Guía de Beej para la programación en red

Uso de enchufes de Internet

Brian "Beej Jorgensen" Hall

v3.2.11, Derechos de autor © 8

de julio de 2025

Contenido

1 Introducción 1
1.1 Audiencia ... 1
1.2 Plataforma y compilador ... 1
1.3 Página oficial y libros a la venta ... 1
1.4 Nota para los programadores de Solaris/SunOS/illumos ... 1
1.5 Nota para programadores de Windows ... 2
1.6 Política de correo electrónico ... 4
1.7 Espejado ... 4
1.8 Nota para los traductores ... 4
1.9 Derechos de autor, distribución y legal ... 4
1.10 Dedicación ... 5
1.11 Información de publicación ... 5

2 ¿Qué es un enchufe? 6
2.1 Dos tipos de enchufes de Internet ... 6
2.2 Tonterías de bajo nivel y teoría de redes ... 7

3 Direcciones IP, `estructuras,` y Munging de Datos 9
3.1 Direcciones IP, versiones 4 y 6 ... 9
3.1.1 Subredes ... 10
3.1.2 Números de puerto ... 11
3.2 Orden de bytes ... 11
3.3 `Estructuras` ... 12
3.4 Direcciones IP, Part Deux ... 15
3.4.1 Redes privadas (o desconectadas) ... 16

4 Salto de IPv4 a IPv6 17

5 Llamadas al sistema o caída 19
5.1 `getaddrinfo()`—¡Prepárate para el lanzamiento! ... 19
5.2 `socket()`—¡Obtén el descriptor de archivo! ... 22
5.3 `bind()`—¿En qué puerto estoy? ... 23
5.4 `conectar()`—¡Oye, tú! ... 25
5.5 `escuchar()`—¿Alguien podría llamarme, por favor? ... 26
5.6 `aceptar()`—"Gracias por llamar al puerto 3490." ... 26
5.7 `send()` y `recv()`—¡Háblame, nena! ... 27
5.8 `sendto()` y `recvfrom()`—Háblame, al estilo DGRAM ... 28
5.9 `cerrar()` y `shutdown()`"¡Quítate de mi cara!" ... 29
5.10 `getpeername()`—¿Quién es usted? ... 30
5.11 `gethostname()`—¿Quién soy yo? ... 30

6 Antecedentes cliente-servidor 32
6.1 Un servidor de transmisión simple ... 32
6.2 Un cliente de transmisión simple ... 35
6.3 Sockets de datagramas ... 37

7 Técnicas ligeramente avanzadas 42
7.1 Bloqueante ... 42

7.2 encuesta()—Multiplexación de E/S síncrona ... 43
7.3 seleccionar()—Multiplexación de E/S síncrona, vieja escuela ... 50
7.4 Manejo parcial send()s ... 56
7.5 Serialización: cómo empaquetar datos ... 57
7.6 Hijo de la encapsulación de datos ... 70
7.7 Paquetes de transmisión—¡Hola, mundo! ... 72

8 Preguntas comunes 76

9 Páginas de manual 82
9.1 aceptar() ... 82
9.2 bind() ... 84
9.3 conectar() ... 85
9.4 cerrar() ... 87
9.5 getaddrinfo(), freeaddrinfo(), gai_strerror() ... 87
9.6 gethostname() ... 91
9.7 gethostbyname(), gethostbyaddr() ... 91
9.8 getnameinfo() ... 94
9.9 getpeername() ... 95
9.10 errno ... 96
9.11 fcntl() ... 97
9.12 htons(), htonl(), ntohs(), ntohl() ... 97
9.13 inet_ntoa(), inet_aton(), inet_addr ... 99
9.14 inet_ntop(), inet_pton() ... 100
9.15 escuchar() ... 102
9.16 perror(), strerror() ... 103
9.17 encuesta() ... 104
9.18 recv(), recvfrom() ... 106
9.19 seleccionar() ... 108
9.20 setsockopt(), getsockopt() ... 110
9.21 send(), sendto() ... 111
9.22 shutdown() ... 113
9.23 socket() ... 114
9.24 struct sockaddr y amigos ... 115

10 Más referencias 118
10.1 Libros ... 118
10.2 Referencias Web ... 118
10.3 RFC ... 119

Capítulo 1

Introducción

¡Eh! ¿La programación de sockets te deprimió? ¿Es esto demasiado difícil de descifrar a partir de las páginas del `manual`? Quieres hacer una programación genial en Internet, pero no tienes tiempo para vadear un montón de `estructuras tratando de averiguar si tienes que llamar a bind()` antes de `conectarte(),` etc., etc.

Bueno, ¡adivina qué! ¡Ya he hecho este desagradable negocio, y me muero de ganas de compartir la información con todo el mundo! Has venido al lugar correcto. Este documento debería darle al programador de C competente promedio la ventaja que necesita para controlar este ruido de red.

Y compruébalo: ¡finalmente me he puesto al día con el futuro (¡también justo a tiempo!) y he actualizado la Guía para IPv6. ¡Disfrutar!

1.1 Audiencia

Este documento ha sido escrito como un tutorial, no como una referencia completa. Probablemente esté en su mejor momento cuando lo lean las personas que recién comienzan con la programación de sockets y buscan un punto de apoyo. Ciertamente no es la *guía completa y total* para la programación de sockets, de ninguna manera.

Con suerte, sin embargo, será suficiente para que esas páginas de manual comiencen a tener sentido... `:-)`

1.2 Plataforma y compilador

El código contenido en este documento fue compilado en una PC Linux utilizando el `compilador gcc` de Gnu. Sin embargo, debería basarse en casi cualquier plataforma que utilice `gcc`. Naturalmente, esto no se aplica si está programando para Windows: consulte la sección sobre programación de Windows, a continuación.

1.3 Página oficial y libros a la venta

Esta ubicación oficial de este documento es:

- `https://beej.us/guide/bgnet/`

Allí también encontrará ejemplos de código y traducciones de la guía a varios idiomas.

Para comprar ejemplares impresos bien encuadernados (algunos los llaman "libros"),

visite:

- `https://beej.us/guide/url/bgbuy`

¡Agradeceré la compra porque ayuda a mantener mi estilo de vida de escritura de documentos!

1.4 Nota para los programadores de Solaris/SunOS/illumos

Al compilar para una variante de Solaris o SunOS, debe especificar algunos modificadores de línea de comandos adicionales para vincular en las bibliotecas adecuadas. Para hacer esto, simplemente agregue "`-lnsl -lsocket -lresolv`" a la etiqueta

fin del comando compile, de la siguiente manera:

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

Si aún recibe errores, puede intentar agregar un `-lxnet` al final de esa línea de comandos. No sé qué hace eso exactamente, pero algunas personas parecen necesitarlo.

Otro lugar en el que puedes encontrar problemas es en la llamada a `setsockopt()`. El prototipo difiere del de mi máquina Linux, así que en lugar de:

```
int sí=1;
```

Entra en esto:

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

Como no tengo una caja Sun, no he probado ninguna de la información anterior, es solo lo que la gente me ha dicho por correo electrónico.

1.5 Nota para programadores de Windows

En este punto de la guía, históricamente, he hecho un poco de embolsado en Windows, simplemente por el hecho de que no me gusta mucho. Pero luego Windows y Microsoft (como empresa) mejoraron mucho. Windows 10 junto con WSL (abajo) en realidad lo convierte en un sistema operativo decente. Realmente no hay mucho de qué quejarse.

Bueno, un poco, por ejemplo, estoy escribiendo esto (en 2025) en una computadora portátil de 2015 que solía ejecutar Windows 10. Finalmente se volvió demasiado lento e instalé Linux en él. Y lo he estado usando desde entonces.

Pero ahora tenemos Windows 11 que aparentemente requiere un hardware más robusto que Windows 10. No soy fanático de eso. El sistema operativo debe ser lo más discreto posible y no requerir que gaste más dinero. La potencia adicional de la CPU debe ser para las aplicaciones, no para el sistema operativo. Además, Microsoft sabe lo que quieres, ¡y lo que quieres es más publicidad! ¿Derecha? ¡En tu sistema operativo! ¿No te lo estabas perdiendo? Ahora puedes tenerlo con Windows 11.

Así que... Todavía te animo a que pruebes Linux[1], BSD[2], illumos[3] o cualquier otro sabor de Unix en lugar de Windows. ¿Cómo llegó esa tribuna allí?

Pero a la gente le gusta lo que le gusta, y a la gente de Windows le complacerá saber que esta información es generalmente aplicable a Windows, con algunos cambios menores.

Una cosa que debe considerar seriamente es el Subsistema de Windows para Linux[4]. Básicamente, esto le permite instalar una máquina virtual de Linux en Windows 10. Eso también te situará definitivamente, y podrás crear y ejecutar estos programas tal cual.

Otra cosa que puedes hacer es instalar Cygwin[5], que es una colección de herramientas Unix para Windows. He oído en la vid que hacerlo permite que todos estos programas se compilen sin modificar, pero nunca lo he probado.

Es posible que algunos de ustedes quieran hacer las cosas a la manera de Windows Pure. Eso es muy valiente de tu parte, y esto es lo que tienes que hacer: ¡salir corriendo y obtener Unix de inmediato! No, no, estoy bromeando. Se supone que soy amigable con Windows en estos días ...

Está bien. Voy a seguir con ello.

Esto es lo que tendrás que hacer: primero, ignora casi todos los archivos de encabezado del sistema que menciono aquí. En su lugar, incluya:

---

[1] https://www.linux.com/
[2] https://bsd.org/
[3] https://www.illumos.org/
[4] https://learn.microsoft.com/en-us/windows/wsl/
[5] https://cygwin.com/

```
#include <winsock2.h>
#include <ws2tcpip.h>
```

`winsock2` es la "nueva" versión (alrededor de 1994) de la biblioteca de sockets de Windows.

Desafortunadamente, si incluye `windows.h`, automáticamente extrae el archivo de `encabezado winsock.h` (versión 1) anterior que entra en conflicto con `winsock2.h`. Momentos divertidos.

Por lo tanto, si tiene que incluir `windows.h`, debe definir una macro para que *no* incluya el encabezado anterior:

```
#define WIN32_LEAN_AND_MEAN // Di esto...

#include <windows.h>          Y ahora podemos incluir eso.
#include <winsock2.h>         Y esto.
```

¡Esperar! También tienes que hacer una llamada a `WSAStartup()` antes de hacer cualquier otra cosa con la biblioteca de sockets. Pasa la versión de Winsock que desee a esta función (por ejemplo, la versión 2.2). Y luego puede verificar el resultado para asegurarse de que la versión esté disponible.

El código para hacerlo es similar al siguiente:

```
#include <winsock2.h>

{
    WSADATA wsaData;

    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        fprintf(stderr, "WSAStartup falló.\n");
        salida(1);
    }

    if (LOBYTE(wsaData.wVersión) != 2 ||
        HIBYTE(wsaData.wVersión) != 2)
    {
        fprintf(stderr,"La versión 2.2 de Winsock no está
        disponible.\n"); WSACleanup();
        salida(2);
    }
```

Tenga en cuenta que la llamada a `WSACleanup()` está allí. Eso es lo que quieres llamar cuando termines con la biblioteca de Winsock.

También tienes que decirle a tu compilador que enlace en la biblioteca de Winsock, llamada `ws2_32.lib` para Winsock 2. En VC++, esto se puede hacer a través del `menú Proyecto`, en `Configuración.` Haga clic en la `pestaña Enlace` y
busque la casilla titulada "Módulos de objeto/biblioteca". Agregue "ws2_32.lib" (o la biblioteca que prefiera) a esa lista.

O eso es lo que escucho.

Una vez hecho esto, el resto de los ejemplos de este tutorial deberían aplicarse generalmente, con algunas excepciones. Por un lado, no puedes usar `close()` para cerrar un socket, necesitas usar `closesocket()`, en su lugar. Además, `select()` solo funciona con descriptores de socket, no con descriptores de archivo (como `0` para `stdin`).

También hay una clase de socket que puede usar, `CSocket` Consulte las páginas de ayuda de su compilador para obtener más información.

Para obtener más información sobre Winsock, consulte la página oficial de Microsoft.

Finalmente, escuché que Windows no tiene una `llamada al sistema fork()` que, desafortunadamente, se usa en algunos de mis ejemplos. Tal vez tengas que enlazar en una biblioteca POSIX o algo así para que funcione, o puedes usar `CreateProcess()` en su lugar. `fork()` no toma argumentos, y `CreateProcess()` toma alrededor de 48 mil millones de argumentos. Si no estás preparado para eso, el `CreateThread()` es un poco más fácil de digerir... Desgraciadamente, un

La discusión sobre el subprocesamiento múltiple está más allá del alcance de este documento. No puedo hablar de mucho, ¿sabes?

Por último, Steven Mitchell ha portado varios de los ejemplos[6] a Winsock. Echa un vistazo a esas cosas.

1.6 Política de correo electrónico

Por lo general, estoy disponible para ayudar con las preguntas por correo electrónico, así que no dude en escribir, pero no puedo garantizar una respuesta. Llevo una vida bastante ocupada y hay momentos en los que simplemente no puedo responder a una pregunta que tienes. Cuando ese es el caso, por lo general simplemente elimino el mensaje. No es nada personal; Simplemente nunca tendré tiempo para dar la respuesta detallada que necesita.

Como regla general, cuanto más compleja sea la pregunta, menos probable es que responda. Si puedes acotar tu pregunta antes de enviarla por correo y asegurarte de incluir cualquier información pertinente (como la plataforma, el compilador, los mensajes de error que recibes y cualquier otra cosa que creas que pueda ayudarme a solucionar el problema), es mucho más probable que obtengas una respuesta. Para obtener más consejos, lea el documento de ESR, Cómo hacer preguntas de manera inteligente [7].

Si no obtienes una respuesta, hazlo un poco más, trata de encontrar la respuesta, y si aún es difícil de alcanzar, entonces escríbeme de nuevo con la información que has encontrado y espero que sea suficiente para que te ayude.

Ahora que te he molestado sobre cómo escribir y no escribirme, me gustaría hacerte saber que aprecio plenamente todos los elogios que la guía ha recibido a lo largo de los años. Es una verdadera inyección de moral, ¡y me alegra saber que se está utilizando para el bien! `:-)` ¡Gracias!

1.7 Espejado

Eres más que bienvenido a duplicar este sitio, ya sea pública o privadamente. Si reflejas públicamente el sitio y quieres que lo enlace desde la página principal, escríbeme a `beej@beej.us`.

1.8 Nota para los traductores

Si quieres traducir la guía a otro idioma, escríbeme a `beej@beej.us` y enlazaré a tu traducción desde la página principal. No dude en añadir su nombre e información de contacto a la traducción.

Este documento de Markdown de origen utiliza la codificación UTF-8.

Tenga en cuenta las restricciones de la licencia en la sección Derechos de autor, distribución y legal, a continuación.

Si quieres que me encargue de la traducción, solo tienes que pedirlo. También lo enlazaré si quieres alojarlo; De cualquier manera está bien.

1.9 Derechos de autor, distribución y legal

[6] https://www.tallyhawk.net/WinsockExamples/

[7] http://www.catb.org/~esr/faqs/smart-questions.html

1.10 Dedicación

Gracias a todos los que me han ayudado en el pasado y en el futuro a escribir esta guía. Y gracias a todas las personas que producen el software libre y los paquetes que utilizo para hacer la Guía: GNU, Linux, Slackware, vim, Python, Inkscape, pandoc, muchos otros. Y, por último, un gran agradecimiento a los literalmente miles de ustedes que han escrito con sugerencias de mejoras y palabras de aliento.

Dedico esta guía a algunos de mis mayores héroes e inspiradores en el mundo de las computadoras: Donald Knuth, Bruce Schneier, W. Richard Stevens y The Woz, a mis lectores y a toda la comunidad de software libre y de código abierto.

1.11 Información de publicación

Este libro está escrito en Markdown usando el editor vim en una caja Arch Linux cargada con herramientas GNU. El "arte" de la portada y los diagramas se producen con Inkscape. El Markdown se convierte a HTML y LaTex/PDF mediante Python, Pandoc y XeLaTeX, utilizando fuentes Liberation. La cadena de herramientas está compuesta por software 100% libre y de código abierto.

Capítulo 2

¿Qué es un enchufe?

Escuchas hablar de "enchufes" todo el tiempo, y tal vez te estés preguntando qué son exactamente. Bueno, son esto: una forma de hablar con otros programas usando descriptores de archivo Unix estándar.

¿Qué?

Ok, es posible que hayas escuchado a algún hacker de Unix decir: "¡Dios mío, *todo* en Unix es un archivo!" De lo que esa persona puede haber estado hablando es del hecho de que cuando los programas Unix hacen cualquier tipo de E/S, lo hacen leyendo o escribiendo en un descriptor de archivo. Un descriptor de archivo es simplemente un número entero asociado a un archivo abierto. Pero (y aquí está el truco), ese archivo puede ser una conexión de red, un FIFO, una tubería, un terminal, un archivo real en el disco o casi cualquier otra cosa. ¡Todo en Unix *es* un archivo! Así que cuando quieras comunicarte con otro programa a través de Internet lo harás a través de un descriptor de archivo, será mejor que lo creas.

"¿Dónde puedo obtener este descriptor de archivo para la comunicación de red, Sr. Smarty-Pants?" es probablemente la última pregunta en su mente en este momento, pero voy a responderla de todos modos: Usted hace una llamada a la rutina del `sistema socket()`. Devuelve el descriptor de socket, y usted se comunica a través de él usando las llamadas de socket `especializadas send()` y `recv()` (`man send`, `man recv`).

"¡Pero, oye!", podrías estar exclamando en este momento. "Si es un descriptor de archivo, ¿por qué en nombre de Neptune no puedo usar las llamadas normales `read()` y `write()` para comunicarme a través del socket?" La respuesta corta es: "¡Puedes!" La respuesta más larga es: "Puedes, pero `send()` y `recv()` ofrecen un control mucho mayor sobre tu transmisión de datos".

¿Y ahora qué? ¿Qué te parece esto?: hay todo tipo de enchufes. Hay direcciones de Internet DARPA (Internet Sockets), nombres de ruta en un nodo local (Unix Sockets), direcciones CCITT X.25 (X.25 Sockets que puede ignorar con seguridad) y probablemente muchos otros dependiendo del tipo de Unix que ejecute. Este documento se ocupa solo del primero: los enchufes de Internet.

2.1 Dos tipos de enchufes de Internet

¿Qué es esto? ¿Hay dos tipos de enchufes de Internet? Sí. Pues no. Estoy mintiendo. Hay más, pero no quería asustarte. Aquí solo voy a hablar de dos tipos. Excepto por esta frase, donde te voy a decir que los "Raw Sockets" también son muy poderosos y deberías buscarlos.

Muy bien, ya. ¿Cuáles son los dos tipos? Uno es "Stream Sockets"; el otro es "Datagram Sockets", que en lo sucesivo se denominarán "SOCK_STREAM" y "`SOCK_DGRAM`", respectivamente. Los calcetines de datagramas se denominan a veces "sockets sin conexión". (Aunque pueden ser `connect()`d si realmente quieres. Véase `connect()`, a continuación).

Los sockets de flujo son flujos de comunicación conectados bidireccionalmente confiables. Si imprime dos elementos en el zócalo en el orden "1, 2", llegarán en el orden "1, 2" en el extremo opuesto. Además, estarán libres de errores. Estoy tan seguro, de hecho, de que estarán libres de errores, que simplemente me pondré los dedos en los oídos y cantaré *la la la la la* si alguien intenta afirmar lo contrario.

¿Qué usos usan los sockets de transmisión? Bueno, es posible que hayas oído hablar de las `aplicaciones telnet` o `ssh`, ¿verdad? Utilizan sockets de flujo. Todos los caracteres que escribas deben llegar en el mismo orden en que los escribes, ¿verdad? Además

Los navegadores web utilizan el Protocolo de transferencia de hipertexto (HTTP), que utiliza sockets de flujo para obtener páginas. De hecho, si conectas por telnet a un sitio web en el puerto 80, y escribes "`GET / HTTP/1.0`" y pulsas RETURN dos veces, ¡te devolverá el HTML!

> Si no tiene `telnet` instalado y no desea instalarlo, o si su `telnet` está siendo exigente con la conexión a los clientes, la guía viene con un `programa similar` a telnet llamado `telnot`[a]. Esto debería funcionar bien para todas las necesidades de la guía. (Tenga en cuenta que telnet es en realidad un Protocolo de red especificado[B],
> Y Telnot no implementa este protocolo en absoluto).
>
> [Un]https://beej.us/guide/bgnet/source/examples/telnot.c [bhttps://tools.ietf.org/html/rfc854]

¿Cómo logran los sockets de flujo este alto nivel de calidad de transmisión de datos? Utilizan un protocolo llamado "El Protocolo de Control de Transmisión", también conocido como "TCP" (consulte RFC 793[1] para obtener información extremadamente detallada sobre TCP). TCP se asegura de que sus datos lleguen de forma secuencial y sin errores. Es posible que haya escuchado "TCP" antes como la mejor mitad de "TCP / IP", donde "IP" significa "Protocolo de Internet" (consulte RFC 791[2]). La IP se ocupa principalmente del enrutamiento de Internet y, por lo general, no es responsable de la integridad de los datos.

Fresco. ¿Qué pasa con los sockets de datagramas? ¿Por qué se llaman sin conexión? ¿Cuál es el problema, aquí, de todos modos? ¿Por qué no son fiables? Bueno, aquí hay algunos datos: si envías un datagrama, puede llegar. Puede llegar fuera de servicio. Si llega, los datos dentro del paquete estarán libres de errores.

Los sockets de datagramas también usan IP para el enrutamiento, pero no usan TCP; usan el "Protocolo de datagramas de usuario" o "UDP" (ver RFC 768[3]).

¿Por qué no tienen conexión? Bueno, básicamente, es porque no tienes que mantener una conexión abierta como lo haces con los sockets de transmisión. Sólo tienes que crear un paquete, ponerle un encabezado IP con información de destino y enviarlo. No se necesita conexión. Por lo general, se utilizan cuando una pila TCP no está disponible o cuando unos pocos paquetes caídos aquí y allá no significan el fin del Universo. Ejemplos de aplicaciones: `tftp` (protocolo trivial de transferencia de archivos, un hermano pequeño de FTP), `dhcpcd` (un cliente DHCP), juegos multijugador, transmisión de audio, videoconferencias, etc.

"¡Espera un minuto! `¡TFTP` y `DHCPCD` se utilizan para transferir aplicaciones binarias de un host a otro! ¡Los datos no se pueden perder si espera que la aplicación funcione cuando llegue! ¿Qué tipo de magia oscura es esta?"

Bueno, mi amigo humano, `tftp` y programas similares tienen su propio protocolo además de UDP. Por ejemplo, el protocolo tftp dice que por cada paquete que se envía, el destinatario tiene que devolver un paquete que diga: "¡Lo tengo!" (un paquete "ACK"). Si el remitente del paquete original no recibe respuesta en, digamos, cinco segundos, retransmitirá el paquete hasta que finalmente obtenga un ACK. Este procedimiento de reconocimiento es muy importante cuando se implementan `aplicaciones de SOCK_DGRAM` confiables.

En el caso de aplicaciones poco fiables como juegos, audio o vídeo, simplemente ignora los paquetes perdidos, o tal vez intente compensarlos de forma inteligente. (Los jugadores de Quake conocerán la manifestación de este efecto por el término técnico: *Retraso maldito*. La palabra "maldito", en este caso, representa cualquier expresión extremadamente profana.)

¿Por qué utilizar un protocolo subyacente poco fiable? Dos razones: la velocidad y la velocidad. Es mucho más rápido disparar y olvidar que hacer un seguimiento de lo que ha llegado de forma segura y asegurarse de que está en orden y todo eso. Si está enviando mensajes de chat, TCP es excelente; si estás enviando 40 actualizaciones de posición por segundo de los jugadores en el mundo, tal vez no importe tanto si uno o dos se eliminan, y UDP es una buena opción.

2.2 Tonterías de bajo nivel y teoría de redes

Ya que acabo de mencionar la estratificación de protocolos, es hora de hablar sobre cómo funcionan realmente las redes y mostrar algunos ejemplos de cómo se construyen SOCK_DGRAM paquetes. Prácticamente, probablemente puedas saltarte esta sección. Sin embargo, es un buen antecedente.

¡Hola, niños, es hora de aprender sobre *la encapsulación de datos*! Esto es muy, muy importante. Es tan importante que podrías aprender sobre ello si tomas el curso de redes aquí en Chico State `;-)`. Básicamente, dice

[1]https://tools.ietf.org/html/rfc793 [2]
https://tools.ietf.org/html/rfc791

3https://tools.ietf.org/html/rfc768

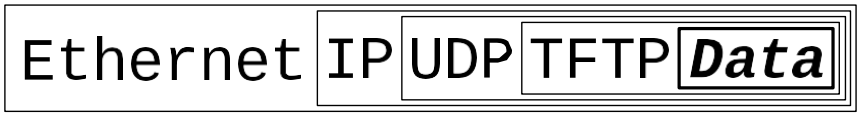

Figura 2.1: Encapsulación de datos.

esto: nace un paquete, el paquete se envuelve ("encapsula") en un encabezado (y rara vez un pie de página) por el primer protocolo (por ejemplo, el protocolo TFTP), luego todo (encabezado TFTP incluido) es encapsulado nuevamente por el siguiente protocolo (por ejemplo, UDP), luego nuevamente por el siguiente (IP), luego nuevamente por el protocolo final en la capa de hardware (física) (digamos, Ethernet).

Cuando otra computadora recibe el paquete, el hardware elimina el encabezado Ethernet, el kernel elimina los encabezados IP y UDP, el programa TFTP elimina el encabezado TFTP y finalmente tiene los datos.

Ahora finalmente puedo hablar sobre el infame modelo de *red en capas* (también conocido como "ISO/OSI"). Este modelo de red describe un sistema de funcionalidad de red que tiene muchas ventajas sobre otros modelos. Por ejemplo, puede escribir programas de sockets que sean exactamente iguales sin importar cómo se transmiten físicamente los datos (serie, Ethernet delgado, AUI, lo que sea) porque los programas de niveles inferiores se encargan de ello por usted. El hardware y la topología de red reales son transparentes para el programador de sockets.

Sin más preámbulos, presentaré las capas del modelo en toda regla. Recuerde esto para los exámenes de la clase de red:

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

La capa física es el hardware (serie, Ethernet, etc.). La capa de aplicación está tan lejos de la capa física como se pueda imaginar: es el lugar donde los usuarios interactúan con la red.

Ahora, este modelo es tan general que probablemente podría usarlo como una guía de reparación de automóviles si realmente quisiera. Un modelo en capas más coherente con Unix podría ser:

- Capa de aplicación (*telnet, ftp, etc.*)
- Capa de transporte de host a host (*TCP, UDP*)
- Capa de Internet (*IP y enrutamiento*)
- Capa de acceso a la red (*Ethernet, wi-fi o lo que sea*)

En este momento, probablemente pueda ver cómo estas capas corresponden a la encapsulación de los datos originales.

¿Ves cuánto trabajo hay en la construcción de un paquete simple? ¡Cielos! ¡Y tienes que escribir los encabezados de los paquetes tú mismo usando "`gato`"! Es broma. Todo lo que tienes que hacer para los sockets de flujo es `enviar ()` los datos. Todo lo que tienes que hacer para los sockets de datagramas es encapsular el paquete en el método que elijas y `enviarlo a` enviar. El kernel construye la capa de transporte y la capa de Internet y el hardware hace la capa de acceso a la red. Ah, la tecnología moderna.

Así termina nuestra breve incursión en la teoría de redes. Oh, sí, olvidé decirte todo lo que quería decir sobre el enrutamiento: ¡nada! Así es, no voy a hablar de eso en absoluto. El router quita el paquete a la cabecera IP, consulta su tabla de enrutamiento, *bla bla bla*. Echa un vistazo a la IP RFC[4] si realmente te importa. Si nunca lo aprendes, bueno, vivirás.

[4]https://tools.ietf.org/html/rfc791

Capítulo 3

Direcciones IP, `estructuras y limpieza` de datos

Esta es la parte del juego en la que podemos hablar de código para variar.

Pero primero, ¡analicemos más sobre el no código! ¡Yay! Primero quiero hablar sobre las direcciones IP y los puertos por un momento, así que lo hemos resuelto. Luego hablaremos sobre cómo la API de sockets almacena y manipula las direcciones IP y otros datos.

3.1 Direcciones IP, versiones 4 y 6

En los viejos tiempos, cuando Ben Kenobi todavía se llamaba Obi Wan Kenobi, había un maravilloso sistema de enrutamiento de red llamado El Protocolo de Internet Versión 4, también llamado IPv4. Tenía direcciones compuestas por cuatro bytes (también conocido como cuatro "octetos"), y se escribía comúnmente en forma de "puntos y números", así: `192.0.2.111`.

Probablemente lo hayas visto por ahí.

De hecho, en el momento de escribir este artículo, prácticamente todos los sitios de Internet utilizan IPv4.

Todos, incluido Obi Wan, estaban felices. Las cosas iban muy bien, hasta que un detractor llamado Vint Cerf advirtió a todo el mundo que estábamos a punto de quedarnos sin direcciones IPv4.

(Además de advertir a todo el mundo del próximo apocalipsis de IPv4 de fatalidad y pesimismo, Vint Cerf[1] también es conocido por ser el padre de Internet. Así que realmente no estoy en posición de cuestionar su juicio).

¿Te has quedado sin direcciones? ¿Cómo puede ser esto? Quiero decir, hay como miles de millones de direcciones IP en una dirección IPv4 de 32 bits. ¿Realmente tenemos miles de millones de computadoras por ahí?

Sí.

Además, al principio, cuando solo había unas pocas computadoras y todos pensaban que mil millones era un número imposiblemente grande, a algunas grandes organizaciones se les asignaron generosamente millones de direcciones IP para su propio uso. (Como Xerox, MIT, Ford, HP, IBM, GE, AT&T y alguna pequeña empresa llamada Apple, por nombrar algunas).

De hecho, si no fuera por varias medidas provisionales, se habríamos agotado hace mucho tiempo.

Pero ahora vivimos en una era en la que estamos hablando de que cada ser humano tiene una dirección IP, cada computadora, cada calculadora, cada teléfono, cada parquímetro y (por qué no) también cada cachorro de perro.

Y así nació IPv6. Dado que Vint Cerf es probablemente inmortal (incluso si su forma física pasara, Dios no lo quiera, probablemente ya exista como una especie de programa ELIZA 2 hiperinteligente en las profundidades de Internet), nadie quiere tener que escucharlo decir de nuevo "Te lo dije" si no tenemos suficientes direcciones en la próxima versión del Protocolo de Internet.

[1] https://en.wikipedia.org/wiki/Vint_Cerf
[2] https://en.wikipedia.org/wiki/ELIZA

¿Qué te sugiere esto?

Que necesitamos *muchas* más direcciones. ¡Que no solo necesitamos el doble de direcciones, ni mil millones de veces más, ni mil billones de billones de veces más, sino *79 MILLONES DE BILLONES DE BILLONES DE VECES tantas direcciones posibles!* ¡Eso los mostrará!

Estás diciendo: "Beej, ¿es eso cierto? Tengo todas las razones para no creer en los grandes números". Bueno, la diferencia entre 32 bits y 128 bits puede no parecer mucha; Son solo 96 bits más, ¿verdad? Pero recuerde, aquí estamos hablando de potencias: 32 bits representan unos 4 mil millones de números (232), mientras que 128 bits representan alrededor de 340 billones de billones de billones de números (de verdad, 2128). Eso es como un millón de Internets IPv4 por *cada estrella en el Universo*.

Olvídese también de este aspecto de puntos y números de IPv4; Ahora tenemos una representación hexadecimal, con cada fragmento de dos bytes separado por dos puntos, así:

```
2001:0db8:c9d2:aee5:73e3:934a:a5ae:9551
```

¡Eso no es todo! Muchas veces, tendrás una dirección IP con muchos ceros y puedes comprimirlos entre dos puntos. Y puede omitir ceros a la izquierda para cada par de bytes. Por ejemplo, cada uno de estos pares de direcciones son equivalentes:

```
2001:0db8:c9d2:0012:0000:0000:0000:0051
2001:db8:c9d2:12::51

2001:0db8:ab00:0000:0000:0000:0000:0000
2001:db8:ab00::

0000:0000:0000:0000:0000:0000:0000:0001
::1
```

La dirección `::1` es la *dirección de bucle invertido*. Siempre significa "esta máquina en la que estoy funcionando ahora". En IPv4, la dirección de bucle invertido es `127.0.0.1`.

Por último, hay un modo de compatibilidad con IPv4 para las direcciones IPv6 con las que te puedes encontrar. Si desea, por ejemplo, representar la dirección IPv4 `192.0.2.33` como una dirección IPv6, utilice la siguiente notación: "`::ffff:192.0.2.33`".

Estamos hablando de diversión seria.

De hecho, es tan divertido que los creadores de IPv6 han cortado de manera bastante arrogante billones y billones de direcciones para uso reservado, pero tenemos tantas, francamente, ¿quién está contando ya? Hay mucho de sobra para cada hombre, mujer, niño, cachorro y parquímetro en cada planeta de la galaxia. Y créeme, todos los planetas de la galaxia tienen parquímetros. Sabes que es verdad.

3.1.1 Subredes

Por razones organizativas, a veces es conveniente declarar que "esta primera parte de esta dirección IP a través de este bit es la *parte de red* de la dirección IP, y el resto es la parte del *host*.

Por ejemplo, con IPv4, podría tener `192.0.2.12`, y podríamos decir que los primeros tres bytes son la red y el último byte es el host. O, dicho de otra manera, estamos hablando del host `12` en la red `192.0.2.0` (vea cómo ponemos a cero el byte que era el host).

¡Y ahora para más información desactualizada! ¿Listo? En la antigüedad, había "clases" de subredes, donde el primer uno, dos o tres bytes de la dirección era la parte de la red. Si tuvieras la suerte de tener un byte para la red y tres para el host, podrías tener 24 bits de hosts en tu red (16 millones más o menos). Se trataba de una red de "Clase A". En el extremo opuesto había una "Clase C", con tres bytes de red y un byte de host (256 hosts, menos un par que estaban reservados).

Entonces, como pueden ver, solo había unas pocas Clase A, una gran pila de Clase C y algunas Clase B en el medio.

La parte de red de la dirección IP se describe mediante algo llamado máscara de *red*, que se utiliza bit a bit Y con la dirección IP para obtener el número de red de ella. La máscara de red suele tener un aspecto similar a `255.255.255.0`. (Por ejemplo, con esa máscara de red, si su IP es `192.0.2.12`, entonces su red es `192.0.2.12` Y `255.255.255.0` que da `192.0.2.0`.)

Desafortunadamente, resultó que esto no era lo suficientemente detallado para las necesidades eventuales de Internet; nos estábamos quedando sin redes de Clase C con bastante rapidez, y definitivamente estábamos fuera de Clase A, así que ni siquiera se moleste en preguntar. Para remediar esto, los poderes fácticos permitieron que la máscara de red fuera un número arbitrario de bits, no solo 8, 16 o 24. Por lo tanto, es posible que tenga una máscara de red de, digamos, `255.255.255.252`, que son 30 bits de red, y 2 bits de host que permiten cuatro hosts en la red. (Tenga en cuenta que la máscara de red es *SIEMPRE* un montón de 1 bit seguido de un montón de 0 bits).

Pero es un poco difícil de manejar usar una gran cadena de números como `255.192.0.0` como máscara de red. En primer lugar, la gente no tiene una idea intuitiva de cuántos bits son y, en segundo lugar, realmente no es compacto. Así que llegó el Nuevo Estilo, y es mucho más bonito. Simplemente coloque una barra diagonal después de la dirección IP y luego siga por el número de bits de red en decimales. Así: `192.0.2.12/30`.

O, para IPv6, algo como esto: `2001:db8::/32` o `2001:db8:5413:4028::9db9/64`.

3.1.2 Números de puerto

Si recuerdan, les presenté anteriormente el modelo de red en capas que tenía la capa de Internet (IP) separada de la capa de transporte de host a host (TCP y UDP). Ponte al día con eso antes del siguiente párrafo.

Resulta que además de una dirección IP (utilizada por la capa IP), hay otra dirección que es utilizada por TCP (sockets de flujo) y, casualmente, por UDP (sockets de datagramas). Es el número de *puerto*. Es un número de 16 bits que es como la dirección local de la conexión.

Piense en la dirección IP como la dirección postal de un hotel, y el número de puerto como el número de habitación. Esa es una analogía decente; tal vez más adelante se me ocurra uno que involucre a la industria automotriz.

Supongamos que desea tener una computadora que maneje el correo entrante Y los servicios web, ¿cómo diferencia entre los dos en una computadora con una sola dirección IP?

Bueno, los diferentes servicios en Internet tienen diferentes números de puerto conocidos. Puede verlos todos en la Gran Lista de Puertos IANA[3] o, si está en una caja Unix, en su archivo `/etc/services`. HTTP (la web) es el puerto 80, telnet es el puerto 23, SMTP es el puerto 25, el juego DOOM[4] usó el puerto 666, etc. y así sucesivamente. Los puertos inferiores a 1024 a menudo se consideran especiales y, por lo general, requieren privilegios especiales del sistema operativo para su uso.

¡Y eso es todo!

3.2 Orden de bytes

¡Por orden del reino! Habrá dos ordenaciones de bytes, que en lo sucesivo se denominarán Cojo y Magnífico.

Bromeo, pero uno realmente es mejor que el otro. `:-)`

Realmente no hay una manera fácil de decir esto, así que lo diré claramente: es posible que su computadora haya estado almacenando bytes en orden inverso a sus espaldas. ¡Lo sé! Nadie quería tener que decírtelo.

La cuestión es que todo el mundo en el mundo de Internet ha estado de acuerdo en que si quieres representar el número hexadecimal de dos bytes, digamos `b34f`, lo almacenarás en dos bytes secuenciales `b3` seguido de `4f`. Tiene sentido y, como Wilford Brimley[5] te diría, es lo correcto. Este número, almacenado primero con el extremo grande, se llama *Big-Endian*.

Desafortunadamente, algunas computadoras dispersas aquí y allá por todo el mundo, es decir, cualquier cosa con un procesador Intel o compatible con Intel, almacenan los bytes invertidos, por lo que `b34f` se almacenaría en la memoria como los bytes secuenciales `4f` seguidos de `b3`. Este método de almacenamiento se llama *Little-Endian*.

[3] https://www.iana.org/assignments/port-numbers

[4] https://en.wikipedia.org/wiki/Doom_(1993_video_game) [5] https://en.wikipedia.org/wiki/Wilford_Brimley

Pero espera, ¡aún no he terminado con la terminología! El *Big-Endian más sensato* también se llama *Orden de bytes de red* porque ese es el orden que nos gusta a los tipos de red.

Su computadora almacena los números en *orden de bytes del host*. Si se trata de un Intel 80x86, el orden de bytes del host es Little-Endian. Si se trata de un Motorola 68k, el orden de bytes del host es Big-Endian. Si se trata de un PowerPC, el orden de bytes del host es... Bueno, ¡depende!

Muchas veces, cuando está creando paquetes o completando estructuras de datos, deberá asegurarse de que sus números de dos y cuatro bytes estén en orden de bytes de red. Pero, ¿cómo puede hacer esto si no conoce el orden de bytes del host nativo?

¡Buenas noticias! Solo tienes que asumir que el orden de bytes del host no es el correcto, y siempre ejecutas el valor a través de una función para establecerlo en el orden de bytes de la red. La función hará la conversión mágica si es necesario, y de esta manera su código es portátil a máquinas de diferente endianidad.

Muy bien. Hay dos tipos de números que se pueden convertir: `cortos` (dos bytes) y `largos` (cuatro bytes). Estas funciones también funcionan para las `variaciones sin signo`. Supongamos que desea convertir un `corto` de Orden de bytes de host a Orden de bytes de red. Comience con "h" para "host", siga con "a", luego "n" para "red" y "s" para "corto": h-a-n-s, o `htons()` (léase: "Host to Network Short").

Es casi demasiado fácil...

Puedes usar todas las combinaciones de "n", "h", "s" y "l" que quieras, sin contar las realmente estúpidas. Por ejemplo, NO hay una `función stolh()` ("Host corto a largo"), al menos no en esta fiesta. Pero sí los hay:

Función	Descripción
`htons()`	`anfitrión de n`etwork `corto`
`htonl()`	`host a n`etwork `red`
`ntohs()`	`larga a h`ost `red`
`ntohl()`	`corta a h`ost `larga`

Básicamente, querrá convertir los números a Orden de bytes de red antes de que salgan en el cable, y convertirlos en Orden de bytes del host a medida que ingresan por cable.

No conozco una variante de 64 bits, lo siento. Y si quieres hacer coma flotante, echa un vistazo a la sección sobre serialización, muy abajo.

Suponga que los números de este documento están en orden de bytes de host a menos que diga lo contrario.

3.3 `Estructuras`

Bueno, finalmente estamos aquí. Es hora de hablar de programación. En esta sección, cubriré varios tipos de datos utilizados por la interfaz de sockets, ya que algunos de ellos son realmente difíciles de descifrar.

Primero el fácil: un descriptor de socket. Un descriptor de socket es del siguiente tipo:

```
Int
```

Solo un `int regular`.

Las cosas se ponen raras a partir de aquí, así que léelo y ten paciencia conmigo.

Mi primera estructura™: `struct addrinfo`. Esta estructura es una invención más reciente, y se utiliza para preparar las estructuras de direcciones de socket para su uso posterior. También se usa en búsquedas de nombres de host y búsquedas de nombres de servicio. Eso tendrá más sentido más adelante cuando lleguemos al uso real, pero sepa por ahora que es una de las primeras cosas que llamará al establecer una conexión.

```
struct addrinfo {
    tú                ai_flags;      AI_PASSIVE, AI_CANONNAME, etc.
    eres              ai_family;     AF_INET, AF_INET6 AF_UNSPEC
    tú, tú            ai_socktype;   // SOCK_STREAM, SOCK_DGRAM
    eres              ai_protocol;   use 0 para "cualquiera"
    size_t            ai_addrlen;    Tamaño de ai_addr en bytes
    struct sockaddr *ai_addr;        struct sockaddr_in o _in6
    carbonizar       *ai_canonname; Nombre de host canónico

    struct addrinfo *ai_next;        Lista enlazada, Nodo
};
```

Cargará un poco esta estructura y, a continuación, llamará a `getaddrinfo()`. Volverá un puntero a una nueva lista enlazada de estas estructuras completada con todas las golosinas que necesitas.

Puede obligarlo a usar IPv4 o IPv6 en el campo `ai_family`, o dejarlo como `AF_UNSPEC` para usar lo que sea. Esto es genial porque su código puede ser independiente de la versión de IP.

Tenga en cuenta que esta es una lista enlazada: `ai_next` puntos en el siguiente elemento, podría haber varios resultados para que elija. Usaría el primer resultado que funcionó, pero es posible que tenga diferentes necesidades comerciales; ¡No lo sé todo, hombre!

Verás que el campo `ai_addr` en struct `addrinfo` es un puntero a un `struct sockaddr`. Aquí es donde comenzamos a entrar en los detalles esenciales de lo que hay dentro de una estructura de direcciones IP.

Es posible que normalmente no necesites escribir en estas estructuras; a menudo, todo `lo que necesitas es una llamada a getaddrinfo()` para que rellene tu `struct addrinfo` por ti. Sin embargo, tendrá que mirar dentro de estas `estructuras para obtener los valores, por lo que los presento aquí.`

(Además, todo el código escrito antes `de que se inventara struct addrinfo`, lo empaquetamos todo a mano, por lo que verá una gran cantidad de código IPv4 en la naturaleza que hace exactamente eso. Ya sabes, en versiones antiguas de esta guía y así sucesivamente).

Alguno `Estructuras` son IPv4, algunos son IPv6 y otros son ambos. Tomaré notas de cuáles

son qué. De todos modos, el `struct sockaddr` Contiene información de dirección de socket

```
struct sockaddr {
    Corto sin firmar   sa_family;    Dirección: Familia, AF_xxx
    carbonizar         sa_data[14]; 14 bytes de dirección de protocolo
};
```

para muchos tipos de sockets.

`sa_family` puede ser una variedad de cosas, pero será `AF_INET` (IPv4) o `AF_INET6` (IPv6) para todo lo que hacemos en este documento. `sa_data` contiene una dirección de destino y un número de puerto para el socket. Esto es bastante difícil de manejar, ya que no desea empaquetar tediosamente la dirección en el `sa_data` a mano.

Para lidiar con `struct sockaddr`, los programadores crearon una estructura paralela: `struct sockaddr_in` ("in" para "Internet") para ser utilizado con IPv4.

Y *esto es lo importante* : un puntero a un `struct sockaddr_in` se puede convertir en un puntero a un struct `sockaddr` y viceversa. Entonces, aunque `connect()` quiera un `struct sockaddr*`, aún puedes usar un `struct sockaddr_in` y lanzarlo en el último minuto.

```
(Solo IPv4: consulte el sockaddr_in6 de
estructuras para IPv6)

struct sockaddr_in {                  Diríjase a la familia,
    Corto intsin_family ; unsigned    AF_INET
    struct in_addr      sin_addr;     Número de puerto
    char sin signo      sin_zero[8];  Mismo tamaño que struct
};
```

Esta estructura facilita la referencia a elementos de la dirección del socket. Tenga en cuenta que `sin_zero` (que se incluye para rellenar la estructura a la longitud de un `calcetín de estructura`) debe establecerse en ceros con la función `memset()`. Además, observe que `sin_family` corresponde a `sa_family` en un `calcetín de estructura` y debe establecerse en "`AF_INET`". Finalmente, el `sin_port` debe estar en *orden de bytes de red* (usando `htons()`!)

¡Profundicemos más! Verá que el `campo sin_addr` es una `estructura in_addr`. ¿Qué es esa cosa? Bueno, no quiero ser demasiado dramático, pero es una de las uniones más aterradoras de todos los tiempos:

```
(Solo IPv4: consulte el in6_addr de estructuras para IPv6)

Dirección de Internet (una estructura por razones históricas)
struct in_addr {
    uint32_t s_addr;  Es un INT de 32 bits (4 bytes)
};
```

¡Whoa! Bueno, *solía* ser un sindicato, pero ahora esos días parecen haberse ido. Buen viaje. Por lo tanto, si ha declarado `que ina` es de tipo `struct sockaddr_in`, entonces `ina.sin_addr.s_addr` hace referencia a la dirección IP de 4 bytes (en orden de bytes de red). Tenga en cuenta que incluso si su sistema todavía usa la unión horrible para la `estructura in_addr`, aún puede hacer referencia a la dirección IP de 4 bytes exactamente de la misma manera que lo hice anteriormente (esto debido a `#defines).`

¿Qué pasa con IPv6? También existen `estructuras` similares para él:

```
(Solo IPv6: consulte struct sockaddr_in y struct in_addr para IPv4)

struct sockaddr_in6 {
    u_int16_t       sin6_family;    Dirección: familia, AF_INET6
    u_int16_t       sin6_port;      puerto, orden de bytes de
    u_int32_t       sin6_flowinfo;  Información de flujo
    Estructura in6_addr sin6_addr;  Dirección IPv6
    u_int32_t       sin6_scope_id; Id. de ámbito
};

 struct in6_addr {
   char sin firmar   s6_addr[16];    Dirección IPv6
};
```

Tenga en cuenta que IPv6 tiene una dirección IPv6 y un número de puerto, al igual que IPv4 tiene una dirección IPv4 y un número de puerto.

También tenga en cuenta que no voy a hablar sobre la información de flujo IPv6 o los campos de ID de alcance por el momento ... Esta es solo una guía de inicio. `:-)`

Por último, pero no menos importante, aquí hay otra estructura simple, `struct sockaddr_storage` que está diseñada para ser lo suficientemente grande como para contener estructuras IPv4 e IPv6. Verás, para algunas llamadas, a veces no sabes de antemano si va a llenar tu `struct sockaddr` con una dirección IPv4 o IPv6. Así que se pasa esta estructura paralela, muy similar a `struct sockaddr` excepto que es más grande, y luego se convierte al tipo que necesita:

```
struct sockaddr_storage {
    sa_family_t ss_family;      Dirección Familia

    Todo esto es relleno, específico de la implementación,
    ignóralo:
    carbonizar __ss_pad1[_SS_PAD1SIZE];
    int64_t    __ss_align;
};
```

Lo importante es que puede ver la familia de direcciones en el campo `ss_family` (verifique esto para ver si es `AF_INET` o `AF_INET6` (para IPv4 o IPv6). A continuación, puede convertirlo en una `estructura sockaddr_in` o

`struct sockaddr_in6` si quieres.

3.4 Direcciones IP, Part Deux

Afortunadamente para ti, hay un montón de funciones que te permiten manipular las direcciones IP. No es necesario descifrarlos a mano y meterlos a mano con el operador `<<`.

En primer lugar, supongamos que tiene una `estructura sockaddr_in ina` y tiene una dirección IP "`10.12.110.57`" o "`2001:db8:63b3:1::3490`" que desea almacenar en ella. La función que desea utilizar, `inet_pton()`, convierte una dirección IP en notación de números y puntos en un in_addr de `estructura` o en un `in6_addr de estructura`, dependiendo de si especifica `AF_INET` o `AF_INET6`. ("`pton`" significa "Presentación a la red": puede llamarlo "imprimible a la red" si eso es más fácil de recordar). La conversión se puede realizar de la siguiente manera para IPv4 e IPv6:

```
Estructura sockaddr_in su; IPv4
estructura sockaddr_in6 sa6;  IPv6

inet_pton(AF_INET, "10.12.110.57", &(sa.sin_addr));
inet_pton(AF_INET6, "2001:db8:63b3:1:3490", &(sa6.sin6_addr);
```

(Nota rápida: la antigua forma de hacer las cosas usaba una función llamada `inet_addr()` u otra función llamada
`inet_aton()`; estos ahora están obsoletos y no funcionan con IPv6).

Ahora, el fragmento de código anterior no es muy robusto porque no hay verificación de errores. Verás, `inet_pton()` devuelve `-1` en caso de error, o 0 si la dirección está desordenada. ¡Así que verifique que el resultado sea mayor que 0 antes de usar!

Muy bien, ahora puede convertir direcciones IP de cadena a sus representaciones binarias. ¿Y al revés? ¿Qué pasa si tienes un in_addr de `estructura` y quieres imprimirlo en notación de números y puntos? (O un `struct in6_addr` que desee en, eh, notación "hexadecimal y dos puntos"). En este caso, querrás usar la función `inet_ntop()` ("ntop" significa "de red a presentación", puedes llamarlo "de red a imprimible" si eso es más fácil de recordar), así:

```
IPv4:

char IP4[INET_ADDRSTRLEN];  espacio para contener la cadena IPv4
Estructura sockaddr_in su;   fingir que esto está cargado de algo

inet_ntop(AF_INET, &(sa.sin_addr), IP4, INET_ADDRSTRLEN);

printf("La dirección IPv4 es: %s\n", ip4);


IPv6:

char IP6[INET6_ADDRSTRLEN];  espacio para contener la cadena IPv6
Estructura sockaddr_in6 sa6; fingir que esto está cargado de algo

inet_ntop(AF_INET6, &(sa6.sin6_addr), IP6, INET6_ADDRSTRLEN);

printf("La dirección es: %s\n", ip6);
```

Al llamarlo, pasará el tipo de dirección (IPv4 o IPv6), la dirección, un puntero a una cadena para contener el resultado y la longitud máxima de esa cadena. (Dos macros contienen convenientemente el tamaño de la cadena que necesitará para contener la dirección IPv4 o IPv6 más grande: `INET_ADDRSTRLEN` y `INET6_ADDRSTRLEN`.)

(Otra nota rápida para mencionar una vez más la antigua forma de hacer las cosas: la función histórica de hacer esta conversión se llamaba `inet_ntoa()`. También está obsoleto y no funcionará con IPv6).

Por último, estas funciones solo funcionan con direcciones IP numéricas, no harán ninguna búsqueda de DNS del servidor de nombres en un nombre de host, como "`www.example.com`". Usarás `getaddrinfo()` para hacer eso, como verás más adelante.

3.4.1 Redes privadas (o desconectadas)

Muchos lugares tienen un firewall que oculta la red del resto del mundo para su propia protección. Y a menudo, el firewall traduce las direcciones IP "internas" a direcciones IP "externas" (que todos los demás en el mundo conocen) mediante un proceso llamado *traducción de direcciones de red* o NAT.

¿Ya te estás poniendo nervioso? —¿A dónde va con todas estas cosas raras?

Bueno, relájate y cómprate una bebida sin alcohol (o alcohólica), porque como principiante, ni siquiera tienes que preocuparte por NAT, ya que está hecho para ti de manera transparente. Pero quería hablar sobre la red detrás del firewall en caso de que comenzara a confundirse con los números de red que estaba viendo.

Por ejemplo, tengo un cortafuegos en casa. Tengo dos direcciones IPv4 estáticas asignadas a mí por la empresa DSL y, sin embargo, tengo siete computadoras en la red. ¿Cómo es esto posible? Dos ordenadores no pueden compartir la misma dirección IP, o de lo contrario los datos no sabrían a cuál ir.

La respuesta es: no comparten las mismas direcciones IP. Están en una red privada con 24 millones de direcciones IP asignadas. Todos son solo para mí. Bueno, todo para mí en lo que a nadie más se refiere. Esto es lo que está pasando:

Si inicio sesión en una computadora remota, me dice que he iniciado sesión desde 192.0.2.33, que es la dirección IP pública que mi ISP me ha proporcionado. Pero si le pregunto a mi computadora local cuál es su dirección IP, dice 10.0.0.5. ¿Quién traduce la dirección IP de uno a otro? Así es, ¡el cortafuegos! ¡Está haciendo NAT!

`10.x.x.x` es una de las pocas redes reservadas que solo se deben usar en redes completamente desconectadas o en redes que están detrás de firewalls. Los detalles de los números de red privada que están disponibles para su uso se describen en RFC 1918[6], pero algunos de los más comunes son `10.x.x.x` y `192.168.x.x`, donde `x` es 0-255, generalmente. Menos común es `172.y.x.x`, donde `y` va entre 16 y 31.

Las redes detrás de un firewall NATing no *necesitan* estar en una de estas redes reservadas, pero comúnmente lo están.

(¡Dato curioso! Mi dirección IP externa no es realmente `192.0.2.33`. La red `192.0.2.x` está reservada para direcciones IP "reales" ficticias que se utilizarán en la documentación, ¡como esta guía! ¡Guau!)

IPv6 también tiene redes privadas, en cierto sentido. Comenzarán con `fdXX:` (o tal vez en el futuro `fcXX:`), según RFC 4193[7]. Sin embargo, NAT e IPv6 generalmente no se mezclan (a menos que esté haciendo lo de la puerta de enlace de IPv6 a IPv4, que está más allá del alcance de este documento), en teoría tendrá tantas direcciones a su disposición que ya no necesitará usar NAT. Pero si desea asignar direcciones para usted en una red que no se enrutará hacia el exterior, esta es la forma de hacerlo.

[6]https://tools.ietf.org/html/rfc1918
[7]https://tools.ietf.org/html/rfc4193

Capítulo 4

Salto de IPv4 a IPv6

¡Pero solo quiero saber qué cambiar en mi código para que funcione con IPv6! ¡Cuéntame ahora! ¡De acuerdo! ¡De acuerdo!

Casi todo lo que hay aquí es algo que he repasado más arriba, pero es la versión corta para los impacientes. (Por supuesto, hay más que esto, pero esto es lo que se aplica a la guía).

1. En primer lugar, intente usar `getaddrinfo()` para obtener toda la información de struct `sockaddr`, en lugar de empaquetar las estructuras a mano. Esto lo mantendrá independiente de la versión de IP y eliminará muchos de los pasos posteriores.
2. En cualquier lugar en el que descubras que estás codificando cualquier cosa relacionada con la versión de IP, intenta envolverlo en una función auxiliar.
3. Cambie `AF_INET` a `AF_INET6`.
4. Cambie `PF_INET` a `PF_INET6`.
5. Cambie `INADDR_ANY` las tareas a in6addr_any las tareas, que son ligeramente diferentes:

   ```
   struct sockaddr_in sa;
   struct sockaddr_in6 sa6;

   S.A.sin_addr.s_addr = INADDR_ANY;  usar mi dirección IPv4
   sa6.sin6_addr = in6addr_any;  usar mi dirección IPv6
   ```

   Además, el valor `IN6ADDR_ANY_INIT` se puede usar como inicializador cuando se declara el `in6_addr struct`, así:

   ```
   estructura in6_addr ia6 = IN6ADDR_ANY_INIT;
   ```

6. En lugar de `struct sockaddr_in` use `struct sockaddr_in6`, asegúrese de agregar "6" a los campos según corresponda (consulte `structs`, arriba). No hay campo `sin6_zero`.
7. En lugar de `struct in_addr` use `struct in6_addr`, asegúrese de agregar "6" a los campos según corresponda (consulte `structs`, arriba).
8. En lugar de `inet_aton()` o `inet_addr()`, usa `inet_pton()`.
9. En lugar de `inet_ntoa()`, use `inet_ntop()`.
10. En lugar de `gethostbyname()`, usa la versión superior `getaddrinfo()`.
11. En lugar de `gethostbyaddr()`, usa el superior `getnameinfo()` (aunque `gethostbyaddr()` aún puede funcionar con IPv6).
12. `INADDR_BROADCAST` ya no funciona. En su lugar, utilice la multidifusión IPv6.

Et voilà!

Capítulo 5

Llamadas al sistema o caída

Esta es la sección en la que entramos en las llamadas al sistema (y otras llamadas a la biblioteca) que le permiten acceder a la funcionalidad de red de una máquina Unix, o cualquier máquina que admita la API de sockets (BSD, Windows, Linux, Mac, lo que sea). Cuando llamas a una de estas funciones, el kernel se hace cargo y hace todo el trabajo por ti de forma automágica.

El lugar donde la mayoría de la gente se queda atascada aquí es en qué orden llamar a estas cosas. En eso, las páginas de manual no sirven para nada, como probablemente hayas descubierto. Bueno, para ayudar con esa terrible situación, he tratado de diseñar las llamadas al sistema en las siguientes secciones *exactamente* (aproximadamente) en el mismo orden en que necesitará llamarlas en sus programas.

Eso, junto con algunos fragmentos de código de muestra aquí y allá, algo de leche y galletas (que me temo que tendrá que suministrarse usted mismo), y algunas agallas y coraje, ¡y estará transmitiendo datos por Internet como el hijo de Jon Postel!

(Tenga en cuenta que, por razones de brevedad, muchos fragmentos de código a continuación no incluyen la verificación de errores necesaria. Y muy comúnmente asumen que el resultado de las llamadas a `getaddrinfo()` tiene éxito y devuelve una entrada válida en la lista enlazada. Sin embargo, ambas situaciones se abordan correctamente en los programas independientes, así que utilícelos como modelo).

5.1 `getaddrinfo()`—¡Prepárate para el lanzamiento!

Este es un verdadero caballo de batalla de una función con muchas opciones, pero el uso es en realidad bastante simple. Ayuda a configurar las `estructuras que necesita más adelante.`

Un poco de historia: solía ser que usabas una función llamada `gethostbyname()` para hacer búsquedas de DNS. A continuación, cargaría esa información a mano en un `sockaddr_in de estructura` y la usaría en sus llamadas.

Esto ya no es necesario, afortunadamente. (¡Tampoco es deseable, si desea escribir código que funcione tanto para IPv4 como para IPv6!) En estos tiempos modernos, ahora tienes la función `getaddrinfo()` que hace todo tipo de cosas buenas por ti, incluidas las búsquedas de DNS y nombres de servicio, ¡y además completa las `estructuras` que necesitas!

¡Echemos un vistazo!

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

int getaddrinfo(const char *nodo,  por ejemplo, "www.example.com" o
                IP  const  char  *service,   por  ejemplo,  "http"  o
                número de puerto const struct addrinfo *hints,
                struct addrinfo **res);
```

Le da a esta función tres parámetros de entrada, y le da un puntero a una lista vinculada, `res`, de resultados.

El parámetro `de nodo` es el nombre de host al que se va a conectar o una dirección IP.

El siguiente es el parámetro `service`, que puede ser un número de puerto, como "80", o el nombre de un servicio en particular (que se encuentra en la Lista de Puertos IANA[1] o en el `archivo /etc/services` en su máquina Unix) como "http" o "ftp" o "telnet" o "smtp" o lo que sea.

Por último, el `parámetro hints` apunta a un `struct addrinfo` que ya ha rellenado con información relevante.

Este es un ejemplo de llamada si es un servidor que desea escuchar en la dirección IP de su host, el puerto 3490. Tenga en cuenta que esto en realidad no realiza ninguna escucha o configuración de red; simplemente establece estructuras que usaremos más adelante:

```
estado int;
struct addrinfo sugerencias;
struct addrinfo *servinfo;  señalará los resultados

memset(&Consejos, 0, tamañode Consejos); Asegúrese de que la
estructura esté vacía Consejos.ai_family = AF_UNSPEC;   no me
importa IPv4 o IPv6 Consejos.ai_socktype = SOCK_STREAM; Sockets de
flujo TCP Consejos.ai_flags = AI_PASSIVE;rellene mi IP por mí

if ((status = getaddrinfo(NULL, "3490", &hints, & servinfo))!= 0) {
    fprintf(stderr, "gai error: %s\n", gai_strerror(status);
    salida(1);
}

servinfo ahora apunta a una lista enlazada de 1 o más
struct addrinfos

// ... hacer todo hasta que ya no necesite servinfo ....

freeaddrinfo(servinfo);  Libera la lista enlazada
```

Observe que establezco el `ai_family` en `AF_UNSPEC`, diciendo que no me importa si usamos IPv4 o IPv6. Puede configurarlo en `AF_INET` o `AF_INET6` si desea uno u otro específicamente.

Además, verás la bandera `AI_PASSIVE` allí; esto le dice a `getaddrinfo()` que asigne la dirección de mi host local a las estructuras de socket. Esto es bueno porque entonces no tienes que codificarlo. (O puede poner una dirección específica como primer parámetro para `getaddrinfo()` donde actualmente tengo `NULL,` allí arriba.)

Entonces hacemos la llamada. Si hay un error (`getaddrinfo()` devuelve un valor distinto de cero), podemos imprimirlo usando la función `gai_strerror(),` como ves. Sin embargo, si todo funciona correctamente, `servinfo` apuntará a una lista enlazada de `struct addrinfos,` cada uno de los cuales contiene un `struct sockaddr` de algún tipo que podemos usar más adelante. ¡Ingenioso!

Finalmente, cuando finalmente hayamos terminado con la lista enlazada que `getaddrinfo()` tan amablemente nos asignó, podemos (y debemos) liberarla toda con una llamada a `freeaddrinfo().`

A continuación, se muestra un ejemplo de llamada si es un cliente que desea conectarse a un servidor en particular, por ejemplo, el puerto "www.example.net" 3490. De nuevo, esto no se conecta realmente, pero establece las estructuras que usaremos más adelante:

```
estado int;
struct addrinfo sugerencias;
struct addrinfo *servinfo;  señalará los resultados

memset(&Consejos, 0, tamañode Consejos); Asegúrese de que la
estructura esté vacía Consejos.ai_family = AF_UNSPEC;   no me
importa IPv4 o IPv6 Consejos.ai_socktype = SOCK_STREAM; Sockets de
flujo TCP
```

[1] https://www.iana.org/assignments/port-numbers

```
Prepárate para conectarte
status = getaddrinfo("www.example.net", "3490", & sugerencias, & servinfo);

servinfo ahora apunta a una lista enlazada de 1 o más
struct addrinfos

etcetera.
```

Sigo diciendo que `servinfo` es una lista enlazada con todo tipo de información de direcciones. Escribamos un programa de demostración rápido para mostrar esta información. Este breve programa[2] imprimirá las direcciones IP para cualquier host que especifique en la línea de comandos:

```
/*
** mostrar.c
**
** mostrar las direcciones IP de un host dadas en la línea de comandos
*/

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <netinet/in.h>

int main(int argc, char *argv[])
{
    struct addrinfo sugerencias, *res, *p;
    estado int;
    char ipstr[INET6_ADDRSTRLEN];

    if (argc != 2) {
        fprintf(stderr,"uso: showip nombre de
        host\n"); retorno 1;
    }

    memset(&hints, 0, sizeof hints);
    Pistas.ai_family = AF_UNSPEC;  IPv4 o IPv6
    Pistas.ai_socktype = SOCK_STREAM;

    if ((status = getaddrinfo(argv[1], NULL, & hints, & res))!= 0) {
        fprintf(stderr,  "getaddrinfo:  %s\n",  gai_strerror(status);
        retorno 2;
    }

    printf("Direcciones IP para %s:\n\n", argv[1]);

    for(p = res;p != NULL; p = p->ai_next) {
        nulo *addr;
        char *ipver;
        estructura sockaddr_in
        *IPv4; estructura
        sockaddr_in6 *IPv6;

        obtener el puntero a la propia dirección
```

```
        diferentes campos en IPv4 e IPv6:
        if (p->ai_family == AF_INET) {  IPv4
            IPv4 = (estructura sockaddr_in *)p->ai_addr;
            addr = &(IPv4->sin_addr);
            ipver = "IPv4";
        } else {  IPv6
            IPv6 = (estructura sockaddr_in6 *)p->ai_addr;
            addr = &(IPv6->sin6_addr);
            ipver = "IPv6";
        }

         convertir la IP en una cadena e imprimirla:
        inet_ntop(p->ai_family, addr, ipstr, sizeof ipstr);
        printf(" %s: %s\n", ipver, ipstr);
    }

    freeaddrinfo(res);  Libera la lista enlazada

    devuelve 0;
}
```

Como ves, el código llama a `getaddrinfo()` en lo que sea que pases en la línea de comandos, que llena la lista enlazada apuntada por `res`, y luego podemos iterar sobre la lista e imprimir cosas o hacer lo que sea.

(Hay un poco de fealdad allí donde tenemos que profundizar en los diferentes tipos de `struct sockaddrs` dependiendo de la versión de IP. ¡Lo siento! No estoy seguro de una mejor manera de evitarlo).

¡Carrera de muestra! A todo el mundo le encantan las capturas de pantalla:

```
$ showip www.example.net
Direcciones IP para www.example.net:

  IPv4: 192.0.2.88

$ showip ipv6.example.com
Direcciones IP para ipv6.example.com:

  IPv4: 192.0.2.101
  IPv6: 2001:db8:8c00:22::171
```

Ahora que tenemos eso bajo control, usaremos los resultados que obtengamos de `getaddrinfo()` para pasar a otras funciones de socket y, por fin, ¡establecer nuestra conexión de red! ¡Sigue leyendo!

5.2 `socket()`—¡Obtén el descriptor de archivo!

Supongo que no puedo posponerlo más, tengo que hablar de la llamada al `sistema socket()`. Este es el desglose:

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

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

Pero, ¿cuáles son estos argumentos? Le permiten decir qué tipo de socket desea (IPv4 o IPv6, flujo o datagrama y TCP o UDP).

Solía ser que la gente codificaba estos valores, y todavía se puede hacer eso. (`el dominio` es `PF_INET` o `PF_INET6`, `el tipo` es `SOCK_STREAM` o `SOCK_DGRAM`, y el `protocolo` se puede establecer en `0` para elegir el

para el `tipo dado`. O puedes llamar a `getprotobyname()` para buscar el protocolo que quieres, "tcp" o "udp").

(Este `PF_INET` es un pariente cercano del `AF_INET` que puede usar al inicializar el campo `sin_family` en el sockaddr_in `de estructura`. De hecho, están tan estrechamente relacionados que en realidad tienen el mismo valor, y muchos programadores llamarán a `socket()` y pasarán `AF_INET` como primer argumento en lugar de `PF_INET`. Ahora, toma un poco de leche y galletas, porque es hora de una historia. Érase una vez, hace mucho tiempo, se pensó que tal vez una familia de direcciones (lo que significa el "AF" en "`AF_INET`") podría admitir varios protocolos a los que se refería su familia de protocolos (lo que significa el "PF" en "`PF_INET`"). Eso no sucedió. Y todos vivieron felices para siempre, The End. Por lo tanto, lo más correcto es usar `AF_INET` en su `struct sockaddr_in` y `PF_INET` en su llamada a `socket()`.)

De todos modos, basta de eso. Lo que realmente quieres hacer es usar los valores de los resultados de la llamada para
`getaddrinfo()`, y alójalos en `socket()` directamente de esta manera:

```
int s;
struct addrinfo sugerencias, *res;

Hacer la búsqueda
[finge que ya hemos rellenado la estructura de "sugerencias"]
getaddrinfo("www.example.com", "http", &hints, &res);

De nuevo, deberías hacer una comprobación de errores en getaddrinfo() y caminar
la lista enlazada "res" en busca de entradas válidas en lugar de solo
asumiendo que el primero es bueno (como lo hacen muchos de estos ejemplos).
Consulte la sección sobre cliente/servidor para ver ejemplos reales.

s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

`socket()` simplemente te devuelve un *descriptor de socket* que puedes usar en llamadas posteriores al sistema, o `-1` en caso de error. La variable global `errno` se establece en el valor del error (consulte la página del `comando man errno` para obtener más detalles y una nota rápida sobre el uso `de errno` en programas multiproceso).

Bien, bien, bien, pero ¿de qué sirve este enchufe? La respuesta es que realmente no es bueno por sí mismo, y necesita seguir leyendo y hacer más llamadas al sistema para que tenga sentido.

5.3 `bind()`—¿En qué puerto estoy?

Una vez que tenga un socket, es posible que deba asociarlo con un puerto en su máquina local. (Esto se hace comúnmente si va a `listen()` para las conexiones entrantes en un puerto específico: los juegos de red multijugador hacen esto cuando te dicen "conéctate al puerto 192.168.5.10 3490"). El número de puerto es utilizado por el kernel para hacer coincidir un paquete entrante con el descriptor de socket de un determinado proceso. Si solo vas a hacer un `connect()` (porque eres el cliente, no el servidor), esto probablemente sea innecesario. Léelo de todos modos, solo por diversión.

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 del fichero de socket devuelto por `socket()`. `my_addr` es un puntero a un `struct sockaddr` que contiene información sobre su dirección, es decir, el puerto y la dirección IP. `addrlen` es la longitud en bytes de esa dirección.

Vaya. Eso es un poco para absorber en un trozo. Veamos un ejemplo que vincula el socket al host en el que se ejecuta el programa, el puerto 3490:

```
struct addrinfo sugerencias, *res;
int calcetín;

Primero, cargue las estructuras de direcciones con getaddrinfo():

memset(&hints, 0, sizeof hints);
Pistas.ai_family = AF_UNSPEC;  usar IPv4 o IPv6, lo que ocurra
Pistas.ai_socktype = SOCK_STREAM;
Consejos.ai_flags = AI_PASSIVE;   rellene mi IP por mí

getaddrinfo(NULL, "3490", &hints, &res);

Hacer un enchufe:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

Vincúlalo al puerto que pasamos a getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);
```

Al usar la marca `AI_PASSIVE`, le digo al programa que se vincule a la IP del host en el que se está ejecutando. Si quieres enlazar a una dirección IP local específica, quita la `AI_PASSIVE` y pon una dirección IP como primer argumento para `getaddrinfo()`.

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

Muchos códigos antiguos empaquetan manualmente el `struct sockaddr_in` antes de llamar a `bind()`. Obviamente, esto es específico de IPv4, pero realmente no hay nada que te impida hacer lo mismo con IPv6, excepto que usar `getaddrinfo()` va a ser más fácil, generalmente. De todos modos, el código antiguo se parece a esto:

```
// !!! ESTA ES LA VIEJA FORMA !!

int calcetín;
estructura sockaddr_in my_addr;

sockfd = socket(PF_INET, SOCK_STREAM, 0);

my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(MYPORT);      corto, orden de bytes de red
my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");
memset(my_addr.sin_zero, '\0', tamañode my_addr.sin_zero);

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

En el código anterior, también puede asignar `INADDR_ANY` al campo `s_addr` si desea vincular a su dirección IP local (como la bandera `AI_PASSIVE`, arriba). La versión IPv6 de `INADDR_ANY` es una variable global `in6addr_any` que se asigna al campo `sin6_addr` del sockaddr_in6 de `estructura`. (También hay una macro `IN6ADDR_ANY_INIT` que se puede usar en un inicializador de variables).

Otra cosa a tener en cuenta al llamar a `bind()`: no vayas por la borda con tus números de puerto. ¡Todos los puertos por debajo de 1024 están RESERVADOS (a menos que seas el superusuario)! Puede tener cualquier número de puerto por encima de eso, hasta 65535 (siempre que no estén siendo utilizados por otro programa).

A veces, puede notar que intenta volver a ejecutar un servidor y `bind()` falla, reclamando "La dirección ya está en uso". ¿Qué significa eso? Bueno, un poco de un socket que estaba conectado todavía está dando vueltas en el kernel, y está acaparando el puerto. Puede esperar a que se borre (un minuto más o menos) o agregar código a su programa que le permita reutilizar el puerto, de esta manera:

```
int sí=1;
char yes='1'; La gente de Solaris usa esto

perder el molesto mensaje de error "Dirección ya en uso"
setsockopt(oyente, SOL_SOCKET, SO_REUSEADDR, &yeah, sizeof yeah);
```

Una pequeña nota final adicional sobre `bind()`: hay ocasiones en las que no tendrás que llamarlo absolutamente. Si te estás `conectando()` a una máquina remota y no te importa cuál es tu puerto local (como es el caso de `telnet` donde solo te importa el puerto remoto), simplemente puedes llamar a `connect()`, comprobará si el socket está desvinculado, y lo `vinculará()` a un puerto local no utilizado si es necesario.

5.4 `connect()`—¡Oye, tú!

Supongamos por unos minutos que es una aplicación telnet. Su usuario le ordena (al igual que en la película *TRON)* que obtenga un descriptor de archivo de socket. Cumples y llamas a `socket()`. A continuación, el usuario le indica que se conecte a "`10.12.110.57`" en el puerto "`23`" (el puerto telnet estándar). ¡Yow! ¿A qué te dedicas ahora?

Por suerte para ti, programa, ahora estás examinando la sección sobre `connect()`—cómo conectarse a un host remoto. ¡Así que sigue leyendo furiosamente! ¡No hay tiempo que perder!

La llamada `a connect()` es la siguiente:

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

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

`sockfd` es nuestro descriptor de archivo de socket de vecindario amigable, según lo devuelto por la `llamada` socket(), `serv_addr` es un `struct sockaddr` que contiene el puerto de destino y la dirección IP, y `addrlen` es la longitud en bytes de la estructura de direcciones del servidor.

Toda esta información se puede obtener de los resultados de la llamada `getaddrinfo()`, que es genial.

¿Está empezando a tener más sentido esto? No puedo escucharte desde aquí, así que tendré que esperar que así sea. Veamos un ejemplo en el que hacemos una conexión de socket a "`www.example.com`", puerto `3490`:

```
struct addrinfo sugerencias, *res;
int calcetín;

Primero, cargue las estructuras de direcciones con getaddrinfo():

memset(&hints, 0, sizeof hints);
Pistas.ai_family = AF_UNSPEC;
Pistas.ai_socktype = SOCK_STREAM;

getaddrinfo("www.example.com", "3490", &hints, &res);

Hacer un enchufe:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

¡conectar!

conectar (sockfd, res->ai_addr, res->ai_addrlen);
```

Una vez más, los programas de la vieja escuela completaron su propio `sockaddr_ins de estructura para pasar a connect()`. Puedes hacerlo si quieres. Véase la nota similar en la `sección` bind(), más arriba.

Asegúrate de verificar el valor devuelto de `connect()`: devolverá `-1` en caso de error y establecerá la variable `errno`.

Además, observe que no llamamos a `bind()`. Básicamente, no nos importa nuestro número de puerto local; Solo nos importa a dónde vamos (el puerto remoto). El kernel elegirá un puerto local para nosotros, y el sitio al que nos conectemos obtendrá automáticamente esta información de nosotros. No te preocupes.

5.5 `listen()`—¿Alguien podría llamarme, por favor?

Bien, es hora de un cambio de ritmo. ¿Qué sucede si no desea conectarse a un host remoto? Digamos, solo por diversión, que desea esperar las conexiones entrantes y manejarlas de alguna manera. El proceso consta de dos pasos: primero escuchas `()`, luego aceptas `()` (ver más abajo).

La llamada `listen()` es bastante simple, pero requiere un poco de explicación:

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

`sockfd` es el descriptor habitual del fichero de socket de la llamada al `sistema` socket(). `El backlog` es el número de conexiones permitidas en la cola entrante. ¿Qué significa eso? Bueno, las conexiones entrantes van a esperar en esta cola hasta que las `aceptes` (ver más abajo) y este es el límite de cuántas pueden ponerse en cola. La mayoría de los sistemas limitan silenciosamente este número a unos 20; Probablemente pueda salirse con la suya configurándolo en `5` o `10`.

De nuevo, como de costumbre, `listen()` devuelve `-1` y establece `errno` en caso de error.

Bueno, como probablemente puedas imaginar, necesitamos llamar a `bind()` antes de llamar a `listen()` para que el servidor se ejecute en un puerto específico. (¡Tienes que ser capaz de decirle a tus amigos a qué puerto conectarse!) Por lo tanto, si va a estar atento a las conexiones entrantes, la secuencia de llamadas al sistema que realizará es:

```
1 getaddrinfo();
2 socket();
3 vincular();
4 escuchar();
5 /* accept() va aquí */
```

Lo dejaré en el lugar del código de muestra, ya que se explica por sí mismo. (El código en el archivo`aceptar()` la sección más abajo, es más completa). La parte realmente complicada de todo este sha-bang es el llamado a `aceptar.`

5.6 `accept()`—"Gracias por llamar al puerto 3490."

Prepárate, ¡la `llamada accept()` es un poco extraña! Lo que va a suceder es lo siguiente: alguien muy, muy lejos intentará `conectarse a` tu máquina en un puerto en el que estés `escuchando`. Su conexión se pondrá en cola a la espera de ser `aceptada`. Llamas a `accept()` y le dices que obtenga la conexión pendiente. ¡Le devolverá un *nuevo descriptor de archivo de socket* para usar para esta única conexión! Así es, ¡de repente tienes *dos descriptores de archivo de socket* por el precio de uno! El original todavía está esperando más conexiones nuevas, y el recién creado finalmente está listo para `enviar()` y `recv()`. ¡Ahí estamos!

La convocatoria es la siguiente:

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

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

`sockfd` es el descriptor del socket `listen()`. Bastante fácil. `addr` suele ser un puntero a un `sockaddr_storage de estructura` local. Aquí es donde irá la información sobre la conexión entrante (y con ella puede determinar qué host lo está llamando desde qué puerto). `addrlen` es una variable entera local que debe establecerse en `sizeof(struct sockaddr_storage) antes de` que su dirección se pase a `accept()`. `accept()` no pondrá más de esa cantidad de bytes en `addr`. Si pone menos, cambiará el valor de `addrlen` para reflejarlo.

¿Adivina qué? `accept()` devuelve `-1` y establece `errno` si ocurre un error. Betcha no se dio cuenta de eso.

Al igual que antes, este es un montón para absorber en un fragmento, así que aquí hay un fragmento de código de muestra para su lectura:

```
#include <cadena.h>
#include <sys/tipos.h>
#include <sys/socket.h>
#include <netdb.h>

#define MYPORT "3490" el puerto al que se conectarán los usuarios
#define CARTERA DE PEDIDOS 10  Cuántas conexiones pendientes contiene la cola

Int principal(vacío)
{
    Estructura sockaddr_storage their_addr;
    socklen_t addr_size;
    Estructura Sugerencias de addrinfo, *Res;
    Int calcetín, new_fd;

    // !! ¡No olvide su verificación de errores para estas llamadas!

    Primero, cargue las estructuras de direcciones con getaddrinfo():

    memset(&Consejos, 0, tamañode Consejos);
    Consejos.ai_family = AF_UNSPEC; usar IPv4 o IPv6, lo que ocurra
    Consejos.ai_socktype = SOCK_STREAM;
    Consejos.ai_flags = AI_PASSIVE;   rellene mi IP por mí

    getaddrinfo(NULO, MYPORT, &Consejos, &Res);

    Haz un socket, átalo y escucha en él:

    calcetín = enchufe(Res->ai_family, Res->ai_socktype,
                                                Res->ai_protocol);
    atar(calcetín, Res->ai_addr, Res->ai_addrlen);
    escuchar(calcetín, CARTERA DE PEDIDOS);

    Ahora acepte una conexión entrante:

    addr_size = tamañode their_addr;
    new_fd = aceptar(calcetín, (Estructura calcetín *)&their_addr,
                                                    &addr_size);

    ¡Listo para comunicarse en el descriptor de socket new_fd!
    .
    .
    .
```

De nuevo, ten en cuenta que usaremos el descriptor de socket `new_fd` para todas las `llamadas a send()` y `recv()`. Si solo obtienes una única conexión, puedes `cerrar ()` el `calcetín` de escucha para evitar más conexiones entrantes en el mismo puerto, si así lo deseas.

5.7 `send()` y `recv()`—¡Háblame, nena!

Estas dos funciones son para comunicarse a través de sockets de flujo o sockets de datagramas conectados. Si quieres usar sockets de datagramas regulares no conectados, necesitarás ver la sección sobre `sendto()` y `recvfrom()`, a continuación.

> Aquí hay algo que podría (o no) ser nuevo para ti: estas son llamadas *de bloqueo*. Es decir, `recv()` se bloqueará hasta que haya algunos datos listos para recibir. "¡¿Pero qué significa 'bloquear', ya?" Significa que su programa se detendrá allí, en esa llamada al sistema, hasta que alguien le envíe algo. (La jerga técnica de OS para "detener" en esa oración es en realidad *sueño*, por lo que podría usar esos términos indistintamente). `send()` también puede bloquear si las cosas que estás enviando están atascadas de alguna manera, pero eso es más raro. Volveremos a revisar este concepto más adelante y hablaremos sobre cómo evitarlo cuando lo necesites.

Esta es la `llamada a send()`:

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

`sockfd` es el descriptor de socket al que desea enviar datos (ya sea el que devuelve `socket()` o el que obtuvo con `accept()`). `msg` es un puntero a los datos que desea enviar, y `len` es la longitud de esos datos en bytes. Simplemente establezca `las banderas` en `0`. (Véase la `send()` para obtener más información sobre los indicadores.)

Algunos códigos de ejemplo podrían ser:

```
1  char *msg = "¡Beej estuvo
2  aquí!"; int len, bytes_sent;
3  .
4  .
5  .
6  len = strlen(msg);
7  bytes_sent = enviar(calcetín, mensaje, len, 0);
8  .
9  .
10 .
```

`send()` devuelve el número de bytes realmente enviados, *¡esto podría ser menor que el número que le dijiste que enviara!* Mira, a veces le dices que envíe una gran cantidad de datos y simplemente no puede manejarlos. Disparará la mayor cantidad de datos que pueda y confiará en que usted enviará el resto más tarde. Recuerda, si el valor devuelto por `send()` no coincide con el valor en `len`, depende de ti enviar el resto de la cadena. La buena noticia es esta: si el paquete es pequeño (menos de 1K más o menos) *probablemente* logrará enviar todo de una sola vez. Otra vez
`-1` se devuelve en caso de error y `errno` se establece en el

número de error. La `llamada recv()` es similar en

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

muchos aspectos:

`sockfd` es el descriptor del socket desde el que se va a leer, `buf` es el búfer en el que se lee la información, `len` es la longitud máxima del búfer y `las banderas` se pueden establecer de nuevo en `0`. (Véase la `recv()` para información de banderas.)

`recv()` devuelve el número de bytes realmente leídos en el búfer, o `-1` en caso de error (con `errno` establecido, según corresponda).

¡Esperar! `recv()` puede devolver `0`. Esto solo puede significar una cosa: ¡el lado remoto le ha cerrado la conexión! Un valor devuelto de `0` es la forma que `tiene recv()` de informarte de que esto ha ocurrido.

Allí fue fácil, ¿no? ¡Ahora puede pasar datos de un lado a otro en los sockets de transmisión! ¡Whee! ¡Eres un programador de redes Unix!

5.8 `sendto()` y `recvfrom()`: háblame, al estilo DGRAM

"Todo esto está muy bien", te oigo decir, "¿pero dónde me deja esto con los sockets de datagramas sin conectar?" No problemo, amigo. Tenemos justo lo que necesitamos.

Dado que los sockets de datagramas no están conectados a un host remoto, ¿adivina qué información debemos dar antes de enviar un paquete? ¡Así es! ¡La dirección de destino! Esta es la primicia:

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

Como puedes ver, esta llamada es básicamente la misma que la llamada a `send()` con la adición de otras dos piezas de información. `to` es un puntero a un `struct sockaddr` (que probablemente será otro `struct sockaddr_in` o `struct sockaddr_in6` o `struct sockaddr_storage` que haya lanzado en el último minuto) que contiene la dirección IP y el puerto de destino. `tolen`, un `int` en profundidad, simplemente se puede establecer en `sizeof *to` o `sizeof(struct sockaddr_storage)`.

Para tener en tus manos la estructura de direcciones de destino, probablemente la obtendrás de `getaddrinfo()`, o de `recvfrom()`, a continuación, o la completarás a mano.

Al igual que con `send()`, `sendto()` devuelve el número de bytes realmente enviados (¡que, de nuevo, podría ser menor que el número de bytes que le dijiste que enviara!), o `-1` en caso de error.

Igualmente similares son `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, esto es como `recv()` con la adición de un par de campos. `from` es un puntero a un `sockaddr_storage de estructura` local que se rellenará con la dirección IP y el puerto de la máquina de origen. `fromlen` es un puntero a un `int` local que debe inicializarse en `sizeof *from` o `sizeof(struct sockaddr_storage)`. Cuando la función regrese, `fromlen` contendrá la longitud de la dirección realmente almacenada en `from`.

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

Entonces, aquí hay una pregunta: ¿por qué usamos `struct sockaddr_storage` como tipo de socket? ¿Por qué no `estructurar sockaddr_in`? Porque, como ven, no queremos atarnos a IPv4 o IPv6. Así que usamos el struct genérico `sockaddr_storage` que sabemos que será lo suficientemente grande para cualquiera de los dos.

(Entonces... Aquí hay otra pregunta: ¿Por qué `struct sockaddr` no es lo suficientemente grande para cualquier dirección? ¡Incluso lanzamos el struct de propósito general `sockaddr_storage al` struct sockaddr `de propósito general`! Parece extraño y redundante, ¿eh? La respuesta es que simplemente no es lo suficientemente grande, y supongo que cambiarlo en este momento sería problemático. Así que hicieron uno nuevo).

Recuerda, si `conectas ()` un socket de datagramas, simplemente puedes usar `send()` y `recv()` para todas tus transacciones. El socket en sí sigue siendo un socket de datagramas y los paquetes aún usan UDP, pero la interfaz del socket agregará automáticamente la información de destino y origen por usted.

5.9 `close()` y `shutdown()`—¡Sal de mi cara!

¡Vaya! Has estado `enviando y recibiendo` datos todo el día, y los has tenido. Ya está listo para cerrar la conexión en el descriptor de socket. Esto es fácil. Puedes usar la función close() del descriptor de archivo Unix normal :

```
cerrar (calcetín);
```

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

En caso de que quieras un poco más de control sobre cómo se cierra el socket, puedes usar la `función shutdown()`. Te permite cortar la comunicación en una determinada dirección, o en ambos sentidos (al igual que `hace close()`). Sinopsis:

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

`sockfd` es el descriptor del archivo de socket que desea apagar, y `how` es uno de los siguientes:

¿Cómo	Efecto
0	No se permiten más recepciones
1	No se permiten envíos adicionales
2	No se permiten más envíos y recepciones (como `close()`)

`shutdown()` devuelve `0` en caso de éxito y `-1` en caso de error (con `errno` establecido en consecuencia).

Si se digna a usar `shutdown()` en sockets de datagramas no conectados, simplemente hará que el socket no esté disponible para futuras `llamadas a send()` y `recv()` (recuerde que puede usarlas si `conecta` () su socket de datagramas).

Es importante tener en cuenta que `shutdown()` en realidad no cierra el descriptor del archivo, solo cambia su usabilidad. Para liberar un descriptor de socket, necesitas usar `close()`.

Nada que ver.

(Excepto para recordar que si está usando Windows y Winsock, debe llamar a `closesocket()` en lugar de `close()`.)

5.10 `getpeername()`—¿Quién eres?

Esta función es muy fácil.

Es tan fácil que casi no le di su propia sección. Pero aquí está de todos modos.

La función `getpeername()` te dirá quién está en el otro extremo 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 un `struct sockaddr` (o un `struct sockaddr_in`) que contendrá la información sobre el otro lado de la conexión, y `addrlen` es un puntero a un `int`, que debe inicializarse en `sizeof *addr` o `sizeof(struct sockaddr)`.

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

Una vez que tengas su dirección, puedes usar `inet_ntop()`, `getnameinfo()` o `gethostbyaddr()` para imprimir u obtener más información. No, no puede obtener su nombre de inicio de sesión. (Ok, ok. Si el otro equipo está ejecutando un demonio ident, esto es posible. Esto, sin embargo, está más allá del alcance de este documento. Echa un vistazo a RFC 1413[3] para obtener más información).

5.11 `gethostname()`—¿Quién soy yo?

Incluso más fácil que `getpeername()` es la función `gethostname()`. Devuelve el nombre del equipo en el que se está ejecutando el programa. El nombre puede ser utilizado por `getaddrinfo()`, arriba, para determinar la dirección IP de tu máquina local.

¿Qué podría ser más divertido? Podría pensar en algunas cosas, pero no pertenecen a la programación de sockets. De todos modos, aquí está el desglose:

```
#include <unistd.h>

int gethostname (char *hostname, tamaño size_t);
```

Los argumentos son simples: `hostname` es un puntero a una matriz de caracteres que contendrá el nombre de host cuando se devuelva la función, y size es la longitud en bytes de la matriz de nombres de `host`.

[3]https://tools.ietf.org/html/rfc1413

La función devuelve `0` si se completa correctamente y `-1` si se produce un error, estableciendo `errno` como de costumbre.

Capítulo 6

Antecedentes cliente-servidor

Es un mundo cliente-servidor, nena. Casi todo en la red se ocupa de los procesos del cliente, se comunica con los procesos del servidor y viceversa. Tomemos `telnet`, por ejemplo. Cuando se conecta a un host remoto en el puerto 23 con telnet (el cliente), un programa en ese host (llamado `telnetd`, el servidor) cobra vida. Maneja la conexión telnet entrante, lo configura con un mensaje de inicio de sesión, etc.

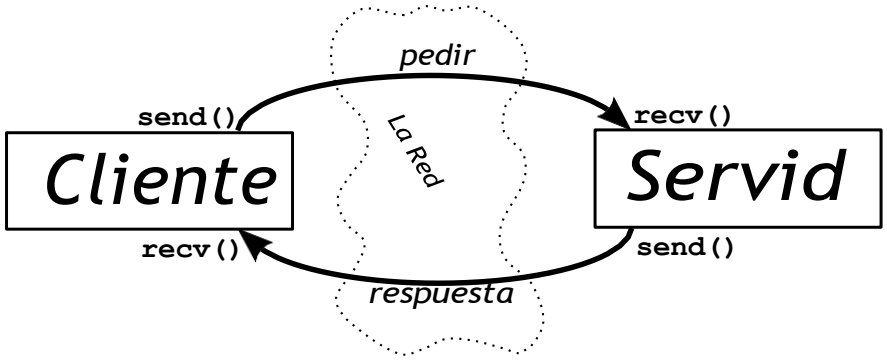

Figura 6.1: Interacción cliente-servidor.

El intercambio de información entre el cliente y el servidor se resume en el diagrama anterior.

Tenga en cuenta que el par cliente-servidor puede hablar `SOCK_STREAM`, `SOCK_DGRAM` o cualquier otra cosa (siempre que hablen lo mismo). Algunos buenos ejemplos de pares cliente-servidor son `telnet`/`telnetd`, `ftp`/`ftpd` o `Firefox`/`Apache`. Cada vez que usas `ftp`, hay un programa remoto, `ftpd`, que te sirve.

A menudo, solo habrá un servidor en una máquina, y ese servidor manejará múltiples clientes usando `fork()`. La rutina básica es: el servidor esperará una conexión, `la aceptará` y `fork()` un proceso hijo para que la maneje. Esto es lo que hace nuestro servidor de ejemplo en la siguiente sección.

6.1 Un servidor de transmisión simple

Todo lo que hace este servidor es enviar la cadena "`¡Hola, mundo!`" a través de una conexión de transmisión. Todo lo que necesita hacer para probar este servidor es ejecutarlo en una ventana y conectarlo por telnet desde otra con:

```
$ telnet remotehostname 3490
```

donde `remotehostname` es el nombre de la máquina en la que se está

ejecutando. El código del servidor[1]:

[1]https://beej.us/guide/bgnet/source/examples/server.c

```
/*
** server.c -- una demostración del servidor de socket de transmisión
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <cadena.h>
#include <sys/tipos.h>
#include <sys/socket.h>
#include <netinet/en.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <señal.h>

PUERTO #define "3490" el puerto al que se conectarán los usuarios

#define CARTERA DE PEDIDOS 10  Cuántas conexiones pendientes se mantendrá en la cola

vacío sigchld_handler(Int s)
{
    (vacío)s; Advertencia de variable silenciosa no utilizada

    waitpid() podría sobrescribir errno, así que lo guardamos y restauramos:
    Int saved_errno = errno;

    mientras(Waitpid(-1, NULO, WNOHANG) > 0);

    errno = saved_errno;
}


obtener sockaddr, IPv4 o IPv6:
vacío *get_in_addr(Estructura calcetín *Sa)
{
    si (Sa->sa_family == AF_INET) {
        devolución &(((Estructura sockaddr_in*)Sa)->sin_addr);
    }

    devolución &(((Estructura sockaddr_in6*)Sa)->sin6_addr);
}

Int principal(vacío)
{
    Escúchalo en sock_fd, Nueva conexión en new_fd
    Int calcetín, new_fd;
    Estructura Sugerencias de addrinfo, *servinfo, *p;
    Estructura sockaddr_storage their_addr; Información de la dirección del conector
    socklen_t sin_size;
    Estructura Sigaction SA;
    Int sí=1;
    carbonizar s[INET6_ADDRSTRLEN];
    Int Rv;

    memset(&Consejos, 0, tamañode Consejos);
```

```
58      Consejos.ai_family = AF_INET;
59      Consejos.ai_socktype = SOCK_STREAM;
60      Consejos.ai_flags = AI_PASSIVE; usar mi IP
61
62      si ((Rv = getaddrinfo(NULO, PUERTO, &Consejos, &servinfo)) != 0) {
63          fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(Rv));
64          devolución 1;
65      }
66
67      recorrer todos los resultados y enlazar al primero que podamos
68      para(p = servinfo; p != NULO; p = p->ai_next) {
69          si ((calcetín = enchufe(p->ai_family, p->ai_socktype,
70                  p->ai_protocol)) == -1) {
71              perror("Servidor: socket");
72              continuar;
73          }
74
75          si (setsockopt(calcetín, SOL_SOCKET, SO_REUSEADDR, &Sí,
76                  tamañode(Int)) == -1) {
77              perror("setsockopt");
78              salida(1);
79          }
80
81          si (atar(calcetín, p->ai_addr, p->ai_addrlen) == -1) {
82              cerrar(calcetín);
83              perror("Servidor: Enlace");
84              continuar;
85          }
86
87          quebrar;
88      }
89
90      freeaddrinfo(servinfo); todo hecho con esta estructura
91
92      si (p == NULO) {
93          fprintf(stderr, "Servidor: No se pudo enlazar\n");
94          salida(1);
95      }
96
97      si (escuchar(calcetín, CARTERA DE PEDIDOS) == -1) {
98          perror("Escuchar");
99          salida(1);
100     }
101
102     Sa.sa_handler = sigchld_handler; Cosechar todos los procesos muertos
103     sigemptyset(&Sa.sa_mask);
104     Sa.sa_flags = SA_RESTART;
105     si (Sigacción(SIGCHLD, &Sa, NULO) == -1) {
106         perror("Sigacción");
107         salida(1);
108     }
109
110     printf("Servidor: Esperando conexiones...\n");
111
112     mientras(1) { bucle principal accept()
113         sin_size = tamañode their_addr;
114         new_fd = aceptar(calcetín, (Estructura calcetín *)&their_addr,
```

```
            &sin_size);
        if (new_fd == -1) {
            perror("aceptar");
            continuar;
        }

        inet_ntop(their_addr.ss_family,
            get_in_addr((struct sockaddr *)&their_addr),
            s, tamañode s);
        printf("servidor: obtuvo la conexión de %s\n", s);

        Si ( !fork()) {  este es el proceso hijo
            close(sockfd);  niño no necesita al oyente si
            (send(new_fd, "¡Hola, mundo!", 13, 0) == -1)
                error ("enviar");
            cerrar(new_fd);
            salida(0);
        }
        cerrar(new_fd);  El padre no necesita esto
    }

    devuelve 0;
}
```

En caso de que tengas curiosidad, tengo el código en una gran `función main()` para (siento) claridad sintáctica. Siéntete libre de dividirlo en funciones más pequeñas si te hace sentir mejor.

(Además, todo esto `de sigaction()` puede ser nuevo para ti, está bien. El código que está allí es responsable de cosechar los procesos zombis que aparecen cuando salen los `procesos hijos bifurcados`. Si haces muchos zombis y no los cosechas, el administrador de tu sistema se agitará).

Puede obtener los datos de este servidor mediante el cliente que se muestra en la siguiente sección.

6.2 Un cliente de transmisión simple

Este tipo es incluso más fácil que el servidor. Todo lo que hace este cliente es conectarse al host que especifique en la línea de comandos, el puerto 3490. Obtiene la cadena que envía el servidor.

La fuente del cliente[2]:

```
/*
** client.c -- una demostración del cliente de socket 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>

#include <arpa/inet.h>
```

[2]https://beej.us/guide/bgnet/source/examples/client.c

```
PUERTO #define "3490" El cliente del puerto al que se conectará

#define MAXDATASIZE 100 Número máximo de bytes que podemos obtener a la vez

obtener sockaddr, IPv4 o IPv6:
vacío *get_in_addr(Estructura calcetín *Sa)
{
    si (Sa->sa_family == AF_INET) {
        devolución &(((Estructura sockaddr_in*)Sa)->sin_addr);
    }

    devolución &(((Estructura sockaddr_in6*)Sa)->sin6_addr);
}

Int principal(Int ARGC, carbonizar *argv[])
{
    Int calcetín, numbytes;
    carbonizar Buf[MAXDATASIZE];
    Estructura Sugerencias de addrinfo, *servinfo, *p;
    Int Rv;
    carbonizar s[INET6_ADDRSTRLEN];

    si (ARGC != 2) {
        fprintf(stderr,"Uso: nombre de host del cliente\n");
        salida(1);
    }

    memset(&Consejos, 0, tamañode Consejos);
    Consejos.ai_family = AF_UNSPEC;
    Consejos.ai_socktype = SOCK_STREAM;

    si ((Rv = getaddrinfo(argv[1], PUERTO, &Consejos, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(Rv));
        devolución 1;
    }

    recorrer todos los resultados y conectarnos a los primeros que podamos
    para(p = servinfo; p != NULO; p = p->ai_next) {
        si ((calcetín = enchufe(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("Cliente: socket");
            continuar;
        }

        inet_ntop(p->ai_family,
            get_in_addr((Estructura calcetín *)p->ai_addr),
            s, tamañode s);
        printf("cliente: intentando conectarse a %s\n", s);

         Si (connect(sockfd, p->ai_addr  p->ai_addrlen) == -1) {
           perror("cliente: conectar");
            cerrar (calcetín);
            continuo;
        }

        descanso;
    }
```

```
74
75      if (p == NULL) {
76          fprintf(stderr, "cliente: no se pudo conectar\n");
77          retorno 2;
78      }
79
80      inet_ntop(p->ai_family,
81              get_in_addr((struct sockaddr *)p->ai_addr),
82              s, tamaño de s);
83      printf("cliente: conectado a %s\n", s);
84
85      freeaddrinfo(servinfo);  todo hecho con esta estructura
86
87      if ((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {
88          perror("recv");
89          salida(1);
90      }
91
92      buf[númerobytes] = '\0';
93
94      printf("cliente: recibido '%s'\n",buf); cerrar
95
96      (calcetín);
97
98      devuelve 0;
99  }
```

Ten en cuenta que si no ejecutas el servidor antes de ejecutar el cliente, `connect()` devuelve "Conexión rechazada". Muy útil.

6.3 Sockets de datagramas

Ya hemos cubierto los conceptos básicos de los sockets de datagramas UDP con nuestra discusión de `sendto()` y `recvfrom()`, arriba, así que solo presentaré un par de programas de muestra: `talker.c` y `listener.c`.

`El oyente` se encuentra en una máquina esperando un paquete entrante en el puerto 4950. `Talker` envía un paquete a ese puerto, en la máquina especificada, que contiene lo que el usuario ingresa en la línea de comandos.

Debido a que los sockets de datagramas no tienen conexión y simplemente disparan paquetes al éter con un desprecio cruel por el éxito, le diremos al cliente y al servidor que usen específicamente IPv6. De esta manera evitamos la situación en la que el servidor está escuchando en IPv6 y el cliente envía en IPv4; Los datos simplemente no se recibirían. (En nuestro mundo de sockets de flujo TCP conectados, es posible que aún tengamos la discrepancia, pero el error en `connect()` para una familia de direcciones haría que volviéramos a intentarlo para la otra).

Aquí está la fuente para `listener.c`[3]:

```
1   /*
2   ** listener.c -- una demostración de "servidor" de sockets de datagramas
3   */
4
5   #include <stdio.h>
6   #include <stdlib.h>
7   #include <unistd.h>
8   #include <errno.h>
9   #include <string.h>
10  #include <sys/types.h>
```

[3]https://beej.us/guide/bgnet/source/examples/listener.c

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

#define MYPORT "4950"    El puerto al que se conectarán los    Para
usuarios

#define MAXBUFLEN 100

obtener sockaddr, IPv4 o IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(vacío)
{
    int calcetín;
    struct addrinfo sugerencias, *servinfo, *p;
    int rv;
    int numbytes;
    estructura sockaddr_storage their_addr;
    char buf[MAXBUFLEN];
    socklen_t addr_len;
    char s[INET6_ADDRSTRLEN];

    memset(&hints, 0, sizeof hints);
    Pistas.ai_family = AF_INET6;  establecido en AF_INET para
    usar IPv4
    Pistas.ai_socktype = SOCK_DGRAM;
    Pistas.ai_flags = AI_PASSIVE;  usar mi IP

    if ((rv = getaddrinfo(NULL, MYPORT, &hints, &servinfo)) != 0)  {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv);
        retorno 1;
    }

    recorrer todos los resultados y enlazar al primero que podamos
    for(p = servinfo; p != NULL; p = p->ai_next) {
        si ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("oyente: socket");
            continuar;
        }

        if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            cerrar (calcetín);
            perror("oyente: vincular");
            continuo;
        }

        descanso;
    }

```

```
    if (p == NULL) {
        fprintf(stderr, "oyente: no se pudo enlazar el
        socket\n"); retorno 2;
    }

    freeaddrinfo(servinfo);

    printf("oyente: esperando a recvfrom...\n");

    addr_len = tamaño de their_addr;
    if ((numbytes = recvfrom(sockfd, buf, MAXBUFLEN-1 , 0,
        (struct sockaddr *)&their_addr, &addr_len)) == -1) {
        perror("recvfrom");
        salida(1);
    }

    printf("listener: obtuvo el paquete de %s\n",
        inet_ntop(their_addr.ss_family,
            get_in_addr((struct sockaddr *)&their_addr),
            s, sizeof s));
    printf("oyente: el paquete es %d bytes long\n", numbytes);
    buf[númerobytes] = '\0';
    printf("listener: el paquete contiene \"%s\"\n", buf);

    cerrar (calcetín);

    devuelve 0;
}
```

Observa que en nuestra llamada a `getaddrinfo()` finalmente estamos usando SOCK_DGRAM. Además, ten en cuenta que no es necesario `escuchar ()` o `aceptar()`. ¡Esta es una de las ventajas de usar sockets de datagramas no conectados!

Luego viene la fuente para `talker.c`[4]:

```
/*
** talker.c -- una demostración de datagramas "cliente"
*/

#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 PUERTO DE SERVIDOR "4950" el puerto al que se conectarán los usuarios

int main(int argc, char *argv[])
{
    int calcetín;
    struct addrinfo sugerencias, *servinfo, *p;
    int rv;
```

[4]https://beej.us/guide/bgnet/source/examples/talker.c

```
    int numbytes;

    if (argc != 3) {
        fprintf(stderr,"usage: talker hostname message\n");
        salida(1);
    }

    memset(&hints, 0, sizeof hints);
    Pistas.ai_family = AF_INET6;  establecido en AF_INET para
    usar IPv4
    Pistas.ai_socktype = SOCK_DGRAM;

    rv = getaddrinfo(argv[1], SERVERPORT, &hints, &servinfo);
    if (rv != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv);
        retorno 1;
    }

    Recorra todos los resultados y haga un socket
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("hablador: socket");
            continuar;
        }

        descanso;
    }

    if (p == NULL) {
        fprintf(stderr, "talker: failed to create socket\n");
        retorno 2;
    }

    if ((numbytes = sendto(sockfd, argv[2], strlen(argv[2]), 0,
             p->ai_addr, p->ai_addrlen)) == -1) {
        perror("hablador: enviar");
        salida(1);
    }

    freeaddrinfo(servinfo);

    printf("hablador: enviado %d bytes a %s\n", numbytes,
    argv[1]); cerrar (calcetín);

}
```

¡Míralos comunicarse! ¡Diversión para toda la familia nuclear!

¡Ni siquiera tienes que ejecutar el servidor esta vez! Puede ejecutar `talker` por sí mismo, y simplemente dispara paquetes felizmente al éter donde desaparecen si nadie está listo con un `recvfrom()` en el otro lado. Recuerde: ¡no se garantiza que los datos enviados mediante sockets de datagramas UDP lleguen!

Excepto por un pequeño detalle más que he mencionado muchas veces en el pasado: los sockets de datagramas conectados. Necesito hablar de esto aquí, ya que estamos en la sección de datagramas del documento. Digamos que `el hablante` llama a `connect()` y especifica la dirección del `oyente`. A partir de ese momento, `el hablante` solo puede enviar y recibir desde la dirección especificada por `connect()`. Por esta razón, no tienes que usar `sendto()`

y `recvfrom()`; simplemente puedes usar `send()` y `recv()`.

Capítulo 7

Técnicas ligeramente avanzadas

Estos no son *realmente* avanzados, pero están saliendo de los niveles más básicos que ya hemos cubierto. De hecho, si has llegado hasta aquí, deberías considerarte bastante experto en los conceptos básicos de la programación en red Unix. ¡Felicidades!

Así que aquí entramos en el nuevo mundo de algunas de las cosas más esotéricas que quizás quieras aprender sobre los enchufes. ¡Hazlo!

7.1 Bloqueante

Bloqueante. Has oído hablar de él, ¿qué diablos es? En pocas palabras, "bloquear" es la jerga técnica para "dormir". Probablemente hayas notado que cuando ejecutas `el oyente`, arriba, simplemente se queda allí hasta que llega un paquete. Lo que pasó es que llamó a `recvfrom()`, no había datos, por lo que se dice que `recvfrom()` "bloquea" (es decir, duerme allí) hasta que lleguen algunos datos.

Bloque de muchas funciones. `accept()`. Todas las funciones del `bloque recv()`. La razón por la que pueden hacer esto es porque se les permite. Cuando creas por primera vez el descriptor de socket con `socket()`, el kernel lo establece en bloqueo. Si no quieres que un socket se bloquee, tienes que hacer una llamada a `fcntl()`:

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

Al establecer un socket para que no bloquee, puede "sondear" eficazmente el socket para obtener información. Si intenta leer desde un socket que no bloquea y no hay datos allí, no está permitido bloquear: devolverá `-1` y `errno` se establecerá en `EAGAIN` o `EWOULDBLOCK`.

(Espere, ¿puede devolver `EAGAIN` *o* `EWOULDBLOCK`? ¿Cuál revisas? En realidad, la especificación no especifica cuál devolverá su sistema, por lo que para la portabilidad, verifique ambos).

Sin embargo, en términos generales, este tipo de sondeo es una mala idea. Si pones tu programa en una espera ocupada buscando datos en el socket, absorberás tiempo de CPU como si estuviera pasando de moda. Una solución más elegante para verificar si hay datos esperando ser leídos viene en la siguiente sección en `poll()`.

7.2 `poll()`: multiplexación de E/S síncrona

Lo que realmente quieres ser capaz de hacer es de alguna manera monitorear un *montón* de sockets a la vez y luego manejar los que tienen datos listos. De esta manera, no tiene que sondear continuamente todos esos sockets para ver cuáles están listos para leer.

> *Una advertencia: `poll()` es terriblemente lento cuando se trata de un número gigante de conexiones. En esas circunstancias, obtendrá un mejor rendimiento de una biblioteca de eventos como libevent[a] que intenta utilizar el método más rápido posible disponible en su sistema.*
>
> [a] https://libevent.org/

Entonces, ¿cómo se pueden evitar las encuestas? No es un poco irónico que se pueda evitar el sondeo utilizando la llamada al `sistema poll().` En pocas palabras, vamos a pedirle al sistema operativo que haga todo el trabajo sucio por nosotros, y solo nos avisará cuando algunos datos estén listos para leer en qué sockets. Mientras tanto, nuestro proceso puede entrar en suspensión, ahorrando recursos del sistema.

El plan de juego general es mantener una matriz de encuestas de `estructuras con información sobre qué descriptores de socket` queremos monitorear y qué tipo de eventos queremos monitorear. El sistema operativo se bloqueará en la `llamada poll()` hasta que ocurra uno de esos eventos (por ejemplo, "¡socket listo para leer!") o hasta que ocurra un tiempo de espera especificado por el usuario.

Útilmente, un `socket listen()`ing devolverá "listo para leer" cuando una nueva conexión entrante esté lista para ser `aceptada`.

Basta de bromas. ¿Cómo usamos esto?

```
#include <poll.h>

int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

`fds` es nuestra matriz de información (qué sockets monitorear para qué), `nfds` es el recuento de elementos en la matriz y `timeout` es un tiempo de espera en milisegundos. Devuelve el número de elementos de la matriz en los que se ha producido un evento.

Echemos un vistazo a esa `estructura`:

```
struct pollfd {
    Int Fd;         El descriptor de socket
    corto Eventos;  Mapa de bits de los eventos que nos interesan
     reventos cortos;  A la vuelta, mapa de bits de los eventos que ocurrieron
};
```

Así que vamos a tener una matriz de esos, y estableceremos el `campo fd` para cada elemento en un descriptor de socket que nos interese monitorear. Y luego estableceremos el campo de `eventos` para indicar el tipo de eventos que nos interesan.

El campo events es el OR bit a bit de lo siguiente:

Macro	Descripción
`POLÍN`	Avisarme cuando los datos estén listos para `recv()` en este zócalo.
`ENCUESTA`	Avísame cuando pueda `send()` datos a este socket sin bloquear.
`POLLHUP`	Avísame cuando el control remoto cerró la conexión.

Una vez que tenga su matriz de `struct pollfds en orden, puede pasarla a poll(),` pasando también el tamaño de la matriz, así como un valor de tiempo de espera en milisegundos. (Puede especificar un tiempo de espera negativo para esperar eternamente).

Después de `que poll()` devuelva, puede verificar el `campo revents` para ver si `POLLIN` o `POLLOUT` está establecido, lo que indica que el evento ocurrió.

(En realidad, hay más cosas que puedes hacer con la `llamada poll().` Consulte la página del `comando man poll(),` a continuación, para obtener más detalles).

Aquí hay un ejemplo[1] en el que esperaremos 2,5 segundos para que los datos estén listos para leer desde la entrada estándar, es decir, cuando presione `RETURN:`

```
#include <stdio.h>
#include <poll.h>

int main(vacío)
{
    Estructura Pollfd PFDS[1];  Más si quieres monitorizar más

    dfps[0].fd = 0;           Entrada estándar
    dfps[0].eventos = POLLIN;  Dime cuándo estoy listo para leer

    Si también necesita monitorear otras cosas:
    pfds[1].fd = some_socket; Algunos descriptores de socket
    pfds[1].events = POLLIN; // Dígame cuando esté listo para

    leer printf("Presione RETURN o espere 2.5 segundos para que

    se agote el tiempo de espera\n"); int num_events = sondeo

    (pfds, 1, 2500);  Tiempo de espera de 2,5 segundos

    if (num_events == 0) {
        printf("¡Se agotó el tiempo de espera de la
        encuesta!\n");
    } else {
        int pollin_happened = dfps[0].revents y POLLIN;

        if (pollin_happened) {
            printf("El descriptor de archivo %d está listo
                    para leer\n", pfds[0].FD);
        } else {
            printf("Ocurrió un evento inesperado: %d\n",
                    dfps[0].reventos);
        }
}
```

han producido eventos. No le dice *qué* elementos en la matriz (todavía tiene que buscar eso), pero sí le dice cuántas entradas tienen un campo revents distinto de cero (por lo que puede detener el escaneo después de encontrar tantos).

Un par de preguntas podrían surgir aquí: ¿cómo agregar nuevos descriptores de archivo al conjunto que paso a `poll()`? Para esto, simplemente asegúrese de tener suficiente espacio en la matriz para todo lo que necesite, o `realloc()` más espacio según sea necesario.

¿Qué pasa con la eliminación de objetos del conjunto? Para ello, puede copiar el último elemento de la matriz por encima del que está eliminando. Y luego pasa uno menos como el conteo a `poll()`. Otra opción es que puede establecer cualquier `campo fd` en un número negativo y `poll()` lo ignorará.

¿Cómo podemos ponerlo todo junto en un servidor de chat al que se pueda `conectar por` telnet?

Lo que haremos es iniciar un socket de escucha y agregarlo al conjunto de descriptores de archivo a `poll()`. (Se mostrará listo para leer cuando haya una conexión entrante).

A continuación, agregaremos nuevas conexiones a nuestra `matriz struct pollfd`. Y lo haremos crecer dinámicamente si ejecutamos

[1]https://beej.us/guide/bgnet/source/examples/poll.c

fuera del espacio.

Cuando se cierra una conexión, la eliminaremos de la matriz.

Y cuando una conexión esté lista para leer, leeremos los datos de ella y enviaremos esos datos a todas las demás conexiones para que puedan ver lo que escribieron los otros usuarios.

Así que dale una oportunidad a este servidor de encuestas[2]. Ejecútelo en una ventana, luego `telnet localhost 9034` desde varias otras ventanas de terminal. Deberías poder ver lo que escribes en una ventana en las otras (después de pulsar INTRO).

No solo eso, sino que si presiona `CTRL-]` y escribe `quit` para salir `de telnet`, el servidor debería detectar la desconexión y eliminarlo de la matriz de descriptores de archivo.

```
/*
** pollserver.c -- un cursi servidor de chat multipersona
*/

#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>
#include <netdb.h>
#include <poll.h>

PUERTO #define "9034"  Puerto en el que estamos escuchando

/*
 * Convierta el socket en una cadena de dirección IP.
 * addr: struct sockaddr_in o struct sockaddr_in6
 */
const char *inet_ntop2(void *addr, char *buf, tamaño size_t)
{
    struct sockaddr_storage *sas = addr;
    estructura sockaddr_in *sa4;
    estructura sockaddr_in6 *sa6;
    nulo *src;

    switch (sas->ss_family) {
        case AF_INET:
            sa4 = sumr;
            src = &(sa4->sin_addr);
            descanso;
        Caso AF_INET6:
            sa6 = ADDR;
            src = &(sa6->sin6_addr);
            descanso;
        Predeterminado:
            devuelve CERO;
    }

    inet_ntop de retorno (sas->ss_family, src, buf, tamaño);
}
```

```
45  /*
46   * Devolver una toma de escucha.
47   */
48  Int get_listener_socket(vacío)
49  {
50      Int oyente;        Descriptor de socket de escucha
51      Int sí=1;          Para setsockopt() SO_REUSEADDR, a continuación
52      Int Rv;
53
54      Estructura Sugerencias de addrinfo, *IA, *p;
55
56      Consíguenos un enchufe y átalo
57      memset(&Consejos, 0, tamañode Consejos);
58      Consejos.ai_family = AF_INET;
59      Consejos.ai_socktype = SOCK_STREAM;
60      Consejos.ai_flags = AI_PASSIVE;
61      si ((Rv = getaddrinfo(NULO, PUERTO, &Consejos, &IA)) != 0) {
62          fprintf(stderr, "Servidor de encuestas: %s\n", gai_strerror(Rv));
63          salida(1);
64      }
65
66      para(p = IA; p != NULO; p = p->ai_next) {
67          oyente = enchufe(p->ai_family, p->ai_socktype,
68                  p->ai_protocol);
69          si (oyente < 0) {
70              continuar;
71          }
72
73          Pierde el molesto mensaje de error "dirección ya en uso"
74          setsockopt(oyente, SOL_SOCKET, SO_REUSEADDR, &Sí,
75                  tamañode(Int));
76
77          si (atar(oyente, p->ai_addr, p->ai_addrlen) < 0) {
78              cerrar(oyente);
79              continuar;
80          }
81
82          quebrar;
83      }
84
85      Si llegamos aquí, significa que no quedamos atados
86      si (p == NULO) {
87          devolución -1;
88      }
89
90      freeaddrinfo(IA); Todo hecho con esto
91
92      Escuchar
93      si (escuchar(oyente, 10) == -1) {
94          devolución -1;
95      }
96
97      devolución oyente;
98  }
99
100 /*
101  * Agregue un nuevo descriptor de archivo al conjunto.
```

```
 */
vacío add_to_pfds(Estructura pollfd **PFD, Int newfd, Int *fd_count,
        Int *fd_size)
{
    Si no tenemos espacio, agregue más espacio en la matriz pfds
    si (*fd_count == *fd_size) {
        *fd_size *= 2; El doble
        *PFD = realloc(*PFD, tamañode(**PFD) * (*fd_size));
    }

    (*PFD)[*fd_count].Fd = newfd;
    (*PFD)[*fd_count].Eventos = POLÍN; Revisa listo para leer
    (*PFD)[*fd_count].revents = 0;

    (*fd_count)++;
}

/*
 * Eliminar un descriptor de archivo en un índice dado del conjunto.
 */
vacío del_from_pfds(Estructura Pfds de POLLFD[], Int Yo, Int *fd_count)
{
    Copia el del final sobre este
    PFD[Yo] = PFD[*fd_count-1];

    (*fd_count)--;
}

/*
 * Manejar las conexiones entrantes.
 */
vacío handle_new_connection(Int oyente, Int *fd_count,
        Int *fd_size, Estructura pollfd **PFD)
{
    Estructura sockaddr_storage remoteaddr; Dirección del cliente
    socklen_t addrlen;
    Int newfd; Descriptor de socket Newly accept()ed
    carbonizar remoteIP[INET6_ADDRSTRLEN];

    addrlen = tamañode remoteaddr;
    newfd = aceptar(oyente, (Estructura calcetín *)&remoteaddr,
            &addrlen);

    si (newfd == -1) {
        perror("Aceptar");
    } más {
        add_to_pfds(PFD, newfd, fd_count, fd_size);

        printf("Pollserver: Nueva conexión desde %s en el zócalo %d\n",
                inet_ntop2(&remoteaddr, remoteIP, tamañode remoteIP),
                newfd);
    }
}

/*
 * Manejar los datos regulares del cliente o los cuelgues del cliente.
 */
```

```
vacío handle_client_data(Int oyente, Int *fd_count,
        Estructura pollfd *PFD, Int *pfd_i)
{
    carbonizar Buf[256];       Búfer para datos de cliente

    Int nbytes = recv(PFD[*pfd_i].Fd, Buf, tamañode Buf, 0);

    Int sender_fd = PFD[*pfd_i].Fd;

    si (nbytes <= 0) { El cliente ha cerrado un error o una conexión
        si (nbytes == 0) {
            Conexión cerrada
            printf("Servidor de encuestas: socket %d Cuelga\n", sender_fd);
        } más {
            perror("recv");
        }

        cerrar(PFD[*pfd_i].Fd); ¡Adiós!

        del_from_pfds(PFD, *pfd_i, fd_count);

        Vuelva a examinar la ranura que acabamos de eliminar
        (*pfd_i)--;

    } más { Obtuvimos algunos buenos datos de un cliente
        printf("Servidor de encuestas: Recv de FD %d: %.*s", sender_fd,
                nbytes, Buf);
        ¡Enviar a todo el mundo!
        para(Int j = 0; j < *fd_count; j++) {
            Int dest_fd = PFD[j].Fd;

            //      Excepto el oyente y nos
            si    (dest_fd != oyente && dest_fd != sender_fd) {
                    if (send(dest_fd, nbytes, 0) == -1) {
                            buffet,
                     perror("enviar");
                }
            }
        }
    }
}

/*
 * Procesar todas las conexiones existentes.
 */
void process_connections(int oyente,    int *fd_count, int *fd_size,
        struct pollfd **pfds)
{
    for(int i = 0; Yo < *fd_count; I++) {

        Comprueba si alguien está listo leer
        para
        Si ((*dfp)[i].revents & (POLLIN | POLLHUP)) {
            ¡¡Tenemos uno!!

            Si ((*dfp)[i].fd == oyente) {
                Si somos el oyente, es una nueva conexión
                handle_new_connection(oyente, fd_count, fd_size,
                        PFD);
```

```
            } más {
                De lo contrario, somos solo un cliente habitual
                handle_client_data(oyente, fd_count, *PFD, &Yo);
            }
        }
    }
}

/*
 * Principal: crea un oyente y un conjunto de conexiones, bucle para siempre
 * Procesamiento de conexiones.
 */
Int principal(vacío)
{
    Int oyente;        Descriptor de socket de escucha

    Comience con espacio para 5 conexiones
    (Haremos realloc según sea necesario)
    Int fd_size = 5;
    Int fd_count = 0;
    Estructura pollfd *PFD = Malloc(tamañode *PFD * fd_size);

    Configurar y obtener un conector de escucha
    oyente = get_listener_socket();

    si (oyente == -1) {
        fprintf(stderr, "Error al obtener el socket de escucha\n");
        salida(1);
    }

    Agregue el oyente para establecer;
    Informe listo para leer en la conexión entrante
    PFD[0].Fd = oyente;
    PFD[0].Eventos = POLÍN;

    fd_count = 1; Para el oyente

    Pone("Pollserver: Esperando conexiones...");

    Bucle principal
    para(;;) {
        Int poll_count = encuesta(PFD, fd_count, -1);

        si (poll_count == -1) {
            perror("encuesta");
            salida(1);
        }

        Ejecutar conexiones en busca de datos para leer
        process_connections(oyente, &fd_count, &fd_size, &PFD);
    }

    Gratis(PFD);
}
```

En la siguiente sección, veremos una función similar y más antigua llamada `select()`. Tanto `select()` como `poll()`
ofrecen una funcionalidad y un rendimiento similares, y solo difieren realmente en cómo se utilizan. `select()` podría

ser un poco más portátil, pero quizás sea un poco más torpe en su uso. Elija el que más le guste, siempre que sea compatible con su sistema.

7.3 `select()`: multiplexación de E/S síncrona, de la vieja escuela

Esta función es algo extraña, pero es muy útil. Tomemos la siguiente situación: usted es un servidor y desea escuchar las conexiones entrantes, así como seguir leyendo de las conexiones que ya tiene.

No hay problema, dices, solo un `accept()` y un par de `recv()`s. ¡No tan rápido, buster! ¿Qué pasa si estás bloqueando una `llamada accept()?` ¿Cómo vas a `recv()` `datos` al mismo tiempo? "¡Use enchufes que no bloqueen!" ¡No es posible! No quieres ser un acaparador de CPU. ¿Qué, entonces?

`select()` le da el poder de monitorear varios sockets al mismo tiempo. Te dirá cuáles están listos para leer, cuáles están listos para escribir y qué sockets han levantado excepciones, si realmente quieres saberlo.

> *Una advertencia: `select()`, aunque es muy portátil, es terriblemente lento cuando se trata de un gran número de conexiones. En esas circunstancias, obtendrá un mejor rendimiento de una biblioteca de eventos como libevent[a] que intenta utilizar el método más rápido posible disponible en su sistema.*
>
> [a] https://libevent.org/

Sin más preámbulos, ofreceré 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 monitorea "conjuntos" de descriptores de archivo, en particular `readfds`, `writefds` y `exceptfds`. Si desea ver si puede leer desde la entrada estándar y algún descriptor de socket, `sockfd`, simplemente agregue los descriptores de archivo `0` y `sockfd` al conjunto `readfds`. El parámetro `numfds` debe establecerse en los valores del descriptor de archivo más alto más uno. En este ejemplo, debe establecerse en `sockfd+1`, ya que seguramente es más alto que la entrada estándar (`0`).

Cuando `select()` devuelve, `readfds` se modificará para reflejar cuál de los descriptores de archivo que seleccionó está listo para leer. Puedes probarlos con la macro `FD_ISSET()`, a continuación.

Antes de seguir avanzando, hablaré de cómo manipular estos conjuntos. Cada conjunto es del tipo `fd_set`. Las siguientes macros operan en este tipo:

Función	Descripción
`FD_SET(int fd, fd_set *set);`	Agregue `fd` al `conjunto`.
`FD_CLR(int fd, fd_set *set);`	Elimine `fd` del `conjunto`.
`FD_ISSET(int fd, fd_set *set);`	Retorna true si `fd` está en el `conjunto`.
`FD_ZERO(fd_set *siete);`	Borra todas las entradas del `conjunto`.

Finalmente, ¿qué es este extraño `struct timeval`? Bueno, a veces no quieres esperar una eternidad a que alguien te envíe algunos datos. Tal vez cada 96 segundos quieras imprimir "Still Going..." a la terminal a pesar de que no ha pasado nada. Esta estructura de tiempo le permite especificar un período de tiempo de espera. Si se excede el tiempo y `select()` aún no ha encontrado ningún descriptor de archivo listo, regresará para que pueda continuar con el procesamiento.

La `struct timeval` tiene los siguientes campos:

```
struct timeval {
    Int tv_sec;      sobras
    Int tv_usec;     Microsegundos
};
```

Solo tienes que establecer `tv_sec` en el número de segundos que se van a esperar y tv_usec en el número de microsegundos que se van a esperar. Sí, eso es _micro_seconds, no milisegundos. Hay 1.000 microsegundos en un milisegundo y 1.000 milisegundos en un segundo. Por lo tanto, hay 1.000.000 de microsegundos en un segundo. ¿Por qué es "usec"? Se supone que la "u" se parece a la letra griega μ (Mu) que usamos para "micro". Además, cuando vuelva la función, es `posible que el` *tiempo de espera* se actualice para mostrar el tiempo restante. Esto depende del tipo de Unix que esté ejecutando.

¡Yay! ¡Tenemos un temporizador de resolución de microsegundos! Bueno, no cuentes con eso. Probablemente tendrá que esperar una parte de su timeclip estándar de Unix, sin importar cuán pequeño sea el `tiempo de su estructura`.

Otras cosas de interés: Si establece los campos en su `struct timeval` en `0`, `select()` agotará el tiempo de espera de inmediato, sondeando efectivamente todos los descriptores de archivo en sus conjuntos. Si establece el parámetro `timeout` en NULL, nunca se agotará el tiempo de espera y esperará hasta que el primer descriptor de archivo esté listo. Finalmente, si no te importa esperar un determinado conjunto, puedes establecerlo en NULL en la llamada a `select()`.

El siguiente fragmento de código[3] espera 2,5 segundos para que aparezca algo en la entrada estándar:

```
/*
** select.c -- una demostración de select()
*/

#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define Descriptor de archivo STDIN 0  para entrada estándar

int main(vacío)
{
    Estructura TimeVal
    TV; fd_set readfds;

    televisión.tv_sec = 2;
    televisión.tv_usec = 500000;

    FD_ZERO(&readfds);
    FD_SET(STDIN, &readfds);

    No me importan los writefds y exceptfds:
    seleccione(STDIN+1, &readfds, NULL, NULL, &tv);

    if (FD_ISSET(STDIN, & readfds))
        printf("¡Se ha pulsado una tecla!\n");
    más
        printf("Se ha agotado el tiempo de espera.\n");

    devuelve 0;
}
```

de espera de todos modos. Ahora, algunos de ustedes podrían pensar que esta es una excelente manera de esperar datos en un socket de datagramas, y tienen razón:

[3] https://beej.us/guide/bgnet/source/examples/select.c

Podría ser. Algunos Unices pueden usar select de esta manera y otros no. Deberías ver lo que dice tu página de manual local sobre el asunto si quieres intentarlo.

Algunos Unices actualizan el tiempo en su `struct timeval` para reflejar la cantidad de tiempo que aún queda antes de un tiempo de espera. Pero otros no. No confíes en que eso ocurra si quieres ser portátil. (Uso `gettimeofday()` si necesitas realizar un seguimiento del tiempo transcurrido. Es un fastidio, lo sé, pero así son las cosas).

¿Qué sucede si un socket en el conjunto de lectura cierra la conexión? Bueno, en ese caso, `select()` devuelve con ese descriptor de socket establecido como "listo para leer". Cuando realmente hagas `recv()` desde él, `recv()` devolverá `0`. Así es como sabes que el cliente ha cerrado la conexión.

Una nota más de interés sobre `select()`: si tienes un socket que está `listen()`ing, puedes verificar si hay una nueva conexión colocando el descriptor de archivo de ese socket en el `conjunto readfds`.

Y eso, amigos míos, es un resumen rápido de la todopoderosa `función` select().

Pero, por demanda popular, aquí hay un ejemplo en profundidad. Desafortunadamente, la diferencia entre el ejemplo simple de arriba, y este de aquí es significativa. Pero eche un vistazo, luego lea la descripción que le sigue.

Este programa[4] actúa como un simple servidor de chat multiusuario. Ejecútelo ejecutándolo en una ventana, luego `telnet` a él ("`telnet hostname 9034`") desde varias otras ventanas. Cuando escribe algo en una sesión `de telnet`, debería aparecer en todas las demás.

```
/*
** selectserver.c -- un cursi servidor de chat multipersona
*/

#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>
#include <netdb.h>

PUERTO #define "9034"  puerto en el que estamos escuchando

/*
 * Convierta el socket en una cadena de dirección IP.
 * addr: struct sockaddr_in o struct sockaddr_in6
 */
const char *inet_ntop2(void *addr, char *buf, tamaño size_t)
{
    struct sockaddr_storage *sas = addr;
    estructura sockaddr_in *sa4;
    estructura sockaddr_in6 *sa6;
    nulo *src;

    switch (sas->ss_family) {
        case AF_INET:
            sa4 = sumr;
            src = &(sa4->sin_addr);
            descanso;
        Caso AF_INET6:
            sa6 = ADDR;
            src = &(sa6->sin6_addr);
```

```
            descanso;
        Predeterminado:
            devuelve CERO;
    }

    inet_ntop de retorno (sas->ss_family, src, buf, tamaño);
}

/*
 * Devolver una toma de escucha
 */
int get_listener_socket(vacío)
{
    struct addrinfo sugerencias, *ai, *p;
    Int sí=1;      para setsockopt() SO_REUSEADDR, a continuación
    int rv;
    oyente int;

    Consíguenos un socket y átalo
    memset(&hints, 0, sizeof hints);
    Pistas.ai_family = AF_UNSPEC;
    Pistas.ai_socktype = SOCK_STREAM;
    Pistas.ai_flags = AI_PASSIVE;
    if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) {
        fprintf(stderr, "selectserver: %s\n", gai_strerror(rv));
        salida(1);
    }

    for(p = ai; p != NULL; p = p->ai_next) {
        oyente = socket(p->ai_family, p->ai_socktype,
                P->ai_protocol);
        if (oyente < 0) {
            continuo;
        }

        Pierde el molesto mensaje de error "Dirección ya en uso"
        setsoccopt (oyente, SOL_SOCKET, SO_REUSEADDR, y sí,
                sizeof(int);

        if (bind(listener, p->ai_addr, p->ai_addrlen) < 0) {
            close(oyente);
            continuar;
        }

        descanso;
    }

    Si llegamos aquí, significa que no quedamos atados
    if (p == NULL) {
        fprintf(stderr, "selectserver: failed to bind\n");
        salida(2);
    }

    freeaddrinfo(AI);  todo hecho con esto

    escuchar
    if (escuchar(oyente, 10) == -1) {
```

```
93          perror("escuchar");
94          salida(3);
95      }
96
97      Devolver oyente;
98  }
99
100 /*
101  * Agregue nuevas conexiones entrantes a los conjuntos adecuados
102  */
103 void handle_new_connection(oyente int, fd_set *master, int *fdmax)
104 {
105     socklen_t addrlen;
106     Int newfd;        descriptor de socket new accept()ed
107     Estructura sockaddr_storage remoteaddr; Dirección del
108     cliente carbonizar remoteIP[INET6_ADDRSTRLEN];
109
110     addrlen = tamaño del remotoaddr;
111     newfd = aceptar(oyente,
112         (struct sockaddr *)&remoteaddr,
113         &addrlen);
114
115     if (newfd == -1) {
116         perror("aceptar");
117     } else {
118         FD_SET(newfd, maestro);  Añadir al conjunto maestro
119         if (newfd > *fdmax) {  realizar un seguimiento del máximo
120             *fdmax = newfd;
121         }
122         printf("selectserver: nueva conexión de %s en "
123             "socket %d\n",
124             inet_ntop2(&remoteaddr, remoteIP, sizeof remoteIP),
125             newfd);
126     }
127 }
128
129 /*
130  * Transmitir un mensaje a todos los clientes
131  */
132 Difusión (char *buff, int nbytes, int listener, int s,
133                 fd_set *master, int fdmax)
134 {
135     for(int j = 0; j <= fdmax; j++) {
136         ¡Envíalo a todo el mundo!
137         if (FD_ISSET(j, maestro)) {
138             excepto el oyente y nosotros mismos
139             if (j != oyente && j != s) {
140                 if (send(j, buf, nbytes, 0) == -1) {
141                     perror("enviar");
142                 }
143             }
144         }
145     }
146 }
147
148 /*
149  * Manejar los datos del cliente y los bloqueos
```

```
 */
vacío handle_client_data(Int s, Int oyente, fd_set *maestro,
                         Int fdmax)
{
    carbonizar Buf[256];       Búfer para datos de cliente
    Int nbytes;

    Controlar datos de un cliente
    si ((nbytes = recv(s, Buf, tamañode Buf, 0)) <= 0) {
        El cliente ha cerrado un error o una conexión
        si (nbytes == 0) {
            Conexión cerrada
            printf("Selectserver: socket %d Cuelga\n", s);
        } más {
            perror("recv");
        }
        cerrar(s); ¡Adiós!
        FD_CLR(s, maestro); Quitar del conjunto maestro
    } más {
        Obtuvimos algunos datos de un cliente
        emisión(Buf, nbytes, oyente, s, maestro, fdmax);
    }
}

/*
 *Principal
 */
Int principal(vacío)
{
    fd_set maestro;    Lista de descriptores de archivos maestros
    fd_set read_fds; Lista de descriptores de archivos temporales para select()
    Int fdmax;         Número máximo de descriptores de archivo

    Int oyente;        Descriptor de socket de escucha

    FD_ZERO(&maestro);    Borrar los conjuntos maestro y temporal
    FD_ZERO(&read_fds);

    oyente = get_listener_socket();

    Agregar el oyente al conjunto maestro
    FD_SET(oyente, &maestro);

    Realice un seguimiento del descriptor de archivo más grande
    fdmax = oyente; Hasta ahora, es este

    Bucle principal
    para(;;) {
        read_fds = maestro; Cópialo
        si (escoger(fdmax+1, &read_fds, NULO, NULO, NULO) == -1) {
            perror("Seleccionar");
            salida(4);
        }

        Ejecute las conexiones existentes en busca de datos
        leer
        para(Int Yo = 0; Yo <= fdmax; I++) {
```

```
            if (FD_ISSET(i, &read_fds)) {  ¡¡tenemos uno!!
                if (i == oyente)
                    handle_new_connection(i, &master, &fdmax);
                más
                    handle_client_data(I, oyente, &master, fdmax);
            }
        }
    }

    devuelve 0;
}
```

Observe que tengo dos conjuntos de descriptores de archivo en el código: `master` y `read_fds`. El primero, `master`, contiene todos los descriptores de socket que están conectados actualmente, así como el descriptor de socket que está escuchando nuevas conexiones.

La razón por la que tengo el `conjunto maestro` es que `select()` en realidad *cambia* el conjunto que le pasas para reflejar qué sockets están listos para leer. Dado que tengo que realizar un seguimiento de las conexiones de una llamada de `select()` a la siguiente, debo almacenarlas de forma segura en algún lugar. En el último minuto, copio el `maestro` en el `read_fds` y luego llamo a `select()`.

Pero, ¿no significa esto que cada vez que obtengo una nueva conexión, tengo que agregarla al `conjunto maestro`? ¡Sí! ¿Y cada vez que se cierra una conexión, tengo que eliminarla del conjunto `maestro`? Sí, lo hace.

Observe que compruebo cuándo el socket del `oyente` está listo para leer. Cuando lo es, significa que tengo una nueva conexión pendiente, y la `acepto()` y la agrego al conjunto `maestro`. Del mismo modo, cuando una conexión de cliente está lista para leerse, y `recv()` devuelve `0`, sé que el cliente ha cerrado la conexión y debo eliminarla del conjunto `maestro`.

Sin embargo, si el cliente `recv()` devuelve un valor distinto de cero, sé que se han recibido algunos datos. Así que lo consigo, y luego reviso la `lista maestra` y envío esos datos a todos los demás clientes conectados.

Y eso, amigos míos, es una descripción general menos que simple de la todopoderosa `función` select().

Nota rápida para todos los fanáticos de Linux: a veces, en raras circunstancias, el `select() de Linux` puede devolver "listo para leer" y luego no estar realmente listo para leer. ¡Esto significa que se bloqueará en el `read()` después de que el `select()` diga que no lo hará! ¡Por qué tú, pequeño...! De todos modos, la solución alternativa es establecer el `indicador O_NONBLOCK` en el socket receptor para que se produzcan errores con `EWOULDBLOCK` (que puede ignorar con seguridad si ocurre). Consulta la `página de referencia` de fcntl() para obtener más información sobre cómo configurar un socket para que no se bloquee.

Además, aquí hay una última idea adicional: hay otra función llamada `poll()` que se comporta de la misma manera que `select()`, pero con un sistema diferente para administrar los conjuntos de descriptores de archivos. ¡Compruébalo !

7.4 Manejo de `envíos` parciales

¿Recuerdas en la sección sobre `send()`, arriba, cuando dije que `send()` podría no enviar todos los bytes a los que le pediste? Es decir, quieres que envíe 512 bytes, pero devuelve 412. ¿Qué pasó con los 100 bytes restantes?

Bueno, todavía están en su pequeño búfer esperando ser enviados. Debido a circunstancias fuera de su control, el kernel decidió no enviar todos los datos en un solo trozo, y ahora, amigo mío, depende de usted hacer que los datos salgan a la luz.

También podrías escribir una función como esta para hacerlo:

```
#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 nos quedan por enviar
    int n;

    while(total < *len) {
        n = enviar(s, buf+total, bytesleft,
        0); if (n == -1) { descanso; }
        total += n;
        bytesleft -= n;
    }

    *len = total;  Número de devolución realmente enviado aquí

    return n==-1?-1:0;  return -1 en caso de error, 0 en caso
}
```

En este ejemplo, `s` es el socket al que desea enviar los datos, `buf` es el búfer que contiene los datos y `len` es un puntero a un `int` que contiene el número de bytes en el búfer.

La función devuelve `-1` en caso de error (y `errno` todavía se establece desde la llamada a `send()`). Además, el número de bytes realmente enviados se devuelve en `len`. Este será el mismo número de bytes que le pidió que enviara, a menos que haya habido un error. `sendall()` hará todo lo posible, resoplando y resoplando, para enviar los datos, pero si hay un error, se pone en contacto contigo de inmediato.

Para completar, este es un ejemplo de llamada a la función:

```
char buf[10] = "¡Beej!";
int len;

len = strlen(buf);
if (sendall(s, buf, &len) == -1) {
    perror("sendall");
    printf("¡Solo enviamos %d bytes debido al error!\n", len);
}
```

¿Qué sucede en el extremo del receptor cuando llega parte de un paquete? Si los paquetes son de longitud variable, ¿cómo sabe el receptor cuándo termina un paquete y comienza otro? Sí, los escenarios del mundo real son un verdadero dolor de cabeza. Probablemente tenga *que encapsular* (¿recuerda eso de la sección de encapsulación de datos al principio?) ¡Sigue leyendo para conocer los detalles!

7.5 Serialización: cómo empaquetar datos

Es bastante fácil enviar datos de texto a través de la red, pero ¿qué sucede si desea enviar algunos datos "binarios" como `ints o floats? Resulta que tienes algunas opciones.`

1. Convierte el número en texto con una función como `sprintf(),` luego envía el texto. El receptor analizará el texto de nuevo en un número usando una función como `strtol()`.
2. Simplemente envíe los datos sin procesar, pasando un puntero a los datos para `enviar()`.
3. Codifique el número en una forma binaria portátil. El receptor lo

decodificará. ¡Adelanto! ¡Solo esta noche!

[*Se levanta el telón*]

Beej dice: "¡Prefiero el Método Tres,

arriba!" [*EL FIN]*

(Antes de comenzar esta sección en serio, debo decirles que hay bibliotecas para hacer esto, y rodar la suya propia y seguir siendo portátil y libre de errores es todo un desafío. Así que busque y haga su tarea antes de decidirse a implementar esto usted mismo. Incluyo la información aquí para aquellos que tengan curiosidad por saber cómo funcionan cosas como esta).

En realidad, todos los métodos, anteriores, tienen sus inconvenientes y ventajas, pero, como dije, en general, prefiero el tercer método. Primero, sin embargo, hablemos de algunos de los inconvenientes y ventajas de los otros dos.

El primer método, codificar los números como texto antes de enviarlos, tiene la ventaja de que puede imprimir y leer fácilmente los datos que llegan por cable. A veces, un protocolo legible por humanos es excelente para usar en una situación que no requiere un uso intensivo del ancho de banda, como con Internet Relay Chat (IRC)[5]. Sin embargo, tiene la desventaja de que es lento de convertir y los resultados casi siempre ocupan más espacio que el número original.

Método dos: pasar los datos sin procesar. Esto es bastante fácil (¡pero peligroso!): simplemente tome un puntero a los datos para enviar y llame a enviar con él.

```
doble d = 3490.15926535;

enviar(s, &d, tamañode d, 0); /* PELIGRO... ¡no portátil! */
```

El receptor lo obtiene de la siguiente manera:

```
Doble D;

recv(s, &d, tamañode d, 0); /* PELIGRO... ¡no portátil! */
```

Rápido, sencillo, ¿qué es lo que no te gusta? Bueno, resulta que no todas las arquitecturas representan un `double` (o `int` para el caso) con la misma representación de bits o incluso el mismo orden de bytes. El código es decididamente no portátil. (Oye, tal vez no necesites portabilidad, en cuyo caso esto es agradable y rápido).

Al empaquetar tipos enteros, ya hemos visto cómo las funciones de la clase `htons()` pueden ayudar a mantener las cosas portátiles al transformar los números en orden de bytes de red, y cómo eso es lo correcto. Desafortunadamente, no hay funciones similares para los tipos flotantes. ¿Se ha perdido toda esperanza?

¡No temas! (¿Tuviste miedo allí por un segundo? ¿No? ¿Ni siquiera un poco?) Hay algo que podemos hacer: podemos empaquetar (o "marshal", o "serializar", o uno de los miles de millones de otros nombres) los datos en un formato binario conocido que el receptor puede descomprimir en el lado remoto.

¿A qué me refiero con "formato binario conocido"? Bueno, ya hemos visto el ejemplo de `htons()`, ¿verdad? Cambia (o "codifica", si quieres pensarlo de esa manera) un número de cualquiera que sea el formato de host en el orden de bytes de red. Para invertir (descodificar) el número, el receptor llama a `ntohs()`.

Pero, ¿no acabo de terminar de decir que no había tal función para otros tipos no enteros? Sí. Así es. Y dado que no hay una forma estándar en C de hacer esto, es un poco complicado (ese juego de palabras gratuito para los fanáticos de Python).

Lo que hay que hacer es empaquetar los datos en un formato conocido y enviarlo por cable para su decodificación. Por ejemplo, para empacar `flotadores, aquí hay` algo rápido y sucio con mucho margen de mejora[6]:

```
1  #include <stdint.h>
2
3  uint32_t htonf(flotador f)
4  {
5      uint32_t p;
6      uint32_t signo;
7
8      if (f < 0) { signo = 1; f = -f; }
```

[5]https://en.wikipedia.org/wiki/Internet_Relay_Chat

[6]https://beej.us/guide/bgnet/source/examples/pack.c

```
    else { signo = 0; }

    Parte entera y signo
    p = ((((uint32_t)f)&0x7fff)<<16) | (signo<<31);

    fracción
    p |= (uint32_t)(((f - (int)f) * 65536.0f))&0xffff;

    retorno p;
}

flotador ntohf(uint32_t p)
{
    flotador f = ((p>>16)&0x7fff);  Parte entera
    f += (p&0xffff) / 65536.0f;  fracción

    if (((p>>31)&0x1) == 0x1) { f = -f; }  Conjunto de bits de signo

    retorno f;
}
```

El código anterior es una especie de implementación ingenua que almacena un `float` en un número de 32 bits. La parte alta
(31) se utiliza para almacenar el signo del número ("1" significa negativo), y los siguientes siete bits (30-16) se utilizan para almacenar la parte del número entero del `flotador`. Finalmente, los bits restantes (15-0) se utilizan para almacenar la parte fraccionaria del número.

El uso es bastante sencillo:

```
#include <stdio.h>

int main(vacío)
{
    flotador f = 3.1415926, f2;
    uint32_t netf;

    netf = htonf(f);  Convertir a la forma "Red"
    f2 = ntohf(netf);  Volver a probar

    printf("Original: %f\n", f);         // 3.141593
    printf(" Red: 0x%08X\n", netf);  0x0003243F
    printf("Desempacado: %f\n", F2);     // 3.141586

    devuelve 0;
}
```

En el lado positivo, es pequeño, simple y rápido. En el lado negativo, no es un uso eficiente del espacio y el alcance está severamente restringido: intente almacenar un número mayor que 32767 allí y no será muy feliz. También puede ver en el ejemplo anterior que los últimos dos decimales no se conservan correctamente.

¿Qué podemos hacer en su lugar? Bueno, *el* estándar para almacenar números de coma flotante se conoce como IEEE-754[7]. La mayoría de las computadoras usan este formato internamente para hacer matemáticas de coma flotante, por lo que en esos casos, estrictamente hablando, no sería necesario realizar la conversión. Pero si quieres que tu código fuente sea portátil, esa es una suposición que no necesariamente puedes hacer. (Por otro lado, si quieres que las cosas sean rápidas, ¡debes optimizarlo en plataformas que no necesiten hacerlo! Eso es lo que `hacen htons()` y los de su calaña).

A continuación, se muestra un código que codifica flotantes y dobles en el formato IEEE-754[8]. (En su mayoría, no codifica NaN o Infinity, pero podría modificarse para hacer eso).

[7] https://en.wikipedia.org/wiki/IEEE_754 [8] https://beej.us/guide/bgnet/source/examples/ieee754.c

```
#define pack754_32(f) (paquete754((f), 32, 8))
#define pack754_64(f) (paquete754((f), 64, 11))
#define unpack754_32(Yo) (desembalar754((Yo), 32, 8))
#define unpack754_64(Yo) (desembalar754((Yo), 64, 11))

uint64_t paquete754(Doble largo f, Unsigned Bits, Unsigned Expbits)
{
    Doble largo fnorm;
    Int turno;
    largo, largo firmar, Exp, Significación;

    -1 para el bit de signo
    Unsigned significandbits = Bits - Expbits - 1;

    si (f == 0.0) devolución 0; Quita este caso especial del camino

    Marque el signo y comience la normalización
    si (f < 0) { firmar = 1; fnorm = -f; }
    más { firmar = 0; fnorm = f; }

    Obtén la forma normalizada de f y rastrea el exponente
    turno = 0;
    mientras(fnorm >= 2.0) { fnorm /= 2.0; Mayús++; }
    mientras(fnorm < 1.0) { fnorm *= 2.0; turno--; }
    fnorm = fnorm - 1.0;

    Calcular la forma binaria (no flotante) de los datos significativos
    Significación = fnorm * ((1LL<<significandbits) + 0.5f);

    Obtener el exponente sesgado
    Exp = turno + ((1<<(Expbits-1)) - 1); Cambio + Sesgo

    Devuelve la respuesta final
    devolución (signo<<(bits-1)) | (exp<<(bits-Expbits-1)) | Significación;
}

Doble largo desembalar754(uint64_t Yo, Unsigned Bits, Unsigned Expbits)
{
    Doble largo resultado;
    largo, largo turno;
    Unsigned predisposición;

    -1 para el bit de signo
    Unsigned significandbits = Bits - Expbits - 1;

    si (Yo == 0) devolución 0.0;

    Tira de la significación
    resultado = (Yo&((1LL<<significandbits)-1)); máscara
    resultado /= (1LL<<significandbits); Convertir de nuevo a flotante
    resultado += 1.0f; Agregue el uno de nuevo en

    Tratar con el exponente
    predisposición = (1<<(Expbits-1)) - 1;
    turno = ((i>>significandbits)&((1LL<<expbits)-1)) - predisposición;
    mientras(turno > 0) { resultado *= 2.0; turno--; }
    mientras(turno < 0) { resultado /= 2.0; Mayús++; }
```

```
58
59      Fírmalo
60      resultado *= (i>>(bits-1))&1? -1.0:
61      1.0;
62
63  }
```

Puse algunas macros útiles en la parte superior para empaquetar y desempaquetar números de 32 bits (probablemente un `float`) y de 64 bits (probablemente un `doble`), pero la función `pack754()` podría llamarse directamente y decirle que codifique `bits` de datos (`expbits` de los cuales están reservados para el exponente del número normalizado).

A continuación, se muestra un ejemplo de uso:

```
1
2   #include <stdio.h>
3   #include <stdint.h>  define uintN_t tipos
4   #include <inttypes.h>  define macros PRIx
5
6   int main(vacío)
7   {
8       flotador f = 3.1415926, f2;
9       doble d = 3,14159265358979323, d2;
10      uint32_t fi;
11      uint64_t de;
12
13      fi = pack754_32(f);
14      f2 = unpack754_32(fi);
15
16      di = pack754_64(d);
17      d2 = unpack754_64(di);
18
19      printf("float antes de : %.7f\n", f);
20      printf("float encoded: 0x%08" PRIx32 "\n", fi);
21      printf("flotar después de : %.7f\n\n", f2);
22
23      printf("doble antes de : %.20lf\n", d);
24      printf("doble codificado: 0x%016" PRIx64 "\n", di);
25      printf("doble después de : %.20lf\n", d2);
26
27      devuelve 0;
28  }
```

El código anterior produce este resultado:

```
float antes de : 3.1415925
float codificado:
0x40490FDA float después :
3.1415925

doble antes : 3.14159265358979311600
doble codificado: 0x400921FB54442D18
```

Otra pregunta que puede tener es ¿cómo se empaquetan `las estructuras?` Desafortunadamente para ti, el compilador es libre de poner relleno por todas partes en una `estructura`, y eso significa que no puedes enviar todo de manera portátil a través del cable en un solo fragmento. (¿No te estás cansando de escuchar "no puedo hacer esto", "no puedo hacer aquello"? ¡Arrepentido! Citando a un amigo: "Cada vez que algo sale mal, siempre culpo a Microsoft". Puede que esto no sea culpa de Microsoft, es cierto, pero la afirmación de mi amigo es completamente cierta).

Volviendo a esto: la mejor manera de enviar la `estructura` a través del cable es empaquetar cada campo de forma independiente y luego

Desempaquételos en la `estructura` cuando lleguen al otro lado.

Eso es mucho trabajo, es lo que estás pensando. Sí. Una cosa que puede hacer es escribir una función auxiliar para que le ayude a empaquetar los datos. ¡Será divertido! ¡Realmente!

En el libro *The Practice of Programming*[9] de Kernighan y Pike, implementan `funciones similares a` printf() llamadas `pack()` y `unpack()` que hacen exactamente esto. Enlazaría con ellos, pero aparentemente esas funciones no están en línea con el resto de la fuente del libro.

(*La Práctica de la Programación* es una excelente lectura. Zeus salva a un gatito cada vez que se lo recomiendo).

En este punto, voy a dejar caer un puntero a una implementación de Protocol Buffers en C[10] que nunca he usado, pero parece completamente respetable. Los programadores de Python y Perl querrán revisar las funciones `pack()` y `unpack()` de su lenguaje para lograr lo mismo. Y Java tiene una gran interfaz serializable que se puede usar de manera similar.

Pero si quieres escribir tu propia utilidad de empaquetamiento en C, el truco de K&P es usar listas de argumentos variables para crear `funciones tipo printf()` para construir los paquetes. Aquí hay una versión que cociné[11] por mi cuenta basada en eso, que espero que sea suficiente para darle una idea de cómo puede funcionar tal cosa.

(Este código hace referencia a las `funciones pack754(),` anteriores. Las funciones `packi*()` funcionan como las conocidas
`htons(),` excepto que se empaquetan en una `matriz char` en lugar de en otro número entero).

```c
#include <stdio.h>
#include <ctype.h>
#include <stdarg.h>
#include <string.h>

/*
** packi16() -- almacena un int de 16 bits en un búfer de caracteres (como
htons())
*/
void packi16(unsigned char *buf, unsigned int i)
{
    *buf++ = i>>8; *buf++ = y;
}

/*
** packi32() -- almacena un int de 32 bits en un búfer char (como htonl())
*/
void packi32(unsigned char *buf, unsigned long int i)
{
    *buf++ = i>>24; *buf++ = y>>16;
    *buf++ = i>>8; *buf++ = y;
}

/*
** packi64() -- almacena un int de 64 bits en un búfer de caracteres (como
htonl())
*/
void packi64(unsigned char *buf, unsigned long long int i)
{
    *buf++ = i>>56; *buf++ = i>>48;
    *buf++ = i>>40; *buf++ = y>>32;
    *buf++ = i>>24; *buf++ = y>>16;
    *buf++ = i>>8; *buf++ = y;
}
```

[9] https://beej.us/guide/bgtpop

[10] https://github.com/protobuf-c/protobuf-c

[11] https://beej.us/guide/bgnet/source/examples/pack2.c

```
** unpacki16() -- desempaqueta un int de 16 bits de un búfer char (como
**                 ntohs())
*/
Int desembalajei16(char sin firmar *Buf)
{
    unsigned int i2 = ((unsigned int)Buf[0]<<8) | Buf[1];
    Int Yo;

    Cambiar números sin firmar a firmados
    si (i2 <= 0x7fffu) { Yo = i2; }
    más { Yo = -1 - (unsigned int)(0xffffu - i2); }

    devolución Yo;
}

/*
** unpacku16() -- desempaqueta un unsigned de 16 bits de un búfer char (como
**                 ntohs())
*/
unsigned int unpacku16(char sin firmar *Buf)
{
    devolución ((unsigned int)Buf[0]<<8) | Buf[1];
}

/*
** unpacki32() -- desempaqueta un int de 32 bits de un búfer char (como
**                 ntohl())
*/
long int Desembalari32(char sin firmar *Buf)
{
    unsigned long int i2 = ((unsigned long int)Buf[0]<<24) |
                           ((unsigned long int)Buf[1]<<16) |
                           ((unsigned long int)Buf[2]<<8) |
                           Buf[3];
    long int Yo;

    Cambiar números sin firmar a firmados
    si (i2 <= 0x7fffffffu) { Yo = i2; }
    más { Yo = -1 - (long int)(0xffffffffu - i2); }

    devolución Yo;
}

/*
** unpacku32() -- desempaqueta un unsigned de 32 bits de un búfer de caracteres (como
**                 ntohl())
*/
unsigned long int unpacku32(char sin firmar *Buf)
{
    devolución ((unsigned long int)Buf[0]<<24) |
           ((unsigned long int)Buf[1]<<16) |
           ((unsigned long int)Buf[2]<<8) |
           Buf[3];
}

/*
** unpacki64() -- desempaqueta un int de 64 bits de un búfer char (como
```

```
92  **                ntohl())
93  */
94  long long int unpacki64(unsigned char *buf)
95  {
96      unsigned long long int i2 =
97          ((sin firmar long long                |
            int)buf[0]<<56)
98          ((sin firmar, largo, largo,           |
            int)buf[1]<<48)
99          ((sin firmar, largo, largo,           |
            int)buf[2]<<40)
100         ((sin firmar, largo, largo,           |
            int)buf[3]<<32)
101         ((sin firmar long long                |
            int)buf[4]<<24)
102         ((sin firmar long long                |
            int)buf[5]<<16)
103         ((unsigned long long int)buf[6]<<8)   |
104         buf[7];
105     largo largo int i;
106
107     Cambiar números sin firmar a firmados
108     if (i2 <= 0x7fffffffffu) { i = i2;        }
109     else { yo  -1 -(largo largo int)(0xffffffffffffu    - i2); }
               =
110
111     devolución Yo;
112 }
113
114 /*
115 ** unpacku64() -- desempaqueta un unsigned de 64 bits de un búfer char (como
116 **                ntohl())
117 */
118 unsigned long long int desembalarU64(char sin firmar *Buf)
119 {
120     devolución ((unsigned long long int)Buf[0]<<56) |
121             ((unsigned long long int)Buf[1]<<48) |
122             ((unsigned long long int)Buf[2]<<40) |
123             ((unsigned long long int)Buf[3]<<32) |
124             ((unsigned long long int)Buf[4]<<24) |
125             ((unsigned long long int)Buf[5]<<16) |
126             ((unsigned long long int)Buf[6]<<8) |
127             Buf[7];
128 }
129
130 /*
131 ** pack() -- almacena los datos dictados por la cadena de formato en el búfer
132 **
133 **   bits |firmado  Unsigned   flotar   cuerda
134 **   -----+----------------------------------
135 **      8 |   c        C
136 **     16 |   h        H         f
137 **     32 |   l        L         d
138 **     64 |   q        Q         g
139 **      - |                              s
140 **
141 ** (la longitud sin signo de 16 bits se antepone automáticamente a las cadenas)
142 */
143
144 unsigned int empaquetar(char sin firmar *Buf, carbonizar *formato, ...)
145 {
146     va_list Ap;
147
148     char firmado c;             8 bits
```

```
    carácter C sin signo;

    Int h;                      16 bits
    int H sin signo;

    long int l;                 32 bits
    sin signo largo int L;

    long long int q;            64 bits
    sin signo long long int Q;

    flotar f;                   Flotadores
    Doble D;
    Doble G larga;
    sin firmar long long int fhold;

    carbonizar *s;              instrumentos de cuerda
    unsigned int len; tamaño

    int sin signo  = 0;

    va_start(AP, formato);

    for(; *formato != '\0'; formato++) {
        switch(*formato) {
        Caso 'C':  8 bits
            tamaño += 1;
         c = (carácter firmado)va_arg(ap, int);  Promovido
            *buf++ = c;
            descanso;

        caso 'C':  8 bits sin firmar
            tamaño += 1;
    C = (carácter sin signo)va_arg(ap, entero sin signo); // Promovid
                                                             o
            *buf++ = C;
            descanso;

        Caso 'H':  16 bits
            tamaño += 2;
            h = va_arg(ap, int);
            Packi16(BUF, H);
            buf += 2;
            descanso;

        caso 'H':  16 bits sin firmar
            tamaño += 2;
            H = va_arg(ap, entero sin signo);
            packi16(carne de res, H);
            buf += 2;
            descanso;

        Caso 'L':  32 bits
            tamaño += 4;
            l = va_arg(ap, entero largo);
            packi32(buf, l);
            buf += 4;
            descanso;
```

```
206
207         caso 'L':  32 bits sin firmar
208             tamaño += 4;
209             L = va_arg(ap, entero largo sin
                signo);
210             packi32(buf, L);
211             buf += 4;
212             descanso;
213
214         Caso 'Q':  64 bits
215             tamaño += 8;
216             q = va_arg(ap, largo largo int);
217             Packi64(buf, q);
218             buf += 8;
219             descanso;
220
221         caso 'Q':  64 bits sin firmar
222             tamaño += 8;
223             Q = va_arg(ap, sin signo largo     int);
                largo
224             packi64(buf, Q);
225             buf += 8;
226             descanso;
227
228         Caso 'F':  Float-16
229             tamaño += 2;
230             f = (flotante)va_arg(ap, doble);   Promovido
                //
231             fhold = pack754_16(f);  convertir  Pa  IEEE  754
                                                   ra
232             packi16(buf, fhold);
233             buf += 2;
234             descanso;
235
236         Caso 'D':  Float-32
237             tamaño += 4;
238             d = va_arg(ap, doble);
239             fhold = pack754_32(d);  convertir  Pa  IEEE  754
                                                   ra
240             packi32(buf, fhold);
241             buf += 4;
242             descanso;
243
244         Caso 'G':  Float-64
245             tamaño += 8;
246             g = va_arg(ap, doble largo);
247             fhold = pack754_64(g);  convertir  Pa  IEEE  754
                                                   ra
248             packi64(buf, fhold);
249             buf += 8;
250             descanso;
251
252         caso 's':  cadena
253             s = va_arg(ap, char*);
254             len = strlen(s);
255             tamaño += len + 2;
256             Packi16(buf, len);
257             buf += 2;
258             memcpy(buf, s, len);
259             buf += len;
260             descanso;
261         }
262     }
```

```
263
264     va_end(AP);
265
266     tamaño de devolución;
267 }
268
269 /*
270 ** unpack() -- desempaqueta los datos dictados por la     en el
    cadena de formato
271 **             búfer
272 **
273 **    bits |firmado   Unsigned    flotar   cuerda
274 **    -----+----------------------------------
275 **       8 |    c         C
276 **      16 |    h         H          f
277 **      32 |    l         L          d
278 **      64 |    q         Q          g
279 **       - |                                 s
280 **
281 ** (la cadena se extrae en función de su longitud almacenada, pero 's' puede ser
282 ** antepuesto con una longitud máxima)
283 */
284 vacío desempaquetar(char sin firmar *Buf, carbonizar *formato, ...)
285 {
286     va_list Ap;
287
288     char firmado *c;              8 bits
289     char sin firmar *C;
290
291     Int *h;                       16 bits
292     unsigned int *H;
293
294     long int *l;                  32 bits
295     unsigned long int *L;
296
297     long long int *q;             64 bits
298     unsigned long long int *Q;
299
300     flotar *f;                    Flotadores
301     doble *d;
302     Doble largo *g;
303     unsigned long long int Fhold;
304
305     char *s;
306     unsigned int len, maxstrlen=0, count;
307
308     va_start(AP, formato);
309
310     for(; *formato != '\0'; formato++) {
311         switch(*formato) {
312         Caso 'C':  8 bits
313             c = va_arg(ap, firmado char*);
314             if (*buf <= 0x7f) { *c = *buf;}  Volver a firmar
315             else { *c = -1 - (unsigned char)(0xffu - *buf);  }
316             buf++;
317             descanso;
318
319         caso 'C':  8 bits sin firmar
```

```
            C = va_arg(Ap, char sin firmar*);
            *C = *buf++;
            quebrar;

        caso 'h': 16 bits
            h = va_arg(Ap, Int*);
            *h = desembalajei16(Buf);
            Buf += 2;
            quebrar;

        caso 'H': 16 bits sin firmar
            H = va_arg(Ap, unsigned int*);
            *H = unpacku16(Buf);
            Buf += 2;
            quebrar;

        caso 'l': 32 bits
            l = va_arg(Ap, long int*);
            *l = Desembalari32(Buf);
            Buf += 4;
            quebrar;

        caso 'L': 32 bits sin firmar
            L = va_arg(Ap, unsigned long int*);
            *L = unpacku32(Buf);
            Buf += 4;
            quebrar;

        caso 'Q': 64 bits
            q = va_arg(Ap, long long int*);
            *q = desembalar64(Buf);
            Buf += 8;
            quebrar;

        caso 'Q': 64 bits sin firmar
            Q = va_arg(Ap, unsigned long long int*);
            *Q = desembalarU64(Buf);
            Buf += 8;
            quebrar;

        caso 'f': flotar
            f = va_arg(Ap, flotar*);
            Fhold = unpacku16(Buf);
            *f = unpack754_16(Fhold);
            Buf += 2;
            quebrar;

        caso 'd': flotador-32
            d = va_arg(Ap, doble*);
            Fhold = unpacku32(Buf);
            *d = unpack754_32(Fhold);
            Buf += 4;
            quebrar;

        caso 'g': flotador-64
            g = va_arg(Ap, Doble largo*);
            Fhold = desembalarU64(Buf);
```

```
            *g = unpack754_64(fhold);
            buf += 8;
            descanso;

        caso 's':  cadena
            s = va_arg (ap, char*);
            len = unpacku16
            (buffet); buffet += 2;
            if (maxstrlen  > 0 && len > maxstrlen)
                count = maxstrlen - 1;
            más
                conteo = len;
            memcpy(s, buf, conde);
            s[recuento] = '\0';
            buf += len;
            camioneta;

        Predeterminado:
            if (isdigit(*format)) {  track max str len
                maxstrlen = maxstrlen * 10 + (*format-'0');
            }
        }

        Si ( !isdigit(*formato)) maxstrlen = 0;
    }

    va_end(AP);
}
```

Y aquí hay un programa de demostración[12] del código anterior que empaqueta algunos datos en `buf` y luego los desempaqueta en variables. Tenga en cuenta que al llamar a `unpack()` con un argumento de cadena (especificador de formato "`s`"), es aconsejable poner un recuento de longitud máxima delante de él para evitar una saturación del búfer, por ejemplo, "`96s`". Tenga cuidado al descomprimir los datos que obtiene a través de la red: un usuario malintencionado podría enviar paquetes mal construidos en un esfuerzo por atacar su sistema.

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

Si tiene un compilador C23 #if
STDC_VERSION_________ >= 202311L
#include <stdfloat.h>
#else
De lo contrario, definamos el nuestro.
¡Varía para diferentes arquitecturas! Pero es probable que:
Typedef float32_t flotante;
typedef doble float64_t;
#endif

int main(vacío)
{
    uint8_t buf[1024];
    int8_t magia;
    int16_t conteo de
    monos; int32_t
    altitud;
```

[12] https://beej.us/guide/bgnet/source/examples/pack2.c

```
    float32_t factor absurdo;
    char *s = "¡Gran Zot sin paliativos! ¡Has encontrado el Bastón
    Rúnico!"; char S2[96];
    int16_t tamaño del paquete, PS2;

    tamaño del paquete = paquete(buffet, "chhlsf", (int8_t)'B',
            (int16_t)0, (int16_t)37, (int32_t)-5, s, (float32_t)-
            3490.6677);
    Packi16(buf+1, tamaño del paquete);  Almacenar el tamaño del paquete para
    kicks

    printf("el paquete es %" PRId32 " bytes\n", tamaño del

    paquete); unpack(buf, "chhl96sf", &magic, &ps2, &monkeycount,
    printf("'%c' %" PRId32" %" PRId16 " %" PRId32
            " \"%s\" %f\n", magia, ps2, monkeycount,
    &altitude,
            altitud, s2, factor absurdo);
            S2, &AbsurdityFactor);
}
```

Ya sea que lances tu propio código o uses el de otra persona, es una buena idea tener un conjunto general de rutinas de empaquetado de datos para mantener los errores bajo control, en lugar de empaquetar cada bit a mano cada vez.

Al empaquetar los datos, ¿cuál es un buen formato para usar? Excelente pregunta. Afortunadamente, RFC 4506[13], el Estándar de Representación de Datos Externos, ya define formatos binarios para un montón de tipos diferentes, como tipos de punto flotante, tipos enteros, matrices, datos sin procesar, etc. Te sugiero que te conformes con eso si vas a rodar los datos tú mismo. Pero no estás obligado a hacerlo. La Policía de Paquetes no está justo afuera de su puerta. Al menos, no creo *que* lo sean.

En cualquier caso, codificar los datos de una forma u otra antes de enviarlos es la forma correcta de hacer las cosas.

7.6 Hijo de la encapsulación de datos

De todos modos, ¿qué significa realmente encapsular datos? En el caso más simple, significa que colocará un encabezado allí con alguna información de identificación o una longitud de paquete, o ambas.

¿Cómo debería verse tu encabezado? Bueno, son solo algunos datos binarios que representan lo que creas que es necesario para completar tu proyecto.

Uau. Eso es vago.

Bien. Por ejemplo, supongamos que tiene un programa de chat multiusuario que utiliza `SOCK_STREAMs`. Cuando un usuario escribe ("dice") algo, es necesario transmitir al servidor dos piezas de información: lo que se dijo y quién lo dijo.

¿Hasta ahora, bien? "¿Cuál es el problema?", te estás preguntando.

El problema es que los mensajes pueden ser de diferentes longitudes. Una persona llamada "tom" podría decir: "Hola", y otra persona llamada "Benjamin" podría decir: "Hola chicos, ¿qué pasa?"

Así que envías todo esto a los clientes a medida que llega. El flujo de datos salientes tiene el siguiente aspecto:

```
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 sucesivamente. ¿Cómo sabe el cliente cuándo se inicia un mensaje y se detiene otro? Podría, si quisiera, hacer que todos los mensajes tuvieran la misma longitud y simplemente llamar al `sendall()` que implementamos anteriormente. ¡Pero eso desperdicia ancho de banda! No queremos `enviar ()` 1024 bytes solo para que "tom" pueda decir "Hola".

Por lo tanto, encapsulamos los datos en un pequeño encabezado y estructura de paquetes. Tanto el cliente como el servidor saben cómo empaquetar y desempaquetar (a veces denominados "marshal" y "unmarshal") estos datos. No mires ahora, pero estamos empezando a definir un *protocolo* que describe cómo se comunican un cliente y un servidor.

[13] https://tools.ietf.org/html/rfc4506

En este caso, supongamos que el nombre de usuario tiene una longitud fija de 8 caracteres, rellenada con `'\0'`. Y luego supongamos que los datos tienen una longitud variable, hasta un máximo de 128 caracteres. Echemos un vistazo a una estructura de paquetes de muestra que podríamos usar en esta situación:

1. `len` (1 byte, sin signo): la longitud total del paquete, contando el nombre de usuario de 8 bytes y los datos de chat.
2. `name` (8 bytes): el nombre del usuario, rellenado con NUL si es necesario.
3. `chatdata` (n-bytes): los datos en sí, no más de 128 bytes. La longitud del paquete debe calcularse como la longitud de estos datos más 8 (la longitud del campo de nombre, arriba).

¿Por qué elegí los límites de 8 bytes y 128 bytes para los campos? Los saqué del aire, asumiendo que serían lo suficientemente largos. Tal vez, sin embargo, 8 bytes sea demasiado restrictivo para sus necesidades, y puede tener un campo de nombre de 30 bytes, o lo que sea. La elección depende de ti.

Usando la definición de paquete anterior, 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 es similar:

```
   18     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 se almacena en el orden de bytes de la red, por supuesto. En este caso, es solo un byte, por lo que no importa, pero en términos generales, querrá que todos sus enteros binarios se almacenen en orden de bytes de red en sus paquetes).

Cuando envíes estos datos, debes estar seguro y usar un comando similar a `sendall()`, arriba, para que sepas que todos los datos se envían, incluso si se necesitan varias llamadas a `send()` para sacarlos todos.

Del mismo modo, cuando recibe estos datos, debe hacer un poco de trabajo adicional. Para estar seguro, debes asumir que podrías recibir un paquete parcial (como tal vez recibimos "`18 42 65 6E 6A`" de Benjamin, arriba, pero eso es todo lo que obtenemos en esta llamada a `recv()`). Necesitamos llamar a `recv()` una y otra vez hasta que el paquete se reciba por completo.

¿Pero cómo? Bueno, sabemos el número de bytes que necesitamos recibir en total para que el paquete esté completo, ya que ese número está agregado en la parte delantera del paquete. También sabemos que el tamaño máximo del paquete es 1 + 8 + 128, o 137 bytes (porque así es como definimos el paquete).

En realidad, hay un par de cosas que puedes hacer aquí. Como sabes que cada paquete comienza con una longitud, puedes llamar a `recv()` solo para obtener la longitud del paquete. Luego, una vez que tenga eso, puede llamarlo nuevamente especificando exactamente la longitud restante del paquete (posiblemente repetidamente para obtener todos los datos) hasta que tenga el paquete completo. La ventaja de este método es que solo necesitas un búfer lo suficientemente grande para un paquete, mientras que la desventaja es que necesitas llamar a `recv()` al menos dos veces para obtener todos los datos.

Otra opción es simplemente llamar a `recv()` y decir que la cantidad que estás dispuesto a recibir es el número máximo de bytes en un paquete. Luego, lo que sea que obtenga, péguelo en la parte posterior de un búfer y finalmente verifique si el paquete está completo. Por supuesto, es posible que obtenga algo del próximo paquete, por lo que deberá tener espacio para eso.

Lo que puede hacer es declarar una matriz lo suficientemente grande para dos paquetes. Esta es su matriz de trabajo donde reconstruirá los paquetes a medida que lleguen.

Cada vez que `recv()` datos, los agregará al búfer de trabajo y verificará si el paquete está completo. Es decir, el número de bytes del búfer es mayor o igual que la longitud especificada en el encabezado (+1, porque la longitud del encabezado no incluye el byte de la longitud en sí). Si el número de bytes en el búfer es menor que 1, el paquete no está completo, obviamente. Sin embargo, debe hacer un caso especial para esto, ya que el primer byte es basura y no puede confiar en él para la longitud correcta del paquete.

Una vez que el paquete esté completo, puede hacer con él lo que desee. Úselo y elimínelo de su búfer de trabajo.

¡Vaya! ¿Ya estás haciendo malabarismos con eso en tu cabeza? Bueno, aquí está el segundo de los dos golpes: es posible que hayas leído más allá del final de un paquete y en el siguiente en una sola `llamada recv().` Es decir, tiene un búfer de trabajo con un paquete completo y una parte incompleta del siguiente paquete. Maldita sea. (Pero esta es la razón por la que hizo que su búfer de trabajo fuera lo suficientemente grande como para contener *dos* paquetes, ¡en caso de que esto sucediera!)

Dado que conoce la longitud del primer paquete del encabezado y ha estado realizando un seguimiento del número de bytes en el búfer de trabajo, puede restar y calcular cuántos de los bytes en el búfer de trabajo pertenecen al segundo paquete (incompleto). Cuando hayas manejado el primero, puedes borrarlo del búfer de trabajo y mover el segundo paquete parcial hacia abajo hasta el frente del búfer para que todo esté listo para el siguiente `recv()`.

(Algunos de ustedes, lectores, notarán que en realidad mover el segundo paquete parcial al comienzo del búfer de trabajo lleva tiempo, y el programa se puede codificar para que no lo requiera mediante el uso de un búfer circular. Desafortunadamente para el resto de ustedes, una discusión sobre los búferes circulares está más allá del alcance de este artículo. Si todavía tienes curiosidad, toma un libro de estructuras de datos y comienza desde allí).

Nunca dije que fuera fácil. Ok, dije que era fácil. Y lo es; Solo necesitas práctica y muy pronto te llegará de forma natural. ¡Por Excalibur lo juro!

7.7 Paquetes de transmisión—¡Hola, mundo!

Hasta ahora, esta guía ha hablado sobre el envío de datos de un host a otro host. ¡Pero es posible, insisto, que puedas, con la autoridad adecuada, enviar datos a varios hosts *al mismo tiempo*!

Con UDP (solo UDP, no TCP) e IPv4 estándar, esto se hace a través de un mecanismo llamado *radiodifusión*. Con IPv6, la transmisión no es compatible y hay que recurrir a la técnica a menudo superior de la *multidifusión*, de la que, lamentablemente, no hablaré en este momento. Pero basta del futuro estrellado: estamos atrapados en el presente de 32 bits.

¡Pero espera! No puedes simplemente salir corriendo y comenzar a transmitir de cualquier manera; Debe configurar la opción de socket `SO_BROADCAST` antes de poder enviar un paquete de transmisión a la red. ¡Es como una de esas pequeñas cubiertas de plástico que ponen sobre el interruptor de lanzamiento de misiles! ¡Esa es la cantidad de poder que tienes en tus manos!

Pero en serio, sin embargo, existe un peligro en el uso de paquetes de difusión, y es que cada sistema que recibe un paquete de difusión debe deshacer todas las capas de encapsulación de datos hasta que averigüe a qué puerto están destinados los datos. Y luego entrega los datos o los descarta. En cualquier caso, es mucho trabajo para cada máquina que recibe el paquete de difusión, y dado que están todas en la red local, podría ser una gran cantidad de máquinas haciendo mucho trabajo innecesario. Cuando el juego Doom salió por primera vez, se trataba de una queja sobre su código de red.

Ahora, hay más de una forma de despellejar a un gato... Espera un momento. ¿Realmente hay más de una forma de despellejar a un gato? ¿Qué clase de expresión es esa? Del mismo modo, hay más de una forma de enviar un paquete de transmisión. Entonces, para llegar al meollo de la cuestión: ¿cómo se especifica la dirección de destino para un mensaje de transmisión? Hay dos formas comunes:

1. Envíe los datos a la dirección de difusión de una subred específica. Este es el número de red de la subred con todos los bits de un bit establecidos para la parte del host de la dirección. Por ejemplo, en casa mi red es `192.168.1.0`, mi máscara de red es `255.255.255.0`, por lo que el último byte de la dirección es mi número de host (porque los primeros tres bytes, según la máscara de red, son el número de red). Así que mi dirección de transmisión es `192.168.1.255`. En Unix, el comando `ifconfig` te dará todos estos datos. (Si tienes curiosidad, la lógica bit a bit para obtener tu dirección de transmisión es `network_number` O (NO `máscara de red`).) Puede enviar este tipo de paquete de difusión a redes remotas, así como a su red local, pero corre el riesgo de que el paquete sea descartado por el enrutador de destino. (Si no lo soltan, entonces algún pitufo al azar podría comenzar a inundar su LAN con tráfico de transmisión).
2. Envíe los datos a la dirección de transmisión "global". Esto es `255.255.255.255,` también conocido como `INADDR_BROADCAST`. Muchas máquinas automáticamente bit a bit AND this con su número de red para convertirlo en una dirección de transmisión de red, pero algunas no lo harán. Varía. Irónicamente, los routers no reenvían este tipo de paquete de difusión fuera de la red local.

Entonces, ¿qué sucede si intenta enviar datos en la dirección de transmisión sin configurar primero el `SO_BROADCAST`
¿Opción de enchufe? Bueno, vamos a encender al buen `hablador` y `oyente` y ver qué pasa.

```
$ talker 192.168.1.2 foo
envió 3 bytes a 192.168.1.2
$ talker 192.168.1.255 foo
sendto: Permiso denegado
$ talker 255.255.255.255 foo
sendto: Permiso denegado
```

Sí, no es nada feliz... porque no configuramos la opción de `SO_BROADCAST` socket. ¡Haz eso, y ahora puedes `enviar a()` a cualquier lugar que quieras!

De hecho, esa es la *única diferencia* entre una aplicación UDP que puede transmitir y una que no. Así que tomemos la antigua aplicación `talker` y agreguemos una sección que establezca la opción de `socket SO_BROADCAST`. Llamaremos a este programa `broadcaster.c`[14]:

```
/*
** broadcaster.c -- un "cliente" de datagramas como talker.c, excepto
**                  este puede transmitir
*/

#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 PUERTO DE SERVIDOR 4950   el puerto al que se conectarán los usuarios

int main(int argc, char *argv[])
{
    int calcetín;
    estructura sockaddr_in their_addr;  Información de la dirección del conector
    struct hostent *he;
    int numbytes;
    int difusión = 1;
    char broadcast = '1'; Si eso no funciona, pruebe esto

    if (argc != 3) {
        fprintf(stderr,"uso: mensaje de nombre de host de la
        emisora\n"); salida(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) {  obtener la información del host
        perror("gethostbyname");
        salida(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) ==-1) {
        perror("socket");
```

```
        salida(1);
    }

    Esta llamada es lo que permite que se envíen paquetes de difusión:
    si (setsockopt(calcetín, SOL_SOCKET, SO_BROADCAST, &emisión,
        tamañode emisión) == -1) {
        perror("Setsockopt (SO_BROADCAST)");
        salida(1);
    }

    their_addr.sin_family = AF_INET;      Orden de bytes de host
    their_addr.sin_port = htons(PUERTO DEL SERVIDOR); Orden de bytes de red
    their_addr.sin_addr = *((Estructura in_addr *)él->h_addr);
    memset(their_addr.sin_zero, '\0', tamañode their_addr.sin_zero);

    numbytes = Enviar a(calcetín, argv[2], strlen(argv[2]), 0,
              (Estructura calcetín *)&their_addr, tamañode their_addr);

    si (numbytes == -1) {
        perror("Enviar");
        salida(1);
    }

    printf("Enviado %d bytes a %s\n", numbytes,
        inet_ntoa(their_addr.sin_addr));

    cerrar(calcetín);

    devolución 0;
}
```

¿Cuál es la diferencia entre esto y una situación cliente/servidor UDP "normal"? ¡Nada! (Con la excepción de que el cliente puede enviar paquetes de difusión en este caso). Como tal, continúe y ejecute el antiguo programa de oyente UDP en una ventana y el `emisor` en otra. Ahora debería poder realizar todos esos envíos que fallaron, arriba.

```
$ broadcaster 192.168.1.2 foo
envió 3 bytes a 192.168.1.2
$ emisora 192.168.1.255 foo
envió 3 bytes a 192.168.1.255
$ emisora 255.255.255.255 foo envió
3 bytes a 255.255.255.255
```

Y debería ver que `el oyente` responde que obtuvo los paquetes. (Si El `oyente` no responde, podría deberse a que está vinculado a una dirección IPv6. Intente cambiar el `AF_INET6` en `listener.c` para `AF_INET` para forzar IPv4).

Bueno, eso es algo emocionante. Pero ahora encienda el `oyente` en otra máquina a su lado en la misma red para que tenga dos copias en marcha, una en cada máquina, y vuelva a ejecutar `broadcaster` con su dirección de transmisión ... ¡Eh! ¡Ambos `oyentes` reciben el paquete a pesar de que solo llamaste a `sendto()` una vez! ¡Fresco!

Si el `oyente` obtiene datos que le envías directamente, pero no datos de la dirección de difusión, es posible que tengas un firewall en tu máquina local que esté bloqueando los paquetes. (Sí, Pat y Bapper, gracias por darse cuenta antes de que yo lo hiciera de que esta es la razón por la que mi código de muestra no funcionaba. Te dije que te mencionaría en la guía, y aquí estás. Así que *nyah*.)

De nuevo, tenga cuidado con los paquetes de difusión. Dado que cada máquina en la LAN se verá obligada a lidiar con el paquete, ya sea que `lo reciba o` no, puede presentar una gran carga para toda la red informática.

Definitivamente, deben usarse con moderación y de manera adecuada.

Capítulo 8

Preguntas comunes

¿Dónde puedo obtener esos archivos de encabezado?

Si aún no los tiene en su sistema, probablemente no los necesite. Consulte el manual de su plataforma en particular. Si está compilando para Windows, solo necesita `#include <winsock.h>`.

¿Qué hago cuando `bind()` informa "Dirección ya en uso"?

Tienes que usar `setsockopt()` con la opción `SO_REUSEADDR` en el socket de escucha. Echa un vistazo a la sección sobre `bind()` y la sección sobre `select()` para ver un ejemplo.

¿Cómo obtengo una lista de sockets abiertos en el sistema?

Utilice el `netstat`. Consulte la página del `manual` para obtener todos los detalles, pero debería obtener una buena salida simplemente escribiendo:

```
$ netstat
```

El único truco es determinar qué socket está asociado con qué programa. `:-)`

¿Cómo puedo ver la tabla de enrutamiento?

Ejecute el `comando route` (en `/sbin` en la mayoría de los Linuxes) o el comando `netstat -r`. O el comando
`Ruta IP`.

¿Cómo puedo ejecutar los programas cliente y servidor si solo tengo una computadora? ¿No necesito una red para escribir programas de red?

Afortunadamente para usted, prácticamente todas las máquinas implementan un "dispositivo" de red de bucle invertido que se encuentra en el kernel y pretende ser una tarjeta de red. (Esta es la interfaz listada como "`lo`" en la tabla de enrutamiento).

Imagina que has iniciado sesión en una máquina llamada "`goat`". Ejecute el cliente en una ventana y el servidor en otra. O inicie el servidor en segundo plano ("`servidor &`") y ejecute el cliente en la misma ventana. El resultado del dispositivo de bucle invertido es que puede usar el `cliente goat` o el `cliente localhost` (ya que "`localhost`" probablemente esté definido en su `archivo /etc/`hosts) y tendrá al cliente hablando con el servidor sin una red.

En resumen, no es necesario realizar cambios en el código para que se ejecute en una sola máquina no conectada a la red. ¡Vaya!

¿Cómo puedo saber si el lado remoto tiene una conexión cerrada?

Puedes saberlo porque `recv()` devolverá `0`.

¿Cómo implemento una utilidad "ping"? ¿Qué es ICMP? ¿Dónde puedo encontrar más información sobre los sockets sin procesar y `SOCK_RAW`?

Todas sus preguntas sobre sockets sin procesar serán respondidas en los libros de programación de redes UNIX de W. Richard Stevens . Además, busque en el `subdirectorio ping/` en el código fuente de programación de redes UNIX de Stevens, disponible en línea[1].

[1] http://www.unpbook.com/src.html

¿Cómo puedo cambiar o acortar el tiempo de espera de una llamada a `connect()`?

En lugar de darle exactamente la misma respuesta que W. Richard Stevens le daría, simplemente le remitiré a `lib/connect_nonb.c` en el código fuente 2 de UNIX Network Programming.

Lo esencial de esto es que creas un descriptor de socket con `socket()`, lo configuras como non-blocking, llamas a `connect()`, y si todo va bien, `connect()` devolverá `-1` inmediatamente y `errno` se establecerá en `EINPROGRESS`. A continuación, se llama a `select()` con el tiempo de espera que se desee, pasando el descriptor de socket en los conjuntos de lectura y escritura. Si no se agota el tiempo de espera, significa que la `llamada connect()` se ha completado. En este punto, tendrás que usar `getsockopt()` con la opción `SO_ERROR` para obtener el valor devuelto de la llamada `connect()`, que debería ser cero si no hubo ningún error.

Finalmente, es probable que desee volver a configurar el socket para que se bloquee nuevamente antes de comenzar a transferir datos a través de él.

Tenga en cuenta que esto tiene el beneficio adicional de permitir que su programa haga otra cosa mientras se conecta, también. Podrías, por ejemplo, establecer el tiempo de espera en algo bajo, como 500 ms, y actualizar un indicador en pantalla cada tiempo de espera, luego llamar a `select()` nuevamente. Cuando hayas llamado a `select()` y se haya agotado el tiempo de espera, digamos, 20 veces, sabrás que es hora de renunciar a la conexión.

Como dije, echa un vistazo a la fuente de Stevens para ver un ejemplo perfectamente excelente.

¿Cómo se compila para Windows?

Primero, elimine Windows e instale Linux o BSD. `};-)`. No, en realidad, solo vea la sección sobre la creación para Windows en la introducción.

¿Cómo se compila para Solaris/SunOS? ¡Sigo recibiendo errores de enlace cuando intento compilar!

Los errores del enlazador se producen porque las cajas Sun no se compilan automáticamente en las bibliotecas de sockets. Consulte la sección sobre la creación para Solaris/SunOS en la introducción para ver un ejemplo de cómo hacerlo.

¿Por qué `select()` sigue cayendo en una señal?

Las señales tienden a hacer que las llamadas al sistema bloqueadas devuelvan `-1` con `errno` establecido en `EINTR`. Cuando configuras un manejador de señales con `sigaction()`, puedes establecer el indicador `SA_RESTART`, que se supone que reinicia la llamada al sistema después de que se interrumpió.

Naturalmente, esto no siempre funciona.

Mi solución favorita a esto implica una declaración goto. Sabes que esto irrita a tus profesores sin fin, ¡así que adelante!

```
select_restart:
if ((err = select(fdmax+1, &readfds, NULL, NULL, NULL)) == -1) {
    if (errno == EINTR) {
        Alguna señal nos interrumpió, así que reinicie
        ir a select_restart;
    }
    Maneja el error real aquí:
    perror("seleccionar");
}
```

Claro, no necesitas usar `goto` en este caso; puedes usar otras estructuras para controlarlo. Pero creo que el `goto` es en realidad más limpia.

¿Cómo puedo implementar un tiempo de espera en una llamada a `recv()`?

¡Utilice `select()`! Le permite especificar un parámetro de tiempo de espera para los descriptores de socket desde los que desea leer. O bien, puede envolver toda la funcionalidad en una sola función, como esta:

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

[2] http://www.unpbook.com/src.html

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

int recvtimeout(int s, char *buf, int len, int timeout)
{
    fd_set FDS;
    int n;
    Estructura TimeVal TV;

    Configurar el conjunto de descriptores de archivo
    FD_ZERO(&fds);
    FD_SET(s, y fds);

    Configurar la estructura TimeVal para el tiempo de espera
    televisión.tv_sec = tiempo
    de espera;
    televisión.tv_usec = 0;

     esperar hasta que se agote el tiempo
    de espera o se reciban los datos n =
    select(s+1, &fds, NULL, NULL, &tv); si
    (n == 0) devuelve -2;  ¡interrupción!
    si (n == -1) devuelve -1;  error

    Los datos deben estar aquí, así que haz un recv() normal
    return recv(s, buf, len, 0);
}
.
.
.
Ejemplo de llamada a recvtimeout():
 n = recvtimeout(s, buf, sizeof buf, 10);  Tiempo de espera de 10 segundos

if (n == -1) {
    Se ha producido un error
    perror("recvtimeout");
}
else if (n == -2) {
    Se ha agotado el tiempo de espera
} else {
    Tengo algunos datos en buf
}
.
```

devolver `0`? Bueno, si recuerdas, un valor devuelto de `0` en una llamada a `recv()` significa que el lado remoto cerró la conexión. Por lo tanto, ese valor de retorno ya está mencionado, y `-1` significa "error", por lo que elegí `-2` como mi indicador de tiempo de espera.

¿Cómo puedo cifrar o comprimir los datos antes de enviarlos a través del socket?

Una forma fácil de hacer el cifrado es usar SSL (capa de sockets seguros), pero eso está más allá del alcance de esta guía. (Echa un vistazo a la OpenSSL project[3] para más información.)

Pero suponiendo que desee conectar o implementar su propio compresor o sistema de cifrado, es solo cuestión de pensar en sus datos como si se ejecutaran a través de una secuencia de pasos entre ambos extremos. Cada paso cambia los datos de alguna manera.

1. El servidor lee los datos del archivo (o donde sea)

[3] https://www.openssl.org/

2. El servidor encripta/comprime los datos (se añade esta parte)
3. datos cifrados del servidor

`send()` Ahora al revés:

1. Datos cifrados del `cliente` recv()
2. El cliente descifra/descomprime los datos (se agrega esta parte)
3. El cliente escribe datos en el archivo (o donde sea)

Si vas a comprimir y cifrar, recuerda comprimir primero. `:-)`

Siempre y cuando el cliente deshaga correctamente lo que hace el servidor, los datos estarán bien al final sin importar cuántos pasos intermedios agregue.

Así que todo lo que necesitas hacer para usar mi código es encontrar el lugar entre donde se leen los datos y los datos se envían (usando `send()`) a través de la red, y pegar algún código allí que haga el cifrado.

¿Qué es este "`PF_INET`" que sigo viendo? ¿Está relacionado con `AF_INET`?

Sí, sí lo es. Consulte la sección sobre `socket()` para obtener más detalles.

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

Para simplificar, digamos que el cliente `connect()`s, `send()`s y `close()`s la conexión (es decir, no hay llamadas posteriores al sistema sin que el cliente se conecte de nuevo).

El proceso que sigue el cliente es el siguiente:

1. `connect()` al servidor
2. `send("/sbin/ls > /tmp/client.out")`
3. `close()` la conexión

Mientras tanto, el servidor maneja los datos y los ejecuta:

1. `accept()` la conexión desde el cliente
2. `recv(str)` la cadena de comandos
3. `close()` la conexión
4. `system(str)` para ejecutar el comando

¡Cuidado! Hacer que el servidor ejecute lo que dice el cliente es como dar acceso remoto al shell y las personas pueden hacer cosas en su cuenta cuando se conectan al servidor. Por ejemplo, en el ejemplo anterior, ¿qué pasa si el cliente envía "`rm -rf ~`"? Elimina todo en tu cuenta, ¡eso es lo que hay!

Así que te vuelves sabio y evitas que el cliente use cualquiera excepto un par de utilidades que sabes que son seguras, como la `utilidad foobar`:

```
Si (!strncmp(str, "foobar", 6)) { sprintf(sysstr,
    "%s > /tmp/server.out", str); sistema
    (sysstr);
}
```

Pero sigue siendo inseguro, por desgracia: ¿qué pasa si el cliente entra en "`foobar; rm -rf ~`"? Lo más seguro es escribir una pequeña rutina que ponga un carácter de escape ("`\`") delante de todos los caracteres no alfanuméricos (incluidos los espacios, si corresponde) en los argumentos del comando.

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

Estoy enviando una gran cantidad de datos, pero cuando `recv()`, solo recibe 536 bytes o 1460 bytes a la vez. Pero si lo ejecuto en mi máquina local, recibe todos los datos al mismo tiempo. ¿Qué pasa?

Estás alcanzando la MTU, el tamaño máximo que el medio físico puede manejar. En la máquina local, está utilizando el dispositivo de bucle invertido que puede manejar 8K o más sin problemas. Pero en Ethernet, que solo puede manejar 1500 bytes con un encabezado, se alcanza ese límite. A través de un módem, con 576 MTU (de nuevo, con encabezado), se alcanza el límite aún más bajo.

En primer lugar, debe asegurarse de que se envíen todos los datos. (Véase la `sendall()` para obtener más detalles). Una vez que estés seguro de eso, entonces necesitas llamar a `recv()` en un bucle hasta que todos tus datos sean leídos.

Lea la sección Hijo de la encapsulación de datos para obtener detalles sobre cómo recibir paquetes completos de datos utilizando múltiples llamadas a `recv()`.

Estoy en una caja de Windows y no tengo la llamada al `sistema fork()` ni ningún tipo de `sigacción de estructura`. ¿Qué hacer?

Si están en algún lugar, estarán en las bibliotecas POSIX que pueden haberse enviado con el compilador. Como no tengo una caja de Windows, realmente no puedo decirte la respuesta, pero creo recordar que Microsoft tiene una capa de compatibilidad POSIX y ahí es donde `estaría fork()`. (Y tal vez incluso `Sigacción`.)

Busque en la ayuda que viene con VC++ "bifurcación" o "POSIX" y vea si le da alguna pista.

Si eso no funciona en absoluto, deshazte de la `bifurcación()`/`sigaction` y reemplázala con el equivalente de Win32: `CreateProcess()`. No sé cómo usar `CreateProcess()`, toma un montón de argumentos, pero debería cubrirse en los documentos que vienen con VC++.

Estoy detrás de un cortafuegos, ¿cómo puedo informar a las personas que están fuera del cortafuegos de mi dirección IP para que puedan conectarse a mi equipo?

Desafortunadamente, el propósito de un firewall es evitar que las personas fuera del firewall se conecten a máquinas dentro del firewall, por lo que permitirles hacerlo se considera básicamente una violación de la seguridad.

Esto no quiere decir que todo esté perdido. Por un lado, a menudo todavía puedes `conectarte()` a través del cortafuegos si está haciendo algún tipo de enmascaramiento o NAT o algo así. Sólo tienes que diseñar tus programas para que siempre seas tú quien inicie la conexión, y estarás bien.

Si eso no es satisfactorio, puede pedirle a sus administradores de sistemas que hagan un agujero en el firewall para que las personas puedan conectarse con usted. El firewall puede reenviarlo a través de su software NAT, o a través de un proxy o algo así.

Tenga en cuenta que un agujero en el cortafuegos no es nada que deba tomarse a la ligera. Tienes que asegurarte de no dar acceso a las malas personas a la red interna; Si eres un principiante, es mucho más difícil hacer que el software sea seguro de lo que imaginas.

No hagas que tu administrador de sistemas se enfade conmigo. `;-)`

¿Cómo escribo un rastreador de paquetes? ¿Cómo pongo mi interfaz Ethernet en modo promiscuo?

Para aquellos que no lo saben, cuando una tarjeta de red está en "modo promiscuo", reenviará TODOS los paquetes al sistema operativo, no solo los que se dirigieron a esta máquina en particular. (Aquí estamos hablando de direcciones de capa Ethernet, no de direcciones IP, pero dado que Ethernet es de capa inferior a IP, todas las direcciones IP también se reenvían de manera efectiva. Consulte la sección Tonterías de bajo nivel y teoría de redes para obtener más información).

Esta es la base de cómo funciona un rastreador de paquetes. Pone la interfaz en modo promiscuo, luego el sistema operativo obtiene cada paquete que pasa por el cable. Tendrás un socket de algún tipo desde el que puedes leer estos datos.

Desafortunadamente, la respuesta a la pregunta varía según la plataforma, pero si buscas en Google, por ejemplo, "windows promiscuous ioctl", probablemente llegarás a alguna parte. Para Linux, también hay lo que parece ser un útil subproceso 4 de Stack Overflow.

¿Cómo puedo establecer un valor de tiempo de espera personalizado para un socket TCP o UDP?

Depende de su sistema. Puede buscar en la red SO_RCVTIMEO y `SO_SNDTIMEO` (para usar con `setsockopt()`) para ver si su sistema soporta dicha funcionalidad.

La página del manual de Linux sugiere usar `alarm()` o `setitimer()` como sustituto.

¿Cómo puedo saber qué puertos están disponibles para usar? ¿Existe una lista de números de puerto "oficiales"?

Por lo general, esto no es un problema. Si estás escribiendo, por ejemplo, un servidor web, entonces es una buena idea usar el conocido puerto 80 para tu software. Si está escribiendo solo su propio servidor especializado, elija un puerto al azar (pero mayor que 1023) y pruébelo.

[4]https://stackoverflow.com/questions/21323023/

Si el puerto ya está en uso, obtendrás un error de "Dirección ya en uso" cuando intentes vincular `()`. Elija otro puerto. (Es una buena idea permitir que el usuario del software especifique un puerto alternativo con un archivo de configuración o un modificador de línea de comandos).

Hay una lista de números de puerto oficiales[5] mantenida por la Autoridad de Números Asignados de Internet (IANA). El hecho de que algo (más de 1023) esté en esa lista no significa que no pueda usar el puerto. Por ejemplo, DOOM de Id Software usa el mismo puerto que "mdqs", sea lo que sea. Todo lo que importa es que nadie más *en la misma máquina* esté usando ese puerto cuando usted desea usarlo.

[5]https://www.iana.org/assignments/port-numbers

Capítulo 9

Páginas de manual

En el mundo Unix, hay una gran cantidad de manuales. Tienen pequeñas secciones que describen las funciones individuales que tiene a su disposición.

Por supuesto, `manual` sería demasiado para escribir. Quiero decir, a nadie en el mundo Unix, incluyéndome a mí, le gusta escribir tanto. De hecho, podría seguir y seguir hablando largo y tendido sobre lo mucho que prefiero ser conciso, pero en lugar de eso, seré breve y no los aburriré con largas diatribas sobre lo asombrosamente breve que prefiero ser en prácticamente todas las circunstancias en su totalidad.

[Aplausos]

Gracias. Lo que quiero decir es que estas páginas se llaman "páginas de manual" en el mundo Unix, y he incluido aquí mi propia variante truncada personal para su disfrute de la lectura. La cuestión es que muchas de estas funciones son mucho más generales de lo que estoy dejando, pero solo voy a presentar las partes que son relevantes para la programación de sockets de Internet.

¡Pero espera! Eso no es todo lo que está mal con mis páginas de manual:

- Están incompletos y solo muestran lo básico de la guía.
- Hay muchas más páginas de manual que esta en el mundo real.
- Son diferentes a los de su sistema.
- Los archivos de encabezado pueden ser diferentes para ciertas funciones del sistema.
- Los parámetros de la función pueden ser diferentes para ciertas funciones del sistema.

Si quieres la información real, revisa tus páginas de manual locales de Unix escribiendo `man lo que sea`, donde "lo que sea" es algo que te interesa increíblemente, como "`aceptar`". (Estoy seguro de que Microsoft Visual Studio tiene algo similar en su sección de ayuda. Pero "hombre" es mejor porque es un byte más conciso que "ayuda". ¡Unix gana de nuevo!)

Entonces, si estos son tan defectuosos, ¿por qué incluirlos en la Guía? Bueno, hay algunas razones, pero las mejores son que (a) estas versiones están orientadas específicamente a la programación en red y son más fáciles de digerir que las reales, y (b) ¡estas versiones contienen ejemplos!

¡Oh! Y hablando de los ejemplos, no tiendo a poner toda la verificación de errores porque realmente aumenta la longitud del código. Pero debes hacer una verificación de errores prácticamente cada vez que hagas cualquiera de las llamadas al sistema, a menos que estés totalmente 100% seguro de que no va a fallar, ¡y probablemente deberías hacerlo incluso entonces!

9.1 `aceptar()`

Aceptar una conexión entrante en un socket de escucha

Sinopsis

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

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

Descripción

Una vez que te hayas tomado la molestia de obtener un `SOCK_STREAM` socket y configurarlo para las conexiones entrantes con `listen()`, entonces llamas `a accept()` para obtener un nuevo descriptor de socket para usar en la comunicación posterior con el cliente recién conectado.

El antiguo socket que está usando para escuchar todavía está allí, y se usará para más llamadas `accept()` a medida que ingresen.

Parámetro	Descripción
`s`	El `escuchar()`ing socket descriptor.
`Addr`	Esto se completa con la dirección del sitio que se está conectando con usted.
`addrlen`	Esto se rellena con el `tamañode()` La estructura devuelta en el archivo `Addr` parámetro. Puede ignorarlo con seguridad si asume que está recibiendo un `Estructura sockaddr_in` atrás, que sabes que eres, porque ese es el tipo por el que pasaste `Addr`.

`accept()` normalmente se bloqueará, y puedes usar `select()` para echar un vistazo al descriptor del socket de escucha antes de tiempo para ver si está "listo para leer". Si es así, ¡entonces hay una nueva conexión esperando a ser `aceptada`! ¡Yay! Alternativamente, puede establecer el `indicador O_NONBLOCK` en el socket de escucha usando `fcntl()`, y luego nunca se bloqueará, eligiendo en su lugar devolver `-1` con `errno` establecido en `EWOULDBLOCK`.

El descriptor de socket devuelto por `accept()` es un descriptor de socket de buena fe, abierto y conectado al host remoto. Tienes que `cerrarlo cuando` hayas terminado con él.

Valor devuelto

`accept()` devuelve el descriptor de socket recién conectado, o `-1` en caso de error, con `errno` establecido apropiadamente.

Ejemplo

```
estructura sockaddr_storage
their_addr; socklen_t addr_size;
struct addrinfo sugerencias, *res;
int sockfd, new_fd;

Primero, cargue las estructuras de direcciones con getaddrinfo():

memset(&hints, 0, sizeof hints);
Pistas.ai_family = AF_UNSPEC;  usar IPv4 o IPv6, lo que ocurra
Pistas.ai_socktype = SOCK_STREAM;
Consejos.ai_flags = AI_PASSIVE;   rellene mi IP por mí

getaddrinfo(NULL, MYPORT, &hints, &res);

Haz un socket, átalo y escucha en él:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);
escuchar (sockfd, BACKLOG);
```

```
20
21  Ahora acepte una conexión entrante:
22
23  addr_size = tamaño de their_addr;
24  new_fd = aceptar(calcetín, (struct calcetín *)&their_addr, &addr_size);
25
26  ¡Listo para comunicarse en el descriptor de socket new_fd!
```

Consulte también

`socket()`, `getaddrinfo()`, `listen()`, `struct sockaddr_in`

9.2 `bind()`

Asociar un socket con una dirección IP y un número de puerto

Sinopsis

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

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

Descripción

Cuando una máquina remota quiere conectarse a su programa de servidor, necesita dos datos: la dirección IP y el número de puerto. La llamada `bind()` te permite hacer precisamente eso.

Primero, llama a `getaddrinfo()` para cargar un `struct sockaddr` con la dirección de destino y la información del puerto. Luego llamas `a socket()` para obtener un descriptor de socket, y luego pasas el socket y la dirección a `bind()`, ¡y la dirección IP y el puerto se vinculan mágicamente (usando magia real) al socket!

Si no conoces tu dirección IP, o sabes que solo tienes una dirección IP en la máquina, o no te importa cuál de las direcciones IP de la máquina se utiliza, simplemente puedes pasar el `indicador AI_PASSIVE` en el parámetro `hints` a `getaddrinfo()`. Lo que esto hace es rellenar la parte de la dirección IP del `struct sockaddr` con un valor especial que le dice `a bind()` que debe rellenar automáticamente la dirección IP de este host.

¿Qué qué? ¿Qué valor especial se carga en la `dirección IP de struct sockaddr` para que rellene automáticamente la dirección con el host actual? Te lo diré, pero ten en cuenta que esto es solo si estás completando el `struct sockaddr` a mano; si no, usa los resultados de `getaddrinfo()`, como se indica arriba. En IPv4, el `campo sin_addr.s_addr` de la `estructura sockaddr_in` estructura se establece en `INADDR_ANY`. En IPv6, el campo `sin6_addr` de la `estructura sockaddr_in6` estructura se asigna a partir de la variable global in6addr_any. O bien, si va a declarar un nuevo `in6_addr struct`, puede inicializarlo en `IN6ADDR_ANY_INIT`.

Por último, el `parámetro addrlen` debe establecerse en `sizeof my_addr`.

Valor devuelto

Devuelve cero en caso de éxito, o `-1` en caso de error (y `errno` se establecerá en consecuencia).

Ejemplo

```
Forma moderna de hacer las cosas con getaddrinfo()

struct addrinfo sugerencias, *res;
int calcetín;

Primero, cargue las estructuras de direcciones con getaddrinfo():

memset(&hints, 0, sizeof hints);
Pistas.ai_family = AF_UNSPEC;  usar IPv4 o IPv6, lo que ocurra
Pistas.ai_socktype = SOCK_STREAM;
Consejos.ai_flags = AI_PASSIVE;   rellene mi IP por mí

getaddrinfo(NULL, "3490", &hints, &res);

Hacer un enchufe:
(¡De hecho, debería recorrer la lista de enlaces "res" y verificar errores!)

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

Vincúlalo al puerto que pasamos a getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);
```

```
ejemplo de empaquetado de una estructura a mano, IPv4

struct sockaddr_in myaddr;
int s;

myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(3490);

puede especificar una dirección IP:
inet_pton(AF_INET, "63.161.169.137", &(myaddr.sin_addr));

O puede dejar que seleccione automáticamente uno:
myaddr.sin_addr.s_addr = INADDR_ANY;

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&myaddr, sizeof myaddr);
```

Consulte también

`getaddrinfo()`, `socket()`, `struct sockaddr_in`, `struct in_addr`

9.3 `conectar()`

Conectar un socket a un servidor

Sinopsis

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

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

Descripción

Una vez que hayas creado un descriptor de socket con la llamada socket(), puedes conectar() ese socket a un servidor remoto usando la llamada al sistema connect() bien llamada. Todo lo que necesita hacer es pasarle el descriptor de socket y la dirección del servidor que le interesa conocer mejor. (Ah, y la longitud de la dirección, que comúnmente se pasa a funciones como esta).

Por lo general, esta información llega como resultado de una llamada a getaddrinfo(), pero puede completar su propio struct sockaddr si lo desea.

Si aún no has llamado a bind() en el descriptor de socket, se vincula automáticamente a tu dirección IP y a un puerto local aleatorio. Por lo general, esto está bien para ti si no eres un servidor, ya que realmente no te importa cuál sea tu puerto local; Solo le importa cuál es el puerto remoto para poder ponerlo en el parámetro serv_addr. Puedes llamar a bind() si realmente quieres que tu socket de cliente esté en una dirección IP y un puerto específicos, pero esto es bastante raro.

Una vez que el socket está conectado(), eres libre de enviar datos() y recv() a tu antojo. Nota especial: si conectas() un socket UDP SOCK_DGRAM a un host remoto, puedes usar send() y recv(), así como sendto() y recvfrom(). Si quieres.

Valor devuelto

Devuelve cero en caso de éxito, o -1 en caso de error (y errno se establecerá en consecuencia).

Ejemplo

```
 Conéctese al  puerto www.example.com 80 (http)

struct addrinfo sugerencias, *res;
int calcetín;

Primero, cargue las estructuras de direcciones con getaddrinfo():

memset(&hints, 0, sizeof hints);
Pistas.ai_family = AF_UNSPEC;  usar IPv4 o IPv6, lo que ocurra
Pistas.ai_socktype = SOCK_STREAM;

Podríamos poner "80" en lugar de "http" en la siguiente línea:
getaddrinfo("www.example.com", "http", &hints, &res);

Hacer un enchufe:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

Conéctalo a la dirección y al puerto que pasamos a getaddrinfo():

conectar (sockfd, res->ai_addr, res->ai_addrlen);
```

Consulte también

`socket()`, `bind()`

9.4 `cerrar()`

Cierre un descriptor de socket

Sinopsis

```
#include <unistd.h>

int close(int s);
```

Descripción

Una vez que hayas terminado de usar el socket para cualquier plan demente que hayas inventado y no quieras `enviar ()` o `recv()` o, de hecho, hacer *cualquier otra cosa* con el socket, puedes `cerrarlo()`, y se liberará, para no volver a usarlo nunca más.

El lado remoto puede decir si esto sucede de una de dos maneras. Uno: si el lado remoto llama a `recv()`, devolverá `0`. Dos: si el lado remoto llama a `send()`, recibirá una señal `SIGPIPE` y send() devolverá `-1` y `errno` se establecerá en `EPIPE`.

Usuarios de Windows: la función que debe usar se llama `closesocket()`, no `close()`. Si intentas usar `close()` en un descriptor de socket, es posible que Windows se enfade... Y no te gustaría que estuviera enfadado.

Valor devuelto

Devuelve cero en caso de éxito, o `-1` en caso de error (y `errno` se establecerá en consecuencia).

Ejemplo

```
s = socket(PF_INET, SOCK_DGRAM, 0);
.
.
.
un montón de cosas...*BRRRONNNN!*
.
.
.
cierre(s);  No hay mucho que hacer, en realidad.
```

Consulte también

`socket()`, `shutdown()`

9.5 `getaddrinfo()`, `freeaddrinfo()`, `gai_strerror()`

Obtenga información sobre un nombre de host y/o servicio y cargue un `struct sockaddr` con el resultado.

Sinopsis

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

Int Zetaddrinfo (Const four *Nodenname, Const four *Servname,
                 const struct addrinfo *hints,
                 struct addrinfo **res);

void freeaddrinfo(struct addrinfo *ai);

const char *gai_strerror(int ecode);

struct addrinfo {
  Int      ai_flags;           AI_PASSIVE, AI_CANONNAME, ...
  Int      ai_family;          AF_xxx
  Int      ai_socktype;        SOCK_xxx
  Int      ai_protocol;        0 (automático) o IPPROTO_TCP,
  IPPROTO_UDP
                               Duración de ai_addr
  carbonizar                   Nombre canónico para NodeName
  struct sockaddr *ai_addr;  Dirección binaria
  struct addrinfo *ai_next;  Siguiente estructura en la lista
};
```

Descripción

`getaddrinfo()` es una excelente función que devolverá información sobre un nombre de host en particular (como su dirección IP) y cargará un `struct sockaddr` por usted, encargándose de los detalles arenosos (como si es IPv4 o IPv6). Reemplaza las antiguas funciones `gethostbyname()` y `getservbyname()`. La descripción, a continuación, contiene mucha información que puede ser un poco desalentadora, pero el uso real es bastante simple. Podría valer la pena revisar primero los ejemplos.

El nombre de host que le interesa va en el parámetro `nodename`. La dirección puede ser un nombre de host, como "www.example.com", o una dirección IPv4 o IPv6 (pasada como una cadena). Este parámetro también puede ser `NULL` si está utilizando la `marca AI_PASSIVE` (consulte a continuación).

El parámetro `servname` es básicamente el número de puerto. Puede ser un número de puerto (pasado como una cadena, como "80"), o puede ser un nombre de servicio, como "http" o "tftp" o "smtp" o "pop", etc. Los nombres de servicios conocidos se pueden encontrar en la Lista de puertos 1 de la IANA o en el archivo `/etc/services`.

Por último, para los parámetros de entrada, tenemos `sugerencias`. Aquí es donde realmente puedes definir lo que la función `getaddrinfo()` va a hacer. Ponga a cero toda la estructura antes de usarla con `memset()`. Echemos un vistazo a los campos que debe configurar antes de usarlos.

El `ai_flags` se puede configurar para una variedad de cosas, pero aquí hay un par de cosas importantes. (Se pueden especificar varios indicadores mediante la opción OR bit a bit junto con el `|` operador). Revisa tu página de manual para ver la lista completa de banderas.

`AI_CANONNAME` hace que la `ai_canonname` del resultado se rellene con el nombre canónico (real) del host. `AI_PASSIVE` hace que la dirección IP del resultado se rellene con `INADDR_ANY` (IPv4) o `in6addr_any` (IPv6); esto hace que una llamada posterior a `bind()` rellene automáticamente la dirección IP de la `estructura sockaddr` con la dirección del host actual. Eso es excelente para configurar un servidor cuando no desea codificar la dirección.

Si usas el AI_PASSIVE, flag, entonces puedes pasar `NULL` en el `nombre del nodo` (ya que `bind()` lo completará por ti más tarde).

[1]https://www.iana.org/assignments/port-numbers

Continuando con los parámetros de entrada, es probable que desees establecer `ai_family` en `AF_UNSPEC` que le dice a `getaddrinfo()` que busque direcciones IPv4 e IPv6. También puede restringirse a uno u otro con `AF_INET` o `AF_INET6`.

A continuación, el `campo socktype` debe establecerse en `SOCK_STREAM` o `SOCK_DGRAM`, según el tipo de socket que desee.

Por último, solo tienes `que dejar ai_protocol` en `0` para elegir automáticamente el tipo de protocolo.

Ahora, después de que tengas todo eso allí, *¡finalmente* puedes hacer la llamada a `getaddrinfo()`!

Por supuesto, aquí es donde comienza la diversión. La `res` ahora apuntará a una lista enlazada de `struct addrinfos, y puede ir a través de esta lista para obtener todas las direcciones que coincidan con lo que pasó con las sugerencias.`

Ahora, es posible obtener algunas direcciones que no funcionan por una razón u otra, por lo que lo que hace la página de manual de Linux es recorrer la lista haciendo una llamada a `socket()` y `connect()` (o `bind()` si está configurando un servidor con la `bandera AI_PASSIVE`) hasta que tenga éxito.

Finalmente, cuando hayas terminado con la lista enlazada, necesitas llamar a `freeaddrinfo()` para liberar la memoria (o se filtrará, y algunas personas se molestarán).

Valor devuelto

Devuelve cero en caso de éxito o distinto de cero en caso de error. Si devuelve un valor distinto de cero, puede usar la función
`gai_strerror()` para obtener una versión imprimible del código de error en el valor devuelto.

Ejemplo

```
Código para un cliente que se conecta a un servidor
es decir, un socket de flujo para www.example.com En el puerto 80 (http)
IPv4 o IPv6

Int calcetín;
Estructura Sugerencias de addrinfo, *servinfo, *p;
Int Rv;

memset(&Consejos, 0, tamañode Consejos);
Consejos.ai_family = AF_UNSPEC; usar AF_INET6 para forzar IPv6
Consejos.ai_socktype = SOCK_STREAM;

Rv = getaddrinfo("www.example.com", "http", &Consejos, &servinfo);
si (Rv != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(Rv));
    salida(1);
}

recorrer todos los resultados y conectarnos a los primeros que podamos
para(p = servinfo; p != NULO; p = p->ai_next) {
    si ((calcetín = enchufe(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
        perror("zócalo");
        continuar;
    }

    si (conectar(calcetín, p->ai_addr, p->ai_addrlen) == -1) {
        perror("Conectar");
        cerrar(calcetín);
        continuar;
    }

```

```
    descanso;  Si llegamos hasta aquí, debemos habernos conectado con éxito
}

if (p == NULL) {
    bucle al final de la lista sin conexión
    fprintf(stderr, "no se pudo
    conectar\n"); salida(2);
}

freeaddrinfo(servinfo);  todo hecho con esta estructura
```

```
Código para un servidor en espera de conexiones
es decir, un socket de flujo en el puerto 3490, en la IP de este host
ya sea IPv4 o IPv6.

Int calcetín;
Estructura Sugerencias de addrinfo, *servinfo, *p;
Int Rv;

memset(&Consejos, 0, tamañode Consejos);
Consejos.ai_family = AF_UNSPEC; usar AF_INET6 para forzar IPv6
Consejos.ai_socktype = SOCK_STREAM;
Consejos.ai_flags = AI_PASSIVE; usar mi dirección IP

si ((Rv = getaddrinfo(NULO, "3490", &Consejos, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(Rv));
    salida(1);
}

recorrer todos los resultados y enlazar al primero que podamos
para(p = servinfo; p != NULO; p = p->ai_next) {
    si ((calcetín = enchufe(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
        perror("zócalo");
        continuar;
    }

    si (atar(calcetín, p->ai_addr, p->ai_addrlen) == -1) {
        cerrar(calcetín);
        perror("Ligar");
        continuar;
    }

    quebrar; Si llegamos hasta aquí, debemos habernos conectado con éxito
}

si (p == NULO) {
    se reproduce en bucle al final de la lista sin un enlace exitoso
    fprintf(stderr, "No se pudo vincular el socket\n");
    salida(2);
}

freeaddrinfo(servinfo); todo hecho con esta estructura
```

Consulte también

`gethostbyname()`, `getnameinfo()`

9.6 `gethostname()`

Devuelve el nombre del sistema

Sinopsis

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

int gethostname (char *nombre, len size_t);
```

Descripción

Su sistema tiene un nombre. Todos lo hacen. Esta es una cosa un poco más Unixy que el resto de las cosas de red de las que hemos estado hablando, pero todavía tiene sus usos.

Por ejemplo, puede obtener su nombre de host y luego llamar a `gethostbyname()` para averiguar su dirección IP.

El `nombre` del parámetro debe apuntar a un búfer que contendrá el nombre de host y `len` es el tamaño de ese búfer en bytes. `gethostname()` no sobrescribirá el final del búfer (podría devolver un error, o simplemente podría dejar de escribir), y `terminará la cadena NUL si hay espacio para ella en el búfer.`

Valor devuelto

Devuelve cero en caso de éxito, o `-1` en caso de error (y `errno` se establecerá en consecuencia).

Ejemplo

```
1 nombre  de host char[128];
2
3 gethostname (nombre de host, tamaño del
4 nombre de host); printf("Mi nombre de host:
  %s\n",  nombre de host);
```

Consulte también

`gethostbyname()`

9.7 `gethostbyname()`, `gethostbyaddr()`

Obtener una dirección IP para un nombre de host, o viceversa

Sinopsis

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

struct hostent *gethostbyname(const char *nombre);  ¡EN DESUSO!
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

Descripción

TENGA EN CUENTA: ¡estas dos funciones son reemplazadas por `getaddrinfo()` *y* `getnameinfo()`*!* En particular, `gethostbyname()` no funciona bien con IPv6.

Estas funciones se asignan de un lado a otro entre los nombres de host y las direcciones IP. Por ejemplo, si tiene "www.example.com", puede usar `gethostbyname()` para obtener su dirección IP y almacenarla en un `in_addr de estructura`.

Por el contrario, si tiene un `in_addr de estructura` o un `in6_addr de estructura`, puede usar `gethostbyaddr()` para recuperar el nombre de host. `gethostbyaddr()` *es* compatible con IPv6, pero deberías usar el más reciente y brillante `getnameinfo()` en su lugar.

(Si tiene una cadena que contiene una dirección IP en formato de puntos y números de la que desea buscar el nombre de host, sería mejor usar `getaddrinfo()` con la marca `AI_CANONNAME`).

`gethostbyname()` toma una cadena como "www.yahoo.com" y devuelve un `host struct` que contiene toneladas de información, incluida la dirección IP. (Otra información es el nombre de host oficial, una lista de alias, el tipo de dirección, la longitud de las direcciones y la lista de direcciones; es una estructura de propósito general que es bastante fácil de usar para nuestros propósitos específicos una vez que vea cómo).

`gethostbyaddr()` toma un `struct in_addr` o `struct in6_addr` y te muestra un nombre de host correspondiente (si hay uno), por lo que es algo así como lo contrario de `gethostbyname()`. En cuanto a los parámetros, aunque `addr` es un `char*`, en realidad desea pasar un puntero a una `estructura in_addr`. `len` debe ser `sizeof(struct in_addr)` y `type` debe ser `AF_INET`.

Entonces, ¿qué es este `host de estructura` que se devuelve? Tiene una serie de campos que contienen información sobre el host en cuestión.

Campo	Descripción
`char *h_name`	El nombre de host canónico real.
`char **h_aliases`	Una lista de alias a los que se puede acceder con matrices: el último elemento es `NULO`
`int h_addrtype`	El tipo de dirección del resultado, que realmente debería ser `AF_INET` Para nuestro Propósitos.
`Longitud int`	La longitud de las direcciones en bytes, que es 4 para las direcciones IP (versión 4).
`char **h_addr_list`	Una lista de direcciones IP para este host. A pesar de que se trata de un `carbonizar**`, es en realidad, una serie de `in_addr de estructura` disfrazados. El último elemento de la matriz es `NULL`.
`h_addr`	Un alias comúnmente definido para `h_addr_list[0]`. Si solo desea cualquier dirección IP antigua para este host (sí, pueden tener más de una) simplemente use este campo.

Valor devuelto

Devuelve un puntero a un `host struct resultante` en caso de éxito o `NULL` en caso de error.

En lugar del normal `perror()` y todas esas cosas que normalmente usarías para reportar errores, estas funciones tienen resultados paralelos en la variable `h_errno`, que se puede imprimir usando las funciones `herror()` o `hstrerror()`. Funcionan igual que las funciones clásicas `errno`, `perror()` y `strerror()` a las que estás acostumbrado.

Ejemplo

```
1  /* ESTE ES UN MÉTODO OBSOLETO PARA OBTENER NOMBRES DE HOST
2 ¡Usa getaddrinfo() en su lugar! */
3
4 #include <stdio.h>
5 #include <errno.h>
```

```c
#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[])
{
    Int I;
    struct hostent *he;
    struct in_addr **addr_list;

    if (argc != 2) {
        fprintf(stderr,"uso: ghbn nombre de
        host\n"); retorno 1;
    }

    if ((he = gethostbyname(argv[1])) == NULL) {  get host info
        herror("gethostbyname");
        retorno 2;
    }

    Imprimir información sobre este host:
    printf("El nombre oficial es: %s\n", él-
    >h_name); printf("     Direcciones IP: ");
    addr_list = (struct in_addr **)he->h_addr_list;
    for(i = 0; addr_list[i] != NULO; i++) {
        printf("%s ", inet_ntoa(*addr_list[i]));
    }
    printf("\n");

    devuelve 0;
}
```

```c
ESTO HA SIDO REEMPLAZADO
¡Usa getnameinfo() en su lugar!

struct hostent *he;
struct in_addr IPv4addr;
struct in6_addr IPv6addr;

inet_pton(AF_INET, "192.0.2.34", &ipv4addr);
él = gethostbyaddr(&iv4addr, sizeof iv4addr, AF_INET);
printf("Nombre de host: %s\n", he->h_name);

inet_pton(AF_INET6, "2001:db8:63b3:1::beef", & iv6addr);
él = gethostbyaddr(&iv6addr, sizeof iv6addr, AF_INET6);
printf("Nombre de host: %s\n", he->h_name);
```

Consulte también

`getaddrinfo()`, `getnameinfo()`, `gethostname()`, `errno`, `error()`, `strerror()`, `estructura in_addr`

9.8 `getnameinfo()`

Busque el nombre de host y la información del nombre del servicio para un `struct sockaddr determinado`.

Sinopsis

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

int getnameinfo (const struct sockaddr *sa, socklen_t años,
                 char *host, size_t hostlen,
                 char *serv, size_t servlene, int flags);
```

Descripción

Esta función es lo opuesto a `getaddrinfo()`, es decir, esta función toma un `struct sockaddr ya cargado` y realiza una búsqueda de nombre y nombre de servicio en él. Reemplaza las antiguas `funciones` gethostbyaddr() `y` getservbyport().

Tienes que pasar un puntero a un `struct sockaddr` (que en realidad es probablemente un `struct sockaddr_in` o `struct sockaddr_in6` que has lanzado) en el parámetro `sa`, y la longitud de esa `estructura` en el `salen`.

El nombre de host y el nombre del servicio resultantes se escribirán en el área señalada por el `host` y el `servidor`
Parámetros. Por supuesto, debe especificar las longitudes máximas de estos búferes en `hostlen` y `servlen`.

Por último, hay varias banderas que puedes pasar, pero aquí hay un par de buenas. `NI_NOFQDN` hará que el `host` solo contenga el nombre de host, no el nombre de dominio completo. `NI_NAMEREQD` hará que la función falle si el nombre no se puede encontrar con una búsqueda de DNS (si no especifica este indicador y no se puede encontrar el nombre, `getnameinfo()` pondrá una versión de cadena de la dirección IP en el `host` en su lugar).

Como siempre, consulte las páginas de manual locales para obtener la primicia completa.

Valor devuelto

Devuelve cero en caso de éxito o distinto de cero en caso de error. Si el valor devuelto es distinto de cero, se puede pasar a
`gai_strerror()` para obtener una cadena legible por humanos. Consulte `getaddrinfo` para obtener más información.

Ejemplo

```
struct sockaddr_in6 sa;  podría ser IPv4 si lo desea
anfitrión
char[1024];
servicio de
carbonización[20];

Pretend SA está lleno de buena información sobre el host y el puerto...

getnameinfo(&sa, tamañode, host, tamañode host,
            servicio, tamañodelservicio, tamañodel
            servicio, 0);
```

Consulte también

`getaddrinfo()`, `gethostbyaddr()`

9.9 `getpeername()`

Información de la dirección de retorno sobre el lado remoto de la conexión

Sinopsis

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

int getpeername(int s, struct sockaddr *addr, socklen_t *len);
```

Descripción

Una vez que hayas `aceptado una` conexión remota o `te hayas conectado` a un servidor, ahora tienes lo que se conoce como un *par*. Su par es simplemente la computadora a la que está conectado, identificada por una dirección IP y un puerto. Así que...

`getpeername()` simplemente devuelve una `estructura sockaddr_in` llena de información sobre la máquina a la que está conectado.

¿Por qué se le llama "nombre"? Bueno, hay muchos tipos diferentes de sockets, no solo sockets de Internet como los que estamos usando en esta guía, por lo que "nombre" era un buen término genérico que cubría todos los casos. En nuestro caso, sin embargo, el "nombre" del par es su dirección IP y puerto.

Aunque la función devuelve el tamaño de la dirección resultante en `len`, debe precargar `len` con el tamaño de `addr`.

Valor devuelto

Devuelve cero en caso de éxito, o `-1` en caso de error (y `errno` se establecerá en consecuencia).

Ejemplo

```
Supongamos que S es un socket conectado

socklen_t len;
struct sockaddr_storage addr;
char ipstr[INET6_ADDRSTRLEN];
puerto int;

len = tamañode addr;
getpeername(s, (struct sockaddr*)&addr, &len);

tratar tanto con IPv4 como con IPv6:
Si (addr.ss_family == AF_INET) {
    struct sockaddr_in *s = (struct sockaddr_in *)&addr;
    port = ntohs(s->sin_port);
    inet_ntop(AF_INET, & s->sin_addr, ipstr, tamaño de ipstr);
} else {  AF_INET6
    struct sockaddr_in6 *s = (struct sockaddr_in6 *)&addr;
    port = ntohs(s->sin6_port);
    inet_ntop(AF_INET6, & s->sin6_addr, ipstr, tamaño de ipstr);
}

printf("Dirección IP del mismo nivel:
%s\n", ipstr); printf("Puerto del mismo
```

Consulte también

`gethostname()`, `gethostbyname()`, `gethostbyaddr()`

9.10 `errno`

Contiene el código de error de la última llamada al sistema

Sinopsis

```
#include <errno.h>

int errno;
```

Descripción

Esta es la variable que contiene información de error para muchas llamadas al sistema. Si recuerdas, cosas como `socket()` y `listen()` devuelven `-1` en caso de error, y establecen el valor exacto de `errno` para que sepas específicamente qué error ocurrió.

El archivo de cabecera `errno.h` enumera un montón de nombres simbólicos constantes para errores, como `EADDRINUSE`, `EPIPE`, `ECONNREFUSED`, etc. Las páginas del manual local le indicarán qué códigos se pueden devolver como un error, y puede usarlos en tiempo de ejecución para controlar diferentes errores de diferentes maneras.

O, más comúnmente, puede llamar a `perror()` o `strerror()` para obtener una versión legible por humanos del error.

Una cosa a tener en cuenta, para los entusiastas de los subprocesos múltiples, es que en la mayoría de los sistemas, `errno` se define de manera segura para subprocesos. (Es decir, en realidad no es una variable global, pero se comporta como lo haría una variable global en un entorno de un solo subproceso).

Valor devuelto

El valor de la variable es el último error que se ha producido, que podría ser el código de "éxito" si la última acción se realizó correctamente.

Ejemplo

```
s = socket(PF_INET, SOCK_STREAM, 0);
if (s ==-1) {
    perror("socket");  o use strerror()
}

Inténtalo de nuevo:
if (select(n, &readfds, NULL, NULL) == -1) {
    ¡Se ha producido un error!

    Si solo fuimos interrumpidos, simplemente reiniciemos la llamada select():
    if (errno == EINTR) vaya a intentar de nuevo;  ¡AAAA! ¡Vaya!!

    de lo contrario, es un error más grave:
    perror("seleccionar"
    ); salida(1);
}
```

Consulte también

`perror()`, `strerror()`

9.11 `fcntl()`

Descriptores de socket de control

Sinopsis

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

int fcntl(int s, int cmd, arg largo);
```

Descripción

Esta función se usa normalmente para hacer el bloqueo de archivos y otras cosas orientadas a archivos, pero también tiene un par de funciones relacionadas con el socket que puede ver o usar de vez en cuando.

El parámetro `s` es el descriptor de socket en el que desea operar, `cmd` debe establecerse en `F_SETFL` y `arg` puede ser uno de los siguientes comandos. (Como dije, hay más que `fcntl()` de lo que estoy dejando aquí, pero estoy tratando de mantenerme orientado al socket).

`Cmd`	Descripción
`O_NONBLOCK`	Configure el zócalo para que no bloquee. Consulte la sección sobre bloqueo para obtener más detalles.
`O_ASYNC`	Configure el socket para realizar E/S asíncrona. Cuando los datos estén listos para ser `recv()`d en el socket, se generará la señal `SIGIO`. Esto es raro de ver, y está más allá del alcance de la guía. Y creo que solo está disponible en ciertos sistemas.

Valor devuelto

Devuelve cero en caso de éxito, o `-1` en caso de error (y `errno` se establecerá en consecuencia).

Los diferentes usos de la llamada al `sistema fcntl()` en realidad tienen diferentes valores de retorno, pero no los he cubierto aquí porque no están relacionados con el socket. Consulte la página del `comando man fcntl()` local para obtener más información.

Ejemplo

```
1 int s = socket(PF_INET, SOCK_STREAM, 0);
2
3 fcntl(s, F_SETFL, O_NONBLOCK);  Establecido en No bloqueo
4 fcntl(s, F_SETFL, O_ASYNC);      establecido en E/S asincrónica
```

Consulte también

Bloqueo, `send()`

9.12 `htons()`, `htonl()`, `ntohs()`, `ntohl()`

Convierta tipos enteros de varios bytes de orden de bytes de host a orden de bytes de red

Sinopsis

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

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Descripción

Solo para que te sientas realmente descontento, diferentes computadoras usan diferentes ordenaciones de bytes internamente para sus enteros multibyte (es decir, cualquier número entero que sea más grande que un `carácter`). El resultado de esto es que si `envías un` int corto `de dos bytes` desde una caja Intel a una Mac (antes de que también se convirtieran en cajas Intel), lo que una computadora piensa que es el número `1`, la otra pensará que es el número `256`, y viceversa.

La forma de sortear este problema es que todos dejen de lado sus diferencias y estén de acuerdo en que Motorola e IBM tenían razón, e Intel lo hizo de la manera extraña, por lo que todos convertimos nuestros ordenamientos de bytes a "big-endian" antes de enviarlos. Dado que Intel es una máquina "little-endian", es mucho más políticamente correcto llamar a nuestro orden de bytes preferido "Orden de bytes de red". Por lo tanto, estas funciones convierten el orden de bytes nativo al orden de bytes de la red y viceversa.

(Esto significa que en Intel estas funciones intercambian todos los bytes, y en PowerPC no hacen nada porque los bytes ya están en el orden de bytes de la red. Pero siempre debe usarlos en su código de todos modos, ya que alguien podría querer compilarlo en una máquina Intel y aún así hacer que las cosas funcionen correctamente).

Tenga en cuenta que los tipos involucrados son números de 32 bits (4 bytes, probablemente `int`) y de 16 bits (2 bytes, muy probablemente `cortos`). Las máquinas de 64 bits pueden tener un `htonll()` para enteros de 64 `bits, pero no lo he visto.` Solo tendrás que escribir el tuyo propio.

De todos modos, la forma en que funcionan estas funciones es que primero decide si está convirtiendo *desde el* orden de bytes del host (su máquina) o desde el orden de bytes de la red. Si es "host", la primera letra de la función a la que vas a llamar es "h". De lo contrario, es "n" para "red". El centro del nombre de la función siempre es "a" porque está convirtiendo de un "a" otro, y la penúltima letra muestra a qué se está convirtiendo. La última letra es el tamaño de los datos, "s" para abreviar o "l" para largo. Así:

Descripción de la función
`htons()` host `to` network `corto`
`htonl()` host `to` network `long`
`ntohs()` network `to` host `short`
`ntohl()` network `to` host `long`

Valor devuelto

Cada función devuelve el valor convertido.

Ejemplo

```
uint32_t some_long = 10;
uint16_t some_short = 20;

uint32_t network_byte_order;

Convertir y enviar
network_byte_order = htonl(some_long);
enviar(s, &network_byte_order, tamañode(uint32_t), 0);

```

```
some_short == ntohs(htons(some_short));  Esta expresión es cierta
```

9.13 `inet_ntoa()`, `inet_aton()`, `inet_addr`

Convertir direcciones IP de una cadena de puntos y números a una `estructura in_addr` y viceversa

Sinopsis

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

 ¡TODO ESTO ESTÁ EN DESUSO!
¡Usa inet_pton() o inet_ntop() en su lugar!

char *inet_ntoa(estructura in_addr en);
int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
```

Descripción

Estas funciones están en desuso porque no controlan IPv6. Utilice `inet_ntop()` o `inet_pton()` ¡en lugar de! Se incluyen aquí porque todavía se pueden encontrar en la naturaleza.

Todas estas funciones se convierten de un `in_addr de estructura` (probablemente parte de su `sockaddr_in de estructura`) a una cadena en formato de puntos y números (por ejemplo, "192.168.5.10") y viceversa. Si tiene una dirección IP pasada en la línea de comandos o algo así, esta es la forma más fácil de obtener una `estructura in_addr` a `la que conectarse()`, o lo que sea. Si necesita más potencia, pruebe algunas de las funciones de DNS como `gethostbyname()` o intente un *golpe de estado* en su país local.

La función `inet_ntoa()` convierte una dirección de red en un `in_addr de estructura` en una cadena de formato de puntos y números. La "n" en "ntoa" significa red, y la "a" significa ASCII por razones históricas (por lo que es "Network To ASCII"; el sufijo "toa" tiene un amigo análogo en la biblioteca C llamado `atoi()` que convierte una cadena ASCII en un número entero).

La función `inet_aton()` es lo contrario, convirtiéndose de una cadena de puntos y números en una in_addr_t (que es el tipo de campo `s_addr` en el `in_addr de estructura`).

Finalmente, la función `inet_addr()` es una función más antigua que hace básicamente lo mismo que `inet_aton()`. Teóricamente está en desuso, pero lo verás mucho y la policía no vendrá a por ti si lo usas.

Valor devuelto

`inet_aton()` devuelve un valor distinto de cero si la dirección es válida, y devuelve cero si la dirección no es válida.

`inet_ntoa()` devuelve la cadena de puntos y números en un búfer estático que se sobrescribe con cada llamada a la función.

`inet_addr()` devuelve la dirección como un `in_addr_t`, o `-1` si hay un error. (Ese es el mismo resultado que si intentara convertir la cadena "`255.255.255.255`", que es una dirección IP válida. Es por eso `que inet_aton()` es mejor).

Ejemplo

```
estructura sockaddr_in antílope;
char *some_addr;

inet_aton("10.0.0.1", &antílope.sin_addr);  almacenar IP en antílope

some_addr = inet_ntoa(antílope.sin_addr);  devolver la IP
printf("%s\n", some_addr);  imprime "10.0.0.1"

Y esta llamada es la misma que la llamada inet_aton(), arriba:
antílope.sin_addr.s_addr = inet_addr("10.0.0.1");
```

Consulte también

`inet_ntop()`, `inet_pton()`, `gethostbyname()`, `gethostbyaddr()`

9.14 `inet_ntop()`, `inet_pton()`

Convierta direcciones IP a un formato legible por humanos y viceversa.

Sinopsis

```
#include <arpa/inet.h>

const char *inet_ntop(int af, const void *src,
                      char *dst, tamaño socklen_t);

int inet_pton(int af, const char *src, void *dst);
```

Descripción

Estas funciones son para tratar con direcciones IP legibles por humanos y convertirlas a su representación binaria para su uso con diversas funciones y llamadas al sistema. La "n" significa "red" y la "p" significa "presentación". O "presentación de textos". Pero puedes pensar en ello como "imprimible". "ntop" es "de red a imprimible". ¿Ver?

A veces no quieres mirar una pila de números binarios cuando miras una dirección IP. Lo quieres en un bonito formato imprimible, como `192.0.2.180`, o `2001:db8:8714:3a90::12`. En ese caso, `inet_ntop()` es para ti.

`inet_ntop()` toma la familia de direcciones en el `parámetro af` (ya sea `AF_INET` o `AF_INET6`). El parámetro `src` debe ser un puntero a un `in_addr de estructura` o `a un in6_addr de estructura` que contenga la dirección que desea convertir en una cadena. Por último, `dst` y `size` son el puntero a la cadena de destino y la longitud máxima de esa cadena.

¿Cuál debe ser la longitud máxima de la `cadena dst`? ¿Cuál es la longitud máxima de las direcciones IPv4 e IPv6? Afortunadamente, hay un par de macros para ayudarte. Las longitudes máximas son: `INET_ADDRSTRLEN` y `INET6_ADDRSTRLEN`.

Otras veces, es posible que tenga una cadena que contenga una dirección IP en forma legible y desee empaquetarla en un `sockaddr_in de estructura` o en un `sockaddr_in6 de estructura`. En ese caso, la función opuesta `inet_pton()` es lo que buscas.

`inet_pton()` también toma una familia de direcciones (ya sea `AF_INET` o `AF_INET6`) en el parámetro `af`. El parámetro `src` es un puntero a una cadena que contiene la dirección IP en forma imprimible. Por último, el `parámetro dst` señala dónde se debe almacenar el resultado, que probablemente sea un `in_addr de estructura` o un `in6_addr de estructura`.

Estas funciones no realizan búsquedas de DNS, necesitarás `getaddrinfo()` para eso.

Valor devuelto

`inet_ntop()` devuelve el parámetro `dst` en caso de éxito, o `NULL` en caso de error (y `se establece errno`).

`inet_pton()` devuelve `1` en caso de éxito. Devuelve `-1` si hubo un error (`se establece errno`) o `0` si la entrada no es una dirección IP válida.

Ejemplo

```
Demostración de IPv4 de inet_ntop() y inet_pton()

struct sockaddr_in sa;
char str[INET_ADDRSTRLEN];

almacenar esta dirección IP en sa:
inet_pton(AF_INET, "192.0.2.33", &(sa.sin_addr));

Ahora recupéralo e imprímelo
inet_ntop(AF_INET, & sa.sin_addr), str, INET_ADDRSTRLEN);

printf("%s\n", str);  imprime "192.0.2.33"
```

```
Demostración de IPv6 de inet_ntop() y inet_pton()
(básicamente lo mismo, excepto con un montón de 6 lanzados por ahí)

struct sockaddr_in6 sa;
char str[INET6_ADDRSTRLEN];

almacenar esta dirección IP en sa:
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &(sa.sin6_addr));

Ahora recupéralo e imprímelo
inet_ntop(AF_INET6, & sa.sin6_addr), str, INET6_ADDRSTRLEN);

printf("%s\n", str);  Impresiones "2001:DB8:8714:3A90::12"
```

```
Función auxiliar que puede utilizar:

Convierta una dirección struct sockaddr en una cadena, IPv4 e IPv6:

char *get_ip_str(const struct sockaddr *sa, char *s, size_t maxlen)
{
    switch(sa->sa_family) {
        case AF_INET:
            inet_ntop(AF_INET,
                    &(((struct sockaddr_in *)sa)->sin_addr), s,
                    maxlen);
            descanso;

        Caso AF_INET6:
            inet_ntop(AF_INET6,
                    &(((struct sockaddr_in6 *)sa)->sin6_addr), s,
                    maxlen);
            descanso;
```

```
19
20          Predeterminado:
21              strncpy(s, "AF desconocido",
22              maxlen); devuelve NULL;
23      }
24
25      devolver s;
26  }
```

Consulte también

`getaddrinfo()`

9.15 `escuchar()`

Decirle a un socket que escuche las conexiones entrantes

Sinopsis

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

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

Descripción

Puede tomar su descriptor de socket (hecho con la llamada al `sistema socket())` y decirle que escuche las conexiones entrantes. Esto es lo que diferencia a los servidores de los clientes, chicos.

El parámetro `de backlog` puede significar un par de cosas diferentes dependiendo del sistema en el que se encuentre, pero a grandes rasgos es el número de conexiones pendientes que puede tener antes de que el kernel comience a rechazar las nuevas. Por lo tanto, a medida que llegan las nuevas conexiones, debe aceptarlas rápidamente para que el trabajo pendiente no se llene. Intente configurarlo en 10 más o menos, y si sus clientes comienzan a recibir "Conexión rechazada" bajo una carga pesada, configúrelo más alto.

Antes de llamar a `listen()`, tu servidor debe llamar a `bind()` para conectarse a un número de puerto específico. Ese número de puerto (en la dirección IP del servidor) será al que se conecten los clientes.

Valor devuelto

Devuelve cero en caso de éxito, o `-1` en caso de error (y `errno` se establecerá en consecuencia).

Ejemplo

```
1   struct addrinfo sugerencias, *res;
2   int calcetín;
3
4   Primero, cargue las estructuras de direcciones con getaddrinfo():
5
6   memset(&hints, 0, sizeof hints);
7   Pistas.ai_family = AF_UNSPEC;  usar IPv4 o IPv6, lo que ocurra
8   Pistas.ai_socktype = SOCK_STREAM;
9   Consejos.ai_flags = AI_PASSIVE;   rellene mi IP por mí
10
11  getaddrinfo(NULL, "3490", &hints, &res);
12
```

```
Hacer un enchufe:

sockfd = socket(res->ai_family, res->ai_socktype,
    res->ai_protocol);

 Enlazarlo al puerto que pasamos a getAddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen); escuchar

(calcetín, 10);  Configurar sockfd para que sea un socket

de servidor
```

Consulte también

`accept()`, `bind()`, `socket()`

9.16 `perror()`, `strerror()`

Imprimir un error como una cadena legible por humanos

Sinopsis

```
#include <stdio.h>
#include <cadena.h>    para strtube()

void perror(const char *s);
char *strerror(int errnum);
```

Descripción

Dado que muchas funciones devuelven `-1` en caso de error y establecen el valor de la variable `errno` como un número, seguro que sería bueno si pudieras imprimirlo fácilmente en una forma que tuviera sentido para ti.

Afortunadamente, `perror()` hace eso. Si desea que se imprima más descripción antes del error, puede apuntar el parámetro `s` a ella (o puede dejar `s s` como `NULL` y no se imprimirá nada adicional).

En pocas palabras, esta función toma `valores errno`, como `ECONNRESET,` y los imprime bien, como "Restablecimiento de conexión por par".

La función `strerror()` es muy similar a `perror(),` excepto que devuelve un puntero a la cadena del mensaje de error para un valor dado (normalmente se pasa la variable `errno`).

Valor devuelto

`strerror()` devuelve un puntero a la cadena del mensaje de error.

Ejemplo

```
int s;

s = socket(PF_INET, SOCK_STREAM, 0);

if (s == -1) {  se ha producido algún error
```

```
    imprime "socket error: " + el mensaje de error:
    perror("error de socket");
}

semejantemente:
if (escucha(s, 10) == -1) {
    Esto imprime "An ERROR: " + el mensaje de error de errno:
    printf("un error: %s\n", strerror(errno));
}
```

Consulte también

`errno`

9.17 `encuesta()`

Pruebe los eventos en varios sockets simultáneamente

Sinopsis

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

int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

Descripción

Esta función es muy similar a `select()` en el sentido de que ambos observan conjuntos de descriptores de archivos para eventos, como datos entrantes listos para `recv()`, socket listo para `enviar` datos () a, datos fuera de banda listos para `recv()`, errores, etc.

La idea básica es que se pasa una matriz de `nfds struct pollfd`s en `ufds`, junto con un tiempo de espera en milisegundos (1000 milisegundos en un segundo). El `tiempo de espera` puede ser negativo si desea esperar para siempre. Si no ocurre ningún evento en ninguno de los descriptores del socket por el tiempo de espera, `poll()` devolverá.

Cada elemento de la matriz de `struct pollfds representa` un descriptor de socket y contiene los siguientes campos:

```
struct pollfd {
    Int Fd;         El descriptor de socket
    corto Eventos;  Mapa de bits de los eventos que nos interesan
     reventos cortos;  after return, mapa de bits de los eventos que ocurrieron
};
```

Antes de llamar a `poll()`, cargue `fd` con el descriptor de socket (si establece `fd` en un número negativo, esta `estructura pollfd` se ignora y su campo `revents` se establece en cero) y luego construya el campo events mediante OR bit a bit las siguientes macros:

Macro	Descripción
`POLÍN`	Avisarme cuando los datos estén listos para `recv()` en este zócalo.
`ENCUESTA`	Avísame cuando pueda `send()` datos a este socket sin bloquear.
`POLLPRI`	Avisarme cuando los datos fuera de banda estén listos para `recv()` en este zócalo.

Una vez que `se devuelve la llamada` poll(), el campo `revents` se construirá como un OR bit a bit de los campos anteriores, lo que le indicará qué descriptores realmente han tenido ese evento. Además, estos otros campos pueden estar presentes:

Macro	Descripción
`POLLERR`	Se ha producido un error en este socket.
`POLLHUP`	El lado remoto de la conexión se colgó.
`POLLNVAL`	Algo andaba mal con el descriptor de socket `Fd`—¿Quizás no está inicializado?

Valor devuelto

Devuelve el número de elementos de la matriz `ufds` en los que se ha producido un evento; puede ser cero si se ha agotado el tiempo de espera. También devuelve `-1` en caso de error (y `errno` se establecerá en consecuencia).

Ejemplo

```
Int S1, T2;
Int Rv;
carbonizar buf1[256], buf2[256];
Estructura POLLFD UFDs[2];

S1 = enchufe(PF_INET, SOCK_STREAM, 0);
T2 = enchufe(PF_INET, SOCK_STREAM, 0);

Supongamos que hemos conectado ambos a un servidor en este punto
Conectar(S1, ...)...
Conectar(S2, ...)...

Configure la matriz de descriptores de archivo.
//
En este ejemplo, queremos saber cuándo hay normal o
datos fuera de banda (OOB) listos para ser recv()'d...

UFDs[0].Fd = S1;
UFDs[0].Eventos = POLÍN | POLLPRI; comprobar si es normal o OOB

UFDs[1].Fd = T2;
UFDs[1].Eventos = POLÍN; Compruebe solo los datos normales

Espere eventos en los sockets, tiempo de espera de 3,5 segundos
Rv = encuesta(UFDs, 2, 3500);

si (Rv == -1) {
    perror("encuesta"); Se ha producido un error en poll()
} de lo contrario, si (Rv == 0) {
    printf("¡Se produjo el tiempo muerto! No hay datos después de 3,5 segundos.\n");
} más {
    Comprobar si hay eventos en S1:
    si (UFDs[0].revents & POLÍN) {
        recv(S1, buf1, tamañode buf1, 0); Recibir datos normales
    }
    si (UFDs[0].revents & POLLPRI) {
        recv(S1, buf1, tamañode buf1, MSG_OOB); Datos fuera de banda
    }

    Comprobar si hay eventos en S2:
```

```
    Si (UFDS[1].revents & POLLIN) {
        recv(s1, buf2, tamañode buf2,
        0);
}
```

Consulte también

`seleccionar()`

9.18 `recv()`, `recvfrom()`

Recepción de datos en un socket

Sinopsis

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

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

Descripción

Una vez que tenga un socket encendido y conectado, puede leer los datos entrantes desde el lado remoto usando el
`recv()` (para sockets TCP `SOCK_STREAM`) y `recvfrom()` (para sockets UDP `SOCK_DGRAM`).

Ambas funciones toman el descriptor de socket `s`, un puntero al búfer buf, el tamaño (en bytes) del búfer `len`, y un conjunto de `indicadores` que controlan cómo funcionan las funciones.

Además, `el recvfrom()` toma un `struct sockaddr*`, `de` quien le dirá de dónde provienen los datos, y completará `fromlen` con el tamaño de `struct sockaddr`. (También debe inicializar `fromlen` para ser del tamaño de `from` o `struct sockaddr`.)

Entonces, ¿qué banderas maravillosas puedes pasar a esta función? Estos son algunos de ellos, pero debe consultar las páginas de manual locales para obtener más información y lo que realmente se admite en su sistema. Bit a bit-o estos juntos, o simplemente establece `las banderas` en `0` si quieres que sea un `recv() de vainilla normal`.

Macro	Descripción
`MSG_OOB`	Recepción de datos fuera de banda. Esta es la forma de obtener los datos que se le han enviado con el indicador `MSG_OOB` en `send()`. Como lado receptor, se le habrá generado una señal `SIGURG` que le indica que hay datos urgentes. En tu manejador para esa señal, podrías llamar a `recv()` con este indicador `MSG_OOB`.
`MSG_PEEK`	Si quieres llamar a `recv()` "solo para fingir", puedes llamarlo con esta bandera. Esto te dirá lo que está esperando en el búfer cuando llames a `recv()` "de verdad" (es decir, *sin* el indicador `MSG_PEEK`). Es como un adelanto de la próxima llamada `a recv()`.
`MSG_WAITALL`	Dígale `a recv()` que no regrese hasta que se hayan recibido todos los datos que especificó en el `parámetro len`. Sin embargo, ignorará sus deseos en circunstancias extremas, como si una señal interrumpe la llamada o si ocurre algún error o si el lado remoto cierra la conexión, etc. No te enfades con eso.

Cuando llamas a `recv()`, se bloqueará hasta que haya algunos datos para leer. Si no desea bloquear, configure el socket para que no bloquee o verifique con `select()` o `poll()` para ver si hay datos entrantes antes de llamar

`recv()` o `recvfrom()`.

Valor devuelto

Devuelve el número de bytes realmente recibidos (que puede ser menor que el solicitado en el `parámetro len`), o `-1` en caso de error (y `errno` se establecerá en consecuencia).

Si el lado remoto ha cerrado la conexión, `recv()` devolverá `0`. Este es el método normal para determinar si el lado remoto ha cerrado la conexión. ¡La normalidad es buena, rebelde!

Ejemplo

```
sockets de flujo y recv()

struct addrinfo sugerencias, *res;
int calcetín;
Char Buf[512];
int byte_count;

Obtener información del host, hacer socket y conectarlo
memset(&hints, 0, sizeof hints);
Pistas.ai_family = AF_UNSPEC;  use IPv4 o IPv6, lo que
indique.ai_socktype = SOCK_STREAM;
getaddrinfo("www.example.com", "3490", &hints, &res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol); conectar
(sockfd, res->ai_addr, res->ai_addrlen);

¡Muy bien! Ahora que estamos conectados, ¡podemos recibir algunos datos!
byte_count = recv(calcetín, buf, tamañode buf, 0);
printf("recv()'d %d bytes de datos en buf\n", byte_count);
```

```
sockets de datagramas y recvfrom()

struct addrinfo sugerencias, *res;
int calcetín;
int byte_count;
socklen_t fromlen;
struct sockaddr_storage addr;
Char Buf[512];
char ipstr[INET6_ADDRSTRLEN];

obtener información del host, crear un socket, vincularlo al puerto 4950
memset(&hints, 0, sizeof hints);
Pistas.ai_family = AF_UNSPEC;  usar IPv4 o IPv6, lo que ocurra
Pistas.ai_socktype = SOCK_DGRAM;
Pistas.ai_flags = AI_PASSIVE; getaddrinfo
(NULL, "4950", & hints, & res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);

no es necesario accept(), solo recvfrom():

fromlen = tamañode addr;
byte_count = recvfrom(sockfd, buf, sizeof buf, 0, &addr, &fromlen);

printf("recv()'d %d bytes de datos en buf\n", byte_count);
```

```
26 printf("de la dirección IP %s\n",
27     inet_ntop(addr.ss_family,
28         addr.ss_family == AF_INET?
29             ((struct  sockaddr_in *)&addr)->sin_addr:
30             ((struct sockaddr_in6 *)&addr)->sin6_addr,
31         ipstr, tamaño de ipstr);
```

Consulte también

`send()`, `sendto()`, `select()`, `poll()`, Bloqueo

9.19 `seleccionar()`

Compruebe si los descriptores de sockets están listos para leer/escribir

Sinopsis

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

int select(int n, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

FD_SET(int fd, fd_set *set);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_ZERO (fd_set *set);
```

Descripción

La función `select()` te da una forma de comprobar simultáneamente varios sockets para ver si tienen datos esperando ser `recv()`d, o si puedes `enviarles` datos sin bloquear, o si ha ocurrido alguna excepción.

Rellenas tus conjuntos de descriptores de socket usando las macros, como `FD_SET()`, arriba. Una vez que tenga el conjunto, páselo a la función como uno de los siguientes parámetros: `readfds` si desea saber cuándo alguno de los sockets del conjunto está listo para `recibir datos de recv()`, `writefds` si alguno de los sockets está listo para `enviar` datos de () a, y/o `exceptfds` si necesita saber cuándo ocurre una excepción (error) en cualquiera de los sockets. Cualquiera o todos estos parámetros pueden ser `NULL` si no está interesado en esos tipos de eventos. Después de `que select()` devuelva, los valores de los conjuntos se cambiarán para mostrar cuáles están listos para leer o escribir, y cuáles tienen excepciones.

El primer parámetro, `n,` es el descriptor de socket con el número más alto (son solo `enteros,` ¿recuerdas?) más uno.

Por último, la `struct timeval`, `timeout`, al final: esto le permite decirle `a select()` cuánto tiempo verificar estos conjuntos. Volverá después del tiempo de espera, o cuando ocurra un evento, lo que ocurra primero. La `estructura timeval` tiene dos campos: `tv_sec` es el número de segundos, al que se suma `tv_usec`, el número de microsegundos (1.000.000 de microsegundos en un segundo).

Las macros auxiliares hacen lo siguiente:

Macro	Descripción
`FD_SET(int fd, fd_set *set); FD_CLR(int fd, fd_set *set);`	Agregue `fd` al `conjunto`. Elimine fd `del` conjunto.

Macro	Descripción
`FD_ISSET(int fd, fd_set *set); FD_ZERO(fd_set *conjunto);`	Retorna true si `fd` está en el `conjunto`. Borra todas las entradas del `conjunto`.

Nota para los usuarios de Linux: el `select()` de Linux puede devolver "listo para leer" y luego no estar realmente listo para leer, lo que hace que la llamada a `read()` posterior se bloquee. Puede solucionar este error configurando `O_NONBLOCK` marca en el socket receptor para que genere errores con `EWOULDBLOCK`, y luego ignorando este error si ocurre. Consulte la `página del comando man` fcntl() para obtener más información sobre cómo configurar un socket para que no esté bloqueado.

Valor devuelto

Devuelve el número de descriptores del conjunto en caso de éxito, `0` si se alcanzó el tiempo de espera o `-1` en caso de error (y `errno` se establecerá en consecuencia). Además, los conjuntos se modifican para mostrar qué sockets están listos.

Ejemplo

```
int S1, S2, N;
fd_set readfds;
Estructura TimeVal
TV;
char buf1[256], buf2[256];

Supongamos que hemos conectado ambos a un servidor en este punto
s1 = zócalo(...);
s2 = zócalo(...);
Conectar(S1, ...)...
Conectar(S2, ...)...

Despeja el conjunto con anticipación
FD_ZERO(&readfds);

Agregue nuestros descriptores al conjunto
FD_SET(s1, &readfds);
FD_SET(s2, &readfds);

Dado que obtuvimos S2 en segundo lugar, es el "mayor", así que lo usamos para
El parámetro n en select()
n = s2 + 1;

Espere hasta que cualquiera de los sockets tenga datos listos para ser recv()d
(tiempo de espera 10,5 segundos)
televisión.tv_sec = 10;
televisión.tv_usec = 500000;
rv = select(n, &readfds, NULL, NULL, &tv);

if (rv == -1) {
    perror("seleccionar");  Se ha producido un error en select()
} else if (rv == 0) {
    printf("¡Se ha agotado el tiempo de espera! No hay datos después de 10,5
    segundos.\n");
} else {
    Uno o ambos descriptores tienen datos
    if (FD_ISSET(s1, &readfds)) {
        recv(s1, buf1, tamañode buf1, 0);
```

```
39          recv(s2, buf2, tamañode buf2,
40      }
41  }
```

Consulte también

`encuesta()`

9.20 `setsockopt()`, `getsockopt()`

Establecer varias opciones para un zócalo

Sinopsis

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

int getsockopt(int s, int level, int optname, void *optval,
               socklen_t *optlen);
int setsockopt(int s, int level, int optname, const void *optval,
               socklen_t optlen);
```

Descripción

Los sockets son bestias bastante configurables. De hecho, son tan configurables que ni siquiera voy a cubrirlo todo aquí. De todos modos, probablemente dependa del sistema. Pero hablaré de lo básico.

Obviamente, estas funciones obtienen y establecen ciertas opciones en un socket. En una máquina Linux, toda la información sobre sockets se encuentra en la página de manual para socket en la sección 7. (Tipo: "`man 7 socket`" para obtener todas estas golosinas).

En cuanto a los parámetros, `s` es el socket del que está hablando, level debe establecerse en `SOL_SOCKET`. A continuación, establezca el `optname` en el nombre que le interese. De nuevo, consulta tu página de manual para ver todas las opciones, pero aquí tienes algunas de las más divertidas:

`Nombre de la opción`	Descripción
`SO_BINDTODEVICE`	Vincule este socket a un nombre de dispositivo simbólico como `eth0` en lugar de usar `bind()` para vincularlo a una dirección IP. Escriba el comando `ifconfig` en Unix para ver los nombres de los dispositivos.
`SO_REUSEADDR`	Permite que otros sockets `enlacen()` a este puerto, a menos que ya haya un socket de escucha activa vinculado al puerto. Esto le permite sortear esos mensajes de error "Dirección ya en uso" cuando intenta reiniciar su servidor después de un bloqueo.
`SO_BROADCAST`	Permite que los sockets de datagramas UDP (`SOCK_DGRAM`) envíen y reciban paquetes enviados hacia y desde la dirección de difusión. ¡¡No hace nada, *NADA!!*—¡A los sockets de flujo TCP! ¡Jajaja!

En cuanto al parámetro `optval`, suele ser un puntero a un `int` que indica el valor en cuestión. En el caso de los booleanos, cero es falso y distinto de cero es verdadero. Y eso es un hecho absoluto, a menos que sea diferente en su sistema. Si no hay ningún parámetro que pasar, `optval` puede ser `NULL`.

El parámetro final, `optlen`, debe establecerse en la longitud de `optval`, probablemente `sizeof(int)`, pero varía según la opción. Tenga en cuenta que en el caso de `getsockopt()`, este es un puntero a un `socklen_t` y especifica el objeto de tamaño máximo que se almacenará en `optval` (para evitar desbordamientos de búfer). Y `getsockopt()` modificará el valor de `optlen` para reflejar el número de bytes realmente establecidos.

Advertencia: en algunos sistemas (especialmente Sun y Windows), la opción puede ser un `char` en lugar de un `int`, y se establece, por ejemplo, en un valor de carácter `de '1'` en lugar de un `valor int` de `1`. De nuevo, consulte sus propias páginas de manual para obtener más información con "`man setsockopt`" y "`man 7 socket`".

Valor devuelto

Devuelve cero en caso de éxito, o `-1` en caso de error (y `errno` se establecerá en consecuencia).

Ejemplo

```
int optval;
int optlen;
char *optval2;

Establezca SO_REUSEADDR en un socket en true (1):
optval = 1;
setsoccopt(s1, SOL_SOCKET, SO_REUSEADDR, & optval, sizeof optval);

Vincular un socket a un nombre de dispositivo (es posible que no funcione en
todos los sistemas):
optval2 = "eth1";  4 bytes de largo, por lo que 4, a continuación:
setsoccopt(S2, SOL_SOCKET, SO_BINDTODEVICE, Optval2, 4);

Vea si el indicador SO_BROADCAST está establecido:
getsoccopt(S3, SOL_SOCKET, SO_BROADCAST, & Optval, &
Optlen); if (optval != 0) {
    print("SO_BROADCAST habilitado en S3!\n");
}
```

Consulte también

`fcntl()`

9.21 `send()`, `sendto()`

Envío de datos a través de un socket

Sinopsis

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

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

Descripción

Estas funciones envían datos a un socket. En términos generales, `send()` se usa para TCP `SOCK_STREAM` sockets conectados, y `sendto()` se usa para UDP `SOCK_DGRAM` sockets de datagramas no conectados. Con los sockets no conectados, debes especificar el destino de un paquete cada vez que envías uno, y es por eso que los últimos parámetros de `sendto()` definen hacia dónde va el paquete.

Tanto con `send()` como `con sendto()`, el parámetro `s` es el socket, `buf` es un puntero a los datos que desea enviar, `len` es el número de bytes que desea enviar y `flags` le permite especificar más información sobre cómo se enviarán los datos. Establezca `las banderas` en cero si desea que sean datos "normales". Estas son algunas de las banderas más utilizadas, pero consulte las páginas del manual local `send()` para obtener más detalles:

Macro	Descripción
`MSG_OOB`	Enviar como datos "fuera de banda". TCP admite esto, y es una forma de decirle al sistema receptor que estos datos tienen una prioridad más alta que los datos normales. El receptor recibirá la señal `SIGURG` y entonces podrá recibir estos datos sin recibir primero el resto de los datos normales de la cola.
`MSG_DONTROUTE`	No envíes estos datos a través de un enrutador, simplemente mantenlos locales.
`MSG_DONTWAIT`	Si `send()` se bloquea porque el tráfico saliente está obstruido, haz que devuelva `EAGAIN.` Esto es como un "habilitar el no bloqueo solo para este envío". Consulte la sección sobre bloqueo para obtener más detalles.
`MSG_NOSIGNAL`	Si `envías ()` a un host remoto que ya no es `recv()`, normalmente obtendrás la señal `SIGPIPE.` Agregar este indicador evita que se genere esa señal.

Valor devuelto

Devuelve el número de bytes realmente enviados, o `-1` en caso de error (y `errno` se establecerá en consecuencia). Tenga en cuenta que el número de bytes realmente enviados puede ser menor que el número que le pidió que enviara. Consulte la sección sobre el manejo de `send()`s parciales para obtener una función auxiliar para evitar esto.

Además, si el socket ha sido cerrado por ambos lados, el proceso que llama a `send()` obtendrá la señal `SIGPIPE.` (A menos que `send()` se llamó con el indicador `MSG_NOSIGNAL`.)

Ejemplo

```
int spatula_count = 3490;
char *secret_message = "El queso está en la tostadora";

int stream_socket, dgram_socket;
struct sockaddr_in dest;
int temp;

primero con sockets de flujo TCP:

Suponga que los enchufes están hechos y conectados
stream_socket = zócalo(...
conectar(stream_socket, ...

Convertir en orden de bytes de red
temp = htonl(spatula_count);
Enviar datos normalmente:
send(stream_socket, & temp, sizeof temp, 0);

Enviar mensaje secreto fuera de banda:
send(stream_socket, secret_message, strlen(secret_message)+1,
        MSG_OOB);

ahora con sockets de datagramas UDP:
getaddrinfo(...
dest = ... // supongamos que "dest" contiene la dirección del destino
dgram_socket = zócalo(...
```

```
28 Enviar mensaje secreto normalmente:
29 sendto(dgram_socket, secret_message, strlen(secret_message)+1, 0,
30        (struct sockaddr*)&hand,  tamaño de la mano);
```

Consulte también

`recv()`, `recvfrom()`

9.22 `shutdown()`

Detener más envíos y recepciones en un socket

Sinopsis

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

int shutdown(int s, int cómo);
```

Descripción

¡Eso es todo! ¡Lo he tenido! No se permiten más `send()`s en este socket, ¡pero todavía quiero recibir datos de `recv()` en él! ¡O viceversa! ¿Cómo puedo hacer esto?

Cuando `cierras ()` un descriptor de socket, cierra ambos lados del socket para leer y escribir, y libera el descriptor de socket. Si solo quieres cerrar un lado o el otro, puedes usar esta `llamada` shutdown().

En cuanto a los parámetros, `s` es obviamente el socket en el que desea realizar esta acción, y qué acción es se puede especificar con el parámetro `how`. ¿`Cómo` se puede `SHUT_RD` para evitar más `recv()`s, `SHUT_WR` para prohibir más `send()`s, o `SHUT_RDWR` para hacer ambas cosas.

Tenga en cuenta que `shutdown()` no libera el descriptor de socket, por lo que aún tiene que `cerrar eventualmente ()` el socket incluso si se ha apagado por completo.

Esta es una llamada al sistema que se usa con poca frecuencia.

Valor devuelto

Devuelve cero en caso de éxito, o `-1` en caso de error (y `errno` se establecerá en consecuencia).

Ejemplo

```
1 int s = socket(PF_INET, SOCK_STREAM, 0);
2
3 // ... hacer algunos envíos ()s y cosas por aquí...
4
5 Y ahora que hemos terminado, no permitas más envíos()s:
6 parada(s), SHUT_WR);
```

Consulte también

`cerrar()`

9.23 `socket()`

Asignación de un descriptor de socket

Sinopsis

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

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

Descripción

Devuelve un nuevo descriptor de socket que puede usar para hacer cosas de sockety. Esta es generalmente la primera llamada en el enorme proceso de escribir un programa de socket, y puede usar el resultado para llamadas posteriores a `listen()`, `bind()`, `accept()` o una variedad de otras funciones.

En el uso habitual, se obtienen los valores de estos parámetros de una llamada a `getaddrinfo()`, como se muestra en el siguiente ejemplo. Pero puedes rellenarlos a mano si realmente quieres.

Parámetro	Descripción
`dominio`	`dominio` Describe el tipo de socket que le interesa. Esto puede, créeme, ser una amplia variedad de cosas, pero dado que esta es una guía de sockets, lo será `PF_INET` para IPv4, y `PF_INET6` para IPv6.
`tipo`	Además, el `tipo` puede ser varias cosas, pero probablemente lo establecerá en cualquiera de las dos `SOCK_STREAM` para sockets TCP fiables (`send()`, `recv()`) o bien `SOCK_DGRAM` para sockets UDP rápidos poco fiables (`sendto()`, `recvfrom()`). (Otro tipo de socket interesante es `SOCK_RAW` que se puede utilizar para construir paquetes a mano. Es genial).
`protocolo`	Por último, el `protocolo` indica qué protocolo usar con un determinado tipo de socket. Como ya he dicho, por ejemplo, `SOCK_STREAM` utiliza TCP. Afortunadamente para ti, cuando usas `SOCK_STREAM` o `SOCK_DGRAM`, puede establecer el protocolo en 0 y utilizará el protocolo adecuado automáticamente. De lo contrario, puede usar `getprotobyname()` para buscar el número de protocolo adecuado.

Valor devuelto

El nuevo descriptor de socket que se utilizará en las llamadas subsiguientes, o `-1` en caso de error (y `errno` se establecerá según corresponda).

Ejemplo

```
struct addrinfo sugerencias, *res;
int calcetín;

Primero, cargue las estructuras de direcciones con getaddrinfo():

memset(&hints, 0, sizeof hints);
Consejos.ai_family = AF_UNSPEC;   AF_INET, AF_INET6 o AF_UNSPEC
Pistas.ai_socktype = SOCK_STREAM;  SOCK_STREAM o SOCK_DGRAM

getaddrinfo("www.example.com", "3490", &hints, &res);

Crea un socket usando la información obtenida de getaddrinfo():
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

Consulte también

`accept()`, `bind()`, `getaddrinfo()`, `listen()`

9.24 `Struct Sockaddr` y amigos

Estructuras para el manejo de direcciones de Internet

Sinopsis

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

Todos los punteros a las estructuras de direcciones de socket a menudo se convierten
en
punteros a este tipo antes de su uso en diversas funciones y sistemas
Llamadas:

struct sockaddr {
    Corto sin firmar   sa_family;     Dirección: Familia, AF_xxx
    carbonizar         sa_data[14]; 14 bytes de dirección de protocolo
};


Sockets AF_INET IPv4:

struct sockaddr_in {
    corto             sin_family;    por ejemplo, AF_INET, AF_INET6
    Corto sin firmar  sin_port;      Por ejemplo, HTONS(3490)
    Estructura in_addr               sin_addr;Consulte Struct in_addr, a continuación
    carbonizar        sin_zero[8]; Ponga a cero esto si desea
};

struct in_addr {
    largo sin firmar s_addr;         cargar con inet_pton()
};


Sockets AF_INET6 IPv6:

struct sockaddr_in6 {
    u_int16_t        sin6_family;    Dirección: familia, AF_INET6
    u_int16_t        sin6_port;      Número de puerto, orden de
    red u_int32_t    sin6_flowinfo; Información de flujo IPv6
    Estructura in6_addr sin6_addr;  Dirección IPv6
    u_int32_t        sin6_scope_id; Id. de ámbito
};

struct in6_addr {
    char sin firmar  s6_addr[16];    cargar con inet_pton()
};


Estructura de sujeción de dirección de socket general, lo suficientemente grande como
para contener
Struct sockaddr_in o struct sockaddr_in6 datos:

struct sockaddr_storage {
```

```
    sa_family_t ss_family;      Dirección Familia

    Todo esto es relleno, específico de la implementación,
    ignóralo:
    carbonizar __ss_pad1[_SS_PAD1SIZE];
    int64_t    __ss_align;
};
```

Descripción

Estas son las estructuras básicas para todas las llamadas al sistema y las funciones que se ocupan de las direcciones de Internet. A menudo usarás `getaddrinfo()` para completar estas estructuras, y luego las leerás cuando tengas que hacerlo.

En la memoria, el `struct sockaddr_in` y `struct sockaddr_in6` comparten la misma estructura inicial que `struct sockaddr`, y puede lanzar libremente el puntero de un tipo al otro sin ningún daño, excepto el posible final del universo.

Es broma sobre eso del fin del universo... Si el universo se acaba cuando lanzas una `estructura sockaddr_in*`
a un `calcetín de estructura*`, te prometo que es pura coincidencia y ni siquiera deberías preocuparte por eso.

Entonces, con eso en mente, recuerde que siempre que una función diga que toma un `struct sockaddr*`, puede convertir su `struct sockaddr_in*`, `struct sockaddr_in6*` o `struct sockaddr_storage*` a ese tipo con facilidad y seguridad.

`struct sockaddr_in` es la estructura utilizada con las direcciones IPv4 (por ejemplo, "192.0.2.10"). Contiene una familia de direcciones (`AF_INET`), un puerto en `sin_port` y una dirección IPv4 en `sin_addr`.

También está este campo `sin_zero` en `struct sockaddr_in` que algunas personas afirman que debe establecerse en cero. Otras personas no afirman nada al respecto (la documentación de Linux ni siquiera lo menciona), y ponerlo a cero no parece ser realmente necesario. Así que, si te apetece, ponlo a cero usando `memset()`.

Ahora, esa `estructura in_addr` es una bestia extraña en diferentes sistemas. A veces es una unión loca con todo tipo de `#defines y otras tonterías.` Pero lo que debes hacer es usar solo el `campo s_addr` en esta estructura, porque muchos sistemas solo implementan ese.

`struct sockaddr_in6` y `struct in6_addr` son muy similares, excepto que se usan para IPv6.

`struct sockaddr_storage` es una estructura que puedes pasar a `accept()` o `recvfrom()` cuando intentas escribir código independiente de la versión de IP y no sabes si la nueva dirección va a ser IPv4 o IPv6. La `estructura sockaddr_storage` estructura es lo suficientemente grande como para contener ambos tipos, a diferencia del calcetín de estructura pequeña original.

Ejemplo

```
IPv4:

struct sockaddr_in ip4addr;
int s;

ip4addr.sin_family = AF_INET;
ip4addr.sin_port = htons(3490);
inet_pton(AF_INET, "10.0.0.1", &ip4addr.sin_addr);

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip4addr, sizeof ip4addr);
```

```
IPv6:

struct sockaddr_in6 ip6addr;
```

```
int s;

IP6addr.sin6_family = AF_INET6;
IP6addr.sin6_port = htons(4950);
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &ip6addr.sin6_addr);

s = socket(PF_INET6, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip6addr, sizeof ip6addr);
```

Consulte también

`aceptar()`, `vincular()`, `conectar()`, `inet_aton()`, `inet_ntoa()`

Capítulo 10

Más referencias

¡Has llegado hasta aquí y ahora estás pidiendo a gritos más! ¿Dónde más puedes ir para aprender más sobre todo esto?

10.1 Libros

Para libros de papel pulpa de la vieja escuela que se sostienen en la mano, pruebe algunos de los siguientes libros excelentes. Estos redirigen a enlaces de afiliados con un librero popular, lo que me da buenos sobornos. Si simplemente te sientes generoso, puedes PayPal una donación para `beej@beej.us`. `:-)`

Programación de redes Unix, volúmenes 1-2 por W. Richard Stevens. Publicado por Addison-Wesley Pro- fessional y Prentice Hall. ISBN de los volúmenes 1 y 2: 978-0131411555[1], 978-0130810816[2].

Interconexión con TCP/IP, volumen I por Douglas E. Comer. Publicado por Pearson. ISBN 978-0136085300 [3].

TCP/IP Illustrated, volúmenes 1-3 por W. Richard Stevens y Gary R. Wright. Publicado por Addison Wesley. ISBN para los volúmenes 1, 2 y 3 (y un conjunto de 3 volúmenes): 978-0201633467[4], 978-0201633542[5], 978-
0201634952[6], (978-0201776317[7]).

Administración de redes TCP/IP por Craig Hunt. Publicado por O'Reilly & Associates, Inc. ISBN 978-0596002978 [8].

Programación avanzada en el entorno UNIX por W. Richard Stevens. Publicado por Addison Wesley. ISBN 978-0321637734[9].

10.2 Referencias Web

En la web:

BSD Sockets: A Quick And Dirty Primer[10] (¡información de programación del sistema Unix, también!)

El socket Unix FAQ[11]

PREGUNTAS

FRECUENTES SOBRE

TCP/IP[12]

[1]https://beej.us/guide/url/unixnet1
[2]https://beej.us/guide/url/unixnet2
[3]https://beej.us/guide/url/intertcp1
[4]https://beej.us/guide/url/tcpi1 [5]
https://beej.us/guide/url/tcpi2
[6]https://beej.us/guide/url/tcpi3
[7]https://beej.us/guide/url/tcpi123
[8]https://beej.us/guide/url/tcpna
[9]https://beej.us/guide/url/advunix
[10]https://cis.temple.edu/~giorgio/old/cis307s96/readings/docs/sockets.html
[11]https://developerweb.net/?f=70
[12]http://www.faqs.org/faqs/internet/tcp-ip/tcp-ip-faq/part1/

118

Preguntas frecuentes sobre Winsock[13]

Y aquí hay algunas páginas relevantes de Wikipedia:

Sockets Berkeley[14]

Protocolo de Internet

(IP)[15]

Protocolo de control de transmisión

(TCP)[16] **Protocolo de datagramas de**

usuario (UDP)[17]

Cliente-Servidor[18]

Serialización[19] (empaquetado y desempaquetado de datos)

10.3 RFC

RFCs[20]: ¡la verdadera suciedad! Se trata de documentos que describen los números asignados, las API de programación y los protocolos que se utilizan en Internet. He incluido enlaces a algunos de ellos aquí para su disfrute, así que tome un cubo de palomitas de maíz y póngase su gorra de pensar:

RFC 1[21] —El primer RFC; esto le da una idea de cómo era "Internet" justo cuando estaba cobrando vida, y una idea de cómo se estaba diseñando desde cero. (¡Este RFC está completamente obsoleto, obviamente!)

RFC 768[22] —El protocolo de datagramas de usuario (UDP)

RFC 791[23] —El Protocolo de Internet (IP)

RFC 793[24] —El protocolo de control de transmisión (TCP)

RFC 854[25] —El protocolo Telnet

RFC 959[26] —Protocolo de transferencia de archivos (FTP)

RFC 1350[27] —El Protocolo Trivial de Transferencia de Archivos (TFTP)

RFC 1459[28] —Protocolo de chat de retransmisión de Internet (IRC)

RFC 1918[29] —Asignación de direcciones para Internets privadas

RFC 2131[30] —Protocolo de configuración dinámica de host (DHCP)

RFC 9110[31] —Protocolo de transferencia de hipertexto

(HTTP) **RFC 2821**[32] —Protocolo simple de

transferencia de correo (SMTP) **RFC 3330**[33] —

Direcciones IPv4 de uso especial

[13]https://tangentsoft.net/wskfaq/
[14]https://en.wikipedia.org/wiki/Berkeley_sockets
[15]https://en.wikipedia.org/wiki/Internet_Protocol
[16]https://en.wikipedia.org/wiki/Transmission_Control_Protocol
[17]https://en.wikipedia.org/wiki/User_Datagram_Protocol
[18]https://en.wikipedia.org/wiki/Client-server
[19]https://en.wikipedia.org/wiki/Serialization
[20]https://www.rfc-editor.org/
[21]https://tools.ietf.org/html/rfc1
[22]https://tools.ietf.org/html/rfc768
[23]https://tools.ietf.org/html/rfc791
[24]https://tools.ietf.org/html/rfc793
[25]https://tools.ietf.org/html/rfc854
[26]https://tools.ietf.org/html/rfc959
[27]https://tools.ietf.org/html/rfc1350
[28]https://tools.ietf.org/html/rfc1459
[29]https://tools.ietf.org/html/rfc1918
[30]https://tools.ietf.org/html/rfc2131
[31]https://tools.ietf.org/html/rfc9110
[32]https://tools.ietf.org/html/rfc2821
[33]https://tools.ietf.org/html/rfc3330

RFC 3493[34] —Extensiones básicas de interfaz de socket para IPv6

RFC 3542[35] —Interfaz de programación de aplicaciones (API) de sockets avanzados para IPv6

RFC 3849[36] —Prefijo de dirección IPv6 reservado para documentación **RFC 3920**[37] —Protocolo extensible de mensajería y presencia (XMPP) **RFC 3977**[38] —Protocolo de transferencia de noticias de red (NNTP)

RFC 4193[39] —Direcciones de unidifusión IPv6 locales únicas

RFC 4506[40] —Estándar de representación de datos externos (XDR)

El IETF tiene una buena herramienta en línea para buscar y navegar por RFC[41].

[34] https://tools.ietf.org/html/rfc3493
[35] https://tools.ietf.org/html/rfc3542
[36] https://tools.ietf.org/html/rfc3849
[37] https://tools.ietf.org/html/rfc3920
[38] https://tools.ietf.org/html/rfc3977
[39] https://tools.ietf.org/html/rfc4193
[40] https://tools.ietf.org/html/rfc4506
[41] https://tools.ietf.org/rfc/

Índice

10.x.x.x, 16
192.168.x.x, 16
255.255.255.255, 72, 99

accept() función, 26, *82*
Dirección ya en uso, 24, 76
AF_INET macro , 13 , 23 , 79
AF_INET6 macro, 13

Bapper, 74 años
Función bind(), 23, 25, 76, *84*
    implícito,
25 bla bla bla, 8
Bloqueo, 28, 42
Difusión, 72
BSD, 2
Pedido de bytes, 11, 14, 58, 98

Cliente
    datagrama, 39–40
    Corriente, 35–37
Cliente/Servidor, 32–41
close() función, 29, *87*
Función closesocket(), 3, 30, *87*
Compiladores
    CCG, 1
Compresión, 78
conectar(), *23*
    en sockets de datagramas, 86
Función connect(), 6, 25, *85*
    en sockets de datagramas, 29, 40
Conexión denegada, 37
Función CreateProcess(), 3, 80
Función CreateThread(), 3
Clase CSocket , 3
Cygwin, 2

Encabezado de
    encapsulación
    de datos, 8
Encapsulación de datos, 7, 57
    pie de página, 8
Sockets de datagramas, 6–7
DHCP, 119
Burros, 57

EAGAIN macro, 42, 112
Enviar un correo electrónico a Beej, 4
Encriptación, 78
Macro EPIPE , 87
Errno variable, *96*, 103
Ethernet, 8
Macro EWOULDBLOCK , 42
Excalibur, 72 años

F_SETFL macro, *97*
fcntl(), 42, 83, *97*
FD_CLR() macro, 50, *108*
FD_ISSET() macro, 50, *109*
FD_SET() macro, 50, *108*
FD_ZERO() macro, 50, *109*
Descriptor de archivo, 6
Cortafuegos, 16, 74, 80
    agujeros, 80
fork(), 3, 32, 80
Función freeaddrinfo(), *87*
FTP, 119

gai_strerror(), *87*
Función getaddrinfo(), 13, 17, 19, 30, *87*
Función getHostByAddr(), 30, *91*
gethostbyname(), *91*, 91
gethostname() función, 30, *91*
getnameinfo(), 17, 30, *94*
getpeername() función, 30, *95*
getprotobyname() función, 114
getsockopt(), *110*
Función getTimeOfDay(), 52
Cabra, 76
Declaración de GoTo , 77

Archivos de cabecera, 76
herror(), *92*
Función hstrerror(), *92*
Función htonl(), 12, *97*
Función htons(), 12, 14, 58, *97*
Protocolo HTTP, 7, 119

ICMP, 76
IEEE-754, 59
illumos, 2
INADDR_BROADCAST macro, 72
inet_addr() función, 15, *99*
inet_aton() función, 15, *99*
Función inet_ntoa(), 15, *99*
inet_ntop() funcionalidad, 15, 30, *100*
Función inet_pton(), 15, *100*
Función ioctl(), 80
IP, 7, 8, 119
Dirección IP, 9, 15, 23, 29, 30
Comando de ruta IP ,

IPv4, 9
IPv6, 9, 14, 16, 17
IRC, 58, 119
ISO/OSI, 8

Modelo de red en capas, 8
Linux, 2
listen(), 23, 26, *102*
atrasos, 26
con select(), 52
localhost, 76
Dispositivo de bucle invertido, 76

páginas de manual, 82
Reflejando la Guía, 4
MSG_DONTROUTE macro, *112*
MSG_DONTWAIT macro, *112*
MSG_NOSIGNAL macro, *112*
MSG_OOB macro , *106* , *112*
MSG_PEEK macro, *106*
MSG_WAITALL macro, *106*
MTU, 79

NAT, 16
comando netstat , 76
NNTP, 120
Enchufes sin bloqueo, 42, 112
Función ntohl(), 12, *97*
Función ntohs(), 12, *97*

O_ASYNC macro , *97*
O_NONBLOCK macro, 56, 83, *97*, 109
OpenSSL, 78
Datos fuera de banda, 106, 112

Rastreador de paquetes, 80
Pat, 74 años
Función perror(), 96, *103*
PF_INET macro , 79 , *114*
comando ping , 76
poll(), 43–50
poll(), 42, 56, *104*
Puerto, 23, 24, 29
Red privada, 16
Modo promiscuo, 80

Zócalos en bruto, 6, 76
función read(), 6
Función recv(), 6, 28, *106*
tiempo muerto, 77
Función recvfrom(), 29, *106*
recvtimeout(), 78 Referencias
libros, 118
FRFC, 119–120
Basado en la web, 118–119
RFC, 119–120
comando de ruta , 76

SA_RESTART macro, 77
Seguridad, 79
Función select( ), 3, 50–56, 76, 77, *108*
con escuchar(), 52
send(), 6, 8, 28, *111*
sendall(), 56–57, 70
sendto(), 8, *111*
Serialización, 57–
70 Servidor
datagrama, 37–39
Arroyo, 32–35
Función setsockopt(), 24, 72, 76, 80, *110*
SHUT_RD macro, *113*
SHUT_RDWR macro, *113*
SHUT_WR macro, *113*
shutdown() función, 29, *113*
Función sigaction(), 35, 77
Señal SIGIO , 97
SIGPIPE macro, 87, 112
SIGURG macro, *106*, *112*
SMTP, 119
SO_BINDTODEVICE macro, *110*
SO_BROADCAST macro, 72, *110*
SO_RCVTIMEO macro, 80
SO_REUSEADDR macro , 24 , 76 , *110*
SO_SNDTIMEO macro, 80
SOCK_DGRAM macro, 6, 7, 28, 106, *114*
SOCK_RAW macro , 76 , *114*
SOCK_STREAM macro , 6 , 106 , *114*
Descriptor de socket, 6, 12
función socket(), 6, 22, *114*
SOL_SOCKET macro, *110*
Solaris, 1, 111
SSL, 78
Enchufes de flujo, 6
strerror() función, 96, *103*
struct addrinfo tipo, 12
struct tipo hostent , *92*
struct in6_addr tipo, *115*
struct in_addr tipo, *115* struct
pollfd tipo, 43, *104*
Tipo de calcetín estructural , 13, 29, 106, *115*
Tipo de sockaddr_in de
estructura , *115* Tipo de
sockaddr_in6 de estructura , *115*
Tipo de sockaddr_storage de
estructura , *115* Tipo de
estructura Timeval , 50–52, *108*
SunOS, 1, 111

TCP, 7, 119
Telnet, 6, 119
TFTP, 8, 119
Interrupción
configuración, 80
Traduciendo la guía, 4
TRON, 25

UDP, 7, 8, 72, 119

Vint Cerf, 9

Ventanas, 2, 30, 76, 87, 111
Subsistema de Windows para Linux, 2
Winsock, 2, 30
`write()`, 6
`Función WSACleanup()`, 3
`Función WSAStartup()`, 3
WSL, 2

XDR, 70, 120
XMPP, 120

Proceso zombie, 35