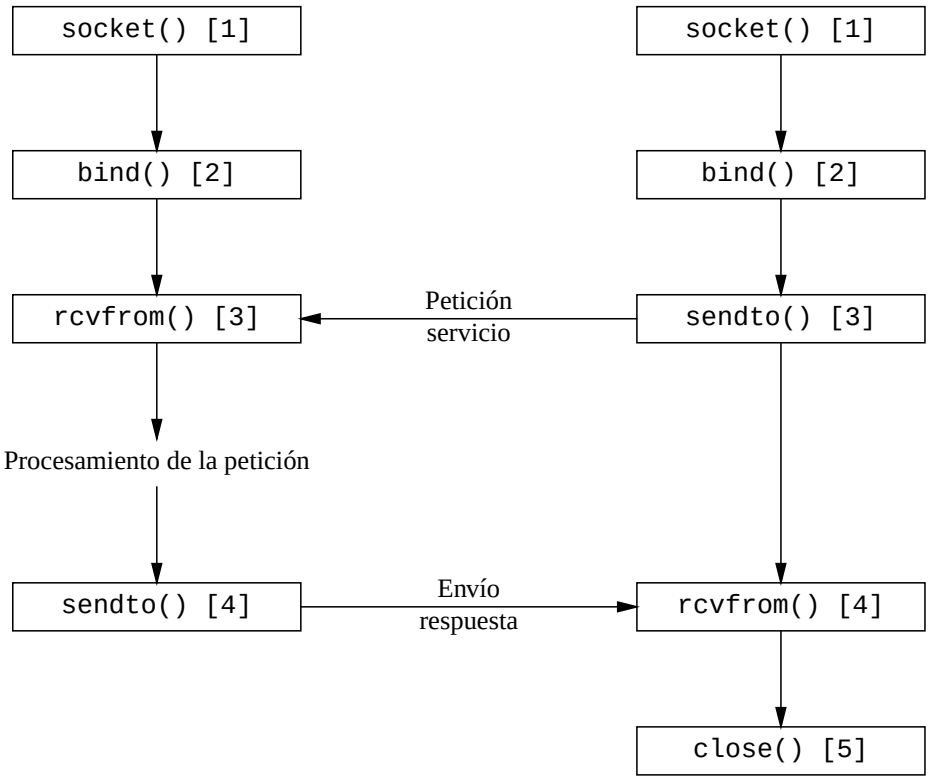

Figura 4: Una conversación utilizando *sockets* datagrama entre cliente (a la derecha) y servidor iterativo (a la izquierda).

3. Llamadas al sistema para el manejo de *sockets*

A continuación se muestra el interfaz que se ofrece al programador para la utilización de los *sockets*. Como de costumbre, este interfaz se ofrece a través de funciones y estructuras de datos del lenguaje C.

Direcciones de red

Como dijimos, cada *socket* tiene un nombre. El nombre depende de la familia a la que el *socket* pertenezca (que a su vez condiciona el uso que queremos hacer del mismo).

La dirección de un *socket* (para todas las familias) se especifica con la estructura `sockaddr`, presente en el fichero `<sys/socket.h>`:

```
/* En <sys/socket.h> */
struct sockaddr
{
    u_short sa_family;      /* Familia de sockets. Se emplean
                            constantes de la forma "AF_xxx" que
                            veremos después */
    char    sa_data[14];    /* Espacio reservado para la dirección.
                            Su significado depende de la familia */
};
```

El contenido de `sa_data` depende de la familia. Así, si utilizamos la familia Internet, las direcciones de red estarán definidas por la estructura `sockaddr_in` del fichero `<netinet/in.h>`:

```
/* En <netinet/in.h> */
struct in_addr
{
    u_long s_addr;  /* 32 bits que contienen la dirección IP */
};

struct sockaddr_in
{
    u_short sin_family;         /* Familia: en este caso AF_INET */
    u_short sin_port;           /* 16 bits: número de puerto */
    struct  in_addr sin_addr;   /* Dirección IP */
    char    sin_zero[8];        /* Bytes a 0 (no usados) */
};
```

El campo `sin_family` es equivalente a `sa_family`, y los campos `sin_port` y `sin_addr` especifican una dirección completa de un *socket* de una aplicación.

En cuanto a la familia UNIX, la estructura que se emplea está en el fichero `<sys/un.h>`:

```
/* En <sys/un.h> */
struct sockaddr_un
{
    u_short sun_family;         /* Familia: en este caso AF_UNIX */
    char    sun_path[108];      /* Ruta */
};
```

Las direcciones son en realidad rutas de ficheros. Los *sockets* UNIX equivalen a tuberías nombradas (*named pipes*) bidireccionales.

Como curiosidad, muchos programas de uso común (como el sistema X-Window) utilizan *sockets* UNIX para la comunicación con el resto de aplicaciones. Por ejemplo, haciendo un listado del directorio `/tmp/.font-unix`, correspondiente al servidor de fuentes del sistema X-Window, vemos lo siguiente:

```
# ls -l /tmp/.font-unix/
total 0
srwxrwxrwx   1 xfs      xfs             0 Feb 26 17:52 fs-1
```

la “s” inicial indica “socket”.

Creación de un *socket*: `socket`

La llamada de creación de un *socket*, `socket`, se define como sigue:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int af, int type, int protocol);
```

`af` especifica la familia, que puede ser, como hemos visto, `AF_UNIX` ó `AF_INET`. El campo `type` especificará el tipo del *socket*: *stream* ó datagrama, respectivamente, `SOCK_STREAM` y `SOCK_DGRAM`. `protocol` especifica el protocolo específico. Para nuestras necesidades, siempre será 0: protocolo por defecto dependiendo del tipo de *socket* (TCP para *stream* y UDP para datagrama).

Como resultado, la llamada devolverá un descriptor de *socket* como un entero (muy parecido a un descriptor de fichero). Si hay un error, devuelve -1.

Ligar un *socket* a un puerto: `bind`

La llamada `bind` se utiliza para ligar a un *socket* con una dirección IP y un puerto específicos. Como vimos, se utiliza por el servidor para establecer su punto de escucha si utilizamos *sockets stream* o por ambos, cliente y servidor, si utilizamos *sockets* datagrama. Su declaración es como sigue:

```
#include <sys/socket.h>

/* Sólo para la familia AF_UNIX */
#include <sys/un.h>
/* Sólo para la familia AF_INET */
#include <netinet/in.h>

int bind(int sfd, const void* addr, int addrlen);
```

De esta definición se pueden extraer ciertas indicaciones importantes:

- `sfd` especifica un identificador de *socket* previamente abierto utilizando la llamada `socket`.
- `addr` es un puntero sin tipo a una zona de memoria que guarda una estructura *del tipo* `struct sockaddr`. Como vimos, hay dos, una para UNIX y otra para Internet; respectivamente, `sockaddr_un` y `sockaddr_in`.
- `addrlen` indica la longitud de los datos introducidos en `addr`. Como vimos, distintas direcciones de red tienen distintos tamaños. Normalmente se utiliza el operador `sizeof` de C con la estructura adecuada a la familia de *sockets*.

Para la familia UNIX, se crea el fichero especificado en `sun_path`. Para la familia Internet, el número de puerto puede ser 0; en este caso el sistema le asigna uno (véase esta utilidad en las comunicaciones basadas en *sockets* datagrama en la página 6).

Escuchar un *socket*: `listen`

Los procesos que actúan con el rol de servidor deben indicarlo así en sus *sockets*. Para esto se utiliza la llamada `listen`:

```
#include <sys/socket.h>

int listen(int sfd, int backlog);
```

La llamada a `listen` habilita la cola de almacenamiento de peticiones de conexión. El *socket* debe ser de tipo *stream*. El parámetro `backlog` indica el tamaño de la cola (5 ó 6 es un valor aceptable). Devuelve 0 si no hubo errores y -1 en otro caso.

Conectar: `connect`

Para que un cliente conecte con un servidor, debe utilizar la llamada `connect`:

```
#include <sys/socket.h>

/* Sólo para la familia AF_UNIX */
#include <sys/un.h>
/* Sólo para la familia AF_INET */
#include <netinet/in.h>

int connect(int sfd, const void* addr, int addrlen);
```

Al igual que antes, `sfd` es un descriptor de *socket*. `addr` y `addrlen` especifican la dirección con la que se conectará (al contrario que en `bind`, en donde se especificaba la propia).

Si el *socket* es de tipo `SOCK_DGRAM`, esta llamada especifica la dirección a la que se enviarán los siguientes mensajes, aunque no se conectará con él.

Si el *socket* es de tipo `SOCK_STREAM`, `connect` intenta contactar con el ordenador remoto y así unir el *socket* especificado en `sfd` con el especificado en la dirección `addr`.

Si todo fue bien, devuelve 0; en caso contrario, -1.

Aceptación de una conexión: `accept`

Los procesos servidores aceptan peticiones de servicio gracias a la función `accept`:

```
#include <sys/socket.h>

int accept(int sfd, void* addr, int *addrlen);
```

Sólo se utiliza con *sockets* de tipo `SOCK_STREAM`. `sfd` corresponde a un descriptor de *socket* que se ha creado previamente con la llamada `socket` y se ha ligado a un puerto con la llamada `bind`. También se debe haber creado una cola de peticiones con la llamada `listen`.

`accept` devuelve tres datos de interés. El primero es `addr`. Esta variable debe apuntar en el momento de la llamada a una zona de memoria que `accept` escribirá con la dirección del cliente que requiere una conexión. Como complemento, también rellenará el valor de la variable `addrlen` con el tamaño de la dirección.

El tercer resultado devuelto por `accept` es el descriptor de un nuevo *socket* que se puede utilizar para hablar con el cliente. Este valor es el valor `int` devuelto por la función, que tendrá el valor -1 si hay un error. El *socket* original, `sfd` permanece abierto y puede atender a más conexiones. Sin embargo, el nuevo identificador de *socket* devuelto no puede aceptar más conexiones.

Envío y recepción de datos

La potencia de los *sockets* radica en que una vez establecidos, se pueden utilizar casi como si fueran ficheros. Esto se traduce en que, para los *sockets stream*, se pueden utilizar de forma normal las llamadas `read` y `write` tradicionales de ficheros con la misma semántica.

Para los *sockets* datagrama existen unas funciones especiales, ya que éstos están orientados a mensajes puntuales y no a un tráfico continuo. Ya vimos las llamadas en el esquema de funcionamiento de los *sockets* datagrama en la página 6. Éstas son `sendto` y `recvfrom`:

```
#include <sys/socket.h>

int recvfrom(int sfd, void *buf, int len, int flags, void *from, int *fromlen);

int sendto(int sfd, void *buf, int len, int flags, const void *to, int tolen);
```

Su semántica es sencilla, ya que el primer parámetro, `sfd`, al igual que en todas las llamadas hasta ahora representa al *socket* al que se va a enviar un mensaje. El mensaje está descrito por dos parámetros: un puntero a una zona de memoria, `buf`, de una longitud `len` bytes. `flags` es normalmente 0.

Las direcciones `from`/`fromlen` y `to`/`tolen` especifican, respectivamente, de qué dirección y puerto se recibe y a qué dirección enviar datos.

Cierre del canal

La llamada normal `close` se puede utilizar cuando una aplicación ya no necesita más un *socket*. Otra función, `shutdown`, se puede utilizar para llevar un control más fino del cierre de la comunicación:

```
#include <sys/socket.h>

int shutdown(int sfd, int how);
```

Dependiendo del valor de `how`, se cierra el *socket*:

- 0: No se puede recibir nada más por el *socket*.
- 1: No se puede enviar nada más por el *socket*.
- 2: No se puede ni enviar ni recibir nada más por el *socket*.

Network Byte Order

Cuando se trabaja con una red (*sockets* de tipo Internet), es posible que estén conectados a la misma ordenadores con distinta ordenación de bytes (*big-endian*, *little-endian*). Por ello, cosas como las direcciones deben codificarse de una forma común en todas las arquitecturas. Para ello, se ofrecen un conjunto de llamadas que permiten transformar números en formato de la máquina (*host* en terminología de red) al formato común de red y viceversa:

```
#include <sys/types.h>
#include <netinet/in.h>

unsigned long htonl (unsigned long hostlong);

unsigned short htons (unsigned short hostshort);

unsigned long ntohl (unsigned long netlong);

unsigned short ntohs (unsigned short netshort);
```

Las distintas llamadas realizan lo siguiente:

- `htonl` convierte un `unsigned long` del formato que maneja el ordenador al formato de la red.
- `htons` convierte un `unsigned short` del formato que maneja el ordenador al formato de la red.
- `ntohl` convierte un `unsigned long` del formato que maneja la red al formato que maneja el ordenador.
- `ntohs` convierte un `unsigned short` del formato que maneja la red al formato que maneja el ordenador.

Manejo de direcciones IP

Existen dos funciones que nos permiten trabajar de forma más cómoda con direcciones en formato Internet. Así, permiten pasar de notación punto en una cadena (por ejemplo, `"155.54.12.241"` a formato de `unsigned long`, en este caso `0x9b360cf1`) y viceversa. Estas funciones son `inet_addr` e `inet_ntoa`:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr (const char* cp);

char *inet_ntoa(struct in_addr in);
```

La primera de ellas, `inet_addr` convierte una cadena en formato punto a un `unsigned long` ya preparado para la red. La siguiente ejecuta el paso contrario. Estas funciones ayudan al rellenar correctamente la estructura `struct sockaddr_in`. He aquí un ejemplo:

```
/* La dirección IP */
#define DIR_IP "155.54.12.241"
/* El puerto: HTTP */
#define PUERTO 80

/* La dirección remota */
struct sockaddr_in r_addr;

r_addr.sin_family = AF_INET;                    /* Familia Internet */
r_addr.sin_addr.s_addr = inet_addr( DIR_IP );   /* Dirección IP */
r_addr.sin_port = htons( PUERTO );              /* Puerto */
```

Aquí, `r_addr` representa una dirección remota a la que querremos conectar o enviar un mensaje con `sendto`. Se especifica la familia `AF_INET`, la estructura `struct in_addr sin_addr` (página 8) se rellena con la llamada a `inet_addr` y utilizando la dirección IP remota deseada[1]. Finalmente, la dirección del puerto se escribe en `sin_port` utilizando la ya comentada función `htons`.

Si por el contrario lo que queremos es saber la dirección IP de un ordenador del que nos han dado su nombre (por ejemplo «google.com»), tendremos que usar la función `gethostbyname`:

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

Que retorna una estructura de tipo `hostent`:

```
struct hostent {
  char    *h_name;        /* official name of host */
  char    **h_aliases;    /* alias list */
  int     h_addrtype;     /* host address type */
  int     h_length;       /* length of address */
  char    **h_addr_list;  /* list of addresses */
};
#define h_addr  h_addr_list[0]  /* for backward compatibility */
```

[1] Para la dirección propia que se utilizará con `bind`, se puede utilizar la constante `INADDR_ANY` para indicar la dirección IP propia del *host* donde se está ejecutando el programa.

De todos los campos, el más útil para conseguir la dirección IP del ordenador remoto es el campo `h_addr` (definido como el primer campo de `h_addr_list`. Así, el código anterior para construir la dirección usando `gethostbyname` queda:

```
/* El nombre del ordenador */
#define NOMBRE "google.com"
/* El puerto: HTTP */
#define PUERTO 80

/* Estructura hostent */
struct hostent* he;

/* La dirección remota */
struct sockaddr_in r_addr;

he = gethostbyname (NOMBRE);

r_addr.sin_family = AF_INET;                        /* Familia Internet */
r_addr.sin_addr = *((struct in_addr*)he->h_addr);   /* Dirección IP */
r_addr.sin_port = htons( PUERTO );                  /* Puerto */
```

4. Resumen

Este apartado recoge los conceptos más importantes presentados en este boletín, que deberían estar perfectamente claros al finalizar la sesión de prácticas.

- Concepto de comunicación entre procesos.
- Pila de protocolos TCP/IP.
- Diferencias entre TCP y UDP.
- Concepto de Socket.
- Familias de *sockets*: UNIX e Internet.
- Tipos de *sockets*: *stream* (TCP), datagrama (UDP) y *raw*.
- Concepto de puerto.
- *Well-known ports*: puertos 0 a 1023 reservados para los servicios más comunes.
- Fichero `/etc/services`: asociaciones estándar entre puertos y servicios.
- Direccionamiento de *sockets*: dirección IP + puerto + protocolo de transporte.
- Mecanismo de traducción de direcciones (DNS).
- Utilización de las llamadas al sistema para el manejo de *sockets stream* (programas `cliente_tcp.c` y `servidor_tpc.c`) y *sockets datagrama* (programas `cliente_udp.c` y `servidor_udp.c`).
- Ordenación de bytes *network byte order* (red) y *host byte order* (máquina).
- Conversión de una dirección IP en una cadena de caracteres y viceversa.

5. Ejercicios propuestos

1. Modifica los programas `cliente_tcp.c` y `cliente_udp.c` para poder asignar el *socket* del cliente a un puerto local preestablecido.
2. Modifica el programa `cliente_tcp.c` para que:
   a) Se conecte al puerto 80 del servidor.
   b) Al establecer la conexión, envíe la cadena:

   ```
   GET /index.html HTTP/1.0\n\n
   ```

   Imprima lo que recibe del servidor que deberá ser una página web. Por ejemplo, el cliente se puede probar como:

   ```
   ./cliente_tcp 155.54.212.103
   ```

   En este caso, el cliente intenta obtener el fichero `index.html` de `www.um.es` (`155.54.212.103`).
3. Modifica los programas `cliente_tcp.c` y `servidor_tcp.c` para garantizar que las operaciones de envío/recepción de un paquete del protocolo se completan con normalidad.

6. Programas de ejemplo

Esta sección incluye un ejemplo de uso de *sockets* TCP, `servidor_tcp.c` y `cliente_tcp.c`, y otro de *sockets* UDP, `servidor_udp.c` y `cliente_udp`. Las figuras 5 y 6 corresponden a los diagramas de flujo de los ejemplos TCP y UDP, respectivamente. El ejemplo `getmyip.c` imprime la dirección IP y el nombre del *host*.

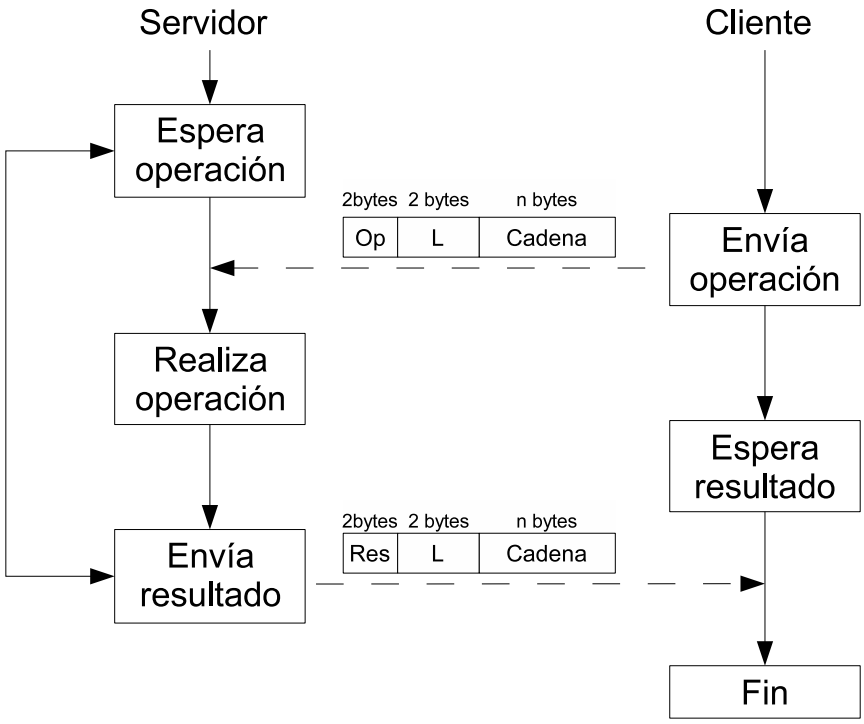

Figura 5: Diagrama de flujo de `servidor_tcp.c` y `cliente_tcp.c`.

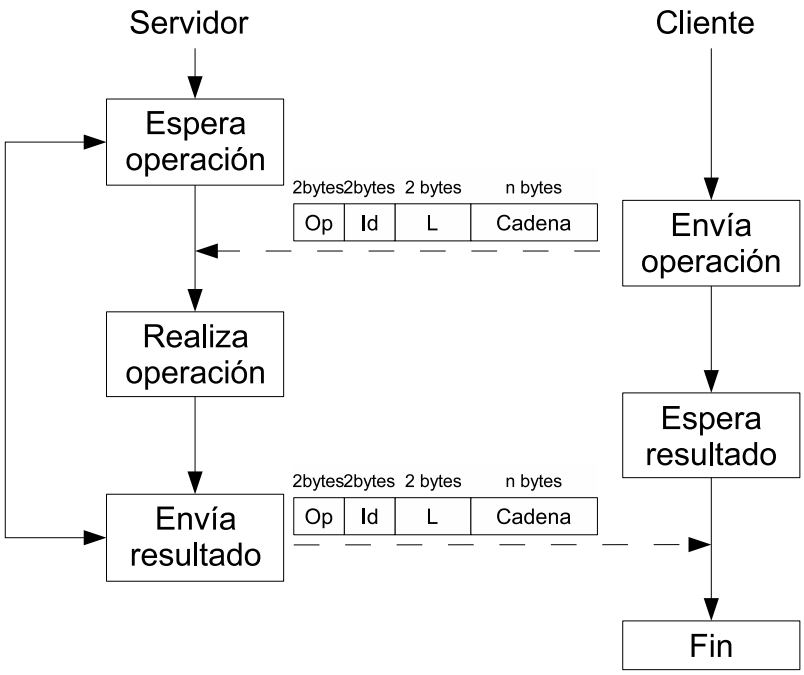

Figura 6: Diagrama de flujo de `servidor_udp.c` y `cliente_udp.c`.

6.1. common.h

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <pthread.h>
#include <arpa/inet.h>
#include <ctype.h>

#define PUERTO 3490    /* puerto en el servidor */

#define BACKLOG 10     /* numero maximo de conexiones pendientes en cola */

#define MAXDATASIZE 256 /* maximo numero de bytes que podemos recibir */

#define HEADER_LEN (sizeof(unsigned short) * 2)

/* formato de la unidad de datos de aplicacion para Stream*/
struct appdata
{
        unsigned short op;                      /* codigo de operacion */
        unsigned short len;                     /* longitud de datos */
        char data[MAXDATASIZE - HEADER_LEN]; /* datos */
};

#define ID_HEADER_LEN (sizeof (unsigned short) * 3)
```

```
/* formato de la unidad de datos de aplicacion para Datagramas*/
struct idappdata
{
        unsigned short op;                      /* codigo de operacion */
        unsigned short id;                      /* identificador */
        unsigned short len;                     /* longitud de datos */
        char data[MAXDATASIZE - ID_HEADER_LEN]; /* datos */
};

/* codigos de operacion (appdata.op) */
#define OP_MAYUSCULAS       0x0001 /* mayusculas */
#define OP_MINUSCULAS       0x0002 /* minusculas */
#define OP_RESULTADO        0x1000 /* resultado */
#define OP_ERROR            0xFFFF /* error */
```

6.2. cliente_tcp.c

```
/* FICHERO: cliente_tcp.c
 * DESCRIPCION: codigo del cliente con sockets stream */

#include "common.h"

#define PUERTO_REMOTO PUERTO /* puerto remoto en el servidor al que conecta el cliente */

int main (int argc, char *argv[])
{
        int sockfd;                     /* conexion sobre sockfd */
        char buf[MAXDATASIZE];          /* buffer de recepcion */
        struct sockaddr_in their_addr;  /* informacion de la direccion del servidor */
        struct appdata operation;       /* mensaje de operacion enviado */
        struct appdata resultado;       /* mensaje de respuesta recibido */
        int numbytes;                   /* numero de bytes recibidos o enviados */
        int len;

        /* obtiene parametros */
        if (argc != 2)
        {
                fprintf (stderr, "uso: cliente hostname\n");
                exit (1);
        }

        /* crea el socket */
        if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) == -1)
        {
                perror ("socket");
                exit (1);
        }

        their_addr.sin_family = AF_INET; /* Familia: ordenacion de bytes de la maquina */
        their_addr.sin_port = htons (PUERTO_REMOTO);    /* Puerto: ordenacion de bytes de la red */
        their_addr.sin_addr.s_addr =                    /* IP: ordenacion de bytes de la red */
                inet_addr ( argv[1] );
        memset (&(their_addr.sin_zero), '\0', 8); /* Pone a cero el resto de la estructura */

        memset (buf, '\0', MAXDATASIZE); /* Pone a cero el buffer inicialmente */

        /* conecta con el servidor */
        if (connect (sockfd, (void*)&their_addr, sizeof (struct sockaddr_in)) == -1)
        {
                perror ("connect");
                exit (1);
        }
```

```
        printf ("(cliente) conexion establecida con servidor "
                "[nombre%s IP%s puerto remoto%d]\n",
                argv[1], inet_ntoa(their_addr.sin_addr), ntohs(their_addr.sin_port));

        /* envia mensaje de operacion al servidor */
        operation.op = htons(OP_MINUSCULAS); /* op */
        strcpy(operation.data, "Esta es Una PRUEBA"); /* data */
        len = strlen (operation.data);
        operation.len = htons(len);        /* len */
        if ((numbytes = write (sockfd, (char *) &operation, len + HEADER_LEN)) == -1)
                perror ("write");
        else
                printf ("(cliente) mensaje enviado al servidor [longitud%d]\n", numbytes);

        printf ("(cliente) operacion solicitada [op 0x%x longitud%d contenido%s]\n",
                ntohs(operation.op), len, operation.data);

        /* espera resultado de la operacion */
        if ((numbytes = read (sockfd, buf, HEADER_LEN)) == -1) /* leemos tipo de respuesta y la longitud */
        {
                perror ("read");
                exit (1);
        }
        if (numbytes != HEADER_LEN) /* comprueba el número de bytes recibidos */
        {
                printf ("(cliente) cabecera de la unidad de datos recibida de manera incompleta "
                        "[longitud esperada%d longitud recibida%d]",
                        HEADER_LEN, numbytes);
                exit (1);
        }

        /* tenemos el tipo de respuesta y la longitud */
        resultado.op = ntohs((*((unsigned short *)(buf))));
        resultado.len = ntohs((*((unsigned short *)(buf + sizeof(unsigned short)))));
        memset (resultado.data, '\0', MAXDATASIZE - HEADER_LEN);

        if ((numbytes = read (sockfd, resultado.data, resultado.len)) == -1) /* leemos los datos */
        {
                perror ("read");
                exit (1);
        }
        printf ("(cliente) mensaje recibido del servidor [longitud%d]\n", numbytes + HEADER_LEN);

        if (numbytes != resultado.len) /* comprueba el número de bytes recibidos */
                printf ("(cliente) datos de la unidad de datos recibida de manera incompleta "
                        "[longitud esperada%d longitud recibida%d]",
                        resultado.len, numbytes);
        else
                printf ("(cliente) resultado de la operacion solicitada"
                        "[res 0x%x longitud%d contenido%s]\n",
                        resultado.op, resultado.len, resultado.data);

        /* cierra el socket */
        close (sockfd);
        printf ("(cliente) conexion cerrada con servidor\n");

        return 0;
}
```

6.3. servidor_tcp.c

```
/* FICHERO: servidor_tcp.c
 * DESCRIPCION: codigo del servidor con sockets stream */

#include "common.h"

#define PUERTO_LOCAL PUERTO /* puerto local en el servidor al que se conectan los clientes */

int main (int argc, char* argv[])
{
        int sockfd;                             /* escucha sobre sock_fd */
        int new_fd;                             /* nueva conexion sobre new_fd */
        struct sockaddr_in my_addr;             /* informacion de mi direccion */
        struct sockaddr_in their_addr;          /* informacion de la direccion del cliente */
        char buf[MAXDATASIZE];                  /* buffer de recepcion */
        int numbytes;                           /* numero de bytes enviados o recibidos */
        struct appdata operation;               /* mensaje de operacion recibido */
        struct appdata resultado;               /* mensaje de respuesta enviado */
        int error;                              /* indica la existencia de un error */
        int cont;
        int len;
        size_t sin_size;

        /* crea el socket */
        if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) == -1)
        {
                perror ("socket");
                exit (1);
        }

        my_addr.sin_family = AF_INET;  /* Familia: ordenacion de bytes de la maquina */
        my_addr.sin_port = htons (PUERTO_LOCAL);/* Puerto: ordenacion de bytes de la red */
        my_addr.sin_addr.s_addr = INADDR_ANY;  /* IP: ordenacion de bytes de la red */
        memset (&(my_addr.sin_zero), '\0', 8);     /* Pone a cero el resto de la estructura */

        /* asigna el socket a un puerto local */
        if (bind (sockfd, (void*)&my_addr, sizeof (struct sockaddr_in)) == -1)
        {
                perror ("bind");
                exit (1);
        }

        /* escucha peticiones de conexion */
        if (listen (sockfd, BACKLOG) == -1)
        {
                perror ("listen");
                exit (1);
        }

        /* accept() loop... */
        while (1)
        {
                printf ("(servidor) escuchando peticiones de conexion [puerto local%d]\n",
                        ntohs (my_addr.sin_port));

                /* acepta peticion de conexion de un cliente */
                sin_size = sizeof (struct sockaddr_in);
                if ((new_fd = accept (sockfd, (void*)&their_addr, &sin_size)) == -1)
                {
                        perror ("accept");
                        continue;
                }
                printf ("(servidor) conexion establecida desde cliente "
                        "[IP%s puerto remoto%d]\n",
                        inet_ntoa (their_addr.sin_addr), ntohs (their_addr.sin_port));
```

```
/* espera mensaje de operacion del cliente */
memset (buf, '\0', MAXDATASIZE);       /* Pone a cero el buffer inicialmente */
if ((numbytes = read (new_fd, buf, HEADER_LEN)) == -1)
{
        perror ("read");
        continue;
}
if (numbytes != HEADER_LEN) /* comprueba el número de bytes recibidos */
{
        printf ("(servidor) cabecera de la unidad de datos recibida de manera incompleta "
                "[longitud esperada%d longitud recibida%d]",
                HEADER_LEN, numbytes);
        continue;
}

/* tenemos el tipo de operacion y la longitud */
operation.op = ntohs((*((unsigned short *)(buf))));
operation.len = ntohs((*((unsigned short *)(buf + sizeof(unsigned short)))));
memset (operation.data, '\0', MAXDATASIZE - HEADER_LEN);

if ((numbytes = read (new_fd, operation.data, operation.len)) == -1)
{
        perror ("read");
        continue;
}
printf ("(servidor) mensaje recibido del cliente [longitud%d]\n",
        numbytes + HEADER_LEN);

if (numbytes != operation.len) /* comprueba el número de bytes recibidos */
{
        printf ("(servidor) datos de la unidad de datos recibida de manera incompleta "
                "[longitud esperada%d longitud recibida%d]",
                operation.len, numbytes);
        continue;
}
else
        printf ("(servidor) operacion solicitada [op 0x%x longitud%d contenido%s]\n",
                operation.op, operation.len, operation.data);

/* realiza operacion solicitada por el cliente */
memset (resultado.data, '\0', MAXDATASIZE - HEADER_LEN); error = 0;
switch (operation.op)
{
case OP_MINUSCULAS: /* minusculas */
        resultado.op = htons(OP_RESULTADO); /* op */
        for(cont = 0; cont < operation.len; cont++){ /* data */
                if (isupper(operation.data[cont]))
                        resultado.data[cont] = tolower(operation.data[cont]);
                else
                        resultado.data[cont] = operation.data[cont];
        }
        len = cont;
        resultado.len = htons(len); /* len */
        break;
case OP_MAYUSCULAS: /* mayusculas */
        resultado.op = htons(OP_RESULTADO); /* op */
        for(cont = 0; cont < operation.len; cont++){ /* data */
                if (islower(operation.data[cont]))
                        resultado.data[cont] = toupper(operation.data[cont]);
                else
                        resultado.data[cont] = operation.data[cont];
        }
        len = cont;
        resultado.len = htons(len); /* len */
        break;
```

```
                default: /* operacion desconocida */
                        resultado.op = htons(OP_ERROR); /* op */
                        strcpy(resultado.data, "Operacion desconocida"); /* data */
                        len = strlen (resultado.data);
                        resultado.len = htons(len);     /* len */
                        error = 1;
                        break;
                }

                /* envia resultado de la operacion solicitada por el cliente */
                if ((numbytes = write (new_fd, (char *) &resultado, len + HEADER_LEN)) == -1)
                {
                        perror ("write");
                        continue;
                }
                else
                        printf ("(servidor) mensaje enviado al cliente [longitud%d]\n", numbytes);

                printf ("(servidor) resultado de la operacion solicitada"
                        "[res 0x%x longitud%d contenido%s]\n",
                        ntohs(resultado.op), len, resultado.data);

                /* cierra socket */
                close (new_fd);

                printf ("(servidor) conexion cerrada con cliente\n");
        }
        /* cierra socket (no se ejecuta nunca) */
        close(sockfd);

        return 0;
}
```

6.4. cliente_udp.c

```
/* FICHERO: cliente_udp.c
 * DESCRIPCION: codigo del cliente con sockets datagrama */

#include "common.h"

#define PUERTO_REMOTO PUERTO /* puerto remoto en el servidor al que se envian los mensajes */

int main (int argc, char *argv[])
{
        int sockfd;
        struct sockaddr_in their_addr;  /* informacion de la direccion del servidor */
        char buf[MAXDATASIZE];          /* buffer de recepcion */
        struct idappdata operation;     /* mensaje de operacion enviado */
        struct idappdata *resultado;    /* mensaje de respuesta recibido */
        int numbytes;                   /* numero de bytes recibidos o enviados */
        size_t sin_size;

        if (argc != 2)
        {
                fprintf (stderr, "uso: cliente hostname\n");
                exit (1);
        }

        /* crea el socket */
        if ((sockfd = socket (AF_INET, SOCK_DGRAM, 0)) == -1)
        {
                perror ("socket");
                exit (1);
        }
```

```
        their_addr.sin_family = AF_INET;      /* Familia: ordenacion de bytes de la maquina */
        their_addr.sin_port =
                htons (PUERTO_REMOTO); /* Puerto: ordenacion de bytes de la red */
        their_addr.sin_addr.s_addr =                    /* IP: ordenacion de bytes de la red */
             inet_addr (argv[1]);
        memset (&(their_addr.sin_zero), 0, 8);          /* Pone a cero el resto de la estructura */

        memset (buf, '\0', MAXDATASIZE); /* Pone a cero el buffer inicialmente */

        /* envia mensaje de operacion al servidor */
        operation.op = OP_MINUSCULAS; /* op */
        operation.id = 1; /* id */
        strcpy(operation.data, "Esta es Una PRUEBA");   /* data */
        operation.len = strlen (operation.data);            /* len */
        if ((numbytes = sendto (sockfd, (char *) &operation,
                                operation.len + ID_HEADER_LEN, 0,
                                (void*)&their_addr,
                                sizeof(struct sockaddr_in))) == -1)
        {
                perror ("send");
                exit (1);
        }
        else
                printf ("(cliente) mensaje enviado al servidor [longitud%d]\n", numbytes);

        printf ("(cliente) operacion solicitada [op 0x%x id%d longitud%d contenido%s]\n",
                operation.op, operation.id, operation.len, operation.data);

        /* espera resultado de la operacion */
        sin_size = sizeof(struct sockaddr_in);
        if ((numbytes = recvfrom (sockfd, buf, MAXDATASIZE, 0 /* flags */,
                                  (void*)&their_addr, &sin_size)) == -1)
        {
                perror ("recv");
                exit (1);
        }
        printf ("(cliente) mensaje recibido del servidor [longitud%d]\n", numbytes);


        resultado = (struct idappdata *) &buf;
        /* comprueba el número de bytes recibidos */
        if ((numbytes < ID_HEADER_LEN) || (numbytes != resultado->len + ID_HEADER_LEN))
                printf ("(cliente) unidad de datos recibida de manera incompleta \n");
        else
                if (resultado->id != operation.id)      /* comprueba el identificador del resultado */
                        printf ("(cliente) unidad de datos recibida con identificador erroneo \n");
                else
                        printf ("(cliente) resultado de la operacion solicitada "
                                "[res 0x%x id%d longitud%d contenido%s]\n",
                                resultado->op, resultado->id, resultado->len, resultado->data);

        /* cierra el socket */
        close (sockfd);

        return 0;
}
```

6.5. servidor_udp.c

```
/* FICHERO: servidor_udp.c
 * DESCRIPCION: codigo del servidor con sockets datagrama */

#include "common.h"

#define PUERTO_LOCAL PUERTO /* puerto local en el servidor por el que se reciben los mensajes */

int main (int argc, char* argv[])
{
        int sockfd;
        struct sockaddr_in my_addr;     /* informacion de mi direccion */
        struct sockaddr_in their_addr;  /* informacion de la direccion del cliente */
        char buf[MAXDATASIZE];          /* buffer de recepcion */
        int numbytes;                   /* numero de bytes recibidos o enviados */
        struct idappdata *operation;    /* mensaje de operacion recibido */
        struct idappdata resultado;     /* mensaje de respuesta enviado */
        int error;                      /* indica la existencia de un error */
        int cont;
        size_t sin_size;

        /* crea el socket */
        if ((sockfd = socket (AF_INET, SOCK_DGRAM, 0)) == -1)
        {
                perror ("socket");
                exit (1);
        }

        my_addr.sin_family = AF_INET;   /* Familia: ordenacion de bytes de la maquina */
        my_addr.sin_port = htons (PUERTO_LOCAL); /* Puerto: ordenacion de bytes de la red */
        my_addr.sin_addr.s_addr = INADDR_ANY;    /* IP: ordenacion de bytes de la red */
        memset (&(my_addr.sin_zero), '\0', 8);   /* Pone a cero el resto de la estructura */

        /* asigna el socket a un puerto local */
        if (bind (sockfd, (void*) &my_addr, sizeof (struct sockaddr_in)) == -1)
        {
                perror ("bind");
                exit (1);
        }

        /* recvfrom() loop... */
        while (1)
        {
                printf ("(servidor) esperando mensajes [puerto local%d]\n",
                        ntohs (my_addr.sin_port));

                /* recibe un mensaje de un cliente */
                memset (buf, '\0', MAXDATASIZE); /* Pone a cero el buffer inicialmente */
                sin_size = sizeof(struct sockaddr_in);
                if ((numbytes = recvfrom (sockfd, buf, MAXDATASIZE, 0,
                                        (void*)&their_addr, &sin_size))== -1)
                {
                        perror ("recvfrom");
                        continue;
                }
                printf("(servidor) mensaje recibido de IP%s puerto%d [longitud%d]\n",
                        inet_ntoa(their_addr.sin_addr), ntohs(their_addr.sin_port), numbytes);

                operation = (struct idappdata *) &buf;
                /* comprueba el número de bytes recibidos */
                if ((numbytes < ID_HEADER_LEN) || (numbytes != operation->len + ID_HEADER_LEN))
                {
                        printf ("(servidor) unidad de datos recibida de manera incompleta \n");
                        continue;
                }
```

```
        else
                printf ("(servidor) operacion solicitada [op 0x%x id%d longitud%d contenido%s]\n",
                        operation->op, operation->id, operation->len, operation->data);

        /* realiza operacion solicitada por el cliente */
        error = 0;
        resultado.id = operation->id; /* id */
        switch (operation->op)
        {
        case OP_MINUSCULAS: /* minusculas */
                resultado.op = OP_RESULTADO; /* op */
                for(cont = 0; cont < operation->len; cont++){ /* data */
                        if (isupper(operation->data[cont]))
                                resultado.data[cont] = tolower(operation->data[cont]);
                        else
                                resultado.data[cont] = operation->data[cont];
                }
                resultado.len = cont; /* len */
                break;
        case OP_MAYUSCULAS: /* mayusculas */
                resultado.op = OP_RESULTADO; /* op */
                for(cont = 0; cont < operation->len; cont++){ /* data */
                        if (islower(operation->data[cont]))
                                resultado.data[cont] = toupper(operation->data[cont]);
                        else
                                resultado.data[cont] = operation->data[cont];
                }
                resultado.len = cont; /* len */
                break;
        default: /* operacion desconocida */
                resultado.op = OP_ERROR; /* op */
                strcpy(resultado.data, "Operacion desconocida"); /* data */
                resultado.len = strlen (resultado.data);        /* len */
                error = 1;
                break;
        }

        /* envia resultado de la operacion solicitada por el cliente */
        if ((numbytes = sendto(sockfd, (char *) &resultado,
                                resultado.len + ID_HEADER_LEN, 0,
                                (void*)&their_addr,
                                sizeof(struct sockaddr_in))) == -1)
        {
                perror ("recv");
                continue;
        }
        else
                printf ("(servidor) mensaje enviado al cliente [longitud%d]\n", numbytes);

        printf ("(servidor) resultado de la operacion solicitada "
                "[res 0x%x id%d longitud%d contenido%s]\n",
                resultado.op, resultado.id, resultado.len, resultado.data);
}

/* cierra socket (no se ejecuta nunca) */
close (sockfd);

return 0;
}
```

6.6. getmyip.c

```
/* -*- mode: c; c-basic-offset: 8; -*- $Id: getmyip2.c 801 2005-09-08 00:39:46Z dsevilla $ */
/* FICHERO: getmyip3.c
 * DESCRIPCION: programa que imprime la direccion IP del host */

#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <linux/if.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <ifaddrs.h>    /* Linux, FreeBSD */

int main (int argc, char *argv[])
{
        struct ifaddrs* ifa;
        struct ifaddrs* ifa_tmp;

        if (0 == getifaddrs (&ifa))
        {
                ifa_tmp = ifa;
                do
                {
                        /* Bridges have no addr. */
                        if (ifa_tmp->ifa_addr &&
                            ifa_tmp->ifa_addr->sa_family == AF_INET)
                        {
                                printf ("Interface%s con IP%s\n",
                                        ifa_tmp->ifa_name,
                                        inet_ntoa( ((struct sockaddr_in*)
                                                    (ifa_tmp->ifa_addr))->sin_addr) );
                        }
                        ifa_tmp = ifa_tmp->ifa_next;
                } while (ifa_tmp);

                /* free memory */
                freeifaddrs (ifa);

        } else {
                perror ("getifaddrs");
                exit (-1);
        }

        return 0;
}
```

Bibliografía

- Free Software Foundation, Inc. «*The GNU C Library Reference Manual*».
- W. Richard Stevens, Bill Fenner, Andrew M. Rudolf. «*UNIX network programming. Vol. 1: The sockets networking API*». Editorial Addison-Wesley, 3ª edición, 2004.
  - Capítulo 3: Introducción a los *sockets*.
  - Capítulo 4: *Sockets* TCP.
  - Capítulo 8: *Sockets* UDP.