Universidad de Alcalá

Departamento de Automática

ARQUITECTURA DE REDES
Laboratorio

PRÁCTICA 2: **MANUAL DE SOCKETS EN C**

Grado en Ingeniería Informática
Curso 2017/18

ÍNDICE

1. INTRODUCCIÓN

1.1 La familia de protocolos TCP/IP.

La familia de protocolos TCP/IP fue creada a finales de los años 60 cuando aún no se había definido el modelo OSI. Consta de una serie de protocolos que cubren todos los niveles OSI uniendo varios de ellos.

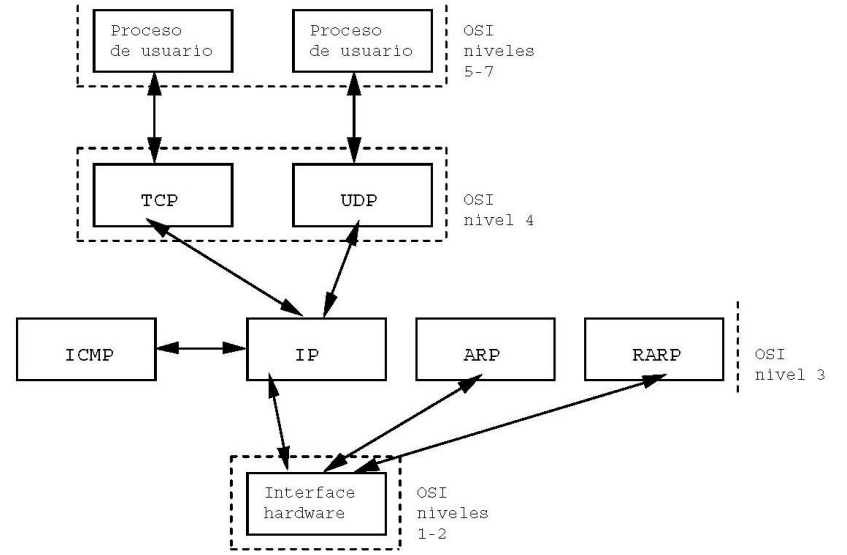


Figura 1-1: Familia TCP/IP y niveles OSI

- Los niveles 5-7 de OSI se incluyen dentro de los servicios al usuario ofrecidos por la familia TCP/IP:
- El nivel 4 se corresponde con *Transmission Control Protocol* (TCP) y *User Datagram Protocol* (UDP) que cubren el nivel de transporte dando servicios con caracteristicas distintas.
- *Internet Protocol* (IP): se corresponde con el nivel 3 de red de OSI.
- *Data Link Layer*, cubre los niveles 1 y 2 de OSI. Existen multitud de protocolos usados por TCP/IP ya que también existen muchos canales físicos de comunicación que permiten la conexión entre dos máquinas: líneas telefónicas, Ethernet, token ring, línea serie...

1.2 Nivel de red (IP).

El protocolo IP de nivel de red nos proporciona un servicio sin conexión y no seguro (*connectionless, datagram service*).

Para comprender qué es un servicio sin conexión lo compararemos con el servicio postal. Cada mensaje lleva consigo las direcciones completas de origen y destino, por lo que cada uno de ellos se puede encaminar de forma independiente a través del

sistema. Al igual que en correos, un mensaje puede sufrir ciertos retardos independientes del resto de mensajes por lo que dos mensajes con el mismo origen y destino pueden recibirse en distinto orden a como se enviaron.

No es un servicio seguro ya que no garantiza que todos los mensajes lleguen a su destino o que lleguen correctamente. Esta será tarea de los niveles superiores.

1.3 Nivel de transporte (TCP, UDP).

La familia TCP/IP nos ofrece dos protocolos de nivel de transporte TCP y UDP. La diferencia fundamental es que mientras TCP es un servicio orientado a la conexión (*connection-oriented, stream service*), y UDP no lo es. Por tanto en TCP, un proceso que desea comunicarse con otro debe establecer una conexión y una vez terminada la comunicación debe romper esa conexión.

TCP acepta mensajes de longitud arbitrariamente grandes, que deberá separar en pedazos (segmentación) que no excedan de los 64Kbytes (máximo aceptado por IP), enviándolos como mensajes separados. Como IP no garantiza el orden de los mensajes, es trabajo de TCP reensamblar de forma correcta los submensajes antes de entregar el mensaje completo y libre de errores al destinatario.

Además utiliza números de 16 bits para identificar varios destinos dentro de una misma máquina. Son los llamados puertos". Un proceso que quiera contactar con otro de una máquina remota debe conocer además de la dirección Internet de la otra máquina, el número de puerto en el que el otro proceso está escuchando.

Un servicio UDP por otro lado, ofrece en esencia lo mismo que el protocolo IP de nivel 3, con dos diferencias fundamentales:

1. Ofrece la posibilidad de detección de errores (pero se puede perder un mensaje entero y UDP no nos avisará).
2. Al igual que TCP, gestiona números de puerto permitiendo varias comunicaciones simultáneas en una misma máquina.

2. SOCKETS EN C

2.1. Conceptos básicos.

Los sockets son una de las herramientas que ofrecen los Sistemas Operativos para la comunicación entre diferentes procesos. La particularidad que tienen frente a otros mecanismos de comunicación entre procesos (IPC – Inter-Process Communication) es que posibilitan la comunicación aún cuando ambos procesos estén corriendo en distintos sistemas unidos mediante una red. De hecho, el API de sockets es la base de cualquier aplicación que funcione en red puesto que ofrece una librería de funciones básicas que el programador puede usar para desarrollar aplicaciones en red.

Este API permite la comunicación sobre una gran variedad de redes, pero en este manual nos concentraremos en su uso sobre redes TCP/IP.

Los sockets para TCP/IP permiten la comunicación de dos procesos que estén conectados a través de una red TCP/IP. En una red de este tipo, cada máquina está identificada por medio de su dirección IP que debe ser única. Sin embargo, en cada máquina pueden estar ejecutándose múltiples procesos simultáneamente. Cada uno de estos procesos se asocia con un número de puerto, para poder así diferenciar los distintos paquetes que reciba la máquina (proceso de multiplexación).

Un socket se identifica unívocamente por la dupla **dirección IP + número de puerto**. Una comunicación entre dos procesos se identifica mediante la asociación de los sockets que estos emplean para enviar y recibir información hacia y desde la red: **identificador de socket origen + identificador de socket destino**.

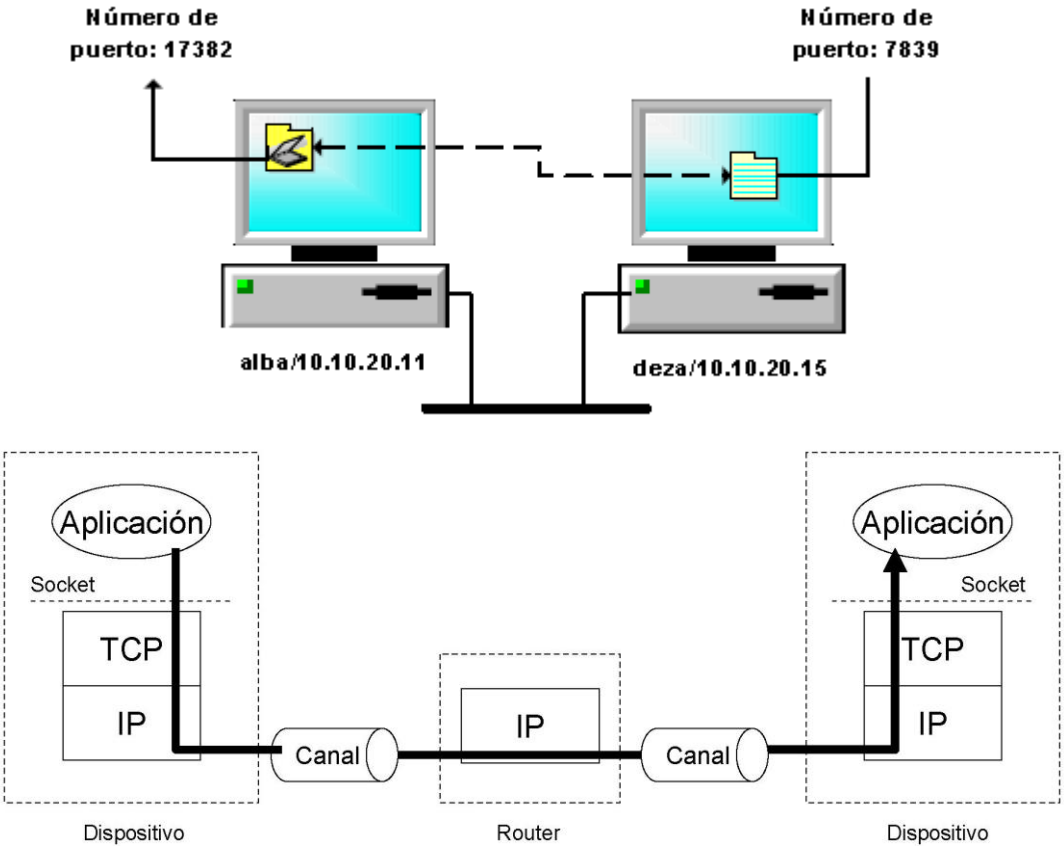


Figura 2-1: Comunicación entre aplicaciones en una red TCP/IP

Un socket es una abstracción a través de la cual una aplicación puede enviar y recibir información de manera muy similar a como se escribe y lee de un fichero. La información que una aplicación envía por su socket origen puede ser recibida por otra aplicación en el socket destino y viceversa.

Existen diferentes tipos de sockets dependiendo de la pila de protocolos sobre la que se cree dicho socket. Nos centraremos en la pila de protocolos TCP/IP.

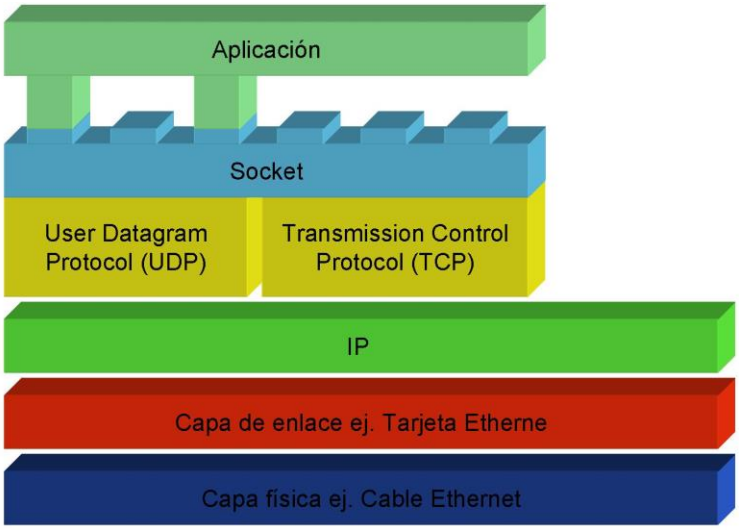


Figura 2-2: Interfaz de sockets

En este manual se verá como se le pueden asociar estos parámetros a los sockets creados.

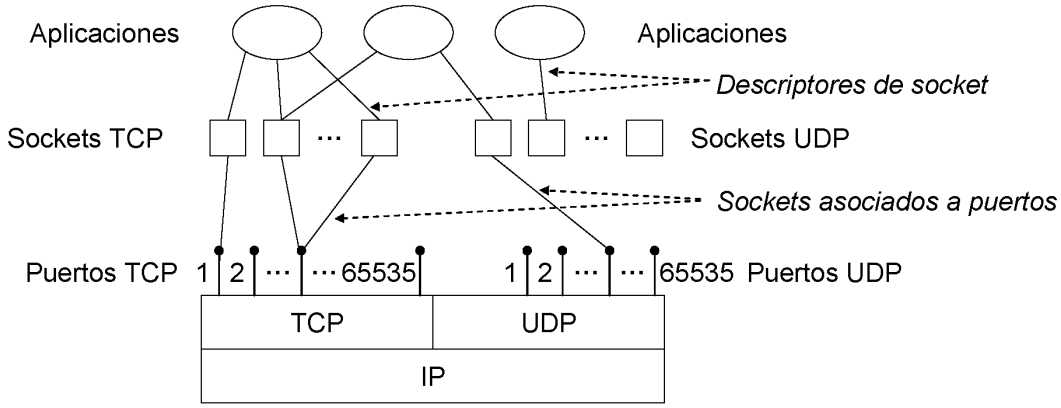


Figura 2-3: Sockets, protocolos y puertos

La Figura 2-3 muestra la relación lógica entre aplicaciones, sockets, protocolos y puertos en un dispositivo. Hay varios aspectos a destacar en esta relación. Primero, un programa puede usar más de un socket al mismo tiempo. Segundo, diferentes programas pueden usar el mismo socket al mismo tiempo aunque esto es menos común. Como se muestra en la Figura 2-3, cada socket tiene asociado un puerto TCP o UDP, según sea el caso. Cuando se recibe un paquete dirigido a dicho puerto, este paquete se pasa a la aplicación correspondiente.

2.2 Dominios de comunicación.

Los sockets se crean dentro de lo que se denomina un dominio de comunicación, al igual que un archivo se crea dentro de un sistema de ficheros concreto. El dominio de comunicación permite definir el lugar donde se encuentran los procesos que se van comunicar.

Los dominios que se definen en el lenguaje C son los siguientes:

- AF_UNIX: representa el dominio característico de los procesos que se comunican dentro en un mismo sistema UNIX.
- AF_INET: es el dominio que utilizan los procesos que se comunican a través de cualquier red TCP/IP.
- Alguno más que no vamos a estudiar.

Destacar que en esta introducción únicamente se hará referencia a sockets creados bajo el dominio AF_INET.

2.3 Tipos de sockets.

En el dominio AF_INET se definen los siguientes tipos de sockets:

- Sockets Stream
- Sockets Datagram
- Sockets Raw

El tipo de *sockets Stream* hace uso del protocolo TCP (protocolo de la capa de transporte) que provee un flujo de datos bidireccional, orientado a conexión, secuenciado, sin duplicación de paquetes y libre de errores.

El tipo de *sockets Datagram* hacen uso del protocolo UDP (protocolo de la capa de transporte), el cual provee un flujo de datos bidireccional, no orientado a conexión, en el cual los paquetes pueden llegar fuera de secuencia, puede haber pérdidas de paquetes o pueden llegar con errores.

El tipo de *sockets Raw* permiten un acceso a más bajo nivel, pudiendo acceder directamente al protocolo IP del nivel de Red. Su uso está mucho más limitado ya que está pensado principalmente para desarrollar nuevos protocolos de comunicación, o para obviar los protocolos del nivel de transporte.

2.4 Ordenación de los bytes.

Los microprocesadores que incorporan los ordenadores poseen dos formas diferentes de representar los números, primero el byte más significativo y luego el menos significativo (*Big-Endian*) o al revés (*Little-Endian*).

Esta diferencia, que dentro de un microprocesador no deja de ser una anécdota, adquiere su importancia en el mundo de las redes, pues si cada ordenador manda los datos como los representa internamente (*Big-Endian o Little-Endian*), el ordenador que recibe dichos datos puede interpretarlos correctamente o incorrectamente, dependiendo de si su formato de representación coincide con el del emisor o no.

Por ejemplo, si un ordenador indica que desea establecer una conexión con el puerto 80 (servidor Web) de otro ordenador, si lo envía en formato *Big-Endian* los bytes que enviará serán: 0x00 0x50, mientras que en formato *Little-Endian* enviará los bytes 0x50 0x00. Si el otro ordenador representa los datos en formato diferente, al recibir, por ejemplo, 0x00 0x50 lo traducirá como el puerto 0x50 0x00, esto es 20480, que obviamente no corresponde al servidor Web.

Para solventar este problema, se definió un formato conocido como Orden de los Bytes en la Red (*Network Byte Order*), que establece un formato común para los datos que se envían por la red, como son el número de puerto, etc., de forma que todos los datos, antes de ser enviados a las funciones que manejan las conexiones en la red, deben ser convertidos a este formato de red. A continuación, podemos ver las cuatro funciones que permiten manejar estos datos.

htonl.

```
#include <netinet/in.h>
unsigned long int htonl(unsigned long int hostlong);
```

La función *htonl* convierte el entero de 32 bits dado por *hostlong* desde el orden de bytes del hosts al orden de bytes de la red.

Ejemplo:

```
unsigned long int red, host;
...
red=htonl(host);
```

htons.

```
#include <netinet/in.h>
unsigned short int htons(unsigned short int hostshort);
```

La función *htons* convierte el entero de 16 bits dado por *hostshort* desde el orden de bytes del hosts al orden de bytes de la red.

Ejemplo:

```
unsigned short int red, host;
...
red=htons(host);
```

ntohl.

```
#include <netinet/in.h>
unsigned long int ntohl(unsigned long int netlong);
```

La función *ntohl* convierte el entero de 32 bits dado por *netlong* desde el orden de bytes de la red al orden de bytes del hosts.

Ejemplo:

```
unsigned long int host, red;
...
host=ntohl(red);
```

ntohs.

```
#include <netinet/in.h>
unsigned short int ntohs(unsigned short int netshort);
```

La función *ntohs* convierte el entero de 16 bits dado por *netshort* desde el orden de bytes de la red al orden de bytes del hosts.

Ejemplo:

```
unsigned short int host, red;
...
host=ntohs(red);
```

2.5 Servicio orientado a conexión (TCP)

2.5.1 El modelo utilizado

El modelo más utilizado para el desarrollo de aplicaciones en red es el de *cliente-servidor.* Un servidor es un proceso que espera contactar con algún proceso cliente para darle algún tipo de servicio. Un proceso de comunicación típico siguiendo este modelo es el siguiente:

1. El proceso servidor se pone en ejecución en algún computador. Inicializa sus estructuras de datos y se queda esperando a que algún cliente requiera sus servicios.
2. Un proceso es puesto en ejecución en el mismo computador o en otro de la red. En algún momento este proceso necesita acceder al servidor, se convierte en este momento en proceso cliente, enviando una petición de servicio a través de la red y se queda esperando respuesta.
3. El servidor despierta, atiende la petición, responde a su cliente actual y se queda de nuevo esperando otro cliente.

A continuación se describen los pasos a realizar para la creación de un servicio orientado a conexión (TCP), tanto en la parte cliente como en la parte del servidor. En la siguiente figura se muestran los pasos a realizar en ambos casos, invocando diversas funciones, cuyo funcionamiento será detallado a continuación.

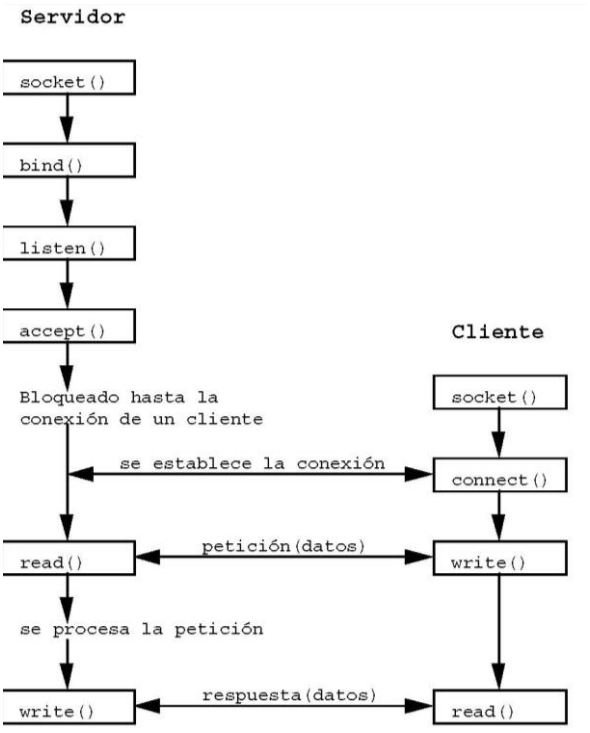


Figura 2-4 Llamadas al sistema para sockets en un protocolo orientado a la conexión

El servidor creará inicialmente un extremo de la conexión pidiendo un socket (`socket()`) y dándole una dirección local (`bind()`). El servidor puede en algún momento recibir varias peticiones de conexión simultáneas por lo que se ofrece una llamada que permite especificar el número máximo de conexiones pendientes (`listen()`). Después queda esperando la petición de conexión por parte de algún cliente (`accept()`).

Por otro lado el cliente creará un socket, le dará una dirección local y lanzará una petición de conexión al servidor (`connect()`). Si el servidor está disponible (ha ejecutado `accept` y no hay peticiones anteriores encoladas) inmediatamente se desbloquean tanto cliente como servidor. En la parte del servidor se habrá creado un nuevo socket (cuyo identificador devuelve `accept()`) que es el que realmente está conectado con el cliente, de forma que el original sigue sirviendo al servidor para atender nuevas peticiones de conexión.

Cliente y servidor se intercambiarán datos mediante `read()` y `write()`(también se utilizan las funciones `send()` y `recv()` para realizar la transferencia de datos), pudiendo finalmente cerrar la conexión mediante la llamada `close()`.

2.5.2 Resumen de llamadas al sistema.

a) Creación de socket.

Para crear un socket se usa la llamada al sistema `socket()` especificando la familia de protocolos que usará y el tipo de conexión. La función devuelve un entero que llamaremos descriptor de socket y servirá para referenciar al socket en el resto de las llamadas (al igual que *el file descriptor* en el acceso a ficheros).

b) Asignación de dirección local

La llamada `socket()` no asigna una dirección al socket creado. Automáticamente las llamadas `send()` y `connect()` asignan a nuestro socket la dirección IP de nuestra máquina y un número de puerto libre. Sin embargo hay casos en que necesitaremos dar una dirección determinada a un socket.
Todos los clientes deben conocer la dirección de su servidor para poder pedirle servicios, para ello el servidor creará un socket con una dirección IP y un puerto que hará públicos. Para asignar una dirección determinada a un socket se usa la llamada `bind()`.

c) La llamada al sistema listen()

Una vez realizada la llamada `listen()`, todas las peticiones de conexión que lleguen al socket serán guardadas en su cola de peticiones. A continuación el proceso propietario del socket podrá atender la primera de ellas mediante la llamada `accept()`.

d) La llamada al sistema accept()

Si al realizar esta llamada la cola de peticiones estaba vacía el proceso quedará bloqueado hasta la llegada de una nueva petición. Una vez ha llegado una petición el sistema crea un nuevo socket cuyo descriptor devuelve la llamada `accept()`, y que será el utilizado en la comunicación.

e) Estableciendo conexión.

La llamada `connect()` sirve al cliente para realizar una petición de conexión al servidor. Esta petición será inmediatamente atendida si el servidor estaba bloqueado en `accept()`, o será encolada si la cola no estaba vacía o el servidor no acepta conexión en ese momento.

f) Transferencia de datos.

Cuando dos procesos ya han quedado "enganchados" por una conexión, pueden enviar-recibir datos, para ello se usan las llamadas `send()` y `recv()`. Estas llamadas sólo trabajan con sockets "conectados" ya que no permiten especificar dirección destino.

g) Cerrar la conexión

Cuando hagamos `close()`, el sistema nos asegurará que antes de destruir el socket todos los datos que han sido enviados serán recibidos en el otro extremo.

2.6. Creación de un socket

Los sockets se crean llamando a la función `socket()`, que devuelve el identificador de socket, de tipo entero (es equivalente al concepto de identificador de fichero).

En caso de que se haya producido algún error durante la creación del socket, la función devuelve -1 y la variable global `errno` se establece con un valor que indica el error que se ha producido. La función `perror("...")` muestra por pantalla un mensaje explicativo sobre el error que ha ocurrido.

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

int socket(int domain, int type, int protocol);
      http://linux.die.net/man/2/socket
```

El prototipo de la función `socket()` es el siguiente:

```
sockfd = socket ( int dominio, int tipo, int protocolo );
```

- `sockfd`: Identificador de socket. Se utilizará para conectarse, recibir conexiones, enviar y recibir datos, etc.
- `dominio`: Dominio donde se realiza la conexión. En este caso, el dominio será siempre `AF_INET`.
- `tipo`: Se corresponde con el tipo de socket que se va a crear, y puede tomar los siguientes valores (definidos como constantes en las librerías): `SOCK_STREAM`, `SOCK_DGRAM` o `SOCK_RAW`.
- `protocolo`: Indica el protocolo que se va a utilizar. El valor 0 indica que seleccione el protocolo más apropiado (TCP para `SOCK_STREAM`, UDP para `SOCK_DGRAM`).

A continuación se muestra un ejemplo de utilización:

```
#include <sys/types.h>
#include <sys/socket.h>
...
int sockfd;
sockfd = socket ( AF_INET, SOCK_STREAM, 0 );
...
```

2.6.1 Estructuras de datos

La función `socket()` únicamente crea un socket, pero no le asigna ningún valor, esto es, no lo asocia a ninguna dirección IP ni a ningún puerto.

Para poder realizar esta asociación, lo primero es conocer una serie de estructuras que son necesarias para llevarla a cabo.

```
struct sockaddr
{
  unsigned short            sa_family;    // AF_*
  char                      sa_data[14];  // Dirección de protocolo.
};

struct sockaddr_in
{
  short int          sin_family;   // AF_INET
  unsigned short     sin_port;     // Numero de puerto.
  struct in_addr     sin_addr;     // Dirección IP.
  unsigned char      sin_zero[8];  // Relleno.
};

struct in_addr
{
unsigned long    s_addr;   // 4 bytes.
};
```

La estructura `sockaddr` es la estructura genérica que se usa en las diferentes funciones definidas en el API de Sockets. Como puede comprobarse, sólo define el dominio mientras que los datos (los parámetros del socket) quedan sin especificar (simplemente se reserva espacio en memoria, pero sin ninguna estructura específica). Esto permite que esta estructura pueda emplearse en cualquiera que sea el dominio con el que se haya definido el socket.

Para poder definir los parámetros que se quieren asociar al socket, se usan estructuras específicas para cada dominio. La estructura TCP/IP es la `sockaddr_in`, equivalente a la estructura `sockaddr`, pero que permite referenciar a sus elementos de forma más sencilla.

Los campos de la estructura `sockaddr_in` son los siguientes:

- **`sin_family`**: Tomará siempre el valor `AF_INET`.
- **`sin_port`**: Representa el número de puerto, y debe estar en la ordenación de bytes de la red.
- **`sin_zero`**: Se utiliza simplemente de relleno para completar la longitud de sockaddr.
- **`sin_addr`**: Este campo representa la dirección IP y se almacena en un formato específico, que se detalla a continuación.

Para convertir una dirección IP en formato texto (por ejemplo, “193.144.57.67”) a un `unsigned long` con la ordenación de bytes adecuada, se utiliza la función `inet_addr()`. Esta función convierte únicamente direcciones IP a formato numérico, **NO** convierte nombres de máquinas. En caso de error devuelve –1 y activa la variable global `errno`.

2.6.2 Asociación de un socket a unos parámetros determinados

La función `bind()` se utiliza para asociar el socket a estos parámetros, mediante una dirección IP y número de puerto de la máquina local, a través de los que se enviarán y recibirán datos.

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

      int bind(int sockfd, struct sockaddr *my_addr, int addrlen);

      http://linux.die.net/man/2/bind
```

El prototipo de la función es el siguiente:

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

- `sockfd`: Identificador de socket devuelto por la función `socket()`.
- `my_addr`: Es un puntero a una estructura `sockaddr` que contiene la IP de la máquina local y el número de puerto que se va a asignar al socket (esta estructura se detalla en la siguiente sección).
- `addrlen`: debe estar establecido al tamaño de la estructura anterior, utilizando para ello la función sizeof().

Ejemplo:

```
bind ( sockfd, (struct sockaddr *) &sin, sizeof (sin) );
```

Previamente a la llamada a la función `bind()` es necesario asignar valores a una variable de tipo `sockaddr_in` que serán los parámetros que se asociarán al socket.

Ejemplo:

```
...
struct sockaddr_in sin;
...
sin.sin_family = AF_INET;
sin.sin port = htons ( 1234 );
            // Número de puerto donde  recibirá paquetes el programa
sin.sin_addr.s_addr = inet_addr ("132.241.5.10");
            // IP por la que recibirá paquetes el programa
 ...
```

- Existen algunos casos especiales a la hora de asignar valores a ciertos campos:
- En caso de asignar el valor cero a `sin_port`, el sistema fijará el número de puerto al primero que encuentre disponible.
- Para realizar la asignación de la dirección IP de forma automática (sin tener por que conocer previamente la dirección IP donde se va a ejecutar el programa) se puede utilizar la constante: `INADDR_ANY`. Esto le indica al sistema que el programa recibirá mensajes por cualquier IP válida de la máquina, en caso de disponer de varias.

Ejemplo:

```
...
sin.sin_port = 0;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
...
```

2.6.3 Habilitar un socket para recibir peticiones de conexión

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

int listen(int sockfd, int backlog);

http://linux.die.net/man/2/listen
```

El primer paso para la comunicación usando un servicio orientado a la conexión (protocolo de transporte TCP) es el establecimiento de una conexión TCP. En este sentido, aún cuando se haya creado un socket de tipo `SOCK_STREAM`, el socket no está preparado para establecer dicha conexión.

Para habilitar un socket para recibir peticiones de conexión y proceder al establecimiento de dicha conexión, es necesario utilizar la función `listen()`.

La función `listen()` se invoca únicamente desde el servidor, y habilita al socket para poder recibir conexiones. Únicamente se aplica a sockets de tipo `SOCK_STREAM`.

El prototipo de la función es el siguiente:

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

- `sockfd`: Identificador de socket obtenido en la función `socket()`, que será utilizado para recibir conexiones.
- `backlog`: Número máximo de conexiones en la cola de entrada de conexiones. Las conexiones entrantes quedan en estado de espera en esta cola hasta que sean aceptadas mediante la función `accept()`.

2.6.4 Aceptar peticiones de conexión

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

int accept(int sockfd, struct sockaddr *addr, socklen_t
*addrlen);

http://linux.die.net/man/2/accept
```

Si bien la función `listen`() prepara el socket y lo habilita para recibir peticiones de establecimiento de conexión, es la función `accept`() la que realmente queda a la espera de estas peticiones. Cuando una petición realizada desde un proceso remoto (cliente) es recibida, la conexión se completa en el servidor siempre y cuando éste esté esperando en la función `accept`().

La función `accept`() es utilizada en el servidor una vez que se ha invocado a la función `listen`(). Esta función espera hasta que algún cliente establezca una conexión con el servidor. Es una llamada bloqueante, esto es, la función no finalizará hasta que se haya producido una conexión o sea interrumpida por una señal.

Es conveniente destacar que una vez que se ha producido la conexión, la función `accept`() devuelve un nuevo identificador de socket que será utilizado para la comunicación con el cliente que se ha conectado.

El prototipo de la función es el siguiente:

```
int accept (int sockfd, struct sockaddr *addr, socklen_t
*addrlen);
```

- `sockfd`: Identificador de socket habilitado para recibir conexiones.
- `addr`: Puntero a una estructura `sockaddr` (en nuestro caso su equivalente `sockadd_in`), donde se almacenará la información (dirección IP y número de puerto) del proceso que ha realizado la conexión.
- `addrlen`: Debe contener un puntero a un valor entero que represente el tamaño la estructura `addr`. Debe ser establecido al tamaño de la estructura `sockaddr`, mediante la llamada `sizeof(struct sockaddr)`. Si la función escribe un número de bytes menor, el valor de `addrlen` es modificado a la cantidad de bytes escritos.

Ejemplo:

```
...
int sockfd, new_sockfd;
struct sockaddr_in server_addr;
struct sockaddr_in remote_addr;
int addrlen;
// Creación del socket.
sockfd = socket (PF_INET, SOCK_STREAM, 0 );
// Definir valores en la estructura server_addr.
sin.sin_family = AF_INET;
sin.sin port = htons ( 1234 ); // Número de puerto donde
                          // recibirá paquetes el programa
sin.sin_addr.s_addr = htonl(INADDR_ANY);
// Asociar valores definidos al socket
bind(sockfd, (struct sockaddr *)&server_addr, sizeof(struct
sockaddr));
// Se habilita el socket para poder recibir conexiones.
```

```
listen ( sockfd, 5);
addrlen = sizeof (struct sockaddr );
// Se llama a accept() y el servidor queda en espera de
conexiones.
new_sockfd = accept (sockfd, &remote_addr, &addrlen);
...
```

2.6.5. Lanzar peticiones de conexión

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

int connect(int sockfd, const struct sockaddr *serv_addr,
socklen_t addrlen);

http://linux.die.net/man/2/connect
```

Esta función es invocada desde el cliente para solicitar el establecimiento de una conexión TCP.

La función `connect`() inicia la conexión con el servidor remoto, por parte del cliente. El prototipo de la función es el siguiente:

```
int connect (int sockfd, struct sockaddr *serv_addr, int
addrlen);
```

- `sockfd`: Identificador de socket devuelto por la función `socket()`.
- `serv_addr`: Estructura `sockaddr` que contiene la dirección IP y número de puerto del destino.
- `addrlen`: Debe ser inicializado al tamaño de la estructura `serv_addr`, pasada como parámetro.

Ejemplo:

```
...
int sockfd;
struct sockaddr_in remote_addr;
int addrlen;
// Creación del socket.
sockfd = socket (AF_INET, SOCK_STREAM, 0 );
// Definir valores en la estructura server_addr.
sin.sin_family = AF_INET;
sin.sin port = htons ( 1234 ); // Número de puerto donde
// está esperando el servidor
sin.sin_addr.s_addr = inet_addr("1.2.3.4"); // Dirección IP
// del servidor
addrlen = sizeof (struct sockaddr );
```

```
// Se llama a connect () y se hace la petición de conexión
al servidor
connect (sockfd, &remote_addr, &addrlen);
...
```

2.6.6 Conexión entre los sockets del cliente y servidor

Una vez que el cliente ha hecho la llamada a la función `connect`() y esta ha concluido con éxito (lo que significa que el servidor estaba esperando en la función `accept`() y al salir ha creado el nuevo socket), la situación es la que se muestra en la siguiente figura.

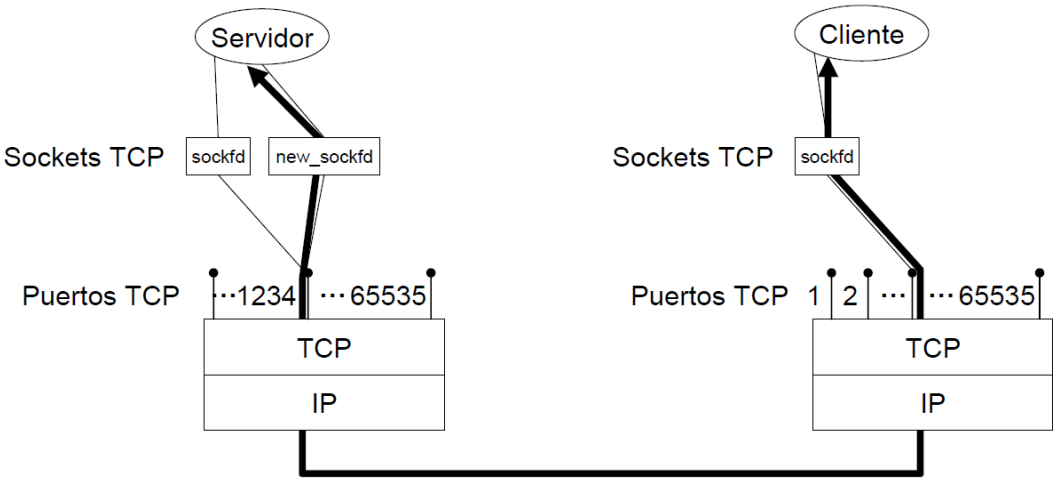


Figura 2-5 Sockets, protocolos y puertos

Como se puede apreciar en la figura, se crea una conexión entre el socket usado por el cliente en la función `connect`() (`sockfd` en nuestro ejemplo) y el socket que devuelve la función `accept`() en el servidor (`new_sockfd` en nuestro ejemplo).

Esta conexión permitirá la comunicación entre ambos procesos usando los sockets que están conectados

2.6.7 Enviar y recibir datos a través de un socket TCP

```
#include <sys/types.h>
#include <sys/socket.h>
     ssize_t send(int s, const void *buf, size_t len, int
     flags);
     http://linux.die.net/man/2/send
                ..........................................................

#include <sys/types.h>
#include <sys/socket.h>
     ssize_t recv(int s, void *buf, size_t len, int flags);
     http://linux.die.net/man/2/recv
```

Una vez que la conexión ha sido establecida, se inicia el intercambio de datos, utilizando para ello las funciones `send`() y `recv`().

El prototipo de la función `send`() es el siguiente:

```
ssize_t send (int sockfd, const void *buf, size_t len, int
flags );
```

- `sockfd`: Identificador de socket para enviar datos.
- `buf`: Puntero a los datos que serán enviados.
- `len`: Número de bytes a enviar.
- `flags`: Por defecto, 0.

La función `send()` devuelve el número de bytes enviados, que puede ser menor que la cantidad indicada en el parámetro `len`.

La función `recv()` se utiliza para recibir datos, y posee un prototipo similar a la anterior:

```
ssize_t recv (int sockfd, void *buf, size_t len, int
flags);
```

- `sockfd`: Identificador de socket para la recepción de los datos.
- `buf`: Puntero a un buffer donde se almacenarán los datos recibidos.
- `len`: Número máximo de bytes a recibir
- `flags`: Por defecto, 0. Para más información, consultar la ayuda en línea.

La función `recv`() es bloqueante, no finalizando hasta que se ha recibido algún tipo de información. Resaltar que el campo `len` indica el número máximo de bytes a recibir, pero no necesariamente se han de recibir exactamente ese número de bytes. La función `recv`() devuelve el número de bytes que se han recibido.

Ejemplo:

```
char mens_serv[100];
...
mens_clien = "Ejemplo";
send(sid, mens_clien, strlen(mens_clien)+1, 0);
...
recv(sid, mens_serv, 100, 0);
...
```

2.6.8 Cierre de un socket

```
#include <unistd.h>
int close(int fd);

http://linux.die.net/man/2/close
```

Simplemente hay que hacer la llamada a la función pasándole como argumento el descriptor del socket que se quiere cerrar.

Es importante que cuando un socket no vaya a usarse más, se haga la llamada a `close`() para indicárselo al Sistema Operativo y que éste lo libere.

Si se desea un poco más de control sobre cómo se cierra el socket se puede usar la función `shutdown()` que permite cortar la comunicación en un sentido, o en los dos (como lo hace `close()`).

```
#include <sys/socket.h>
      int shutdown(int sockfd, int how)

      http://linux.die.net/man/2/shutdown
```

`sockfd` es el descriptor de socket que se desea desconectar, y `how` puede tener uno de los siguientes valores:

0 -- No se permite recibir más datos

1 -- No se permite enviar más datos

2 -- No se permite enviar ni recibir más datos (similar a `close`())

`shutdown`() devuelve 0 si tiene éxito, y -1 en caso de error.

Si se usa `shutdown`() en un *datagram socket*, simplemente inhabilitará el socket para posteriores llamadas a `send`() y `recv`()

`shutdown`() no cierra realmente el descriptor de fichero, sólo cambia sus condiciones de uso. Para liberar un descriptor de socket es necesario usar `close`().

2.7 Diagrama de estados de una conexión TCP

En la siguiente figura se muestran los estados por los que puede pasar una conexión TCP. Asimismo, se detallan los eventos y los segmentos que se intercambian en cada transición de un estado a otro.

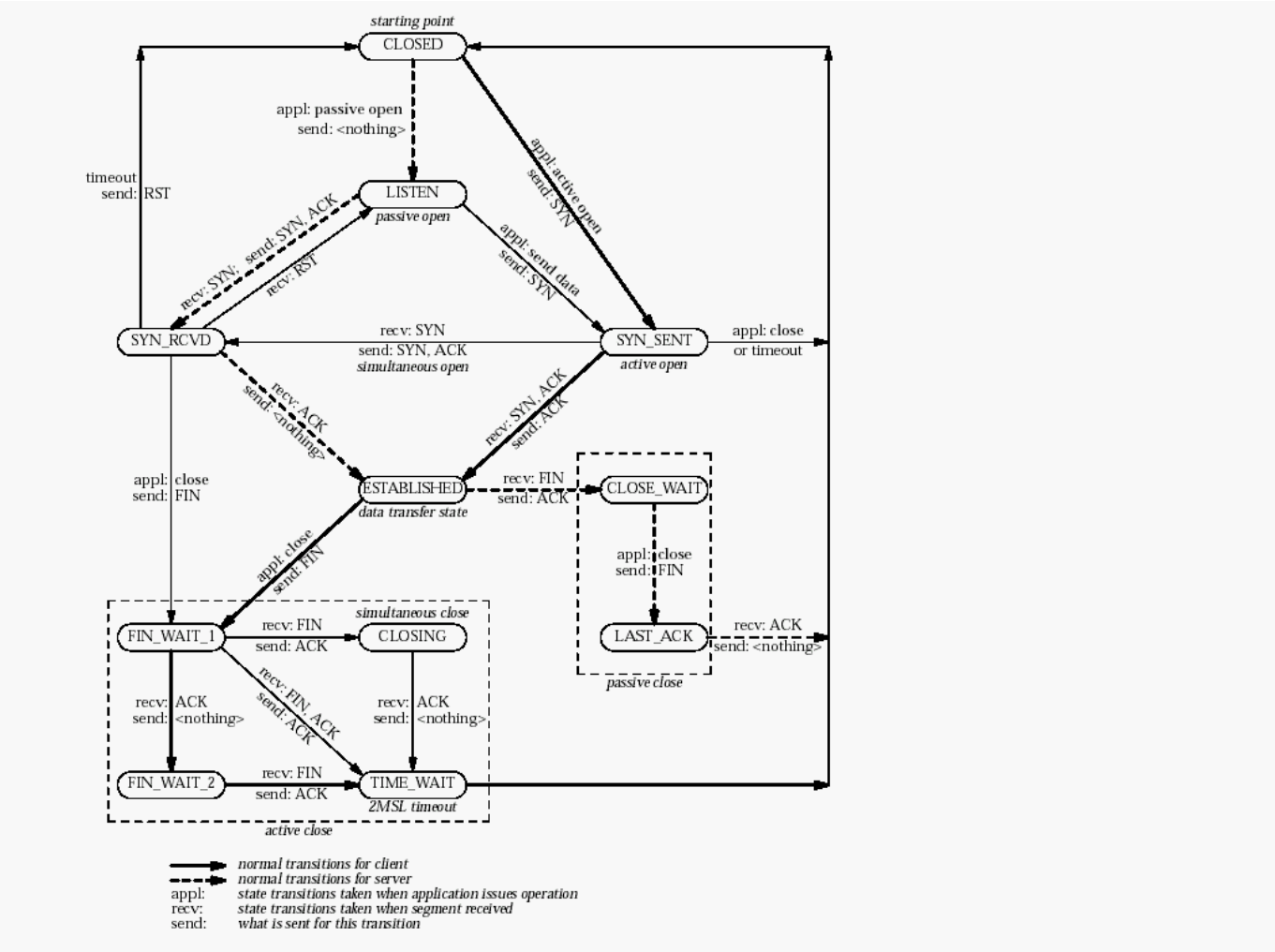


Figura 2-6: Diagrama de estados de una conexión TCP

Tal y como se puede comprobar, un socket al crearse está en el estado `CLOSED`. Existen dos posibilidades para salir de este estado, una apertura activa o una apertura pasiva. Mientras que la primera se da en el cliente cuando este realiza la llamada a la función `connect`() y con ella inicia el proceso de establecimiento de la conexión, la segunda ocurre en el servidor una vez hechas las llamadas a las funciones `listen()` y `accept`(). De esta forma, en el servidor el socket utilizado para recibir peticiones de establecimiento de conexión pasa al estado `LISTEN`, y espera a que desde el cliente se inicie el proceso de establecimiento de la conexión.

Tras el establecimiento de la conexión, tanto el socket creado en el cliente como el socket que devuelve la función `accept`() pasan al estado `ESTABLISHED`. El socket utilizado por el servidor para recibir peticiones de establecimiento de conexión continúa en el estado `LISTEN`.

El intercambio de datos tiene lugar entre los sockets que están en el estado `ESTABLISHED`. Una vez finalizado éste, ambos sockets pasan al estado `CLOSED` (tras

atravesar una serie de estados intermedios). El evento que provoca este cierre es la llamada en ambos procesos (cliente y servidor) a la función `close`().

2.8 Servicio no orientado a conexión (UDP)

2.8.1 El modelo utilizado

Las llamadas ofrecidas para la comunicación sin conexión son más sencillas que las vistas anteriormente. Su esquema de funcionamiento se detalla a continuación:

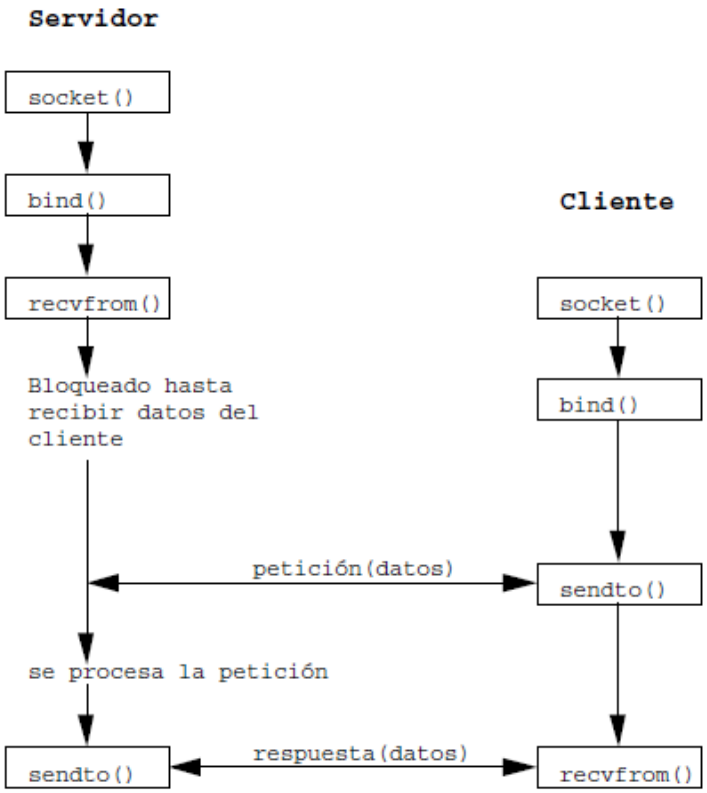


Figura 2-7 Llamadas al sistema para sockets en un protocolo no orientado a la conexión

En primer lugar el servidor crea un socket (llamada `socket`()). En la creación, al socket no se le asigna nombre (dirección local) por lo que el resto de procesos no pueden referenciarlo, y por lo tanto no se pueden recibir mensajes en él.

Mediante la llamada `bind`() se asigna una dirección local al socket (para TCP/IP una dirección local es la dirección IP de la máquina más un número de *port*). Esta dirección deberá ser conocida por los clientes. Una vez configurado el socket, el servidor puede quedar esperando la recepción de un mensaje utilizando la llamada `recvfrom`().

Por otro lado el cliente deberá igualmente crear un socket y asignarle una dirección de su máquina. Una vez hecho esto podrá enviar mensajes (peticiones de servicio) al servidor usando la llamada `sendto`() en la que especificará como dirección destino la del socket creado por el servidor.

La llamada `recvfrom`() proporcionará al servidor la dirección del cliente, que podrá ser usada por el primero para enviar la respuesta. Pudiendo finalmente cerrar la conexión mediante la llamada `close`().

2.8.2 Enviar y recibir datos a través de un socket UDP

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

ssize_t sendto(int s, const void *buf, size_t len, int flags,
const struct sockaddr *to, socklen_t tolen);
```

http://linux.die.net/man/2/send

.....

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

ssize_t recvfrom(int s, void *buf, size_t len, int flags,
struct sockaddr *from, socklen_t *fromlen);
```

http://linux.die.net/man/2/recv

En el caso de envío y recepción de datos a través de sockets no orientados a la conexión, se utilizan estas funciones pues es necesario indicar quien es el destinatario u origen de los datos enviados o recibidos.

El prototipo de la función `sendto()` es el siguiente:

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int
flags, const struct sockaddr *to, socklen_t tolen);
```

- `sockfd`: Identificador de socket para el envío de datos.
- `buf`: Puntero a los datos a ser enviados.
- `len`: Longitud de los datos en bytes.
- `flags`: Por defecto, 0.
- `to`: Puntero a una estructura `sockaddr` que contiene la dirección IP y número de puerto destino.
- tolen: Debe ser inicializado al tamaño de `struct sockaddr`, mediante la función `sizeof()`.

La función `sendto()` devuelve el número de bytes enviados, que puede ser menor que el valor indicado en `len`.

Es importante destacar que al no existir ninguna conexión, cuando se envían datos a través de un socket UDP, es necesario indicar cual debe ser el socket remoto. En la estructura `to`, que en nuestro caso será una estructura `sockaddr_in`, deberá estar contenida esta información.

El prototipo de la función `recvfrom()` es el siguiente:

```
ssize_t recvfrom (int sockfd, void *buf, size_t len, int
flags, struct sockaddr *from, socklen_t *fromlen);
```

- `sockfd`: Identificador de socket para la recepción de datos.
- `buf`: Puntero al buffer donde se almacenarán los datos recibidos.
- `len`: Número máximo de bytes a recibir.
- `flags`: Por defecto, 0.
- `from`: Puntero a una estructura `sockaddr` (vacía) donde se rellenarán la dirección IP y número de puerto del proceso que envía los datos.
- `fromlen`: Debe ser inicializado al tamaño de la estructura from, utilizando la función sizeof().

El campo `fromlen` debe contener un puntero a un valor entero que represente el tamaño la estructura `from`.

La llamada a la función `recvfrom`() es bloqueante, no devolviendo el control hasta que se han recibido datos. Al finalizar devuelve el número de bytes recibidos.

Es importante destacar, que además de recibir los datos y almacenarlos a partir de la posición de memoria a la que apunta `buf`, la llamada a `recvfrom`() permite conocer los parámetros que identifican al socket que envío estos datos. Estos parámetros se almacenan en una estructura `sockaddr` (en nuestro caso `sockaddr_in`) para lo cual es necesario que en la llamada le pasemos como parámetro la estructura, vacía, donde queremos que se almacenen.

Esta es la razón por la que antes de realizar una llamada a `sendto`() en el servidor se haga la llamada a `recvfrom`(). Mientras que el cliente conoce a priori los detalles del socket del servidor, y por tanto podrá rellenar los campos de la estructura `to` antes de la llamada a la función `sendto`(), el servidor no conoce estos datos del cliente (en tanto en cuanto, este servidor puede servir a cualquier cliente en cualquier parte del mundo). Por ello, debe esperar a recibir algo de forma que una vez se ejecute la función `recvfrom`(), los detalles del socket utilizado por el cliente estén almacenados en la estructura `from` y se puedan emplear para enviar la respuesta al cliente.

Ejemplo cliente:

```
...
struct sockaddr_in sserv;
char mens_serv[100];
mens_clien = ”Ejemplo”;
...
// Previamente se ha rellenado la estructura sserv
// con la dirección IP y número de puerto del servidor
sendto(sid,mens_clien, strlen(mens_clien)+1,0, (struct
sockaddr *) &sserv,sizeof(sserv));
...
long = sizeof(sserv);
recvfrom(sid,mens_serv,100,0,(struct sockaddr *)&
sserv,&long);
...
```

Ejemplo servidor:

```
...
struct sockaddr_in sserv;
char mens_cli[100];
...
long = sizeof(sserv);
recvfrom(sid,mens_cli,100,0,(struct sockaddr *)&sserv,
&long);
// La estructura sserv ha sido rellenada con la dirección
// IP y número de puerto del cliente
...
sendto(sid,mens_clien, strlen(mens_clien)+1,0, (struct
sockaddr *) &sserv,sizeof(sserv));
...
```

3. OTRAS FUNCIONALIDADES.

3.1 Función gethostname().

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

      int gethostname(char *hostname, size_t size)
      http://linux.die.net/man/2/gethostname
```

La función `gethostname()` devuelve el nombre del sistema donde se está ejecutando el programa. devuelve 0 cuando se ha ejecutado con éxito. El prototipo es el siguiente:

```
int gethostname ( char *hostname, size_t size )
```

- `hostname`: Puntero a un array de caracteres donde se almacenará el nombre de la máquina.
- `size`: Longitud en bytes del array `hostname`.

La función `gethostname()` también se utiliza para convertir un nombre de máquina en una dirección IP, a través de una estructura de nombre `hostent`. El prototipo es el siguiente:

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

- `name`: Nombre de la máquina que se quiere convertir a dirección IP.

La función devuelve un puntero a una estructura `hostent`, que se define como sigue:

```
struct hostent
{
char *h_name;
char **h_aliases;
int h_addrtype;
int h_length;
char **h_addr_list;
};
#define h_addr h_addr_list[0]
```

- **h_name**: Nombre oficial de la máquina.
- **h_aliases**: Array de nombres alternativos.
- **h_addrtype**: Tipo de dirección de retorno (AF_INET).
- **h_length**: Longitud de la dirección en bytes.
- **h_addr_list**: Array de direcciones de red para la máquina.
- **h_addr**: La primera dirección en h_addr_list.

En el caso de producirse algún error, devuelve NULL y establece la variable h_errno con el código de error.

3.2 Creación de un proceso hijo. Función fork().

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

pid_t fork(void);
http://linux.die.net/man/2/fork
```

La función fork() permite crear un proceso dentro del proceso principal (o proceso padre). El prototipo de la función fork() es el siguiente:

```
pid_t fork(void);
```

La función fork() no recibe ningún parámetro. Al término de su ejecución existirán dos procesos idénticos cuya única diferencia es el valor de la variable que devuelve la función. Este valor es 0 para el nuevo proceso creado (proceso hijo) y para el proceso principal (proceso padre) esa variable es un número mayor que 0 (corresponde con el identificador de proceso del hijo). Si se produce un error, el valor de la variable será -1.

Ejemplo:

```
pid_t cpid;
...
cpid = fork();
if (cpid == -1) { perror("fork"); exit(EXIT_FAILURE); }
if (cpid > 0)
{ // ode executed by parent
printf("Child PID is %ld\n", cpid);
```

```
...
}
else { // Code executed by child
printf("My PID is %ld\n", (long) getpid());
...
}
...
```

El proceso hijo hereda todas las variables del proceso padre. Esta función se puede utilizar para dotar de concurrencia a los servidores TCP.

3.3 Servidores TCP concurrentes y secuenciales

Después de realizada la conexión TCP, en el servidor se crea un socket que es el que se emplea en la comunicación con el cliente.

Si tras acabar de ofrecer el servicio a un cliente (sucesión de envíos y respuestas entre servidor y cliente), el servidor volviera a la función `accept`(), este podría aceptar una nueva petición de conexión por parte de otro cliente. Este modo de operación se denomina *secuencial.* Hasta que no he acabado de servir a un cliente no puedo ofrecer servicio al siguiente.

Si por el contrario queremos que nuestro servidor sea *concurrente,* esto es, poder servir a varios clientes al mismo tiempo. Debemos posibilitar que el servidor pueda aceptar peticiones de conexión mientras se encuentra sirviendo a los clientes.

En principio esto es posible puesto que el socket que se emplea para aceptar peticiones de conexión es distinto al socket que se usa para comunicarse con el cliente. Sin embargo si sólo se tiene un proceso, sólo es posible volver a aceptar peticiones una vez se ha completado el servicio al cliente.

Para poder llevarlo a cabo es necesario crear nuevos procesos cada vez que se acepta un nuevo cliente. De esta forma, los hijos creados atenderán a cada uno de los clientes según vayan llegando mientras el proceso padre vuelve a aceptar a nuevos clientes y a crear nuevos hijos que los atiendan.

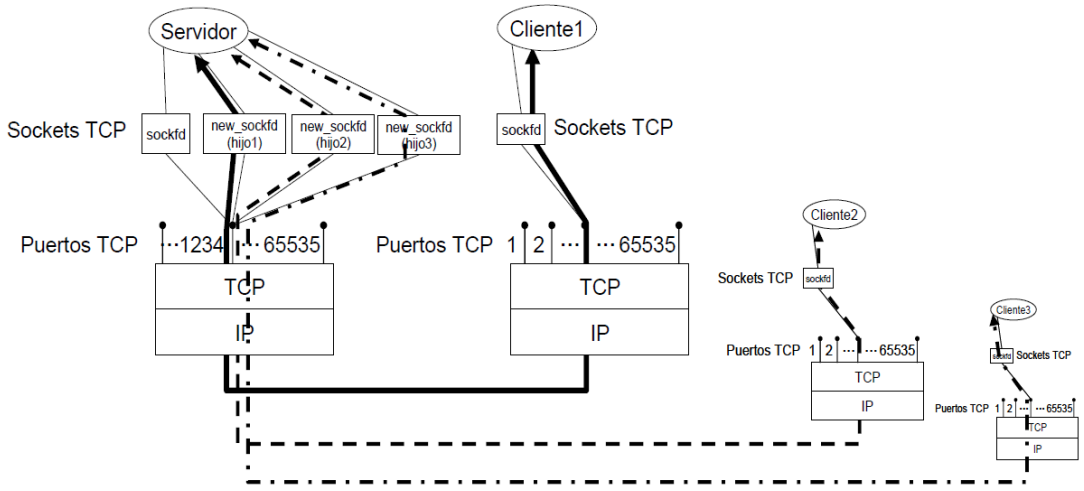
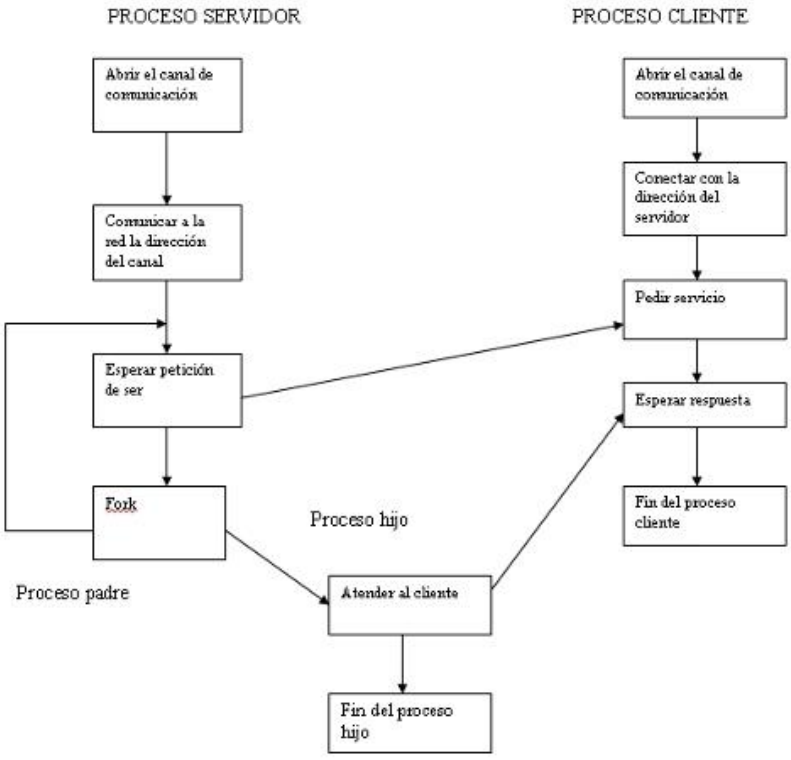


Figura 3-1: Conexiones en un servidor TCP concurrente

A continuación se muestran los pasos a realizar, invocando diversas funciones, cuyo funcionamiento ha sido detallado.

Servicio orientado a la conexión (TCP) concurrente		
Cliente	Servidor	
	Padre	Hijo(s)
	`socket()`	
	`bind()`	
	`listen()`	
	`for(;;){`	
`socket()`	`accept()`	
`[bind()]`		
`connect()`	`fork()`	
`send()`	`}`	`recv()`
`recv()`		`send()`



3.4 Servidores UDP concurrentes y secuenciales

Al contrario que los servidores basados en sockets TCP, los servidores basados en sockets UDP, tienen un único socket que emplean para enviar y recibir información de cualquier cliente.

Por ello, a priori cualquier servidor basado en un socket UDP puede ser concurrente con un único proceso puesto que simplemente tiene que responder (esto es hacer una llamada a `sendto()`) inmediatamente después de recibir los datos del cliente (a través de un `recvfrom()`).

Sin embargo, existen servidores en los que debido a la propia lógica del servicio prestado la concurrencia no es tan sencilla. Estos servicios son todos aquellos que guarden el “estado” de los clientes. Pongamos como ejemplo un servicio que sea un juego de azar. El “estado” de cada cliente será el balance de ganancias o perdidas de cada cliente. Si bien el servidor puede recibir la jugada del cliente, calcular si gana o pierde y responder con el resultado, debe ser capaz de disociar a que cliente corresponde cada una de las jugadas que recibe y modificar su estado de la manera apropiada.

4. EJEMPLO.

4.1 Introducción

En este ejemplo, el cliente nos pedirá que introduzcamos un mensaje por teclado y se lo enviará al servidor que nos lo mostrará por pantalla.

Se debe ejecutar el cliente y el servidor en hosts independientes (aunque dentro de un mismo equipo se puede hacer desde dos ventanas de terminales distintas). Iniciaremos en primer lugar el servidor, y se le deberá asignar un puerto libre con cualquier número entre 2000 y 65535. . Si el puerto está disponible, el servidor se bloqueará hasta que reciba una conexión desde el cliente (no tiene que hacer nada hasta que se establezca una conexión).

Esta es una línea de comandos típica del servidor:

```
./server 55000
```

Para ejecutar el cliente se le tiene que pasar dos argumentos, el nombre del host en el que el servidor está en ejecución y el número de puerto en el que el servidor está escuchando las conexiones Aquí está la línea de comandos del cliente para conectar con el servidor:

```
./client nombrehost 55000
```

Si ambos se están ejecutando en el mismo equipo, para averiguar el nombre del host basta con ejecutar el comando *hostname* en el servidor.

4.2 Descripción del programa Servidor.

Se pasa a describir las partes más importantes del programa.

En primer lugar colocamos las librerías que contienen declaraciones utilizadas en la mayoría de programas que trabajan con sockets.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

A continuación declaramos la siguiente función que se invoca cuando una llamada al sistema falla. Se muestra un mensaje sobre el error y luego terminará el programa.

`perror`. Más información sobre la función al final (*)

```
void error(const char *msg)
  {
  perror(msg);
  exit(1);
  }
```

A continuación declaramos la función principal y las variables principales.

```
 int main (int argc, char *argv[])
  {
  int sockfd, newsockfd, portno;
```

Donde `sockfd` y `newsockfd` son descriptores de fichero. Estas dos variables almacenan los valores devueltos por la llamada al sistema socket. `portno` almacena el número de puerto en el que el servidor acepta conexiones.

El servidor lee los caracteres de la conexión socket en este buffer.

```
 char buffer[256];
```

A `sockaddr_in` es una estructura que contiene una dirección de Internet.

```
 struct sockaddr_in serv_addr, cli_addr;
```

Esta estructura se define en `netinet/in.h` (**)

La variable `serv_addr` contendrá la dirección del servidor, y `cli_addr` contendrá la dirección del cliente que se conecta al servidor.

El usuario tiene que pasar el número de puerto en el que el servidor aceptará las conexiones como argumento. Este código muestra un mensaje de error si el usuario no lo hace.

```
if (argc <2)
  {
  fprintf(stderr, "Error, no hay ningún puerto");
  exit(1);
  }
```

La llamada a `socket()` crea un nuevo socket. Recibe tres argumentos. El primero es el dominio, el segundo es el tipo de socket, el tercero es el protocolo (si el argumento es 0, el sistema selecciona el más adecuado)

```
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd <0)
        error ("Error de apertura de socket");
```

La llamada al socket devuelve un entero, si la llamada falla devuelve -1 y el programa muestra un mensaje de error y se cierra

La función `bzero()` establece todos los valores en un búfer a cero. Tiene dos argumentos, el primero es un puntero a la memoria intermedia y el segundo es el tamaño del búfer. Por lo tanto, esta línea inicializa `serv_addr` a ceros.

```
 bzero((char*)&serv_addr,sizeof(serv_addr));
```

El número de puerto en el que el servidor escuchará las conexiones se pasa como un argumento, y esta declaración utiliza el `atoi()` para convertir una cadena de dígitos en un número entero.

```
portno = atoi(argv[1]);
```

La variable `serv_addr` es una estructura de tipo `struct sockaddr_in`. Esta estructura tiene cuatro campos, el primer campo es `short sin_family` que contiene un código para la familia de direcciones. Siempre se debe establecer en la constante simbólica `AF_INET` .

```
serv_addr.sin_family = AF_INET;
```

El segundo campo es `unsigned short sin_port` , que contiene el número de puerto.

```
serv_addr.sin_port = htons(portno);
```

El tercer campo es una estructura del tipo de `struct in_addr` que contiene sólo un único campo `unsigned long s_addr` .

```
serv_addr.sin_addr.s_addr = INADDR_ANY;
```

Este campo contiene la dirección IP de la máquina. (`INADDR_ANY` obtiene esta dirección.)

La llamada a `bind()` enlaza un conector a una dirección, en este caso la dirección de la máquina actual y el número de puerto en el que el servidor se ejecutará. Toma tres argumentos, el descriptor de fichero socket, la dirección a la que está obligado, y el tamaño de la dirección a la que está enlazado.

El segundo argumento es un puntero a una estructura de tipo `sockaddr` , pero lo que ha pasado es una estructura de tipo `sockaddr_in` , por lo que se debe convertir al tipo correcto.

```
if (bind (sockfd,(struct sockaddr*)&serv_addr,sizeof(serv_addr)) <0)
      error("Error en la conexión");
```

La llamada `listen()` permite que el proceso quede a la escucha de las conexiones. El primer argumento es el descriptor de fichero socket, y el segundo es el tamaño de la cola de registro, es decir, el número de conexiones que pueden esperar mientras que el proceso esta operando con una conexión en particular. (5 es el tamaño máximo permitido por la mayoría de los sistemas).

```
listen(sockfd,5);
```

La llamada al sistema `accept()` hace que el proceso se bloquee hasta que un cliente se conecta al servidor, el proceso se despierta cuando una conexión de un cliente se ha establecido con éxito.

```
clilen = sizeof(cli_addr);
newsockfd = accept (sockfd,(struct sockaddr*)&cli_addr,&clilen);
if (newsockfd <0)
      error("Error en aceptar");
```

Devuelve un nuevo descriptor de fichero, y todas las comunicaciones en este sentido se deben hacer con el nuevo descriptor. El segundo argumento es un puntero de

referencia a la dirección del cliente en el otro extremo de la conexión, y el tercer argumento es el tamaño de esta estructura.

Se debe tener en cuenta que sólo se llega a este punto después de que un cliente se ha conectado a nuestro servidor. Este código inicializa el buffer mediante la función `bzero()`, y luego lee desde el socket. La llamada de lectura utiliza el nuevo descriptor de fichero, el devuelto por `accept()` , no el descriptor de archivo original devuelto por `socket()`.También se bloqueará `read()`hasta que haya algo para leer, es decir, después de que el cliente ha ejecutado un `write()`.

```
bzero (buffer,256);
n =read(newsockfd,buffer,255);
if (n <0)
      error("Error al leer del socket");
printf("Aquí está el mensaje:% s ", Buffer);
```

Una vez que la conexión ha sido establecida, ambos extremos pueden leer y escribir en la conexión.

```
n = write(newsockfd,"Tengo el mensaje",18);
if (n <0)
      error("Error al escribir en el socket");
```

Todo lo escrito por el cliente será leído por el servidor, y todo lo escrito por el servidor será leído por el cliente. Este código simplemente escribe un mensaje corto al cliente.

```
return 0;
 }
```

Con esto terminamos el programa principal.

4.3 Descripción del programa Cliente.

Es bastante similar a lo expuesto anteriormente.

En primer lugar colocamos las librerías que contienen declaraciones utilizadas en la mayoría de programas que trabajan con sockets.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
```

Los archivos de cabecera son los mismos que para el servidor con una adición. El archivo `netdb.h` define la estructura `hostent` (***).

La función `error()`es idéntica a la del servidor, así como las variables `sockfd`, `portno`, y `n`

```
void error(char * msg)
  {
  perror(msg);
  exit(0);
  }

 int main (int argc, char *argv[])
  {
  int sockfd, portno, n;
  struct sockaddr_in serv_addr;
  struct hostent *server;
```

.La variable `serv_addr` contendrá la dirección del servidor al que nos queremos conectar. La variable `server` es un puntero a una estructura del tipo `hostent`.

El resto de código es similar a lo expuesto para el servidor.

```
  char buffer[256];
  if (argc <3)
        {
        fprintf(stderr, "el puerto host %s está en uso", Argv[0]);
        exit(0);
        }
  portno = atoi(argv[2]);
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd <0)
        error("Error de apertura de socket ");
```

La variable `argv[1]` contiene el nombre de un host a través de Internet, por ejemplo, `cs.rrr.org`. (Si ejecutamos ambos procesos en una sola máquina, será el *hostname* de la misma)

```
  server = gethostbyname(argv[1]);
  if (server == NULL)
        {
        fprintf (stderr, "Error, no hay host ");
         exit(0);
        }
```

Si esta estructura es NULL, el sistema no pudo encontrar una máquina con este nombre.

```
 bzero((char *) &serv_addr, sizeof(serv_addr));
 serv_addr.sin_family = AF_INET;
 bcopy((char *) server->h_addr,(char *)&serv_addr.sin_addr.s_addr,
 server->h_length);
 serv_addr.sin_port = htons(portno);
```

Este código establece los ámbitos en los `serv_addr`.

La función `connect` es llamada por el cliente para establecer una conexión con el servidor. Toma tres argumentos, el descriptor de fichero socket, la dirección de la máquina a la que quiere conectarse (incluyendo el número de puerto), y el tamaño de esta dirección.

```
 if (connect(sockfd,(struct sockaddr*),&serv_addr,sizeof (serv_addr))
     <0)
  error("Error de conexión");
```

El resto del código pide al usuario que introduzca un mensaje, usa `fgets` para leer el mensaje de la entrada estándar, escribe el mensaje en el socket, lee la respuesta y muestra esta respuesta en la pantalla.

```
printf("Por favor, introduzca el mensaje:");
bzero(buffer,256);
fgets(buffer,255,stdin);
n = write(sockfd,buffer,strlen(buffer));
if (n <0)
     error("Error al escribir en socket");
bzero(buffer,256);
n = read(sockfd,buffer,255);
if (n <0)
     error("Error al leer del socket");
printf ("%s ",buffer);
close(sockfd);
return 0;
}
```

Todo lo visto anteriormente nos lleva a una única ejecución de los servicios del servidor cuando son requeridos por el cliente. Si deseamos que el servidor siga proporcionando esos servicios sin tener que ser ejecutado de nuevo, necesitamos que el proceso sea concurrente, para lo cual debemos realizar una serie de modificaciones en el código del servidor.

4.4 Descripción de Servidor concurrente y secuencial.

El servidor debe funcionar de forma indefinida y debe tener la capacidad de manejar un número de conexiones simultáneas, cada una en su propio proceso. Esto se realiza normalmente por la bifurcación de un nuevo proceso para manejar cada nueva conexión.

El siguiente código tiene una función ficticia llamada `dostuff(int sockfd)`. Esta función se encargará de la conexión después de que se haya establecido y prestará los servicios que el cliente solicita.

La declaramos al principio junto a la función de error.

```
void dostuff(int);
```

Para permitir que el servidor para manejar múltiples conexiones simultáneas, hacemos los siguientes cambios en el código.

1. Poner en `accept` el siguiente código en un bucle infinito.
2. Después de establecerse una conexión, se llamará a `fork()####` para crear un nuevo proceso.
3. El proceso hijo se cerrará `sockfd ####`y se llamaráa `#dostuff ####`, pasando el descriptor del archivo del nuevo socket como argumento. Cuando los dos procesos han terminado su transferencia, se indicará en `dostuff()####` y este terminará.
4. El proceso padre cierra `newsockfd ####`. Por todo esto es en un bucle infinito, se vuelve a la instrucción aceptar que esperar a la próxima conexión.

Este es el código:

```
while (1)
 {
   newsockfd = accept(sockfd,(struct sockaddr *) &cli_addr, &clilen);
   if (newsockfd < 0)
        error("ERROR en aceptar ");
   pid = fork();
   if (pid < 0)
        error("ERROR en fork");
   if (pid == 0)
        {
         close(sockfd);
         dostuff(newsockfd);
         exit(0);
         }
   else
        close(newsockfd);
 } /* end of while */
```

Y para la gestión independiente de cada una de las conexiones añadimos la mencionada función `dostuff(int sockfd)`

```
void dostuff (int sock)
{
   int n;
   char buffer[256];

   bzero(buffer,256);
   n = read(sock,buffer,255);
   if (n < 0)
         error("ERROR al leer del socket");
   printf("Aquí está el mensaje: %s\n",buffer);
   n = write(sock,"Tengo el mensaje",18);
   if (n < 0)
         error("ERROR al escribir en el socket");
}
```

(*)

```
NOMBRE
     perror, errno - Mensajes de error del sistema

SINOPSIS
     vacío perror (s)
     char * s;

     # Include <errno.h>

     int sys_nerr;
     char * sys_errlist [];
     int errno;

DESCRIPCIÓN
       perror() produce un mensaje de error breve en la norma error que
       describe el último error encontrado durante una llamada para un sistema
       o una función de biblioteca. Si s no es un puntero NULL y no apunta a
       una cadena, la cadena apunta a se imprime, seguida por dos puntos,
       seguido de un espacio, si lowed por el mensaje y una nueva línea. Si s
       es un puntero NULL o apunta a una cadena nula, sólo se imprime el
       mensaje, seguido de una nueva línea. Para ser de mayor utilidad, el
```

```
argumento cadena debe incluir el nombre del programa que se incurra el
error. El número de error se ha tomado del exterior variable errno, que
se establece cuando los errores ocurrir, pero no borra cuando no errónea
llamadas.Para simplificar el formato de los mensajes de la variante, el
vector de mensaje sys_errlist cadenas se proporciona, errno se puede
utilizar como un índice en esta tabla para conseguir la cadena de
mensajes sin       la nueva línea. sys_nerr es el número de mensajes
siempre en la mesa, debe ser revisado porque el error de nuevo códigos
se pueden añadir al sistema antes de que se agregan a la mesa.
```

```
(**)

struct sockaddr_in
  {
  short sin_family; / * debe ser AF_INET * /
  u_short sin_port;
  struct in_addr sin_addr;
  sin_zero char[8]; / * No se utiliza, debe ser cero * /
  };
```

```
(***)

struct hostent
  {
  char * h_name; / * nombre oficial del host * /
  char ** h_aliases; / * lista de alias * /
  int h_addrtype; / * tipo de dirección de host * /
  int h_length; / * longitud de la dirección * /
  char ** h_addr_list; / * lista de direcciones de nombres del servidor * /

  # Define h_addr h_addr_list [0]
  / * Dirección, para la compatibilidad hacia atrás * /
  };

  Define un equipo host en Internet.  Los miembros de esta estructura son:

  h_name Nombre oficial del host.
  h_aliases
  h_addrtype El tipo de dirección que se devuelve; actualmente, siempre
  AF_INET.
  h_length La longitud, en bytes, de la dirección.
  h_addr_list Un puntero a una lista de direcciones de red para el host que se
  ha llamado.  Direcciones de host se evuelven en orden de bytes de red.
```

En las páginas siguientes se añaden los listados completos del cliente y los dos servidores.

4.5 Listados completos de código.

CLIENTE.C

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(const char *msg)
  {
    perror(msg);
    exit(0);
  }

int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];
    if (argc < 3)
        {
         fprintf(stderr,"El puerto host  %s está en uso\n", argv[0]);
         exit(0);
       }

    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    if (sockfd < 0)
        error("ERROR de apertura de socket");

    server = gethostbyname(argv[1]);

    if (server == NULL)
       {
         fprintf(stderr,"ERROR, no hay host \n");
         exit(0);
       }

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;

    bcopy((char *)server->h_addr,(char*)&serv_addr.sin_addr.s_addr,server->h_length);

    serv_addr.sin_port = htons(portno);

    if (connect(sockfd,(struct sockaddr *)&serv_addr,sizeof(serv_addr)) < 0)
        error("ERROR de conexión");

    printf("Por favor, introduzca el mensaje:");

    bzero(buffer,256);
    fgets(buffer,255,stdin);
    n = write(sockfd,buffer,strlen(buffer));

    if (n < 0)
         error("ERROR al escribir en socket");

    bzero(buffer,256);
    n = read(sockfd,buffer,255);

    if (n < 0)
         error("ERROR al leer del socket");

    printf("%s\n",buffer);
    close(sockfd);
    return 0;
}
```

SERVIDOR.C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(const char *msg)
  {
    perror(msg);
    exit(1);
  }

int main(int argc, char *argv[])
{
     int sockfd, newsockfd, portno;
     socklen_t clilen;
     char buffer[256];

     struct sockaddr_in serv_addr, cli_addr;
     int n;

     if (argc < 2)
       {
         fprintf(stderr,"ERROR, no hay ningún puerto\n");
         exit(1);
       }
     sockfd = socket(AF_INET, SOCK_STREAM, 0);

     if (sockfd < 0)
        error("ERROR de apertura de socket");

     bzero((char *) &serv_addr, sizeof(serv_addr));

     portno = atoi(argv[1]);

     serv_addr.sin_family = AF_INET;
     serv_addr.sin_addr.s_addr = INADDR_ANY;
     serv_addr.sin_port = htons(portno);

     if (bind(sockfd, (struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
        error("ERROR en la conexión");

     listen(sockfd,5);
     clilen = sizeof(cli_addr);

     newsockfd = accept(sockfd,(struct sockaddr *) &cli_addr,&clilen);

     if (newsockfd < 0)
        error("ERROR en aceptar");

     bzero(buffer,256);
     n = read(newsockfd,buffer,255);

     if (n < 0)
        error("ERROR al leer del socket");

     printf("Aquí está el mensaje: %s\n",buffer);

     n = write(newsockfd,"Tengo el mensaje",18);
     if (n < 0)
        error("ERROR al escribir en el socket");

     close(newsockfd);
     close(sockfd);
     return 0;
 }
```

SERVIDOR2.C

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void dostuff(int);
void error(const char *msg)
  {
    perror(msg);
    exit(1);
  }

int main(int argc, char *argv[])
{
     int sockfd, newsockfd, portno, pid;
     socklen_t clilen;
     struct sockaddr_in serv_addr, cli_addr;

     if (argc < 2)
       {
         fprintf(stderr,"ERROR, no hay ningún puerto\n");
         exit(1);
       }
     sockfd = socket(AF_INET, SOCK_STREAM, 0);

     if (sockfd < 0)
        error("ERROR de apertura de socket");

     bzero((char *) &serv_addr, sizeof(serv_addr));

     portno = atoi(argv[1]);

     serv_addr.sin_family = AF_INET;
     serv_addr.sin_addr.s_addr = INADDR_ANY;
     serv_addr.sin_port = htons(portno);

     if (bind(sockfd, (struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
        error("ERROR en la conexión");

     listen(sockfd,5);
     clilen = sizeof(cli_addr);

     while (1)
       {
         newsockfd = accept(sockfd,(struct sockaddr *) &cli_addr, &clilen);

         if (newsockfd < 0)
            error("ERROR en aceptar");

         pid = fork();

         if (pid < 0)
            error("ERROR en fork");

         if (pid == 0)
             {
              close(sockfd);
              dostuff(newsockfd);
              exit(0);
            }
         else
              close(newsockfd);
         }
    close(sockfd);
    return 0;
}

/*Gestiona todas las comunicaciones una vez que la conexión ha sido establecida*/

void dostuff (int sock)
   {
```

```
int n;
char buffer[256];

bzero(buffer,256);
n = read(sock,buffer,255);

if (n < 0)
  error("ERROR al leer del socket");

printf("Aquí está el mensaje: %s\n",buffer);
n = write(sock,"Tengo el mensaje",18);
if (n < 0)
   error("ERROR al escribir en el socket");
}
```

4.6 Bibliografía.

- UNIX, Programación avanzada, Ra-ma (2004). *Francisco Manuel Márquez García.*
- Redes de Computadoras: Un Enfoque Descendente, Pearson Educación (2010). *J. Kurose & K.W. Ross*
- Redes de computadoras, Prentice Hall (2003). *Andrew S. Tanenbaum*
- The Linux TCP/IP Stack: Networking for Embedded Systems (2004). *Charles River Media.*
- Comunicaciones en UNIX, McGraw-Hill, (1992). *Rifflet, J. M.*
- UNIX Network Programming, Prentice Hall (1990). *W. R. Stevens.*
- Internetworking with TCP/IP, Volume I-III, Prentice Hall (1993). *D. E. Comer. David L.Stevens.*
- Internetworking with TCP/IP. Client-Server programming and applications. Volume III. Linux-POSIX Socket version, Prentice Hall (2001). *D. E. Comer, David L.Stevens.*
- TCP/IP Sockets in C: Practical Guide for Programmers, 2 *Edition by Michael J. Donahoo, Kenneth L. Calvert*
- TCP/IP Protocol Suite, McGraw-Hill (2003). *Behrouz A. Forouzan.*
- http://linux.die.net/man