UNIVERSIDAD DE CANTABRIA

DEPARTAMENTO DE INGENIERÍA DE COMUNICACIONES

GRUPO DE INGENIERÍA DE COMUNICACIONES

LABORATORIO DE TELEMÁTICA:

MANUAL DE SOCKETS EN C

TABLA DE CONTENIDOS

1
INTRODUCCIÓN

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**.

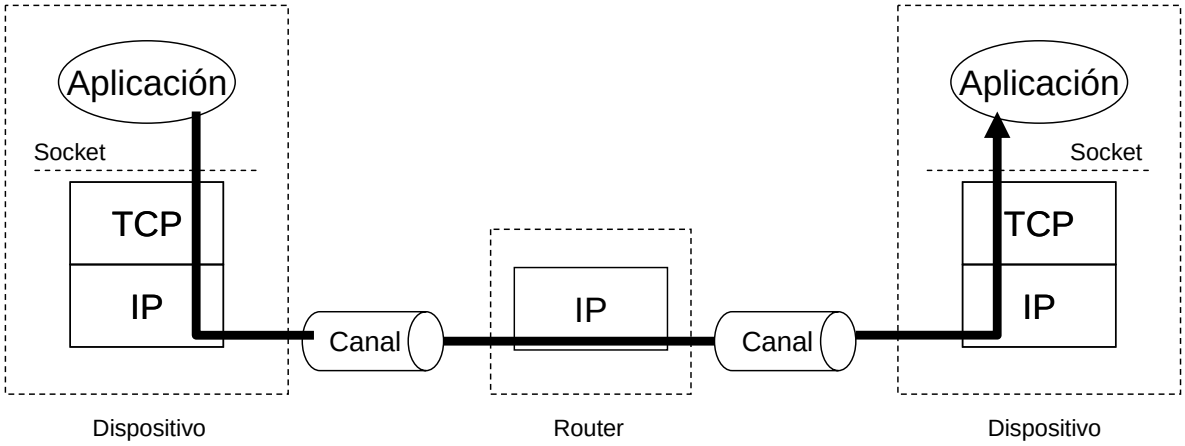


Figura 1-1: Comunicación entre aplicaciones en una red TCP/IP

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. En este manual, como ya se ha dicho anteriormente, nos centraremos en la pila de protocolos TCP/IP.

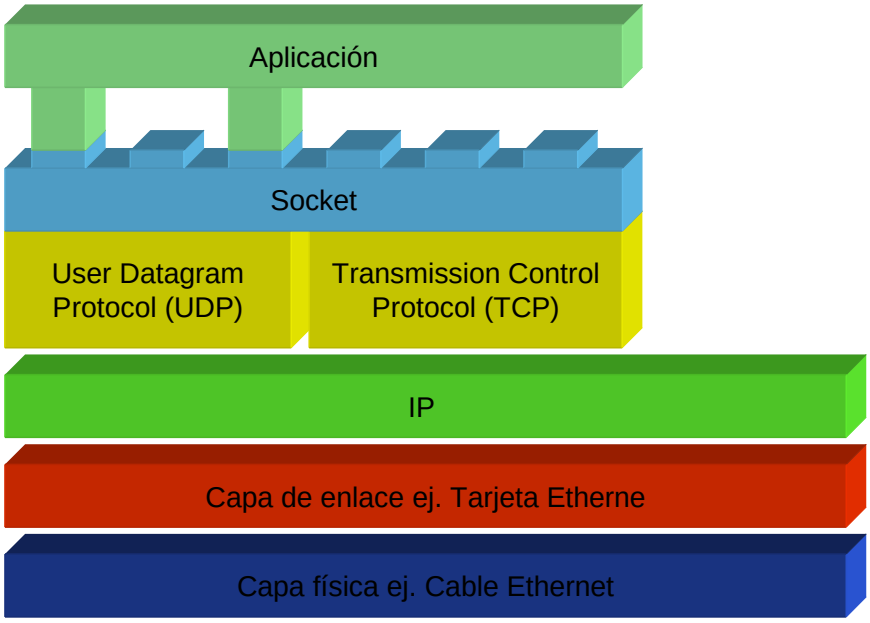


Figura 1-2: Interfaz de sockets

Un socket TCP/IP está unívocamente identificado por una dirección IP, un protocolo de nivel de transporte (TCP o UDP), y un número de puerto. En este manual se verá como se le pueden asociar estos parámetros a los sockets creados.

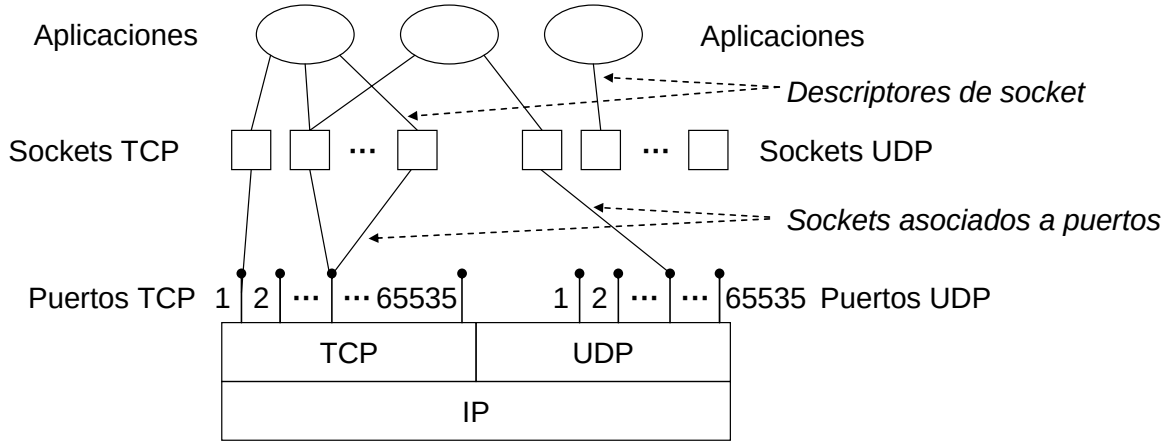


Figura 1-3: Sockets, protocolos y puertos

La Figura 1-3 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 1-3, 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
SOCKETS EN C

2.1 CONCEPTOS BÁSICOS

En esta sección se repasan algunos aspectos que es necesario conocer cuando se trabaja con sockets.

2.1.1 Dominios de comunicación

Los sockets se crean dentro de lo que se denomina un dominio de comunicación que define cual será la pila de protocolos que se usará en la comunicación

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

Dominio	Proposito
PF_UNIX, PF_LOCAL	Procesos que se comunican en un mismo sistema UNIX
PF_INET	Procesos que se comunican usando una red IPv4
PF_INET6	Procesos que se comunican usando una red IPv6

PF_IPX	Procesos que se comunican usando una red Novell
PF_NETLINK	Comunicación con procesos del kernel
PF_X25	ITU-T X.25 / ISO-8208 protocol
PF_AX25	Amateur radio AX.25
PF_ATMPVC	Acceso a PVCs ATM
PF_APPLETALK	Appletalk
PF_PACKET	Interfaz con paquetes de bajo nivel

A lo largo de la práctica sólo se usarán sockets creados bajo el dominio PF_INET.

2.1.2 Tipos de sockets

En el dominio PF_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.

2.1.3 Orden de los bytes

Una particularidad de los Sistemas Operativos es la forma en la que estos almacenan los bytes en memoria.

Para evitar posibles errores, la pila de protocolos TCP/IP define un orden de los bytes que es independiente del tipo de máquina. Este orden de los bytes se denomina orden de red (network byte order) y por su propia definición es equivalente al orden conocido como Big-endian.

Si se trabaja sobre una máquina Big-endian, no habrá que tener este factor en cuenta, sin embargo, cuando se trabaja sobre máquinas Little-endian, es necesario disponer la información en el orden de red.

Para esto existen las siguientes funciones de conversión:

`htons()` host to network short Convierte un short int de formato de máquina al formato de red.

`htonl()` host to network long Convierte un long int de formato de máquina al formato de red.

`ntohs()` network to host short Convierte un short int de formato de red al formato de máquina.

`ntohl()` network to host long Convierte un long int de formato de red al formato de máquina.

Ejemplo:

```
#include <netinet/in.h>
...
port = htons ( 3490 );
...
```

2.2 CREACIÓN DE UN SOCKET

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

int socket(int domain, int type, int protocol);

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

El formato 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. Para esta práctica, el dominio será siempre `PF_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`
- `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 <stdio.h>  /* for perror() */
#include <sys/types.h>
#include <sys/socket.h>
...
int sockfd;
sockfd = socket ( PF_INET, SOCK_STREAM, 0 );
if(sockfd < 0)
     perror(“Error creating the socket”);
...
```

2.3 DIRECCIÓN DE UN SOCKET

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; // PF_*
char sa_data[14]; // Dirección de protocolo.
};

struct sockaddr_in
{
short int sin_family; // PF_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). Esto permite que esta estructura pueda emplearse 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 PF_INET.
- `sin_port`: Representa el número de puerto, y debe estar en la orden de bytes de la red.

- `sin_addr`: Este campo representa la dirección IP. y se almacena en un formato específico. 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`.

Ejemplo:

```
...
struct sockaddr_in sin;
...
sin.sin_addr.s_addr=inet_addr("193.144.57.67");
```

La función inversa se denomina `inet_ntoa()`, que convierte una dirección en formato numérico en una cadena de caracteres.

Ejemplo:

```
...
printf("%s", inet_ntoa(sin.sin_addr));
...
```

- `sin_zero`: Se utiliza simplemente de relleno para completar la longitud de sockaddr.

2.4 ASOCIACIÓN DE UN SOCKET A UNOS PARÁMETROS DETERMINADOS

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

int bind(int local_s, const struct sockaddr *addr, int
addrlen);
```

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

La función `bind()` se utiliza para asociarlo a estos parámetros, una dirección IP y número de puerto de la máquina local a través del que se enviarán y recibirán datos. El formato 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:

```
...
struct sockaddr_in sin;
...
ret = bind (sockfd, (struct sockaddr *)&sin, sizeof (sin));
if(ret < 0)
     perror("Error binding the socket");
...
```

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 = PF_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);
...
```

Es importante destacar que todas las funciones de sockets esperan recibir como parámetro un puntero a una estructura `sockaddr`, por lo que es necesario realizar una conversión de tipos (cast) a este tipo, tal y como se ha realizado en el ejemplo anterior.

La función `bind()` (al igual que la función `socket()`, y en general todas las funciones relacionadas con sockets) devuelve –1 en caso de que se haya producido alguna situación de error, y establece la variable global `errno` al número de error ocurrido, por lo que se puede invocar directamente a la función `perror()` para mostrarlo por pantalla.

2.5 SERVICIO ORIENTADO A LA CONEXIÓN (TCP)

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

Servicio orientado a la conexión (TCP)	
Cliente	Servidor
	`socket()`
	`bind()`
	`listen()`
	`accept()`
`socket()`	
`[bind()]`	
`connect()`	
`send()`	`recv()`
`recv()`	`send()`

Tanto cliente como servidor deben crean un socket mediante la función `socket()`, para poder comunicarse.

La llamada a la función `bind()` en el cliente es opcional, ya que en caso de no ser invocada el sistema la ejecutará automáticamente asignándole un puerto libre (al azar). En cambio, en el servidor es obligatorio ejecutar la llamada a la función `bind()` para reservar un puerto concreto y conocido.

El servidor habilita su socket para poder recibir conexiones, llamando a la función `listen()`. En el cliente este paso no es necesario, ya que no recibirá peticiones de conexión de otros procesos.

El servidor ejecuta la función `accept()` y permanece en estado de espera hasta que un cliente se conecte.

El cliente usa la función `connect()` para realizar el intento de conexión. En ese momento la función `accept()` del servidor devuelve un parámetro que es un nuevo identificador de socket, que es utilizado para realizar la transferencia de datos por la red con el cliente, dejando así libre el socket previamente creado para poder atender nuevas peticiones.

Una vez establecida la conexión se utilizan las funciones `send()` y `recv()` con el descriptor de socket del paso anterior, para realizar la transferencia de datos.

Para finalizar la conexión se utiliza la función `close()`.

A continuación se detallan las llamadas a todas estas funciones.

2.5.1 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 formato 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()`.

2.5.2 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 formato 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 = PF_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);
...
```

2.5.3 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 formato 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 (PF_INET, SOCK_STREAM, 0 );


// Definir valores en la estructura server_addr.
```

```
sin.sin_family = PF_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);
...
```

2.5.4 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 cual significa que en el servidor se estaba esperando en la función `accept()` y se ha salido de ella y por tanto se ha creado el nuevo socket), la situación es la que se muestra en la Figura 2-1.

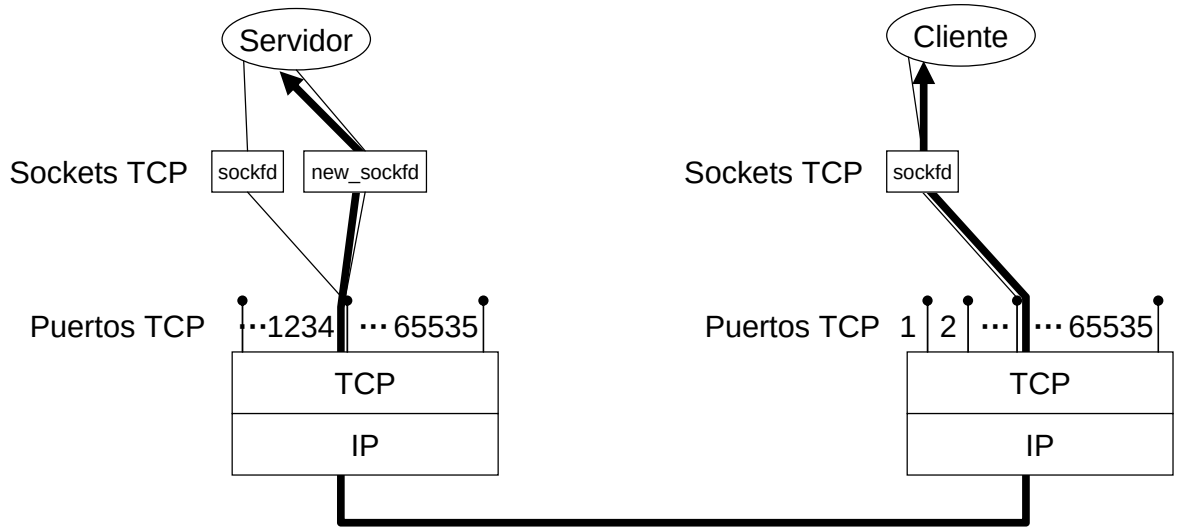


Figura 2-1: Sockets, protocolos y puertos

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.

2.5.5 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 formato de la función `send()` es el siguiente:

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

- `sockfd`: Idenficador de socket para enviar datos.
- `buf`: Puntero a los datos a ser enviados.
- `len`: Número de bytes a enviar.
- `flags`: Por defecto, 0. Para más información, consultar la ayuda en línea.

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 formato 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.

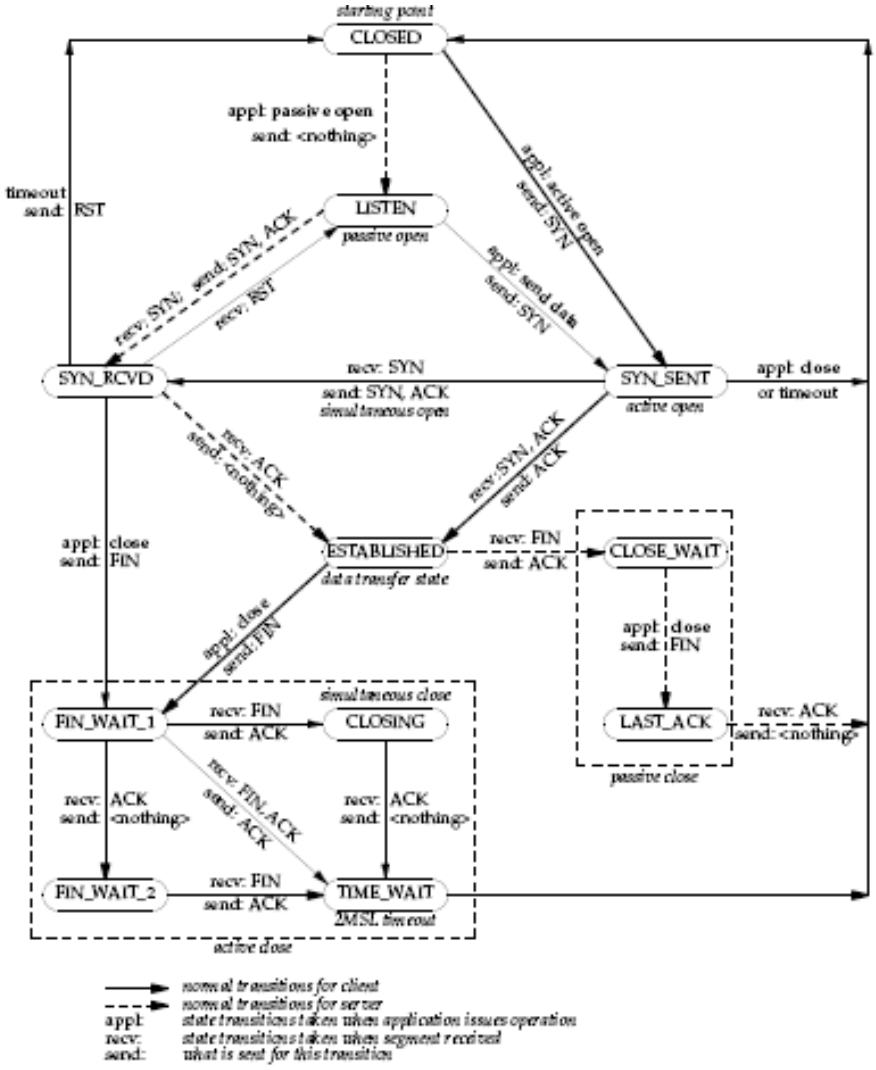
Ejemplo:

```
...
```

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

2.5.6 Diagrama de estados de una conexión TCP

En la Figura 2-2 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.



TCP state transition diagram.

Figura 2-2: 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()`.

2.6 SERVICIO NO ORIENTADO A LA CONEXIÓN (UDP)

A continuación se describen los pasos a realizar para la creación de un servicio no orientado a conexión.

Servicio orientado a la conexión (TCP)	
Cliente	Servidor
	socket()
	bind()
socket()	recvfrom()
[bind()]	
sendto()	
recvfrom()	sendto()

El proceso de creación y de asignación de nombres es similar al caso anterior. En cambio el envío y recepción de datos se realiza por medio de las funciones `sendto()` y `recvfrom()`, que presentan un comportamiento diferente a las anteriores.

Además en este caso, no es necesario realizar el establecimiento de la conexión y los sockets creados tanto en el servidor como en el cliente se pueden emplear para enviar o recibir datos hacia o desde cualquier socket remoto.

A continuación se detallan las llamadas a todas estas funciones.

2.6.1 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 formato 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. Para más información, consultar la ayuda en línea.
- `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 formato 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. Para más información, consultar la ayuda en línea.
- `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 en la secuencia de llamadas descrita en la Sección 2.5.6, 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));
...
```

2.7 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.

3
OTRAS FUNCIONES Y CONSIDERACIONES

En la Sección 2 se han presentado las funciones básicas del API de sockets que permiten llevar a cabo la comunicación entre dos procesos a través de una red TCP/IP.

En esta sección se presentan algunas funciones accesorias.

3.1 CREACIÓN DE UN PROCESO HIJO

```
#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 formato 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.

3.2 SERVIDORES TCP CONCURRENTES Y SECUENCIALES

Como vimos en la Sección 2, 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.

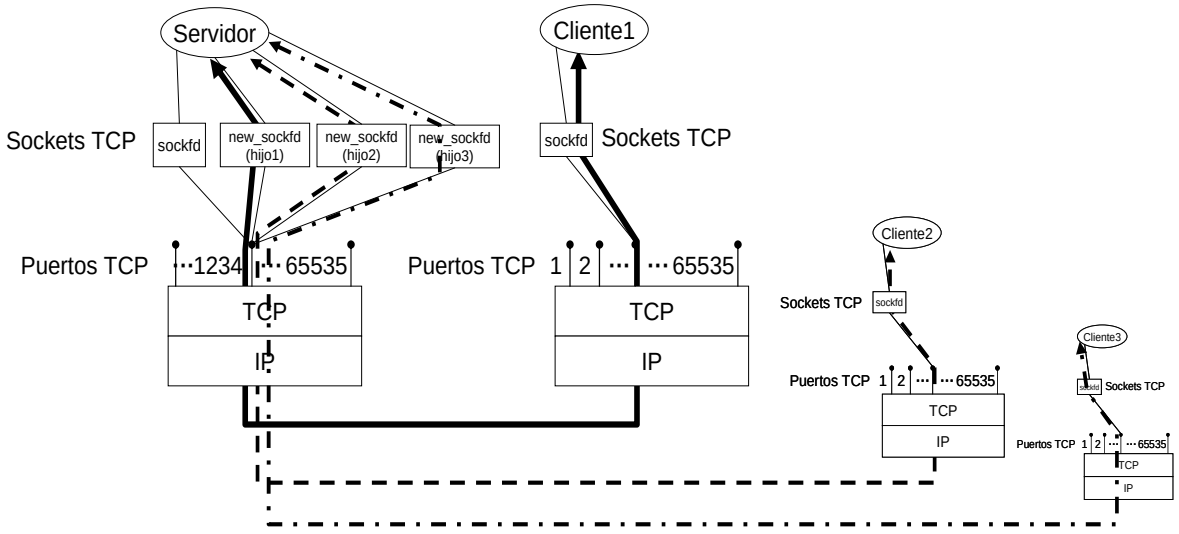


Figura 3-1: Conexiones en un servidor TCP concurrente

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(;;){`
	`accept()`	

`socket()` *`[bind()]`*		

`connect()`
`send()`
`recv()`
	`fork()`
	`}`
		`recv()`
		`send()`

3.3 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 servicios 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 ser sin más concurrente a nivel de comunicación, esto es, 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
INFORMACIÓN ADICIONAL

La descripción detallada de todas las funciones del API de sockets en C se pueden encontrar en las páginas de man de Linux. Se puede acceder a ellas a través del comando `man 2 <nombre de la función>` o bien en la web http://linux.die.net/man/2/.

Otros manual con información adicional es:

[1] TCP/IP Sockets in C: Practical Guide for Programmers, 2 Edition by Michael J. Donahoo, Kenneth L. Calvert