Guía *Beej* de Programación en Redes

Uso de *sockets* de Internet

Brian "Beej" Hall
beej@piratehaven.org

Traducción al español a cargo de Damián Marqués Lluch <dmrq@arrakis.es>

Historial de Revisiones

Versión de la revisión: 1.0.0	Agosto de 1995	Revisado por: beej
Versión Inicial		
Versión de la revisión: 1.5.5	13 de Enero de 1999	Revisado por: beej
Versión más reciente en HTML.		
Versión de la revisión: 2.0.0	6 de Marzo de 2001	Revisado por: beej
Convertido a DocBook XML, correcciones, añadidos.		
Versión de la revisión: 2.3.1	8 de Octubre de 2001	Revisado por: beej
Corregidas algunas erratas, error de sintaxis en client.c, se añaden algunas cosas a la sección de Preguntas más comunes.		

Tabla de Contenidos

1. Introducción

¡Eh!, ¿la programación de *sockets* te quita el sueño? ¿Todo esto es demasiado complicado para desentrañarlo sólo con las páginas de **man**? Quieres hacer buenos programas para Internet, pero no tienes tiempo de pelearte con montones de `structs` para tratar de adivinar si tienes que ejecutar `bind()` antes que `connect()`, etc., etc.

Bueno, ¿sabes qué?, ya me he ocupado de todo ese molesto asunto y estoy deseando compartir la información con todo el mundo. Has venido al lugar adecuado. Este documento debería bastar para que un programador en C medianamente competente empiece a trabajar en un entorno de redes.

1.1. Destinatarios

Este documento se ha escrito a modo de tutorial, y no como una guía de referencia. Probablemente, es de máxima utilidad para personas que están empezando a programar con *sockets* y buscan donde apoyarse. No es, en absoluto, la guía completa de programación de *sockets*.

Sin embargo, con suerte será suficiente para que esas páginas de `man` empiecen a cobrar sentido... :-)

1.2. Plataforma y compilador

El código que se muestra en este documento se compiló en un PC con Linux usando el compilador **gcc** de Gnu. Sin embargo, debería poder compilarse en prácticamente cualquier otra plataforma que use **gcc**. Evidentemente, esto no te sirve si programas para *Windows*--consulta más abajo la sección sobre programación en *Windows*.

1.3. Sitio *web* Oficial

La ubicación oficial de este documento se encuentra en Chico, en la Universidad del Estado de California, en la URL: `http://www.ecst.csuchico.edu/~beej/guide/net/` .

1.4. Nota para programadores *Solaris/SunOS*

Si compilas para *Solaris* o *SunOS*, necesitas indicar algunos conmutadores adicionales en la línea de comandos para enlacer las bibliotecas adecuadas. Para conseguirlo simplemente añade "`-lnsl -lsocket -lresolv`" al final del comando de compilación, como sigue:

```
    $ cc -o server server.c -lnsl -lsocket -lresolv
```

Si todavía tienes errores puedes probar a añadir "`-lxnet`" al final de la línea de comandos. No sé qué es lo que hace exactamente, pero algunas personas parecen necesitarlo para compilar correctamente.

Otro lugar donde podrías encontrar problemas es en la llamada a `setsockopt()`. El prototipo

no es igual que el que tengo en mi Linux, por eso en lugar de:

```
        int yes=1;
```

teclea esto:

```
        char yes='1';
```

Como no tengo una máquina Sun no he probado ninguna de la información anterior--sólamente es lo que la gente me ha comentado por correo electrónico.

1.5. Nota para programadores *Windows*

Particularmente, *Windows* no me gusta en absoluto, y te animo a que pruebes Linux, BSD, o Unix. Dicho esto, puedes usar todo esto con *Windows* .

En primer lugar, ignora complemente todos los ficheros de cabecera que menciono. Todo lo que necesitas incluir es:

```
#include <winsock.h>
```

¡Espera! También tienes que ejecutar `WSAStartup()` antes de hacer nada con la biblioteca de *sockets*. El código para hacerlo es algo así como:

```
#include <winsock.h>
{
    WSADATA wsaData;   // Si esto no funciona
    //WSAData wsaData; // prueba esto en su lugar
    if (WSAStartup(MAKEWORD(1, 1), &wsaData) != 0) {
        fprintf(stderr, "WSAStartup failed.\n");
        exit(1);
    }
```

También tienes que decirle a tu compilador que enlace la biblioteca *Winsock*, que normalmente se llama `wsock32.lib` o `winsock32.lib` o algo así. Con VC++, esto puede hacerse con el menú `Project` , bajo `Settings...` Haz click en la pestaña `Link` y busca el cuadro titulado "`Object/library modules`". Añade "`wsock32.lib` " a esa lista.

Al menos eso es lo que he oído.

Por último, necesitas ejecutar `WSACleanup()` cuando hayas terminado con la biblioteca de *sockets*. Consulta la ayuda en línea para los detalles.

Una vez hecho esto, el resto de ejemplos de este tutorial deberían funcionar con algunas pocas excepciones. En primer lugar, no puedes usar `close()` para cerrar un *socket*--en su lugar, tienes que usar `closesocket()` . Ademas, `select()` sólo funciona con descriptores de *sockets* pero no con descriptores de ficheros (como 0 para `stdin`).

También hay una clase *socket* que puedes utilizar, `Csocket`. Consulta la ayuda de tu compilador para más información

Para más información acerca de *Winsock* lee el FAQ de *Winsock* .

Finalmente, he oído que *Windows* no tiene la llamada al sistema `fork()` que, desgraciadamente, uso en algunos de mis ejemplos. Quizás debas enlazar también una biblioteca `POSIX` o algo para que funcione, o puedes usar `CreateProcess()` en su lugar. `fork()` no tiene argumentos, y `CreateProcess()` tiene alrededor de 48 billones de argumentos. Si no estás dispuesto a lidiar con todos ellos, `CreateThread()` es un poco más fácil de digerir... Desgraciadamente, una explicación en profundidad de las técnicas multihebra (*multithreading*) va mucho más allá del alcance de este documento. ¡No puedo hablar de todo, sabes!

1.6. Política de respuesta a correos electrónicos

En general estoy abierto a ayudar con tus preguntas por correo electrónico, así que no dudes en escribirlas, pero no puedo garantizarte una respuesta. Llevo una vida bastante ocupada y a veces, sencillamente, no puede responder a tu pregunta. Cuando esto ocurre, generalmente borro el mensaje. No es nada personal; sólo es que nunca tendré tiempo para darte la respuesta tan detallada que necesitas.

Como norma general, cuanto más complicada sea tu pregunta, menos probable es que yo responda. Si puedes acotar tu pregunta antes de enviarla y asegurarte de incluir cualquier información que pueda ser de cierta relevancia (plataforma, compilador, mensajes de error que tienes, y cualquier otra cosa que pienses que puede ayudar) es mucho más probable que te responda. Si quieres más pistas, lee el documento de ESR: Cómo hacer preguntas inteligentes

Si no, trabájalo un poco más, trata de encontrar la respuesta, y si el problema se te sigue resistiendo escríbeme otra vez con la información que has reunido y a lo mejor es suficiente para que pueda ayudarte.

Ahora que te he machacado acerca de como debes y como no debes escribirme, me gustaría que supieras que *de verdad* aprecio todos los cumplidos que esta guía ha recibido a lo largo del tiempo. Es un auténtico acicate moral, y me alegra saber que se está usando con alguna finalidad positiva :-) ¡Gracias!

1.7. Replicación

Es una excelente idea si quieres replicar este sitio, bien sea en público o en privado. Si replicas este sitio en público y quieres que añada un enlace desde la página principal escríbeme a: < `beej@piratehaven.org` >.

1.8. Nota para traductores

Si quieres traducir esta guía a otra idioma, escríbeme a <`beej@piratehaven.org` > y añadiré un enlace a tu traducción desde la página principal.

No dudes en añadir tu nombre y tu dirección de correo a la traducción.

Lo siento, pero debido a restricciones de espacio no voy a poder alojar las traducciones.

1.9. *Copyright* y distribución

Contacta con < beej@piratehaven.org > para más información.

2. ¿Qué es un *socket*?

Continuamente oyes hablar de los "*sockets*", y tal vez te estás preguntando qué es lo que son exactamente. Bien, esto es lo que son: una forma de comunicarse con otros programas usando descriptores de fichero estándar de Unix.

¿Qué?

Vale--puede que hayas oído a algún *hacker* de Unix decir: "¡Hey, *todo* en Unix es un fichero!" A lo que esta persona se estaba refiriendo es al hecho de que cuando los programas de Unix hacen cualquier tipo de E/S, lo hacen escribiendo o leyendo un descriptor de fichero. Un descriptor de fichero no es más que un entero asociado a un archivo abierto. Pero (y aquí está el *quid* de la cuestión) ese fichero puede ser una conexión de red, una cola FIFO, un tubo [*pipe*], un terminal, un fichero real de disco, o cualquier otra cosa. ¡Todo en Unix *es* un fichero! Por eso, si quieres comunicarte con otro programa a través de Internet vas a tener que hacerlo con un descriptor de fichero, es mejor que lo creas.

"¿De dónde saco yo este descriptor de fichero para comunicar a través de la red, señor sabelotodo?" Esta es probablemente la última pregunta en la que piensas ahora, pero voy a contestarla de todas maneras: usas la llamada al sistema `socket()`. Te devuelve un descriptor de fichero y tú te comunicas con él usando las llamadas al sistema especializadas `send()` y `recv()` (**`man send`** , **`man recv`**).

"¡Pero, oye!" puede que exclames ahora. "Si es un descriptor de fichero, por qué, en el nombre de Neptuno, no puedo usar las llamadas normales `read()` y `write()` para comunicarme a través de un *socket* ." La respuesta corta es, "¡Sí que puedes!" La respuesta larga es, "Puedes usarlas, pero `send()` y `recv()` te ofrecen mucho más control sobre la transmisión de datos."

¿Y ahora qué? Qué tal esto: hay muchos tipos de *sockets*. Existen las direcciones de Internet DARPA (*sockets* de internet), rutas sobre nodos locales (*sockets* de Unix), direcciones X.25 del CCITT (*sockets* éstos de X.25 que puedes ignorar perfectamente) y probablemente muchos más en función de la modalidad de Unix que estés ejecutando. Este documento se ocupa únicamente de los primeros: los *sockets* de Internet.

2.1. Dos tipos de *sockets* de internet

¿Qué es esto? ¿Hay dos tipos de *sockets* de internet? Sí. Bueno no, estoy mintiendo. En realidad hay más pero no quería asustarte. Aquí sólo voy a hablar de dos tipos. A excepción de esta frase, en la que voy a mencionar que los *sockets* puros [*Raw Sockets*] son también muy potentes y quizás quieras buscarlos más adelante.

Muy bien. ¿Cuáles son estos dos tipos? El primer tipo de *sockets* lo definen los *sockets* de flujo [*Stream sockets*]; El otro, los *sockets* de datagramas [*Datagram sockets*], a los que en adelante me referiré, respectivamente como "`SOCK_STREAM`" y "`SOCK_DGRAM` ". En ocasiones, a los *sockets* de datagramas se les llama también "*sockets* sin conexión". (Aunque se puede usar `connect()` con ellos, si se quiere. Consulta `connect()` , más abajo.)

Los *sockets* de flujo definen flujos de comunicación en dos direcciones, fiables y con conexión.

Si envías dos ítems a través del *socket* en el orden "1, 2" llegarán al otro extremo en el orden "1, 2", y llegarán sin errores. Cualquier error que encuentres es producto de tu extraviada mente, y no lo vamos a discutir aquí.

¿Qué aplicación tienen los *sockets* de flujo? Bueno, quizás has oído hablar del programa **telnet**. ¿Sí? telnet usa *sockets* de flujo. Todos los carácteres que tecleas tienen que llegar en el mismo orden en que tú los tecleas, ¿no? También los navegadores, que usan el protocolo `HTTP` , usan *sockets* de flujo para obtener las páginas. De hecho, si haces telnet a un sitio de la *web* sobre el puerto 80, y escribes " GET / ", recibirás como respuesta el código `HTML`.

¿Cómo consiguen los *sockets* de flujo este alto nivel de calidad en la transmisión de datos? Usan un protocolo llamado "Protocolo de Control de Transmisión", más conocido como "`TCP`" (consulta el RFC-793 para información extremadamente detallada acerca de `TCP`). `TCP` asegura que tu información llega secuencialmente y sin errores. Puede que hayas oído antes "`TCP`" como parte del acrónimo " `TCP/IP` ", donde "`IP`" significa "Protocolo de Internet" (consulta el RFC-791 .) `IP` se encarga básicamente del encaminamiento a través de Internet y en general no es responsable de la integridad de los datos.

Estupendo. ¿Qué hay de los *sockets* de datagramas? ¿Por qué se les llama *sockets* sin conexión? ¿De qué va esto, en definitiva, y por qué no son fiables? Bueno, estos son los hechos: si envías un datagrama, puede que llegue. Puede que llegue fuera de secuencia. Si llega, los datos que contiene el paquete no tendrán errores.

Los *sockets* de datagramas también usan `IP` para el encaminamiento, pero no usan `TCP`; usan el "Protocolo de Datagramas de Usuario" o "`UDP`" (consulta el RFC-768 .)

¿Por qué son sin conexión? Bueno, básicamente porque no tienes que mantener una conexión abierta como harías con los *sockets* de flujo. Simplemente montas un paquete, le metes una cabecera `IP` con la información de destino y lo envías. No se necesita conexión. Generalmente se usan para transferencias de información por paquetes. Aplicaciones que usan este tipo de *sockets* son, por ejemplo, **tftp** y **bootp**.

"¡Basta!" puede que grites. "¿Cómo pueden siquiera funcionar estos programas si los datagramas podrían llegar a perderse?". Bien, mi amigo humano, cada uno tiene su propio protocolo encima de `UDP`. Por ejemplo, el protocolo tftp establece que, para cada paquete enviado, el receptor tiene que devolver un paquete que diga, "¡Lo tengo!" (un paquete "`ACK`"). Si el emisor del paquete original no obtiene ninguna respuesta en, vamos a decir, cinco segundos, retransmitirá el paquete hasta que finalmente reciba un `ACK` . Este procedimiento de confirmaciones es muy importante si se implementan aplicaciones basadas en `SOCK_DGRAM`.

2.2. Tonterías de bajo nivel y teoría de redes

Puesto que acabo de mencionar la disposición en capas de los protocolos, es el momento de hablar acerca de como funcionan las redes realmente, y de mostrar algunos ejemplos de como se construyen los paquetes `SOCK_DGRAM` . Probablemente, en la práctica puedes saltarte esta sección. Sin embargo no está mal como culturilla.

Figura 1. Encapsulación de datos.

¡Eh tíos, es hora de aprender algo sobre *Encapsulación de datos* ! Esto es muy, muy importante. Tan importante que podrías aprenderlo si te matricularas en el curso de redes aquí, en el estado de Chico ;-). Básicamente consiste en esto: un paquete de datos nace, el paquete se envuelve (se "encapsula") con una cabecera (y raramente con un pie) por el primer protocolo (por ejemplo el protocolo `TFTP`), entonces todo ello (cabecera de `TFTP` incluida) se encapsula otra vez por el siguiente protocolo (por ejemplo `UDP`), y otra vez por el siguiente (`IP`) y otra vez por el protocolo final o nivel (físico) del *hardware* (por ejemplo, *Ethernet*)

Cuando otro ordenador recibe el paquete, el *hardware* retira la cabecera *Ethernet*, el núcleo [*kernel*] retira las cabeceras `IP` y `UDP`, el programa `TFTP` retira la cabecera `TFTP` , y finalmente obtiene los datos.

Ahora puedo finalmente hablar del conocido *Modelo de redes en niveles* [*Layered Network Model*]. Este modelo de red describe un sistema de funcionalidades de red que tiene muchas ventajas sobre otros modelos. Por ejemplo, puedes escribir programas de *sockets* sin preocuparte de cómo los datos se transmiten físicamente (serie, *thin ethernet*, AUI, lo que sea) porque los programas en los niveles más bajos se ocupan de eso por ti. El *hardware* y la topología real de la red son transparentes para el programador de *sockets*.

Sin más preámbulos, te presento los niveles del modelo completo. Recuerda esto para tus exámenes de redes:

- Aplicación
- Presentación
- Sesión
- Transporte
- Red
- Enlace de datos
- Físico

El nivel físico es el *hardware* (serie, *Ethernet*, etc.) El nivel de aplicación está tan lejos del nivel físico como puedas imaginar--es el lugar donde los usarios interactúan con la red.

Sin embargo, este modelo es tan general que si quisieras podrías usarlo como una guía para reparar coches. Un modelo de niveles más consistente con Unix podría ser:

- Nivel de aplicación (*telnet, ftp, etc.*)
- Nivel de transporte entre Hosts (*TCP, UDP*)

- Nivel de Internet (*IP y encaminamiento*)
- Nivel de acceso a la red (*Ethernet, ATM, o lo que sea*)

En este momento, probablemente puedes ver como esos niveles se corresponden con la encapsulación de los datos originales.

¿Ves cuánto trabajo se necesita para construir un solo paquete? ¡Dios! ¡Y tienes que teclear la cabecera de los paquetes tú mismo, usando "**`cat`**"! Sólo bromeaba. Todo lo que tienes que hacer con los *sockets* de flujo es usar `send()` para enviar tus datos. Para los *sockets* de datagramas tienes que encapsular el paquete según el método de tu elección y enviarlo usando `sendto()`. El núcleo implementa los niveles de Internet y de Transporte por ti, y el *hardware* el nivel de acceso a la red. Ah, tecnología moderna.

Así finaliza nuestra breve incursión en la teoría de redes. Ah, olvidaba contarte todo lo que quería decir acerca del encaminamiento: ¡nada! Exacto, no voy a hablar de él en absoluto. el encaminador retira la información de la cabecera IP, consulta su tabla de encaminamiento, bla, bla, bla. Consulta el RFC sobre `IP` si de verdad te importa. Si nunca aprendes nada de eso probablemente sobrevivirás.

3. `structs` y manipulación de datos

Bueno, aquí estamos por fin. Es hora de hablar de programación. En esta sección me ocuparé de los diversos tipos de datos que se usan en la interfaz para redes que los *sockets* definen, porque algunos de ellos son difíciles de digerir.

Empecemos por el fácil: un descriptor de *socket*. Un descriptor de *socket* es del tipo siguiente:

```
    int
```

Nada más que un `int`.

A partir de aquí las cosas se complican, así que leételo todo y ten paciencia conmigo. Debes saber, para empezar, que existen dos formas distintas de ordenación de *bytes* (a veces se les llama "octetos"): primero el *byte* más significativo, o primero el *byte* menos significativo. A la primera forma se la llama "Ordenación de *bytes* de la red" [*Network Byte Order*]. Algunos ordenadores almacenan internamente los números según la Ordenación de *bytes* de la red, mientras que otros no. Cuando diga que algo tiene que seguir la Ordenación de *bytes* de la red, tendrás que llamar a alguna función (como por ejemplo `htons()`) para realizar la conversión desde la "Ordenación de *bytes* de máquina" [*Host Byte Order*]. Si no digo "Ordenación de *bytes* de la red", entonces puedes dejarlo en la Ordenación de *bytes* de máquina.

(Para los que sientan curiosidad, la "Ordenación de bytes de la red" también se conoce como ordenación *Big-Endian*.)

Mi Primer Struct™--`struct sockaddr`. Esta estructura mantiene información de direcciones de *socket* para diversos tipos de *sockets*:

```
    struct sockaddr {
        unsigned short    sa_family;    // familia de direcciones, AF_xxx
        char              sa_data[14];  // 14 bytes de la dirección del protocolo
    };
```

`sa_family` admite varios valores, pero será `AF_INET` para todo lo que hagamos en este documento. *`sa_data`* contiene una dirección y número de puerto de destino para el *socket*. Esto resulta bastante farrogoso, porque no resulta nada agradable tener que empaquetar a mano una dirección y un número de puerto dentro de *`sa_data`* .

Por eso, para manejar `struct sockaddr` más cómodamente, los programadores han creado una estructura paralela: `struct sockaddr_in` ("in" por "Internet".)

```
    struct sockaddr_in {
        short int          sin_family;  // familia de direcciones, AF_INET
        unsigned short int sin_port;    // Número de puerto
        struct in_addr     sin_addr;    // Dirección de Internet
        unsigned char      sin_zero[8]; // Relleno para preservar el tamaño
original de struct sockaddr
    };
```

Esta estructura hace más sencillo referirse a los elementos de la dirección de *socket*. Observa que *`sin_zero`* (que se incluye para que la nueva estructura tenga el mismo tamaño que un `struct sockaddr`) debe rellenarse todo a ceros usando la función `memset()`. Además, y

es este un detalle *importante*, un puntero a `struct sockaddr_in` puede forzarse [*cast*] a un puntero a `struct sockaddr` y viceversa. Así, aunque `socket()` exige un `struct sockaddr*` , puedes usar en su lugar un `struct sockaddr_in` y forzarlo en el último momento. Observa también que el *`sin_family`* se corresponde con el *`sa_family`* de `struct sockaddr` y debe asignársele siempre el valor " `AF_INET`". Por último, *`sin_port`* y *`sin_addr`* tienen que seguir la Ordenación de *bytes* de la red.

"Pero", podrías preguntar ahora, "¿cómo puede toda la estructura `struct in_addr sin_addr`, seguir la Ordenación de *bytes* de la red?" Esta pregunta requiere un cuidadoso examen de la estructura `struct in_addr` , una de las peores `unione`s que existen:

```
// Dirección de Internet (una estructura por herencia histórica)
struct in_addr {
    unsigned long s_addr; // Esto es un long de 32 bits, ó 4 bytes
};
```

Bueno, en el pasado *era* una `union`, pero esos tiempos parecen haber pasado. Celebro que así sea. De modo que, si has declarado la variable *`ina`* asignándole el tipo `struct sockaddr_in`, entonces *`ina.sin_addr.s_addr`* se refiere a la dirección `IP` de 4 *bytes* (según la Ordenación de *bytes* de la red). Observa que, aunque tu sistema use todavía esa aberrante `union` para el `struct in_addr`, sigue siendo posible referirse a la dirección `IP` de 4 *bytes* de la misma manera en que yo lo hice antes (debido a algunos `#define`S).

3.1. ¡Convierte a valores nativos!

Todo lo anterior nos lleva a lo que se trata en esta sección. Hemos hablado mucho acerca de la conversión entre la Ordenación de máquina y la Ordenación de la red: ¡Ahora es el momento para la acción!

Muy bien. Existen dos tipos sobre los cuales podemos aplicar la conversión: `short` (dos *bytes*) y `long` (cuatro *bytes*). Las funciones de conversión también funcionan con las respectivas versiones `unsigned` de `short` y `long`. Imagina que quieres convertir un `short` desde la Ordenación de máquina [*Host Byte Order*] a la Ordenación de la red [*Network byte order*]. Empieza con una "h" de "*host*", síguela con "to" (a, hacia,...), luego una "n" de "*network* " y finalmente una "s" de "*short*": h-to-n-s, es decir, `htons()` (se lee: "*Host to Network Short*" -" `short` de máquina a `short` de la red")

Casi resulta demasiado sencillo...

Puedes usar cualquier combinación de "n", "h", "s" y "l" (de `long`), sin contar las absurdas. Por ejemplo, NO hay ninguna función `stolh()` ("*Short to Long Host*" -- `short` de máquina a `long` de máquina). Por lo menos no la hay en lo que a nosotros nos concierne. Sin embargo sí que existen:

- `htons()` -- "*Host to Network Short* " (`short` de máquina a `short` de la red)
- `htonl()` -- "*Host to Network Long*" (`long` de la máquina a `long` de la red)
- `ntohs()` -- "*Network to Host Short* " (`short` de la red a `short` de máquina)

Ahora, puedes creer que le estás cogiendo el truco a esto. Podrías pensar, "¿Qué pasa si tengo que cambiar la Ordenación de *bytes* de un `char` ?", entonces podrías pensar, "Bueno, en realidad no importa". También podrías pensar que, puesto que tu máquina 68000 ya sigue la Ordenación de *bytes* de la red, no necesitas llamar a `htonl()` sobre tus direcciones `IP`. Tendrías razón, PERO si intentas portar tu código a una máquina que siga la ordenación contraria tu programa fallará. ¡Sé portable! ¡Este es un mundo Unix! (Tanto como a Bill Gates le gustaría que no lo fuera). Recuerda: dispón tus *bytes* según la Ordenación de *bytes* de la red antes de ponerlos en la red.

Una cuestión final: ¿por qué *`sin_addr`* y *`sin_port`* necesitan seguir la Ordenación de *bytes* de la red, pero *`sin_family`* no, estando todos en la misma estructura `struct sockaddr_in`? La razón es que *`sin_addr`* y *`sin_port`* se encapsulan en un paquete en los niveles `IP` y `UDP`, respectivamente. Por eso, deben seguir la Ordenación de *bytes* de la red. Por contra, el núcleo solamente utiliza el campo *`sin_family`* para determinar qué tipo de dirección contiene la estructura, así que debe seguir la Ordenación de *bytes* de máquina. Además, como *`sin_family`* *no* se envía a través de la red, puede preservar la Ordenación de máquina.

3.2. Direcciones IP y como tratarlas

Afortunadamente para ti, hay un montón de funciones que te permiten manejar direcciones `IP`. No hay necesidad de complicarse usando el operador << sobre un `long`, ni cosas así.

Para empezar, supón que tienes una estructura `struct sockaddr_in ina` , y que quieres guardar en ella la dirección `IP` " `10.12.110.57` ". La función que necesitas usar, `inet_addr()`, convierte una dirección `IP` dada en la notación de cifras y puntos en un `unsigned long` . La asignación se puede hacer así:

```
    ina.sin_addr.s_addr = inet_addr("10.12.110.57");
```

Fíjate en que `inet_addr()` ya devuelve la dirección según la Ordenación de *bytes* de la red--no necesitas llamar a `htonl()`. ¡Magnífico!

Sin embargo, el fragmento de código de arriba no es demasiado robusto porque no se hace ninguna comprobación de errores. `inet_addr()` devuelve el valor -1 en caso de error. ¿Recuerdas los números binarios? ¡Resulta que (`unsigned`) -1 se corresponde con la dirección `IP` 255.255.255.255! La dirección de difusión. Malo. Recuerda comprobar adecuadamente las condiciones de error.

La verdad es que hay una interfaz aún más limpia que puedes usar en lugar de `inet_addr()`: se llama `inet_aton()` ("aton" significa " *ascii to network*"):

```
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>
    int inet_aton(const char *cp, struct in_addr *inp);
```

Y a continuación un ejemplo de cómo se usa al construir una estructura `struct sockaddr_in` (entenderás mejor el ejemplo cuando llegues a las secciones sobre `bind()` y `connect()` .)

```
    struct sockaddr_in my_addr;
    my_addr.sin_family = AF_INET;         // Ordenación de máquina
    my_addr.sin_port = htons(MYPORT);     // short, Ordenación de la red
    inet_aton("10.12.110.57", &(my_addr.sin_addr));
    memset(&(my_addr.sin_zero), '\0', 8); // Poner a cero el resto de la estructura
```

`inet_aton()`, *en contra de lo que hacen prácticamente todas las otras funciones de sockets*, devuelve un valor distinto de cero si tiene éxito, y cero cuando falla. Y la dirección se devuelve en *`inp`* .

Desgraciadamente, no todas las plataformas implementan `inet_aton()` así que, aunque su uso se recomienda, en esta guía usaré `inet_addr()` que, aunque más antigua, está más extendida.

Muy bien, ahora ya puedes convertir direcciones `IP` en formato carácter a su correspondiente representación binaria. ¿Qué hay del camino inverso? Qué pasa si tienes una estructura `struct in_addr` y quieres imprimirla en la notación de cifras y puntos. En ese caso necesitarás usar la función `inet_ntoa()` ("ntoa" significa "*network to ascii* ") según se muestra a continuación:

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

Eso imprimirá la dirección `IP`. Fíjate en que `inet_ntoa()` toma un `struct in_addr` como argumento, y no un `long`. Date cuenta también de que devuelve un puntero a `char`. Éste apunta a una zona estática de memoria dentro de `inet_ntoa()`, así que cada vez que llames a `inet_nota()` se perderá la última dirección `IP` que pediste. Por ejemplo:

```
    char *a1, *a2;
    .
    .
    a1 = inet_ntoa(ina1.sin_addr);  // esta es 192.168.4.14
    a2 = inet_ntoa(ina2.sin_addr);  // esta es 10.12.110.57
    printf("address 1: %s\n",a1);
    printf("address 2: %s\n",a2);
```

imprimirá:

```
    address 1: 10.12.110.57
    address 2: 10.12.110.57
```

Si necesitas conservar la dirección, usa `strcpy()` para copiarla a tu propia variable.

Eso es todo sobre este asunto por ahora. Más adelante, aprenderás a convertir una cadena como "whitehouse.gov" en su correspondiente dirección `IP` (Consulta DNS , más abajo.)

4. Llamadas al sistema

Esta sección está dedicada a las llamadas al sistema que te permiten acceder a la funcionalidad de red de una máquina Unix. Cuando llamas a una de estas funciones, el núcleo toma el control y realiza todo el trabajo por ti automágicamente.

Lo que más confunde a la gente es el orden en que deben realizarse las llamadas. En esto las páginas **man** son completamente inútiles, como probablemente ya has descubierto. Como ayuda en una situación tan desagradable, he tratado de disponer las llamadas al sistema en las siguientes secciones en el orden (más o menos) *exacto* en que debes llamarlas en tus programas.

Esto, unido con unos pocos fragmentos de código por aquí y por allí, un poco de leche con galletas (que me temo que tendrás que aportar tú) y un poco de convencimiento y valor, ¡y estarás enviando datos a la red como un poseso!

4.1. `socket()` --¡Consigue el descriptor de fichero!

Supongo que ya no puedo postponerlo más--Tengo que hablarte de la llamada al sistema `socket()`. Ahí van los detalles:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Pero, ¿qué son esos argumentos? En primer lugar, *`domain`* tiene que ser "`AF_INET`", igual que en la estructura `struct sochaddr_in` de arriba. Además, el argumento *`type`* le dice al núcleo qué tipo de *socket* es este: `SOCK_STREAM` o `SOCK_DGRAM` . Por último, basta con asignar a *`protocol`* un "0" para que `socket()` elija el protocolo correcto en función del tipo (*`type`*). (Notas: Hay muchos más dominios (`domain`) de los que yo he listado. También hay muchos más tipos (`type`) de los que yo he listado. Consulta la página `man` de `select()` . Además, hay una manera mejor de obtener el protocolo (`protocol`). Consulta la página `man` de `getprotobyname()`.)

`socket()` tan sólo te devuelve un descriptor de *socket* que puedes usar en posteriores llamadas al sistema, o -1 en caso de error. En ese caso, a la variable global `errno` se el asigna un valor de error (consulta la página `man` de `perror()` .)

En alguna documentación se menciona un valor místico: "`PF_INET` ". Se trata de una extraña y etérea bestia que rara vez se deja ver en la naturaleza, pero que voy a clarificar un poco aquí. Hace mucho tiempo se pensaba que tal vez una familia de direcciones (es lo que significa " `AF` " en "`AF_INET`": *Address Family* - familia de direcciones) diversos protocolos que serían referenciados por su familia de protocolos (que es lo que significa "`PF`" en "`PF_INET`": *Protocol Family* - familia de protocolos). Eso nunca ocurrió. De acuerdo, lo correcto entonces es usar `AF_INET` en la estructura `struct sockaddr_in` y `PF_INET` en la llamada a `socket()`. Pero en la práctica puedes usar `AF_INET` en todas partes. Y puesto que eso es lo que W. Richard Stevens hace en su libro, eso es lo que yo voy a hacer aquí.

Bien, bien, bien, pero ¿para qué sirve este *socket*? La respuesta es que, en sí mismo, no sirve

para gran cosa y necesitas seguir leyendo y hacer más llamadas al sistema para que esto tenga algún sentido.

4.2. `bind()`--¿En qué puerto estoy?

Una vez que tienes tu *socket*, tendrías que asociarlo con algún puerto de tú máquina local. (Esto es lo que comúnmente se hace si vas a escuchar [`listen()`] a la espera de conexiones entrantes sobre un puerto específico--Esto es lo que hacen cuando te dicen que hagas a telnet a x.y.z puerto 6969). El núcleo usa el número de puerto para asociar los paquetes entrantes con un descriptor de *socket* de un cierto proceso. Si solamente vas a hacer un `connect()` esto es innecesario. De todas formas léelo, sólo por saberlo.

Esta es la sinopsis de la llamada al sistema `bind()` :

```
    #include <sys/types.h>
    #include <sys/socket.h>
    int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

`sockfd` es el descriptor de fichero de *socket* que devolvió `socket()`. *`my_addr`* es un puntero a una estructura `struct sockaddr` que contiene información acerca de tu propia dirección, a saber, puerto y dirección `IP`. *`addrlen`* se puede asignar a `sizeof(struct sockaddr)`.

Bueno. Esto es demasiado para absorberlo de una sola vez. Veamos un ejemplo:

```
    #include <string.h>
    #include <sys/types.h>
    #include <sys/socket.h>
    #include <netinet/in.h>
    #define MYPORT 3490
    main()
    {
        int sockfd;
        struct sockaddr_in my_addr;
        sockfd = socket(AF_INET, SOCK_STREAM, 0); // ¡Comprueba que no hay errores!
        my_addr.sin_family = AF_INET;         // Ordenación de máquina
        my_addr.sin_port = htons(MYPORT);     // short, Ordenación de la red
        my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");
        memset(&(my_addr.sin_zero), '\0', 8); // Poner a cero el resto de la
estructura
        // no olvides comprobar los errores de bind():
        bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
        .
        .
        .
```

Hay unas pocas cosas a destacar aquí: *`my_addr.sin_port`* sigue la Ordenación de *bytes* de la red. También la sigue *`my_addr.sin_addr.s_addr`* . Otra cosa para estar atentos es que los ficheros de cabecera podrían ser distintos en sistemas distintos. Para asegurarte, revisa tus páginas locales de **man**.

Para finalizar con `bind()`, tendría que mencionar que parte del proceso de obtención de tu propia dirección `IP` y/o número de puerto puede automatizarse:

```
        my_addr.sin_port = 0; // Elige aleatoriamente un puerto que no esté en uso
        my_addr.sin_addr.s_addr = INADDR_ANY;  // usa mi dirección IP
```

puerto por ti. Del mismo modo, al asignarle a `my_addr.sin_addr.s_addr` el valor `INADDR_ANY` , le estás diciendo que escoja automáticamente la dirección `IP` de la máquina sobre la que está ejecutando el proceso.

Si te estás percatando de los detalles tal vez hayas visto que no puse `INADDR_ANY` en la Ordenación de *bytes* de la red. Qué travieso soy. Sin embargo tengo información privilegiada: ¡en realidad `INADDR_ANY` es cero! Cero es siempre cero, aunque reordenes los *bytes*. Sin embargo, los puristas pueden aducir que podría existir una dimensión paralela donde `INADD_ANY` fuera, por ejemplo, 12, y que mi código no funcionaría allí. Está bien:

```
        my_addr.sin_port = htons(0); // Elige aleatoriamente un puerto que no esté
en uso
        my_addr.sin_addr.s_addr = htonl(INADDR_ANY);  // usa mi dirección IP
```

Ahora somos tan portables que probablemente no lo creerías. Solamente quería señalarlo, porque en la mayoría del código que te encuentres no habrán pasado `INADDR_ANY` a través de `htonl()`.

`bind()` también devuelve `-1` en caso de error y el asigna a *`errno`* el valor del error.

Otra cosa a controlar cuando llames a `bind()`: No uses números de puerto demasiado pequeños. Todos los puertos por debajo del 1024 están RESERVADOS (a menos que seas el superusuario). Puedes usar cualquier número de puerto por encima de ese, hasta el 65535 (siempre y cuando no lo esté usando ya otro programa).

En ocasiones, puedes encontrarte con que, al volver a ejecutar `bind()` en un servidor obtienes el error "*Address already in use*" (La dirección ya se está usando). ¿Qué significa? Bueno, una parte de un *socket* que estuvo conectado, está todavía colgando en el núcleo y está bloqueando el puerto. Puedes esperar a que se libere (alrededor de un minuto) o añadirle código a tu programa permitiendo reutilizar el puerto, de este modo:

```
    int yes=1;
        //char yes='1'; // La gente de Solaris usa esto
    // Olvidémonos del error "Address already in use" [La dirección ya se está
usando]
    if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1) {
        perror("setsockopt");
        exit(1);
    }
```

Una nota final más acerca de `bind()`: en ocasiones no tendrás que usar esta función en absoluto. Si vas a conectar (`connect()`) a una máquina remota y no te importa cuál sea tu puerto local (como es el caso de **telnet**, donde sólo te importa el puerto remoto), puedes llamar sólo a `connect()`, que se encargará de comprobar si el puerto está o no asociado y, si es el caso, lo asociará con un puerto local que no se esté usando.

4.3. `connect()` --¡eh, tú!

Supongamos por un momento que eres una aplicación telnet. Tu usuario te ordena (como en la película *TRON*) que obtengas un descriptor de fichero de *socket*. Tú obedeces y ejecutas `socket()`. Ahora el usuario te pide que conectes al puerto "23" de "10.12.110.57" (el puerto estándar de telnet). ¿Qué haces ahora?

Afortunadamente para ti, programa, estás leyendo ahora la sección `connect()` -- o cómo conectar con una máquina remota. ¡Así que devóralo! ¡No hay tiempo que perder!

La llamada `connect()` es como sigue:

```
    #include <sys/types.h>
    #include <sys/socket.h>
    int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

`sockfd` es nuestro famoso descriptor de fichero de *socket*, tal y como nos lo devolvió nuestra llamada a `socket()`, *`serv_addr`* es una estructura `struct sockaddr` que contiene el puerto y la dirección `IP` de destino, y a *`addrlen`* le podemos asignar el valor `sizeof(struct sockaddr)`.

¿No está esto empezando a tener más sentido? Veamos un ejemplo:

```
    #include <string.h>
    #include <sys/types.h>
    #include <sys/socket.h>
    #include <netinet/in.h>
    #define DEST_IP   "10.12.110.57"
    #define DEST_PORT 23
    main()
    {
        int sockfd;
        struct sockaddr_in dest_addr;   // Guardará la dirección de destino
        sockfd = socket(AF_INET, SOCK_STREAM, 0); // ¡Comprueba errores!
        dest_addr.sin_family = AF_INET;          // Ordenación de máquina
        dest_addr.sin_port = htons(DEST_PORT);   // short, Ordenación de la red
        dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
        memset(&(dest_addr.sin_zero), '\0', 8);  // Poner a cero el resto de la
estructura
        // no olvides comprobar los errores de connect()!
        connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
        .
        .
        .
```

Como siempre, asegúrate de comprobar el valor de retorno de `connect()` --devolverá -1 en caso de error y establecerá la variable *`errno`* .

Además fíjate en que no hemos llamado a `bind()`. Básicamente nos da igual nuestro número local de puerto; Sólo nos importa a dónde nos dirigimos (el puerto remoto). El núcleo elegirá un puerto local por nosotros, y el sitio al que nos conectamos será informado automáticamente. No hay de qué preocuparse.

4.4. `listen()` --Por favor, que alguien me llame

Muy bien, es momento para un cambio de ritmo. ¿Qué pasa si no te quieres conectar a una máquina remota? Supongamos, sólo por probar, que quieres esperar a que te lleguen conexiones de entrada y gestionarlas de alguna manera. El proceso consta de dos pasos: en primer lugar escuchas (`listen()`) y después aceptas (`accept()`) (mira más abajo)

La llamada `listen()` es bastante sencilla, pero requiere una breve explicación:

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

sockfd es el habitual descriptor de fichero de *socket* que nos fue devuelto por la llamada al sistema `socket()`. *backlog* es el número de conexiones permitidas en la cola de entrada. ¿Qué significa eso? Bueno, las conexiones entrantes van a esperar en esta cola hasta que tú las aceptes (`accept()`) (mira más abajo) y éste es el límite de conexiones que puede haber en cola. La mayoría de los sistemas limitan automáticamente esta cifra a 20; probablemente puedes apañarte asignando el valor 5 ó 10.

Como siempre, `listen()` devuelve -1 en caso de error y establece *errno*.

Bueno, como probablemente imaginas, necesitamos llamar a `bind()` antes de poder llamar a `listen()`, de lo contrario el núcleo nos tendrá esperando en un puerto aleatorio. Así que si vas a estar escuchando a la espera de conexiones entrantes, la secuencia de llamadas que te corresponde hacer es:

```
socket();
bind();
listen();
/* accept() va aquí */
```

Lo doy por válido como código de ejemplo porque es bastante claro. (El código de la sección `accept()`, a continuación, es más completo). El auténtico truco de todo esto está en la llamada a `accept()`.

4.5. `accept()` --"Gracias por llamar al puerto 3490."

Prepárate--la llamada al sistema `accept()` es un tanto extraña. Lo que va a suceder es lo siguiente: alguien muy, muy lejano intentará conectar (`connect()`) con tu máquina en un puerto en el que tú estás escuchando (`listen()`). Su conexión pasará a cola, esperando a ser aceptada (`accept()`). Cuando llamas a `accept()` le estás diciendo que quieres obtener una conexión pendiente. La llamada al sistema, a su vez, te devolverá un descriptor de fichero de *socket* completamente nuevo para que lo uses en esta nueva conexión. Exacto. De repente, y por el precio de uno, tienes dos descriptores de fichero de *socket*. El original está todavía escuchando en tu puerto, y el de nueva creación está listo para enviar (`send()`) y recibir (`recv()`). ¡Ya casi estamos ahí!

La llamada es como sigue:

```
#include <sys/socket.h>
int accept(int sockfd, void *addr, int *addrlen);
```

sockfd es el descriptor de fichero donde estás escuchando (`listen()`). Es muy fácil. *addr* es normalmente un puntero a una estructura `struct sockaddr_in` local. Ahí es donde se guardará la información de la conexión entrante (y con ella puedes averiguar que máquina te está llamando, y desde qué puerto). *addrlen* es un puntero a una variable local `int` a la que deberías asignar el valor de `sizeof(struct sockaddr_in)`. `accept()` pondrá dentro de *addr* un máximo de *addrlen bytes*. Si pone menos, cambiará el valor de *addrlen* para que refleje la cantidad real de *bytes* almacenados.

¿Sabes qué? `accept()` devuelve -1 en caso de error y establece la variable *errno*. Debiste suponerlo.

Como dije antes, esto es un buen cacho para digerirlo de una sola vez, así que ahí va un fragmento de código de ejemplo para que te lo estudies:

```
    #include <string.h>
    #include <sys/types.h>
    #include <sys/socket.h>
    #include <netinet/in.h>
    #define MYPORT 3490    // Puerto al que conectarán los usuarios
    #define BACKLOG 10     // Cuántas conexiones vamos a mantener en cola
    main()
    {
        int sockfd, new_fd;  // se escucha sobre sock_fd, Nuevas conexiones sobre
new_fd
        struct sockaddr_in my_addr;    // Información sobre mi dirección
        struct sockaddr_in their_addr; // Información sobre la dirección remota
        int sin_size;
        sockfd = socket(AF_INET, SOCK_STREAM, 0); // ¡Comprobar errores!
        my_addr.sin_family = AF_INET;         // Ordenación de máquina
        my_addr.sin_port = htons(MYPORT);     // short, Ordenación de la red
        my_addr.sin_addr.s_addr = INADDR_ANY; // Rellenar con mi dirección IP
        memset(&(my_addr.sin_zero), '\0', 8); // Poner a cero el resto de la
estructura
        // no olvides comprobar errores para estas llamadas:
        bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
        listen(sockfd, BACKLOG);
        sin_size = sizeof(struct sockaddr_in);
        new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
        .
        .
        .
```

Como te decía, usaremos el descriptor de fichero de *socket* *`new_fd`* en las llamadas al sistema `send()` y `recv()` . Si solamente esperas recibir una conexión puedes cerrar (`close()`) el descriptor *`sockfd`* que está escuchando para evitar que lleguen nuevas conexiones de entrada al mismo puerto.

4.6. `send()` y `recv()`--¡Háblame, *baby*!

Estas dos funciones sirven para comunicarse a través de *sockets* de flujo o *sockets* de datagramas conectados. Si quieres usar *sockets* de datagramas desconectados normales tienes que leer la sección `sendto()` y `recvfrom()` , a continuación.

La llamada al sistema `send()`:

```
    int send(int sockfd, const void *msg, int len, int flags);
```

`sockfd` es el descriptor de *socket* al que quieres enviar datos (bien sea el devuelto por `socket()` , bien el devuelto por `accept()`.) *`msg`* es un puntero a los datos que quieres enviar, y *`len`* es la longitud de esos datos en *bytes*. Asigna a *`flags`* el valor 0 (Revisa la página `man` de `send()` para más información relativa a los *flags*).

Lo siguiente podría servir como código de ejemplo:

```
    char *msg = "Beej was here!";
    int len, bytes_sent;
    .
    .
    len = strlen(msg);
    bytes_sent = send(sockfd, msg, len, 0);
```

```
    .
    .
    .
```

`send()` devuelve el múmero de bytes que se enviaron en realidad--*¡y podrían ser menos de los que tú pediste que se enviaran!* Por ejemplo hay veces que solicitas enviar todo un montón de datos y el sistema sencillamente no puede manejarlos todos. Procesará tantos datos como pueda y confiará en que tú envíes el resto después. Recuerda, si el valor devuelto por `send()` no coincide con el valor de *`len`* , depende de ti enviar el resto de la cadena. La buena noticia es esta: si el paquete es pequeño (menos de 1K o así) *probablemente* se las apañará para enviarlo todo de una tacada. Como siempre, en caso de error devuelve -1 y se establece la variable *`errno`*.

La llamada al sistema `recv()` es similar en muchos aspectos:

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

`sockfd` es el descriptor del fichero del que se va a leer, *`buff`* es el *buffer* donde se va a depositar la información leida, *`len`* es la longitud máxima del *buffer*, y *`flags`*, como antes, puede asignarse a 0 (Revisa la página `man` de `recv()` para información sobre *flags*)

`recv()` devuelve el número de *bytes* que se leyeron en realidad, o -1 en caso de error (y entonces establece *`errno`* apropiadamente).

¡Espera! `recv()` puede devolver 0. Esto sólo puede significar una cosa: ¡la máquina remota ha cerrado su conexión contigo! Un valor de retorno igual a cero es la forma que tiene `recv()` de comunicarte que éso ha sucedido.

Bueno, fue sencillo, ¿no? ¡Ahora puedes pasar datos en los dos sentidos a través de un *socket* de flujo! ¡Estupendo! ¡Eres un programador Unix de redes!

4.7. `sendto()` y `recvfrom()`--Háblame al estilo DGRAM

"Todo esto está muy bien", te escucho decir, "pero a dónde me lleva esto si lo que quiero es usar *sockets* de datagramas desconectados". No problemo, amigo. Estamos en ello.

Puesto que los *sockets* de datagramas no están conectados a una máquina remota, ¿adivina qué información necesitamos aportar antes de poder enviar un paquete? ¡Exacto! ¡La dirección de destino! Aquí está un avance:

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
           const struct sockaddr *to, int tolen);
```

Como ves, esta llamada es básicamente la misma que `send()` añadiendo dos items más de información. *`to`* es un puntero a una estructura `struct sockaddr` (probablemente usarás una estructura `struct sockaddr_in` y la forzarás a `struct sockaddr` en el último momento) que contiene la dirección `IP` y el puerto de destino. Al argumento *`tolen`* asígnale el valor `sizeof(struct sockaddr)`.

Lo mismo que `send()`, `sendto()` devuelve el número de *bytes* que realmente se enviaron

(que, igual que antes, podrían ser menos de los que tú pediste enviar) o -1 en caso de error (con *errno* establecido convenientemente).

La misma semejanza presentan `recv()` y `recvfrom()`. La sinopsis de `recvfrom()` es:

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

De nuevo, es igual que `recv()` pero con dos argumentos más. *from* es un puntero a una estructura `struct sockaddr` local que será rellenada con la dirección IP y el puerto de la máquina de origen. *fromlen* es un puntero a un `int` local que tiene que inicializarse a `sizeof(struct sockaddr)`. Cuando la función finalice, *fromlen* contendrá la longitud de la dirección almacenada en *from*.

`recvfrom()` devuelve el número de *bytes* recibidos, o -1 en caso de error (con *errno* establecido convenientemente).

Recuerda, si conectas (`connect()`) un *socket* de datagramas simplemente tienes que usar `send()` y `recv()` para todas tus transaciones. El *socket* mismo es aún un *socket* de datagramas y los paquetes seguirán usando UDP, pero el interfaz de *sockets* añadirá automáticamente por ti la información de origen y destino.

4.8. `close()` y `shutdown()`--¡Fuera de mi vista!

Bueno, has estado enviando (`send()`) y recibiendo (`recv()`) datos todo el día y ahora has terminado. Estás listo para cerrar la conexión de tu descriptor de *socket*. Esto es sencillo. Sólo hay que usar normalmente la función Unix `close()` que cierra descriptores de fichero:

```
close(sockfd);
```

Esto impedirá más lecturas y escrituras al *socket*. Cualquiera que intente leer o escribir sobre el *socket* en el extremo remoto recibirá un error.

Sólo en el caso que quieras un poco más de control sobre cómo se cierra el *socket* puedes usar la función `shutdown()`. Te permite cortar la comunicación en un cierto sentido, o en los dos (tal y como lo hace `close()`). Sinopsis:

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

sockfd es el descriptor de *socket* que quieres desconectar, y *how* es uno de los siguientes valores:
is the socket file descriptor you want to shutdown, and *how* is one of the following:

- `0` -- No se permite recibir más datos
- `1` -- No se permite enviar más datos
- `2` -- No se permite enviar ni recibir más datos (lo mismo que `close()`)

`shutdown()` devuelve 0 si tiene éxito, y -1 en caso de error (con *`errno`* establecido adecuadamente)

Si usas `shutdown()` en un *socket* de datagramas sin conexión, simplemente inhabilitará el *socket* para posteriores llamadas a `send()` y `recv()` (recuerda que puedes usarlas si llamaste a `connect()` sobre tu *socket* de datagramas).

Es importante destacar que `shutdown()` no cierra realmente el descriptor de fichero --sólo cambia sus condiciones de uso. Para liberar un descriptor de *socket* necesitas usar `close()`.

Nada más.

4.9. `getpeername()` --¿Quién eres tú?

Esta función es fácil.

Tan fácil, que estuve a punto de no otorgarle una sección propia. En cualquier caso, aquí está.

La función `getpeername()` te dirá quién está al otro lado de un *socket* de flujo conectado. La sinopsis:

```
    #include <sys/socket.h>
    int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

`sockfd` es el descriptor del *socket* de flujo conectado, *`addr`* es un puntero a una estructura `struct sockaddr` (o `struct sockaddr_in`) que guardará la información acerca del otro lado de la conexión, y *`addrlen`* es un puntero a un `int`, que deberías inicializar a `sizeof(struct sockaddr)`.

La función devuelve -1 en caso de error y establece *`errno`* apropiadamente.

Una vez que tienes su dirección, puedes usar `inet_ntoa()` or `gethostbyaddr()` para imprimir u obtener más información. No, no puedes saber su nombre de login. (Vale, vale, si el otro ordenador está ejecutando un demonio `ident`, sí es posible. Esto, sin embargo, va más allá del alcance de este documento. Revisa el RFC-1413 para más información.)

4.10. `gethostname()` --¿Quién soy yo?

La función `gethostname()` es incluso más fácil que `getpeername()`. Devuelve el nombre del ordenador sobre el que tu programa se está ejecutando. El nombre puede usarse entonces con `gethostbyname()`, como se indica en la siguiente sección, para determinar la dirección `IP` de tu máquina local.

¿Qué puede haber más divertido? Se me ocurren un par de cosas, pero no tienen nada que ver con la programación de *sockets*. En todo caso, ahí van los detalles:

```
    #include <unistd.h>
    int gethostname(char *hostname, size_t size);
```

Los argumentos son sencillos: *`hostname`* es un puntero a una cadena de carácteres donde se

almacenará el nombre de la máquina cuando la función retorne, y *size* es la longitud en *bytes* de esa cadena de caracteres.

La función devuelve 0 si se completa sin errores, y -1 en caso contrario, estableciendo *errno* de la forma habitual.

4.11. DNS--Tú dices "whitehouse.gov", yo digo "198.137.240.92"

Por si acaso no sabes qué es DNS, te diré que significa "Servicio de Nombres de Dominio" [*Domain Name Service*]. En una palabra, tú le dices cuál es la dirección de un sitio en forma humanamente legible y él te devuelve la dirección IP (para que puedas usarla con bind(), connect(), sendto(), o donde sea que la necesites). Así, cuando alguien escribe:

```
$ telnet whitehouse.gov
```

telnet puede averiguar que necesita conectarse (connect()) a "198.137.240.92"

Pero, ¿cómo funciona? Tú vas a usar la función gethostbyname() :

```
#include <netdb.h>

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

Como puedes ver, devuelve un puntero a una estructura struct hostent , cuyo desglose es el siguiente:

```
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]
```

Y estas son las descripciones de los campos de la estructura struct hostent:

- *h_name* -- Nombre oficial de la máquina.
- *h_aliases* -- Un vector [*array*] terminado en NULL de nombres alternativos de máquina.
- *h_addrtype* -- Tipo de la dirección que se devuelve; usualmente *AF_INET*.
- *h_length* -- La longitud de la dirección en *bytes*.
- *h_addr_list* -- Un vector terminado en cero de direcciones de red de la máquina. Las direcciones siguen la Ordenación de *bytes* de la red.
- *h_addr* -- La primera dirección de *h_addr_list*.

`gethostbyname()` devuelve un puntero a la estructura `struct hostent` que se ha llenado, o NULL en caso de error. (Sin embargo no se establece *`errno`*, sino *`h_errno`*. Consulta `herror()` más adelante).

Pero, ¿cómo se usa? A veces (nos damos cuenta leyendo manuales de informática), no es suficiente con vomitarle la información al lector. En realidad, esta función es más fácil de usar de lo que parece.

Aquí hay un programa de ejemplo :

```
/*
** getip.c -- ejemplo de búsqueda DNS
*/
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main(int argc, char *argv[])
{
    struct hostent *h;
    if (argc != 2) {  // Comprobación de errores en la línea de comandos
        fprintf(stderr,"usage: getip address\n");
        exit(1);
    }
    if ((h=gethostbyname(argv[1])) == NULL) {  // Obtener información del host
        herror("gethostbyname");
        exit(1);
    }
    printf("Host name  : %s\n", h->h_name);
    printf("IP Address : %s\n", inet_ntoa(*((struct in_addr *)h->h_addr)));

   return 0;
}
```

Con `gethostbyname()`, no puedes usar `perror()` para imprimir mensajes de error (puesto que a *`errno`* no se le asigna valor alguno). En su lugar debes llamar a `herror()`.

Está bastante claro. Simplemente pasas la cadena que tiene el nombre de la máquina ("whitehouse.gov") a `gethostbyname()`, y recuperas la información que te han devuelto en la estructura `struct hostent`.

La única cosa rara que podrías encontrarte sería en el momento de imprimir la dirección `IP`. *`h->h_addr`* es un `char *` mientras que `inet_ntoa()` necesita una estructura `struct in_addr`. Por eso, yo suelo forzar *`h->h_addr`* a `struct in_addr*`, y desreferencio para obtener los datos.

5. Modelo Cliente-Servidor

Amigo, este es un mundo cliente-servidor. Casi cualquier cosa en la red tiene que ver con procesos clientes que dialogan con procesos servidores y viceversa. Consideremos **telnet**, por ejemplo. Cuando conectas al puerto 23 de una máquina remota mediante telnet (el cliente) un programa de aquella máquina (llamado **telnetd**, el servidor) despierta a la vida. Gestiona la conexión telnet entrante, te presenta una pantalla de *login* , etc.

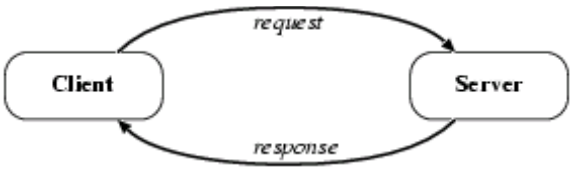

Figura 2. Interacción Cliente-Servidor.

El intercambio de información entre cliente y servidor se resume en la Figura 2 .

Observa que el par cliente-servidor pueden hablar `SOCK_STREAM` , `SOCK_DGRAM` o cualquier otra cosa (siempre y cuando los dos hablen lo mismo). Algunos buenos ejemplos de parejas cliente-servidor son **telnet/telnetd** , **ftp/ftpd**, o **bootp/bootpd**. Cada vez que usas **ftp**, hay un programa remoto, **ftpd**, que te sirve.

Con frecuencia, solamente habra un servidor en una máquina determinada, que atenderá a múltiples clientes usando `fork()`. El funcionamiento básico es: el servidor espera una conexión, la acepta (`accept()`) y usa `fork()` para obtener un proceso hijo que la atienda. Eso es lo que hace nuestro servidor de ejemplo en la siguiente sección.

5.1. Un servidor sencillo

Todo lo que hace este servidor es enviar la cadena "`Hello, World!\n`" sobre una conexión de flujo. Todo lo que necesitas para probar este servidor es ejecutarlo en una ventana y atacarlo con telnet desde otra con:

```
    $ telnet remotehostname 3490
```

donde `remotehostname` es el nombre de la máquina donde estas ejecutando.

El código servidor : (Nota: una barra invertida al final de una línea indica que esa línea se continúa en la siguiente.)

```
    /*
    ** server.c -- Ejemplo de servidor de sockets de flujo
    */
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <errno.h>
    #include <string.h>
    #include <sys/types.h>
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>
    #include <sys/wait.h>
    #include <signal.h>
    #define MYPORT 3490    // Puerto al que conectarán los usuarios
```

```
    #define BACKLOG 10     // Cuántas conexiones pendientes se mantienen en cola
    void sigchld_handler(int s)
    {
        while(wait(NULL) > 0);
    }
    int main(void)
    {
        int sockfd, new_fd;  // Escuchar sobre sock_fd, nuevas conexiones sobre
new_fd
        struct sockaddr_in my_addr;    // información sobre mi dirección
        struct sockaddr_in their_addr; // información sobre la dirección del
cliente
        int sin_size;
        struct sigaction sa;
        int yes=1;
        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
            perror("socket");
            exit(1);
        }
        if (setsockopt(sockfd,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1) {
            perror("setsockopt");
            exit(1);
        }

        my_addr.sin_family = AF_INET;         // Ordenación de bytes de la máquina
        my_addr.sin_port = htons(MYPORT);     // short, Ordenación de bytes de la
red
        my_addr.sin_addr.s_addr = INADDR_ANY; // Rellenar con mi dirección IP
        memset(&(my_addr.sin_zero), '\0', 8); // Poner a cero el resto de la
estructura
        if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr))
                                                                    == -1) {
            perror("bind");
            exit(1);
        }
        if (listen(sockfd, BACKLOG) == -1) {
            perror("listen");
            exit(1);
        }
        sa.sa_handler = sigchld_handler; // Eliminar procesos muertos
        sigemptyset(&sa.sa_mask);
        sa.sa_flags = SA_RESTART;
        if (sigaction(SIGCHLD, &sa, NULL) == -1) {
            perror("sigaction");
            exit(1);
        }
        while(1) {  // main accept() loop
            sin_size = sizeof(struct sockaddr_in);
            if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr,
                                                        &sin_size)) == -1) {
                perror("accept");
                continue;
            }
            printf("server: got connection from %s\n",
                                            inet_ntoa(their_addr.sin_addr));
            if (!fork()) { // Este es el proceso hijo
                close(sockfd); // El hijo no necesita este descriptor
                if (send(new_fd, "Hello, world!\n", 14, 0) == -1)
                    perror("send");
                close(new_fd);
                exit(0);
            }
            close(new_fd);  // El proceso padre no lo necesita
        }
        return 0;
    }
```

Por si sientes curiosidad, tengo todo el código en una gran función `main()` porque me parece más claro. Puedes partirlo en funciones más pequeñas si eso te hace sentir mejor.

(Además, esto del `sigaction()` podría ser nuevo para ti--es normal. El código que hay ahí se

encarga de limpiar los procesos zombis que pueden aparecer cuando los procesos hijos finalizan. Si creas muchos procesos zombis y no los eliminas, tu administrador de sistema se mosqueará)

Puedes interactuar con este servidor usando el cliente de la siguiente sección.

5.2. Un cliente sencillo

Este tío es incluso más sencillo que el servidor. Todo lo que hace es conectar al puerto 3490 de la máquina que indicas en la línea de comandos y obtiene la cadena que el servidor envía.

El código cliente :

```
    /*
    ** client.c -- Ejemplo de cliente de sockets de flujo
    */
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <errno.h>
    #include <string.h>
    #include <netdb.h>
    #include <sys/types.h>
    #include <netinet/in.h>
    #include <sys/socket.h>
    #define PORT 3490 // puerto al que vamos a conectar
    #define MAXDATASIZE 100 // máximo número de bytes que se pueden leer de una vez
    int main(int argc, char *argv[])
    {
        int sockfd, numbytes;
        char buf[MAXDATASIZE];
        struct hostent *he;
        struct sockaddr_in their_addr; // información de la dirección de destino
        if (argc != 2) {
            fprintf(stderr,"usage: client hostname\n");
            exit(1);
        }
        if ((he=gethostbyname(argv[1])) == NULL) {  // obtener información de
máquina
            perror("gethostbyname");
            exit(1);
        }
        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
            perror("socket");
            exit(1);
        }
        their_addr.sin_family = AF_INET;    // Ordenación de bytes de la máquina
        their_addr.sin_port = htons(PORT);  // short, Ordenación de bytes de la red
        their_addr.sin_addr = *((struct in_addr *)he->h_addr);
        memset(&(their_addr.sin_zero), 8);  // poner a cero el resto de la
estructura
        if (connect(sockfd, (struct sockaddr *)&their_addr,
                                              sizeof(struct sockaddr)) == -1) {
            perror("connect");
            exit(1);
        }
        if ((numbytes=recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {
            perror("recv");
            exit(1);
        }
        buf[numbytes] = '\0';
        printf("Received: %s",buf);
        close(sockfd);
        return 0;
    }
```

Observa que si no ejecutas el servidor antes de llamar al cliente, `connect()` devuelve "Connection refused" (Conexión rechazada). Muy útil.

5.3. Sockets de datagramas

En realidad no hay demasiado que contar aquí, así que sólo presentaré un par de programas de ejemplo: `talker.c` y `listener.c`.

listener se sienta a esperar en la máquina hasta que llega un paquete al puerto 4950. **talker** envía un paquete a ese puerto en la máquina indicada que contiene lo que el usuario haya escrito en la línea de comandos.

Este es el código fuente de `listener.c` :

```
    /*
    ** listener.c -- Ejemplo de servidor de sockets de datagramas
    */
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <errno.h>
    #include <string.h>
    #include <sys/types.h>
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>
    #define MYPORT 4950    // puerto al que conectarán los clientes
    #define MAXBUFLEN 100
    int main(void)
    {
        int sockfd;
        struct sockaddr_in my_addr;    // información sobre mi dirección
        struct sockaddr_in their_addr; // información sobre la dirección del
cliente
        int addr_len, numbytes;
        char buf[MAXBUFLEN];
        if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
            perror("socket");
            exit(1);
        }
        my_addr.sin_family = AF_INET;         // Ordenación de bytes de máquina
        my_addr.sin_port = htons(MYPORT);     // short, Ordenación de bytes de la
red
        my_addr.sin_addr.s_addr = INADDR_ANY; // rellenar con mi dirección IP
        memset(&(my_addr.sin_zero), '\0', 8); // poner a cero el resto de la
estructura
        if (bind(sockfd, (struct sockaddr *)&my_addr,
                                              sizeof(struct sockaddr)) == -1) {
            perror("bind");
            exit(1);
        }
        addr_len = sizeof(struct sockaddr);
        if ((numbytes=recvfrom(sockfd,buf, MAXBUFLEN-1, 0,
                        (struct sockaddr *)&their_addr, &addr_len)) == -1) {
            perror("recvfrom");
            exit(1);
        }
        printf("got packet from %s\n",inet_ntoa(their_addr.sin_addr));
        printf("packet is %d bytes long\n",numbytes);
        buf[numbytes] = '\0';
        printf("packet contains \"%s\"\n",buf);
        close(sockfd);
        return 0;
    }
```

Observa que en nuestra llamada a `socket()` finalmente estamos usando `SOCK_DGRAM`. Observa también que no hay necesidad de escuchar (`listen()`) o aceptar (`accept()`). ¡Esa es una de las ventajas de usar *sockets* de datagramas sin conexión!

A continuación el código fuente de `talker.c` :

```
    /*
    ** talker.c -- ejemplo de cliente de datagramas
    */
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <errno.h>
    #include <string.h>
    #include <sys/types.h>
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>
    #include <netdb.h>
    #define MYPORT 4950    // puerto donde vamos a conectarnos
    int main(int argc, char *argv[])
    {
        int sockfd;
        struct sockaddr_in their_addr; // información sobre la dirección del
servidor
        struct hostent *he;
        int numbytes;
        if (argc != 3) {
            fprintf(stderr,"usage: talker hostname message\n");
            exit(1);
        }
        if ((he=gethostbyname(argv[1])) == NULL) {  // obtener información de
máquina
            perror("gethostbyname");
            exit(1);
        }
        if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
            perror("socket");
            exit(1);
        }
        their_addr.sin_family = AF_INET;     // Ordenación de bytes de máquina
        their_addr.sin_port = htons(MYPORT); // short, Ordenación de bytes de la
red
        their_addr.sin_addr = *((struct in_addr *)he->h_addr);
        memset(&(their_addr.sin_zero), '\0', 8); // poner a cero el resto de la
estructura
        if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0,
             (struct sockaddr *)&their_addr, sizeof(struct sockaddr))) == -1) {
            perror("sendto");
            exit(1);
        }
        printf("sent %d bytes to %s\n", numbytes,
                                              inet_ntoa(their_addr.sin_addr));
        close(sockfd);
        return 0;
    }
```

¡Y esto es todo! Ejecuta **listener** en una máquina y luego llama a **talker** en otra. ¡Observa cómo se comunican! ¡Disfruta de toda la excitación de la familia nuclear entera!

Excepto un pequeño detalle que he mencionado muchas veces con anterioridad: *sockets* de datagramas con conexión. Tengo que hablar de ellos aquí, puesto que estamos en la sección de datagramas del documento. Supongamos que **talker** llama a `connect()` e indica la dirección de **listener**. A partir de ese momento, **talker** solamente puede enviar a y recibir de la dirección especificada en `connect()`. Por ese motivo no tienes que usar `sendto()` y `recvfrom()`; tienes que usar simplemente `send()` y `recv()`.

6. Técnicas moderadamente avanzadas

En realidad no son *verdaderamente* avanzadas, pero se salen de los niveles más básicos que ya hemos cubierto. De hecho, si has llegado hasta aquí, puedes considerarte conocedor de los principios básicos de la programación de redes Unix. ¡Enhorabuena!

Así que ahora entramos en el nuevo mundo de las cosas más esotéricas que querrías aprender sobre los *sockets*. ¡Ahí las tienes!

6.1. Bloqueo

Bloqueo. Has oído hablar de él--pero ¿qué carajo es? En una palabra, "bloquear" es el tecnicismo para "dormir". Probablemente te has dado cuenta de que, cuando ejecutas **listener**, más arriba, simplemente se sienta a esperar a que llegue un paquete. Lo que sucedió es que llamó a `recvfrom()` y no había datos que recibir. Por eso se dice que `recvfrom()` se bloquea (es decir, se duerme) hasta que llega algún dato.

Muchas funciones se bloquean. `accept()` se bloquea. Todas las funciones `recv()` se bloquean. Lo hacen porque les está permitido hacerlo. Cuando creas un descriptor de *socket* con `socket()`, el núcleo lo configura como bloqueante. Si quieres que un *socket* no sea bloqueante, tienes que hacer una llamada a `fcntl()`:

```
#include <unistd.h>
#include <fcntl.h>
.
.
sockfd = socket(AF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
.
.
```

Al establecer un *socket* como no bloqueante, puedes de hecho "interrogar" al *socket* por su información. Si intentas leer de un *socket* no bloqueante y no hay datos disponibles, la función no está autorizada a bloquearse -- devolverá -1 y asignará a *`errno`* el valor `EWOULDBLOCK`.

En líneas generales, no obstante, este tipo de interrogaciones son una mala idea. Si pones tu programa a esperar sobre un bucle a que un *socket* tenga datos, estarás consumiendo en vano el tiempo de la CPU como se hacía antaño. Una solución más elegante para comprobar si hay datos esperando que se puedan leer, se presenta en la siguiente sección: `select()`.

6.2. `select()` --Multiplexado de E/S síncrona

Esta función es un tanto extraña, pero resulta muy útil. Considera la siguiente situación: eres un servidor y quieres escuchar nuevas conexiones entrantes al mismo tiempo que sigues leyendo de las conexiones que ya tienes.

Sin problemas, dices tú, un simple `accept()` y un par de `recv()`. ¡No tan deprisa! ¿qué pasa si te quedas bloqueado en la llamada a `accept()`? ¿cómo vas a recibir (`recv()`) datos al mismo tiempo? "Usa *sockets* no bloqueantes" ¡De ningún modo! No quieres comerte toda la CPU. Entonces ¿qué?

`select()` te da la posibilidad de comprobar varios *sockets* al mismo tiempo. Te dirá cuáles están listos para leer, cuáles están listos para escribir, y cuáles han generado excepciones, si estás interesado en saber eso.

Sin más preámbulos, esta es la sinopsis de `select()`:

```
    #include <sys/time.h>
    #include <sys/types.h>
    #include <unistd.h>
    int select(int numfds, fd_set *readfds, fd_set *writefds,
               fd_set *exceptfds, struct timeval *timeout);
```

La función comprueba "conjuntos" de descriptores de fichero; en concreto *`readfds`*, *`writefds`*, y *`exceptfds`*. Si quieres saber si es posible leer de la entrada estándar y de un cierto descriptor de *socket*, *`sockfd`*, simplemente añade los descriptores de fichero 0 y *`sockfd`* al conjunto *`readfs`*. El parámetro *`numfds`* debe tener el valor del mayor descriptor de fichero más uno. En este ejemplo, deberíamos asignarle el valor *`sockfd + 1`*, puesto que es seguro que tendrá un valor mayor que la entrada estándar (0).

Cuando `select()` regrese, *`readfs`* contendrá los descriptores de fichero que están listos para lectura. Puedes comprobarlos con la macro `FD_ISSET()` que se muestra a continuación.

Antes de progresar más, te contaré como manipular los conjuntos de descriptores. Cada conjunto es del tipo `fd_set`. Las siguientes macros funcionan sobre ese tipo.

- `FD_ZERO(fd_set *set)` -- borra un conjunto de descriptores de fichero
- `FD_SET(int fd, fd_set *set)` -- añade *`fd`* al conjunto
- `FD_CLR(int fd, fd_set *set)` -- quita *`fd`* del conjunto
- `FD_ISSET(int fd, fd_set *set)` -- pregunta si *`fd`* está en el conjunto

Por último, ¿qué es esa extraña estructura `struct timeval`? Bueno, a veces no quieres esperar toda la vida a que alguien te envíe datos. Quizás cada 96 segundos quieras imprimir "Aún estoy vivo..." aunque no haya sucedido nada. Esta estructura de tiempo te permite establecer un período máximo de espera. Si transcurre ese tiempo y `select()` no ha encontrado aún ningún descriptor de fichero que esté listo, la función regresará para que puedas seguir procesando.

La estructura `struct timeval` tiene los siguientes campos:

```
struct timeval {
    int tv_sec;     // segundos
    int tv_usec;    // microsegundos
};
```

Establece *`tv_sec`* al número de segundos que quieres esperar, y *`tv_usec`* al número de microsegundos. Sí, *micro*segundos, no milisegundos. Hay 1.000 microsegundos en un milisegundo, y 1.000 milisegundos en un segundo. Así que hay 1.000.000 de microsegundos en un segundo. ¿Por qué se llama "usec"? Se supone que la "u" es la letra griega μ (Mu) que suele usarse para abreviar "micro". Además, cuando `select()` regresa, *`timeout podría`*

haberse actualizado al tiempo que queda para que el temporizador indicado expire. Depende del sistema Unix que estés usando.

¡Fantástico! ¡Tenemos un reloj con precisión de microsegundos! No cuentes con ello. El límite en un sistema Unix estándar está alrededor de los 100 milisegundos, así que seguramente tendrás que esperar eso como mínimo, por muy pequeño que sea el valor con que establezcas la estructura `struct timeval`.

Otras cosas de interés: si estableces los campos de `struct timeval` a 0, `select()` regresará inmediatamente después de interrogar todos tus descriptores de fichero incluidos en los conjuntos. Si estableces el parámetro *`timeout`* a NULL el temporizador nunca expirará y tendrás que esperar hasta que algún descriptor de fichero esté listo. Por último, si no estás interesado en esperar sobre algún conjunto de descriptores de fichero en particular, sólo tienes que usar el parámetro NULL en la llamada a `select()`.

El siguiente fragmento de código espera 2.5 segundos a que algo aparezca por la entrada estándar:

```
/*
** select.c -- ejemplo de select()
*/
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#define STDIN 0  // descriptor de fichero de la entrada estándar
int main(void)
{
    struct timeval tv;
    fd_set readfds;
    tv.tv_sec = 2;
    tv.tv_usec = 500000;
    FD_ZERO(&readfds);
    FD_SET(STDIN, &readfds);
    // no nos preocupemos de writefds and exceptfds:
    select(STDIN+1, &readfds, NULL, NULL, &tv);
    if (FD_ISSET(STDIN, &readfds))
        printf("A key was pressed!\n");
    else
        printf("Timed out.\n");
    return 0;
}
```

Si estás en un terminal con *buffer* de líneas, tendrás que pulsar la tecla RETURN porque en cualquier otro caso el temporizador expirará.

Ahora, algunos de vosotros podrías pensar que esta es una forma excelente de esperar datos sobre un *socket* de datagramas --y teneis razón: *podría* serlo. Algunos Unix permiten usar `select()` de este modo mientras que otros no. Deberías consultar las páginas locales de `man` sobre este asunto antes de intentar usarlo.

Algunos Unix actualizan el tiempo en `struct timeval` para mostrar el tiempo que falta para que venza el temporizador. Pero otros no. No confíes en que esto ocurra si quieres ser portable. (Usa `gettimeofday()` si necesitas controlar el tiempo transcurrido. Sé que es un palo, pero así son las cosas)

¿Qué pasa si un *socket* en el conjunto de lectura cierra la conexión? En este caso `select()`

retorna con el descriptor de *socket* marcado como "listo para leer". Cuando uses `recv()`, devolverá 0. Así es como sabes que el cliente ha cerrado la conexión.

Una nota más de interés acerca de `select()`: si tienes un *socket* que está escuchando (`listen()`), puedes comprobar si hay una nueva conexión poniendo ese descriptor de *socket* en el conjunto *`readfs`* .

Y esto, amigos míos, es una visión general sencilla de la todopoderosa función `select()`.

Pero, por aclamación popular, ahí va un ejemplo en profundidad. Desgraciadamente, la diferencia entre el sencillo ejemplo anterior y el que sigue es muy importante. Pero échale un vstazo y lee las descripciones que siguen.

Este programa actúa como un simple servidor multiusuario de *chat*. Inicialo en una ventana y luego atácalo con **telnet** ("**telnet hostname 9034** ") desde varias ventanas distintas. Cuando escribas algo en una sesión **telnet**, tiene que aparecer en todas las otras.

```
    /*
    ** selectserver.c -- servidor de chat multiusuario
    */
    #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 <arpa/inet.h>
    #define PORT 9034   // puerto en el que escuchamos
    int main(void)
    {
        fd_set master;   // conjunto maestro de descriptores de fichero
        fd_set read_fds; // conjunto temporal de descriptores de fichero para
select()
        struct sockaddr_in myaddr;     // dirección del servidor
        struct sockaddr_in remoteaddr; // dirección del cliente
        int fdmax;        // número máximo de descriptores de fichero
        int listener;     // descriptor de socket a la escucha
        int newfd;        // descriptor de socket de nueva conexión aceptada
        char buf[256];    // buffer para datos del cliente
        int nbytes;
        int yes=1;        // para setsockopt() SO_REUSEADDR, más abajo
        int addrlen;
        int i, j;
        FD_ZERO(&master);    // borra los conjuntos maestro y temporal
        FD_ZERO(&read_fds);
        // obtener socket a la escucha
        if ((listener = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
            perror("socket");
            exit(1);
        }
        // obviar el mensaje "address already in use" (la dirección ya se está
usando)
        if (setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes,
                                                    sizeof(int)) == -1) {
            perror("setsockopt");
            exit(1);
        }
        // enlazar
        myaddr.sin_family = AF_INET;
        myaddr.sin_addr.s_addr = INADDR_ANY;
        myaddr.sin_port = htons(PORT);
        memset(&(myaddr.sin_zero), '\0', 8);
        if (bind(listener, (struct sockaddr *)&myaddr, sizeof(myaddr)) == -1) {
            perror("bind");
```

```
            exit(1);
        }
        // escuchar
        if (listen(listener, 10) == -1) {
            perror("listen");
            exit(1);
        }
        // añadir listener al conjunto maestro
        FD_SET(listener, &master);
        // seguir la pista del descriptor de fichero mayor
        fdmax = listener; // por ahora es éste
        // bucle principal
        for(;;) {
            read_fds = master; // cópialo
            if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
                perror("select");
                exit(1);
            }
            // explorar conexiones existentes en busca de datos que leer
            for(i = 0; i <= fdmax; i++) {
                if (FD_ISSET(i, &read_fds)) { // ¡¡tenemos datos!!
                    if (i == listener) {
                        // gestionar nuevas conexiones
                        addrlen = sizeof(remoteaddr);
                        if ((newfd = accept(listener, (struct sockaddr
*)&remoteaddr,
                                                            &addrlen)) == -1)
{
                            perror("accept");
                        } else {
                            FD_SET(newfd, &master); // añadir al conjunto maestro
                            if (newfd > fdmax) {    // actualizar el máximo
                                fdmax = newfd;
                            }
                            printf("selectserver: new connection from %s on "
                                "socket %d\n", inet_ntoa(remoteaddr.sin_addr),
newfd);
                        }
                    } else {
                        // gestionar datos de un cliente
                        if ((nbytes = recv(i, buf, sizeof(buf), 0)) <= 0) {
                            // error o conexión cerrada por el cliente
                            if (nbytes == 0) {
                                // conexión cerrada
                                printf("selectserver: socket %d hung up\n", i);
                            } else {
                                perror("recv");
                            }
                            close(i); // ¡Hasta luego!
                            FD_CLR(i, &master); // eliminar del conjunto maestro
                        } else {
                            // tenemos datos de algún cliente
                            for(j = 0; j <= fdmax; j++) {
                                // ¡enviar a todo el mundo!
                                if (FD_ISSET(j, &master)) {
                                    // excepto al listener y a nosotros mismos
                                    if (j != listener && j != i) {
                                        if (send(j, buf, nbytes, 0) == -1) {
                                            perror("send");
                                        }
                                    }
                                }
                            }
                        } // Esto es ¡TAN FEO!
                    }
                }
            }

            return 0;
        }
```

Observa que en el código uso dos conjuntos de descriptores de fichero: *`master`* y *`read_fds`*.

El primero, *master* , contiene todos los descriptores de fichero que están actualmente conectados, incluyendo el descriptor de *socket* que está escuchando para nuevas conexiones.

La razón por la cual uso el conjunto *master* es que `select()` va a cambiar el conjunto que le paso para reflejar qué sockets están listos para lectura. Como tengo que recordar las conexiones activas entre cada llamada de `select()`, necesito almacenar ese conjunto en algún lugar seguro. En el último momento copio *master* sobre *read_fs* y entonces llamo a `select()`.

Pero, ¿eso no significa que, cada vez que llegue una nueva conexión, tengo que añadirla al conjunto *master*? ¡Yup! y cada vez que una conexión se cierra, ¿no tengo que borrarla del conjunto *master* . Efectivamente.

Fíjate que compruebo si el *socket listener* está listo para lectura. Si lo está, tengo una nueva conexión pendiente: la acepto (`accept()`) y la añado al conjunto *master*. Del mismo modo, cuando una conexión de cliente está lista para lectura y `recv()` devuelve 0, sé que el cliente ha cerrado la conexión y tengo que borrarlo del conjunto *master*.

Sin embargo, si el cliente `recv()` devuelve un valor distinto de cero, sé que se han recibido datos así que los leo y recorro la lista *master* para enviarlos a todos los clientes conectados.

Y esto, amigos míos, es una visión general no tan sencilla de la todopoderosa función `select()`.

6.3. Gestión de envíos parciales con `send()`s

¿Recuerdas antes en la seción sobre `send()` , cuando dije que `send()` podría no enviar todos los *bytes* que pediste? Es decir, tú quieres enviar 512 *bytes*, pero `send()` devuelve el valor 412. ¿qué le ocurrió a los restantes 100 *bytes* ?

Bien, siguen estando en tu pequeño *buffer* esperando ser enviados. Debido a circunstancias que escapan a tu control, el núcleo decidió no enviar todos los datos de una sola vez, y ahora, amigo mío, depende de ti que los datos se envíen.

También podrías escribir una función como esta para conseguirlo:

```
#include <sys/types.h>
#include <sys/socket.h>
int sendall(int s, char *buf, int *len)
{
    int total = 0;        // cuántos bytes hemos enviado
    int bytesleft = *len; // cuántos se han quedado pendientes
    int n;
    while(total < *len) {
        n = send(s, buf+total, bytesleft, 0);
        if (n == -1) { break; }
        total += n;
        bytesleft -= n;
    }
    *len = total; // devuelve aquí la cantidad enviada en realidad
    return n==-1?-1:0; // devuelve -1 si hay fallo, 0 en otro caso
}
```

En este ejemplo, *s* es el *socket* al que quieres enviar los datos, *buf* es el *buffer* que contiene

los datos, y *len* es un puntero a un `int` que contiene el número de *bytes* que hay en el *buffer*.

La función devuelve -1 en caso de error (y *errno* está establecido por causa de la llamada a `send()`.) Además, el número de *bytes* enviados realmente se devuelven en *len*. Este será el mismo número de *bytes* que pediste enviar, a menos que sucediera un error. `sendall()` hará lo que pueda tratando de enviar los datos, pero si sucede un error regresará en seguida.

Este es un ejemplo completo de cómo usar la función:

```
    char buf[10] = "Beej!";
    int len;
    len = strlen(buf);
    if (sendall(s, buf, &len) == -1) {
        perror("sendall");
        printf("We only sent %d bytes because of the error!\n", len);
    }
```

¿Qué ocurre en el extremo receptor cuando llega un paquete? Si los paquetes tienen longitud variable, ¿cómo sabe el receptor cuando un paquete ha finalizado? Efectivamente, las situaciones del mundo real son un auténtico quebradero de cabeza. Probablemente tengas que *encapsular* tus datos (¿te acuerdas de esto, en la sección de encapsulación de datos más arriba al principio?) ¡Sigue leyendo para más detalles!

6.4. Consecuencias de la encapsulación de datos

En definitiva, ¿qué significa realmente encapsular los datos? En el caso más simple, significa que añadirás una cabecera con cierta información de identidad, con la longitud del paquete, o con ambas cosas.

¿Cómo tendría que ser esta cabecera? Bien, sencillamente algunos datos en binario que representen cualquier información que creas que es necesaria para tus propósitos.

Un poco impreciso, ¿no?.

Muy bien, por ejemplo, supongamos que tienes un programa de *chat* multiusuario que usa `SOCK_STREAM`. Cuando un usuario escribe ("dice") algo, es necesario transmitir al servidor dos tipos de información: qué se ha dicho y quién lo ha dicho.

¿Hasta aquí bien? "¿Cuál es el problema?", estás pensando.

El problema es que los mensajes pueden ser de longitudes distintas. Una persona que se llame "tom" podría decir "Hi" ["*hola*"], mientras que otra persona que se llame "Benjamin" podría decir "Hey guys what is up?" ["*Hey, ¿qué pasa tíos?*"]

Así que envías (`send()`) todo eso a los clientes tal y como llega. Tu cadena de datos salientes se parece a esto:

```
    t o m H i B e n j a m i n H e y g u y s w h a t i s u p ?
```

Y así siempre. ¿Cómo sabe un cliente cuándo empieza un mensaje y cuándo acaba? Si quisieras, podrías hacer que todos los mensajes tuvieran la misma longitud, y simplemente

llamar a la función `sendall()` que hemos implementado, más arriba . ¡Pero así desperdicias el ancho de banda! no queremos enviar (`send()`) 1024 *bytes* sólo para que "tom" pueda decir "Hi".

Así que *encapsulamos* los datos en una pequeña estructura de paquete con cabecera. Tanto el servidor como el cliente deben saber cómo empaquetar y desempaquetar los datos (algunas veces a esto se le llama, respectivamente, "marshal" y "unmarshal"). No mires aún, pero estamos empezando a definir un *protocolo* que describe cómo se comunican el servidor y el cliente.

En este caso, supongamos que el nombre de usuario tiene una longitud fija de 8 carácteres, rellenados por la derecha con `'\0'`. y supongamos que los datos son de longitud variable, hasta un máximo de 128 carácteres. Echemos un vistazo a un ejemplo de estructura de paquete que podríamos usar en esta situación:

1. `len` (1 *byte*, sin signo) -- La longitud total del paquete, que incluye los 8 *bytes* del nombre de usuario, y los datos de *chat*.

2. `name` (8 *bytes*) -- El nombre de usuario, completado con carácteres NUL si es necesario.

3. `chatdata` (*n-bytes*) -- Los datos propiamente dichos, hasta un máximo de 128 *bytes*. La longitud del paquete se calcula como la suma de la longitud de estos datos más 8 (la longitud del nombre de usuario).

¿Porqué elegí limites de 8 y 128 *bytes*? Los tomé al azar, suponiendo que serían lo bastante largos. Sin embargo, tal vez 8 *bytes* es demasiado restrictivo para tus necesidades, y quieras tener un campo nombre de 30 *bytes*, o más. La decisión es tuya.

Usando esta definición de paquete, el primer paquete constaría de la siguiente información (en hexadecimal y ASCII):

```
     0A     74 6F 6D 00 00 00 00 00      48 69
(longitud)  T  o  m    (relleno)         H  i
```

Y el segundo sería muy similar:

```
     14     42 65 6E 6A 61 6D 69 6E      48 65 79 20 67 75 79 73 20 77 ...
(longitud)  B  e  n  j  a  m  i  n       H  e  y     g  u  y  s     w  ...
```

(La longitud sigue la Ordenación de *bytes* de la red, por supuesto. En este caso no importa, porque se trata sólo de un *byte*, pero en general, querrás que todos tus enteros binarios de tus paquetes sigan la Ordenación de *bytes* de la red).

Al enviar estos datos, deberías ser prudente y usar una función del tipo de `sendall()` , así te aseguras de que se envían todos los datos incluso si hacen falta varias llamadas a `send()` para conseguirlo.

Del mismo modo, al recibir estos datos, necesitas hacer un poco de trabajo extra. Para asegurarte, deberías suponer que puedes recibir sólo una parte del paquete (por ejemplo, en el caso anterior podríamos recibir sólo " `00 14 42 65 6E`" del nombre "Benjamin" en nuestra llamada a `recv()`). Así que necesitaremos llamar a `recv()` una y otra vez hasta que el

paquete completo se reciba.

Pero, ¿cómo? Bueno, sabemos el número total de *bytes* que hemos de recibir para que el paquete esté completo, puesto que ese número está al principio del paquete. También sabemos que el tamaño máximo de un paquete es 1+8+128, es decir 137 *bytes* (lo sabemos porque así es como hemos definido el paquete)

Lo que puedes hacer es declarar un vector [*array*] lo bastante grande como para contener dos paquetes. Este será tu *buffer* de trabajo donde reconstruirás los paquetes a medida que lleguen.

Cada vez que recibas (`recv()`) datos los meterás en tu *buffer* y comprobarás si el paquete está compelto. Es decir, si el número de *bytes* en el *buffer* es mayor o igual a la longitud indicada en la cabecera (+1, porque la longitud de la cabecera no incluye al propio *byte* que indica la longitud). Si el número de *bytes* en el *buffer* es menor que 1, el paquete, obviamente, no está completo. Sin embargo, tienes que hacer un caso especial para esto ya que el primer *byte* es basura y no puedes confiar en que contenga una longitud de paquete correcta.

Una vez que el paquete está completo, puedes hacer con él lo que quieras. Úsalo y bórralo del *buffer*.

¡Bueno! ¿Aún estás dándole vueltas en la cabeza? Bien, ahí llega la segunda parte: en una sola llamada a `recv()`, podrías haber leído más allá del final de un paquete, sobre el principio del siguiente. Es decir, tienes un *buffer* con un paquete completo y una parte del siguiente paquete. Maldita sea. (Pero esta es la razón por la que hiciste que tu *buffer* fuera lo bastante grande como para contener *dos* paquetes-- ¡Por si sucedía esto!)

Como, gracias a la cabecera, sabes la longitud del primer paquete y además has controlado cuántos *bytes* del *buffer* has usado, con una sencilla resta puedes calcular cuántos de los *bytes* del *buffer* corresponden al segundo paquete (incompleto). Cuando hayas procesado el primer paquete puedes borrarlo del *buffer* y mover el fragmento del segundo paquete al principio del *buffer* para poder seguir con la siguiente llamada a `recv()`.

(Algunos de vosotros os habeis dado cuenta de que mover el fragmento del segundo paquete al principo de *buffer* lleva tiempo, y que el programa se puede diseñar de forma que esto no sea necesario por medio del uso de un *buffer* circular. Desgraciadamente para el resto, el examen de los *buffers* circulares va más allá del alcance de este artículo. Si todavía sientes curiosidad, píllate un libro de estructuras de datos y consúltalo.)

Nunca dije que fuera fácil. Está bien, sí que lo dije. Y lo es. Sólo necesitas práctica y muy pronto resultará para ti de lo más natural. ¡Lo juro por Excalibur!

7. Referencias adicionales

¡Has llegado hasta aquí, y ahora quieres más! ¿A dónde puedes ir para aprender más de todo esto?

7.1. Páginas del manual (`man`)

Revisa las siguientes páginas `man`, para información inicial:

- `htonl()`
- `htons()`
- `ntohl()`
- `ntohs()`
- `inet_aton()`
- `inet_addr()`
- `inet_ntoa()`
- `socket()`
- `socket options`
- `bind()`
- `connect()`
- `listen()`
- `accept()`
- `send()`
- `recv()`
- `sendto()`
- `recvfrom()`
- `close()`
- `shutdown()`
- `getpeername()`
- `getsockname()`
- `gethostbyname()`

- `gethostbyaddr()`
- `getprotobyname()`
- `fcntl()`
- `select()`
- `perror()`
- `gettimeofday()`

7.2. Libros

Si buscas libros auténticos que se pueden tocar revisa alguna de las excelentes guías que hay a continuación. Fíjate en el destacado logo de Amazon.com. Lo que esta descarada inducción al consumismo significa es que, básicamente, obtengo una comisión por vender esos libros desde esta guía (en realidad, obtengo libros). Por eso, si de todas formas vas a comprar uno de esos libros, ¿por qué no enviarme un agradecimiento especial iniciando tu compra desde uno de los enlaces de ahí abajo?

Además, más libros para mí redundan, en última instancia, en más guías para ti. ; -)

Unix Network Programming, volumes 1-2 de W. Richard Stevens. Editado por Prentice Hall. ISBNs para los volúmenes 1-2: 013490012X , 0130810819 .

Internetworking with TCP/IP, volumes I-III de Douglas E. Comer y David L. Stevens. Editado por Prentice Hall. ISBNs para los volúmenes I, II, y III: 0130183806 , 0139738436 , 0138487146 .

TCP/IP Illustrated, volumes 1-3 de W. Richard Stevens y Gary R. Wright. Editado por Addison Wesley. ISBNs para los volúmenes 1, 2, y 3: 0201633469 , 020163354X , 0201634953 .

TCP/IP Network Administration de Craig Hunt. Editado por O'Reilly & Associates, Inc. ISBN 1565923227 .

Advanced Programming in the UNIX Environment de W. Richard Stevens. Editado por Addison Wesley. ISBN 0201563177 .

Using C on the UNIX System de David A. Curry. Editado por O'Reilly & Associates, Inc. ISBN 0937175234. *Fuera de catálogo.*

7.3. Referencias en la *web*

En la *web*:

BSD Sockets: A Quick And Dirty Primer (¡Contiene además información de otros aspectos de la

programación en UNIX!)

The Unix Socket FAQ

Client-Server Computing

Intro to TCP/IP (gopher)

Internet Protocol Frequently Asked Questions

The Winsock FAQ

7.4. RFCs

RFCs --La verdadera materia:

RFC-768 --El Protocolo de Datagramas de Usuario (UDP)

RFC-791 --El protocolo de Internet (IP)

RFC-793 --El Protocolo de Control de Transmisión (TCP)

RFC-854 --El Protocolo *Telnet*

RFC-951 --El Protocolo de Arranque [*Bootstrap*] (BOOTP)

RFC-1350 --El Protocolo Trivial de Transferencia de Archivos. (TFTP)

8. Preguntas más comunes

Q: ¿De dónde saco esos archivos de cabecera?

A: Si no están ya en tu sistema, probablemente no los necesitas. Consulta el manual de tu plataforma. Si estás compilando en *Windows* sólo necesitas incluir `#include <winsock.h>`.

Q: ¿Qué hago cuando `bind()` responde "Address already in use" [La dirección ya se está usando]?

A: Tienes que usar `setsockopt()` con la opción SO_REUSEADDR sobre el *socket* en el que estás escuchando (`listen()`). Consulta la sección sobre `bind()` y la sección sobre `select()` para ver un ejemplo.

Q: ¿Cómo puedo obtener una lista de los *sockets* abiertos en el sistema?

A: Usa **`netstat`** . Revisa la página **`man`** para conocer a fondo los detalles, pero deberías tener un resultado aceptable con sólo teclear:

```
$ netstat
```

El único truco consiste en averiguar qué *socket* está asociado con qué programa. :-)

Q: ¿Cómo se ve la tabla de encaminamiento?

A: Usa el comando **`route`** (En la mayoría de Linuxes lo encontrarás en `/sbin`) o el comando **`netstat -r`**.

Q: ¿Cómo puedo ejecutar el servidor y el cliente si solamente tengo un ordenador? ¿No necesito una red para escribir programas de redes?

A: Afortunadamente para ti, no. Virtualmente todas las máquinas implementan un "dispositivo" de red de cierre de circuito [*loopback*] que reside en el núcleo y simula ser una tarjeta de red. (Este es el interfaz que se lista como "lo" en la tabla de encaminamiento.)

Supón que has iniciado sesión en una máquina que se llama "`goat`". Ejecuta el cliente en una ventana y el servidor en otra. O inicia el servidor en segundo plano [*background*] (escribe "**`servidor &`**") y ejecuta el cliente en la misma ventana. Como consecuencia del dispositivo de cierre de circuito puedes sencillamente ejecutar **`cliente goat`** o **`cliente localhost`** (puesto que "`localhost`", muy probablemente, está definido en tu fichero `/etc/hosts`) y tu cliente estará hablando de inmediato con el servidor.

En resumen, ¡no es necesario cambiar nada para que el código funcione en una sola máquina sin conexión a la red! ¡Magnífico!

Q: ¿Cómo puedo saber si el sistema remoto ha cerrado la conexión?

A: Lo sabes porque `recv()` devuelve `0`.

Q: ¿Cómo se implementa la utilidad "ping"? ¿Qué es ICMP? ¿Dónde puedo averiguar más cosas sobre los *sockets* puros [*raw sockets*]?

A: Todas tus dudas referidas a *sockets* puros tienen respuesta en los libros de programación UNIX en redes de W. Richard Stevens. Consulta la sección de libros de esta guía.

Q: ¿Cómo compilo sobre *Windows* ?

A: Empieza borrando *Windows* e instalando en su lugar Linux o BSD. `};-)` . No, en realidad sólo has de seguir las recomendaciones de la sección sobre compilación en *Windows* de la introducción.

Q: ¿Cómo compilo sobre *Solaris/SunOS* ? ¡Todavía tengo errores al enlazar!

A: Los errores de enlace suceden porque las máquinas *Sun* no enlazan automáticamente las bibliotecas de *sockets*. Consulta la sección sobre compilación en *Solaris/SunOS* de la introducción, donde hallarás un ejemplo de cómo hacerlo.

Q: ¿Por qué termina `select()` al recibir una señal?

A: Las señales suelen causar que las llamadas al sistema que están bloqueadas devuelvan el valor -1 con *errno* establecifo a EINTR. Cuando preparas un manejador de señales con `sigaction()` , puedes establecer el indicador SA_RESTART, que se supone que reiniciará la llamada al sistema que fue interrumpida.

Naturalmente, esto no siempre funciona.

Mi solución favorita a esto, requiere el uso de una sentencia `goto`. Sabes que esto irrita a tus profesores sobremanera, así que ¡al ataque!

```
select_restart:
    if ((err = select(fdmax+1, &readfds, NULL, NULL, NULL)) == -1) {
        if (errno == EINTR) {
            // Alguna señal nos ha interrumpido, así que regresemos a select()
            goto select_restart;
        }
        // Los errores reales de select() se manejan aquí:
        perror("select");
    }
```

Por supuesto que, en este caso, no necesitas una sentencia `goto`; puedes usar otras estructuras de control. Pero creo que `goto` es una solución más limpia.

Q: ¿Cómo puedo implementar un temporizador [*timeout*] en una llamada a `recv()`?

A: ¡Usa `select()` ! Te permite indicar un parámetro de control de tiempo sobre los descriptores de *socket* en los que esperas leer. También puedes implementar toda esa funcionalidad en una única función como esta:

```
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
int recvtimeout(int s, char *buf, int len, int timeout)
{
    fd_set fds;
    int n;
    struct timeval tv;
    // Construir el conjunto de descriptores de fichero
```

```
    FD_ZERO(&fds);
    FD_SET(s, &fds);
    // Construir la estructura timeval del temporizador
    tv.tv_sec = timeout;
    tv.tv_usec = 0;
    // Esperar hasta que se reciban datos o venza el temporizador
    n = select(s+1, &fds, NULL, NULL, &tv);
    if (n == 0) return -2; // ¡El temporizador ha vencido!
    if (n == -1) return -1; // error
    // Los datos deben estar ahí, así que llamo normalmente a recv()
    return recv(s, buf, len, 0);
}
// Ejemplo de llamada a recvtimeout():
    .
    .
    n = recvtimeout(s, buf, sizeof(buf), 10); // 10 segundos de espera
    if (n == -1) {
        // ocurrió un error
        perror("recvtimeout");
    }
    else if (n == -2) {
        // El temporizador venció
    } else {
        // Hay datos en el buffer
    }
    .
    .
```

Nota que `recvtimeout()` devuelve `-2` en caso de que venza el temporizador. ¿Por qué no devolver `0`? Bueno, si recuerdas un valor de retorno de 0 para una llamada a `recv()` significa que la máquina remota ha cerrado la conexión. Así que ese valor de retorno ya tiene un significado. El valor -1 significa "error", así que escogí -2 como mi indicador de temporizador vencido.

Q: ¿Cómo puedo comprimir o encriptar los datos antes de enviarlos al *socket*?

A: Una forma sencilla de encriptar es usar SSL (*secure sockets layer*), pero eso va más allá del alcance de esta guía.

Pero suponiendo que quieres implementar tu propio sistema de compresión o encriptado, sólo es cuestión de pensar que tus datos siguen una secuencia de etapas entre ambos extremos. Cada paso altera los datos en cierta manera.

1. El servidor lee datos de un fichero (o de donde sea)

2. El servidor encripta los datos (tú añades esta parte)

3. El servidor envía (`send()`) los datos encriptados

Y ahora el camino inverso:

4. El cliente recibe (`recv()`) los datos encriptados

5. El cliente desencripta los datos (tú añades esta parte)

6. El cliente escribe los datos en un archivo (o donde sea)

Podrías comprimir los datos en el punto en que arriba se encritan/desencriptan los datos. ¡O

podrías hacer las dos cosas!. Sencillamente recuerda comprimir antes de encriptar. :)

En la medida en que el cliente deshaga correctamente lo que el servidor haga, los datos llegarán correctamente al final, sin importar cuántos pasos intermedios se añadan.

Así que todo lo que necesitas para usar mi código es encontrar el lugar situado entre la lectura y el envío (send()) de los datos, y poner allí el código que se encarga de realizar el encriptado.

Q: ¿Qué es eso de " `PF_INET`" ? ¿Tiene algo que ver con `AF_INET` ?

A: Por supuesto que sí. Revisa la sección `socket()` para los detalles.

Q: ¿Cómo puedo escribir un servidor que acepte comandos de *shell* de un cliente y los ejecute?

A: Para simplificar las cosas, digamos que el cliente conecta (`connect()`), envía (`send()`), y cierra (`close()`) la conexión (es decir no hay llamadas al sistema posteriores sin que el cliente vuelva a conectar.)

El proceso que sigue el cliente es este:

1. `connect()` con el servidor

2. `send("/sbin/ls > /tmp/client.out")`

3. `close()` la conexión

Mientras tanto el servidor procesa los datos de esta manera:

1. `accept()` la conexión del cliente

2. `recv(str)` el comando a ejecutar

3. `close()` la conexión

4. `system(str)` ejecuta el comando

¡Cuidado! Dejar que el servidor ejecute lo que el cliente dice es como dar acceso remoto al *shell* y la gente podría hacer cosas no deseadas con tu cuenta cuando se conectan al servidor. Por ejemplo, en el ejemplo anterior, ¿qué pasa si el cliente envía "**`rm -rf *`** "? Borra todo lo que tengas en tu cuenta, ¡eso es lo que pasa!

Así que aprendes la lección e impides que el cliente use cualquier comando, a excepción de un par de utilidades que sabes que son seguras, como la utilidad **foobar** :

```
    if (!strcmp(str, "foobar")) {
        sprintf(sysstr, "%s > /tmp/server.out", str);
        system(sysstr);
    }
```

Desgraciadamente, todavía no estás seguro: ¿qué pasa si el cliente envía "**`foobar; rm -rf`**

~ "? Lo mejor que se puede hacer es escribir una pequeña rutina que ponga un carácter de escape("\") delante de cualquier carácter no alfanumérico (incluyendo espacios, si es necesario) en los argumentos para el comando.

Como puedes ver, la seguridad es un problema bastante importante cuando el servidor comienza a ejecutar cosas que el cliente envía.

Q: ¿Estoy enviando un montón de datos, pero cuando hago `recv()`, sólo recibo 536 *bytes* ó 1460 *bytes* a la vez. Sin embargo, si ejecuto en mi máquina local recibo todos los datos al mismo tiempo. ¿Qué pasa?

A: Te estás tropezando con la MTU (*Maximum Transfer Unit*)--El tamaño máximo de paquete que el medio físico puede manejar. En la máquina local estás usando el dispositivo de cierre de circuito [*loopback*] que puede manejar sin problemas 8K o más. Pero en una Ethernet, que sólo puede manejar 1500 *bytes* con una cabecera, tienes que plegarte a ese límite. Sobre un módem, con una MTU de 576 (con cabecera), has de plegarte a un límite aún menor.

En primer lugar tienes que asegurarte de que todos los datos se están enviando. (Revisa la implementación de la función `sendall()` para ver los detalles). Una vez que estás seguro de eso, tienes que llamar a `recv()` en un bucle hasta que todos los datos se hayan leído.

Revisa la sección Consecuencias de la encapsulación de datos para los detalles acerca de como recibir paquetes completos de datos usando múltiples llamadas a `recv()`.

Q: Yo uso *Windows* y no tengo la llamada al sistema `fork()` ni nigún tipo de estructura `struct sigaction`. ¿Qué hago?

A: De estar en algún sitio estarán en las bibliotecas `POSIX` que quizás hayan venido con tu compilador. Como no tengo *Windows*, en realidad no puedo darte la respuesta, pero creo recordar que Microsoft tiene una capa de compatibilidad `POSIX`, y ahí es donde `fork()` tendría que estar (y a lo mejor también `sigaction`.)

Busca "fork" o "POSIX" en la ayuda de VC++, por si te da alguna pista.

Si no hay forma de que funcione, olvídate de `fork()/sigaction` y usa en su lugar la función equivalente de *Win32*: `CreateProcess()`. No sé cómo se usa `CreateProcess()` --Tiene tropecientos argumentos, pero seguramente está explicada en la ayuda de VC++

Q: ¿Cómo puedo enviar datos seguros con TCP/IP usando encriptación?

A: Visita el Proyecto OpenSSL.

Q: Estoy detrás de un cortafuegos [*firewall*]--¿Cómo informo a la gente del otro lado del cortafuegos de cual es mi dirección IP para que puedan conectarse a mi máquina?

A: Desgraciadamente, la finalidad de un cortafuegos es evitar que la gente del otro lado del cortafuegos pueda conectarse a las máquinas de este lado del cortafuegos, así que permitir que lo hagan se considera en principio una brecha de seguridad.

No es que diga que todo está perdido. Por una parte, todavía puedes conectar (`connect()`) a

través del cortafuegos si éste está realizando algún tipo de enmascaramiento [*masquerading*] o NAT [*Network Address Translation* - Traducción de direcciones de red] o algo por el estilo. Tan sólo tienes que diseñar tus programas de modo que seas tú siempre quien inicia la conexión, y todo irá bien.

Si esta solución no es satisfactoria, puedes pedir al administrador de tu sistema que practique un agujero en el cortafuegos de modo que la gente pueda conectarse contigo. El cortafuegos puede reenviarte sus datos, bien mediante el software NAT, bien mediante un proxy o algo similar.

Ten en cuenta que un agujero en el cortafuegos no es algo que pueda uno tomarse a la ligera. Tienes que estar muy seguro de que no das acceso a la red interior a personas con malas intenciones; si eres un principiante, hacer software seguro es mucho más difícil de lo que puedas imaginar.

No hagas que el administrador de tu sistema se enfade conmigo. ; -)

9. Declinación de responsabilidad y solicitud de ayuda

Bien, esto es todo. Con suerte, por lo menos parte de la información contenida en este documento habrá sido remotamente exacta, y sinceramente espero que no haya errores muy descarados. Bueno, seguro, siempre los hay.

¡Así que esto te sirva de advertencia! Lamento mucho que alguna de las imprecisiones que has encontrado te haya causado problemas, pero sencillamente no puedes culparme. Legalmente hablando, no me responsabilizo de una sola de las palabras de este documento. Todo él podría ser completamente erróneo.

Aunque probablemente no es ese el caso. Al fin y al cabo, he invertido un montón de horas mareándome con todo esto, y en el trabajo he implementado varias aplicaciones de red `TCP/IP`. Además, he escrito los núcleos de algunos juegos multiusuario, y algunas cosas más. Pero no soy el dios de los *sockets*. Solamente soy un tío.

Por cierto, si alguien tiene alguna crítica constructiva (o destructiva) sobre este documento, por favor que la envíe a < `beej@piratehaven.org` > y trataré de hacer un esfuerzo para corregirla.

Por si acaso te preguntas por qué hice esto, bueno, lo hice por dinero. Ha! en realidad no. Lo hice porque un montón de gente me preguntaba cosas acerca de los *sockets*, y cuando les decía que había estado pensando en ponerlo todo junto en una página de *sockets*, me decían, "¡estupendo!" Además, me parecía que todo este conocimiento que tanto me ha costado adquirir iba a desperdiciarse si no podía compartirlo con otros. La *web* solamente es el vehículo perfecto. Animo a los demás a suministrar información parecida en la medida en que sea posible.

¡Basta de charlas--volved al trabajo! ;-)

Nota del traductor
Con la sola excepción de este párrafo, que obviamente no está en la versión original inglesa, he tratado de expresar en español, fielmente, el contenido de esta guía. Espero que te sea de alguna ayuda. Sin embargo, debe quedar claro que usas la versión española de esta guía bajo tu entera responsabilidad, puesto que yo la declino, ante cualquier problema de cualquier índole que pudiera derivarse de una imprecisión o error en la traducción. Honestamente, espero que no haya otros errores, más que los inevitables errores de tecleo.