Beej’s Guide to Network Programming

Using Internet Sockets

Brian “Beej Jorgensen” Hall

Contents

Chapter 1

Introducción

¿La programación de sockets te tiene deprimido? ¿Te resulta demasiado difícil entender esto desde las páginas `man`? Quieres hacer una buena programación de Internet, pero no tienes tiempo para vadear a través de un montón de `struct`s tratando de averiguar si tienes que llamar a `bind()` antes de `connect()`, etc, etc.

Bueno, ¡adivina qué! Yo ya lo he hecho, ¡y me muero por compartir la información con todo el mundo! Has venido al lugar adecuado. Este documento debería dar al programador C competente, la ventaja que necesita para controlar este ruido de redes.

Y compruébalo: Por fin me he puesto al día con el futuro (¡justo a tiempo, además!) y he actualizado la Guía para IPv6. ¡Así que, aproveche!

1.1 Audiencia

Este documento ha sido escrito como un tutorial, no como una referencia completa. Es probablemente mejor cuando lo leen personas que están empezando con la programación de sockets y están buscando un punto de apoyo. Ciertamente no es la guía *completa y total* para la programación de sockets, ni mucho menos.

Con suerte, será suficiente para que esas páginas de manual empiecen a tener sentido… `:-)`

1.2 Plataforma y compilador

El código contenido en este documento fue compilado en un PC con Linux utilizando el compilador de Gnu `gcc`. Sin embargo, debería compilarse en casi cualquier plataforma que utilice `gcc`. Naturalmente, esto no se aplica si estás programando para Windows—ver la sección sobre programación para Windows, más abajo.

1.3 Página oficial y libros a la venta

La 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 me ayuda a mantener mi estilo de vida de escritor de documentos.

1.4 Nota para programadores de Solaris/SunOS/illumos

Al compilar para variantes de Solaris o SunOS, necesitaras especificar algunos modificadores extra en la línea de comandos para enlazar las librerías apropiadas. Para ello, simplemente añada "`-lnsl -lsocket -lresolv`" al final de la orden de compilación, de la siguiente manera:

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

Si sigues obteniendo errores, puedes intentar añadir `-lxnet` al final de la línea de comandos. No sé qué hace eso exactamente, pero algunas personas parecen necesitarlo.

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

```
int yes=1;
```

introduce esto:

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

Como no tengo caja Sun, no he comprobado ninguna de las informaciones anteriores; es sólo lo que la gente me ha dicho por correo electrónico.

1.5 Nota para programadores Windows

En este punto de la guía, históricamente, he hecho un poco de bolsa en Windows, simplemente debido al hecho de que no me gusta mucho. Pero entonces Windows y Microsoft (como empresa) mejoraron mucho. Windows 9 y 10 junto con WSL (abajo) eran realmente sistemas operativos decentes. No hay mucho de lo que quejarse.

Bueno, un poco; por ejemplo, estoy escribiendo esto (en 2025) en un portátil de 2015 que solía ejecutar Windows 10. Con el tiempo se puso demasiado lento y he instalado Linux en él. Y lo he estado usando desde entonces. Y ahora se sabe que Windows 11 requerirá un hardware más potente que Windows 10. Eso no me gusta. El sistema operativo debe ser tan discreto como sea posible y no requerir que gastes más dinero. La potencia extra de la CPU debería ser para las aplicaciones, no para el sistema operativo.

Así que te animo a que pruebes Linux[1], BSD[2], illumos[3] o algún otro sabor de Unix, en su lugar.

¿Cómo llegó ahí esa caja de jabón?

Pero a la gente le gusta lo que le gusta, y a ustedes, los de Windows, les gustará saber que esta información es generalmente aplicable a Windows, con algunos pequeños cambios. Oh, hey, ¡la tribuna está de vuelta!

Una cosa que deberías tener muy en cuenta es el Subsistema Windows para Linux[4]. Esto básicamente le permite instalar una cosa Linux VM-ish en Windows 10. Eso también te situará definitivamente, y podrás construir 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 por ahí que hacerlo permite compilar todos estos programas sin modificarlos, pero nunca lo he probado.

Algunos de ustedes quieren hacer las cosas a la manera de Windows. Eso es muy valiente de tu parte, y esto es lo que tienes que hacer: ¡corre y consigue Unix inmediatamente! No, no… estoy bromeando. Se supone que soy "Amigable" con Windows estos días…

De acuerdo. Me pondré a ello.

[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/

Esto es lo que tendrás que hacer: primero, ignora casi todos los ficheros de cabecera del sistema que menciono aquí. En su lugar, incluye:

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

`winsock2` es la «nueva» versión (circa 1994) de la librería de sockets de Windows.

Desafortunadamente, si incluyes `windows.h`, automáticamente te trae el antiguo fichero de cabecera `winsock.h` (versión 1) que entra en conflicto con `winsock2.h`. Qué divertido.

Así que si tienes que incluir `windows.h`, necesitas definir una macro para que *no* incluya la cabecera antigua:

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

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

Espera! También tienes que hacer una llamada a `WSAStartup()` antes de hacer cualquier otra cosa con la librería de sockets. A esta función le pasas la versión de Winsock que desees (por ejemplo, la versión 2.2). Y entonces puedes comprobar el resultado para asegurarte de que esa versión está disponible.

El código para hacerlo se parece a esto:

```
#include <winsock2.h>

{
    WSADATA wsaData;

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

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

Tenga en cuenta que la llamada a `WSACleanup()`. Eso es lo que quieres llamar cuando hayas terminado con la librería Winsock.

También tienes que decirle a tu compilador que enlace la librería Winsock, llamada `ws2_32.lib` para Winsock 2. En VC++, esto se puede hacer a través del menú `Project`, en `Settings...`. Haz click en la pestaña `Link`, y busca la casilla titulada «Object/library modules». Añade «ws2_32.lib» (o la librería que prefieras) a la lista.

O eso he oído.

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

También hay una clase socket que puedes usar,i[`CSocket` class]] `CSocket` Consulta las páginas de ayuda de tu compilador para más información.

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

Por último, he oído que Windows no tiene `fork()`que, por desgracia, se utiliza en algunos de mis ejemplos. Quizás tengas que enlazar una librería POSIX o algo 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, la función `CreateThread()` es un poco más fácil de digerir… desafortunadamente una discusión sobre multithreading está más allá del alcance de este documento. No puedo hablar de mucho, ¡ya sabes!

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

1.6 Política de correo electrónico

Generalmente estoy disponible para ayudar con preguntas por correo electrónico, así que siéntete libre de escribirme, pero no puedo garantizar una respuesta. Llevo una vida muy ajetreada y a veces no puedo responder a tus preguntas. Cuando es así, suelo borrar el mensaje. No es nada personal; simplemente, nunca tendré tiempo para dar la respuesta detallada que necesitas.

Por regla general, cuanto más compleja es la pregunta, menos probable es que responda. Si puedes reducir tu pregunta antes de enviarla y te aseguras de incluir toda la información pertinente (como la plataforma, el compilador, los mensajes de error que recibes y cualquier otra cosa que creas que puede ayudarme a solucionar el problema), es mucho más probable que recibas una respuesta. Para más información, lea el documento de ESR Cómo hacer preguntas de forma inteligente[7].

Si no recibes respuesta, dale más vueltas, intenta encontrar la respuesta y, si sigue sin aparecer, vuelve a escribirme con la información que hayas encontrado y, con suerte, será suficiente para que pueda ayudarte.

Ahora que ya te he dado la lata con lo de escribirme y no escribirme, me gustaría que supieras que agradezco *completamente* todos los elogios que ha recibido la guía a lo largo de los años. Es una auténtica inyección de moral, ¡y me alegra saber que se utiliza para el bien! Gracias.

1.7 Mirroring

Eres más que bienvenido a replicar este sitio, ya sea pública o privadamente. Si lo haces público 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 pondré un enlace a tu traducción desde la página principal. No dudes en añadir tu nombre e información de contacto a la traducción.

Este documento fuente markdown utiliza codificación UTF-8.

Por favor, ten en cuenta las restricciones de licencia en la sección Copyright, Distribución y Legal, más abajo.

Si quieres que aloje la traducción, sólo tienes que pedírmelo. También pondré un enlace a ella si quieres alojarla; cualquiera de las dos opciones está bien.

1.9 Copyright, distribución y legal

[6] https://www.tallyhawk.net/WinsockExamples/
[7] http://www.catb.org/~esr/faqs/smartquestions.html

o envíe una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, EE.UU.

Una excepción específica a la parte de la licencia relativa a la «Prohibición de obras derivadas» es la siguiente: esta guía puede traducirse libremente a cualquier idioma, siempre que la traducción sea exacta, y la guía se reimprima en su totalidad. Se aplicarán a la traducción las mismas restricciones de licencia que a la guía original. La traducción también puede incluir el nombre y la información de contacto del traductor.

El código fuente en C presentado en este documento es de dominio público y está totalmente libre de cualquier restricción de licencia.

Se anima libremente a los educadores a recomendar o suministrar copias de esta guía a sus alumnos.

A menos que las partes acuerden lo contrario por escrito, el autor ofrece la obra tal como está y no hace representaciones o garantías de ningún tipo con respecto a la obra, expresas, implícitas, estatutarias o de otro tipo, incluyendo, sin limitación, garantías de título, comerciabilidad, idoneidad para un propósito particular, no infracción, o la ausencia de defectos latentes o de otro tipo, exactitud, o la presencia o ausencia de errores, sean o no descubribles.

Salvo en la medida en que lo exija la legislación aplicable, en ningún caso el autor será responsable ante usted, bajo ninguna teoría legal, de ningún daño especial, incidental, consecuente, punitivo o ejemplar derivado del uso de la obra, incluso si el autor ha sido advertido de la posibilidad de tales daños. Contact `beej@beej.us` for more information.

1.10 Dedicatoria

Gracias a todos los que me han ayudado en el pasado y en el futuro a escribir esta guía. Y gracias a toda la gente que produce 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 miles de personas que me han escrito con sugerencias de mejora y palabras de ánimo.

Dedico esta guía a algunos de mis mayores héroes e inspiradores en el mundo de la informática: Donald Knuth, Bruce Schneier, W. Richard Stevens y The Woz, a mis lectores y a toda la comunidad del 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 están hechos con Inkscape. El Markdown se convierte a HTML y LaTex/PDF con Python, Pandoc y XeLaTeX, utilizando fuentes Liberation. La cadena de herramientas está compuesta al 100% por software libre y de código abierto.

Chapter 2

¿Qué es un socket?

Oyes hablar de «sockets» todo el tiempo, y quizá te preguntes qué son exactamente. Bueno, son esto: una forma de hablar con otros programas usando descriptores de fichero estándar de Unix .

¿Qué?

Vale… puede que hayas oído a algún hacker de Unix decir: «¡Jesús, *todo* en Unix es un archivo!». A lo que esa persona puede haberse referido, es al hecho de que cuando los programas Unix hacen cualquier tipo de E/S, lo hacen leyendo o escribiendo en un descriptor de fichero. Un descriptor de fichero es simplemente un número entero asociado a un fichero abierto. Pero (y aquí está el truco), ese fichero puede ser una conexión de red, un FIFO, una tubería, un terminal, un fichero real en el disco, o cualquier otra cosa. ¡Todo en Unix *es* un fichero! Así que cuando quieras comunicarte con otro programa a través de Internet vas a hacerlo a través de un descriptor de fichero, mejor que lo creas.

"¿De dónde saco este descriptor de fichero para la comunicación en red, Sr. Sabelotodo?" es probablemente la última pregunta que ronda por tu cabeza ahora mismo, pero voy a responderla de todos modos: Haces una llamada a la función del sistema `socket()`. Devuelve el descriptor de socket , y te comunicas a través de él usando las funciones especializadas `send()` y `recv()` (`man send`, `man recv`).

"Pero, ¡eh!", puedes estar exclamando en este momento. "Si es un descriptor de fichero, ¿por qué en el nombre de Neptuno no puedo usar la función normal `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 tal esto: hay todo tipo de sockets. Existen Direcciones DARPA de Internet (Internet Sockets), nombres de ruta en un nodo local (Unix Sockets), direcciones CCITT X.25 (X.25 Sockets que puedes ignorar sin problemas), y probablemente muchos otros dependiendo del tipo de Unix que utilices. Este documento sólo trata del primero: Internet Sockets.

2.1 Dos Tipos de Sockets de Internet

¿Qué es esto? ¿Hay dos tipos de sockets de Internet? Sí. Bueno, no. Estoy mintiendo. Hay más, pero no quería asustarte. Aquí sólo voy a hablar de dos tipos. Excepto en esta frase, donde te voy a decir que `Raw Sockets` también son muy potentes y deberías buscarlos.

Muy bien, ya. ¿Cuáles son los dos tipos? Uno es "Stream Sockets"; el otro es "Datagram Sockets", que a partir de ahora se llamarán `SOCK_STREAM` y `SOCK_DGRAM`, respectivamente. Los sockets de datagramas se denominan a veces "sockets sin conexión". (Aunque pueden ser `connect()` si realmente quieres. Ver `connect()`, más abajo).

Los stream sockets son flujos de comunicación bidireccionales conectados de forma fiable. Si introduce dos elementos en la toma en el orden «1, 2», llegarán en el orden «1, 2» al extremo opuesto. También estarán libres de errores. De hecho, estoy tan seguro de que no habrá errores que me pondré los dedos en las orejas y cantaré *la la la la* si alguien intenta decir lo contrario.

¿Para qué sirven los stream sockets? Bueno, puede que hayas oído hablar de `telnet` o `ssh`, ¿verdad? Usan stream sockets. Todos los caracteres que escribes tienen que 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 te conectas por telnet a un sitio web en el puerto 80, escribes `GET / HTTP/1.0` y pulsas RETURN dos veces, ¡te devolverá el HTML!

> Si no tienes `telnet` instalado y no quieres instalarlo, o tu `telnet` está siendo quisquilloso 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. (Nótese que telnet es en realidad un protocolo de red especificado[b], y `telnot` no implementa este protocolo en absoluto).
>
> [a] https://beej.us/guide/bgnet/source/examples/telnot.c
> [b] https://tools.ietf.org/html/rfc854

¿Cómo consiguen los stream sockets este alto nivel de calidad en la transmisión de datos? Utilizan un protocolo llamado "Protocolo de Control de Transmisión", también conocido como `TCP` (vea RFC 793[1] para información extremadamente detallada sobre TCP). TCP se encarga de que tus datos lleguen secuencialmente y sin errores. Es posible que haya oído antes TCP como la mejor mitad de TCP/IP donde `IP` significa "Protocolo de Internet" (véase RFC 791[2]). IP se ocupa principalmente del enrutamiento en Internet y, por lo general, no es responsable de la integridad de los datos.

Genial. ¿Qué pasa con los sockets Datagram? ¿Por qué se llaman sin conexión? ¿De qué se trata? ¿Por qué no son fiables? Bueno, aquí hay algunos hechos: si envías un datagrama, puede llegar. Puede llegar desordenado. Si llega, los datos dentro del paquete no tendrán errores.

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

¿Por qué son sin conexión? Bueno, básicamente porque no tienes que mantener una conexión abierta como haces con los stream sockets. Simplemente construyes un paquete, le pegas una cabecera IP con información de destino y lo envías. No se necesita conexión. Generalmente se usan cuando una pila TCP no está disponible o cuando unos pocos paquetes perdidos aquí y allá no significan el fin del Universo. Ejemplos de aplicaciones: `tftp` (protocolo trivial de transferencia de archivos, un hermano pequeño del FTP), `dhcpcd` (un cliente DHCP), juegos multijugador, streaming de audio, videoconferencia, etc.

"¡Un momento! ¡`tftp` y `dhcpcd` se utilizan para transferir aplicaciones binarias de un host a otro! ¡Los datos no pueden perderse si esperas que la aplicación funcione cuando llegue! ¿Qué clase de magia oscura es esta?"

Bueno, amigo humano, `tftp` y programas similares tienen su propio protocolo sobre UDP. Por ejemplo, el protocolo tftp dice que por cada paquete que se envía, el destinatario tiene que enviar de vuelta 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 acuse de recibo es muy importante cuando se implementan aplicaciones `SOCK_DGRAM` fiables.

Para aplicaciones no fiables como juegos, audio o vídeo, simplemente ignora los paquetes perdidos, o quizás intenta compensarlos inteligentemente. (Los jugadores de Quake conocerán la manifestación de este efecto por el término técnico: *accursed lag (retraso maldito)*. La palabra "maldito", en este caso, representa cualquier expresión extremadamente profana).

¿Por qué utilizar un protocolo subyacente poco fiable? Por dos razones: velocidad y rapidez. Es mucho más rápido disparar y olvidar, que hacer un seguimiento de lo que ha llegado sin problemas y asegurarse de que está en orden y todo eso. Si estás enviando mensajes de chat, TCP es genial; si estás enviando 40 actualizaciones de posición por segundo de los jugadores del mundo, quizá no importe tanto si una o dos se caen, y UDP es una buena opción.

[1] https://tools.ietf.org/html/rfc793
[2] https://tools.ietf.org/html/rfc791
[3] https://tools.ietf.org/html/rfc768

2.2 Tonterías de bajo nivel y teoría de redes

Ya que acabo de mencionar la estratificación de protocolos, es hora de hablar de cómo funcionan realmente las redes, y de mostrar algunos ejemplos de cómo se construyen los paquetes `SOCK_DGRAM`. En la práctica, probablemente puedas saltarte esta sección. Sin embargo, es una buena base.

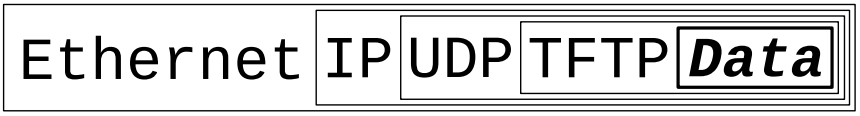

Figure 2.1: Data Encapsulation.

Hola chicos, es hora de aprender sobre *¡Capsulación de datos*! Esto es muy muy importante. Es tan importante que puede que lo aprendas si tomas el curso de redes aquí en Chico State `;-)`. Básicamente, dice esto: nace un paquete, el paquete es envuelto (encapsulado) en una cabecera (y raramente un pie) por el primer protocolo (digamos, el protocolo TFTP protocol), luego todo (cabecera TFTP incluida) es encapsulado de nuevo por el siguiente protocolo (digamos, UDP), por el siguiente (IP), y por el protocolo final en la capa (física) de hardware (digamos, Ethernet).

Cuando otro ordenador recibe el paquete, el hardware despoja la cabecera Ethernet, el núcleo, despoja las cabeceras IP y UDP, el programa TFTP despoja la cabecera TFTP, y finalmente tiene los datos.

Ahora por fin puedo hablar del infame *Modelo de red por 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, puedes escribir programas de sockets que sean exactamente iguales sin preocuparte de cómo se transmiten físicamente los datos (serie, Ethernet delgada, AUI, lo que sea) porque los programas de niveles inferiores se ocupan de ello por ti. El hardware de red real y la topología son transparentes para el programador de sockets.

Sin más preámbulos, presentaré las capas del modelo completo. Recuerda esto para los exámenes de la clase de redes:

- 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 puedas imaginar: es el lugar donde los usuarios interactúan con la red.

Ahora bien, este modelo es tan general que probablemente podrías utilizarlo como guía de reparación de automóviles si realmente quisieras. Un modelo de capas más consistente 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 puedas ver cómo estas capas corresponden a la encapsulación de los datos originales.

¿Ves cuánto trabajo hay en construir un simple paquete? ¡Caramba! ¡Y tienes que escribir tú mismo las cabeceras del paquete usando `cat`! Es broma. Todo lo que tienes que hacer para los sockets de flujo es `send()` (eviar) los datos. Todo lo que tienes que hacer para los sockets de datagramas es encapsular el paquete en el método de tu elección y `sendto()`. El núcleo, construye la Capa de Transporte y la Capa de Internet por ti 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. Ah, sí, se me ha olvidado decirles todo lo que quería decirles sobre el encaminamiento: ¡nada! Así es, no voy a hablar de ello en absoluto. El router

desnuda el paquete hasta la cabecera IP, consulta su tabla de enrutamiento, bla bla bla_. Echa un vistazo al IP RFC[4] si realmente te importa. Si nunca te enteras, bueno, vivirás.

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

Chapter 3

Direcciones IP, `structs` y Data Munging

Esta es la parte del juego en la que (para variar) vamos a hablar de código.

Pero primero, ¡discutiremos más sobre no código!. Quiero hablar sobre direcciones IP y puertos para que lo tengamos claro. Luego hablaremos de 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 aún se llamaba Obi Wan Kenobi, existía un maravilloso sistema de enrutamiento de red llamado Protocolo de Internet versión 4, también llamado IPv4 . Tenía direcciones formadas por cuatro bytes (también conocidos como cuatro "octetos"), y se escribía habitualmente en forma de "puntos y números", como por ejemplo `192.0.2.111`.

Seguro que lo has visto por ahí.

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

Todo el mundo, incluido Obi Wan, estaba contento. Todo iba genial, hasta que un pesimista llamado Vint Cerf advirtió a todo el mundo de que estábamos a punto de quedarnos sin direcciones IPv4.

(Además de advertir a todo el mundo de la llegada del Apocalipsis IPv4, Vint Cerf[1]también es conocido por ser el padre de Internet. Así que no estoy en posición de cuestionar su juicio).

¿Se han quedado sin direcciones? ¿Cómo es posible? Hay miles de millones de direcciones IP en una dirección IPv4 de 32 bits. ¿Realmente tenemos miles de millones de ordenadores ahí fuera?

Sí.

Además, al principio, cuando sólo había unos pocos ordenadores y todo el mundo pensaba 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 una pequeña empresa llamada Apple, por nombrar algunas).

De hecho, si no fuera por varias medidas provisionales, nos habríamos quedado sin ellas hace mucho tiempo.

Pero ahora vivimos en una era en la que hablamos de que cada ser humano tiene una dirección IP, cada ordenador, cada calculadora, cada teléfono, cada parquímetro y (por qué no) también cada perrito.

Y así, IPv6 nació. Como Vint Cerf es probablemente inmortal (aunque su forma física falleciera, Dios no lo quiera, probablemente ya exista como una especie de programa hiperinteligente ELIZA[2] en las

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

profundidades de Internet2), nadie quiere tener que oírle decir otra vez "te lo dije" si no tenemos suficientes direcciones en la próxima versión del Protocolo de Internet.

¿Qué le sugiere esto?

Que necesitamos *muchas* más direcciones. Que necesitamos no sólo el doble de direcciones, no mil millones de veces más, no mil billones de veces más, sino ¡79 MILLONES DE MILLONES DE TRILLONES de veces más direcciones posibles!_ ¡Eso les enseñará!

Dices: "B.J., ¿es cierto? Tengo motivos para no creer en los números grandes". Bueno, la diferencia entre 32 bits y 128 bits puede no parecer mucha; son sólo 96 bits más, ¿verdad? Pero recuerde que estamos hablando de potencias: 32 bits representan unos 4.000 millones de números (2^{32}), mientras que 128 bits representan unos 340 billones de billones de billones de números (de verdad, 2^{128}). Eso es como un millón de Internets IPv4 para *cada estrella del Universo*.

Olvídate también del aspecto de puntos y números de IPv4; ahora tenemos una representación hexadecimal, con cada trozo de dos bytes separado por dos puntos, así:

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

Pero eso no es todo. Muchas veces, tendrás una dirección IP con muchos ceros, y puedes comprimirlos entre dos dos puntos. Y puedes dejar 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 *loopback*. Siempre significa "esta máquina, en la que estoy funcionando ahora". En IPv4, la dirección loopback es `127.0.0.1`.

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

Estamos hablando de diversión en serio.

De hecho, es tan divertido que los creadores de IPv6 han recortado con bastante displicencia billones y billones de direcciones para uso reservado, pero tenemos tantas que, francamente, ¿quién lleva ya la cuenta? Sobran para todos los hombres, mujeres, niños, cachorros y parquímetros de todos los planetas de la galaxia. Y créeme, todos los planetas de la galaxia tienen parquímetros. Sabes que es verdad.

3.1.1 Subredes

Por razones organizacionales, a veces es conveniente declarar que "esta primera parte de esta dirección IP, hasta este bit es la porción de red de la dirección IP, y el resto es la porción de host".

Por ejemplo, con IPv4, podrías 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 (observa cómo ponemos en cero el byte que era el host).

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

Como puedes ver, había solo unas pocas redes Clase A, una gran cantidad de redes Clase C, y algunas redes Clase B en el medio.

La porción de red de la dirección IP se describe con algo llamado "máscara de red", con la cual haces una operación bit a bit "AND" con la dirección IP para obtener el número de red. La máscara de red usualmente se ve algo como 255.255.255.0. (Por ejemplo, con esa máscara de red, si tu IP es 192.0.2.12, entonces tu red es 192.0.2.12 Y 255.255.255.0, lo que da 192.0.2.0).

Desafortunadamente, resultó que esto no era lo suficientemente detallado para las necesidades futuras de Internet; nos estábamos quedando sin redes Clase C bastante rápido, y definitivamente nos habíamos quedado sin redes Clase A, así que ni siquiera te molestes en preguntar. Para solucionar esto, los poderes que sean permitieron que la máscara de red fuera un número arbitrario de bits, no solo 8, 16 o 24. Así que podrías tener una máscara de red de, por ejemplo, 255.255.255.252, que son 30 bits de red, y 2 bits de host, lo que permite cuatro hosts en la red. (Ten en cuenta que la máscara de red es SIEMPRE un montón de bits en 1 seguidos por un montón de bits en 0).

Pero es un poco incómodo usar una larga cadena de números como 255.192.0.0 como una máscara de red. En primer lugar, las personas no tienen una idea intuitiva de cuántos bits son, y en segundo lugar, no es realmente compacto. Así que apareció el "nuevo estilo", y es mucho más agradable. Simplemente colocas una barra diagonal después de la dirección IP, y luego sigues eso con el número de bits de red en decimal. Como esto: 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 eres tan amable de recordar, te presenté antes el modelo de red por capas(Layered Network Model) que tenía la Capa de Internet (IP) dividida de la Capa de Transporte de Host a Host (TCP y UDP). Ponte al día antes del siguiente párrafo.

Resulta que además de una dirección IP (usada por la capa IP), hay otra dirección que es usada por TCP (stream sockets) y, coincidentemente, por UDP (datagram sockets). Se trata del *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 de la calle de un hotel, y en el número de puerto como el número de la habitación. Es una analogía decente; quizá más adelante se me ocurra una que tenga que ver con la industria del automóvil. Digamos que quieres tener un ordenador que gestione el correo entrante y los servicios web… ¿cómo diferenciar ambos en un ordenador con una única dirección IP

Bueno, los diferentes servicios de Internet tienen diferentes números de puerto bien conocidos. Puedes verlos todos en la gran lista de puertos de IANA[3] o, si estás en una máquina Unix, en tu archivo `/etc/services`. HTTP (la web) es el puerto 80, telnet es el puerto 23, SMTP es el puerto 25, el juego DOOM[4] usaba el puerto 666, etc. y así sucesivamente. Los puertos por debajo de 1024 suelen considerarse especiales, y normalmente requieren privilegios especiales del sistema operativo para su uso.

Y eso es todo.

3.2 Ordenación de bytes

¡Por orden del Reino! Habrá dos ordenaciones de bytes, a partir de ahora conocidas como ¡Lame y Magnificent!

Es broma, pero una es mejor que la otra. `:-)`

No hay forma fácil de decirlo, así que lo diré sin rodeos: puede que tu ordenador haya estado almacenando bytes en orden inverso a tus espaldas. ¡Ya lo sé! Nadie quería tener que decírtelo.

El caso es que todo el mundo en Internet está de acuerdo en que si quieres representar un número hexadecimal de dos bytes, digamos `b34f`, lo almacenarás en dos bytes secuenciales `b3` seguidos de `4f`. Tiene

[3] https://www.iana.org/assignments/port-numbers
[4] https://en.wikipedia.org/wiki/Doom_(1993_video_game)

sentido y, como te diría Wilford Brimley[5], es lo correcto. Este número, almacenado con el extremo grande primero, se llama *Big-Endian*.

Desgraciadamente, unos pocos ordenadores repartidos por el mundo, es decir, los que tienen un procesador Intel o compatible con Intel, almacenan los bytes al revés, de modo que `b34f` se almacenaría en memoria como los bytes secuenciales `4f` seguidos de `b3`. Este método de almacenamiento se llama *Little-Endian*.

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

Tu ordenador almacena los números en *Host Byte Order*. Si es un Intel 80x86, el orden de los bytes es Little-Endian. Si es un Motorola 68k, el Host Byte Order es Big-Endian. Si es un PowerPC, el orden de bytes del host es… bueno, ¡depende!

Muchas veces cuando estás construyendo paquetes o rellenando estructuras de datos necesitarás asegurarte de que tus números de dos y cuatro bytes están en Network Byte Order. Pero, ¿cómo puedes hacerlo si no conoces el orden de bytes nativo del host?

Buenas noticias. Sólo tienes que asumir que el Orden de Bytes del Host no es correcto, y siempre pasas el valor a través de una función para ajustarlo al Orden de Bytes de la Red. La función hará la conversión mágica si es necesario, y de esta manera tu código es portable a máquinas de diferente endianness.

Muy bien. Hay dos tipos de números que puedes convertir: `short` (dos bytes) y `long` (cuatro bytes). Estas funciones también funcionan para sus variaciones `unsigned`. Digamos que quieres convertir un `short` de Host Byte Order a Network Byte Order. Empieza con «h» para «host», sigue con «to», luego «n» para «network», y «s» para «short»: h-to-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()` («Short to long host») –no en esta fiesta, al menos. Pero la hay:

Función	Descripción
`htons()`	host `to` network `s`hort
`htonl()`	host `to` network `l`ong
`ntohs()`	network `to` host `s`hort
`ntohl()`	network `to` host `l`ong

Básicamente, querrás convertir los números al orden de bytes de red antes de que salgan por el cable, y convertirlos al orden de bytes de host cuando entren por el cable.

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

Asuma que los números en este documento están en el orden de bytes del host a menos que yo diga lo contrario.

3.3 `structs`

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

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

```
int
```

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

Sólo un `int` normal.

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

Mi primera Struct™—`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. Esto tendrá sentido más adelante, cuando lleguemos al uso real, pero por ahora sepa que es una de las primeras cosas que llamará cuando haga una conexión.

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // utilice 0 para "cualquiera"
    size_t           ai_addrlen;   // tamaño de ai_addr en bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in o _in6
    char            *ai_canonname; // nombre de host canónico completo

    struct addrinfo *ai_next;      // lista enlazada, nodo siguiente
};
```

Cargarás un poco esta estructura y luego llamarás a la función `getaddrinfo()`. Devolverá un puntero a una nueva lista enlazada de estas estructuras, rellenada con todo lo que necesites.

Puedes forzar el uso de IPv4 o IPv6 en el campo `ai_family`, o dejarlo como `AF_UNSPEC` para usar lo que quieras. Esto es genial porque tu código puede ser agnóstico a la versión IP

Ten en cuenta que se trata de una lista enlazada: `ai_next` apunta al siguiente elemento—podría haber varios resultados entre los que elegir. Yo usaría el primer resultado que funcionara, pero puede que tengas necesidades de negocio diferentes; ¡no lo sé todo!

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

Normalmente no necesitarás escribir en estas estructuras; a menudo, una llamada a `getaddrinfo()` para rellenar tu `struct addrinfo` es todo lo que necesitarás. Sin embargo, tendrás que mirar dentro de estas `struct`s para obtener los valores, así que te los presento aquí.

(Además, todo el código escrito antes de que se inventara la `struct addrinfo` empaquetaba todo esto a mano, así que verás mucho código IPv4 que hace exactamente eso). Ya sabes, en versiones antiguas de esta guía y demás).

Algunas `struct`s son IPv4, algunas son IPv6, y algunas son ambas. Tomaré nota de cuales son cuales.

De todos modos, la `struct sockaddr` contiene información de direcciones de sockets para muchos tipos de sockets.

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

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

Para tratar con `struct sockaddr`, los programadores crearon una estructura paralela: `struct sockaddr_in` («in» por «Internet») para ser usada con IPv4.

Y *esto es lo importante*: un puntero a `struct sockaddr_in` puede convertirse en un puntero a `struct sockaddr` y viceversa. Así que aunque `connect()` quiera una `struct sockaddr*`, puedes usar una `struct sockaddr_in` y convertirla en el último momento.

```
// (Sólo IPv4 - véase struct sockaddr_in6 para IPv6)

struct sockaddr_in {
    short int          sin_family;  // Familia de direcciónes, AF_INET
    unsigned short int sin_port;    // Número de puerto.
    struct in_addr     sin_addr;    // Dirección de Internet
    unsigned char      sin_zero[8]; // Mismo tamaño que struct sockaddr
};
```

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 una `struct sockaddr`) debe establecer cada byte completamnete a cero con la función `memset()`. Además, observe que `sin_family` corresponde a `sa_family` en una `struct sockaddr` y debe establecerse a `AF_INET`. Por último, `sin_port` debe estar en *Network Byte Order* (usando `htons()`)

Profundicemos un poco más. Ves que el campo `sin_addr` es una `struct in_addr`. ¿Qué es eso? Bueno, no quiero ser demasiado dramático, pero es una de las uniones más aterradoras de todos los tiempos:

```
// (sólo IPv4; véase struct in6_addr 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)
};
```

¡Vaya! Bueno, solía ser un sindicato, pero ahora parece que esos días se han ido. Hasta nunca. Así que si has declarado que `ina` es del 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 utiliza la horrible unión para `struct in_addr`, todavía puede hacer referencia a la dirección IP de 4 bytes exactamente de la misma manera que lo hice anteriormente (esto debido a `#define`).

¿Qué pasa con IPv6? También existen `struct`s similares para ello:

```
// (sólo IPv6; véase struct sockaddr_in y struct in_addr para IPv4)

struct sockaddr_in6 {
    u_int16_t       sin6_family;   // Familia de direcciónes, AF_INET6
    u_int16_t       sin6_port;     // número de puerto, orden de bytes de red
    u_int32_t       sin6_flowinfo; // Información de flujo IPv6
    struct in6_addr sin6_addr;     // Dirección IPv6
    u_int32_t       sin6_scope_id; // Alcance ID
};

struct in6_addr {
    unsigned char   s6_addr[16];   // Dirección IPv6
};
```

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

También ten en cuenta que no voy a hablar de la información de flujo IPv6 o de los campos Scope ID por el momento… esto es sólo una guía para principiantes. `:-)`

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 se va a rellenar tu `struct sockaddr` con una dirección IPv4 o IPv6. Así que pasas esta estructura paralela, muy similar a `struct sockaddr` excepto que es más grande, y luego la transformas al tipo que necesitas:

```
struct sockaddr_storage {
    sa_family_t  ss_family;     // familia de direcciones

    // todo esto es relleno, específico de la implementación, ignóralo:
    char      __ss_pad1[_SS_PAD1SIZE];
    int64_t   __ss_align;
    char      __ss_pad2[_SS_PAD2SIZE];
};
```

Lo importante es que puedas ver la familia de direcciones en el campo `ss_family`; comprueba si es `AF_INET` o `AF_INET6` (para IPv4 o IPv6). Entonces puedes convertirlo en una `struct sockaddr_in` o `struct sockaddr_in6` si quieres.

3.4 Direcciones IP, segunda parte

Afortunadamente para ti, hay un montón de funciones que te permiten manipular direcciones IP. No necesitas calcularlas a mano y meterlas en un `long` con el operador `<<`.

Primero, digamos que tienes una `struct sockaddr_in ina`, y tienes una dirección IP `10.12.110.57` o `2001:db8:63b3:1::3490` que quieres almacenar en ella. La función que quieres usar, `inet_pton()`, convierte una dirección IP en notación de números y puntos en una `struct in_addr` o una `struct in6_addr` dependiendo de si especifica `AF_INET` o `AF_INET6`. (`pton` significa "Presentation to Network"–puedes llamarlo "Print to Network" si te resulta más fácil de recordar). La conversión puede hacerse de la siguiente manera:

```
struct sockaddr_in sa; // IPv4
struct sockaddr_in6 sa6; // IPv6

inet_pton(AF_INET, "10.12.110.57", &(sa.sin_addr)); // IPv4
inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr)); // IPv6
```

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

Ahora bien, el fragmento de código anterior no es muy robusto porque no hay comprobación de errores. Verás, `inet_pton()` devuelve `-1` en caso de error, o 0 si la dirección es incorrecta. Así que comprueba que el resultado es mayor que 0 antes de usarlo.

Muy bien, ahora puedes convertir direcciones IP de cadena a sus representaciones binarias. ¿Y al revés? ¿Qué pasa si tienes una `struct in_addr` y quieres imprimirla en notación de números y puntos? (O una `struct in6_addr` que quieres en, uh, notación «hex-and-colons».) En este caso, querrás usar la función `inet_ntop()` (`ntop` significa "Network to presentation"–puedes llamarlo "Network to print" si es más fácil de recordar), así:

```
// IPv4:

char ip4[INET_ADDRSTRLEN];  // espacio para la cadena IPv4
struct sockaddr_in sa;      // pretender que esto se carga con algo

inet_ntop(AF_INET, &(sa.sin_addr), ip4, INET_ADDRSTRLEN);

printf("The IPv4 address is: %s\n", ip4);


// IPv6:

char ip6[INET6_ADDRSTRLEN]; // espacio para la cadena IPv6
```

```
struct sockaddr_in6 sa6;    // pretender que esto se carga con algo

inet_ntop(AF_INET6, &(sa6.sin6_addr), ip6, INET6_ADDRSTRLEN);

printf("The address is: %s\n", ip6);
```

Cuando la llame, le pasará el tipo de dirección (IPv4 o IPv6), la dirección, un puntero a una cadena para guardar el resultado y la longitud máxima de esa cadena. (Dos macros contienen convenientemente el tamaño de la cadena que necesitarás 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 para hacer esta conversión se llamaba `inet_ntoa()`. También está obsoleta y no funcionará con IPv6).

Por último, estas funciones sólo trabajan con direcciones IP numéricas—no harán ninguna búsqueda DNS en un nombre de host, como `www.example.com`. Para ello, como verás más adelante, deberás usar `getaddrinfo()`.

3.4.1 Redes privadas (o desconectadas)

Muchos lugares tienen un cortafuegos que oculta la red del resto del mundo para su propia protección. Y muchas veces, el cortafuegos traduce las direcciones IP «internas» a direcciones IP «externas» (que todo el mundo conoce) mediante un proceso llamado *Network Address Translation*, o NAT.

¿Ya te estás poniendo nervioso? "¿A dónde quiere llegar 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 se hace por ti de forma transparente. Pero quería hablar de la red detrás del cortafuegos por si empezabas a confundirte con los números de red que veías.

Por ejemplo, tengo un cortafuegos en casa. Tengo dos direcciones IPv4 estáticas asignadas por la compañía de DSL, y aún así tengo siete ordenadores en la red. ¿Cómo es posible? Dos ordenadores no pueden compartir la misma dirección IP, de lo contrario los datos no sabrían a cuál dirigirse.

La respuesta es: no comparten las mismas direcciones IP. Están en una red privada con 24 millones de direcciones IP asignadas. Son todas para mí. Bueno, todas para mí en lo que a los demás se refiere. Esto es lo que ocurre:

Si me conecto a un ordenador remoto, me dice que estoy conectado desde 192.0.2.33, que es la dirección IP pública que me ha proporcionado mi ISP. Pero si le pregunto a mi ordenador local cuál es su dirección IP, me dice 10.0.0.5. ¿Quién está traduciendo la dirección IP de una a otra? ¡Eso es, el cortafuegos! ¡Está haciendo NAT!

`10.x.x.x` es una de las pocas redes reservadas que sólo deben utilizarse en redes totalmente desconectadas o en redes que estén detrás de cortafuegos. Los detalles de qué números de red privada están disponibles para su uso, se describen en RFC 1918[6], pero algunos de los más comunes que verá son `10.x.x.x` y 192.168.x.x», donde "x" es de 0 a 255, generalmente. Menos común es `172.y.x.x`, donde `y` está entre 16 y 31.

Las redes detrás de un cortafuegos NAT no *necesitan* estar en una de estas redes reservadas, pero suelen estarlo.

(¡Un 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» falsas que se utilizan en la documentación, ¡como en esta guía! Wowzers!)

En cierto sentido, IPv6 también tiene redes privadas. Empezarán con `fdXX:` (o quizás en el futuro `fcXX:`), según RFC 4193[7]. NAT e IPv6 generalmente no se mezclan, sin embargo (a menos que estés haciendo lo de la pasarela IPv6 a IPv4, que está más allá del alcance de este documento)—en teoría tendrás tantas direcciones a tu disposición que no necesitarás usar NAT nunca más. Pero si quieres asignar direcciones para ti mismo en una red que no enrutará fuera, así es cómo hacerlo.

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

Chapter 4

Salto de IPv4 a IPv6

Pero sólo quiero saber qué cambiar en mi código para que funcione con IPv6. ¡Dímelo ahora!

¡Vale! ¡Ok!

Casi todo lo que hay aquí es algo que ya 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, intenta usar `getaddrinfo()` para obtener toda la información de `struct sockaddr`, en lugar de empaquetar las estructuras a mano. Esto mantendrá tu IP agnóstica a la versión, y eliminará muchos de los pasos posteriores.
2. En cualquier lugar donde encuentres que estás codificando algo relacionado con la versión IP, intenta envolverlo en una función de ayuda.
3. Cambia `AF_INET` por `AF_INET6`.
4. Cambia `PF_INET` por `PF_INET6`.
5. Cambie las asignaciones `INADDR_ANY` por `in6addr_any`, que son ligeramente diferentes:

   ```
   struct sockaddr_in sa;
   struct sockaddr_in6 sa6;

   sa.sin_addr.s_addr = INADDR_ANY;  // utilizar mi dirección IPv4
   sa6.sin6_addr = in6addr_any; // utilizar mi dirección IPv6
   ```

   Además, el valor `IN6ADDR_ANY_INIT` se puede utilizar como inicializador cuando se declara la `struct in6_addr`, de esta forma:

   ```
   struct in6_addr ia6 = IN6ADDR_ANY_INIT;
   ```

6. En lugar de `struct sockaddr_in` utilice `struct sockaddr_in6`, asegurándose de añadir «6» a los campos según corresponda (véase `struct`s, más arriba). No existe el campo `sin6_zero`.
7. En lugar de `struct in_addr` utilice `struct in6_addr`, asegurándose de añadir «6» a los campos según corresponda (véase `struct`s, más arriba).
8. En lugar de `inet_aton()` o `inet_addr()`, utilice `inet_pton()`.
9. En lugar de `inet_ntoa()`, use `inet_ntop()`.
10. En lugar de `gethostbyname()`, utilice la superior `getaddrinfo()`.
11. En lugar de `gethostbyaddr()`, utilice la función superior `getnameinfo()` (aunque `gethostbyaddr()` todavía puede funcionar con IPv6).
12. La función `INADDR_BROADCAST` ya no funciona. Usa multicast IPv6 en su lugar.

Et voila!

Chapter 5

Llamadas al sistema

Esta es la sección donde entramos en las llamadas al sistema (y otras llamadas a librerías) que te permiten acceder a la funcionalidad de red de una máquina Unix, o de cualquier máquina que soporte la API de sockets (BSD, Windows, Linux, Mac, etc.) Cuando llamas a una de estas funciones, el kernel toma el control y hace todo el trabajo por ti automáticamente.

Donde la mayoría de la gente se queda atascada es en qué orden llamar a estas cosas. En eso, las páginas `man` (como probablemente hayas descubierto) no sirven de nada. Bien, para ayudar con esa terrible situación, he intentado presentar las llamadas al sistema en las siguientes secciones en *exactamente* (aproximadamente) el mismo orden en que necesitarás llamarlas en tus programas.

Eso, junto con unas pocas piezas de código de ejemplo aquí y allá, un poco de leche y galletas (que me temo, tendrás que suministrar tú mismo), y algunas agallas y coraje, ¡y estarás transmitiendo datos por Internet como el Hijo de Jon Postel!

(Ten en cuenta que, por razones de brevedad, muchos de los fragmentos de código que aparecen a continuación no incluyen la necesaria comprobación de errores. Y muy comúnmente asumen que el resultado de las llamadas a `getaddrinfo()` tienen éxito y devuelven una entrada válida en la lista enlazada. Ambas situaciones se abordan adecuadamente en los programas independientes, así que utilícelos como modelo.

5.1 `getaddrinfo()`– ¡Prepárate para lanzar!

Se trata de una función con muchas opciones, pero su uso es bastante sencillo. Ayuda a configurar las `structs` que necesitarás más adelante.

Un poco de historia: antes se usaba una función llamada `gethostbyname()` para hacer búsquedas DNS. Luego cargabas esa información a mano en una `struct sockaddr_in`, y la usabas en tus llamadas.

Afortunadamente, esto ya no es necesario. (Tampoco es deseable, si quieres 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, incluyendo búsquedas de DNS y nombres de servicio, ¡y además rellena las `structs` que necesitas!

Echémosle un vistazo.

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

int getaddrinfo(const char *node,     // e.g. "www.example.com" o IP
                const char *service,  // e.g. "http" o número de puerto
                const struct addrinfo *hints,
                struct addrinfo **res);
```

Si le das a esta función tres parámetros de entrada, te dará un puntero a una lista enlazada, `res`, de resultados.

El parámetro `node` es el nombre del host al que conectarse, o una dirección IP.

A continuación está el parámetro `service`, que puede ser un número de puerto, como «80», o el nombre de un servicio concreto (que se encuentra en La lista de puertos de IANA[1] o en el fichero `/etc/services` de su máquina Unix) como «http» o «ftp» o «telnet» o «smtp» o lo que sea.

Finalmente, el parámetro `hints` apunta a una `struct addrinfo` que ya has rellenado con información relevante.

Aquí tienes un ejemplo de llamada si eres un servidor que quiere escuchar en la dirección IP de tu host, puerto 3490. Ten en cuenta que esto en realidad no hace ninguna escucha o configuración de red; simplemente configura estructuras que usaremos más tarde:

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // apuntará a los resultados

memset(&hints, 0, sizeof hints); // asegúrese de que la estructura está vacía
hints.ai_family = AF_UNSPEC;     // no importa IPv4 o IPv6
hints.ai_socktype = SOCK_STREAM; // Sockets de flujo TCP
hints.ai_flags = AI_PASSIVE;     // rellena mi IP por mí

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

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

// ... hazlo todo hasta que ya no necesites servinfo ....

freeaddrinfo(servinfo); // liberar la lista enlazada
```

Fíjate en que he puesto `ai_family` a `AF_UNSPEC`, diciendo así que no me importa si usamos IPv4 o IPv6. Puedes ponerlo a `AF_INET` o `AF_INET6` si quieres uno u otro específicamente.

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

Entonces hacemos la llamada. Si hay un error (`getaddrinfo()` devuelve distinto de cero), podemos imprimirlo usando la función `gai_strerror()`. Si todo funciona correctamente, `servinfo` apuntará a una lista enlazada de `struct addrinfo`s, ¡cada una de las cuales contiene una `struct sockaddr` de algún tipo que podremos usar más tarde! ¡Genial!

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

Aquí hay un ejemplo de llamada si eres un cliente que quiere conectarse a un servidor en particular, digamos “www.example.net” puerto 3490. De nuevo, esto no conecta realmente, pero establece las estructuras que usaremos más tarde:

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // apuntará a los resultados
```

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

```
memset(&hints, 0, sizeof hints); // asegúrese de que la estructura está vacía
hints.ai_family = AF_UNSPEC;     // no importa IPv4 o IPv6
hints.ai_socktype = SOCK_STREAM; // Sockets de flujo TCP

// prepárese para conectarse
status = getaddrinfo("www.example.net", "3490", &hints, &servinfo);

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

// etc.
```

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

```
/*
** showip.c -- muestra las direcciones IP de una máquina dada
** 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 hints, *res, *p;
    int status;
    char ipstr[INET6_ADDRSTRLEN];

    if (argc != 2) {
        fprintf(stderr,"usage: showip hostname\n");
        return 1;
    }

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC; // AF_INET o AF_INET6 para forzar la versión
    hints.ai_socktype = SOCK_STREAM;

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

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

    for(p = res;p != NULL; p = p->ai_next) {
        void *addr;
        char *ipver;

        // obtener el puntero a la propia dirección,
```

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

```
        // campos diferentes en IPv4 e IPv6:
        if (p->ai_family == AF_INET) { // IPv4
            struct sockaddr_in *ipv4 = (struct sockaddr_in *)p->ai_addr;
            addr = &(ipv4->sin_addr);
            ipver = "IPv4";
        } else { // IPv6
            struct sockaddr_in6 *ipv6 = (struct sockaddr_in6 *)p->ai_addr;
            addr = &(ipv6->sin6_addr);
            ipver = "IPv6";
        }

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

    freeaddrinfo(res); // liberar la lista enlazada

    return 0;
}
```

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

(Hay un poco de fealdad ahí donde tenemos que indagar en los diferentes tipos de `struct sockaddr`s dependiendo de la versión IP. Lo siento. No estoy seguro de una mejor manera de evitarlo).

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

```
$ showip www.example.net
IP addresses for www.example.net:

  IPv4: 192.0.2.88

$ showip ipv6.example.com
IP addresses for ipv6.example.com:

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

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

5.2 `socket()`–¡Consigue el Descriptor de Archivo!

Supongo que no puedo posponerlo más… tengo que hablar de la llamada al sistema `socket()` . Aquí está el desglose:

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

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

Pero, ¿qué son estos argumentos? Permiten determinar qué tipo de socket se desea (IPv4 o IPv6, stream o datagrama, y TCP o UDP).

Antes la gente codificaba estos valores, y todavía se puede hacer. (domain es PF_INET o PF_INET6, type es SOCK_STREAM o SOCK_DGRAM, y protocol puede establecerse a 0 para elegir el protocolo apropiado para el type dado. O puede llamar a getprotobyname() para buscar el protocolo que desee, «tcp» o «udp»).

(Este PF_INET es un pariente cercano de la AF_INET que puedes usar al inicializar el campo sin_family en tu struct sockaddr_in. De hecho, están tan relacionadas que tienen el mismo valor, y muchos programadores llaman a socket() y pasan AF_INET como primer argumento en lugar de PF_INET. Ahora, coge leche y galletas, porque es hora de un cuento. Érase una vez, hace mucho tiempo, se pensó que tal vez una familia de direcciones (lo que significa «AF» en AF_INET) podría soportar varios protocolos que se referían a su familia de protocolos (lo que significa «PF» en PF_INET). Eso no ocurrió. Y todos vivieron felices para siempre, The End. Así que lo más correcto es usar AF_INET en tu struct sockaddr_in y PF_INET en tu llamada a socket()).

En fin, basta ya. Lo que realmente quieres hacer es usar los valores de los resultados de la llamada a getaddrinfo(), e introducirlos en socket() directamente así:

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

// hacer la búsqueda
// [fingir que ya hemos rellenado la estructura «hints»]
getaddrinfo("www.example.com", "http", &hints, &res);

// de nuevo, deberías hacer una comprobación de errores en getaddrinfo(), y
// recorrer la lista enlazada «res» buscando entradas válidas en lugar
// de simplemente asumir que la primera es buena (como hacen muchos
// de estos ejemplos).
// Ver la sección cliente/servidor para ejemplos reales.

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

socket() simplemente le devuelve un *descriptor de socket* que puede usar en posteriores llamadas al sistema, o -1 en caso de error. La variable global errno se establece al valor del error (vea la página de manual errno para más detalles, y una nota rápida sobre el uso de errno en programas multihilo).

Bien, bien, bien, pero ¿para qué sirve este socket? La respuesta es que realmente no sirve para nada por sí mismo, y necesitas seguir leyendo y hacer más llamadas al sistema para que tenga algún sentido.

5.3 bind()—¿En qué puerto estoy?

Una vez que tengas un socket, puede que tengas que asociar ese socket con un puerto en tu máquina local. (Esto se hace comúnmente si vas a listen() para conexiones entrantes en un puerto específico—los juegos multijugador en red hacen esto cuando te dicen “conéctate a 192.168.5.10 puerto 3490”). El número de puerto es utilizado por el núcleo para hacer coincidir un paquete entrante con el descriptor de socket de un determinado proceso. Si sólo vas a hacer una función connect() (porque eres el cliente, no el servidor), esto es probablemente innecesario. Léelo de todos modos, sólo por diversión.

Aquí está la sinopsis de la llamada al sistema bind():

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

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

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

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

```
struct addrinfo hints, *res;
int sockfd;

// primero, carga los structs de direcciones con getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // utilice IPv4 o IPv6, lo que corresponda
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // rellena mi IP por mí

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

// crea un socket:

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

// bind (vincularlo) al puerto que pasamos a getaddrinfo():

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

Al usar la bandera `AI_PASSIVE`, le estoy diciendo al programa que se enlace a la IP del host en el que se está ejecutando. Si quiere enlazarse a una dirección IP local específica, elimine la opción `AI_PASSIVE` y ponga una dirección IP como primer argumento de `getaddrinfo()`.

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

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

```
//¡¡¡ESTE ES EL VIEJO CAMINO !!!

int sockfd;
struct sockaddr_in my_addr;

sockfd = socket(PF_INET, SOCK_STREAM, 0);

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

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

En el código anterior, también podrías asignar `INADDR_ANY` al campo `s_addr` si quisieras enlazar con tu 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` de tu `struct sockaddr_in6`. (También existe una macro `IN6ADDR_ANY_INIT` que puedes usar en un inicializador de variables).

Otra cosa a tener en cuenta al llamar a `bind()`: no te pases con los números de puerto. Todos los puertos por debajo de 1024 están RESERVADOS (a menos que seas el superusuario). Puedes tener cualquier número de puerto por encima de ese, hasta 65535 (siempre que no estén siendo usados por otro programa).

A veces, puede que te des cuenta, intentas volver a ejecutar un servidor y `bind()` falla, alegando Address already in use (Dirección ya en uso). ¿Qué significa esto? Bueno, un pedacito de un socket que estaba conectado todavía está dando vueltas en el kernel, y está acaparando el puerto. Puedes esperar a que se

borre (un minuto más o menos), o añadir código a tu programa permitiéndole reutilizar el puerto, como esto:

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

// desaparece el molesto mensaje de error "Address already in use"
if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof yes) == -1) {
    perror("setsockopt");
    exit(1);
}
```

Una pequeña nota final sobre `bind()`: hay veces en las que no es absolutamente necesario llamarla. Si estás `connect()` a una máquina remota y no te importa cuál es tu puerto local (como es el caso de `telnet` donde sólo te importa el puerto remoto), puedes simplemente llamar a `connect()`, que comprobará si el socket está desatado y hace `bind()` a un puerto local no utilizado si es necesario.

5.4 `connect()`–¡Hey, tú!

Imaginemos por unos minutos que eres una aplicación telnet. Tu usuario te ordena (como en la película *TRON*) que obtengas un descriptor de fichero socket. Tu cumples y llamas a `socket()`. A continuación, el usuario te dice que te conectes a `10.12.110.57` en el puerto `23` (el puerto telnet estándar). ¡Vaya! ¿Qué haces ahora?

Por suerte para ti, ahora estás leyendo 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 fichero de socket, devuelto por la llamada a `socket()`, `serv_addr` es una `struct sockaddr` que contiene el puerto de destino y la dirección IP, y `addrlen` es la longitud en bytes de la estructura de la dirección del servidor.

Toda esta información se puede deducir de los resultados de la llamada `getaddrinfo()`, que es lo mejor.

¿Esto empieza a tener más sentido? No te oigo desde aquí, así que espero que sí. Veamos un ejemplo en el que hacemos una conexión de socket a `www.example.com`, puerto `3490`:

```
struct addrinfo hints, *res;
int sockfd;

// primero, carga los structs de direcciones con getaddrinfo():

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

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

// crea un socket:

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

```
// connect!

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

De nuevo, los programas de la vieja escuela rellenaban su propia `struct sockaddr_in` para pasarla a `connect()`. Puedes hacerlo si quieres. Vea la nota similar en la sección sección `bind()`, arriba.

Asegúrese de comprobar el valor de retorno de `connect()`—devolverá `-1` en caso de error y establecerá la variable `errno`.

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

5.5 `listen()`—¿Puede alguien llamarme, por favor?

Bien, es hora de cambiar de ritmo. ¿Qué pasa si usted no desea conectarse a un host remoto. Digamos, sólo por diversión, que quieres esperar las conexiones entrantes y manejarlas de alguna manera. El proceso tiene dos pasos: primero `listen()`, luego `accept()` (ver más abajo).

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

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

`sockfd` es el descriptor de fichero de socket habitual de la llamada al sistema `socket()`. `backlog` es el número de conexiones permitidas en la cola de entrada. ¿Qué significa esto? Bueno, las conexiones entrantes van a esperar en esta cola hasta que las `acept()` (ver más abajo) y este es el límite de cuántas pueden esperar en la cola. La mayoría de los sistemas limitan silenciosamente este número a unas 20; probablemente puedas dejarlo en `5` o `10`.

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

Bien, como puedes 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!) Así que si vas a estar escuchando conexiones entrantes, la secuencia de llamadas al sistema que harás es:

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

Dejaré esto en lugar del código de ejemplo, ya que se explica por sí mismo. (El código de la sección `accept()`, más abajo, es más completo). La parte realmente complicada de todo esto es la llamada a `accept()`.

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

Prepárate… ¡la llamada a `accept()` es un poco rara! Lo que va a ocurrir es lo siguiente: alguien muy muy lejano intentará `conect()` a tu máquina en un puerto en el que estás `listen()`. Su conexión estará en cola esperando a ser aceptada. Llamas a `accept()` y le dices que obtenga la conexión pendiente. Te devolverá un *nuevo descriptor de fichero de socket* para usar en esta única conexión. Así es, ¡de repente tienes *dos descriptores de socket* por el precio de uno! El original sigue esperando nuevas conexiones, y el recién creado está finalmente listo para `send()` y `recv()`. ¡Ya está!

La llamada 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 de socket `listen()`. Bastante fácil. `addr` será normalmente un puntero a una `struct sockaddr_storage` local. Aquí es donde irá la información sobre la conexión entrante (y con ella puedes determinar qué host te está llamando desde qué puerto). `addrlen` es una variable entera local que debe ser ajustada a `sizeof(struct sockaddr_storage)` antes de que su dirección sea pasada a `accept()`. `accept()` no pondrá más que esa cantidad de bytes en `addr`. Si pone menos, cambiará el valor de `addrlen` para reflejarlo.

¿Adivina qué? `accept()` devuelve `-1` y pone `errno` si se produce un error. Apuesto a que no te lo imaginabas.

Como antes, esto es mucho para absorber en un trozo, así que aquí tienes un fragmento de código de ejemplo para que lo veas:

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

#define MYPORT "3490"  // el puerto al que se conectarán los usuarios
#define BACKLOG 10     // cuántas conexiones pendientes tendrá la cola

int main(void)
{
    struct sockaddr_storage their_addr;
    socklen_t addr_size;
    struct addrinfo hints, *res;
    int sockfd, new_fd;

    // ¡¡¡no olvides tu comprobación de errores para estas llamadas!!!

    // primero, carga los structs de direcciones con getaddrinfo():

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;  // utilice IPv4 o IPv6, lo que corresponda
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;     // rellena mi IP por mí

    getaddrinfo(NULL, MYPORT, &hints, &res);

    // crear un socket, enlazarlo y escuchar en él:

    sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    bind(sockfd, res->ai_addr, res->ai_addrlen);
    listen(sockfd, BACKLOG);

    // ahora acepta una conexión entrante:

    addr_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);

    // ¡listo para comunicar en el descriptor de socket new_fd!
    .
    .
```

```
    .
```

De nuevo, ten en cuenta que usaremos el descriptor de socket `new_fd` para todas las llamadas `send()` y `recv()`. Si sólo estás recibiendo una única conexión, puedes `close()` el `sockfd` 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 sirven para comunicarse a través de sockets de flujo o sockets de datagramas conectados. Si desea utilizar sockets de datagramas normales no conectados, tendrá que ver la sección sobre `sendto()` and `recvfrom()`, más abajo.

> Aquí hay algo que puede (o no) ser nuevo para ti: estas son llamadas *bloqueantes*. Esto es, `recv()` se *bloqueará* hasta que haya datos listos para recibir. «¡¿Pero qué significa `bloquear ¡¿Ya?! Significa que tu programa se va a detener ahí, en esa llamada al sistema, hasta qu` también puede bloquearse si el material que estás enviando está atascado de alguna manera, pero eso es más raro. Revisaremos este concepto más adelante](#blocking), y hablaremos de cómo evitarlo cuando lo necesites.

Here's the `send()` call:

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

`sockfd` es el descriptor de socket al que quieres enviar datos (ya sea el devuelto por `socket()` o el que obtuviste con `accept()`). `msg` es un puntero a los datos que quieres enviar, y `len` es la longitud de esos datos en bytes. Sólo tienes que poner `flags` a `0`. (Vea la página man `send()` para más información sobre flags).

Un ejemplo de código podría ser:

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

`send()` devuelve el número de bytes realmente enviados—¡esto podría ser menor que el número que le dijiste que enviara!_ Verás, a veces le dices que envíe un montón de datos y simplemente no puede manejarlo. Enviará tantos datos como pueda, y confiará en que le envíes 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 o así) *probablemente* se las arreglará para enviarlo todo de una vez. De nuevo, `-1` se devuelve en caso de error, y `errno` se establece al número de error.

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

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

`sockfd` es el descriptor de socket del que leer, `buf` es el buffer en el que leer la información, `len` es la longitud máxima del buffer, y `flags` puede ser de nuevo `0`. (Ver la página man de `recv()` para información sobre banderas).

`Recv()` devuelve el número de bytes realmente leídos en el buffer, o `-1` en caso de error (con `errno` configurado).

Espere, recv() puede devolver 0. Esto sólo puede significar una cosa: ¡el lado remoto ha cerrado la conexión! Un valor de retorno 0 es la forma que tiene recv() de hacerte saber que esto ha ocurrido.

Ya está, ha sido fácil, ¿verdad? ¡Ahora puedes pasar datos de un lado a otro en sockets de flujo! ¡Vaya! ¡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 desconectados?". No hay problema, amigo. Tenemos justo lo que necesitas.

Como los sockets de datagramas no están conectados a un host remoto, ¿adivina qué información necesitamos dar antes de enviar un paquete? Exacto. La dirección de destino. Aquí está la primicia:

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

Como puede 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 una struct sockaddr (que probablemente será otra struct sockaddr_in o struct sockaddr_in6 o struct sockaddr_storage que has lanzado en el último momento) que contiene el destino.Dirección IP y puerto. tolen, un int en el fondo, puede ser simplemente puesto a sizeof *to o sizeof(struct sockaddr_stora).

Para obtener la estructura de direcciones de destino, probablemente la obtendrá de getaddrinfo(), o de recvfrom(), más abajo, o la rellenará a mano.

Al igual que con send(), sendto() devuelve el número de bytes realmente enviados (que, de nuevo, puede ser menor que el número de bytes que le dijo 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 local struct sockaddr_storage 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 ser inicializado a sizeof *from o sizeof(struct sockaddr_storage). Cuando la función retorna, fromlen contendrá la longitud de la dirección almacenada en from.

La función recvfrom() devuelve el número de bytes recibidos, o -1 en caso de error (con el valor errno correspondiente).

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

(Así que… aquí hay otra pregunta: ¿por qué la propia struct sockaddr no es lo suficientemente grande para cualquier dirección? Incluso convertimos la struct sockaddr_storage de propósito general en la struct sockaddr de propósito general. Parece superfluo y redundante. La respuesta es, simplemente no es lo suficientemente grande, y supongo que cambiarlo en este punto sería problemático. Así que hicieron uno nuevo).

Recuerde, si usted connect() un socket de datagramas, puedes simplemente usar send() y recv() para todas tus transacciones. El socket en sí sigue siendo un socket de datagrama y los paquetes siguen usando UDP, pero la interfaz de socket añadirá automáticamente la información de destino y origen por ti.

5.9 `close()` y `shutdown()`—¡Fuera de mi vista!

¡Uf! Llevas todo el día enviando y recibiendo datos y ya no puedes más. Estás listo para cerrar la conexión en tu descriptor de socket. Esto es fácil. Puedes usar el archivo Unix normal descriptor `close()`:

```
close(sockfd);
```

Esto evitará más lecturas y escrituras en el socket. Cualquiera que intente leer o escribir en 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 ambas (igual que hace `close()`). Sinopsis:

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

`sockfd` es el descriptor de fichero de socket que quieres apagar, y `how` es uno de los siguientes:

`how`	Efecto
`0`	No se admiten más recepciones
`1`	No se admiten más envíos
`2`	Otros envíos y recepciones no están permitidos (como `close()`)

`shutdown()` devuelve `0` en caso de éxito, y `-1` en caso de error (con el valor `errno` correspondiente).

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

Es importante notar que `shutdown()` no cierra realmente el descriptor de fichero—solo cambia su usabilidad. Para liberar un descriptor de socket, necesitas usar `close()`.

No hay nada que hacer.

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

5.10 `getpeername()`—¿Quién es usted?

Esta función es tan fácil.

Es tan fácil que casi no le doy su propia sección. Pero aquí está.

La función `getpeername()` le dirá quién está en el otro extremo de un socket stream conectado. La sinopsis:

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

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

`sockfd` es el descriptor del socket conectado, `addr` es un puntero a `struct sockaddr` (o `struct sockaddr_in`) que contendrá la información sobre el otro lado de la conexión, y `addrlen` es un puntero a `int`, que debería inicializarse a `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 para imprimir u obtener más información. No, no puedes obtener su nombre de usuario. (Vale, de acuerdo. Si el otro ordenador

está ejecutando un demonio ident, esto es posible. Esto, sin embargo, está fuera del alcance de este documento. Consulta RFC 1413[3] para más información).

5.11 `gethostname()`—¿Quién soy yo?

Aún más fácil que `getpeername()` es la función `gethostname()`. Devuelve el nombre del ordenador en el que se ejecuta el programa. El nombre puede ser utilizado por `getaddrinfo()`, arriba, para determinar la dirección IP de su 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, size_t size);
```

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

La función devuelve `0` en caso de éxito, y `-1` en caso de error, estableciendo `errno` como de costumbre.

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

Chapter 6

Cliente-Servidor

El mundo es cliente-servidor. Casi todo en la red tiene que ver con procesos cliente que hablan con procesos servidor y viceversa. Tomemos `telnet`, por ejemplo. Cuando te conectas 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, te prepara un prompt de login, etc.

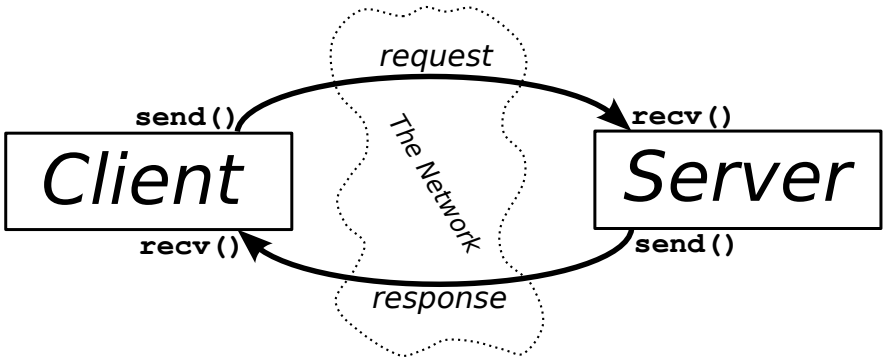

Figure 6.1: Client-Server Interaction.

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

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

6.1 Un simple servidor de streaming

Todo lo que hace este servidor es enviar la cadena `¡Hola, mundo!` a través de una conexión stream. Todo lo que necesitas hacer para probar este servidor es ejecutarlo en una ventana, y utilizar telnet desde otro lugar con:

```
$ telnet remotehostname 3490
```

donde `remotehostname` es el nombre de la máquina en la que lo estás ejecutando.

El código del servidor[1]:

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

```
/*
** server.c -- demostración de un servidor de streaming socket
*/

#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 <netdb.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

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

#define BACKLOG 10   // cuántas conexiones pendientes tendrá la cola

void sigchld_handler(int s)
{
   // waitpid() podría sobreescribir errno, así que lo guardamos y restauramos:
    int saved_errno = errno;

    while(waitpid(-1, NULL, WNOHANG) > 0);

    errno = saved_errno;
}


// 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(void)
{
    int sockfd, new_fd;  // escucha en sock_fd, nueva conexión en new_fd
    struct addrinfo hints, *servinfo, *p;
   struct sockaddr_storage their_addr; // información sobre la dirección del conector
    socklen_t sin_size;
    struct sigaction sa;
    int yes=1;
    char s[INET6_ADDRSTRLEN];
    int rv;

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE; // utilizar mi IP
```

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

   // bucle a través de todos los resultados y se unen a la primera que podemos
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("server: socket");
            continue;
        }

        if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes,
                sizeof(int)) == -1) {
            perror("setsockopt");
            exit(1);
        }

        if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("server: bind");
            continue;
        }

        break;
    }

    freeaddrinfo(servinfo); // todo hecho con esta estructura

    if (p == NULL)  {
        fprintf(stderr, "server: failed to bind\n");
        exit(1);
    }

    if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
    }

    sa.sa_handler = sigchld_handler; // cosechar todos los procesos muertos
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }

    printf("server: waiting for connections...\n");

    while(1) {  // main accept() loop
        sin_size = sizeof their_addr;
        new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
        if (new_fd == -1) {
            perror("accept");
            continue;
```

```
        }

        inet_ntop(their_addr.ss_family,
            get_in_addr((struct sockaddr *)&their_addr),
            s, sizeof s);
        printf("server: got connection from %s\n", s);

        if (!fork()) { // este es el proceso hijo
            close(sockfd); // el niño no necesita la escucha
            if (send(new_fd, "Hello, world!", 13, 0) == -1)
                perror("send");
            close(new_fd);
            exit(0);
        }
        close(new_fd);  // padre no necesita esto
    }

    return 0;
}
```

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

(Además, todo esto de `sigaction()` puede ser nueva para ti—eso está bien. El código que está ahí es responsable de cosechar procesos zombies que aparecen cuando los procesos hijo `fork()` salen. Si haces muchos zombies y no los cosechas, el administrador de tu sistema se agitará).

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

6.2 Un cliente de streaming sencillo

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

The client source[2]:

```
/*
** client.c -- a stream socket client demo
*/

#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>

#define PORT "3490" // el puerto al que se conectará el cliente

#define MAXDATASIZE 100 // número máximo de bytes que podemos obtener a la vez
```

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

```
// 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(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct addrinfo hints, *servinfo, *p;
    int rv;
    char s[INET6_ADDRSTRLEN];

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

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

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

   // bucle a través de todos los resultados y conectarse a la primera que podemos
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("client: socket");
            continue;
        }

        if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("client: connect");
            continue;
        }

        break;
    }

    if (p == NULL) {
        fprintf(stderr, "client: failed to connect\n");
        return 2;
    }

    inet_ntop(p->ai_family, get_in_addr((struct sockaddr *)p->ai_addr),
            s, sizeof s);
    printf("client: connecting to %s\n", s);
```

```
    freeaddrinfo(servinfo); // all done with this structure

    if ((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {
        perror("recv");
        exit(1);
    }

    buf[numbytes] = '\0';

    printf("client: received '%s'\n",buf);

    close(sockfd);

    return 0;
}
```

Observe que si no ejecuta el servidor antes de ejecutar el cliente, `connect()` devuelve “Connection refused”. Muy útil.

6.3 Datagram Sockets (Sockets de datagramas)

Ya hemos cubierto los fundamentos de los sockets de datagramas UDP con nuestra discusión de `sendto()` y `recvfrom()`, más arriba, así que sólo presentaré un par de programas de ejemplo: `talker.c` y `listener.c`.

`listener` se sienta en una máquina esperando un paquete entrante en el puerto 4950. El `talker` envía un paquete a ese puerto, en la máquina especificada, que contiene lo que el usuario introduzca en la línea de comandos.

Dado que los sockets de datagramas no tienen conexión y lanzan paquetes al éter sin tener en cuenta el éxito, vamos a decirle al cliente y al servidor que utilicen 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, aún podríamos tener el desajuste, pero el error en `connect()` para una familia de direcciones nos haría reintentar para la otra).

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

```
/*
** listener.c -- a datagram sockets "server" demo
*/

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

#define MYPORT "4950"    // el puerto al que se conectarán los usuarios

#define MAXBUFLEN 100
```

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

```
// 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(void)
{
    int sockfd;
    struct addrinfo hints, *servinfo, *p;
    int rv;
    int numbytes;
    struct sockaddr_storage their_addr;
    char buf[MAXBUFLEN];
    socklen_t addr_len;
    char s[INET6_ADDRSTRLEN];

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_INET6; // establecer a AF_INET para usar IPv4
    hints.ai_socktype = SOCK_DGRAM;
    hints.ai_flags = AI_PASSIVE; // utilizar mi IP

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

   // bucle a través de todos los resultados y se unen a la primera que podemos
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("listener: socket");
            continue;
        }

        if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("listener: bind");
            continue;
        }

        break;
    }

    if (p == NULL) {
        fprintf(stderr, "listener: failed to bind socket\n");
        return 2;
    }

    freeaddrinfo(servinfo);

    printf("listener: waiting to recvfrom...\n");
```

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

    printf("listener: got packet from %s\n",
        inet_ntop(their_addr.ss_family,
            get_in_addr((struct sockaddr *)&their_addr),
            s, sizeof s));
    printf("listener: packet is %d bytes long\n", numbytes);
    buf[numbytes] = '\0';
    printf("listener: packet contains \"%s\"\n", buf);

    close(sockfd);

    return 0;
}
```

Observa que en nuestra llamada a `getaddrinfo()` finalmente estamos usando `SOCK_DGRAM`. Observe también que no hay necesidad de `listen()` o `accept()`. Esta es una de las ventajas de usar sockets de datagramas desconectados.

A continuación viene el fuente para `talker.c`[4]:

```
/*
** talker.c -- a datagram "client" demo
*/

#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 SERVERPORT "4950"    // el puerto al que se conectarán los usuarios

int main(int argc, char *argv[])
{
    int sockfd;
    struct addrinfo hints, *servinfo, *p;
    int rv;
    int numbytes;

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

    memset(&hints, 0, sizeof hints);
```

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

```
    hints.ai_family = AF_INET6; // establecer a AF_INET para usar IPv4
    hints.ai_socktype = SOCK_DGRAM;

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

    // bucle a través de todos los resultados y hacer 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("talker: socket");
            continue;
        }

        break;
    }

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

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

    freeaddrinfo(servinfo);

    printf("talker: sent %d bytes to %s\n", numbytes, argv[1]);
    close(sockfd);

    return 0;
}
```

Y eso es todo. Ejecuta `listener` en una máquina, luego ejecuta `talker` en otra. ¡Mira cómo se comunican! Diversión para toda la familia.

Esta vez ni siquiera tienes que ejecutar el servidor. Puedes ejecutar `talker` por sí mismo, y simplemente lanzará paquetes al éter donde desaparecerán si nadie está preparado con un `recvfrom()` en el otro lado. Recuerde: ¡no se garantiza que los datos enviados usando sockets de datagramas UDP lleguen!

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

Chapter 7

Técnicas ligeramente avanzadas

Estas no son *realmente* avanzadas, pero salen de los niveles más básicos que ya hemos cubierto. De hecho, si has llegado hasta aquí, ¡deberías considerarte bastante experto en los fundamentos de la programación de redes Unix! ¡Enhorabuena!

Así que, aquí, vamos al valiente nuevo mundo de algunas de las cosas más esotéricas que podrías querer aprender sobre sockets. ¡Adelante!

7.1 Bloqueo

Bloqueo. Ya ha oído hablar de él, pero ¿qué es? En pocas palabras, "block" «bloquear» es la jerga técnica para "sleep" «dormir». Probablemente has notado que cuando ejecutas `listener`, arriba, se queda ahí sentado hasta que llega un paquete. Lo que ocurrió es que llamó a `recvfrom()`, no había datos, y por eso se dice que `recvfrom()` se «bloquea» (es decir, duerme ahí) hasta que llegan algunos datos.

Muchas funciones se bloquean. La función `accept()` se bloquea. Todas las funciones `recv()` se bloquean. La razón por la que pueden hacer esto es porque se les permite. Cuando se crea por primera vez el descriptor del socket con `socket()`, el kernel lo configura como bloqueante. Si no quieres que un socket sea bloqueante, tienes que hacer una llamada a la función `fcntl()` :

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

Al configurar un socket como no bloqueante, puede «sondear» el socket para obtener información. Si intentas leer de un socket no bloqueante y no hay datos allí, no está permitido bloquearlo—devolverá `-1` y `errno` será puesto a `EAGAIN` o `EWOULDBLOCK`.

(Espera—puede devolver `EAGAIN` *o* `EWOULDBLOCK`) `EWOULDBLOCK`? ¿Cuál comprueba? En realidad, la especificación no especifica cuál devolverá su sistema, así que, por motivos de portabilidad, compruebe ambas).

En general, sin embargo, este tipo de sondeo es una mala idea. Si pones tu programa en un busy-wait buscando datos en el socket, chuparás tiempo de CPU como si estuviera pasando de moda. Una solución más elegante para comprobar si hay datos esperando a ser leídos viene en la siguiente sección de `poll()`.

7.2 `poll()`—Multiplexación síncrona de E/S

Lo que realmente quieres es poder monitorizar de alguna manera un *montón* de sockets a la vez y luego manejar los que tienen datos listos. De esta manera no tienes que estar continuamente sondeando todos esos sockets para ver cuáles están listos para leer.

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

¿Cómo evitar las encuestas? Irónicamente, puedes 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 sólo nos haga saber cuándo hay datos listos para leer en qué sockets. Mientras tanto, nuestro proceso puede irse a dormir, ahorrando recursos del sistema.

El plan general es mantener un array de `struct pollfd`s con información sobre qué descriptores de socket queremos monitorizar, y qué tipo de eventos queremos monitorizar. El sistema operativo bloqueará la llamada a `poll()` hasta que ocurra uno de esos eventos (por ejemplo, «socket ready for reading») o hasta que ocurra un tiempo de espera especificado por el usuario.

De forma útil, un socket `listen()` devolverá «ready to read» cuando una nueva conexión entrante esté lista para ser `accept()`.

Basta de bromas. ¿Cómo usamos esto?

```
#include <poll.h>

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

`fds` es nuestro array de información (qué sockets monitorizar y para qué), `nfds` es el número de elementos en el array, y `timeout` es el tiempo de espera en milisegundos. Devuelve el número de elementos del array en los que se ha producido un evento.

Echemos un vistazo a esa `struct`:

```
struct pollfd {
    int fd;         // el descriptor del socket
    short events;   // mapa de bits de los eventos que nos interesan
    short revents;  // cuando poll() retorna, bitmap de eventos ocurridos
};
```

Así que vamos a tener una matriz de estos, y vamos a establecer el campo `fd` para cada elemento a un descriptor de socket que estamos interesados en monitorear. Y luego pondremos el campo `events` para indicar el tipo de eventos que nos interesan.

El campo `events` es el bitwise-OR de lo siguiente:

Macro |

Descripción	
`POLLIN`	Alertarme cuando los datos estén listos para `recv()` en este socket.
`POLLOUT`	Avisame cuando pueda `enviar()` datos a este socket sin bloquear.
`POLLHUP`	Alertarme cuando el remoto cerró la conexión.

Una vez que tengas tu array de `struct pollfd` en orden, entonces puedes pasarlo a `poll()`, pasando también el tamaño del array, así como un valor de tiempo de espera en milisegundos. (Puedes especificar un tiempo de espera negativo para esperar eternamente).

Después de que poll() retorne, puedes comprobar el campo revents para ver si POLLIN o POLLOUT está activado, indicando que el evento ha ocurrido.

(En realidad hay más cosas que puedes hacer con la llamada poll(). Vea la página man de poll(), más abajo, para más detalles).

Aquí está un ejemplo[1] donde esperaremos 2.5 segundos a que los datos estén listos para ser leídos desde la entrada estándar, es decir, cuando pulses RETURN:

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

int main(void)
{
    struct pollfd pfds[1]; // Más si desea supervisar más

    pfds[0].fd = 0;          // Entrada estándar
    pfds[0].events = POLLIN; // Avísame cuando esté listo para leer

    // Si necesitaras controlar también otras cosas:
    //pfds[1].fd = some_socket; // Algún descriptor de socket
    //pfds[1].events = POLLIN; // Avisa cuando esté listo para leer

    printf("Hit RETURN or wait 2.5 seconds for timeout\n");

    int num_events = poll(pfds, 1, 2500); // 2,5 segundos de espera

    if (num_events == 0) {
        printf("Poll timed out!\n");
    } else {
        int pollin_happened = pfds[0].revents & POLLIN;

        if (pollin_happened) {
            printf("File descriptor %d is ready to read\n", pfds[0].fd);
        } else {
            printf("Unexpected event occurred: %d\n", pfds[0].revents);
        }
    }

    return 0;
}
```

Observe de nuevo que poll() devuelve el número de elementos de la matriz pfds para los que se han producido eventos. No te dice *qué* elementos del array (aún tienes que escanearlo), pero te dice cuántas entradas tienen un campo revents distinto de cero (así que puedes dejar de escanear después de encontrar ese número).

Aquí pueden surgir un par de preguntas: ¿cómo añadir nuevos descriptores de fichero al conjunto que paso a poll()? Para esto, simplemente asegúrate de que tienes suficiente espacio en el array para todos los que necesites, o realloc() más espacio si es necesario.

¿Qué pasa con la eliminación de elementos del conjunto? Para esto, puedes copiar el último elemento del array encima del que estás borrando. Y luego pasar uno menos como recuento a poll(). Otra opción es establecer cualquier campo fd a un número negativo y poll() lo ignorará.

¿Cómo podemos juntar todo esto en un servidor de chat al que puedas conectarte por telnet?

Lo que haremos es iniciar un socket de escucha, y añadirlo al conjunto de descriptores de fichero de poll(). (Se mostrará listo para leer cuando haya una conexión entrante).

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

Entonces añadiremos nuevas conexiones a nuestro array `struct pollfd`. Y lo haremos crecer dinámicamente si nos quedamos sin espacio.

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

Y cuando una conexión esté lista para ser leída, leeremos sus datos y los enviaremos a todas las demás conexiones para que puedan ver lo que han escrito los demás usuarios.

Así que pruebe este servidor de pool[2]. Ejecútalo en una ventana, y luego `telnet localhost 9034` desde otras ventanas de terminal. Deberías poder ver lo que escribes en una ventana en las otras (después de pulsar RETURN).

No sólo eso, sino que si pulsas `CTRL-]` y tecleas `quit` para salir de `telnet`, el servidor debería detectar la desconexión y eliminarte del conjunto de descriptores de fichero.

```
/*
** pollserver.c -- a cheezy multiperson chat server
*/

#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>

#define PORT "9034"   // Puerto en el que estamos escuchando

// Get sockaddr, IPv4 or 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);
}

// Devuelve un socket de escucha
int get_listener_socket(void)
{
    int listener;     // Descriptor de socket de escucha
    int yes=1;        // Para setsockopt() SO_REUSEADDR, abajo
    int rv;

    struct addrinfo hints, *ai, *p;

    // Obtener un socket y enlazarlo
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
    if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) {
        fprintf(stderr, "pollserver: %s\n", gai_strerror(rv));
```

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

```
        exit(1);
    }

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

        // Elimina el molesto mensaje de error "address already in use"
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));

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

        break;
    }

    freeaddrinfo(ai); // Todo hecho con esto

    // Si llegamos aquí, significa que no nos ataron
    if (p == NULL) {
        return -1;
    }

    // Escuchar
    if (listen(listener, 10) == -1) {
        return -1;
    }

    return listener;
}

// Añadir un nuevo descriptor de fichero al conjunto
void add_to_pfds(struct pollfd *pfds[], int newfd, int *fd_count, int *fd_size)
{
    // Si no tenemos espacio, añadir más espacio en la matriz pfds
    if (*fd_count == *fd_size) {
        *fd_size *= 2; // Double it

        *pfds = realloc(*pfds, sizeof(**pfds) * (*fd_size));
    }

    (*pfds)[*fd_count].fd = newfd;
    (*pfds)[*fd_count].events = POLLIN; // Check ready-to-read

    (*fd_count)++;
}

// Eliminar un índice del conjunto
void del_from_pfds(struct pollfd pfds[], int i, int *fd_count)
{
    // Copia el del final sobre este
    pfds[i] = pfds[*fd_count-1];
```

```
    (*fd_count)--;
}

// Main
int main(void)
{
    int listener;        // Descriptor de socket de escucha

    int newfd;        // Nuevo descriptor de socket accept()ed
    struct sockaddr_storage remoteaddr; // Dirección del cliente
    socklen_t addrlen;

    char buf[256];    // Buffer para datos del cliente

    char remoteIP[INET6_ADDRSTRLEN];

    // Empezar con espacio para 5 conexiones
    // (Reasignaremos cuando sea necesario)
    int fd_count = 0;
    int fd_size = 5;
    struct pollfd *pfds = malloc(sizeof *pfds * fd_size);

    // Configurar y obtener un socket de escucha
    listener = get_listener_socket();

    if (listener == -1) {
        fprintf(stderr, "error getting listening socket\n");
        exit(1);
    }

    // Añadir el listener a set
    pfds[0].fd = listener;
   pfds[0].events = POLLIN; // Informe listo para leer en la conexión entrante

    fd_count = 1; // Para el oyente

    // Main loop
    for(;;) {
        int poll_count = poll(pfds, fd_count, -1);

        if (poll_count == -1) {
            perror("poll");
            exit(1);
        }

     // Ejecutar a través de las conexiones existentes en busca de datos para leer
        for(int i = 0; i < fd_count; i++) {

            // Comprobar si alguien está listo para leer
            if (pfds[i].revents & (POLLIN | POLLHUP)) { // ¡¡Tenemos uno!!

                if (pfds[i].fd == listener) {
              // Si el receptor está listo para leer, gestiona la nueva conexión

                    addrlen = sizeof remoteaddr;
                    newfd = accept(listener,
                        (struct sockaddr *)&remoteaddr,
```

```
                        &addrlen);

                    if (newfd == -1) {
                        perror("accept");
                    } else {
                        add_to_pfds(&pfds, newfd, &fd_count, &fd_size);

                        printf("pollserver: new connection from %s on "
                            "socket %d\n",
                            inet_ntop(remoteaddr.ss_family,
                                get_in_addr((struct sockaddr*)&remoteaddr),
                                remoteIP, INET6_ADDRSTRLEN),
                            newfd);
                    }
                } else {
                    // Si no es el oyente, somos un cliente normal
                    int nbytes = recv(pfds[i].fd, buf, sizeof buf, 0);

                    int sender_fd = pfds[i].fd;

                    if (nbytes <= 0) {
                        // Error o conexión cerrada por el cliente
                        if (nbytes == 0) {
                            // Connection closed
                        printf("pollserver: socket %d hung up\n", sender_fd);
                        } else {
                            perror("recv");
                        }

                        close(pfds[i].fd); // Bye!

                        del_from_pfds(pfds, i, &fd_count);

                    } else {
                        // Tenemos algunos buenos datos de un cliente

                        for(int j = 0; j < fd_count; j++) {
                            // ¡Enviar a todo el mundo!
                            int dest_fd = pfds[j].fd;

                            // Excepto el oyente y nosotros
                           if (dest_fd != listener && dest_fd != sender_fd) {
                                if (send(dest_fd, buf, nbytes, 0) == -1) {
                                    perror("send");
                                }
                            }
                        }
                    }
                } // END manejar datos del cliente
            } // END obtuvo datos listos para leer de poll()
        } // FINALIZAR el bucle a través de los descriptores de fichero
    } // END for(;;)--¡y pensabas que nunca acabaría!

    return 0;
}
```

En la siguiente sección, veremos una función similar más antigua llamada `select()`. Tanto `select()`

como `poll()` ofrecen una funcionalidad y un rendimiento similares, y sólo difieren realmente en cómo se utilizan. Puede que `select()` sea ligeramente más portable, pero quizás sea un poco más torpe en su uso. Elige el que más te guste, siempre que esté soportado en tu sistema.

7.3 `select()`—Multiplexación síncrona de E/S a la antigua usanza

Esta función es un tanto extraña, pero resulta muy útil. Tomemos la siguiente situación: eres un servidor y quieres escuchar conexiones entrantes así como seguir leyendo de las conexiones que ya tienes.

No hay problema, dices, sólo un `accept()` y un par de `recv()`s. No tan rápido, amigo. ¿Qué pasa si estás bloqueando una llamada a `accept()`? ¿Cómo vas a «recuperar» datos al mismo tiempo? «¡Usa sockets no bloqueantes!» ¡No puede ser! No quieres ser un devorador de CPU. ¿Entonces qué?

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

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

Sin más dilación, te ofrezco 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 monitoriza «conjuntos» de descriptores de fichero; en particular `readfds`, `writefds`, y `exceptfds`. Si quiere ver si puede leer de la entrada estándar y de algún descriptor de socket, `sockfd`, sólo tiene que añadir los descriptores de fichero `0` y `sockfd` al conjunto `readfds`. El parámetro `numfds` debe establecerse con los valores del descriptor de fichero más alto más uno. En este ejemplo, debería ser `sockfd+1`, ya que es seguramente mayor que la entrada estándar (`0`).

Cuando `select()` retorna, `readfds` se modificará para reflejar cuál de los descriptores de fichero seleccionados está listo para ser leído. Puedes probarlos con la macro `FD_ISSET()`, más abajo.

Antes de avanzar mucho más, hablaré de cómo manipular estos conjuntos. Cada conjunto es del tipo `fd_set`. Las siguientes macros operan sobre este tipo:

Función	Descripción
`FD_SET(int fd, fd_set *set);`	Añade `fd` al `set`.
`FD_CLR(int fd, fd_set *set);`	Eliminar `fd` del `set`.
`FD_ISSET(int fd, fd_set *set);`	Devuelve true si `fd` está en el `set`.
`FD_ZERO(fd_set *set);`	Borra todas las entradas del `set`.

Por último, ¿qué es ese extraño `struct timeval`? Bueno, a veces no quieres esperar eternamente a que alguien te envíe algún dato. Puede que cada 96 segundos quieras imprimir «Still Going…» en el terminal aunque no haya pasado nada. Esta estructura de tiempo te permite especificar un periodo de tiempo de espera. Si se excede el tiempo y `select()` todavía no ha encontrado ningún descriptor de fichero listo, volverá para que puedas continuar procesando.

La `struct timeval` tiene los siguientes campos:

```
struct timeval {
    int tv_sec;     // seconds
    int tv_usec;    // microseconds
};
```

Sólo tienes que poner `tv_sec` al número de segundos que hay que esperar, y poner `tv_usec` al número de microsegundos que hay que esperar. Sí, son _micro_segundos, no milisegundos. Hay 1.000 microsegundos en un milisegundo, y 1.000 milisegundos en un segundo. Por tanto, hay 1.000.000 de microsegundos en un segundo. ¿Por qué se dice «usec»? Se supone que la «u» se parece a la letra griega μ (Mu) que utilizamos para «micro». Además, cuando la función regresa, `timeout` *podría* actualizarse para mostrar el tiempo restante. Esto depende del tipo de Unix que estés usando.

¡Sí! ¡Tenemos un temporizador con resolución de microsegundos! Bueno, no cuentes con ello. Probablemente tendrás que esperar una parte del tiempo estándar de Unix, no importa lo pequeño que configures tu `struct timeval`.

Otras cosas de interés: Si estableces los campos de tu `struct timeval` a `0`, `select()` expirará inmediatamente, sondeando efectivamente todos los descriptores de fichero de tus conjuntos. Si establece el parámetro `timeout` a NULL, nunca se agotará el tiempo de espera, y esperará hasta que el primer descriptor de fichero esté listo. Por último, si no te importa esperar a un determinado conjunto, puedes ponerlo a NULL en la llamada a `select()`.

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

```
/*
** select.c -- a select() demo
*/

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

#define STDIN 0  // descriptor de fichero para la entrada estándar

int main(void)
{
    struct timeval tv;
    fd_set readfds;

    tv.tv_sec = 2;
    tv.tv_usec = 500000;

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

    // no te preocupes por writefds y exceptfds:
    select(STDIN+1, &readfds, NULL, NULL, &tv);

    if (FD_ISSET(STDIN, &readfds))
        printf("A key was pressed!\n");
    else
        printf("Timed out.\n");

    return 0;
}
```

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

Si estás en un terminal con búfer de línea, la tecla que pulses debe ser RETURN o se agotará el tiempo de espera.

Ahora, algunos de ustedes podrían pensar que esta es una gran manera de esperar datos en un socket de datagramas–y tienen razón: *podría* serlo. Algunas Unices pueden usar select de esta manera, y otras no. Debería ver lo que dice su página man local al respecto si quiere intentarlo.

Algunas Unices actualizan el tiempo en su `struct timeval` para reflejar la cantidad de tiempo restante antes de un tiempo de espera. Pero otras no lo hacen. No confíes en que eso ocurra si quieres ser portable. (Use `gettimeofday()` si necesitas controlar el tiempo transcurrido. Es un fastidio, lo sé, pero así son las cosas).

¿Qué ocurre si un socket del conjunto de lectura cierra la conexión? Bueno, en ese caso, `select()` vuelve con ese descriptor de socket establecido como «listo para leer». Cuando 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 es `listen()`, puedes comprobar si hay una nueva conexión poniendo el descriptor de fichero de ese socket en el set `readfds`.

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

Pero, por demanda popular, aquí hay un ejemplo en profundidad. Desafortunadamente, la diferencia entre el sencillo ejemplo anterior y este es significativa. Pero échale un vistazo, y luego lee la descripción que sigue.

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

```
/*
** selectserver.c -- un 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>

#define PORT "9034"   // puerto en el que estamos escuchando

// 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(void)
{
    fd_set master; // lista maestra de descriptores de fichero
   fd_set read_fds; // lista temporal de descriptores de fichero para select()
```

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

```
    int fdmax; // número máximo de descriptores de fichero

    int listener; // descriptor de socket a la escucha
    int newfd; // descriptor de socket recién aceptado()ed
    struct sockaddr_storage remoteaddr; // dirección del cliente
    socklen_t addrlen;

    char buf[256];    // búfer para los datos del cliente
    int nbytes;

    char remoteIP[INET6_ADDRSTRLEN];

    int yes=1;        // para setsockopt() SO_REUSEADDR, abajo
    int i, j, rv;

    struct addrinfo hints, *ai, *p;

    FD_ZERO(&master);    // borrar los conjuntos maestro y temporal
    FD_ZERO(&read_fds);

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

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

        // perder el molesto mensaje de error "address already in use"
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));

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

        break;
    }

    // si llegamos aquí, significa que no nos ataron
    if (p == NULL) {
        fprintf(stderr, "selectserver: failed to bind\n");
        exit(2);
    }

    freeaddrinfo(ai); // todo hecho con esto

    // Escuchando
    if (listen(listener, 10) == -1) {
```

```
        perror("listen");
        exit(3);
    }

    // añadir el oyente al conjunto maestro
    FD_SET(listener, &master);

    // llevar la cuenta del mayor descriptor de fichero
    fdmax = listener; // so far, it's this one

    // bucle principal
    for(;;) {
        read_fds = master; // copy it
        if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
            perror("select");
            exit(4);
        }

        // recorre las conexiones existentes buscando datos para leer
        for(i = 0; i <= fdmax; i++) {
            if (FD_ISSET(i, &read_fds)) { // ¡¡tenemos uno!!
                if (i == listener) {
                    // gestionar nuevas conexiones
                    addrlen = sizeof remoteaddr;
                    newfd = accept(listener,
                        (struct sockaddr *)&remoteaddr,
                        &addrlen);

                    if (newfd == -1) {
                        perror("accept");
                    } else {
                      FD_SET(newfd, &master); // añadir al conjunto principal
                       if (newfd > fdmax) {    // llevar la cuenta del máximo
                            fdmax = newfd;
                        }
                        printf("selectserver: new connection from %s on "
                            "socket %d\n",
                            inet_ntop(remoteaddr.ss_family,
                                get_in_addr((struct sockaddr*)&remoteaddr),
                                remoteIP, INET6_ADDRSTRLEN),
                            newfd);
                    }
                } else {
                    // manejar los datos de un cliente
                    if ((nbytes = recv(i, buf, sizeof buf, 0)) <= 0) {
                        // error obtenido o conexión cerrada por el cliente
                        if (nbytes == 0) {
                            // conexión cerrada
                            printf("selectserver: socket %d hung up\n", i);
                        } else {
                            perror("recv");
                        }
                        close(i); // bye!
                       FD_CLR(i, &master); // eliminar del conjunto principal
                    } else {
                        // recibimos algunos datos de un cliente
                        for(j = 0; j <= fdmax; j++) {
```

```
                            // ¡enviar a todo el mundo!
                            if (FD_ISSET(j, &master)) {
                                // excepto el oyente y nosotros
                                if (j != listener && j != i) {
                                    if (send(j, buf, nbytes, 0) == -1) {
                                        perror("send");
                                    }
                                }
                            }
                        }
                    }
                } // END manejar datos del cliente
            } // END obtuvo nueva conexión entrante
        } // FINALIZAR bucle a través de descriptores de fichero
    } // END for(;;)--¡y pensabas que nunca terminaría!

    return 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 actualmente conectados, así como el descriptor de socket que está a la escucha de nuevas conexiones.

La razón por la que tengo el conjunto `master` es que `select()` realmente *cambia* el conjunto que le pasas para reflejar qué sockets están listos para leer. Como tengo que mantener un registro de las conexiones de una llamada de `select()` a la siguiente, debo almacenarlas de forma segura en algún lugar. En el último momento, copio el `master` en el `read_fds`, y entonces llamo a `select()`.

¿Pero esto no significa que cada vez que obtengo una nueva conexión, tengo que añadirla al conjunto `master`? Sí. ¿Y cada vez que se cierra una conexión, tengo que eliminarla del conjunto `master`? Pues sí.

Fíjate que compruebo cuando el socket `listener` está listo para leer. Cuando lo está, significa que tengo una nueva conexión pendiente, y la `acept()` y la añado al conjunto `master`. De forma similar, cuando una conexión cliente está lista para leer, y `recv()` devuelve `0`, sé que el cliente ha cerrado la conexión, y debo eliminarla del conjunto `master`.

Sin embargo, si la función `recv()` del cliente devuelve un valor distinto de cero, sé que se han recibido algunos datos. Así que lo obtengo, y luego recorro la lista de `master` y envío esos datos al resto de los clientes conectados.

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

Nota rápida para todos los fans de Linux: a veces, en raras circunstancias, la función `select()` de Linux puede devolver “ready-to-read” «listo para leer» ¡y no estar realmente listo para leer! Esto significa que bloqueará la función `read()` después de que `select()` diga que no lo hará. ¿Por qué pequeño…? De todas formas, la solución es poner la macro `O_NONBLOCK` en el socket receptor para que se produzca un error con `EWOULDBLOCK` (que puede ignorar con seguridad si ocurre). Vea la página de referencia `fcntl()` para más información sobre cómo configurar un socket como no-bloqueante.

Además, aquí hay una idea extra: hay otra función llamada `poll()` que se comporta de forma muy similar a `select()`, pero con un sistema diferente para gestionar los conjuntos de descriptores de fichero. Check it out!

7.4 Manejo de `send()`s parciales

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

Bueno, siguen en tu pequeño buffer esperando a ser enviados. Debido a circunstancias fuera de tu control, el kernel decidió no enviar todos los datos en un solo trozo, y ahora, amigo mío, depende de ti hacer que

los datos salgan.

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 bytes nos quedan por enviar
    int n;

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

    *len = total; // aquí se devuelve el número realmente enviado

    return n==-1?-1:0; // devuelve -1 en caso de fallo, 0 en caso de éxito
}
```

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

La función devuelve `-1` en caso de error (y `errno` se mantiene 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 hubiera un error. `sendall()` hará todo lo posible, resoplando y resoplando, para enviar los datos, pero si hay un error, se lo devolverá de inmediato.

Para completar, aquí hay 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("We only sent %d bytes because of the error!\n", len);
}
```

¿Qué ocurre en el receptor cuando llega parte de un paquete? Si los paquetes son de longitud variable, ¿cómo sabe el receptor cuándo termina un paquete y empieza otro? Sí, los escenarios del mundo real son un auténtico dolor de burros. Probablemente tengas que *encapsular* (¿recuerdas eso de la sección de encapsulación de datos allá al principio?) Sigue leyendo para más detalles.

7.5 Serialization—Cómo empaquetar datos

Es bastante fácil enviar datos de texto a través de la red, pero ¿qué pasa si quieres enviar datos «binarios» como `int`s o `float`s? Resulta que tienes varias opciones.

1. Convierte el número en texto con una función como `sprintf()`, luego envía el texto. El receptor volverá a convertir el texto en un número utilizando una función como `strtol()`.
2. Simplemente envíe los datos sin procesar, pasando un puntero a los datos a `send()`.
3. Codificar el número en un formato binario portable. El receptor lo decodificará.

¡Preestreno! ¡Sólo esta noche!

[*Se levanta el telón*]

Beej dice: «¡Prefiero el Método Tres, arriba!»

[*THE END*]

(Antes de empezar esta sección en serio, debo decirte que existen librerías para hacer esto, y que crear la tuya propia y mantenerla portable y sin errores es todo un reto. Así que busca y haz los deberes antes de decidirte a implementar esto tú mismo. Incluyo la información aquí para aquellos curiosos sobre cómo funcionan estas cosas).

En realidad, todos los métodos anteriores tienen sus inconvenientes y ventajas, pero, como he dicho, 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 puedes imprimir y leer fácilmente los datos que llegan por el cable. A veces es excelente utilizar un protocolo legible por humanos en una situación que no requiera mucho 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!

Segundo método: pasar los datos en bruto. Este es bastante fácil (¡pero peligroso!): simplemente toma un puntero a los datos a enviar, y llama a send con él.

```
double d = 3490.15926535;

send(s, &d, sizeof d, 0);  /* DANGER--non-portable! */
```

El receptor lo recibe así:

```
double d;

recv(s, &d, sizeof d, 0);  /* PELIGRO--¡no portable! */
```

Rápido, sencillo… ¿qué más se puede pedir? Bueno, resulta que no todas las arquitecturas representan un `doble` (o un `int`) con la misma representación de bits, ni siquiera con el mismo orden de bytes. El código es decididamente no portable. (Hey—quizás no necesites portabilidad, en cuyo caso esto es bonito y rápido).

Al empaquetar tipos enteros, ya hemos visto cómo la clase de funciones `htons()` puede ayudar a mantener las cosas portables transformando los números en Network Byte Order, y cómo eso es lo correcto. Desafortunadamente, no hay funciones similares para los tipos `float`. ¿Se ha perdido toda esperanza?

No tema. (¿Te has asustado por un segundo? ¿No? ¿Ni siquiera un poquito?) Hay algo que podemos hacer: podemos empaquetar (o «marshal», o «serializar», o uno de los mil millones de otros nombres) los datos en un formato binario conocido que el receptor pueda desempaquetar en el lado remoto.

¿Qué quiero decir con “formato binario conocido”? Bueno, ya hemos visto el ejemplo de `htons()`, ¿verdad? Cambia (o «codifica», si quieres verlo así) un número de cualquier formato del host a Network Byte Order. Para invertir (descodificar) el número, el receptor llama a `ntohs()`.

¿Pero no acababa de decir que no existía tal función para otros tipos no enteros? Sí. Pues sí. Y como no hay una forma estándar en C de hacer esto, es un poco complicado (un juego de palabras gratuito para los fans 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 empaquetar `float`s, aquí está algo rápido y sucio con mucho margen de mejora[6]:

[5] https://en.wikipedia.org/wiki/Internet_Relay_Chat
[6] https://beej.us/guide/bgnet/source/examples/pack.c

```
#include <stdint.h>

uint32_t htonf(float f)
{
    uint32_t p;
    uint32_t sign;

    if (f < 0) { sign = 1; f = -f; }
    else { sign = 0; }

    p = ((((uint32_t)f)&0x7fff)<<16) | (sign<<31); // parte entera y signo
    p |= (uint32_t)(((f - (int)f) * 65536.0f))&0xffff; // fracción

    return p;
}

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

    if (((p>>31)&0x1) == 0x1) { f = -f; } // bit de signo activado

    return f;
}
```

El código anterior es una especie de implementación ingenua que almacena un `float` en un número de 32 bits. El bit más alto (31) se utiliza para almacenar el signo del número («1» significa negativo), y los siete bits siguientes (30-16) se utilizan para almacenar la parte entera del `float`. Por último, los bits restantes (15-0) se utilizan para almacenar la parte fraccionaria del número.

Su uso es bastante sencillo:

```
#include <stdio.h>

int main(void)
{
    float f = 3.1415926, f2;
    uint32_t netf;

    netf = htonf(f);  // convertir a formato de "red"
    f2 = ntohf(netf); // volver a convertir en prueba

    printf("Original: %f\n", f);        // 3.141593
    printf(" Network: 0x%08X\n", netf); // 0x0003243F
    printf("Unpacked: %f\n", f2);       // 3.141586

    return 0;
}
```

Lo bueno es que es pequeño, sencillo y rápido. En el lado negativo, no es un uso eficiente del espacio y el rango está severamente restringido - ¡intenta almacenar un número mayor que 32767 y no estará muy contento! También puedes ver en el ejemplo anterior que los últimos decimales no se conservan correctamente.

¿Qué podemos hacer en su lugar? Bueno, *El* estándar para almacenar números en coma flotante se conoce como IEEE-754[7]. La mayoría de los ordenadores utilizan este formato internamente para realizar opera-

[7] https://en.wikipedia.org/wiki/IEEE_754

ciones matemáticas en coma flotante, por lo que en esos casos, estrictamente hablando, no sería necesario realizar la conversión. Pero si quieres que tu código fuente sea portable, esa es una suposición que no puedes hacer necesariamente. (Por otro lado, si quieres que las cosas sean rápidas, ¡deberías optimizar esto en plataformas que no necesiten hacerlo! Eso es lo que hacen `htons()` y similares).

Aquí hay algo de código que codifica flotantes y dobles en formato IEEE-754[8]. (Mayormente—no codifica NaN o Infinito, pero podría ser modificado para hacerlo).

```
#define pack754_32(f) (pack754((f), 32, 8))
#define pack754_64(f) (pack754((f), 64, 11))
#define unpack754_32(i) (unpack754((i), 32, 8))
#define unpack754_64(i) (unpack754((i), 64, 11))

uint64_t pack754(long double f, unsigned bits, unsigned expbits)
{
    long double fnorm;
    int shift;
    long long sign, exp, significand;
    unsigned significandbits = bits - expbits - 1; // -1 for sign bit

    if (f == 0.0) return 0; // quita este caso especial de en medio

    // comprobar signo e iniciar normalización
    if (f < 0) { sign = 1; fnorm = -f; }
    else { sign = 0; fnorm = f; }

    // obtener la forma normalizada de f y seguir el exponente
    shift = 0;
    while(fnorm >= 2.0) { fnorm /= 2.0; shift++; }
    while(fnorm < 1.0) { fnorm *= 2.0; shift--; }
    fnorm = fnorm - 1.0;

    // calcular la forma binaria (no-float) de los datos del significando
    significand = fnorm * ((1LL<<significandbits) + 0.5f);

    // obtener el exponente sesgado
    exp = shift + ((1<<(expbits-1)) - 1); // shift + bias

    // devuelve la respuesta final
    return (sign<<(bits-1)) | (exp<<(bits-expbits-1)) | significand;
}

long double unpack754(uint64_t i, unsigned bits, unsigned expbits)
{
    long double result;
    long long shift;
    unsigned bias;
    unsigned significandbits = bits - expbits - 1; // -1 for sign bit

    if (i == 0) return 0.0;

    // extraer el significante
    result = (i&((1LL<<significandbits)-1)); // mask
    result /= (1LL<<significandbits); // volver a convertir a float
    result += 1.0f; // volver a añadir el
```

[8] https://beej.us/guide/bgnet/source/examples/ieee754.c

```
    // tratar el exponente
    bias = (1<<(expbits-1)) - 1;
    shift = ((i>>significandbits)&((1LL<<expbits)-1)) - bias;
    while(shift > 0) { result *= 2.0; shift--; }
    while(shift < 0) { result /= 2.0; shift++; }

    // fírmelo
    result *= (i>>(bits-1))&1? -1.0: 1.0;

    return result;
}
```

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

Aquí tienes un ejemplo de uso:

```
#include <stdio.h>
#include <stdint.h> // define los tipos uintN_t
#include <inttypes.h> // define las macros PRIx

int main(void)
{
    float f = 3.1415926, f2;
    double d = 3.14159265358979323, d2;
    uint32_t fi;
    uint64_t di;

    fi = pack754_32(f);
    f2 = unpack754_32(fi);

    di = pack754_64(d);
    d2 = unpack754_64(di);

    printf("float before : %.7f\n", f);
    printf("float encoded: 0x%08" PRIx32 "\n", fi);
    printf("float after  : %.7f\n\n", f2);

    printf("double before : %.20lf\n", d);
    printf("double encoded: 0x%016" PRIx64 "\n", di);
    printf("double after  : %.20lf\n", d2);

    return 0;
}
```

El código anterior produce este resultado:

```
float before : 3.1415925
float encoded: 0x40490FDA
float after  : 3.1415925

double before : 3.14159265358979311600
double encoded: 0x400921FB54442D18
double after  : 3.14159265358979311600
```

Desgraciadamente para ti, el compilador es libre de poner relleno por todas partes en una `struct`, lo que

significa que no puedes enviarla por cable en un solo trozo. (¿No estás harto de oír «no se puede hacer esto», «no se puede hacer aquello»? Lo siento. Citando a un amigo: «Siempre que algo va mal, culpo a Microsoft». Puede que ésta no sea culpa de Microsoft, hay que reconocerlo, pero la afirmación de mi amigo es completamente cierta).

Volviendo al tema: la mejor forma de enviar la `struct` por cable es empaquetar cada campo de forma independiente y luego desempaquetarlos en la `struct` cuando llegan al otro lado.

Eso es mucho trabajo, es lo que estás pensando. Pues sí. Una cosa que puedes hacer es escribir una función de ayuda que te 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. Las enlazaría, pero aparentemente esas funciones no están en línea con el resto del código fuente del libro.

(La práctica de la programación es una lectura excelente. Zeus salva un gatito cada vez que lo recomiendo).

En este punto, voy a dejar un puntero a un Protocol Buffers implementation in C[10] que nunca he usado, pero que parece completamente respetable. Los programadores de Python y Perl querrán echar un vistazo a las funciones `pack()` y `unpack()` de sus lenguajes para conseguir lo mismo. Y Java tiene una gran interfaz Serializable que puede usarse de forma similar.

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

(Este código hace referencia a las funciones `pack754()`, arriba. Las funciones `packi*()` operan como la familiar familia `htons()`, excepto que empaquetan en un array `char` en lugar de otro entero).

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

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

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

/*
** packi64() -- almacenar un int de 64 bits en un búfer char (como htonl())
*/
void packi64(unsigned char *buf, unsigned long long int i)
{
    *buf++ = i>>56; *buf++ = i>>48;
```

[9] https://beej.us/guide/url/tpop
[10] https://github.com/protobuf-c/protobuf-c
[11] https://beej.us/guide/bgnet/source/examples/pack2.c

```
    *buf++ = i>>40; *buf++ = i>>32;
    *buf++ = i>>24; *buf++ = i>>16;
    *buf++ = i>>8;  *buf++ = i;
}

/*
** unpacki16() -- descomprimir un int de 16 bits de un búfer char (como ntohs())
*/
int unpacki16(unsigned char *buf)
{
    unsigned int i2 = ((unsigned int)buf[0]<<8) | buf[1];
    int i;

    // change unsigned numbers to signed
    if (i2 <= 0x7fffu) { i = i2; }
    else { i = -1 - (unsigned int)(0xffffu - i2); }

    return i;
}

/*
** unpacku16() -- desempaqueta un unsigned de 16 bits de un buffer char (como ntohs())
*/
unsigned int unpacku16(unsigned char *buf)
{
    return ((unsigned int)buf[0]<<8) | buf[1];
}

/*
** unpacki32() -- descomprimir un int de 32 bits de un búfer char (como ntohl())
*/
long int unpacki32(unsigned char *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 i;

    // cambiar números sin signo a con signo
    if (i2 <= 0x7fffffffu) { i = i2; }
    else { i = -1 - (long int)(0xffffffffu - i2); }

    return i;
}

/*
** unpacku32() -- desempaqueta un unsigned de 32 bits de un buffer char (como ntohl())
*/
unsigned long int unpacku32(unsigned char *buf)
{
    return ((unsigned long int)buf[0]<<24) |
           ((unsigned long int)buf[1]<<16) |
           ((unsigned long int)buf[2]<<8)  |
           buf[3];
}
```

```
/*
** unpacki64() -- descomprime un int de 64 bits de un buffer char (como ntohl())
*/
long long int unpacki64(unsigned char *buf)
{
    unsigned long long int i2 = ((unsigned long long int)buf[0]<<56) |
                                ((unsigned long long int)buf[1]<<48) |
                                ((unsigned long long int)buf[2]<<40) |
                                ((unsigned long long int)buf[3]<<32) |
                                ((unsigned long long int)buf[4]<<24) |
                                ((unsigned long long int)buf[5]<<16) |
                                ((unsigned long long int)buf[6]<<8)  |
                                buf[7];
    long long int i;

    // cambiar números sin signo a con signo
    if (i2 <= 0x7fffffffffffffffu) { i = i2; }
    else { i = -1 -(long long int)(0xffffffffffffffffu - i2); }

    return i;
}

/*
** unpacku64() -- desempaqueta un unsigned de 64 bits de un buffer char (como ntohl())
*/
unsigned long long int unpacku64(unsigned char *buf)
{
    return ((unsigned long long int)buf[0]<<56) |
           ((unsigned long long int)buf[1]<<48) |
           ((unsigned long long int)buf[2]<<40) |
           ((unsigned long long int)buf[3]<<32) |
           ((unsigned long long int)buf[4]<<24) |
           ((unsigned long long int)buf[5]<<16) |
           ((unsigned long long int)buf[6]<<8)  |
           buf[7];
}

/*
** pack() -- almacena los datos dictados por la cadena de formato en el buffer
**
**   bits |signo   sin-signo   float   string
**   -----+----------------------------------
**      8 |   c        C
**     16 |   h        H         f
**     32 |   l        L         d
**     64 |   q        Q         g
**      - |                               s
**
** (La longitud de 16 bits sin signo se añade automáticamente a las cadenas)
*/

unsigned int pack(unsigned char *buf, char *format, ...)
{
    va_list ap;

    signed char c;              // 8-bit
    unsigned char C;
```

```
    int h;                      // 16-bit
    unsigned int H;

    long int l;                 // 32-bit
    unsigned long int L;

    long long int q;            // 64-bit
    unsigned long long int Q;

    float f;                    // floats
    double d;
    long double g;
    unsigned long long int fhold;

    char *s;                    // strings
    unsigned int len;

    unsigned int size = 0;

    va_start(ap, format);

    for(; *format != '\0'; format++) {
        switch(*format) {
        case 'c': // 8-bit
            size += 1;
            c = (signed char)va_arg(ap, int); // promoted
            *buf++ = c;
            break;

        case 'C': // 8-bit unsigned
            size += 1;
            C = (unsigned char)va_arg(ap, unsigned int); // promoted
            *buf++ = C;
            break;

        case 'h': // 16-bit
            size += 2;
            h = va_arg(ap, int);
            packi16(buf, h);
            buf += 2;
            break;

        case 'H': // 16-bit unsigned
            size += 2;
            H = va_arg(ap, unsigned int);
            packi16(buf, H);
            buf += 2;
            break;

        case 'l': // 32-bit
            size += 4;
            l = va_arg(ap, long int);
            packi32(buf, l);
            buf += 4;
            break;
```

```
        case 'L': // 32-bit unsigned
            size += 4;
            L = va_arg(ap, unsigned long int);
            packi32(buf, L);
            buf += 4;
            break;

        case 'q': // 64-bit
            size += 8;
            q = va_arg(ap, long long int);
            packi64(buf, q);
            buf += 8;
            break;

        case 'Q': // 64-bit unsigned
            size += 8;
            Q = va_arg(ap, unsigned long long int);
            packi64(buf, Q);
            buf += 8;
            break;

        case 'f': // float-16
            size += 2;
            f = (float)va_arg(ap, double); // promoted
            fhold = pack754_16(f); // convert to IEEE 754
            packi16(buf, fhold);
            buf += 2;
            break;

        case 'd': // float-32
            size += 4;
            d = va_arg(ap, double);
            fhold = pack754_32(d); // convert to IEEE 754
            packi32(buf, fhold);
            buf += 4;
            break;

        case 'g': // float-64
            size += 8;
            g = va_arg(ap, long double);
            fhold = pack754_64(g); // convert to IEEE 754
            packi64(buf, fhold);
            buf += 8;
            break;

        case 's': // string
            s = va_arg(ap, char*);
            len = strlen(s);
            size += len + 2;
            packi16(buf, len);
            buf += 2;
            memcpy(buf, s, len);
            buf += len;
            break;
        }
    }
```

```
    va_end(ap);

    return size;
}

/*
** unpack() -- desempaqueta los datos dictados por la cadena de formato en el buffer
**
**   bits |signed   unsigned   float   string
**   -----+----------------------------------
**      8 |   c        C
**     16 |   h        H         f
**     32 |   l        L         d
**     64 |   q        Q         g
**      - |                              s
**
** (la cadena se extrae en función de su longitud almacenada, pero 's' se puede
** con una longitud máxima)
*/
void unpack(unsigned char *buf, char *format, ...)
{
    va_list ap;

    signed char *c;              // 8-bit
    unsigned char *C;

    int *h;                      // 16-bit
    unsigned int *H;

    long int *l;                 // 32-bit
    unsigned long int *L;

    long long int *q;            // 64-bit
    unsigned long long int *Q;

    float *f;                    // floats
    double *d;
    long double *g;
    unsigned long long int fhold;

    char *s;
    unsigned int len, maxstrlen=0, count;

    va_start(ap, format);

    for(; *format != '\0'; format++) {
        switch(*format) {
        case 'c': // 8-bit
            c = va_arg(ap, signed char*);
            if (*buf <= 0x7f) { *c = *buf;} // re-sign
            else { *c = -1 - (unsigned char)(0xffu - *buf); }
            buf++;
            break;

        case 'C': // 8-bit unsigned
            C = va_arg(ap, unsigned char*);
            *C = *buf++;
```

```
            break;

        case 'h': // 16-bit
            h = va_arg(ap, int*);
            *h = unpacki16(buf);
            buf += 2;
            break;

        case 'H': // 16-bit unsigned
            H = va_arg(ap, unsigned int*);
            *H = unpacku16(buf);
            buf += 2;
            break;

        case 'l': // 32-bit
            l = va_arg(ap, long int*);
            *l = unpacki32(buf);
            buf += 4;
            break;

        case 'L': // 32-bit unsigned
            L = va_arg(ap, unsigned long int*);
            *L = unpacku32(buf);
            buf += 4;
            break;

        case 'q': // 64-bit
            q = va_arg(ap, long long int*);
            *q = unpacki64(buf);
            buf += 8;
            break;

        case 'Q': // 64-bit unsigned
            Q = va_arg(ap, unsigned long long int*);
            *Q = unpacku64(buf);
            buf += 8;
            break;

        case 'f': // float
            f = va_arg(ap, float*);
            fhold = unpacku16(buf);
            *f = unpack754_16(fhold);
            buf += 2;
            break;

        case 'd': // float-32
            d = va_arg(ap, double*);
            fhold = unpacku32(buf);
            *d = unpack754_32(fhold);
            buf += 4;
            break;

        case 'g': // float-64
            g = va_arg(ap, long double*);
            fhold = unpacku64(buf);
            *g = unpack754_64(fhold);
            buf += 8;
```

```
            break;

        case 's': // string
            s = va_arg(ap, char*);
            len = unpacku16(buf);
            buf += 2;
            if (maxstrlen > 0 && len >= maxstrlen) count = maxstrlen - 1;
            else count = len;
            memcpy(s, buf, count);
            s[count] = '\0';
            buf += len;
            break;

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

        if (!isdigit(*format)) 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 cuando llame a `unpack()` con un argumento de cadena (especificador de formato `s`), es aconsejable poner un contador de longitud máxima delante de él para evitar un desbordamiento del búfer, por ejemplo, `96s`. Tenga cuidado cuando desempaquete datos que reciba a través de la red: ¡un usuario malintencionado podría enviar paquetes mal construidos con la intención de atacar su sistema!

```
#include <stdio.h>

// varios bits para tipos de coma flotante--
// varía según la arquitectura
typedef float float32_t;
typedef double float64_t;

int main(void)
{
    unsigned char buf[1024];
    int8_t magic;
    int16_t monkeycount;
    int32_t altitude;
    float32_t absurdityfactor;
    char *s = "Great unmitigated Zot! You've found the Runestaff!";
    char s2[96];
    int16_t packetsize, ps2;

    packetsize = pack(buf, "chhlsf", (int8_t)'B', (int16_t)0, (int16_t)37,
            (int32_t)-5, s, (float32_t)-3490.6677);
   packi16(buf+1, packetsize); // almacena el tamaño del paquete en packet for kicks

    printf("packet is %" PRId32 " bytes\n", packetsize);
```

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

```
    unpack(buf, "chhl96sf", &magic, &ps2, &monkeycount, &altitude, s2,
        &absurdityfactor);

    printf("'%c' %" PRId32" %" PRId16 " %" PRId32
            " \"%s\" %f\n", magic, ps2, monkeycount,
            altitude, s2, absurdityfactor);

    return 0;
}
```

Tanto si desarrollas tu propio código como si utilizas el de otro, 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.

¿Cuál es el mejor formato para empaquetar los datos? 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 coma flotante, tipos enteros, matrices, datos brutos, etc. Te sugiero que lo sigas si vas a enrollar los datos tú mismo. Pero no estás obligado a hacerlo. La policía de los paquetes no está delante de tu puerta. Al menos, no creo que lo estén.

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

¿Qué significa realmente encapsular datos? En el caso más sencillo, se trata de incluir una cabecera con información identificativa o la longitud del paquete, o ambas cosas.

¿Qué aspecto debe tener la cabecera? Bueno, son sólo algunos datos binarios que representan lo que consideres necesario para completar tu proyecto.

Vaya. Eso es vago.

Bien. Por ejemplo, digamos que tienes un programa de chat multiusuario que usa `SOCK_STREAM`s. Cuando un usuario teclea (dice) algo, hay que transmitir dos informaciones al servidor: qué se ha dicho y quién lo ha dicho.

¿Hasta aquí todo bien? "¿Cuál es el problema?", te preguntarás.

El problema es que los mensajes pueden ser de distinta longitud. Una persona llamada "tom" puede decir: "Hi", y otra persona llamada "Benjamin" puede decir: "Hey guys what is up?".

Así que `send()` todas estas cosas a los clientes a medida que van llegando. Tu flujo de datos salientes se parece a esto:

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

Y así sucesivamente. ¿Cómo sabe el cliente cuándo empieza un mensaje y termina otro? Podrías, si quisieras, hacer que todos los mensajes tuvieran la misma longitud y simplemente llamar a la función que implementamos, arriba. Pero eso desperdicia ancho de banda. No queremos `send()` (enviar) 1024 bytes sólo para que «tom» pueda decir "Hi".

Así que *encapsulamos* los datos en una pequeña estructura de cabecera y paquete. Tanto el cliente como el servidor saben cómo empaquetar y desempaquetar (a veces denominado «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.

En este caso, vamos a suponer que el nombre de usuario tiene una longitud fija de 8 caracteres, rellenados con `'\0'`. Y supongamos que los datos son de longitud variable, hasta un máximo de 128 caracteres. Veamos un ejemplo de estructura de paquete que podríamos utilizar en esta situación:

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

1. `len` (1 byte, sin signo)—La longitud total del paquete, contando el nombre de usuario de 8 bytes y los datos del chat.
2. `name` (8 bytes)—El nombre del usuario, relleno NUL si es necesario.
3. (*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 y 128 bytes para los campos? Los saqué de la nada, suponiendo que serían suficientemente largos. Sin embargo, tal vez 8 bytes sea demasiado restrictivo para tus necesidades y puedas 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 consistiría en la siguiente información (en hexadecimal y ASCII):

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

Y la segunda es similar:

```
   18     42 65 6E 6A 61 6D 69 6E      48 65 79 20 67 75 79 73 20 77 ...
(length)  B  e  n  j  a  m  i  n       H  e  y     g  u  y  s     w  ...
```

(Por supuesto, la longitud se almacena en orden de bytes de red. En este caso, es sólo un byte, así que no importa, pero en general querrás que todos tus enteros binarios se almacenen en Orden de Bytes de Red en tus paquetes)

Cuando estés enviando estos datos, deberías estar seguro y usar un comando similar a `sendall()`, para que sepas que todos los datos han sido enviados, incluso si se necesitan múltiples llamadas a `send()` para obtenerlos todos

Del mismo modo, cuando reciba estos datos, tendrá que hacer un poco más de trabajo. 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 sea recibido completamente.

¿Cómo? Bueno, sabemos el número de bytes que necesitamos recibir en total para que el paquete esté completo, ya que ese número aparece 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 sabe que cada paquete comienza con una longitud, puede llamar a `recv()` sólo para obtener la longitud del paquete. Entonces, una vez que tenga eso, puede llamarla de nuevo 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 sólo necesita un buffer lo suficientemente grande para un paquete, mientras que la desventaja es que necesita 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. Entonces, lo que obtenga, péguelo en la parte posterior de un buffer, y finalmente compruebe si el paquete está completo. Por supuesto, puede que recibas algo del siguiente paquete, así que necesitarás tener espacio para eso.

Lo que puedes hacer es declarar un array lo suficientemente grande para dos paquetes. Este es tu array de trabajo donde reconstruirás los paquetes según vayan llegando.

Cada vez que `recv()` datos, los añadirá al búfer de trabajo y comprobará si el paquete está completo. Esto es, el número de bytes en el buffer es mayor o igual que la longitud especificada en la cabecera (+1, porque la longitud en la cabecera no incluye el byte para la longitud en sí). Si el número de bytes en el búfer es inferior a 1, el paquete no está completo, obviamente. Tienes que hacer un caso especial para esto, ya que el primer byte es basura y no puedes confiar en él para la longitud correcta del paquete

Una vez que el paquete está completo, puedes hacer con él lo que quieras. Utilízalo y elimínalo de tu búfer de trabajo.

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

Como conoces la longitud del primer paquete por la cabecera, y has estado llevando la cuenta del número de bytes en el búfer de trabajo, puedes restar y calcular cuántos de los bytes del búfer de trabajo pertenecen al segundo paquete (incompleto). Cuando haya manejado el primero, puede borrarlo del búfer de trabajo y mover el segundo paquete parcial al frente del búfer para que esté listo para el siguiente `recv()`.

(Algunos de los lectores notarán que mover el segundo paquete parcial al principio del buffer de trabajo toma tiempo, y el programa puede ser codificado para no requerir esto usando un buffer circular. Desafortunadamente para el resto de ustedes, una discusión sobre buffers circulares está más allá del alcance de este artículo. Si todavía tienes curiosidad, coge un libro de estructuras de datos y sigue a partir de ahí).

Nunca dije que fuera fácil. Vale, sí dije que era fácil. Y lo es; sólo necesitas práctica y muy pronto te saldrá de forma natural. Por Excalibur ¡lo juro!

7.7 Paquetes de difusión… ¡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 apropiada, enviar datos a múltiples hosts *al mismo tiempo*!

Con UDP (sólo UDP, no TCP) e IPv4 estándar, esto se hace a través de un mecanismo llamado *broadcasting*. Con IPv6, la difusión no está soportada, y tienes que recurrir a la técnica a menudo superior de *multicasting*, que, por desgracia, no voy a discutir en este momento. Pero basta de futuro, estamos atrapados en el presente de 32 bits.

Pero, ¡espera! No puedes simplemente salir corriendo y empezar a emitir como si nada; Tienes que establecer la opción de socket `SO_BROADCAST` antes de poder enviar un paquete broadcast a la red. ¡Es como una de esas pequeñas tapas de plástico que ponen sobre el interruptor de lanzamiento de misiles! ¡Eso es todo el poder que tienes en tus manos!

Pero en serio, hay 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 averigua a qué puerto están destinados los datos. Y entonces entrega los datos o los descarta. En cualquier caso, es mucho trabajo para cada máquina que recibe el paquete de difusión, y puesto que son todas ellas en la red local, podrían ser muchas máquinas haciendo mucho trabajo innecesario. Cuando el juego Doom salió por primera vez, esta era una queja sobre su código de red.

Ahora, hay más de una manera de despellejar a un gato… espera un minuto. ¿De verdad hay más de una forma de despellejar a un gato? ¿Qué clase de expresión es esa? Uh, y del mismo modo, hay más de una manera de enviar un paquete de difusión. Así que, para llegar a la carne y las patatas de todo el asunto: ¿cómo se especifica la dirección de destino de un mensaje de difusión? Hay dos formas comunes:

1. Enviar 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 la parte de host de la dirección. Por ejemplo, en casa mi red es `192.168.1.0`, mi máscara de red es `255.255.255.0`, así que el último byte de la dirección es mi número de host (porque los tres primeros bytes, según la máscara de red, son el número de red). Así que mi dirección de difusión es `192.168.1.255`. En Unix, el comando `ifconfig` le dará todos estos datos. datos. (Si tienes curiosidad, la lógica bit a bit para obtener tu dirección de transmisión es es `número_de_red` (network_number) O (NO `máscara_de_red`). Puedes 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 descartan, un pitufo al azar podría empezar a inundar su LAN con tráfico de difusión).

2. Envía los datos a la dirección de difusión «global». Esto es `255.255.255.255`, también conocido como `INADDR_BROADCAST`. Muchas máquinas automáticamente bitwise AND esto con su número de red para convertirlo en una dirección de difusión de red, pero algunos no. Esto varía. Irónicamente, los routers no reenvían este tipo de paquetes de difusión fuera de la red local.

Entonces, ¿qué pasa si intentas enviar datos en la dirección de difusión sin configurar primero la opción de socket `SO_BROADCAST`? Bueno, encendamos el viejo `talker` and `listener` y veamos que pasa.

```
$ talker 192.168.1.2 foo
sent 3 bytes to 192.168.1.2
$ talker 192.168.1.255 foo
sendto: Permission denied
$ talker 255.255.255.255 foo
sendto: Permission denied
```

Sí, no está nada contento… porque no hemos puesto la opción de socket `SO_BROADCAST`. Hazlo, ¡y ahora puedes `sendto()` donde quieras!

De hecho, esa es la *única diferencia* entre una aplicación UDP que puede emitir y una que no. Así que tomemos la vieja aplicación `talker` y añadamos 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 que
** que este puede emitir
*/

#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 SERVERPORT 4950 // el puerto al que se conectarán los usuarios

int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in their_addr; // información de la dirección del conector
    struct hostent *he;
    int numbytes;
    int broadcast = 1;
    //char broadcast = '1'; // si eso no funciona, prueba esto

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

    if ((he=gethostbyname(argv[1])) == NULL) {  // get the host info
        perror("gethostbyname");
        exit(1);
    }

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

[14] https://beej.us/guide/bgnet/source/examples/broadcaster.c

```
        exit(1);
    }

    // esta llamada es la que permite enviar paquetes de difusión:
    if (setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &broadcast,
        sizeof broadcast) == -1) {
        perror("setsockopt (SO_BROADCAST)");
        exit(1);
    }

    their_addr.sin_family = AF_INET;     // orden de bytes del host
    their_addr.sin_port = htons(SERVERPORT); // short, orden de bytes de red
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(their_addr.sin_zero, '\0', sizeof their_addr.sin_zero);

    if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0,
             (struct sockaddr *)&their_addr, sizeof their_addr)) == -1) {
        perror("sendto");
        exit(1);
    }

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

    close(sockfd);

    return 0;
}
```

¿Qué diferencia hay entre esto y una situación cliente/servidor UDP «normal»? En nada. (Con la excepción de que al cliente se le permite enviar paquetes de difusión en este caso). Como tal, siga adelante y ejecute el viejo programa UDP `listener` en una ventana, y `broadcaster` en otra. Ahora debería ser capaz de hacer todos los envíos que fallaron, más arriba.

```
$ broadcaster 192.168.1.2 foo
sent 3 bytes to 192.168.1.2
$ broadcaster 192.168.1.255 foo
sent 3 bytes to 192.168.1.255
$ broadcaster 255.255.255.255 foo
sent 3 bytes to 255.255.255.255
```

Y deberías ver a `listener` respondiendo que recibió los paquetes. (Si `listener` no responde, podría ser porque está enlazado a una dirección IPv6. Prueba a cambiar `AF_INET6` en `listener.c` por `AF_INET` para forzar IPv4).

Bueno, eso es emocionante. Pero ahora enciende `listener` en otra máquina a tu lado en la misma red para tener dos copias, una en cada máquina, y ejecuta `broadcaster` de nuevo con tu dirección de emisión… ¡Ambos `listener` reciben el paquete aunque sólo hayas llamado a `sendto()` una vez! ¡Genial!

Si el `listener` recibe los datos que le envías directamente, pero no los datos en la dirección de broadcast, puede ser que tengas un firewall en tu máquina local que esté bloqueando los paquetes. (Sí, Pat y Bapper, gracias por daros cuenta antes que yo de que por eso no funcionaba mi código de ejemplo. Os dije que os mencionaría en la guía, y aquí estáis. Así que *nyah*).

De nuevo, ten cuidado con los paquetes broadcast. Dado que cada máquina de la LAN se verá forzada a tratar con el paquete tanto si lo `recvfrom()`s como si no, puede suponer una gran carga para toda la red informática. Definitivamente deben usarse con moderación y apropiadamente.

Chapter 8

Preguntas frecuentes

¿Dónde puedo conseguir esos archivos de cabecera?

Si no los tienes ya en tu sistema, probablemente no los necesites. Consulta el manual de tu plataforma. Si estás construyendo para Windows, sólo necesitas `#include <winsock.h>`.

¿Qué hago cuando `bind()` informa "Address already in use"?

Tienes que usar `setsockopt()` con la macro `SO_REUSEADDR` en el socket de escucha. Consulta la section on `bind()` y la section on `select()` para ver un ejemplo.

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

Usa el comando `netstat`. Revisa la página `man` para más detalles, pero deberías obtener una buena salida simplemente tecleando:

```
$ netstat
```

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

¿Cómo puedo ver la tabla de enrutamiento?

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

¿Cómo puedo ejecutar los programas cliente y servidor si sólo tengo un ordenador? ¿No necesito una red para escribir programas de red?.

Afortunadamente para ti, prácticamente todas las máquinas implementan un "dispositivo" de red loopback que se sitúa en el kernel y simula ser una tarjeta de red. (Es la interfaz que aparece como «`lo`» en la tabla de enrutamiento).

Imagina que estás conectado a una máquina llamada "`goat`". Ejecuta el cliente en una ventana y el servidor en otra. O inicia el servidor en segundo plano ("`server &`"") y ejecuta el cliente en la misma ventana. El resultado del dispositivo loopback es que puedes ejecutar `client goat` o] utilidad «ping»? ¿Qué es ¿ICMP? ¿Dónde puedo encontrar más información sobre raw sockets y `SOCK_RAW`?**

Todas tus preguntas sobre raw sockets serán respondidas en W. Richard Stevens' UNIX Network Programming books. También, busque en el subdirectorio `ping/` en el código fuente de Programación de Redes UNIX de Stevens, disponible en línea[1].

¿Cómo puedo cambiar o acortar el tiempo de espera de una llamada a `connect()`?

En lugar de darte exactamente la misma respuesta que te daría W. Richard Stevens, te remitiré a `lib/connect_nonb.c` en el código fuente de UNIX Network Programming[2].

[1] http://www.unpbook.com src.html
[2] http:www.unpbook.com/src.html

La esencia es que haces un descriptor de socket con `socket()`, lo configuras como no-bloqueo, llamas a `connect()`, y si todo va bien `connect()` devolverá `-1` inmediatamente y `errno` se configurará como `EINPROGRESS`. Entonces llama a `select()` con el tiempo de espera que quieras, pasando el descriptor del socket tanto en lectura como en escritura. Si no se agota el tiempo de espera, significa que la llamada a `connect()` se ha completado. En este punto, tendrás que usar `getsockopt()` con la opción `SO_ERROR` para obtener el valor de retorno de la llamada a `connect()`, que debería ser cero si no hubo error.

Finalmente, probablemente querrá volver a poner el socket en bloqueo antes de empezar a transferir datos sobre él.

Tenga en cuenta que esto tiene el beneficio añadido de permitir a su programa hacer algo más mientras se está conectando. Podrías, por ejemplo, establecer el tiempo de espera a algo bajo, como 500 ms, y actualizar un indicador en pantalla cada vez que se agote el tiempo de espera, luego llamar a `select()` de nuevo. Cuando hayas llamado a `select()` y se haya agotado el tiempo de espera, digamos, 20 veces, sabrás que es hora de abandonar la conexión.

Como he dicho, echa un vistazo a la fuente de Stevens para un ejemplo perfectamente excelente.

¿Cómo se construye para Windows? Primero, borra Windows e instala Linux o BSD. `};-)`. No, en realidad, sólo consulte la sección sobre la construcción para Windows en la introducción.

¿Cómo puedo compilar para Solaris/SunOS? Sigo recibiendo errores del enlazador cuando intento compilar?

Los errores del enlazador ocurren porque las cajas Sun no compilan automáticamente en las librerías de socket. Consulte la sección sobre compilación 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` puesto a `EINTR`. Cuando configure un manejador de señales con `sigaction()`, puede establecer la bandera `SA_RESTART`, que se supone que reinicia la llamada al sistema después de haber sido interrumpida.

Naturalmente, esto no siempre funciona.

Mi solución favorita para esto implica una `goto`. Sabes que esto irrita mucho a tus profesores, así que ¡hazlo! así que ¡adelante!

```
select_restart:
if ((err = select(fdmax+1, &readfds, NULL, NULL, NULL)) == -1) {
    if (errno == EINTR) {
        // alguna señal acaba de interrumpirnos, así que reinicia
        goto select_restart;
    }
    // maneja el error real aquí:
    perror("select");
}
```

Claro, no *necesitas* usar `goto` en este caso; puedes usar otras estructuras para controlarlo. Pero creo que la sentencia `goto` es realmente más limpia.

¿Cómo puedo implementar un tiempo de espera en una llamada a `recv()`?

Usa `select()`! Te permite especificar un parámetro de tiempo de espera para los descriptores de socket de los que quieres leer. O bien, puede envolver toda la funcionalidad en una sola función, como esta:

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

int recvtimeout(int s, char *buf, int len, int timeout)
```

```
{
    fd_set fds;
    int n;
    struct timeval tv;

    // configurar el conjunto de descriptores de fichero
    FD_ZERO(&fds);
    FD_SET(s, &fds);

    // configurar la estructura timeval para el tiempo de espera
    tv.tv_sec = timeout;
    tv.tv_usec = 0;

    // espera hasta que se agote el tiempo o se reciban datos
    n = select(s+1, &fds, NULL, NULL, &tv);
    if (n == 0) return -2; // timeout!
    if (n == -1) return -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); // 10 second timeout

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
}
.
.
.
```

Notice that `recvtimeout()` returns `-2` in case of a timeout. Why not return `0`? Well, if you recall, a return value of `0` on a call to `recv()` means that the remote side closed the connection. So that return value is already spoken for, and `-1` means "error", so I chose `-2` as my timeout indicator.

How do I encrypt or compress the data before sending it through the socket?

One easy way to do encryption is to use SSL (secure sockets layer), but that's beyond the scope of this guide. (Check out the OpenSSL project[3] for more info.)

But assuming you want to plug in or implement your own compressor or encryption system, it's just a matter of thinking of your data as running through a sequence of steps between both ends. Each step changes the data in some way.

1. server reads data from file (or wherever)
2. server encrypts/compresses data (you add this part)
3. server `send()`s encrypted data

Now the other way around:

[3] https://www.openssl.org/

1. client `recv()`s encrypted data
2. client decrypts/decompresses data (you add this part)
3. client writes data to file (or wherever)

If you're going to compress and encrypt, just remember to compress first. `:-)`

Just as long as the client properly undoes what the server does, the data will be fine in the end no matter how many intermediate steps you add.

So all you need to do to use my code is to find the place between where the data is read and the data is sent (using `send()`) over the network, and stick some code in there that does the encryption.

What is this "`PF_INET`" I keep seeing? Is it related to `AF_INET`?

Yes, yes it is. See the section on `socket()` for details.

How can I write a server that accepts shell commands from a client and executes them?

For simplicity, lets say the client `connect()`s, `send()`s, and `close()`s the connection (that is, there are no subsequent system calls without the client connecting again).

The process the client follows is this:

1. `connect()` to server
2. `send("/sbin/ls > /tmp/client.out")`
3. `close()` the connection

Meanwhile, the server is handling the data and executing it:

1. `accept()` the connection from the client
2. `recv(str)` the command string
3. `close()` the connection
4. `system(str)` to run the command

Beware! Having the server execute what the client says is like giving remote shell access and people can do things to your account when they connect to the server. For instance, in the above example, what if the client sends "`rm -rf ~`"? It deletes everything in your account, that's what!

So you get wise, and you prevent the client from using any except for a couple utilities that you know are safe, like the `foobar` utility:

```
if (!strncmp(str, "foobar", 6)) {
    sprintf(sysstr, "%s > /tmp/server.out", str);
    system(sysstr);
}
```

But you're still unsafe, unfortunately: what if the client enters "`foobar; rm -rf ~`"? The safest thing to do is to write a little routine that puts an escape ("`\`") character in front of all non-alphanumeric characters (including spaces, if appropriate) in the arguments for the command.

As you can see, security is a pretty big issue when the server starts executing things the client sends.

I'm sending a slew of data, but when I `recv()`, it only receives 536 bytes or 1460 bytes at a time. But if I run it on my local machine, it receives all the data at the same time. What's going on?

You're hitting the MTU—the maximum size the physical medium can handle. On the local machine, you're using the loopback device which can handle 8K or more no problem. But on Ethernet, which can only handle 1500 bytes with a header, you hit that limit. Over a modem, with 576 MTU (again, with header), you hit the even lower limit.

You have to make sure all the data is being sent, first of all. (See the `sendall()` function implementation for details.) Once you're sure of that, then you need to call `recv()` in a loop until all your data is read.

Read the section Son of Data Encapsulation for details on receiving complete packets of data using multiple calls to `recv()`.

I'm on a Windows box and I don't have the `fork()` system call or any kind of `struct sigaction`. What to do?

If they're anywhere, they'll be in POSIX libraries that may have shipped with your compiler. Since I don't have a Windows box, I really can't tell you the answer, but I seem to remember that Microsoft has a POSIX compatibility layer and that's where `fork()` would be. (And maybe even `sigaction`.)

Search the help that came with VC++ for "fork" or "POSIX" and see if it gives you any clues.

If that doesn't work at all, ditch the `fork()`/`sigaction` stuff and replace it with the Win32 equivalent: `CreateProcess()`. I don't know how to use `CreateProcess()`—it takes a bazillion arguments, but it should be covered in the docs that came with VC++.

I'm behind a firewall—how do I let people outside the firewall know my IP address so they can connect to my machine?

Unfortunately, the purpose of a firewall is to prevent people outside the firewall from connecting to machines inside the firewall, so allowing them to do so is basically considered a breach of security.

This isn't to say that all is lost. For one thing, you can still often `connect()` through the firewall if it's doing some kind of masquerading or NAT or something like that. Just design your programs so that you're always the one initiating the connection, and you'll be fine.

If that's not satisfactory, you can ask your sysadmins to poke a hole in the firewall so that people can connect to you. The firewall can forward to you either through it's NAT software, or through a proxy or something like that.

Be aware that a hole in the firewall is nothing to be taken lightly. You have to make sure you don't give bad people access to the internal network; if you're a beginner, it's a lot harder to make software secure than you might imagine.

Don't make your sysadmin mad at me. `;-)`

How do I write a packet sniffer? How do I put my Ethernet interface into promiscuous mode?

For those not in the know, when a network card is in "promiscuous mode", it will forward ALL packets to the operating system, not just those that were addressed to this particular machine. (We're talking Ethernet-layer addresses here, not IP addresses–but since ethernet is lower-layer than IP, all IP addresses are effectively forwarded as well. See the section Low Level Nonsense and Network Theory for more info.)

This is the basis for how a packet sniffer works. It puts the interface into promiscuous mode, then the OS gets every single packet that goes by on the wire. You'll have a socket of some type that you can read this data from.

Unfortunately, the answer to the question varies depending on the platform, but if you Google for, for instance, "windows promiscuous ioctl" you'll probably get somewhere. For Linux, there's what looks like a useful Stack Overflow thread[4], as well.

How can I set a custom timeout value for a TCP or UDP socket?

It depends on your system. You might search the net for `SO_RCVTIMEO` and `SO_SNDTIMEO` (for use with `setsockopt()`) to see if your system supports such functionality.

The Linux man page suggests using `alarm()` or `setitimer()` as a substitute.

How can I tell which ports are available to use? Is there a list of "official" port numbers?

Usually this isn't an issue. If you're writing, say, a web server, then it's a good idea to use the well-known port 80 for your software. If you're writing just your own specialized server, then choose a port at random (but greater than 1023) and give it a try.

If the port is already in use, you'll get an "Address already in use" error when you try to `bind()`. Choose another port. (It's a good idea to allow the user of your software to specify an alternate port either with a config file or a command line switch.)

[4] https://stackoverflow.com/questions/21323023/

There is a list of official port numbers[5] maintained by the Internet Assigned Numbers Authority (IANA). Just because something (over 1023) is in that list doesn't mean you can't use the port. For instance, Id Software's DOOM uses the same port as "mdqs", whatever that is. All that matters is that no one else *on the same machine* is using that port when you want to use it.

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

Chapter 9

Man Pages

In the Unix world, there are a lot of manuals. They have little sections that describe individual functions that you have at your disposal.

Of course, `manual` would be too much of a thing to type. I mean, no one in the Unix world, including myself, likes to type that much. Indeed I could go on and on at great length about how much I prefer to be terse but instead I shall be brief and not bore you with long-winded diatribes about how utterly amazingly brief I prefer to be in virtually all circumstances in their entirety.

[Applause]

Thank you. What I am getting at is that these pages are called "man pages" in the Unix world, and I have included my own personal truncated variant here for your reading enjoyment. The thing is, many of these functions are way more general purpose than I'm letting on, but I'm only going to present the parts that are relevant for Internet Sockets Programming.

But wait! That's not all that's wrong with my man pages:

- They are incomplete and only show the basics from the guide.
- There are many more man pages than this in the real world.
- They are different than the ones on your system.
- The header files might be different for certain functions on your system.
- The function parameters might be different for certain functions on your system.

If you want the real information, check your local Unix man pages by typing `man whatever`, where "whatever" is something that you're incredibly interested in, such as "`accept`". (I'm sure Microsoft Visual Studio has something similar in their help section. But "man" is better because it is one byte more concise than "help". Unix wins again!)

So, if these are so flawed, why even include them at all in the Guide? Well, there are a few reasons, but the best are that (a) these versions are geared specifically toward network programming and are easier to digest than the real ones, and (b) these versions contain examples!

Oh! And speaking of the examples, I don't tend to put in all the error checking because it really increases the length of the code. But you should absolutely do error checking pretty much any time you make any of the system calls unless you're totally 100% sure it's not going to fail, and you should probably do it even then!

9.1 `accept()`

Accept an incoming connection on a listening socket

Synopsis

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

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

Description

Once you've gone through the trouble of getting a `SOCK_STREAM` socket and setting it up for incoming connections with `listen()`, then you call `accept()` to actually get yourself a new socket descriptor to use for subsequent communication with the newly connected client.

The old socket that you are using for listening is still there, and will be used for further `accept()` calls as they come in.

Parameter	Description
`s`	The `listen()`ing socket descriptor.
`addr`	This is filled in with the address of the site that's connecting to you.
`addrlen`	This is filled in with the `sizeof()` the structure returned in the `addr` parameter. You can safely ignore it if you assume you're getting a `struct sockaddr_in` back, which you know you are, because that's the type you passed in for `addr`.

`accept()` will normally block, and you can use `select()` to peek on the listening socket descriptor ahead of time to see if it's "ready to read". If so, then there's a new connection waiting to be `accept()`ed! Yay! Alternatively, you could set the `O_NONBLOCK` flag on the listening socket using `fcntl()`, and then it will never block, choosing instead to return `-1` with `errno` set to `EWOULDBLOCK`.

The socket descriptor returned by `accept()` is a bona fide socket descriptor, open and connected to the remote host. You have to `close()` it when you're done with it.

Return Value

`accept()` returns the newly connected socket descriptor, or `-1` on error, with `errno` set appropriately.

Example

```
struct sockaddr_storage their_addr;
socklen_t addr_size;
struct addrinfo hints, *res;
int sockfd, new_fd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, MYPORT, &hints, &res);

// make a socket, bind it, and listen on it:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);
listen(sockfd, BACKLOG);
```

```
// now accept an incoming connection:

addr_size = sizeof their_addr;
new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);

// ready to communicate on socket descriptor new_fd!
```

See Also

`socket()`, `getaddrinfo()`, `listen()`, `struct sockaddr_in`

9.2 `bind()`

Associate a socket with an IP address and port number

Synopsis

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

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

Description

When a remote machine wants to connect to your server program, it needs two pieces of information: the IP address and the port number. The `bind()` call allows you to do just that.

First, you call `getaddrinfo()` to load up a `struct sockaddr` with the destination address and port information. Then you call `socket()` to get a socket descriptor, and then you pass the socket and address into `bind()`, and the IP address and port are magically (using actual magic) bound to the socket!

If you don't know your IP address, or you know you only have one IP address on the machine, or you don't care which of the machine's IP addresses is used, you can simply pass the `AI_PASSIVE` flag in the `hints` parameter to `getaddrinfo()`. What this does is fill in the IP address part of the `struct sockaddr` with a special value that tells `bind()` that it should automatically fill in this host's IP address.

What what? What special value is loaded into the `struct sockaddr`'s IP address to cause it to auto-fill the address with the current host? I'll tell you, but keep in mind this is only if you're filling out the `struct sockaddr` by hand; if not, use the results from `getaddrinfo()`, as per above. In IPv4, the `sin_addr.s_addr` field of the `struct sockaddr_in` structure is set to `INADDR_ANY`. In IPv6, the `sin6_addr` field of the `struct sockaddr_in6` structure is assigned into from the global variable `in6addr_any`. Or, if you're declaring a new `struct in6_addr`, you can initialize it to `IN6ADDR_ANY_INIT`.

Lastly, the `addrlen` parameter should be set to `sizeof my_addr`.

Return Value

Returns zero on success, or `-1` on error (and `errno` will be set accordingly).

Example

```
// modern way of doing things with getaddrinfo()

struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

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

// make a socket:
// (you should actually walk the "res" linked list and error-check!)

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

// bind it to the port we passed in to getaddrinfo():

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

```
// example of packing a struct by hand, IPv4

struct sockaddr_in myaddr;
int s;

myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(3490);

// you can specify an IP address:
inet_pton(AF_INET, "63.161.169.137", &(myaddr.sin_addr));

// or you can let it automatically select one:
myaddr.sin_addr.s_addr = INADDR_ANY;

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&myaddr, sizeof myaddr);
```

See Also

`getaddrinfo()`, `socket()`, `struct sockaddr_in`, `struct in_addr`

9.3 `connect()`

Connect a socket to a server

Synopsis

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

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

Description

Once you've built a socket descriptor with the `socket()` call, you can `connect()` that socket to a remote server using the well-named `connect()` system call. All you need to do is pass it the socket descriptor and the address of the server you're interested in getting to know better. (Oh, and the length of the address, which is commonly passed to functions like this.)

Usually this information comes along as the result of a call to `getaddrinfo()`, but you can fill out your own `struct sockaddr` if you want to.

If you haven't yet called `bind()` on the socket descriptor, it is automatically bound to your IP address and a random local port. This is usually just fine with you if you're not a server, since you really don't care what your local port is; you only care what the remote port is so you can put it in the `serv_addr` parameter. You *can* call `bind()` if you really want your client socket to be on a specific IP address and port, but this is pretty rare.

Once the socket is `connect()`ed, you're free to `send()` and `recv()` data on it to your heart's content.

Special note: if you `connect()` a `SOCK_DGRAM` UDP socket to a remote host, you can use `send()` and `recv()` as well as `sendto()` and `recvfrom()`. If you want.

Return Value

Returns zero on success, or `-1` on error (and `errno` will be set accordingly).

Example

```
// connect to www.example.com port 80 (http)

struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;

// we could put "80" instead on "http" on the next line:
getaddrinfo("www.example.com", "http", &hints, &res);

// make a socket:

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

// connect it to the address and port we passed in to getaddrinfo():

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

See Also

`socket()`, `bind()`

9.4 `close()`

Close a socket descriptor

Synopsis

```
#include <unistd.h>

int close(int s);
```

Description

After you've finished using the socket for whatever demented scheme you have concocted and you don't want to `send()` or `recv()` or, indeed, do *anything else* at all with the socket, you can `close()` it, and it'll be freed up, never to be used again.

The remote side can tell if this happens one of two ways. One: if the remote side calls `recv()`, it will return `0`. Two: if the remote side calls `send()`, it'll receive a signal `SIGPIPE` and send() will return `-1` and `errno` will be set to `EPIPE`.

Windows users: the function you need to use is called `closesocket()`, not `close()`. If you try to use `close()` on a socket descriptor, it's possible Windows will get angry… And you wouldn't like it when it's angry.

Return Value

Returns zero on success, or `-1` on error (and `errno` will be set accordingly).

Example

```
s = socket(PF_INET, SOCK_DGRAM, 0);
.
.
.
// a whole lotta stuff...*BRRRONNNN!*
.
.
.
close(s);  // not much to it, really.
```

See Also

`socket()`, `shutdown()`

9.5 `getaddrinfo()`, `freeaddrinfo()`, `gai_strerror()`

Get information about a host name and/or service and load up a `struct sockaddr` with the result.

Synopsis

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

int getaddrinfo(const char *nodename, const char *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 (auto) or IPPROTO_TCP, IPPROTO_UDP

  socklen_t  ai_addrlen;     // length of ai_addr
  char   *ai_canonname;      // canonical name for nodename
  struct sockaddr  *ai_addr; // binary address
  struct addrinfo  *ai_next; // next structure in linked list
};
```

Description

`getaddrinfo()` is an excellent function that will return information on a particular host name (such as its IP address) and load up a `struct sockaddr` for you, taking care of the gritty details (like if it's IPv4 or IPv6). It replaces the old functions `gethostbyname()` and `getservbyname()`.The description, below, contains a lot of information that might be a little daunting, but actual usage is pretty simple. It might be worth it to check out the examples first.

The host name that you're interested in goes in the `nodename` parameter. The address can be either a host name, like "www.example.com", or an IPv4 or IPv6 address (passed as a string). This parameter can also be `NULL` if you're using the `AI_PASSIVE` flag (see below).

The `servname` parameter is basically the port number. It can be a port number (passed as a string, like "80"), or it can be a service name, like "http" or "tftp" or "smtp" or "pop", etc. Well-known service names can be found in the IANA Port List[1] or in your `/etc/services` file.

Lastly, for input parameters, we have `hints`. This is really where you get to define what the `getaddrinfo()` function is going to do. Zero the whole structure before use with `memset()`. Let's take a look at the fields you need to set up before use.

The `ai_flags` can be set to a variety of things, but here are a couple important ones. (Multiple flags can be specified by bitwise-ORing them together with the `|` operator). Check your man page for the complete list of flags.

`AI_CANONNAME` causes the `ai_canonname` of the result to the filled out with the host's canonical (real) name. `AI_PASSIVE` causes the result's IP address to be filled out with `INADDR_ANY` (IPv4) or `in6addr_any` (IPv6); this causes a subsequent call to `bind()` to auto-fill the IP address of the `struct sockaddr` with the address of the current host. That's excellent for setting up a server when you don't want to hardcode the address.

If you do use the `AI_PASSIVE`, flag, then you can pass `NULL` in the `nodename` (since `bind()` will fill it in for you later).

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

Continuing on with the input parameters, you'll likely want to set `ai_family` to `AF_UNSPEC` which tells `getaddrinfo()` to look for both IPv4 and IPv6 addresses. You can also restrict yourself to one or the other with `AF_INET` or `AF_INET6`.

Next, the `socktype` field should be set to `SOCK_STREAM` or `SOCK_DGRAM`, depending on which type of socket you want.

Finally, just leave `ai_protocol` at `0` to automatically choose your protocol type.

Now, after you get all that stuff in there, you can *finally* make the call to `getaddrinfo()`!

Of course, this is where the fun begins. The `res` will now point to a linked list of `struct addrinfo`s, and you can go through this list to get all the addresses that match what you passed in with the hints.

Now, it's possible to get some addresses that don't work for one reason or another, so what the Linux man page does is loops through the list doing a call to `socket()` and `connect()` (or `bind()` if you're setting up a server with the `AI_PASSIVE` flag) until it succeeds.

Finally, when you're done with the linked list, you need to call `freeaddrinfo()` to free up the memory (or it will be leaked, and Some People will get upset).

Return Value

Returns zero on success, or nonzero on error. If it returns nonzero, you can use the function `gai_strerror()` to get a printable version of the error code in the return value.

Example

```
// code for a client connecting to a server
// namely a stream socket to www.example.com on port 80 (http)
// either IPv4 or IPv6

int sockfd;
struct addrinfo hints, *servinfo, *p;
int rv;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // use AF_INET6 to force IPv6
hints.ai_socktype = SOCK_STREAM;

if ((rv = getaddrinfo("www.example.com", "http", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    exit(1);
}

// loop through all the results and connect to the first we can
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
        perror("socket");
        continue;
    }

    if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
        perror("connect");
        close(sockfd);
        continue;
    }

    break; // if we get here, we must have connected successfully
```

```
}

if (p == NULL) {
    // looped off the end of the list with no connection
    fprintf(stderr, "failed to connect\n");
    exit(2);
}

freeaddrinfo(servinfo); // all done with this structure
```

```
// code for a server waiting for connections
// namely a stream socket on port 3490, on this host's IP
// either IPv4 or IPv6.

int sockfd;
struct addrinfo hints, *servinfo, *p;
int rv;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // use AF_INET6 to force IPv6
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE; // use my IP address

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

// loop through all the results and bind to the first we can
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
        perror("socket");
        continue;
    }

    if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
        close(sockfd);
        perror("bind");
        continue;
    }

    break; // if we get here, we must have connected successfully
}

if (p == NULL) {
    // looped off the end of the list with no successful bind
    fprintf(stderr, "failed to bind socket\n");
    exit(2);
}

freeaddrinfo(servinfo); // all done with this structure
```

See Also

`gethostbyname()`, `getnameinfo()`

9.6 `gethostname()`

Returns the name of the system

Synopsis

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

int gethostname(char *name, size_t len);
```

Description

Your system has a name. They all do. This is a slightly more Unixy thing than the rest of the networky stuff we've been talking about, but it still has its uses.

For instance, you can get your host name, and then call `gethostbyname()` to find out your IP address.

The parameter `name` should point to a buffer that will hold the host name, and `len` is the size of that buffer in bytes. `gethostname()` won't overwrite the end of the buffer (it might return an error, or it might just stop writing), and it will `NUL`-terminate the string if there's room for it in the buffer.

Return Value

Returns zero on success, or `-1` on error (and `errno` will be set accordingly).

Example

```
char hostname[128];

gethostname(hostname, sizeof hostname);
printf("My hostname: %s\n", hostname);
```

See Also

`gethostbyname()`

9.7 `gethostbyname()`, `gethostbyaddr()`

Get an IP address for a hostname, or vice-versa

Synopsis

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

struct hostent *gethostbyname(const char *name); // DEPRECATED!
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

Description

PLEASE NOTE: these two functions are superseded by `getaddrinfo()` and `getnameinfo()`! In particular, `gethostbyname()` doesn't work well with IPv6.

These functions map back and forth between host names and IP addresses. For instance, if you have "www.example.com", you can use `gethostbyname()` to get its IP address and store it in a `struct in_addr`.

Conversely, if you have a `struct in_addr` or a `struct in6_addr`, you can use `gethostbyaddr()` to get the hostname back. `gethostbyaddr()` *is* IPv6 compatible, but you should use the newer shinier `getnameinfo()` instead.

(If you have a string containing an IP address in dots-and-numbers format that you want to look up the hostname of, you'd be better off using `getaddrinfo()` with the `AI_CANONNAME` flag.)

`gethostbyname()` takes a string like "www.yahoo.com", and returns a `struct hostent` which contains tons of information, including the IP address. (Other information is the official host name, a list of aliases, the address type, the length of the addresses, and the list of addresses—it's a general-purpose structure that's pretty easy to use for our specific purposes once you see how.)

`gethostbyaddr()` takes a `struct in_addr` or `struct in6_addr` and brings you up a corresponding host name (if there is one), so it's sort of the reverse of `gethostbyname()`. As for parameters, even though `addr` is a `char*`, you actually want to pass in a pointer to a `struct in_addr`. `len` should be `sizeof(struct in_addr)`, and `type` should be `AF_INET`.

So what is this `struct hostent` that gets returned? It has a number of fields that contain information about the host in question.

Field	Description
`char *h_name`	The real canonical host name.
`char **h_aliases`	A list of aliases that can be accessed with arrays—the last element is `NULL`
`int h_addrtype`	The result's address type, which really should be `AF_INET` for our purposes.
`int length`	The length of the addresses in bytes, which is 4 for IP (version 4) addresses.
`char **h_addr_list`	A list of IP addresses for this host. Although this is a `char**`, it's really an array of `struct in_addr*`s in disguise. The last array element is `NULL`.
`h_addr`	A commonly defined alias for `h_addr_list[0]`. If you just want any old IP address for this host (yeah, they can have more than one) just use this field.

Return Value

Returns a pointer to a resultant `struct hostent` on success, or `NULL` on error.

Instead of the normal `perror()` and all that stuff you'd normally use for error reporting, these functions have parallel results in the variable `h_errno`, which can be printed using the functions `herror()` or `hstrerror()`. These work just like the classic `errno`, `perror()`, and `strerror()` functions you're used to.

Example

```
// THIS IS A DEPRECATED METHOD OF GETTING HOST NAMES
// use getaddrinfo() instead!

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

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

int main(int argc, char *argv[])
{
    int i;
    struct hostent *he;
    struct in_addr **addr_list;

    if (argc != 2) {
        fprintf(stderr,"usage: ghbn hostname\n");
        return 1;
    }

    if ((he = gethostbyname(argv[1])) == NULL) {  // get the host info
        herror("gethostbyname");
        return 2;
    }

    // print information about this host:
    printf("Official name is: %s\n", he->h_name);
    printf("    IP addresses: ");
    addr_list = (struct in_addr **)he->h_addr_list;
    for(i = 0; addr_list[i] != NULL; i++) {
        printf("%s ", inet_ntoa(*addr_list[i]));
    }
    printf("\n");

    return 0;
}
```

```
// THIS HAS BEEN SUPERSEDED
// use getnameinfo() instead!

struct hostent *he;
struct in_addr ipv4addr;
struct in6_addr ipv6addr;

inet_pton(AF_INET, "192.0.2.34", &ipv4addr);
he = gethostbyaddr(&ipv4addr, sizeof ipv4addr, AF_INET);
printf("Host name: %s\n", he->h_name);

inet_pton(AF_INET6, "2001:db8:63b3:1::beef", &ipv6addr);
he = gethostbyaddr(&ipv6addr, sizeof ipv6addr, AF_INET6);
printf("Host name: %s\n", he->h_name);
```

See Also

`getaddrinfo()`, `getnameinfo()`, `gethostname()`, `errno`, `perror()`, `strerror()`, `struct in_addr`

9.8 getnameinfo()

Look up the host name and service name information for a given `struct sockaddr`.

Synopsis

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

int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *host, size_t hostlen,
                char *serv, size_t servlen, int flags);
```

Description

This function is the opposite of `getaddrinfo()`, that is, this function takes an already loaded `struct sockaddr` and does a name and service name lookup on it. It replaces the old `gethostbyaddr()` and `getservbyport()` functions.

You have to pass in a pointer to a `struct sockaddr` (which in actuality is probably a `struct sockaddr_in` or `struct sockaddr_in6` that you've cast) in the `sa` parameter, and the length of that `struct` in the `salen`.

The resultant host name and service name will be written to the area pointed to by the `host` and `serv` parameters. Of course, you have to specify the max lengths of these buffers in `hostlen` and `servlen`.

Finally, there are several flags you can pass, but here a a couple good ones. `NI_NOFQDN` will cause the `host` to only contain the host name, not the whole domain name. `NI_NAMEREQD` will cause the function to fail if the name cannot be found with a DNS lookup (if you don't specify this flag and the name can't be found, `getnameinfo()` will put a string version of the IP address in `host` instead).

As always, check your local man pages for the full scoop.

Return Value

Returns zero on success, or non-zero on error. If the return value is non-zero, it can be passed to `gai_strerror()` to get a human-readable string. See `getaddrinfo` for more information.

Example

```
struct sockaddr_in6 sa; // could be IPv4 if you want
char host[1024];
char service[20];

// pretend sa is full of good information about the host and port...

getnameinfo(&sa, sizeof sa, host, sizeof host, service, sizeof service, 0);

printf("   host: %s\n", host);    // e.g. "www.example.com"
printf("service: %s\n", service); // e.g. "http"
```

See Also

`getaddrinfo()`, `gethostbyaddr()`

9.9 `getpeername()`

Return address info about the remote side of the connection

Synopsis

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

int getpeername(int s, struct sockaddr *addr, socklen_t *len);
```

Description

Once you have either `accept()`ed a remote connection, or `connect()`ed to a server, you now have what is known as a *peer*. Your peer is simply the computer you're connected to, identified by an IP address and a port. So…

`getpeername()` simply returns a `struct sockaddr_in` filled with information about the machine you're connected to.

Why is it called a "name"? Well, there are a lot of different kinds of sockets, not just Internet Sockets like we're using in this guide, and so "name" was a nice generic term that covered all cases. In our case, though, the peer's "name" is it's IP address and port.

Although the function returns the size of the resultant address in `len`, you must preload `len` with the size of `addr`.

Return Value

Returns zero on success, or `-1` on error (and `errno` will be set accordingly).

Example

```
// assume s is a connected socket

socklen_t len;
struct sockaddr_storage addr;
char ipstr[INET6_ADDRSTRLEN];
int port;

len = sizeof addr;
getpeername(s, (struct sockaddr*)&addr, &len);

// deal with both IPv4 and IPv6:
if (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, sizeof 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, sizeof ipstr);
}

printf("Peer IP address: %s\n", ipstr);
printf("Peer port      : %d\n", port);
```

See Also

`gethostname()`, `gethostbyname()`, `gethostbyaddr()`

9.10 `errno`

Holds the error code for the last system call

Synopsis

```
#include <errno.h>

int errno;
```

Description

This is the variable that holds error information for a lot of system calls. If you'll recall, things like `socket()` and `listen()` return `-1` on error, and they set the exact value of `errno` to let you know specifically which error occurred.

The header file `errno.h` lists a bunch of constant symbolic names for errors, such as `EADDRINUSE`, `EPIPE`, `ECONNREFUSED`, etc. Your local man pages will tell you what codes can be returned as an error, and you can use these at run time to handle different errors in different ways.

Or, more commonly, you can call `perror()` or `strerror()` to get a human-readable version of the error.

One thing to note, for you multithreading enthusiasts, is that on most systems `errno` is defined in a threadsafe manner. (That is, it's not actually a global variable, but it behaves just like a global variable would in a single-threaded environment.)

Return Value

The value of the variable is the latest error to have transpired, which might be the code for "success" if the last action succeeded.

Example

```
s = socket(PF_INET, SOCK_STREAM, 0);
if (s == -1) {
    perror("socket"); // or use strerror()
}

tryagain:
if (select(n, &readfds, NULL, NULL) == -1) {
    // an error has occurred!!

    // if we were only interrupted, just restart the select() call:
    if (errno == EINTR) goto tryagain;  // AAAA! goto!!!

    // otherwise it's a more serious error:
    perror("select");
    exit(1);
}
```

See Also

`perror()`, `strerror()`

9.11 `fcntl()`

Control socket descriptors

Synopsis

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

int fcntl(int s, int cmd, long arg);
```

Description

This function is typically used to do file locking and other file-oriented stuff, but it also has a couple socket-related functions that you might see or use from time to time.

Parameter `s` is the socket descriptor you wish to operate on, `cmd` should be set to `F_SETFL`, and `arg` can be one of the following commands. (Like I said, there's more to `fcntl()` than I'm letting on here, but I'm trying to stay socket-oriented.)

`cmd`	Description
`O_NONBLOCK`	Set the socket to be non-blocking. See the section on blocking for more details.
`O_ASYNC`	Set the socket to do asynchronous I/O. When data is ready to be `recv()`'d on the socket, the signal `SIGIO` will be raised. This is rare to see, and beyond the scope of the guide. And I think it's only available on certain systems.

Return Value

Returns zero on success, or `-1` on error (and `errno` will be set accordingly).

Different uses of the `fcntl()` system call actually have different return values, but I haven't covered them here because they're not socket-related. See your local `fcntl()` man page for more information.

Example

```
int s = socket(PF_INET, SOCK_STREAM, 0);

fcntl(s, F_SETFL, O_NONBLOCK);  // set to non-blocking
fcntl(s, F_SETFL, O_ASYNC);     // set to asynchronous I/O
```

See Also

Blocking, `send()`

9.12 `htons()`, `htonl()`, `ntohs()`, `ntohl()`

Convert multi-byte integer types from host byte order to network byte order

Synopsis

```
#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);
```

Description

Just to make you really unhappy, different computers use different byte orderings internally for their multibyte integers (i.e. any integer that's larger than a `char`). The upshot of this is that if you `send()` a two-byte `short int` from an Intel box to a Mac (before they became Intel boxes, too, I mean), what one computer thinks is the number `1`, the other will think is the number `256`, and vice-versa.

The way to get around this problem is for everyone to put aside their differences and agree that Motorola and IBM had it right, and Intel did it the weird way, and so we all convert our byte orderings to "big-endian" before sending them out. Since Intel is a "little-endian" machine, it's far more politically correct to call our preferred byte ordering "Network Byte Order". So these functions convert from your native byte order to network byte order and back again.

(This means on Intel these functions swap all the bytes around, and on PowerPC they do nothing because the bytes are already in Network Byte Order. But you should always use them in your code anyway, since someone might want to build it on an Intel machine and still have things work properly.)

Note that the types involved are 32-bit (4 byte, probably `int`) and 16-bit (2 byte, very likely `short`) numbers. 64-bit machines might have a `htonll()` for 64-bit `int`s, but I've not seen it. You'll just have to write your own.

Anyway, the way these functions work is that you first decide if you're converting *from* host (your machine's) byte order or from network byte order. If "host", the the first letter of the function you're going to call is "h". Otherwise it's "n" for "network". The middle of the function name is always "to" because you're converting from one "to" another, and the penultimate letter shows what you're converting *to*. The last letter is the size of the data, "s" for short, or "l" for long. Thus:

Function	Description
`htons()`	host `to` network `s`hort
`htonl()`	host `to` network `l`ong
`ntohs()`	network `to` host `s`hort
`ntohl()`	network `to` host `l`ong

Return Value

Each function returns the converted value.

Example

```
uint32_t some_long = 10;
uint16_t some_short = 20;

uint32_t network_byte_order;

// convert and send
network_byte_order = htonl(some_long);
send(s, &network_byte_order, sizeof(uint32_t), 0);
```

```
some_short == ntohs(htons(some_short)); // this expression is true
```

9.13 `inet_ntoa()`, `inet_aton()`, `inet_addr`

Convert IP addresses from a dots-and-number string to a `struct in_addr` and back

Synopsis

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

// ALL THESE ARE DEPRECATED! Use inet_pton()  or inet_ntop() instead!!

char *inet_ntoa(struct in_addr in);
int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
```

Description

These functions are deprecated because they don't handle IPv6! Use `inet_ntop()` or `inet_pton()` instead! They are included here because they can still be found in the wild.

All of these functions convert from a `struct in_addr` (part of your `struct sockaddr_in`, most likely) to a string in dots-and-numbers format (e.g. "192.168.5.10") and vice-versa. If you have an IP address passed on the command line or something, this is the easiest way to get a `struct in_addr` to `connect()` to, or whatever. If you need more power, try some of the DNS functions like `gethostbyname()` or attempt a *coup d'État* in your local country.

The function `inet_ntoa()` converts a network address in a `struct in_addr` to a dots-and-numbers format string. The "n" in "ntoa" stands for network, and the "a" stands for ASCII for historical reasons (so it's "Network To ASCII"—the "toa" suffix has an analogous friend in the C library called `atoi()` which converts an ASCII string to an integer).

The function `inet_aton()` is the opposite, converting from a dots-and-numbers string into a `in_addr_t` (which is the type of the field `s_addr` in your `struct in_addr`).

Finally, the function `inet_addr()` is an older function that does basically the same thing as `inet_aton()`. It's theoretically deprecated, but you'll see it a lot and the police won't come get you if you use it.

Return Value

`inet_aton()` returns non-zero if the address is a valid one, and it returns zero if the address is invalid.

`inet_ntoa()` returns the dots-and-numbers string in a static buffer that is overwritten with each call to the function.

`inet_addr()` returns the address as an `in_addr_t`, or `-1` if there's an error. (That is the same result as if you tried to convert the string "`255.255.255.255`", which is a valid IP address. This is why `inet_aton()` is better.)

Example

```
struct sockaddr_in antelope;
char *some_addr;

inet_aton("10.0.0.1", &antelope.sin_addr); // store IP in antelope

some_addr = inet_ntoa(antelope.sin_addr); // return the IP
printf("%s\n", some_addr); // prints "10.0.0.1"

// and this call is the same as the inet_aton() call, above:
antelope.sin_addr.s_addr = inet_addr("10.0.0.1");
```

See Also

`inet_ntop()`, `inet_pton()`, `gethostbyname()`, `gethostbyaddr()`

9.14 `inet_ntop()`, `inet_pton()`

Convert IP addresses to human-readable form and back.

Synopsis

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

const char *inet_ntop(int af, const void *src,
                      char *dst, socklen_t size);

int inet_pton(int af, const char *src, void *dst);
```

Description

These functions are for dealing with human-readable IP addresses and converting them to their binary representation for use with various functions and system calls. The “n” stands for “network”, and “p” for “presentation”. Or “text presentation”. But you can think of it as “printable”. “ntop” is “network to printable”. See?

Sometimes you don’t want to look at a pile of binary numbers when looking at an IP address. You want it in a nice printable form, like `192.0.2.180`, or `2001:db8:8714:3a90::12`. In that case, `inet_ntop()` is for you.

`inet_ntop()` takes the address family in the `af` parameter (either `AF_INET` or `AF_INET6`). The `src` parameter should be a pointer to either a `struct in_addr` or `struct in6_addr` containing the address you wish to convert to a string. Finally `dst` and `size` are the pointer to the destination string and the maximum length of that string.

What should the maximum length of the `dst` string be? What is the maximum length for IPv4 and IPv6 addresses? Fortunately there are a couple of macros to help you out. The maximum lengths are: `INET_ADDRSTRLEN` and `INET6_ADDRSTRLEN`.

Other times, you might have a string containing an IP address in readable form, and you want to pack it into a `struct sockaddr_in` or a `struct sockaddr_in6`. In that case, the opposite function `inet_pton()` is what you’re after.

`inet_pton()` also takes an address family (either `AF_INET` or `AF_INET6`) in the `af` parameter. The `src` parameter is a pointer to a string containing the IP address in printable form. Lastly the `dst` parameter points to where the result should be stored, which is probably a `struct in_addr` or `struct in6_addr`.

These functions don't do DNS lookups—you'll need `getaddrinfo()` for that.

Return Value

`inet_ntop()` returns the `dst` parameter on success, or `NULL` on failure (and `errno` is set).

`inet_pton()` returns `1` on success. It returns `-1` if there was an error (`errno` is set), or `0` if the input isn't a valid IP address.

Example

```
// IPv4 demo of inet_ntop() and inet_pton()

struct sockaddr_in sa;
char str[INET_ADDRSTRLEN];

// store this IP address in sa:
inet_pton(AF_INET, "192.0.2.33", &(sa.sin_addr));

// now get it back and print it
inet_ntop(AF_INET, &(sa.sin_addr), str, INET_ADDRSTRLEN);

printf("%s\n", str); // prints "192.0.2.33"
```

```
// IPv6 demo of inet_ntop() and inet_pton()
// (basically the same except with a bunch of 6s thrown around)

struct sockaddr_in6 sa;
char str[INET6_ADDRSTRLEN];

// store this IP address in sa:
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &(sa.sin6_addr));

// now get it back and print it
inet_ntop(AF_INET6, &(sa.sin6_addr), str, INET6_ADDRSTRLEN);

printf("%s\n", str); // prints "2001:db8:8714:3a90::12"
```

```
// Helper function you can use:

//Convert a struct sockaddr address to a string, IPv4 and 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);
            break;

        case AF_INET6:
            inet_ntop(AF_INET6, &(((struct sockaddr_in6 *)sa)->sin6_addr),
                    s, maxlen);
            break;

        default:
```

```
            strncpy(s, "Unknown AF", maxlen);
            return NULL;
    }

    return s;
}
```

See Also

`getaddrinfo()`

9.15 `listen()`

Tell a socket to listen for incoming connections

Synopsis

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

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

Description

You can take your socket descriptor (made with the `socket()` system call) and tell it to listen for incoming connections. This is what differentiates the servers from the clients, guys.

The `backlog` parameter can mean a couple different things depending on the system you on, but loosely it is how many pending connections you can have before the kernel starts rejecting new ones. So as the new connections come in, you should be quick to `accept()` them so that the backlog doesn't fill. Try setting it to 10 or so, and if your clients start getting "Connection refused" under heavy load, set it higher.

Before calling `listen()`, your server should call `bind()` to attach itself to a specific port number. That port number (on the server's IP address) will be the one that clients connect to.

Return Value

Returns zero on success, or `-1` on error (and `errno` will be set accordingly).

Example

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

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

// make a socket:
```

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

// bind it to the port we passed in to getaddrinfo():

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

listen(sockfd, 10); // set s up to be a server (listening) socket

// then have an accept() loop down here somewhere
```

See Also

`accept()`, `bind()`, `socket()`

9.16 `perror()`, `strerror()`

Print an error as a human-readable string

Synopsis

```
#include <stdio.h>
#include <string.h>   // for strerror()

void perror(const char *s);
char *strerror(int errnum);
```

Description

Since so many functions return `-1` on error and set the value of the variable `errno` to be some number, it would sure be nice if you could easily print that in a form that made sense to you.

Mercifully, `perror()` does that. If you want more description to be printed before the error, you can point the parameter `s` to it (or you can leave `s` as `NULL` and nothing additional will be printed).

In a nutshell, this function takes `errno` values, like `ECONNRESET`, and prints them nicely, like "Connection reset by peer."

The function `strerror()` is very similar to `perror()`, except it returns a pointer to the error message string for a given value (you usually pass in the variable `errno`).

Return Value

`strerror()` returns a pointer to the error message string.

Example

```
int s;

s = socket(PF_INET, SOCK_STREAM, 0);

if (s == -1) { // some error has occurred
    // prints "socket error: " + the error message:
    perror("socket error");
}
```

```
// similarly:
if (listen(s, 10) == -1) {
    // this prints "an error: " + the error message from errno:
    printf("an error: %s\n", strerror(errno));
}
```

See Also

`errno`

9.17 `poll()`

Test for events on multiple sockets simultaneously

Synopsis

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

int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

Description

This function is very similar to `select()` in that they both watch sets of file descriptors for events, such as incoming data ready to `recv()`, socket ready to `send()` data to, out-of-band data ready to `recv()`, errors, etc.

The basic idea is that you pass an array of `nfds` `struct pollfd`s in `ufds`, along with a timeout in milliseconds (1000 milliseconds in a second). The `timeout` can be negative if you want to wait forever. If no event happens on any of the socket descriptors by the timeout, `poll()` will return.

Each element in the array of `struct pollfd`s represents one socket descriptor, and contains the following fields:

```
struct pollfd {
    int fd;         // the socket descriptor
    short events;   // bitmap of events we're interested in
    short revents;  // when poll() returns, bitmap of events that occurred
};
```

Before calling `poll()`, load `fd` with the socket descriptor (if you set `fd` to a negative number, this `struct pollfd` is ignored and its `revents` field is set to zero) and then construct the `events` field by bitwise-ORing the following macros:

Macro	Description
`POLLIN`	Alert me when data is ready to `recv()` on this socket.
`POLLOUT`	Alert me when I can `send()` data to this socket without blocking.
`POLLPRI`	Alert me when out-of-band data is ready to `recv()` on this socket.

Once the `poll()` call returns, the `revents` field will be constructed as a bitwise-OR of the above fields, telling you which descriptors actually have had that event occur. Additionally, these other fields might be present:

Macro	Description
`POLLERR`	An error has occurred on this socket.
`POLLHUP`	The remote side of the connection hung up.
`POLLNVAL`	Something was wrong with the socket descriptor `fd`—maybe it's uninitialized?

Return Value

Returns the number of elements in the `ufds` array that have had event occur on them; this can be zero if the timeout occurred. Also returns `-1` on error (and `errno` will be set accordingly).

Example

```
int s1, s2;
int rv;
char buf1[256], buf2[256];
struct pollfd ufds[2];

s1 = socket(PF_INET, SOCK_STREAM, 0);
s2 = socket(PF_INET, SOCK_STREAM, 0);

// pretend we've connected both to a server at this point
//connect(s1, ...)...
//connect(s2, ...)...

// set up the array of file descriptors.
//
// in this example, we want to know when there's normal or out-of-band
// data ready to be recv()'d...

ufds[0].fd = s1;
ufds[0].events = POLLIN | POLLPRI; // check for normal or out-of-band

ufds[1].fd = s2;
ufds[1].events = POLLIN; // check for just normal data

// wait for events on the sockets, 3.5 second timeout
rv = poll(ufds, 2, 3500);

if (rv == -1) {
    perror("poll"); // error occurred in poll()
} else if (rv == 0) {
    printf("Timeout occurred! No data after 3.5 seconds.\n");
} else {
    // check for events on s1:
    if (ufds[0].revents & POLLIN) {
        recv(s1, buf1, sizeof buf1, 0); // receive normal data
    }
    if (ufds[0].revents & POLLPRI) {
        recv(s1, buf1, sizeof buf1, MSG_OOB); // out-of-band data
    }

    // check for events on s2:
    if (ufds[1].revents & POLLIN) {
        recv(s1, buf2, sizeof buf2, 0);
    }
}
```

See Also

`select()`

9.18 `recv()`, `recvfrom()`

Receive data on a socket

Synopsis

```
#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 *from, socklen_t *fromlen);
```

Description

Once you have a socket up and connected, you can read incoming data from the remote side using the `recv()` (for TCP `SOCK_STREAM` sockets) and `recvfrom()` (for UDP `SOCK_DGRAM` sockets).

Both functions take the socket descriptor `s`, a pointer to the buffer `buf`, the size (in bytes) of the buffer `len`, and a set of `flags` that control how the functions work.

Additionally, the `recvfrom()` takes a `struct sockaddr*`, `from` that will tell you where the data came from, and will fill in `fromlen` with the size of `struct sockaddr`. (You must also initialize `fromlen` to be the size of `from` or `struct sockaddr`.)

So what wondrous flags can you pass into this function? Here are some of them, but you should check your local man pages for more information and what is actually supported on your system. You bitwise-or these together, or just set `flags` to `0` if you want it to be a regular vanilla `recv()`.

Macro	Description
`MSG_OOB`	Receive Out of Band data. This is how to get data that has been sent to you with the `MSG_OOB` flag in `send()`. As the receiving side, you will have had signal `SIGURG` raised telling you there is urgent data. In your handler for that signal, you could call `recv()` with this `MSG_OOB` flag.
`MSG_PEEK`	If you want to call `recv()` "just for pretend", you can call it with this flag. This will tell you what's waiting in the buffer for when you call `recv()` "for real" (i.e. *without* the `MSG_PEEK` flag. It's like a sneak preview into the next `recv()` call.
`MSG_WAITALL`	Tell `recv()` to not return until all the data you specified in the `len` parameter. It will ignore your wishes in extreme circumstances, however, like if a signal interrupts the call or if some error occurs or if the remote side closes the connection, etc. Don't be mad with it.

When you call `recv()`, it will block until there is some data to read. If you want to not block, set the socket to non-blocking or check with `select()` or `poll()` to see if there is incoming data before calling `recv()` or `recvfrom()`.

Return Value

Returns the number of bytes actually received (which might be less than you requested in the `len` parameter), or `-1` on error (and `errno` will be set accordingly).

If the remote side has closed the connection, `recv()` will return `0`. This is the normal method for determining if the remote side has closed the connection. Normality is good, rebel!

Example

```
// stream sockets and recv()

struct addrinfo hints, *res;
int sockfd;
char buf[512];
int byte_count;

// get host info, make socket, and connect it
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
getaddrinfo("www.example.com", "3490", &hints, &res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
connect(sockfd, res->ai_addr, res->ai_addrlen);

// all right! now that we're connected, we can receive some data!
byte_count = recv(sockfd, buf, sizeof buf, 0);
printf("recv()'d %d bytes of data in buf\n", byte_count);
```

```
// datagram sockets and recvfrom()

struct addrinfo hints, *res;
int sockfd;
int byte_count;
socklen_t fromlen;
struct sockaddr_storage addr;
char buf[512];
char ipstr[INET6_ADDRSTRLEN];

// get host info, make socket, bind it to port 4950
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_DGRAM;
hints.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 need to accept(), just recvfrom():

fromlen = sizeof addr;
byte_count = recvfrom(sockfd, buf, sizeof buf, 0, &addr, &fromlen);

printf("recv()'d %d bytes of data in buf\n", byte_count);
printf("from IP address %s\n",
    inet_ntop(addr.ss_family,
        addr.ss_family == AF_INET?
            ((struct sockaddr_in *)&addr)->sin_addr:
            ((struct sockaddr_in6 *)&addr)->sin6_addr,
        ipstr, sizeof ipstr);
```

See Also

send(), sendto(), select(), poll(), Blocking

9.19 select()

Check if sockets descriptors are ready to read/write

Synopsis

```
#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);
```

Description

The select() function gives you a way to simultaneously check multiple sockets to see if they have data waiting to be recv()d, or if you can send() data to them without blocking, or if some exception has occurred.

You populate your sets of socket descriptors using the macros, like FD_SET(), above. Once you have the set, you pass it into the function as one of the following parameters: readfds if you want to know when any of the sockets in the set is ready to recv() data, writefds if any of the sockets is ready to send() data to, and/or exceptfds if you need to know when an exception (error) occurs on any of the sockets. Any or all of these parameters can be NULL if you're not interested in those types of events. After select() returns, the values in the sets will be changed to show which are ready for reading or writing, and which have exceptions.

The first parameter, n is the highest-numbered socket descriptor (they're just ints, remember?) plus one.

Lastly, the struct timeval, timeout, at the end—this lets you tell select() how long to check these sets for. It'll return after the timeout, or when an event occurs, whichever is first. The struct timeval has two fields: tv_sec is the number of seconds, to which is added tv_usec, the number of microseconds (1,000,000 microseconds in a second).

The helper macros do the following:

Macro	Description
FD_SET(int fd, fd_set *set);	Add fd to the set.
FD_CLR(int fd, fd_set *set);	Remove fd from the set.
FD_ISSET(int fd, fd_set *set);	Return true if fd is in the set.
FD_ZERO(fd_set *set);	Clear all entries from the set.

Note for Linux users: Linux's select() can return "ready-to-read" and then not actually be ready to read, thus causing the subsequent read() call to block. You can work around this bug by setting O_NONBLOCK flag on the receiving socket so it errors with EWOULDBLOCK, then ignoring this error if it occurs. See the fcntl() man page for more info on setting a socket to non-blocking.

Return Value

Returns the number of descriptors in the set on success, `0` if the timeout was reached, or `-1` on error (and `errno` will be set accordingly). Also, the sets are modified to show which sockets are ready.

Example

```
int s1, s2, n;
fd_set readfds;
struct timeval tv;
char buf1[256], buf2[256];

// pretend we've connected both to a server at this point
//s1 = socket(...);
//s2 = socket(...);
//connect(s1, ...)...
//connect(s2, ...)...

// clear the set ahead of time
FD_ZERO(&readfds);

// add our descriptors to the set
FD_SET(s1, &readfds);
FD_SET(s2, &readfds);

// since we got s2 second, it's the "greater", so we use that for
// the n param in select()
n = s2 + 1;

// wait until either socket has data ready to be recv()d (timeout 10.5 secs)
tv.tv_sec = 10;
tv.tv_usec = 500000;
rv = select(n, &readfds, NULL, NULL, &tv);

if (rv == -1) {
    perror("select"); // error occurred in select()
} else if (rv == 0) {
    printf("Timeout occurred! No data after 10.5 seconds.\n");
} else {
    // one or both of the descriptors have data
    if (FD_ISSET(s1, &readfds)) {
        recv(s1, buf1, sizeof buf1, 0);
    }
    if (FD_ISSET(s2, &readfds)) {
        recv(s2, buf2, sizeof buf2, 0);
    }
}
```

See Also

`poll()`

9.20 `setsockopt()`, `getsockopt()`

Set various options for a socket

Synopsis

```
#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);
```

Description

Sockets are fairly configurable beasts. In fact, they are so configurable, I'm not even going to cover it all here. It's probably system-dependent anyway. But I will talk about the basics.

Obviously, these functions get and set certain options on a socket. On a Linux box, all the socket information is in the man page for socket in section 7. (Type: "`man 7 socket`" to get all these goodies.)

As for parameters, `s` is the socket you're talking about, level should be set to `SOL_SOCKET`. Then you set the `optname` to the name you're interested in. Again, see your man page for all the options, but here are some of the most fun ones:

`optname`	Description
`SO_BINDTODEVICE`	Bind this socket to a symbolic device name like `eth0` instead of using `bind()` to bind it to an IP address. Type the command `ifconfig` under Unix to see the device names.
`SO_REUSEADDR`	Allows other sockets to `bind()` to this port, unless there is an active listening socket bound to the port already. This enables you to get around those "Address already in use" error messages when you try to restart your server after a crash.
`SOCK_DGRAM`	Allows UDP datagram (`SOCK_DGRAM`) sockets to send and receive packets sent to and from the broadcast address. Does nothing—*NOTHING!!*—to TCP stream sockets! Hahaha!

As for the parameter `optval`, it's usually a pointer to an `int` indicating the value in question. For booleans, zero is false, and non-zero is true. And that's an absolute fact, unless it's different on your system. If there is no parameter to be passed, `optval` can be `NULL`.

The final parameter, `optlen`, should be set to the length of `optval`, probably `sizeof(int)`, but varies depending on the option. Note that in the case of `getsockopt()`, this is a pointer to a `socklen_t`, and it specifies the maximum size object that will be stored in `optval` (to prevent buffer overflows). And `getsockopt()` will modify the value of `optlen` to reflect the number of bytes actually set.

Warning: on some systems (notably Sun and Windows), the option can be a `char` instead of an `int`, and is set to, for example, a character value of `'1'` instead of an `int` value of `1`. Again, check your own man pages for more info with "`man setsockopt`" and "`man 7 socket`"!

Return Value

Returns zero on success, or `-1` on error (and `errno` will be set accordingly).

Example

```
int optval;
int optlen;
char *optval2;
```

```
// set SO_REUSEADDR on a socket to true (1):
optval = 1;
setsockopt(s1, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof optval);

// bind a socket to a device name (might not work on all systems):
optval2 = "eth1"; // 4 bytes long, so 4, below:
setsockopt(s2, SOL_SOCKET, SO_BINDTODEVICE, optval2, 4);

// see if the SO_BROADCAST flag is set:
getsockopt(s3, SOL_SOCKET, SO_BROADCAST, &optval, &optlen);
if (optval != 0) {
    print("SO_BROADCAST enabled on s3!\n");
}
```

See Also

`fcntl()`

9.21 `send()`, `sendto()`

Send data out over a socket

Synopsis

```
#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);
```

Description

These functions send data to a socket. Generally speaking, `send()` is used for TCP `SOCK_STREAM` connected sockets, and `sendto()` is used for UDP `SOCK_DGRAM` unconnected datagram sockets. With the unconnected sockets, you must specify the destination of a packet each time you send one, and that's why the last parameters of `sendto()` define where the packet is going.

With both `send()` and `sendto()`, the parameter `s` is the socket, `buf` is a pointer to the data you want to send, `len` is the number of bytes you want to send, and `flags` allows you to specify more information about how the data is to be sent. Set `flags` to zero if you want it to be "normal" data. Here are some of the commonly used flags, but check your local `send()` man pages for more details:

Macro	Description
`MSG_OOB`	Send as "out of band" data. TCP supports this, and it's a way to tell the receiving system that this data has a higher priority than the normal data. The receiver will receive the signal `SIGURG` and it can then receive this data without first receiving all the rest of the normal data in the queue.
`MSG_DONTROUTE`	Don't send this data over a router, just keep it local.
`MSG_DONTWAIT`	If `send()` would block because outbound traffic is clogged, have it return `EAGAIN`. This is like a "enable non-blocking just for this send." See the section on blocking for more details.

Macro	Description
MSG_NOSIGNAL	If you send() to a remote host which is no longer recv()ing, you'll typically get the signal SIGPIPE. Adding this flag prevents that signal from being raised.

Return Value

Returns the number of bytes actually sent, or -1 on error (and errno will be set accordingly). Note that the number of bytes actually sent might be less than the number you asked it to send! See the section on handling partial send()s for a helper function to get around this.

Also, if the socket has been closed by either side, the process calling send() will get the signal SIGPIPE. (Unless send() was called with the MSG_NOSIGNAL flag.)

Example

```
int spatula_count = 3490;
char *secret_message = "The Cheese is in The Toaster";

int stream_socket, dgram_socket;
struct sockaddr_in dest;
int temp;

// first with TCP stream sockets:

// assume sockets are made and connected
//stream_socket = socket(...
//connect(stream_socket, ...

// convert to network byte order
temp = htonl(spatula_count);
// send data normally:
send(stream_socket, &temp, sizeof temp, 0);

// send secret message out of band:
send(stream_socket, secret_message, strlen(secret_message)+1, MSG_OOB);

// now with UDP datagram sockets:
//getaddrinfo(...
//dest = ... // assume "dest" holds the address of the destination
//dgram_socket = socket(...

// send secret message normally:
sendto(dgram_socket, secret_message, strlen(secret_message)+1, 0,
       (struct sockaddr*)&dest, sizeof dest);
```

See Also

recv(), recvfrom()

9.22 shutdown()

Stop further sends and receives on a socket

Synopsis

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

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

Description

That's it! I've had it! No more `send()`s are allowed on this socket, but I still want to `recv()` data on it! Or vice-versa! How can I do this?

When you `close()` a socket descriptor, it closes both sides of the socket for reading and writing, and frees the socket descriptor. If you just want to close one side or the other, you can use this `shutdown()` call.

As for parameters, `s` is obviously the socket you want to perform this action on, and what action that is can be specified with the `how` parameter. `how` can be `SHUT_RD` to prevent further `recv()`s, `SHUT_WR` to prohibit further `send()`s, or `SHUT_RDWR` to do both.

Note that `shutdown()` doesn't free up the socket descriptor, so you still have to eventually `close()` the socket even if it has been fully shut down.

This is a rarely used system call.

Return Value

Returns zero on success, or `-1` on error (and `errno` will be set accordingly).

Example

```
int s = socket(PF_INET, SOCK_STREAM, 0);

// ...do some send()s and stuff in here...

// and now that we're done, don't allow any more sends()s:
shutdown(s, SHUT_WR);
```

See Also

`close()`

9.23 `socket()`

Allocate a socket descriptor

Synopsis

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

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

Description

Returns a new socket descriptor that you can use to do sockety things with. This is generally the first call in the whopping process of writing a socket program, and you can use the result for subsequent calls to `listen()`, `bind()`, `accept()`, or a variety of other functions.

In usual usage, you get the values for these parameters from a call to `getaddrinfo()`, as shown in the example below. But you can fill them in by hand if you really want to.

Parameter	Description
`domain`	`domain` describes what kind of socket you're interested in. This can, believe me, be a wide variety of things, but since this is a socket guide, it's going to be `PF_INET` for IPv4, and `PF_INET6` for IPv6.
`type`	Also, the `type` parameter can be a number of things, but you'll probably be setting it to either `SOCK_STREAM` for reliable TCP sockets (`send()`, `recv()`) or `SOCK_DGRAM` for unreliable fast UDP sockets (`sendto()`, `recvfrom()`). (Another interesting socket type is `SOCK_RAW` which can be used to construct packets by hand. It's pretty cool.)
`protocol`	Finally, the `protocol` parameter tells which protocol to use with a certain socket type. Like I've already said, for instance, `SOCK_STREAM` uses TCP. Fortunately for you, when using `SOCK_STREAM` or `SOCK_DGRAM`, you can just set the protocol to 0, and it'll use the proper protocol automatically. Otherwise, you can use `getprotobyname()` to look up the proper protocol number.

Return Value

The new socket descriptor to be used in subsequent calls, or `-1` on error (and `errno` will be set accordingly).

Example

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;     // AF_INET, AF_INET6, or AF_UNSPEC
hints.ai_socktype = SOCK_STREAM; // SOCK_STREAM or SOCK_DGRAM

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

// make a socket using the information gleaned from getaddrinfo():
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

See Also

`accept()`, `bind()`, `getaddrinfo()`, `listen()`

9.24 `struct sockaddr` and pals

Structures for handling internet addresses

Synopsis

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

// All pointers to socket address structures are often cast to pointers
// to this type before use in various functions and system calls:

struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};


// IPv4 AF_INET sockets:

struct sockaddr_in {
    short            sin_family;   // e.g. AF_INET, AF_INET6
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, below
    char             sin_zero[8];  // zero this if you want to
};

struct in_addr {
    unsigned long s_addr;          // load with inet_pton()
};


// IPv6 AF_INET6 sockets:

struct sockaddr_in6 {
    u_int16_t       sin6_family;   // address family, AF_INET6
    u_int16_t       sin6_port;     // port number, Network Byte Order
    u_int32_t       sin6_flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr;     // IPv6 address
    u_int32_t       sin6_scope_id; // Scope ID
};

struct in6_addr {
    unsigned char   s6_addr[16];   // load with inet_pton()
};


// General socket address holding structure, big enough to hold either
// struct sockaddr_in or struct sockaddr_in6 data:

struct sockaddr_storage {
    sa_family_t  ss_family;     // address family

    // all this is padding, implementation specific, ignore it:
    char      __ss_pad1[_SS_PAD1SIZE];
    int64_t   __ss_align;
    char      __ss_pad2[_SS_PAD2SIZE];
};
```

Description

These are the basic structures for all syscalls and functions that deal with internet addresses. Often you'll use `getaddrinfo()` to fill these structures out, and then will read them when you have to.

In memory, the `struct sockaddr_in` and `struct sockaddr_in6` share the same beginning structure as `struct sockaddr`, and you can freely cast the pointer of one type to the other without any harm, except the possible end of the universe.

Just kidding on that end-of-the-universe thing…if the universe does end when you cast a `struct sockaddr_in*` to a `struct sockaddr*`, I promise you it's pure coincidence and you shouldn't even worry about it.

So, with that in mind, remember that whenever a function says it takes a `struct sockaddr*` you can cast your `struct sockaddr_in*`, `struct sockaddr_in6*`, or `struct sockaddr_storage*` to that type with ease and safety.

`struct sockaddr_in` is the structure used with IPv4 addresses (e.g. "192.0.2.10"). It holds an address family (`AF_INET`), a port in `sin_port`, and an IPv4 address in `sin_addr`.

There's also this `sin_zero` field in `struct sockaddr_in` which some people claim must be set to zero. Other people don't claim anything about it (the Linux documentation doesn't even mention it at all), and setting it to zero doesn't seem to be actually necessary. So, if you feel like it, set it to zero using `memset()`.

Now, that `struct in_addr` is a weird beast on different systems. Sometimes it's a crazy `union` with all kinds of `#define`s and other nonsense. But what you should do is only use the `s_addr` field in this structure, because many systems only implement that one.

`struct sockaddr_in6` and `struct in6_addr` are very similar, except they're used for IPv6.

`struct sockaddr_storage` is a struct you can pass to `accept()` or `recvfrom()` when you're trying to write IP version-agnostic code and you don't know if the new address is going to be IPv4 or IPv6. The `struct sockaddr_storage` structure is large enough to hold both types, unlike the original small `struct sockaddr`.

Example

```
// 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);
```

See Also

`accept()`, `bind()`, `connect()`, `inet_aton()`, `inet_ntoa()`

Chapter 10

More References

You've come this far, and now you're screaming for more! Where else can you go to learn more about all this stuff?

10.1 Books

For old-school actual hold-it-in-your-hand pulp paper books, try some of the following excellent books. These redirect to affiliate links with a popular bookseller, giving me nice kickbacks. If you're merely feeling generous, you can paypal a donation to `beej@beej.us`. `:-)`

Unix Network Programming, volumes 1-2 by W. Richard Stevens. Published by Addison-Wesley Professional and Prentice Hall. ISBNs for volumes 1-2: 978-0131411555[1], 978-0130810816[2].

Internetworking with TCP/IP, volume I by Douglas E. Comer. Published by Pearson. ISBN 978-0136085300[3].

TCP/IP Illustrated, volumes 1-3 by W. Richard Stevens and Gary R. Wright. Published by Addison Wesley. ISBNs for volumes 1, 2, and 3 (and a 3-volume set): 978-0201633467[4], 978-0201633542[5], 978-0201634952[6], (978-0201776317[7]).

TCP/IP Network Administration by Craig Hunt. Published by O'Reilly & Associates, Inc. ISBN 978-0596002978[8].

Advanced Programming in the UNIX Environment by W. Richard Stevens. Published by Addison Wesley. ISBN 978-0321637734[9].

10.2 Web References

On the web:

BSD Sockets: A Quick And Dirty Primer[10] (Unix system programming info, too!)

The Unix Socket FAQ[11]

TCP/IP FAQ[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/

The Winsock FAQ[13]

And here are some relevant Wikipedia pages:

Berkeley Sockets[14]

Internet Protocol (IP)[15]

Transmission Control Protocol (TCP)[16]

User Datagram Protocol (UDP)[17]

Client-Server[18]

Serialization[19] (packing and unpacking data)

10.3 RFCs

RFCs[20]—the real dirt! These are documents that describe assigned numbers, programming APIs, and protocols that are used on the Internet. I've included links to a few of them here for your enjoyment, so grab a bucket of popcorn and put on your thinking cap:

RFC 1[21] —The First RFC; this gives you an idea of what the "Internet" was like just as it was coming to life, and an insight into how it was being designed from the ground up. (This RFC is completely obsolete, obviously!)

RFC 768[22] —The User Datagram Protocol (UDP)

RFC 791[23] —The Internet Protocol (IP)

RFC 793[24] —The Transmission Control Protocol (TCP)

RFC 854[25] —The Telnet Protocol

RFC 959[26] —File Transfer Protocol (FTP)

RFC 1350[27] —The Trivial File Transfer Protocol (TFTP)

RFC 1459[28] —Internet Relay Chat Protocol (IRC)

RFC 1918[29] —Address Allocation for Private Internets

RFC 2131[30] —Dynamic Host Configuration Protocol (DHCP)

RFC 2616[31] —Hypertext Transfer Protocol (HTTP)

RFC 2821[32] —Simple Mail Transfer Protocol (SMTP)

RFC 3330[33] —Special-Use IPv4 Addresses

[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/rfc2616
[32] https://tools.ietf.org/html/rfc2821
[33] https://tools.ietf.org/html/rfc3330

RFC 3493[34] —Basic Socket Interface Extensions for IPv6

RFC 3542[35] —Advanced Sockets Application Program Interface (API) for IPv6

RFC 3849[36] —IPv6 Address Prefix Reserved for Documentation

RFC 3920[37] —Extensible Messaging and Presence Protocol (XMPP)

RFC 3977[38] —Network News Transfer Protocol (NNTP)

RFC 4193[39] —Unique Local IPv6 Unicast Addresses

RFC 4506[40] —External Data Representation Standard (XDR)

The IETF has a nice online tool for searching and browsing RFCs[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/

Index