Guia Beej's Para Programação em Rede

Usando Internet Sockets

Brian “Beej Jorgensen” Hall

Intro

Hey! Programação de Sockets tem lhe deixado para baixo? É muito difícil entender a partir das páginas `man`? Você quer fazer programas legais para Internet mas você não tem tempo para percorrer diversas `structs` tentando descobrir se você precisa chamar `bind()` antes de `connect()`, etc., etc.

Bem, adivinhe! Eu já fiz este trabalho desagradável, e estou morrendo de vontade de compartilhar com todos o que aprendi! Você veio ao lugar certo. Este documento deve dar ao programador C médio competente o impulso que ele precisa para obter controle sobre a confusão que é redes.

E confira: Eu finalmente caminhei para o futuro (apenas no momento certo!) e atualizei o Guia para IPv6! Divirta-se!

Ao público

Este documento foi escrito como um tutorial, não como uma referência completa. Ele provavelmente será mais eficiente quando lido por pessoas que estejam apenas começando com programação de sockets e à procura de uma direção a seguir. Certamente este não é um guia *completo e total* para programação de sockets, de qualquer forma.

Espero, entretanto, que ele seja suficiente para que as páginas de manual comecem a fazer sentido... `:-)`

Plataforma e Compilador

Os códigos contidos neste documento foram compilados em um PC Linux usando compiladores GNU `gcc`. Estes devem, no entanto, ser compilados em praticamente qualquer plataforma que use `gcc`. Naturalmente, isto não se aplica se você está programando para Windows— consulte a seção sobre programação no Windows, abaixo.

Homepage Oficial e livros a venda

Esta é a localização oficial deste documento:

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

Lá você também encontrará códigos de exemplo e traduções do guia em vários idiomas.

Para comprar cópias impressas bem encadernadas (alguns chamam de "livros"), visite:

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

Eu aprecio a compra porque ajuda a sustentar o meu estilo de vida de escritor de documentações.

Nota para programadores Solaris/SunOS

Ao compilar para Solaris ou SunOS, você precisa especificar algumas opções extras na linha de comando para ligar às bibliotecas adequadas. A fim de fazer isso, basta adicionar "`-lnsl -lsocket -lresolv`" ao final do comando de compilação, como isso:

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

Se você ainda receber erros, você pode tentar ainda adicionar `-lxnet` ao final da linha de comando. Eu não sei por que contece, exatamente, mas algumas pessoas parecem precisar.

Outro lugar em que você pode encontrar problemas é na chamada de `setsockopt()`. O protótipo é diferente do que há no meu ambiente Linux, assim em vez de:

```
int yes=1;
```

Digite o seguinte:

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

Como eu não tenho um ambiente Sun, eu não testei qualquer das instruções a cima, é apenas o que as pessoas me disseram por e-mail.

Observação para programadores Windows

Neste ponto do guia, historicamente, eu fiz pouco caso do Windows, simplesmente devido ao fato de que eu não gosto muito. Mas eu realmente devo ser justo e dizer-lhe que o Windows tem uma enorme base de instalações e é, obviamente, perfeitamente um sistema operacional.

Dizem que a ausência do Windows nos torna pessoas melhores, e, neste caso, eu acredito que seja verdade (Ou talvez seja a idade). Mas o que eu posso dizer é que depois de uma década sem usar sistemas operacionais da Microsoft para o meu trabalho pessoal, Eu sou muito mais feliz! Como tal, eu posso sentar e dizer com segurança: "Claro, sinta-se livre para usar Windows!"... Ok, sim, mas isso me faz cerrar os dentes ao dizer.

Então, eu ainda o encorajo a experimentar Linux [1], BSD[2], ou algum sabor de Unix, em vez disso.

Como as pessoas gostam do que gostam, o pessoal do Windows ficará satisfeito em saber que essas informações são geralmente também aplicáveis ao Windows, com algumas pequenas alterações, se houverem.

Uma coisa legal que você pode fazer é instalar Cygwin [3], que é um conjunto de ferramentas Unix para Windows. Ouvi dizer que, ao fazer isso, todos esses programas são compilados sem modificações.

Outra coisa que você deve considerar é o Subsistema Windows para Linux [4]. Basicamente, isso permite que você instale coias como uma VM-ish Linux no Windows 10. Isso também definitivamente o deixará situado.

Mas alguns de vocês podem querer fazer as coisas na forma puramente Windows. Isso é muito corajoso de sua parte, e é isso que você tem que fazer: correr e pegar um Unix imediatamente! Não, não, estou brincando. Eu echo que sou mais Windows-friendly(er) hoje em dia...

Isto é o que você terá que fazer (a menos que você instale Cygwin!): Em primeiro lugar, ignore praticamente todos os arquivos de cabeçalho do sistema que menciono aqui. Tudo que você precisa incluir é:

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

Espere! Você também precisa fazer uma chamada a `WSAStartup()` antes de fazer qualquer outra coisa com a biblioteca de sockets. O código para isso é algo como:

```
#include <winsock.h>

{
    WSADATA wsaData;   // se isso não funcionar
    //WSAData wsaData; // tente isso em vez de

    // MAKEWORD(1,1) para Winsock 1.1, MAKEWORD(2,0)
 para Winsock 2.0:

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

Você também precisa dizer ao seu compilador para vincular a biblioteca Winsock, usualmente chamada `wsock32.lib` ou `winsock32.lib`, ou `ws2_32.lib` para Winsock 2.0. Em VC++, isso pode ser feito através do menu `Project`, em `Settings...` Clique na guia `Link`, e procure a caixa intitulada "Object/library modules". Adicione "wsock32.lib" (ou qualquer lib de sua preferência) para a lista.

Ou foi pelo menos assim que ouvi.

Finalmente, você precisa chamar `WSACleanup()` quando terminar o uso das bibliotecas de sockets. Consulte a ajuda online para obter detalhes.

Depois de fazer isso, o resto dos exemplos neste tutorial devem no geral se aplicar, com algumas exceções. Por um lado, você não pode usar `close()` para fechar um socket—você precisa usar `closesocket()`, em vez disso. Além disso, `select()` só funciona com descritores de sockets, não descritores de arquivos (como `0` para `stdin`).

Há também uma classe socket que você pode usar, `CSocket`. Verifique as páginas de ajuda de seu compilador para mais informações.

Para mais informações sobre Winsock, leia o Winsock FAQ [5] e inicie por lá.

Finalmente, ouvi dizer que o Windows não possui a chamada de sistema `fork()`, que é, infelizmente, utilizada em alguns dos meus exemplos. Talvez você tenha que usar uma biblioteca POSIX ou algo para fazê-lo funcionar, ou você pode usar `CreateProcess()` em seu lugar. `fork()` não tem argumentos, e `CreateProcess()` leva cerca de 48 bilhões argumentos. Se você não está à altura, a `CreateThread()` é um pouco mais fácil de digerir... Infelizmente uma discussão sobre multithreading está além do escopo deste documento. Eu só posso falar um tanto sobre, você sabe!

Política Email

Eu geralmente estou disponível a ajudar com perguntas por e-mail, então fique à vontade para escrever, mas não posso garantir uma resposta. Eu levo uma vida muito ocupada e há momentos em que simplesmente não consigo responder a uma dúvida que você possua. Quando esse é o caso, normalmente apenas excluo a mensagem. Não é nada pessoal; Eu só nunca terei tempo para dar a resposta detalhada que você precisa.

Como regra geral, quanto mais complexa a questão, é menos provável que eu responda. Se você puder refinar sua pergunta antes de enviá-la e incluir todas as informações pertinentes (como plataforma, compilador, mensagens de erro recebidas e qualquer outra coisa que possa ajudar a solucionar os problemas), é muito mais provável que você receba uma resposta. Para mais dicas, leia o documento do ESR, Como fazer perguntas de maneira inteligente [6].

Se você não obtiver uma resposta, faça um pouco mais, tente encontrar a resposta e, se ainda não for suficiente, escreva-me novamente com as informações que encontrou e espero que seja o suficiente para eu o ajudar.

Agora que te atormentei sobre como escrever e não me escrever, gostaria apenas de lhe dizer que aprecio *plenamente* todos os elogios que o guia recebeu ao longo dos anos. É um verdadeiro impulso moral, e fico feliz em saber que está sendo usado para o bem! `:-)` Obrigado!

Mirroring

Você é mais que bem-vindo para espelhar este site, seja pública ou privadamente. Se você espelhar publicamente o site e quiser que eu faça um link para ele a partir da página principal, me envie uma linha em `beej@beej.us`.

Nota para Tradutores

Se você quiser traduzir o guia para outra língua, escreva-me em beej@beej.us e eu adicionarei uma ligação para sua tradução a partir da página principal. Sinta-se livre para adicionar seu nome e informações de contato à tradução.

Este documento markdown fonte usa codificação UTF-8.

Por favor, observe as restrições de licença na seção Direitos autorais, distribuição e informações legais, abaixo.

Caso queira que eu hospede a tradução, basta pedir. Eu também adicionarei uma ligação para ela caso você a hospede; de qualquer forma, tudo bem.

Direitos autorais, distribuição e informações legais

Dedicatória

Obrigado a todos que me ajudaram no passado e no futuro escrever este guia. E obrigado a todas as pessoas que produzem o software livre e os pacotes que eu uso para criar o Guia: GNU, Linux, Slackware, vim, Python, Inkscape, pandoc, muitos outros. E, finalmente, um grande obrigado aos milhares de vocês que me escreveram com sugestões de melhorias e palavras de encorajamento.

Dedico este guia a alguns dos meus maiores heróis e inspiradores no mundo dos computadores: Donald Knuth, Bruce Schneier, W. Richard Stevens e The Woz, meus leitores, e toda a comunidade de software livre e de código aberto.

Publicando Informações

Este livro foi escrito em Markdown usando o editor vim em um ambiente Arch Linux carregado com ferramentas GNU. A capa “art” e diagramas são produzidos com o Inkscape. O Markdown é convertido em HTML e LaTex/PDF pelo Python, Pandoc e XeLaTeX, usando fontes Liberation. O toolchain é composto 100% por software livre e de código aberto.

O que é um socket?

Você ouve falar em "sockets" o tempo todo, e talvez esteja se perguntando o que são exatamente. Bem, eles são isso: Um modo de se comunicar com outros programas utilizando descritores de arquivos padrão do Unix.

O quê?

Ok, você pode ter ouvido algum hacker de Unix dizer "Ei, *tudo* no Unix é arquivo!". O que essa pessoa disse é o fato de que quando os programas Unix fazem qualquer tipo de E/S, eles fazem isso pela leitura ou escrita em um descritor de arquivo. Um descritor de arquivo é simplesmente um inteiro associado a um arquivo aberto. Mas (e aqui está o segredo) esse arquivo pode ser uma conexão de rede, um FIFO, um pipe, um terminal, um arquivo real no disco, ou qualquer outra coisa. Tudo no Unix *é* um arquivo! Então, quando você quiser se comunicar com outro programa através da Internet você vai fazê-lo através de um descritor de arquivo, é melhor você acreditar.

"Onde posso obter este descritor de arquivo para comunicação de rede, Sr. Sabichão?", É provavelmente a última pergunta em sua mente agora, mas eu vou responder a isso de qualquer maneira: Você faz uma chamada a função `socket()`. Ela retorna o descritor de socket, e você se comunica através dele utilizando as chamadas de sistema especializadas em sockets `send()` e `recv()` (`man send`, `man recv`).

"Mas, Ei!" você pode estar exclamando agora. "Se é um descritor de arquivo, por que, em nome de Netuno, não posso simplesmente usar as chamadas normais `read()` e `write()` para me comunicar através do socket?" A resposta curta é: "Você pode!"; A resposta mais longa é: "Você pode, mas `send()` e `recv()` oferecem muito mais controle sobre a sua transmissão de dados."

O que vem depois? Veja mais: existem diversos tipos de sockets. Como DARPA Internet addresses (Internet Sockets), path names em um nó local (Unix Sockets), CCITT X.25 addresses (Sockets tipo X.25 podem ser ignorados traquilamente de forma segura), e

provavelmente muitos outros, dependendo do sabor Unix executado. Este documento trata somente do primeiro: Internet Sockets.

Dois tipos de Internet Sockets

Como é? Existem dois tipos de Internet sockets? Sim. Bem, não, estou mentindo. Há mais tipos mas eu não quero te assustar. Eu só comentarei sobre dois tipos aqui. Exceto por esta frase, onde eu vou dizer-lhe que "Raw Sockets" também são muito poderosos e você deveria procurá-los.

Tudo bem, agora. Quais são os dois tipos? Um deles é "Stream Socket"; O outro é "Datagram Socket", que daqui em diante podem ser referidos como "`SOCK_STREAM`" e "`SOCK_DGRAM`", respectivamente. Sockets Datagram são às vezes chamados de "sockets sem conexão" (Embora possam ser usados com `connect()` caso você realmente queira. Veja `connect()`, abaixo).

Sockets stream são fluxos de comunicação confiáveis ligados bidirecionalmente. Se você envia dois itens ao socket na ordem "1,2", eles chegarão na ordem "1,2" na extremidade oposta. Eles também serão livres de erros. Eu estou tão certo, na verdade, que eles estarão livres de erros, que estou pondo agora meus dedos em meus ouvidos e cantando *lá lá lá lá* antes que alguém tente dizer o contrário.

O que usa sockets stream? Bem, você pode já ter ouvido falarem da aplicação `telnet`, sim? Ela usa sockets stream. Todos os caracteres que você digita precisam chegar na mesma ordem que você digitou, certo? Além disso, os navegadores web usam o protocolo HTTP que utiliza sockets stream para obter as páginas. De fato, se você executa telnet contra um site na porta 80, e digita "`GET / HTTP/1.0`" e pressiona RETURN duas vezes, ele retornará o HTML a você!

> Se você não possui o `telnet` instalado e não quer o instalar, ou o seu `telnet` está sendo exigente em conectar como cliente, o guia vem com um programa `telnet`-like chamado `telnot` [7]. Isso deve funcionar bem para todas as necessidades do guia. (Observe que o telnet é na realidade uma especificação de protocolo de rede [8], e `telnot` não o implementa completamente.)

Como sockets stream conseguem atingir um nível tão elevado de qualidade de transmissão de dados? Ele usa um protocolo chamado "The Transmission Control Protocol", também conhecido como "TCP" (veja RFC 793 [9] para informações extremamente detalhadas sobre TCP). O TCP garante que seus dados cheguem sequencialmente e livres de erros. Você pode ter ouvido "TCP" antes como a melhor parte do "TCP/IP", onde "IP" significa "Internet Protocol" (veja RFC 791 [10]). O IP lida principalmente com o roteamento da Internet e geralmente não é responsável pela integridade dos dados.

Legal. E quanto aos sockets datagram? Por que eles são chamados sem conexão? Qual é o problema aqui, afinal? Por que eles não são confiáveis? Bem, aqui estão alguns fatos: se você enviar um datagram, ele pode chegar. Pode chegar fora de ordem. Se chegar, os dados dentro do pacote estarão livres de erros.

Sockets datagram também usam IP para roteamento, mas eles não usam TCP; eles usam o "User Datagram Protocol", ou "UDP" (veja RFC 768 [11]).

Por que eles são sem conexão? Bem, basicamente, é porque você não precisa manter uma conexão aberta como você faz com sockets stream. Você apenas constrói um pacote, coloca um cabeçalho IP nele com informações de destino e envia-o para fora. Nenhuma conexão é necessária. Eles geralmente são usados quando uma pilha TCP não está disponível ou quando alguns pacotes descartados aqui e ali não significam o fim do Universo. Aplicações de exemplo: `tftp` (trivial file transfer protocol, um irmão mais novo do FTP), `dhcpcd` (um cliente DHCP), jogos multiplayer, streaming de áudio, video conferência, etc.

"Espere um minuto! `tftp` e `dhcpcd` são usadospara transferir dados e também aplicações binárias de um host a outro! Os dados não podem ser perdidos se você espera que o aplicativo funcione ao chegar! Que tipo de magia negra é essa?"

Bem, meu amigo humano, `tftp` e outros programas semelhantes têm seus próprios protocolos em cima do UDP. Por exemplo, o protocolo tftp diz que para cada pacote que é enviado, o destinatário tem de enviar de volta um pacote que diz: "Eu consegui!" (Um pacote "ACK".). Se o remetente do pacote original não recebe a resposta, digamos, em cinco segundos, ele retransmitirá o pacote até que finalmente receba um ACK. Este procedimento de reconhecimento é muito importante na implementação confiável do `SOCK_DGRAM` em aplicações.

Para aplicações não confiáveis, como jogos, áudio ou vídeo, você pode só ignorar os pacotes perdidos, ou talvez tentar compensá-los de forma inteligente (Jogadores de Quake conhecem bem a manifestação deste efeito pelo termo técnico: *lag maldito*. A palavra "maldito", neste caso, representa qualquer enunciado extremamente profano).

Por que você usaria um protocolo subjacente não confiável? Duas razões: Velocidade e velocidade. É a maneira mais rápida de mirar-e-atirar do que seria para se certificar de que um dado chegou em segurança e ainda ter certeza que o mesmo está na ordem correta. Se você estiver enviando mensagens de bate-papo, o TCP é ótimo; Se você estiver enviando 40 atualizações posicionais por segundo para jogadores ao redor do mundo, talvez não importe muito perder um ou dois pacotes e o UDP seja uma boa escolha.

Baixo nível nonsense e Teoria de Rede

Agora que acabei de mencionar a divisão de camadas de protocolos, é hora de falar sobre como as redes realmente funcionam, e mostrar alguns exemplos de como pacotes `SOCK_DGRAM` são construídos. Na prática, você provavelmente pode pular esta seção. É uma boa teoria de background, no entanto.

Encapsulamento de dados.

Ei, crianças, é hora de aprender sobre *Encapsulamento de Dados*! Isto é muito, muito importante. É tão importante que para que você possa aprender sobre é necessário fazer o curso de redes aqui na Chico State `;-)`. Basicamente, ele diz o seguinte: um pacote nasce, é envolto ("encapsulado") em um cabeçalho (e raramente um rodapé) pelo primeiro protocolo (por exemplo, o protocolo TFTP), o conjunto da coisa (com cabeçalho TFTP já incluído) é encapsulado novamente pelo seguinte protocolo (digamos, UDP), em seguida, novamente pelo

próximo (IP), em seguida, novamente pelo protocolo final sobre a camada de hardware (física) (por exemplo, Ethernet).

Quando outro computador recebe o pacote, o hardware retira o cabeçalho Ethernet, o kernel retira os cabeçalhos IP e UDP, o programa TFTP retira o cabeçalho TFTP, e finalmente você tem os dados.

Agora posso finalmente falar sobre o infame *Modelo de Rede em Camadas* (conhecido como "ISO/OSI"). Este modelo de rede descreve um sistema de funcionalidades de rede que tem muitas vantagens sobre outros modelos. Por exemplo, você pode escrever programas com sockets que são exatamente os mesmos sem se importar com a forma como os dados são transmitidos fisicamente (serial, thin Ethernet, AUI, o que for) porque programas em níveis mais baixos lidam com isso para você. O hardware e a topologia de rede reais são transparentes para o programador do socket.

Sem mais delongas, apresentarei as camadas do modelo de forma completa. Lembre-se disso para seus exames do curso de redes:

- Aplicação
- Apresentação
- Sessão
- Transporte
- Rede
- Enlace
- Física

A camada física é o hardware (serial, Ethernet, etc.). A camada de aplicação é quase tão distante da camada física quanto você possa imaginar—é onde os usuários interagem com a rede.

Agora, este modelo é tão geral que você provavelmente poderia usá-lo como um guia de reparação de automóveis, se você realmente quiser. Um modelo em camadas mais consistente com Unix poderia ser:

- Camada de Aplicação (*telnet, ftp, etc.*)
- Camada de Transporte Host-para-Host (*TCP, UDP*)
- Camada de Internet (*IP e roteamento*)
- Camada de Acesso à Rede (*Ethernet, Wi-Fi, ou outros*)

Neste momento, você provavelmente já pode ver como essas camadas correspondem ao encapsulamento dos dados originais.

Viu quanto trabalho há na construção de um simples pacote? Eita! E você só precisa digitar seus cabeçalhos utilizando "`cat`"! Estou brincando. Tudo o que você precisa fazer para sockets stream é usar `send()` para enviar os dados. E tudo o que você precisa fazer para sockets datagram é encapsular o pacote no método de sua escolher e usar `sendto()`. O kernel constrói a Camada de Transporte e a Camada de Internet para você e o hardware faz a Camada de Acesso à Rede. Ah, a tecnologia moderna.

Assim termina nossa breve incursão à teoria de rede. Ah, sim, eu esqueci de lhe dizer tudo o que eu queria dizer sobre roteamento: nada! Isso mesmo, eu não falarei sobre isso. O roteador abre o cabeçalho IP do pacote, consulta sua tabela de roteamento, *blá-blá-blá*. Confira o IP RFC

[12] caso realmente se importe em saber. E se você nunca souber, bem, você ainda estará vivo.

Endereços IP, `structs`, e Data Munging

Aqui, para variar, é a parte do jogo em que podemos falar sobre código.

Mas, primeiro, vamos discutir mais não-códigos! Sim! Primeiro quero falar um pouco sobre endereços IP e portas e então teremos o assunto resolvido. Em seguida, falaremos sobre como a API de sockets armazena e manipula os endereços IP e outros dados.

Endereços IP, versões 4 e 6

Nos bons e velhos tempos, quando Ben Kenobi ainda era chamado Obi Wan Kenobi, havia um maravilhoso sistema de roteamento de rede chamado The Internet Protocol Version 4, também chamado IPv4. Ele possuia endereços compostos de quatro bytes (ou quatro "octetos"), e era escrito comumente na forma de "pontos e números", assim: `192.0.2.111`.

Você provavelmente já viu isso por aí.

Na verdade, até o momento desta escrita, praticamente todos os sites da Internet usam IPv4.

Todos, incluindo Obi Wan, estavam felizes. As coisas eram ótimas, até que algum opositor com o nome de Vint Cerf avisou que estávamos prestes a ficar sem endereços IPv4!

(Além de avisar a todos sobre o futuro destino apocalíptico de tristeza do IPv4, Vint Cerf [13] também é conhecido por ser o Pai da Internet. Então, eu realmente não estou em condições de o julgar.)

Ficar sem endereços? Como isso poderia acontecer? Quer dizer, existem bilhões de endereços IP em um endereço IPv4 de 32 bits. Nós realmente temos bilhões de computadores por aí?

Sim.

Além disso, no início, quando havia apenas alguns computadores e todos pensavam que um bilhão era um número incrivelmente grande, algumas grandes organizações receberam generosamente milhões de endereços IP para uso próprio (Como Xerox, MIT, Ford, HP, IBM, GE, AT&T e uma pequena empresa chamada Apple, para citar alguns.).

Na verdade, se não fosse por várias medidas paliativas, já teríamos os esgotado há muito tempo.

Mas agora estamos vivendo em uma era em que todos os seres humanos têm um endereço IP, todos os computadores, todas as calculadoras, todos os telefones, todos os parquímetros e (por que não) todos os filhotes de cachorros também.

E assim o IPv6 nasceu. Como Vint Cerf é provavelmente imortal (mesmo que sua forma física passe, Deus nos livre, ele provavelmente já está existindo como uma espécie hiper-inteligente do programa ELIZA [14] nas profundezas do Internet2), ninguém quer ter de ouvi-lo dizer novamente "eu avisei" se não tivermos endereços suficientes na próxima versão do Internet Protocol.

O que isso lhe sugere?

Que precisamos de *muito* mais endereços. Que não precisamos apenas de duas vezes mais de endereços, nem um bilhão de vezes mais, nem mil trilhões de vezes mais, mas *79 MILHÕES BILHÕES TRILHÕES de vezes mais endereços possíveis!* Isso vai mostrar a eles!

Você está dizendo: "Beej, isso é verdade? Eu tenho todos os motivos para descrer de grandes números." Bem, as diferenças entre 32 bits e 128 bits podem não parecer grandes; são apenas mais 96 bits, certo? Mas lembre-se, estamos falando de poderes aqui: 32 bits representam cerca de 4 bilhões de números (2^{32}), enquanto 128 bits representam cerca de 340 trilhões de trilhões de trilhões de números (na verdade, 2^{128}). Isso é como um milhão de Internets IPv4 para *cada estrela no Universo*.

Esqueça a aparência destes pontos-e-números do IPv4, também; agora temos uma representação hexadecimal, com cada bloco de dois bytes separados por dois pontos, como isso:

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

Isso não é tudo! Muitas vezes, você terá um endereço IP com muitos zeros e poderá compactá-los entre dois-pontos. E você pode deixar zeros à esquerda para cada par de bytes. Por exemplo, cada um desses pares de endereços é equivalente:

```
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
```

O endereço `::1` é o *endereço de auto-retorno*. Isto sempre significa "esta máquina que estou executando agora". Em IPv4, o endereço de auto-retorno é `127.0.0.1`.

Finalmente, há um modo de compatibilidade IPv4 para endereços IPv6 com o qual você pode se deparar. Se você quiser, por exemplo, para representar o IPv4 `192.0.2.33` como um endereço IPv6, você usaria a seguinte notação: "`::ffff:192.0.2.33`"

Estamos falando de diversão séria.

Na verdade, é tão divertido, os criadores do IPv6 gentilmente mativeram trilhões e trilhões de endereços para uso reservado, mas temos tantos, francamente, quem ainda está contando? Há muito de sobra para cada homem, mulher, criança, cachorrinho e parquímetro em todos os planetas da galáxia. E acredite em mim, cada planeta da galáxia possui parquímetros. Você sabe que é verdade.

Subnets

Por motivos organizacionais, às vezes é conveniente declarar que "Desta primeira parte do endereço IP até este bit é a *parte da rede* do endereço IP, e o restante é a *parte do host*".

Por exemplo, com IPv4, você pode ter `192.0.2.12`, e poderíamos dizer que os três primeiros campos são a rede e o último o endereço do host. Ou, dito de outra forma, estamos falando do host 12 na rede `192.0.2.0` (veja como podemos zerar o byte de endereço do host.)

E agora, para mais informações desatualizadas! Pronto? Nos tempos antigos, houveram "classes" de sub-redes, onde o primeiro, segundo, ou terceiro bytes do endereço formavam a parte de rede. Se você tivesse a sorte de ter um byte para a rede e três para o host, você poderia ter 24 bits de hosts na sua rede (16 milhões ou mais). Essa era uma rede "Classe A". No extremo oposto, havia uma "Classe C", com três bytes de rede e um byte de host (256 hosts, menos uma dupla que estavam reservados).

Então, como você pode ver, havia apenas alguns Classe A, uma pilha enorme de Classe C e alguns Classe B no meio.

A porção da rede do endereço IP é descrita por algo chamado `netmask`, onde você realiza uma operação bitwise-AND com o endereço IP para obter o número da rede. A netmask geralmente tem um formato parecido com `255.255.255.0`. (Por exemplo, com essa netmask, se o seu IP é `192.0.2.12`, então sua rede é `192.0.2.12` AND `255.255.255.0` o que gera `192.0.2.0`.)

Infelizmente, descobriu-se que isso não era suficiente para as eventuais necessidades da Internet; nós estávamos ficando sem classes C rapidamente, e nós estávamos definitivamente fora das Classes A, então nem se incomode em perguntar. Para remediar isso, os números permitidos para a netmask são combinados arbitrários de bits, não apenas 8, 16 ou 24. Então você pode ter uma netmask de, digamos, `255.255.255.252`, que é de 30 bits de rede e 2 bits de host, permitindo quatro hosts na rede. (Note que a netmask é *SEMPRE* um monte de bits 1 seguidos por um monte de bits 0.)

Mas é um pouco difícil usar uma grande série de números como `255.192.0.0` como netmask. Primeiro de tudo, as pessoas não têm uma ideia intuitiva da quantidade de bits, e em segundo lugar, não é realmente compacta. Então, o Novo Estilo surgiu e é muito melhor. Você apenas precisa colocar uma barra após o endereço IP e, em seguida, o número de bits de rede em decimal. Assim: `192.0.2.12/30`.

Ou, para IPv6, algo como isso: `2001:db8::/32` ou `2001:db8:5413:4028::9db9/64`.

Números de Porta

Se você se lembra, apresentamos anteriormente o Modelo de rede em camadas que possuia a camada de Internet (IP) separada da Camada de Transporte Host-para-Host (TCP e UDP). Atenção a isso antes do próximo parágrafo.

Acontece que, além de um endereço IP (usado pela camada IP), há outro endereço usado pelo TCP (sockets stream) e, coincidentemente, pelo UDP (sockets datagram). É o *número de porta*. Ele é um número de 16-bit que é como o endereço local da conexão.

Pense no endereço IP como o endereço de um hotel e o endereço da porta como o número do quarto. Essa é uma analogia decente; talvez mais tarde eu venha com uma envolvendo a

indústria automobilística.

Digamos que você queira ter um computador que lide com e-mails recebidos E serviços web—como você diferencia os dois em um único computador com um único endereço IP?

Bem, serviços diferentes na Internet têm diferentes números de porta bem conhecidos. Você pode vê-los todos em the Big IANA Port List [15] ou, se você estiver em um ambiente Unix, em seu arquivo `/etc/services`. O HTTP (a web) usa a porta 80, o telnet usa a porta 23, o SMTP a porta 25, o jogo DOOM [16] usa a porta 666, etc. E assim por diante. Portas abaixo de 1024 são frequentemente consideradas especiais, e geralmente exigem privilégios especiais do Sistema Operacional para seu uso.

E é isso!

Byte Order

Por ordem do rei! Haverá duas ordenações de bytes, a seguir conhecidas como ótima e péssima!

Brincadeira, mas uma é realmente melhor do que a outra. `:-)`

Não há uma maneira fácil de dizer isso, então só vou deixar escapar: seu computador pode estar armazenando bytes em ordem inversa bem abaixo do seu nariz. Eu sei! Ninguém queria te dizer.

O fato é que todos no mundo da Internet geralmente concordam que se você quiser representar um número hexadecimal de dois bytes, digamos `b34f`, você poderá armazená-lo em dois bytes sequenciais b3 seguido de `4f`. Faz sentido, e, como Wilford Brimley [17] diria, é a coisa certa a fazer. Esse número, armazenado com a parte mais significativa primeiro, é chamado *Big-Endian*.

Infelizmente, *alguns* computadores espalhados aqui e ali ao longo o mundo, ou seja, qualquer coisa com um processador Intel ou compatível com Intel, armazena os bytes de forma invertida, de modo que `b34f` seria armazenado na memória como os bytes sequenciais `4f` seguido de `b3`. Este método de armazenamento é chamado *Little-Endian*.

Mas espere, ainda não terminei com a terminologia! O *Big-Endian*, o mais sensato, também é chamado de *Network Byte Order* porque essa é a ordem em que os tipos de rede funcionam.

Seu computador salva números em *Host Byte Order*. Se ele é um Intel 80x86, o Host Byte Order é Little-Endian. Se é um Motorola 64k, o Host Byte Order é Big-Endian. Se é um PowerPC, o Host Byte Order é..., isso depende!

Muitas vezes, quando você cria pacotes ou preenche estruturas de dados você precisa se certificar de que seus números de dois e quatro bytes estão em Network Byte Order. Mas como você pode fazer isso se você não conhece o Host Byte Order nativo?

Boas notícias! Você acabou de supor que o Host Byte Order não está correto, e você sempre passa os valore através de uma função para configurá-los para Network Byte Order. A função fará a conversão mágica se for necessário e ,desta forma, o seu código torna-se portátil entre máquinas com endianness diferentes.

Tudo certo. Existem dois tipos de números que você pode converter: `short` (dois bytes) e `long` (quatro bytes). Essas funções funcionam para variações de `unsigned`. Digamos que você queira converter um `short` de Host Byte Order para Network Byte Order. Comece com "h" para "host", siga com "to", depois, "n" para "rede" e "s" para "short": h-to-n-s, ou `htons()` (leia: "Host to Network Short").

É quase fácil demais...

Você pode usar todas as combinações de "n", "h", "s" e "l" desejadas, sem contar as realmente estúpidas. Por exemplo, NÃO há função `stolh()` ("Short to Long Host")—não aqui, de qualquer forma. Mas há:

Funcão	Descrição
`htons()`	host `to` network short
`htonl()`	host `to` network `long`
`ntohs()`	network `to` host short
`ntohl()`	network `to` host `long`

Basicamente, você precisará converter os números para Network Byte Order antes de saírem pelo fio e convertê-los de volta a Host Byte Order quando recebidos pelo fio.

Eu não conheço sobre a variante de 64-bit, desculpe. E se você quiser fazer com ponto flutuante, confira a seção sobre Serialização, bem abaixo.

Assuma que os números neste documento estejam em Host Byte Order, a menos que eu diga o contrário.

`structs`

Bem, finalmente estamos aqui. É hora de falar sobre programação. Nesta seção, cobrirei vários tipos de dados usados pelas interfaces de sockets, uma vez que alguns deles são verdadeiros mistérios a se descobrir.

Primeiro, o mais fácil: o descritor de socket. Um descritor de socket é do seguinte tipo:

```
int
```

Apenas um `int` regular.

As coisas ficam estranhas a partir daqui, então apenas leia comigo e acredite.

Minha Primeira Struct™—`struct addrinfo`. Essa estrutura é uma invenção mais recente e é usada para preparar as estruturas de endereço do socket para uso posterior. Também é usada em pesquisas de nome de host e pesquisas de nome de serviço. Isso fará mais sentido mais tarde quando chegarmos ao seu uso real, mas saberemos por enquanto que é uma das primeiras coisas que você ligará ao fazer uma conexão.

```
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;  // use 0 para "qualquer"
    size_t           ai_addrlen;   // tamanho de ai_addr em
    bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in ou _in6
    char            *ai_canonname; // nome de host canônico e
    completo

    struct addrinfo *ai_next;      // lista ligada, próximo nó
};
```

Você carregará essa struct rapidamente, e depois chamará `getaddrinfo()`. Ela retornará um ponteiro para uma nova lista ligada dessa estrutura preenchida com todos os itens necessários.

Você pode forçá-la a usar IPv4 ou IPv6 no campo `ai_family`, ou deixá-lo como `AF_UNSPEC` para usar qualquer um. Isto é legal porque o seu código pode ser funcional com qualquer versão IP.

Note que esta é uma lista ligada: `ai_next` aponta para o próximo elemento—pode haver vários resultados para você escolher. Eu usaria o primeiro resultado que funcionasse, mas você poderia ter necessidades de negócios diferentes; Eu não sei tudo, man!

Você verá que o campo `ai_addr` na `struct addrinfo` é um ponteiro para uma `struct sockaddr`. É ai que começamos a entrar nos detalhes básicos do que está dentro de uma struct de endereço IP.

Você não precisa escrever a essas estruturas usualmente; muitas vezes, uma chamada a `getaddrinfo()` para preencher a sua `struct addrinfo` é tudo o que você precisa. Você *terá*, no entanto, que espiar dentro destas `structs` para obter seus valores de retorno, então vou aprensentá-los aqui.

(Além disso, todo o código escrito antes de `struct addrinfo` ser inventada eram embalados à mão, então você verá um monte de códigos IPv4 em estado bruto que fazem exatamente isso. Você sabe, em versões antigas deste guia e assim por diante.)

Algumas `structs` são IPv4, algumas são IPv6, e algumas são ambas. Farei anotações de quais são o que.

De qualquer forma, a `struct sockaddr` detém informações de endereço de socket para muitos tipos de sockets.

```
struct sockaddr {
```

```
    unsigned short    sa_family;    // família de endereços,
    AF_xxx
    char              sa_data[14];  // 14 bytes de endereço do
    protocolo
};
```

`sa_family` pode ser uma variedade de coisas, mas será `AF_INET` (IPv4) ou `AF_INET6` (IPv6) para tudo o que fizermos neste documento. `sa_data` contém um endereço de destino e o número da porta para o socket. Isto é bastante complicado, pois você não quer embalar tediosamente o endereço no `sa_data` manualmente.

Para lidar com `struct sockaddr`, os programadores criaram uma estrutura paralela: `struct sockaddr_in` ("in" para "Internet") para uso com IPv4.

E *esta é a parte importante*: um ponteiro para uma `struct sockaddr_in` pode ser convertido para um ponteiro para uma `struct sockaddr` e vice-versa. Assim, mesmo que `connect()` procure uma `struct sockaddr*`, você ainda pode utilizar uma `struct sockaddr_in` e converte-lá no último minuto!

```
// (IPv4 somente--veja struct sockaddr_in6 para IPv6)

struct sockaddr_in {
    short int          sin_family;  // Família de endereços,
    AF_INET
    unsigned short int sin_port;    // Número de Porta
    struct in_addr     sin_addr;    // Endereço Internet
    unsigned char      sin_zero[8]; // Mesmo tamanho que struct
    sockaddr
};
```

Esta estrutura facilita a referência de elementos do endereço do socket. Note que `sin_zero` (que é incluído para preencher a estrutura com o comprimento de uma `struct sockaddr`) deve ser todo definido para zero com a função `memset()`. Além disso, observe que `sin_family` corresponde a `sa_family` em uma `struct sockaddr` e deve ser configurada para `"AF_INET"`. Finalmente, o `sin_port` deve estar em *Network Byte Order* (usando `htons()`!)

Vamos cavar mais fundo! Você vê que o campo `sin_addr` é uma `struct in_addr`. Que coisa é essa? Bem, não para ser excessivamente dramático, mas é uma das mais assustadoras unions de todos os tempos:

```
// (IPv4 somente--veja struct in6_addr para IPv6)

// Endereço Internet (uma estrutura por razões históricas)
struct in_addr {
```

```
    uint32_t s_addr; // é um int de 32 bits (4 bytes)
};
```

Uau! Bem, isso *era usado* para ser uma união, mas agora aqueles dias parecem ter desaparecido. Boa viagem. Então, se você declarou `ina` para ser do tipo `struct sockaddr_in`, então `ina.sin_addr.s_addr` faz referência ao endereço IP de 4 bytes (Network Byte Order). Observe que, mesmo que o seu sistema ainda use a terrível union em `struct em_addr`, você ainda pode referenciar o endereço IP de 4 bytes exatamente da mesma maneira que eu fiz acima (isto devido aos `#defines`.)

E sobre IPv6? Existem `structs` similares para ele, assim:

```
// (IPv6 somente--veja struct sockaddr_in e struct in_addr para
    IPv4)

struct sockaddr_in6 {
    u_int16_t       sin6_family;   // família de endereços,
    AF_INET6
    u_int16_t       sin6_port;     // número da porta, Network
    Byte Order
    u_int32_t       sin6_flowinfo; // informação de fluxo IPv6
    struct in6_addr sin6_addr;     // endereço IPv6
    u_int32_t       sin6_scope_id; // ID do escopo
};

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

Observe que o IPv6 tem um endereço IPv6 e um número de porta, assim como IPv4 tem um endereço IPv4 e um número de porta.

Além disso, note que eu não estou começando a falar sobre as informações de fluxo IPv6 ou campos de identificação de escopo neste momento... este é apenas um guia de iniciação. `:-)`

Por último, mas não menos importante, aqui está outra estrutura simples, `struct sockaddr_storage` que é projetada para ser grande o suficiente para manter ambas as estruturas IPv4 e IPv6. Veja, para algumas chamadas, às vezes você não sabe com antecedência se vai preencher sua `struct sockaddr` com um endereço IPv4 ou IPv6. Assim você passa por esta estrutura paralela, muito semelhante à `struct sockaddr`, exceto maior, e depois a converte para o tipo que você precisa:

```
struct sockaddr_storage {
    sa_family_t  ss_family;     // família de endereços
```

```
    // tudo isso é preenchimento, implementação específica,
    ignore:
    char      __ss_pad1[_SS_PAD1SIZE];
    int64_t   __ss_align;
    char      __ss_pad2[_SS_PAD2SIZE];
};
```

O que é importante é que você pode ver a família de endereços no campo `ss_family` —verifique isso para ver se é `AF_INET` ou `AF_INET6` (para IPv4 ou IPv6). Então você pode converter para uma `struct sockaddr_in` ou `struct sockaddr_in6` se você quiser.

Endereços IP, Parte Dois

Felizmente para você, há diversas funções que lhe permitem manipular endereços IP. Não há necessidade de os descobrir à mão e enchê-los por um `long`o tempo usando o operador `<<`.

Primeiro, digamos que você tenha uma `struct sockaddr_in ina`, e você tem um endereço IP "`10.12.110.57`" ou "`2001:db8:63b3:1::3490`" que você deseja armazenar nela. A função que você desejará usar, `inet_pton()`, converte um endereço IP em notação de números e pontos em uma `struct in_addr` ou uma `struct in6_addr` dependendo se você especificar `AF_INET` ou `AF_INET6`. ("pton" significa "presentation to network"—você pode chamar isso de "printable to network" se for mais fácil para lembrar.) A conversão pode ser feita da seguinte forma:

```
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: as velhas maneiras de fazer as coisas utilizado uma função chamada `inet_addr()` ou outra chamada `inet_aton()` são agora obsoletas e não funcionam com o IPv6.)

Agora, o trecho de código acima não é muito robusto porque não há verificação de erros. Veja, `inet_pton()` retorna `-1` em caso de erro, ou 0 se o endereço é confuso. Portanto, para garantir verifique se o resultado é superior a 0 antes de usar!

Tudo bem, agora você pode converter strings de endereços IP em suas representações binárias. E o contrário? E se você tem uma `struct in_addr` e você quiser imprimi-lá em notação de números-e-pontos? (Ou uma `struct in6_addr` que você quer na notação hex-e-pontos). Neste caso, você vai querer usar a função `inet_ntop()` ("ntop" significa "network to presentation"—você pode chamá-la de "network to printable" se for mais fácil de lembrar), assim:

```c
// IPv4:

char ip4[INET_ADDRSTRLEN];  // espaço para manter a string IPv4
struct sockaddr_in sa;      // fingir que isso é carregado com algo

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

printf("O endereço IPv4 é: %s\n", ip4);


// IPv6:

char ip6[INET6_ADDRSTRLEN]; // espaço para manter a string IPv6
struct sockaddr_in6 sa6;    // fingir que isso é carregado com algo

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

printf("O endereço IPv6 é: %s\n", ip6);
```

Quando você executa, você passa o tipo de endereço (IPv4 ou IPv6), o endereço, um ponteiro para uma string que manterá o resultado e o máximo comprimento dessa string. (Duas macros seguram convenientemente o tamanho da grande string de endereço IPv4 ou IPv6: `INET_ADDRSTRLEN` e `INET6_ADDRSTRLEN`.)

(Outra nota rápida a mencionar, mais uma vez a velha maneira de fazer as coisas: a função histórica para fazer esta conversão foi chamada `inet_ntoa()`. Também é obsoleta e não funcionará com IPv6.)

Por fim, essas funções só funcionam com endereços de IP numéricos—elas não farão qualquer pesquisa de servidor de nomes DNS para um nome de host, como "`www.example.com`". Você usará `getaddrinfo()` para fazer isso, como você verá mais tarde.

Redes Privadas (ou desconectadas)

Muitos lugares têm um firewall que oculta a rede local do restante do mundo para sua própria proteção. E muitas vezes, o firewall traduz endereços IP "internos" para "externos" (que todos os outros no mundo conhecem) usando um processo chamado *Network Address Translation*, ou NAT.

Você ainda está ficando nervoso? "Onde ele está indo com todas estas coisas estranhas?"

Bem, relaxe e compre uma bebida não-alcoólica (ou alcoólica) porque, como iniciante, você

não precisa nem se preocupar com o NAT, ele é feito para você de forma transparente. Mas eu queria falar sobre a rede atrás do firewall no caso de você começar a ficar confuso com números de rede que esteja vendo.

Por exemplo, eu tenho um firewall em casa. Eu tenho dois endereços IPv4 estáticos alocados para meu uso pela empresa do DSL, e ainda tenho sete computadores na rede. Como isso é possível? Dois computadores não podem compartilhar um mesmo endereço IP, ou então os dados não saberiam para qual deles se destinam!

A resposta é: eles não compartilham os mesmos endereços IP. Eles estão em uma rede privada, com 24 milhões de endereços IP alocados para eles. Eles são todos só para mim. Bem, tudo para mim, tanto quanto para qualquer outra pessoa que esteja preocupada. Aqui está o que está acontecendo:

Se eu fizer login em um computador remoto, ele me informará que estou logado em 192.0.2.33, que é o endereço IP público que meu ISP forneceu para mim. Mas se eu perguntar ao meu computador local qual é seu endereço IP, ele diz 10.0.0.5. Quem está traduzindo o endereço IP de um para o outro? Está certo, o firewall! Está fazendo NAT!

`10.x.x.x` é um dos poucos endereços de rede reservados que só deve ser usado em redes totalmente desconectadas, ou em redes que estão atrás de firewalls. Os detalhes de quais números de redes privadas estão disponíveis para uso estão descritos na RFC 1918 [18], mas alguns comuns que você verá são `10.x.x.x` e `192.168.x.x`, onde `x` é 0-255, geralmente. O menos comum é `172.y.x.x`, onde y varia entre 16 e 31.

Redes atrás de um firewall NAT não *necessitam* estar em uma faixa de IP´s reservados, mas elas geralmente estão.

(Curiosidade! Meu endereço IP externo não é realmente `192.0.2.33`. A rede `192.0.2.x` é reservada para simular um IP "real" em documentações, assim como neste guia! Uau!)

O IPv6 também possui redes privadas. Elas começam com `fdXX:` (ou talvez no futuro `fcXX:`), conforme RFC 4193 [19]. NAT e IPv6 não se misturam geralmente, no entanto (a menos que você esteja fazendo gateway IPv6 para IPv4, o que está além do escopo deste documento)—em teoria você terá tantos endereços à sua disposição que você não precisará usar NAT por muito mais tempo. Mas se você quiser alocar endereços para você em uma rede que não será encaminhada para fora, é assim que se faz.

Saltando de IPv4 para IPv6

Mas eu só quero saber o que mudar no meu código para continuar com o IPv6! Diga-me agora!

Ok! Ok!

Quase tudo aqui já foi dito a cima, mas a versão curta para os impacientes. (É claro que há mais do que isso, mas é isso que se aplica ao guia.)

1. Em primeiro lugar, tente usar `getaddrinfo()` para obter todas as informações sobre `struct sockaddr`, em vez de embalar a estrutura manualmente. Isto irá mantê-la compatível entre versões de IP, e eliminará muitos dos passos seguintes.

2. Em qualquer lugar em que você perceba estar codificando qualquer coisa relacionada a versão IP, tente a embalar com uso de uma função auxiliar.

3. Altere `AF_INET` para `AF_INET6`.

4. Altere `PF_INET` para `PF_INET6`.

5. Altere atribuições `INADDR_ANY` para atribuições `in6addr_any`, que são ligeiramente diferentes:

```
struct sockaddr_in sa;
struct sockaddr_in6 sa6;

sa.sin_addr.s_addr = INADDR_ANY;  // usar meu endereço IPv4
sa6.sin6_addr = in6addr_any; // usar meu endereço IPv6
```

   Além disso, o valor `IN6ADDR_ANY_INIT` pode ser usado como um inicializador quando a `struct in6_addr` é declarada, assim:

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

6. Em vez de `struct sockaddr_in` use `struct sockaddr_in6`, certificando-se de adicionar "6" para os campos de forma apropriada (Veja `structs`, acima). Não há campo `sin6_zero`.

7. Em vez de `struct in_addr` use `struct in6_addr`, certificando-se de adicionar "6" para os campos conforme apropriado (Veja `structs`, acima).

8. Em vez de `inet_aton()` ou `inet_addr()`, use `inet_pton()`.

9. Em vez de `inet_ntoa()`, use `inet_ntop()`.

10. Em vez de `gethostbyname()`, utilize a superior `getaddrinfo()`.

11. Em vez de `gethostbyaddr()`, use a superior `getnameinfo()` (embora `gethostbyaddr()` ainda possa trabalhar com IPv6).

12. `INADDR_BROADCAST` não funciona mais. Use IPv6 multicast em seu lugar.

E é isso!

Chamadas de Sistema

Esta é a seção onde nós entramos nas chamadas de sistema (e outras chamadas de bibliotecas) que permitem que você acesse funcionalidades de rede de um ambiente Unix, ou qualquer ambiente que suporte a API de sockets para esses assuntos (BSD, Windows, Linux, Mac, o-que-você-tiver.). Quando você chama uma dessas funções, o kernel assume o controle

e faz todo o trabalho para você automagicamente.

O lugar onde a maioria das pessoas ficam presas aqui é na ordem de chamada das funções. Nisso, as páginas man não são úteis, como você provavelmente já descobriu. Bem, para ajudar nessa terrível situação, tentei expor as chamadas de sistema nas seções a seguir *exatamente* (aproximadamente) na mesma ordem em que você precisará chamá-las em seus programas.

Isso, juntamente com algumas amostras de códigos aqui e ali, um pouco de leite com biscoitos (que temo que você tenha que fornecer a si mesmo), e algumas vísceras cruas com coragem, e você estará transmitindo dados pela Internet como o Filho de Jon Postel!

(Observe que, por brevidade, muitos trechos de código abaixo não fazem as verificações de erros necessárias. E eles muito comumente assumem que os resultados de chamadas a `getaddrinfo()` têm sucesso e retornam uma entrada válida para a lista ligada. Ambas as situações são adequadamente abordadas nos programas independentes, portanto, use-os como um modelo.)

`getaddrinfo()`—Prepare Para Começar!

Esta função é um verdadeiro burro de carga com diversas opções, mas seu uso é realmente muito simples. Ela ajuda a definir as `structs` que você precisará mais tarde.

Um pouco de história: Costumávamos usar uma função chamada `gethostbyname()` para fazer pesquisas de DNS. Então carregávamos essas informações à mão em uma `struct sockaddr_in`, e usávamos isso em nossas chamadas.

Isto não é mais necessário, felizmente (Também não é desejável, se você quer escrever código que funcione tanto para IPv4 quanto para IPv6!). Nestes tempos modernos, você tem agora a função `getaddrinfo()` que faz todos os tipos de coisas boas para você, incluindo pesquisas de nomes DNS e serviço, e preenche as `structs` que você precisa, além disso!

Vamos dar uma olhada!

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

int getaddrinfo(const char *node,     // Ex. "www.example.com"
    ou IP
                const char *service,  // Ex. "http" ou número da
    porta
                const struct addrinfo *hints,
                struct addrinfo **res);
```

Você passa à essa função três parâmetros de entrada, e dá-lhe um ponteiro para uma lista ligada, `res`, para resultados.

O parâmetro node é o nome do host a se conectar, ou um endereço IP.

Em seguida vem o parâmetro `service`, que pode ser um número de porta, como "80", ou o nome de um determinado serviço (encontrados em The IANA Port List [20] ou no arquivo `/etc/services` em sua máquina Unix) como "http" ou "ftp" ou "telnet" ou "smtp" ou qualquer outro.

Finalmente, o parâmetro `hints` aponta para uma `struct addrinfo` já preenchida por você com informações relevantes.

Aqui está uma chamada de exemplo, se você é um servidor no IP do seu próprio host, porta 3490. Observe que isso não faz realmente nenhuma escuta ou configuração de rede; ele simplesmente configura estruturas que usaremos mais tarde:

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // apontará para os resultados

memset(&hints, 0, sizeof hints); // verifique se a estrutura está vazia
hints.ai_family = AF_UNSPEC;     // não me importo se IPv4 ou IPv6
hints.ai_socktype = SOCK_STREAM; // sockets stream TCP
hints.ai_flags = AI_PASSIVE;     // preencha meu IP para mim

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

// servinfo agora aponta para uma lista encadeada de 1
// ou mais structs addrinfos

// ... faça tudo até que você não precise mais de servinfo ....

freeaddrinfo(servinfo); // liberar a lista encadeada
```

Observe que eu defini o `ai_family` para `AF_UNSPEC`, dizendo com isso que eu não me importo se usarmos IPv4 ou IPv6. Você pode configurá-lo para `AF_INET` ou `AF_INET6` se você quiser um ou outro especificamente.

Além disso, você verá a flag `AI_PASSIVE` ali; isto diz a `getaddrinfo()` para atribuir o endereço do meu host local à estrutura do socket. Isso é bom porque você não precisa codificá-lo. (Ou você pode colocar um endereço específico como primeiro parâmetro de

`getaddrinfo()`, onde eu tenho atualmente `NULL`, lá em cima.)

Então nós fazemos a chamada. Se houver um erro (`getaddrinfo()` retornar diferente de zero), podemos imprimi-lo usando a função `gai_strerror()`, como você pode ver. Se tudo funcionar corretamente, porém, `servinfo` irá apontar para uma lista ligada de `struct addrinfos`, cada uma das quais contém uma `struct sockaddr` de algum tipo que podemos usar mais tarde! Bacana!

Finalmente, quando terminamos de usar a lista ligada que `getaddrinfo()` tão graciosamente alocou para nós, nós podemos (e devemos) liberar tudo com uma chamada a `freeaddrinfo()`.

Aqui está um exemplo de chamada se você é um cliente que quer se conectar a um determinado servidor, digamos "www.example.net" na porta 3490. Novamente, isto não faz realmente se conectar, mas configura as estruturas que usaremos mais tarde:

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // apontará para os resultados

memset(&hints, 0, sizeof hints); // verifique se a estrutura está vazia
hints.ai_family = AF_UNSPEC;     // não me importo se IPv4 ou IPv6
hints.ai_socktype = SOCK_STREAM; // sockets stream TCP

// prepare-se para conectar
status = getaddrinfo("www.example.net", "3490", &hints, &servinfo);

// servinfo agora aponta para uma
// lista encadeada de 1 ou mais struct addrinfos

// etc.
```

Eu continuo dizendo que `servinfo` é uma lista ligada com todos os tipos de informações de endereço. Vamos escrever um programa de demonstração rápida para mostrar essas informações. Este pequeno programa [21] imprimirá os endereços IP para qualquer host que você especifique na linha de comando:

```
/*
** showip.c -- mostra endereços IP para um host dado na linha de comando
*/

```

```c
#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,"uso: showip hostname\n");
        return 1;
    }

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC; // AF_INET ou AF_INET6 para forçar versão
    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("Endereço IP para %s:\n\n", argv[1]);

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

        // pegue o ponteiro para o endereço em si,
        // campos diferentes em 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
```

```
46              struct sockaddr_in6 *ipv6 = (struct
    sockaddr_in6 *)p->ai_addr;
47              addr = &(ipv6->sin6_addr);
48              ipver = "IPv6";
49          }
50
51          // converta o IP em uma string e imprima-o:
52          inet_ntop(p->ai_family, addr, ipstr, sizeof
    ipstr);
53          printf("  %s: %s\n", ipver, ipstr);
54      }
55
56      freeaddrinfo(res); // libere a lista ligada
57
58      return 0;
59  }
```

Como você pode ver, o código chama `getaddrinfo()` para o que quer que você passe na linha de comando, que preenche a lista ligada apontada por `res`, e então nós podemos iterar sobre a lista e imprimir coisas ou fazer qualquer coisa.

(Há um pouco de feiúra lá onde nós temos que cavar diferentes tipos de `struct sockaddrs` dependendo da versão IP. Me desculpe por isso! Eu não tenho certeza de uma maneira melhor para contornar isso.)

Execução de exemplo! Todo mundo adora screenshots:

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

  IPv4: 192.0.2.88

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

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

Agora que temos isso sob controle, usaremos os resultados que obtivemos de `getaddrinfo()` para passar a outras funções de socket e, finalmente, estabelecer a nossa conexão de rede! Continue lendo!

`socket()`—Obtenha o descritor de arquivo!

Eu acho que não posso mais deixar de comentar—Eu tenho que comentar sobre a chamada de

sistema `socket()`. Aqui estão os detalhes:

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

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

Mas o que são esses argumentos? Eles permitem que você defina que tipo de socket deseja (IPv4 ou IPv6, stream ou datagram, e TCP ou UDP).

As pessoas costumavam codificar esses valores, e você ainda pode fazer isso. (`domain` é `PF_INET` ou `PF_INET6`, `type` é `SOCK_STREAM` ou `SOCK_DGRAM`, e `protocol` pode ser definido como `0` para que se escolha o protocolo apropriado para `type`. Ou você pode chamar `getprotobyname()` para procurar o protocolo desejado, "tcp" ou "udp".)

(Este `PF_INET` é um parente próximo do `AF_INET` que você pode usar ao inicializar o campo `sin_family` em sua `struct sockaddr_in`. Na verdade, eles são tão intimamente relacionados que possuem realmente o mesmo valor, e muitos programadores chamam `socket()` e passam `AF_INET` como o primeiro argumento em vez de `PF_INET`. Agora, pegue um pouco de leite com biscoitos, porque é hora da história. Era uma vez, há muito tempo, pensava-se que talvez uma família de endereços (o que significa o "AF" em "`AF_INET`") pudesse suportar vários protocolos que foram referidos por sua família de protocolos (o que significa "PF" em "`PF_INET`"). Isso não aconteceu. E todos viveram felizes para sempre, fim. Então a coisa mais certa a fazer é usar `AF_INET` em sua `struct sockaddr_in` e `PF_INET` em sua chamada a `socket()`.)

De qualquer forma, chega disso. O que você realmente precisa fazer é usar os valores a partir dos resultados da execução de `getaddrinfo()`, e alimentá-los em `socket()` diretamente assim:

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

// faça a pesquisa
// [fingir que já preenchemos a estrutura "hints"]
getaddrinfo("www.example.com", "http", &hints, &res);

// novamente, você deve verificar erros em getaddrinfo()
 e percorrer
// a lista vinculada "res" procurando entradas válidas
 em vez de apenas
// supor que a primeira seja boa (como muitos desses
 exemplos).
// Veja a seção cliente/servidor para exemplos reais.

```

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

`socket()` simplesmente retorna a você um *descritor de socket* que você pode usar em chamadas de sistema posteriores, ou `-1` em caso de erro. A variável global `errno` é definida para o valor do erro (veja a página man `errno` para mais detalhes, e uma nota rápida sobre o uso de `errno` em programas multithreaded).

Bem, bem, bem, mas o que há de bom neste socket? A resposta é que ele não é muito bom por si só, e você precisa ler e fazer mais chamadas de sistema para que possa fazer qualquer sentido.

`bind()`—Em que porta eu estou?

Depois de ter um socket, talvez seja necessário o associar a uma porta na sua máquina local. (Isto é comumente feito se você estiver usando `listen()` para conexões de entrada em uma porta específica—jogos de rede multijogador fazem isso quando dizem para "conecte-se a 192.168.5.10 na porta 3490".) O número da porta é usado pelo kernel para asscciar um pacote de entrada ao descritor de socket de um determinado processo. Se você estiver apenas usando `connect()` (porque você é o cliente, não o servidor), isso provavelmente será desnecessário. Leia de qualquer maneira, apenas por diversão.

Aqui está a sinopse para a chamada de sistema `bind()`:

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

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

`sockfd` é o descritor de arquivo de socket retornado por `socket()`. `my_addr` é um ponteiro para uma `struct sockaddr` que contém informações sobre o seu endereço, ou seja, porta e endereço IP. `addrlen` é o comprimento em bytes desse endereço.

Uau. Isso é um pouco para que possamos absorver em pouco tempo. Vamos a um exemplo de uso de sockets com bind() no host onde o programa é executado, porta 3490:

```
1 struct addrinfo hints, *res;
2 int sockfd;
3
4 // primeiro, carregar estruturas de endereço com
   getaddrinfo():
5
6 memset(&hints, 0, sizeof hints);
7 hints.ai_family = AF_UNSPEC;  // usar IPv4 ou IPv6,
   qualquer que seja
8 hints.ai_socktype = SOCK_STREAM;
```

```
hints.ai_flags = AI_PASSIVE;     // preencha meu IP para
 mim

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

// cria o socket:

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

// Usa bind na porta que passamos a getaddrinfo():

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

Ao utilizar a flag `AI_PASSIVE`, estou dizendo ao programa para usar bind() no IP da máquina em que está sendo executado. Se você deseja usar bind em um endereço IP local específico, ignore `AI_PASSIVE` e ponha o endereço IP como primeiro argumento de `getaddrinfo()`.

`bind()` também retorna `-1` em caso de erro e põe em `errno` o valor do erro.

Muitos códigos antigos empacotam manualmente a `struct sockaddr_in` antes de chamarem `bind()`. Obviamente, isso é específico para IPv4, mas não há realmente nada que impeça você de fazer a mesma coisa com IPv6, exceto que o uso de `getaddrinfo()` será mais fácil, em geral. De qualquer forma, o código antigo é algo como isto:

```
// !!! ESTE É O MODO ANTIGO !!!

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, network
 byte order
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);
```

No código acima, você também pode atribuir `INADDR_ANY` ao campo `s_addr`, se você quiser usar bind no seu endereço de IP local (como a flag `AI_PASSIVE`, acima). A versão IPv6 de `INADDR_ANY` é uma variável global `in6addr_any` que é atribuída ao campo `sin6_addr` de sua `struct sockaddr_in6`. (Há também uma macro `IN6ADDR_ANY_INIT` que você pode usar em uma variável inicializadora.)

Outra coisa a observar ao chamar `bind()`: não use números baixos como endereços de portas. Todas as portas abaixo de 1024 são RESERVADAS (a menos que você seja o superusuário)! Você pode ter qualquer número de porta acima disso, até 65535 (desde que não esteja sendo usada por outro programa).

Às vezes, você pode perceber, você tenta executar novamente um servidor e `bind()` falha, alegando "Endereço já em uso." O que significa isso? Bem, parte de um socket que estava conectado ainda está pendurada no kernel e está monopolizando a porta. Você pode esperar que ele seja limpo (um minuto ou mais) ou adicionar ao seu programa um código que lhe permita reutilizar a porta, como isso:

```
int yes=1;
//char yes='1'; // As pessoas do Solaris usam isso

// contornar a mensagem de erro "Endereço já em uso"
if
 (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof
 yes) == -1) {
    perror("setsockopt");
    exit(1);
}
```

Uma pequena nota final extra sobre `bind()`: há momentos em que você absolutamente não precisará chamá-la. Se você está conectado com `connect()` a uma máquina remota e você não se importa com a porta local (como é o caso com `telnet`, onde você só se preocupa com a porta remota), você pode simplesmente chamar `connect()`, ela verificará se o socket está desativado, e fará `bind()` para uma porta local não usada, se necessário.

`connect()`—Ei, você!

Vamos fingir por alguns minutos que você é uma aplicação telnet. Seu usuário ordena que você (assim como no filme *TRON*) obtenha um descritor de arquivo de socket. Você obedece e chama `socket()`. Em seguida, o usuário diz-lhe para conectar-se a "`10.12.110.57`" na porta "23" (a porta padrão para telnet.) Uau! O que você faz agora?

Para sua sorte, programa, você está agora examinando a seção sobre `connect()`—como se conectar a um host remoto. Então leia furiosamente a seguir! Não há tempo a perder!

A chamada `connect()` é a seguinte:

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

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

`sockfd` é o nosso descritor de arquivo de socket amigável, como retornado pela chamada `socket()`, `serv_addr` é uma `struct sockaddr` contendo a porta de destino e o endereço IP, e `addrlen` é o comprimento em bytes da estrutura de endereço do servidor.

Todas essas informações podem ser adquiridas a partir dos resultados da chamada de `getaddrinfo()`.

Isso está começando a fazer mais sentido? Eu não posso lhe ouvir a partir daqui, por isso, eu espero que esteja. Vamos dar um exemplo onde fazemos uma conexão a "`www.example.com`", porta `3490`:

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

// primeiro, carregue as estruturas de endereço com
 getaddrinfo:

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

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

// crie o socket:

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

// conectar!

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

Mais uma vez, os programas old-school preenchiam suas próprias `struct sockaddr_in` para passar a `connect()`. Você pode fazer isso se você quiser. Veja a nota similar na seção bind(), acima.

Certifique-se de verificar o valor de retorno de `connect()`—ela retornará `-1` em caso de erro e configurará a variável `errno`.

Além disso, observe que nós não chamamos `bind()`. Basicamente, nós não nos importamos com o nosso número de porta local; nós só nos importamos com para onde estamos indo (a porta remota). O kernel escolherá uma porta local para nós, e o site ao qual nos conectaremos receberá automaticamente essas informações. Não se preocupe.

`listen()`—Alguém por favor pode me ligar?

Ok, tempo para uma mudança de ritmo. E se você não quiser se conectar a um host remoto? Digo, apenas por diversão, você quer esperar conexões de entrada e tratá-las de alguma forma. O processo é executado em dois passos: primeiro você usa `listen()`, então você usa `accept()` (veja abaixo).

A chamada de escuta (listen) é bastante simples, mas requer um pouco de explicação:

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

`sockfd` é o descritor de arquivo de socket usual da chamada de sistema `socket()`. `backlog` é o número de conexões permitidas na fila de entrada. O que isso significa? Bem, conexões de entrada aguardarão nesta fila até que você as aceite com `accept()` (veja abaixo) e este é o limite de quantas podem entrar na fila. A maioria dos sistemas limita silenciosamente esse número para cerca de 20; você provavelmente pode definir como `5` ou `10`.

Mais uma vez, como de costume, `listen()` retorna `-1` e configura `errno` quando houverem erros.

Bem, como você provavelmente pode imaginar, precisamos chamar `bind()` antes de chamarmos `listen()` para que o servidor esteja sendo executado em uma porta específica. (Você tem que ser capaz de dizer aos seus amigos em que porta conectarem-se!) Então, se você estiver ouvindo as conexões de entrada, a sequência de chamadas de sistema que você fará é:

```
1 getaddrinfo();
2 socket();
3 bind();
4 listen();
5 /* accept() aceita aqui */
```

Eu só deixarei isso como código de exemplo, uma vez que é bastante autoexplicativo. (O código na seção `accept()`, abaixo, é mais completo.) A parte realmente complicada de todas estas coisas é a chamada de `accept()`.

`accept()`—"Obrigado por ligar para a porta 3490."

Prepare-se—a chamada a `accept()` é meio estranha! O que vai acontecer é o seguinte: alguém de muito longe vai tentar usar `connect()` contra a sua máquina em uma porta em que você usou `listen()`. As conexões serão enfileiradas esperando para serem aceitas com `accept()`. Você chama `accept()` e diz a ela para obter a conexão pendente. Ela vai retornar para você um novo *descritor de arquivo socket* para utilizar com esta única conexão! É isso mesmo, de repente você tem *dois descritores de arquivos sockets* pelo preço de um! O original ainda continua esperando novas conexões, e o recém-criado está finalmente pronto para o uso de `send()` e `recv()`. Chegamos lá!

A chamada é a seguinte:

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

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

`sockfd` é o descritor de socket de `listen()`. Bastante fácil. `addr` normalmente será um ponteiro para uma `struct sockaddr_storage` local. É aqui onde as informações sobre a conexão de entrada vão (e com elas você pode determinar qual host está lhe chamando de qual porta). `addrlen` é uma variável local do tipo inteira que deve ser definida para `sizeof(struct sockaddr_storage)` antes de seu endereço ser passado a `accept()`. `accept()` não colocará mais bytes do que `addr` foi configurado para guardar. Se ele colocar um menor número, ele alterará o valor de `addrlen` para refletir isso.

Adivinha? `accept()` retorna `-1` e define `errno` se ocorrer um erro. Aposto que não percebeu.

Como antes, isso é muito para absorver em tão pouco tempo, então aqui está um fragmento de código de exemplo para leitura:

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

#define MYPORT "3490"  // a porta onde os usuários se
 estarão
#define BACKLOG 10     // quantidade de conexões
 enfileiradas

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

    // !! não esqueça de fazer de realizar as
 verificações de erros !!

    // primeiro, carregue as estruturas de endereços com
 getaddrinfo():

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;  // usar IPv4 ou IPv6,
 o que for
```

```
22      hints.ai_socktype = SOCK_STREAM;
23      hints.ai_flags = AI_PASSIVE;     // preencha meu IP
    para mim
24
25      getaddrinfo(NULL, MYPORT, &hints, &res);
26
27      // cria socket, liga-o com bind, e ouve nele com
    listen:
28
29      sockfd = socket(res->ai_family, res->ai_socktype,
    res->ai_protocol);
30      bind(sockfd, res->ai_addr, res->ai_addrlen);
31      listen(sockfd, BACKLOG);
32
33      // agora aceita uma conexão de entrada:
34
35      addr_size = sizeof their_addr;
36      new_fd = accept(sockfd, (struct sockaddr
    *)&their_addr, &addr_size);
37
38      // pronto para se comunicar no descritor de socket
    new_fd!
39      .
40      .
41      .
```

Mais uma vez, note que vamos usar o descritor de socket `new_fd` para todas as chamadas `send()` e `recv()`. Se você está recebendo apenas uma única conexão, você pode fechar com `close()` a escuta de `sockfd` a fim de evitar mais conexões de entrada na mesma porta, se você assim desejar.

`send()` e `recv()`—Fale comigo, baby!

Estas duas funções são para comunicação através de sockets stream ou sockets datagram conectados. Se você quiser usar sockets datagram regulares, desconectados, você precisa ver a seção sobre `sendto()` e `recvfrom()`, abaixo.

A chamada `send()`:

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

`sockfd` é o descritor de socket para o qual você quer enviar dados (seja ele o retornado por `socket()` ou o que você recebeu com `accept()`.) `msg` é um ponteiro para os dados que você deseja enviar, e `len` é o comprimento desses dados em bytes. Basta definir `flags` para `0`. (Veja a página man de `send()` para mais informações sobre flags.)

Um código de exemplo pode ser:

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

`send()` retorna o número de bytes realmente enviados—*isso pode ser menor do que o número que você disse para ela enviar!* Veja, às vezes você diz para enviar um monte de dados e ela simplesmente não pode lidar com isso. Ela irá disparar tanto dos dados quanto possível, e confiará em você para enviar o resto mais tarde. Lembre-se, se o valor retornado por `send()` não coincide com o valor em `len`, cabe a você enviar o resto da string. A boa notícia é a seguinte: se o pacote for pequeno (menos de 1K ou quase isso) ela irá *provavelmente* gerenciar para enviar a coisa toda de uma só vez. Mais uma vez, `-1` é devolvido em caso de erro, e `errno` é definido para o número do erro.

A chamada `recv()` é semelhante em muitos aspectos:

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

`sockfd` é o descritor de socket a ser lido, `buf` é o buffer para receber as informações, `len` é o comprimento máximo do buffer, e `flags` pode ser novamente ajustada para `0`. (Veja a página man de `recv()` para obter informações sobre flags.)

`recv()` retorna o número de bytes realmente lidos no buffer, ou `-1` em caso de erro (com `errno` ajustado de acordo.)

Espere! `recv()` pode retornar `0`. Isso só pode significar uma coisa: o lado remoto terminou a conexão com você! Um valor de retorno `0` é a forma de `recv()` informar que isso ocorreu.

Agora, isso foi fácil, não foi? Agora você pode enviar e receber dados em sockets stream! Uau! Você é um Programador de Rede Unix!

`sendto()` e `recvfrom()`—Fale comigo, DGRAM-style

"Isto é tudo muito bom e elegante," Eu ouvi você dizendo, "mas onde é que isto me leva com sockets datagram desconectados?". Sem problemas, amigo. Nós temos exatamente a coisa.

Como sockets datagram não estão conectados a um host remoto, adivinha qual informação precisamos fornecer antes de enviar um pacote? Está certo! O endereço de destino! Aqui está

o escopo:

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

Como você pode ver, esta chamada é basicamente a mesma que a chamada `send()` com a adição de dois outros pedaços de informação. `to` é um ponteiro para uma `struct sockaddr` (que provavelmente será outra `struct sockaddr_in` ou `struct sockaddr_in6` ou `struct sockaddr_storage` que você converteu no último minuto), que contém o endereço IP de destino e porta. `tolen`, um `int` no fundo, pode simplesmente ser definido para `sizeof *to` ou `sizeof(struct sockaddr_storage)`.

Para colocar as mãos na estrutura de endereço de destino, você provavelmente a obterá de `getaddrinfo()`, ou a partir de `recvfrom()`, abaixo, ou preencherá manualmente.

Assim como com `send()`, `sendto()` retorna o número de bytes realmente enviados (que, novamente, pode ser menor do que o número de bytes que você disse para ela enviar), ou `-1` em caso de erro.

Igualmente semelhantes são `recv()` e `recvfrom()`. A sinopse de `recvfrom()` é:

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

Novamente, isso é exatamente como `recv()` com a adição de alguns campos. `from` é um ponteiro para uma `struct sockaddr_storage` que será preenchida com o endereço IP e a porta da máquina de origem. `fromlen` é um ponteiro para um `int` local, que deve ser inicializado para `sizeof *from` ou `sizeof (struct sockaddr_storage)`. Quando a função retornar, `fromlen` conterá o comprimento do endereço armazenado em `from`.

`recvfrom()` retorna o número de bytes recebidos, ou `-1` em caso de erro (com `errno` ajustado em conformidade.)

Então, aqui vai uma pergunta: Por que usamos `struct sockaddr_storage` como o tipo de socket? Por que não `struct sockaddr_in`? Porque, veja, nós não queremos nos amarrar ao IPv4 ou IPv6. Então, usamos uma struct genérica `struct sockaddr_storage`, que sabemos que será grande o suficiente para ambos.

(Então... aqui está uma outra questão: por que `struct sockaddr` não é grande o suficiente para qualquer endereço? Nós até definimos a `struct sockaddr_storage` de propósito geral para a `struct sockaddr` de propósito geral! Parece estranho e redundante, hum. A resposta é, ela simplesmente não é grande o suficiente, e eu acho que alterá-la neste momento seria problemático. Então, eles fizeram uma nova.)

Lembre-se, se você usa `connect()` com um socket datagram, então você pode simplesmente usar `send()` e `recv()` para todas as suas transações. O socket em si ainda é um socket

datagram e os pacotes ainda usam UDP, mas a interface do socket adicionará automaticamente as informações de destino e fonte para você.

`close()` e `shutdown()`—Não olhe mais na minha cara!

Uau! Você esteve enviando dados com `send()` e recebendo com `recv()` o dia inteiro, e os obteve. Você está pronto para fechar a conexão em seu descritor de socket. Isso é fácil. Você pode apenas usar a função normal de descritores de arquivos Unix `close()`:

```
close(sockfd);
```

Isso evitará mais leituras e gravações no socket. Qualquer um tentando ler ou escrever no socket na extremidade remota receberá um erro.

Apenas no caso de você querer um pouco mais de controle sobre como o socket fecha, você pode usar a função `shutdown()`. Ela permite que você interrompa a comunicação em um determinado sentido, ou em ambos os sentidos (assim como `close()` faz). Sinopse:

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

`sockfd` é o descritor de arquivo socket que você deseja desligar, e `how` é um dos seguintes:

how	Effect
`0`	Recebimentos seguintes desativados
1	Envios seguintes desativados
2	Envios e recebimentos seguintes desativados (como `close()`)

`shutdown()` retorna `0` em caso de sucesso, e `-1` em caso de erro (com `errno` ajustado em conformidade.)

Se você se atrever a usar `shutdown()` em sockets datagram desconectados, ela simplesmente tornará o socket indisponível para posteriores chamadas de `send()` e `recv()` (lembre-se de que você pode usá-las se você usou `connect()` com sockets datagram.)

É importante notar que `shutdown()` na verdade não fecha o descritor de arquivo—apenas altera a sua usabilidade. Para liberar um descritor de socket, você precisa usar `close()`.

Nada mais sobre.

(Exceto para lembrar que, se você estiver usando Windows e Winsock você deverá chamar `closesocket()` em vez de `close()`.)

`getpeername()`—Quem é você?

Essa função é tão fácil.

É tão fácil, que eu quase não lhe dei a sua própria secção. Mas aqui está, de qualquer forma.

A função `getpeername()` informará quem está na outra extremidade de um socket stream conectado. A sinopse:

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

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

`sockfd` é o descritor do socket stream conectado, `addr` é um ponteiro para uma `struct sockaddr` (ou uma `struct sockaddr_in`) que conterá as informações sobre o outro lado da conexão, e `addrlen` é um ponteiro para um `int`, que deve ser inicializado para `sizeof *addr` ou `sizeof (struct sockaddr)`.

A função retorna `-1` em caso de erro e define `errno` em conformidade.

Depois de ter seu endereço, você pode usar `inet_ntop()`, `getnameinfo()` ou `gethostbyaddr()` para imprimir ou obter mais informações. Não, você não pode obter o seu nome de login. (Ok, ok. Se o outro computador está executando um daemon ident, isso é possível. Isso, no entanto, está além do escopo deste documento. Confira RFC 1413[22] para mais informações.)

`gethostname()`—Quem sou eu?

Ainda mais fácil do que `getpeername()` é a função `gethostname()`. Ela retorna o nome do computador em que seu programa está sendo executado. O nome pode ser usado por `gethostbyname()`, a seguir, para determinar o endereço IP da sua máquina local.

O que poderia ser mais divertido? Eu poderia pensar em algumas coisas, mas elas não pertencem à programação de sockets. De qualquer forma, aqui está sua composição:

O que poderia ser mais divertido? Eu poderia pensar em algumas coisas, mas elas não pertencem à programação de sockets. De qualquer forma, aqui está o detalhamento:

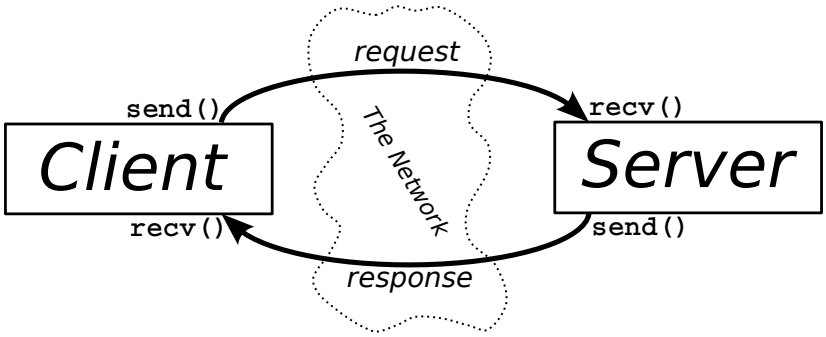
```
#include <unistd.h>

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

Os argumentos são simples: `hostname` é um ponteiro para um vetor de caracteres que conterá o nome do host ao retorno da função, e `size` é o comprimento em bytes do vetor `hostname`.

A função retorna `0` ao completar com sucesso, e `-1` em caso de erro, estabelecendo `errno` como de costume.

Cliente-Servidor Background

Isso é o um mundo cliente-servidor, baby. Quase tudo na rede é baseado em processos clientes falando com processos servidores e vice-versa. Como `telnet`, por exemplo. Quando você se conecta a um servidor remoto na porta 23 com o telnet (o cliente), um programa naquele host (chamado `telnetd`, o servidor) ganha vida. Ele lida com a conexão telnet de entrada, prepara um prompt de login, etc.



Interação Cliente-Servidor.

A troca de informações entre cliente e servidor é resumida no diagrama acima.

Note que o par cliente-servidor podem falar `SOCK_STREAM`, `SOCK_DGRAM`, ou qualquer outra coisa (contanto que eles estejam falando a mesma coisa.) Alguns bons exemplos de pares de cliente-servidor são `telnet/telnetd`, `ftp/ftpd`, ou `Firefox/Apache`. Toda vez que você usa `ftp`, há um programa remoto, `ftpd`, que serve você.

Geralmente, haverá apenas um servidor em uma máquina, e esse servidor lidará com vários clientes usando `fork()`. A rotina básica é: o servidor espera por uma conexão, a aceita com `accept()`, e cria um novo processo filho com `fork()` para lidar com isso. Isso é o que o nosso servidor de exemplo faz na próxima seção.

Um Servidor Stream Simples

Tudo o que esse servidor faz é enviar a string "`Olá, mundo!`" por uma conexão stream. Tudo que você precisa fazer para testar este servidor é executá-lo em uma janela, e com telnet conectar a ele a partir de outra com:

```
$ telnet remotehostname 3490
```

Onde `remotehostname` é o nome da máquina em que você está executando o servidor.

O código do servidor[23]:

```
1 /*
2 ** server.c -- uma demonstração de socket stream como
   servidor
3 */
4
5 #include <stdio.h>
```

```c
#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"  // a porta que os clientes usarão para se conectarem

#define BACKLOG 10   // a quantidade de conexões pendentes a enfileirar

void sigchld_handler(int s)
{
    // waitpid() pode sobrescrever errno, então nós salvamos e restauramos:
    int saved_errno = errno;

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

    errno = saved_errno;
}


// obtém sockaddr, IPv4 ou 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;  // ouça em sock_fd, nova conexão em new_fd
    struct addrinfo hints, *servinfo, *p;
```

```
47      struct sockaddr_storage their_addr; // informações
    de endereço do cliente
48      socklen_t sin_size;
49      struct sigaction sa;
50      int yes=1;
51      char s[INET6_ADDRSTRLEN];
52      int rv;
53
54      memset(&hints, 0, sizeof hints);
55      hints.ai_family = AF_UNSPEC;
56      hints.ai_socktype = SOCK_STREAM;
57      hints.ai_flags = AI_PASSIVE; // use meu IP
58
59      if ((rv = getaddrinfo(NULL, PORT, &hints,
    &servinfo)) != 0) {
60          fprintf(stderr, "getaddrinfo: %s\n",
    gai_strerror(rv));
61          return 1;
62      }
63
64      // loop através de todos os resultados e
65      // fazer bind para o primeiro que pudermos
66      for(p = servinfo; p != NULL; p = p->ai_next) {
67          if ((sockfd = socket(p->ai_family,
    p->ai_socktype,
68                  p->ai_protocol)) == -1) {
69              perror("servidor: socket");
70              continue;
71          }
72
73          if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR,
    &yes,
74                  sizeof(int)) == -1) {
75              perror("setsockopt");
76              exit(1);
77          }
78
79          if (bind(sockfd, p->ai_addr, p->ai_addrlen) ==
    -1) {
80              close(sockfd);
81              perror("servidor: bind");
82              continue;
83          }
84
85          break;
```

```
86      }
87
88      freeaddrinfo(servinfo); // tudo feito com essa
    estrutura
89
90      if (p == NULL)  {
91          fprintf(stderr, "servidor: falha com bind\n");
92          exit(1);
93      }
94
95      if (listen(sockfd, BACKLOG) == -1) {
96          perror("listen");
97          exit(1);
98      }
99
100     sa.sa_handler = sigchld_handler; // colher todos os
    processos mortos
101     sigemptyset(&sa.sa_mask);
102     sa.sa_flags = SA_RESTART;
103     if (sigaction(SIGCHLD, &sa, NULL) == -1) {
104         perror("sigaction");
105         exit(1);
106     }
107
108     printf("servidor: aguardando por conexões...\n");
109
110     while(1) {  // loop principal de accept()
111         sin_size = sizeof their_addr;
112         new_fd = accept(sockfd, (struct sockaddr
    *)&their_addr, &sin_size);
113         if (new_fd == -1) {
114             perror("accept");
115             continue;
116         }
117
118         inet_ntop(their_addr.ss_family,
119             get_in_addr((struct sockaddr *)&their_addr),
120             s, sizeof s);
121         printf("servidor: tenho uma conexão de %s\n",
    s);
122
123         if (!fork()) { // este é o processo filho
124             close(sockfd); // processo filho não precisa
    ouvir
```

```
125             if (send(new_fd, "Hello, world!", 13, 0) ==
     -1)
126                 perror("send");
127             close(new_fd);
128             exit(0);
129         }
130         close(new_fd);  // pai não precisa disso
131     }
132
133     return 0;
134 }
```

Caso você esteja curioso, eu tenho o código em uma grande função `main()` para manter (eu sinto) a clareza sintática. Sinta-se livre para dividi-la em funções menores, se isso te faz se sentir melhor.

(Além disso, toda esta coisa de `sigaction()` pode ser nova para você—isso é ok. O código que está lá é responsável por colher os processos zumbis que aparecem quando os processos filhos de `fork()` terminam. Se você criar lotes de zumbis e não terminá-los, o administrador do sistema se tornará raivoso.)

Você pode obter os dados desse servidor usando o cliente listado na próxima seção.

Um Cliente Stream Simples

Esse cara é ainda mais fácil que o servidor. Tudo o que este cliente faz é conectar-se ao host especificado na linha de comandos, porta 3490. Ele obtém a string que o servidor envia.

O código do cliente[24]:

```
1  /*
2  ** client.c -- uma demonstração de socket stream como
       cliente
3  */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <unistd.h>
8  #include <errno.h>
9  #include <string.h>
10 #include <netdb.h>
11 #include <sys/types.h>
12 #include <netinet/in.h>
13 #include <sys/socket.h>
14
15 #include <arpa/inet.h>
```

```
16
17  #define PORT "3490" // a porta onde o cliente conectará
18
19  #define MAXDATASIZE 100 // número máximo de bytes a
     obter por vez
20
21  // obtém sockaddr, IPv4 ou IPv6:
22  void *get_in_addr(struct sockaddr *sa)
23  {
24      if (sa->sa_family == AF_INET) {
25          return &(((struct sockaddr_in*)sa)->sin_addr);
26      }
27
28      return &(((struct sockaddr_in6*)sa)->sin6_addr);
29  }
30
31  int main(int argc, char *argv[])
32  {
33      int sockfd, numbytes;
34      char buf[MAXDATASIZE];
35      struct addrinfo hints, *servinfo, *p;
36      int rv;
37      char s[INET6_ADDRSTRLEN];
38
39      if (argc != 2) {
40          fprintf(stderr,"uso: cliente hostname\n");
41          exit(1);
42      }
43
44      memset(&hints, 0, sizeof hints);
45      hints.ai_family = AF_UNSPEC;
46      hints.ai_socktype = SOCK_STREAM;
47
48      if ((rv = getaddrinfo(argv[1], PORT, &hints,
     &servinfo)) != 0) {
49          fprintf(stderr, "getaddrinfo: %s\n",
     gai_strerror(rv));
50          return 1;
51      }
52
53      // percorrer todos os resultados e
54      // conectar-se ao primeiro que pudermos
55      for(p = servinfo; p != NULL; p = p->ai_next) {
56          if ((sockfd = socket(p->ai_family,
     p->ai_socktype,
```

```
57                  p->ai_protocol)) == -1) {
58              perror("cliente: socket");
59              continue;
60          }
61
62          if (connect(sockfd, p->ai_addr, p->ai_addrlen)
    == -1) {
63              close(sockfd);
64              perror("cliente: connect");
65              continue;
66          }
67
68          break;
69      }
70
71      if (p == NULL) {
72          fprintf(stderr, "cliente: falha em connect\n");
73          return 2;
74      }
75
76      inet_ntop(p->ai_family, get_in_addr((struct sockaddr
    *)p->ai_addr),
77              s, sizeof s);
78      printf("cliente: conectando a %s\n", s);
79
80      freeaddrinfo(servinfo); // tudo feito com essa
    estrutura
81
82      if ((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0))
    == -1) {
83          perror("recv");
84          exit(1);
85      }
86
87      buf[numbytes] = '\0';
88
89      printf("cliente: recebido '%s'\n",buf);
90
91      close(sockfd);
92
93      return 0;
94  }
```

Observe que, se você não executar o servidor antes de executar o cliente, `connect()` retornará "Conexão recusada". Muito útil.

Sockets Datagram

Nós já cobrimos o básico de sockets UDP datagram com a nossa discussão sobre `sendto()` e `recvfrom()`, acima, então eu vou apresentar apenas alguns programas de exemplo: `talker.c` e `listener.c`.

`listener` fica em uma máquina à espera de um pacote de entrada na porta 4950. `talker` envia um pacote para essa porta, na máquina especificada, que contém o que o usuário digita na linha de comando.

Aqui está o código para `listener.c`[25]:

```
1  /*
2  ** listener.c -- uma demonstração de socket "servidor"
    com datagram
3  */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <unistd.h>
8  #include <errno.h>
9  #include <string.h>
10 #include <sys/types.h>
11 #include <sys/socket.h>
12 #include <netinet/in.h>
13 #include <arpa/inet.h>
14 #include <netdb.h>
15
16 #define MYPORT "4950"   // a porta onde os usuários se
    conectarão
17
18 #define MAXBUFLEN 100
19
20 // obtém sockaddr, IPv4 ou IPv6:
21 void *get_in_addr(struct sockaddr *sa)
22 {
23     if (sa->sa_family == AF_INET) {
24         return &(((struct sockaddr_in*)sa)->sin_addr);
25     }
26
27     return &(((struct sockaddr_in6*)sa)->sin6_addr);
28 }
29
30 int main(void)
31 {
```

```
32      int sockfd;
33      struct addrinfo hints, *servinfo, *p;
34      int rv;
35      int numbytes;
36      struct sockaddr_storage their_addr;
37      char buf[MAXBUFLEN];
38      socklen_t addr_len;
39      char s[INET6_ADDRSTRLEN];
40
41      memset(&hints, 0, sizeof hints);
42      hints.ai_family = AF_UNSPEC; // configure com
    AF_INET para forçar IPv4
43      hints.ai_socktype = SOCK_DGRAM;
44      hints.ai_flags = AI_PASSIVE; // use meu IP
45
46      if ((rv = getaddrinfo(NULL, MYPORT, &hints,
    &servinfo)) != 0) {
47          fprintf(stderr, "getaddrinfo: %s\n",
    gai_strerror(rv));
48          return 1;
49      }
50
51      // loop através dos resultados e bind para o
    primeiro que pudermos
52      for(p = servinfo; p != NULL; p = p->ai_next) {
53          if ((sockfd = socket(p->ai_family,
    p->ai_socktype,
54                  p->ai_protocol)) == -1) {
55              perror("listener: socket");
56              continue;
57          }
58
59          if (bind(sockfd, p->ai_addr, p->ai_addrlen) ==
    -1) {
60              close(sockfd);
61              perror("listener: bind");
62              continue;
63          }
64
65          break;
66      }
67
68      if (p == NULL) {
69          fprintf(stderr, "listener: falha ao fazer bind
    para o socket\n");
```

```
70          return 2;
71      }
72
73      freeaddrinfo(servinfo);
74
75      printf("listener: aguardando por recvfrom...\n");
76
77      addr_len = sizeof their_addr;
78      if ((numbytes = recvfrom(sockfd, buf, MAXBUFLEN-1 ,
    0,
79          (struct sockaddr *)&their_addr, &addr_len)) ==
    -1) {
80          perror("recvfrom");
81          exit(1);
82      }
83
84      printf("listener: tenho um pacote de %s\n",
85          inet_ntop(their_addr.ss_family,
86              get_in_addr((struct sockaddr *)&their_addr),
87              s, sizeof s));
88      printf("listener: o pacote tem %d bytes de
    comprimento\n", numbytes);
89      buf[numbytes] = '\0';
90      printf("listener: conteúdo do pacote \"%s\"\n",
    buf);
91
92      close(sockfd);
93
94      return 0;
95  }
```

Observe que, em nossa chamada a `getaddrinfo()` estamos finalmente usando `SOCK_DGRAM`. Além disso, observe que não há necessidade de `listen()` ou `accept()`. Esta é uma das vantagens de usar sockets datagram desconectados!

Em seguida vem o [código fonte de `talker.c`](#)[26]:

```
1  /*
2  ** talker.c -- um "cliente" de demonstração com datagram
3  */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <unistd.h>
8  #include <errno.h>
```

```c
#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"   // a porta onde se conectar

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

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

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

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

    // loop atráves de todos os resultados e criar 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;
    }
```

```
49
50      if (p == NULL) {
51          fprintf(stderr, "talker: falha ao criar o
    socket\n");
52          return 2;
53      }
54
55      if ((numbytes = sendto(sockfd, argv[2],
    strlen(argv[2]), 0,
56              p->ai_addr, p->ai_addrlen)) == -1) {
57          perror("talker: sendto");
58          exit(1);
59      }
60
61      freeaddrinfo(servinfo);
62
63      printf("talker: enviados %d bytes para %s\n",
    numbytes, argv[1]);
64      close(sockfd);
65
66      return 0;
67  }
```

E isso é tudo o que existe sobre o assunto! Execute `listener` em alguma máquina, em seguida, execute `talker` em outra. Veja-as se comunicarem! Diversão garantida para toda a família!

Você nem precisa executar o servidor dessa vez! Você pode rodar `talker`, por si só, e ele feliz simplesmente dispara pacotes para o além, onde desaparecem se ninguém estiver pronto com `recvfrom()` no outro lado. Lembre-se: dados enviados usando sockets datagram UDP não possuem garantia de entrega!

Com exceção de um pequeno detalhe que eu já mencionei muitas vezes no passado: sockets datagram conectados. Eu preciso falar sobre isso aqui, já que estamos na seção datagram do documento. Digamos que `talker` chame `connect()` e especifique o endereço de `listener`. Daquele ponto em diante, `talker` só pode enviar e receber do endereço especificado por `connect()`. Por esta razão, você não precisa usar `sendto()` e `recvfrom()`; Você pode simplesmente usar `send()` e `recv()`.

Técnicas Ligeiramente Avançadas

Estas técnicas não são *realmente* avançadas, mas elas saem dos níveis mais básicos já cobertos. Na verdade, se você chegou até aqui, você deve se considerar bastante realizado nos fundamentos da programação de rede Unix! Parabéns!

Então, aqui vamos nós para o admirável mundo novo de algumas das coisas mais esotéricas

que você pode querer aprender sobre sockets. Veja agora!

Blocking

Blocking. Você já ouviu falar sobre isso—agora, o que diabos é isso? Em poucas palavras, "block" é jargão técnico para "sleep". Você provavelmente reparou que quando você executa `listener`, acima, ele fica ali até que um pacote chegue. O que acontece é que ele chamou `recvfrom()`, não havia dados e, portanto, por `recvfrom()` é feito "block" (ou seja, dormir lá) até que alguns dados cheguem.

Muitas funções fazem block. `accept()` faz block. Todas funções `recv()` fazem block. A razão pela qual elas podem fazer isso é porque elas estão autorizadas. Quando você cria pela primeira vez o descritor de socket com `socket()`, o kernel o configura para blocking. Se você não quer um socket realizando blocking, você precisa fazer uma chamada a `fcntl()`:

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

Ao definir um socket para non-blocking, você pode efetivamente "consultar" o socket para obter informações. Se você tenta ler de um socket non-blocking e não houverem dados lá, não será permitido fazer blocking—ele retornará `-1` e `errno` será definido como `EAGAIN` ou `EWOULDBLOCK`.

(Espere—ele pode retornar `EAGAIN` *ou* `EWOULDBLOCK`? Por qual você verificará? A especificação na verdade não diz qual o seu sistema retornará, então para a portabilidade, confira ambos.)

De um modo geral, no entanto, este tipo de consulta é uma má idéia. Se você colocar seu programa em uma espera ocupada procurando dados sobre o socket, você drenará tempo de CPU de modo antiquado. Uma solução mais elegante para verificar se há dados esperando para serem lidos vem na seção a seguir em `poll()`.

`poll()`—Multiplexação Síncrona de E/S

O que você realmente deseja fazer é, de alguma forma, monitorar um *monte* de sockets de uma só vez e depois lidar com aqueles que têm dados prontos. Dessa forma, você não precisa consultar continuamente todos os sockets para ver quais estão prontos para leitura.

Uma palavra de aviso: `poll()` é terrivelmente lento quando se trata de um número

> *gigantesco de conexões. Nessas circunstâncias, você obterá melhor desempenho com uma biblioteca de eventos como libevent[27] que tenta usar o método mais rápido possível disponível em seu sistema.*

Então, como você pode evitar as consultas? Não como simples ironia, você pode evitar usando a chamada de sistema `poll()`. Em poucas palavras, pediremos ao sistema operacional que faça todo o trabalho sujo para nós e avise-nos quando alguns dados estiverem prontos para leitura em quais sockets. Enquanto isso, nosso processo pode dormir, economizando recursos do sistema.

O plano geral do jogo é manter uma matriz de `struct pollfds` com informações sobre quais descritores de sockets queremos monitorar e que tipo de eventos queremos monitorar. O sistema operacional bloqueará a chamada `poll()` até que um desses eventos ocorra (por exemplo, "socket pronto para leitura!") ou até que ocorra um tempo limite especificado pelo usuário.

Útilmente, um socket `listen()` retornará "pronto para ler" quando uma nova conexão de entrada estiver pronta para ser aceita com `accept()`.

Isso é divertido o suficiente. Como usamos isso?

```
#include <poll.h>

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

`fds` é nossa matriz de informações (que sockets devem ser monitoradas para quê), `nfds` é a contagem de elementos na matriz e `timeout` é um tempo limite em milissegundos. Retorna o número de elementos na matriz que tiveram um evento ocorrendo.

Vamos dar uma olhada nessa `struct`:

```
struct pollfd {
    int fd;         // o descritor de socket
    short events;   // bitmap de eventos em que estamos
    interessados
    short revents;  // quando poll() retorna, bitmap de
    ocorridos
};
```

Então, teremos uma matriz dessas e veremos o campo `fd` para cada elemento em um descritor de socket que estamos interessados em monitorar. E então definiremos o campo `events` para indicar o tipo de evento em que estamos interessados.

O campo `events` é um bitwise-OR como se segue:

Macro	Descrição

Macro	Descrição
POLLIN	Avise-me quando os dados estiverem prontos para `recv()` neste socket.
POLLOUT	Avise-me quando eu puder utilizar `send()` neste socket sem bloquear.

Depois de ter seu array de `struct pollfds` em ordem, você pode passá-lo para `poll()`, passando também o tamanho do array, assim como um valor de tempo limite em milissegundos. (Você pode especificar um tempo limite negativo para aguardar uma eternidade.)

Após o retorno de `poll()`, você pode verificar o campo `revents` para ver se POLLIN ou POLLOUT está definido, indicando que o evento ocorreu.

(Na verdade, você pode fazer mais com a chamada `poll()`. Consulte a página de manual `poll()`, abaixo, para mais detalhes.)

Aqui está um exemplo[28] onde esperaremos 2,5 segundos para que os dados estejam prontos para leitura da entrada padrão, ou seja, quando você pressionar RETURN:

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

int main(void)
{
    struct pollfd pfds[1]; // Mais se você quiser monitorar mais

    pfds[0].fd = 0;          // Entrada padrão
    pfds[0].events = POLLIN; // Diga-me quando estiver pronto para ler

    // Se você precisava monitorar outras coisas, também:
    //pfds[1].fd = some_socket; // Algum descritor de socket
    //pfds[1].events = POLLIN;  // Diga-me quando pronto para leitura

    printf("Pressione RETURN ou aguarde 2,5 segundos pelo timeout\n");

    int num_events = poll(pfds, 1, 2500); // timeout de 2.5 segundos

    if (num_events == 0) {
        printf("Poll sofreu timeout!\n");
```

```
21      } else {
22          int pollin_happened = pfds[0].revents & POLLIN;
23
24          if (pollin_happened) {
25              printf("Descritor %d pronto para leitura\n",
    pfds[0].fd);
26          } else {
27              printf("Ocorreu um evento inesperado: %d\n",
    pfds[0].revents);
28          }
29      }
30
31      return 0;
32  }
```

Observe novamente que `poll()` retorna o número de elementos na matriz `pfds` para os quais os eventos ocorreram. Ela não informa *quais* elementos da matriz (você ainda precisa procurar por isso), mas informa quantas entradas têm um campo `revents` diferente de zero (para que você pare de procurar depois de encontrar muitos).

Algumas perguntas podem surgir aqui: como adicionar novos descritores de arquivo ao conjunto que passo para `poll()`? Para isso, basta ter espaço suficiente na matriz para tudo o que você precisa ou realoque com `realloc()` mais espaço, conforme necessário.

E quanto a excluir itens do conjunto? Para isso, você pode copiar o último elemento da matriz por cima do que você está excluindo. E depois passe menos um como a contagem para `poll()`. Outra opção é que você pode definir qualquer campo `fd` como um número negativo e `poll()` irá ignorá-lo.

Como podemos reunir tudo isso em um servidor de bate-papo em que você possa executar `telnet` contra?

O que faremos é iniciar um socket ouvinte e adicioná-lo ao conjunto de descritores de arquivos de `poll()`. (Ele será exibido como pronto para leitura quando houver uma conexão de entrada.)

Em seguida, adicionaremos novas conexões ao nosso array `struct pollfd`. E cresceremos dinamicamente se ficarmos sem espaço.

Quando uma conexão é fechada, nós a removeremos da array.

E quando uma conexão estiver pronta para leitura, leremos os dados dela e enviaremos esses dados para todas as outras conexões para que eles possam ver o que os outros usuários digitaram.

Portanto este servidor poll[29] uma tentativa. Execute-o em uma janela e, em seguida, `telnet localhost 9034` de várias outras janelas do terminal. Você deve poder ver o que você digite uma janela nas outras (depois que você pressionar RETURN).

Não apenas isso, mas se você pressionar `CTRL-]` e digitar `quit` para sair do `telnet`, o servidor deverá detectar a desconexão e remover você do array de descritores de arquivos.

```c
/*
** pollserver.c -- um servidor de bate-papo com várias pessoas
*/

#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"   // Porta em que estamos ouvindo

// Obtenha sockaddr, IPv4 ou 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);
}

// Retornar um socket de escuta
int get_listener_socket(void)
{
    int listener;     // Descritor de socket de escuta
    int yes=1;        // Para setsockopt() SO_REUSEADDR, abaixo
    int rv;

    struct addrinfo hints, *ai, *p;

    // Pegue um socket e ligue-o
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
```

```
40      hints.ai_socktype = SOCK_STREAM;
41      hints.ai_flags = AI_PASSIVE;
42      if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) !=
    0) {
43          fprintf(stderr, "selectserver: %s\n",
    gai_strerror(rv));
44          exit(1);
45      }
46
47      for(p = ai; p != NULL; p = p->ai_next) {
48          listener = socket(p->ai_family, p->ai_socktype,
    p->ai_protocol);
49          if (listener < 0) {
50              continue;
51          }
52
53          // Perder a maldita mensagem de erro "endereço
    já em uso"
54          setsockopt(listener, SOL_SOCKET, SO_REUSEADDR,
    &yes, sizeof(int));
55
56          if (bind(listener, p->ai_addr, p->ai_addrlen) <
    0) {
57              close(listener);
58              continue;
59          }
60
61          break;
62      }
63
64      // Se chegamos aqui, significa que não ficamos
    presos
65      if (p == NULL) {
66          return -1;
67      }
68
69      freeaddrinfo(ai); // Tudo feito com isso
70
71      // Listen
72      if (listen(listener, 10) == -1) {
73          return -1;
74      }
75
76      return listener;
77  }
```

```
78
79  // Inclua um novo descritor de arquivo no conjunto
80  void add_to_pfds(struct pollfd *pfds[], int newfd, int
     *fd_count, int *fd_size)
81  {
82      // Se não tivermos espaço, adicione mais espaço na
     matriz pfds
83      if (*fd_count == *fd_size) {
84          *fd_size *= 2; // Dobrar
85
86          *pfds = realloc(*pfds, sizeof(**pfds) *
     (*fd_size));
87      }
88
89      (*pfds)[*fd_count].fd = newfd;
90      (*pfds)[*fd_count].events = POLLIN; // Verifique
     pronto-para-ler
91
92      (*fd_count)++;
93  }
94
95  // Remova um índice do conjunto
96  void del_from_pfds(struct pollfd pfds[], int i, int
     *fd_count)
97  {
98      // Copie o do final sobre este
99      pfds[i] = pfds[*fd_count-1];
100
101     (*fd_count)--;
102 }
103
104 // Main
105 int main(void)
106 {
107     int listener;     // Descritor de socket de escuta
108
109     int newfd;        // Descritor de socket recém-
     aceito com accept()
110     struct sockaddr_storage remoteaddr; // Endereço do
     cliente
111     socklen_t addrlen;
112
113     char buf[256];    // Buffer para dados do cliente
114
115     char remoteIP[INET6_ADDRSTRLEN];
```

```
116
117     // Comece com espaço para 5 conexões
118     // (Realocamos conforme necessário)
119     int fd_count = 0;
120     int fd_size = 5;
121     struct pollfd *pfds = malloc(sizeof *pfds *
    fd_size);
122
123     // Configure e obtenha um socket ouvinte
124     listener = get_listener_socket();
125
126     if (listener == -1) {
127         fprintf(stderr, "erro ao obter o socket de
    escuta\n");
128         exit(1);
129     }
130
131     // Adicione o ouvinte para o conjunto
132     pfds[0].fd = listener;
133     pfds[0].events = POLLIN; // Reporta pronto para
    leitura na conexão recebida
134
135     fd_count = 1; // Para o ouvinte
136
137     // Main loop
138     for(;;) {
139         int poll_count = poll(pfds, fd_count, -1);
140
141         if (poll_count == -1) {
142             perror("poll");
143             exit(1);
144         }
145
146         // Percorra as conexões existentes procurando
    dados para ler
147         for(int i = 0; i < fd_count; i++) {
148
149             // Verifique se alguém está pronto para ler
150             if (pfds[i].revents & POLLIN) { // Temos
    um!!
151
152                 if (pfds[i].fd == listener) {
153                     // Se o ouvinte estiver pronto para
    ler, lide com a nova conexão
154
```

```
155                     addrlen = sizeof remoteaddr;
156                     newfd = accept(listener,
157                         (struct sockaddr *)&remoteaddr,
158                         &addrlen);
159 
160                     if (newfd == -1) {
161                         perror("accept");
162                     } else {
163                         add_to_pfds(&pfds, newfd,
    &fd_count, &fd_size);
164 
165                         printf("servidor poll: nova
    conexão de %s em "
166                             "socket %d\n",
167 
    inet_ntop(remoteaddr.ss_family,
168                                 get_in_addr((struct
    sockaddr*)&remoteaddr),
169                                 remoteIP,
    INET6_ADDRSTRLEN),
170                             newfd);
171                     }
172                 } else {
173                     // Se não o ouvinte, somos apenas um
    cliente comum
174                     int nbytes = recv(pfds[i].fd, buf,
    sizeof buf, 0);
175 
176                     int sender_fd = pfds[i].fd;
177 
178                     if (nbytes <= 0) {
179                         // Erro ou conexão encerrada
    pelo cliente
180                         if (nbytes == 0) {
181                             // Conexão fechada
182                             printf("servidor poll:
    desligar socket %d\n", sender_fd);
183                         } else {
184                             perror("recv");
185                         }
186 
187                         close(pfds[i].fd); // Tchau!
188 
189                         del_from_pfds(pfds, i,
    &fd_count);
190 
```

```
191                     } else {
192                         // Temos bons dados de um
    cliente
193
194                         for(int j = 0; j < fd_count;
    j++) {
195                             // Envie para todos!
196                             int dest_fd = pfds[j].fd;
197
198                             // Exceto o ouvinte e nós
    mesmos
199                             if (dest_fd != listener &&
    dest_fd != sender_fd) {
200                                 if (send(dest_fd, buf,
    nbytes, 0) == -1) {
201                                     perror("send");
202                                 }
203                             }
204                         }
205                     }
206                 } // END manipular dados do cliente
207             } // END ficou pronto para leitura em poll()
208         } // END percorrendo descritores de arquivo
209     } // END for(;;)--e você pensou que isso nunca iria
    acabar!
210
211     return 0;
212 }
```

Na próxima seção veremos uma função mais antiga, semelhante, chamada `select()`. Tanto `select()` quanto `poll()` oferecem funcionalidade e desempenho semelhantes, e diferem realmente apenas na forma como são usadas. `select()` pode ser um pouco mais portátil, mas talvez seja um pouco mais desajeitada em uso. Escolha a que você mais gosta, desde que seja compatível com seu sistema.

`select()`—Multiplexação Síncrona de E/S, Old School

Esta função é um pouco estranha, mas é muito útil. Considere a seguinte situação: você é um servidor e deseja ouvir as conexões de entrada, bem como manter a leitura das conexões que você já possui.

Não há problema, você diz, apenas um `accept()` e alguns pares de `recv()` já resolveriam. Não tão rápido, imbecil! E se você estiver em blocking em uma chamada `accept()`? Como você receberá dados com `recv()` ao mesmo tempo? "Use sockets non-blocking!" De jeito nenhum! Você não quer ser um parasita de CPU. O que, então?

`select()` lhe dá o poder para monitorar vários sockets ao mesmo tempo. Ele lhe dirá quais estão prontos para a leitura, quais estão prontos para a escrita, e quais sockets geraram exceções, se você realmente quiser saber isso.

> *Alguns avisos: `select()`, embora muito portátil, é terrivelmente lento quando se trata de um número gigante de conexões. Nessas circunstâncias, você obtém um melhor desempenho de uma biblioteca de eventos como a libevent[30], que tenta usar o método mais rápido possível disponível em seu sistema.*

Sem mais delongas, vou oferecer a sinopse 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);
```

A função monitora "sets" (conjuntos) de descritores de arquivos; em particular: `readfds`, `writefds` e `exceptfds`. Se você quiser ver se pode ler da entrada padrão e de algum descritor de socket, `sockfd`, apenas defina o descritor de arquivo `0` e `sockfd` para o "set" `readfds`. O parâmetro `numfds` deve ser definido para o valor do mais alto descritor de arquivo mais um. Neste exemplo, ele deve ser definido para `sockfd+1`, uma vez que é seguramente maior que o da entrada padrão (`0`).

Quando `select()` retorna, `readfds` será modificado para refletir quais descritores de arquivo selecionados por você estão prontos para a leitura. Você pode testá-los com a macro `FD_ISSET()`, abaixo.

Antes de avançar muito mais, falarei sobre como manipular esses conjuntos. Cada set é do tipo `fd_set`. As seguintes macros operam neste tipo:

Função	Descrição
`FD_SET(int fd, fd_set *set);`	Adiciona `fd` para `set`.
`FD_CLR(int fd, fd_set *set);`	Remove `fd` de `set`.
`FD_ISSET(int fd, fd_set *set);`	Retorna true se `fd` estiver em `set`.
`FD_ZERO(fd_set *set);`	Limpa todas as entradas de `set`.

Finalmente, o que é a estranha `struct timeval`? Bem, às vezes você não quer esperar para sempre até que alguém lhe envie alguns dados. Talvez a cada 96 segundos você queira imprimir "Ainda em andamento..." no terminal, mesmo que nada tenha acontecido. Esta estrutura de tempo permite que você especifique um período de tempo limite. Se o tempo for excedido e `select()` ainda não tiver encontrado nenhum descritor de arquivo pronto, ele retornará para que você possa continuar o processamento.

A `struct timeval` possui os seguinte campos:

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

Basta definir `tv_sec` para o número de segundos a esperar, e definir `tv_usec` para o número de microssegundos que se deve esperar. Sim, isso é _micro_segundos, não milissegundos. Há 1.000 microssegundos em um milésimo de segundo, e 1.000 milissegundos em um segundo. Assim, existem 1.000.000 microssegundos em um segundo. Por que "usec"? O "u" é supostamente parecido com a letra grega μ (Mi) que usamos para "micro". Além disso, quando a função retorna, `timeout` *poderia* ser atualizado para mostrar o tempo ainda restante. Isso depende de que sabor de Unix você está executando.

Uau! Temos um temporizador com resolução de microssegundos! Bem, não conte com isso. Você provavelmente terá que esperar, em parte, também o tempo padrão do seu Unix, não importando quão mínimo seja o tempo definido para `struct timeval`.

Outras coisas de interesse: Se você definir os campos da sua `struct timeval` para `0`, `select()` irá expirar imediatamente, efetivamente sondando todos os descritores de arquivos em seus sets. Se você definir o parâmetro `timeout` para NULL, ele nunca terá tempo limite e aguardará até que o primeiro descritor de arquivo esteja pronto. Finalmente, se você não se importa em esperar por um determinado conjunto, você pode apenas configurá-lo para NULL na chamada de `select()`.

O seguinte trecho de código[31] aguarda 2,5 segundos para que algo apareça na entrada padrão:

```
1  /*
2  ** select.c -- uma demonstração de select()
3  */
4
5  #include <stdio.h>
6  #include <sys/time.h>
7  #include <sys/types.h>
8  #include <unistd.h>
9
10 #define STDIN 0  // descritor de arquivo para entrada
    padrão
11
12 int main(void)
13 {
14     struct timeval tv;
15     fd_set readfds;
16
17     tv.tv_sec = 2;
18     tv.tv_usec = 500000;
```

```
19
20      FD_ZERO(&readfds);
21      FD_SET(STDIN, &readfds);
22
23      // não me importo com writefds e exceptfds:
24      select(STDIN+1, &readfds, NULL, NULL, &tv);
25
26      if (FD_ISSET(STDIN, &readfds))
27          printf("Uma tecla foi pressionada!\n");
28      else
29          printf("Tempo esgotado.\n");
30
31      return 0;
32  }
```

Se você estiver em um terminal de linha bufferizada, a tecla que você pressionar deverá ser sucedida por RETURN ou o tempo irá expirar de qualquer maneira.

Agora, alguns de vocês podem pensar que esta é uma ótima maneira de esperar por dados em um socket datagram—e você está certo: *pode* ser. Alguns Unix podem usar select desta maneira, e outros não. Você deve ver o que sua página man local diz sobre o assunto, se você quiser tenta-lo.

Alguns Unix atualizam o tempo em sua `struct timeval` para refletir a quantidade de tempo que resta até um tempo limite. Mas outros não. Não confie que isso ocorra se você quer portabilidade. (Use `gettimeofday()` se você precisar controlar o tempo decorrido. É cansativo, eu sei, mas essa é a maneira de ser feito.)

O que acontece se um socket no set de leitura fecha a conexão? Bem, nesse caso, `select()` retorna com o descritor de socket definido como "pronto para ler". Quando você realmente faz `recv()` a partir dele, `recv()` retornará `0`. É assim que você sabe que o cliente fechou a conexão.

Mais uma nota de interesse sobre `select()`: Se você tem um socket executando `listen()`, você pode verificar se há uma nova conexão colocando seu descritor de arquivo no set `readfds`.

E isso, meus amigos, é uma rápida visão geral da poderosa função `select()`.

Mas, por demanda popular, aqui está um exemplo em profundidade. Infelizmente, a diferença entre o exemplo simples, acima, e este aqui é significativa. Mas dê uma olhada, leia a descrição que o segue.

Este programa[32] funciona como um simples servidor de chat multiusuário. Comece executando-o em uma janela e, em seguida, conecte-se via `telnet` a ele ("`telnet hostname 9034`") a partir de várias outras janelas. Quando você digita algo em uma sessão `telnet`, isso deve aparecer em todas as outras.

```c
/*
** selectserver.c -- um servidor de bate-papo com várias pessoas
*/

#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"   // porta onde estaremos ouvindo

// obtém sockaddr, IPv4 ou 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 de descritores de arquivos master
    fd_set read_fds;  // lista de descritores temporários para select()
    int fdmax;        // número máximo de descritores de arquivos

    int listener;     // descritor de socket de escuta
    int newfd;        // novos descritores de socket com accept()
    struct sockaddr_storage remoteaddr; // endereço do cliente
    socklen_t addrlen;

    char buf[256];    // buffer para dados do cliente
    int nbytes;
```

```
40
41      char remoteIP[INET6_ADDRSTRLEN];
42
43      int yes=1;        // para setsockopt() SO_REUSEADDR,
    abaixo
44      int i, j, rv;
45
46      struct addrinfo hints, *ai, *p;
47
48      FD_ZERO(&master);    // limpa sets master e
    temporário
49      FD_ZERO(&read_fds);
50
51      // nos dá um socket e faz bind nele
52      memset(&hints, 0, sizeof hints);
53      hints.ai_family = AF_UNSPEC;
54      hints.ai_socktype = SOCK_STREAM;
55      hints.ai_flags = AI_PASSIVE;
56      if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) !=
    0) {
57          fprintf(stderr, "selectserver: %s\n",
    gai_strerror(rv));
58          exit(1);
59      }
60
61      for(p = ai; p != NULL; p = p->ai_next) {
62          listener = socket(p->ai_family, p->ai_socktype,
    p->ai_protocol);
63          if (listener < 0) {
64              continue;
65          }
66
67          // evitar a irritante mensagem de erro "endereço
    já em uso"
68          setsockopt(listener, SOL_SOCKET, SO_REUSEADDR,
    &yes, sizeof(int));
69
70          if (bind(listener, p->ai_addr, p->ai_addrlen) <
    0) {
71              close(listener);
72              continue;
73          }
74
75          break;
76      }
```

```
77
78      //se chegou até aqui, significa que não fomos
    impedidos
79      if (p == NULL) {
80          fprintf(stderr, "selectserver: failed to
    bind\n");
81          exit(2);
82      }
83
84      freeaddrinfo(ai); // tudo feito com isso
85
86      // listen
87      if (listen(listener, 10) == -1) {
88          perror("listen");
89          exit(3);
90      }
91
92      // adicione listener ao set master
93      FD_SET(listener, &master);
94
95      // acompanhe o maior descritor de arquivos
96      fdmax = listener; // até agora, é esse aqui
97
98      // loop principal
99      for(;;) {
100         read_fds = master; // copie
101         if (select(fdmax+1, &read_fds, NULL, NULL, NULL)
    == -1) {
102             perror("select");
103             exit(4);
104         }
105
106         // percorrer as conexões existentes à procura de
    dados para ler
107         for(i = 0; i <= fdmax; i++) {
108             if (FD_ISSET(i, &read_fds)) { // nós temos
    um!!
109                 if (i == listener) {
110                     // manuseia novas conexões
111                     addrlen = sizeof remoteaddr;
112                     newfd = accept(listener,
113                         (struct sockaddr *)&remoteaddr,
114                         &addrlen);
115
116                     if (newfd == -1) {
```

```
117                         perror("accept");
118                     } else {
119                         FD_SET(newfd, &master); //
    adiciona ao set master
120                         if (newfd > fdmax) {    //
    acompanhe o máximo
121                             fdmax = newfd;
122                         }
123                         printf("selectserver: nova
    conexão de %s no "
124                             "socket %d\n",
125
    inet_ntop(remoteaddr.ss_family,
126                                 get_in_addr((struct
    sockaddr*)&remoteaddr),
127                                 remoteIP,
    INET6_ADDRSTRLEN),
128                             newfd);
129                     }
130                 } else {
131                     // manipular dados do cliente
132                     if ((nbytes = recv(i, buf, sizeof
    buf, 0)) <= 0) {
133                         // há erro ou conexão fechada
    pelo cliente
134                         if (nbytes == 0) {
135                             // conexão fechada
136                             printf("selectserver: socket
    %d desligado\n", i);
137                         } else {
138                             perror("recv");
139                         }
140                         close(i); // tchau!
141                         FD_CLR(i, &master); // remove do
    set master
142                     } else {
143                         // nós temos alguns dados de um
    cliente
144                         for(j = 0; j <= fdmax; j++) {
145                             // envia para todos!
146                             if (FD_ISSET(j, &master)) {
147                                 // exceto listener e nós
    mesmos
148                                 if (j != listener && j
    != i) {
```

```
149                                     if (send(j, buf,
    nbytes, 0) == -1) {
150                                         perror("send");
151                                     }
152                                 }
153                             }
154                         }
155                     }
156                 } // FIM manipular dados do cliente
157             } // FIM receber nova conexão de entrada
158         } // FIM do loop através dos descritores de
    arquivos
159     } // FIM para (;;)--e você pensou que nunca
    terminaria!
160
161     return 0;
162 }
```

Observe que eu tenho dois sets (conjuntos) de descritores de arquivos no código: `master` e `read_fds`. O primeiro, `master`, detém todos os descritores de socket que estão atualmente conectados, bem como o descritor de socket que está escutando novas conexões.

O motivo pelo qual eu tenho o set `master` é que `select()` na verdade *altera* o conjunto que você passa a ela para refletir que sockets estão prontos para leitura. Como tenho que acompanhar as conexões sucessivas de uma chamada `select()`, devo armazená-las com segurança em algum lugar. No último minuto, eu copio a `master` para `read_fds` e, em seguida, chamo `select()`.

Mas isso não significa que toda vez que recebo uma nova conexão, tenho que adicioná-la ao set `master`? Sim! E cada vez que uma conexão fecha, eu tenho que removê-la do set `master`? Sim.

Repare que eu verifico quando o socket `listener` está pronto para leitura. Quando isso acontece, significa que tenho uma nova conexão pendente, e eu a aceito com `accept()` e adiciono-a ao set `master`. Da mesma forma, quando uma conexão de cliente está pronta para leitura, e `recv()` retorna `0`, eu sei que o cliente fechou a conexão, e devo removê-la do set `master`.

Se o cliente, em `recv()`, retorna diferente de zero, no entanto, eu sei que alguns dados foram recebidos. Então, eu os obtenho, e depois percorrendo a lista `master` envio esses dados para o resto dos clientes conectados.

E isso, meus amigos, é uma visão geral simplificadíssima da poderosa função `select()`.

Nota rápida para todos os fãs de Linux por aí: às vezes, em raras circunstâncias, a `select()` do Linux pode retornar "pronto-para-ler" e, em seguida, não estar realmente pronto para ler! Isso significa que ele bloqueará a `read()` depois que `select()` declarar que não! De qualquer

forma, a solução alternativa é definir o sinalizador `O_NONBLOCK` no socket de recebimento para que se gerem erros com `EWOULDBLOCK` (que você pode ignorar com segurança se ocorrerem). Consulte a página de referência `fcntl()` para obter mais informações sobre como configurar um socket para non-blocking.

Em adição, aqui está um acréscimo tardio como bônus: existe outra função chamada `poll()` que se comporta da mesma maneira que `select()`, mas com um sistema diferente para a gestão dos sets de descritores de arquivos. Confira!

Manipulando send()s parcialmente

Lembra-se da seção sobre `send()`, acima, quando eu disse que `send()` pode não enviar todos os bytes que você pediu? Ou seja, você deseja enviar 512 bytes, mas ela retorna 412. O que aconteceu com os 100 bytes restantes?

Bem, eles ainda estão em seu pequeno buffer esperando para serem enviados. Devido a circunstâncias além do seu controle, o kernel decidiu não enviar todos os dados em um único pedaço, e agora, meu amigo, cabe a você enviar os dados restantes.

Você poderia escrever uma função como essa, para fazer isso, também:

```
1  #include <sys/types.h>
2  #include <sys/socket.h>
3
4  int sendall(int s, char *buf, int *len)
5  {
6      int total = 0;        // quantos bytes enviamos
7      int bytesleft = *len; // quantos temos para enviar
8      int n;
9
10     while(total < *len) {
11         n = send(s, buf+total, bytesleft, 0);
12         if (n == -1) { break; }
13         total += n;
14         bytesleft -= n;
15     }
16
17     *len = total; // retorna número realmente enviado
   aqui
18
19     return n==-1?-1:0; // retorna -1 em falha, 0 em
   successo
20 }
```

Neste exemplo, `s` é o socket para o qual você deseja enviar os dados, `buf` é um buffer contendo os dados, e `len` é um ponteiro para um `int` contendo o número de bytes do buffer.

A função retorna `-1` em caso de erro (e `errno` é definido a partir da chamada a `send()`.) Além disso, o número de bytes realmente enviados é retornado em `len`. Este será o mesmo número de bytes que você pediu para enviar, a menos que tenha ocorrido um erro. `sendall()` fará o melhor possível, enviando os dados para fora, mas se houver um erro ele retornará de volta a você imediatamente.

Para completar, aqui está um exemplo de chamada para a função:

```
1 char buf[10] = "Beej!";
2 int len;
3
4 len = strlen(buf);
5 if (sendall(s, buf, &len) == -1) {
6     perror("sendall");
7     printf("Nós só enviamos %d bytes por causa do
   erro!\n", len);
8 }
```

O que acontece ao final da recepção quando chega apenas parte de um pacote? Se os pacotes são de comprimento variável, como o receptor sabe quando um pacote termina e outro começa? Sim, cenários do mundo real são bem mais complexos. Você provavelmente precisará *encapsular* (lembra-se da seção de encapsulamento de dados lá no começo? Leia-a para mais detalhes!).

Serialização—Como embalar Dados

É fácil enviar dados em texto através da rede, você está descobrindo, mas o que acontece se você quiser enviar alguns dados "binários" como `int` ou `float`? Para isso você tem algumas opções.

1. Converter o número em texto com uma função como `sprintf()` e o enviar. O receptor irá analisar o texto recebido e o converterá em um número usando uma função como `strtol()`.

2. Basta enviar os dados em sua forma bruta, passando o ponteiro dos mesmos para `send()`.

3. Codifique o número em uma forma binária portátil. O receptor o decodificará.

Visualização prévia! Apenas esta noite!

[*Cortinas se abrem*]

Beej diz: "Eu prefiro o Método três, acima!"

[*THE END*]

(Antes de começar esta seção a sério, devo dizer-lhe que existem bibliotecas para fazer isso, e criar o seu próprio código e mantê-lo livre de erros é um grande desafio. Então, procure e faça

sua lição de casa antes de decidir implementar essas coisas você mesmo. Eu incluo as informações aqui para aqueles curiosos sobre como coisas como essa funcionam.)

Na verdade todos os métodos, acima, têm suas vantagens e desvantagens, mas como eu disse, em geral, eu prefiro o terceiro método. Primeiro, porém, vamos falar sobre algumas das vantagens e desvantagens para os outros dois.

O primeiro método, codificando os números como texto antes de enviar, tem a vantagem de que você pode facilmente imprimir e ler os dados que estão vindo pelo fio. Às vezes, um protocolo legível é excelente para uso numa situação que não requeira muita largura de banda como no Internet Relay Chat (IRC)[33]. No entanto, tem a desvantagem de ser lento para converter e os resultados quase sempre ocupam mais espaço do que o número original!

Método dois: enviar os dados brutos. Este é bem fácil (mas perigoso!): basta ter um ponteiro para os dados a serem enviados, e chamar send com ele.

```
double d = 3490.15926535;

send(s, &d, sizeof d, 0);  /* PERIGO--não-portátil! */
```

O receptor recebe assim:

```
double d;

recv(s, &d, sizeof d, 0);  /* PERIGO--não-portátil! */
```

Rápido, simples—O que há para não gostar? Bem, acontece que nem todas as arquiteturas representam um `double` (ou `int` para esse assunto), com a mesma representação de bit ou mesmo a mesma ordenação de bytes! O código é definitivamente não portátil. (Ei, talvez você não precise de portabilidade, e nesse caso isso é bom e rápido.)

Ao empacotar tipos inteiros já vimos como a classe de funções `htons()` pode ajudar a manter as coisas portáteis, convertendo os números para Network Byte Order, e como essa é a coisa certa a se fazer. Infelizmente, não há funções semelhantes para tipos `float`. Toda a esperança está perdida?

Não temas! (Você ficou com medo por um segundo? Não? Nem mesmo um pouco?) Há algo que podemos fazer: podemos embalar (ou "empacotar", ou "serializar", ou um dos outros milhões de nomes) os dados em um formato binário conhecido que o receptor possa descompactar no lado remoto.

O que quero dizer com "formato binário conhecido"? Bem, nós já vimos o exemplo de `htons()`, certo? Ele altera (ou "codifica", se você quiser pensar dessa maneira) um número de host em qualquer formato para Network Byte Order. Para inverter (unencode) o número, o receptor chama `ntohs()`.

Mas eu não acabei de dizer que não havia qualquer função para outros tipos não inteiros? Sim. Eu disse. E uma vez que não há nenhuma forma padrão em C para fazer isso, é um pouco pickle (um trocadilho gratuito para os fãs de Python).

A única coisa a fazer é empacotar os dados em um formato conhecido e enviá-los pelo fio para a decodificação. Por exemplo, para embalar `floats`, aqui está algo rápido e sujo com muito espaço para melhorias[34]:

```
#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 inteira e sinal
    p |= (uint32_t)(((f - (int)f) * 65536.0f))&0xffff; // fração

    return p;
}

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

    if (((p>>31)&0x1) == 0x1) { f = -f; } // conjunto de bits de sinal

    return f;
}
```

O código acima é uma espécie de implementação ingênua que armazena um `float` em um número de 32-bit. O bit alto (31) é usado para armazenar o sinal do número ("1" significa negativo), e os próximos sete bits (30-16) são usados para armazenar a porção inteira do número `float`. Finalmente, os bits restantes (15-0) são usados para armazenar a parte fracionária do número.

O uso é bastante simples:

```
#include <stdio.h>

int main(void)
{
```

```
    float f = 3.1415926, f2;
    uint32_t netf;

    netf = htonf(f);  // converte para formato "network"
    f2 = ntohf(netf); // converter de volta para teste

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

    return 0;
}
```

No lado positivo, é pequeno, simples e rápido. No lado negativo, não é uma utilização eficiente do espaço e o intervalo é severamente restrito—tente armazenar um número maior do que 32767 e será muito infeliz! Você também pode ver no exemplo acima que as últimas duas casas decimais não são corretamente preservadas.

O que podemos fazer em vez disso? Bem, *O* padrão para armazenar números de ponto flutuante é conhecido como IEEE-754 [35]. Muitos computadores usam este formato internamente para fazer contas com ponto flutuante, por isso, nesses casos, estritamente falando, a conversão não precisaria ser feita. Mas se você quiser que o seu código fonte seja portátil essa é uma suposição que você não pode necessariamente fazer. (Por outro lado, se você quer que as coisas sejam rápidas, você deve otimizá-lo em plataformas que não precisam fazê-lo! Isso é o que `htons()` e sua turma fazem.)

Aqui está um código que codifica floats e doubles no formato IEEE 754 [36]. (Principalmente—isso não codifica NaN ou infinito, mas poderia ser modificado para fazer isso.)

```
#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
 para sign bit

    if (f == 0.0) return 0; // tirar este caso especial
 do caminho

```

```
15      // verifique o sinal e comece a normalização
16      if (f < 0) { sign = 1; fnorm = -f; }
17      else { sign = 0; fnorm = f; }
18
19      // obtenha a forma normalizada de f e acompanhe o
    expoente
20      shift = 0;
21      while(fnorm >= 2.0) { fnorm /= 2.0; shift++; }
22      while(fnorm < 1.0) { fnorm *= 2.0; shift--; }
23      fnorm = fnorm - 1.0;
24
25      // calcular a forma binária (não flutuante) dos
    dados significativos
26      significand = fnorm * ((1LL<<significandbits) +
    0.5f);
27
28      // obter o expoente tendencioso
29      exp = shift + ((1<<(expbits-1)) - 1); // shift +
    bias
30
31      // retornar a resposta final
32      return (sign<<(bits-1)) | (exp<<(bits-expbits-1)) |
    significand;
33  }
34
35  long double unpack754(uint64_t i, unsigned bits,
    unsigned expbits)
36  {
37      long double result;
38      long long shift;
39      unsigned bias;
40      unsigned significandbits = bits - expbits - 1; // -1
    para bit de sinal
41
42      if (i == 0) return 0.0;
43
44      // puxe o significando
45      result = (i&((1LL<<significandbits)-1)); // mascarar
46      result /= (1LL<<significandbits); // converter de
    volta para float
47      result += 1.0f; // adicione o de volta
48
49      // lidar com o expoente
50      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++; }

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

    return result;
}
```

Eu coloquei alguns macros úteis lá em cima no topo para embalar e desembalar números 32 bits (provavelmente um `float`) e 64 bits (provavelmente um `double`), mas a função `pack754()` poderia ser chamada diretamente e informada para codificar `bits` de dados (`expbits` dos quais estão reservados para o expoente do número normalizado.)

Aqui está um exemplo de uso:

```

#include <stdio.h>
#include <stdint.h> // definir tipos uintN_t
#include <inttypes.h> // definir 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 antes  : %.7f\n", f);
    printf("float encoded: 0x%08" PRIx32 "\n", fi);
    printf("float depois : %.7f\n\n", f2);

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

```

```
27      return 0;
28  }
```

O código acima produz esta saída:

```
float antes   : 3.1415925
float encoded : 0x40490FDA
float depois  : 3.1415925

double antes  : 3.14159265358979311600
double encoded: 0x400921FB54442D18
double depois : 3.14159265358979311600
```

Outra questão que você pode ter é: como embalar `structs`? Infelizmente para você, o compilador é livre para colocar padding em todos os lugares em uma `struct`, e isso significa que você não pode tornar partátil e enviar a coisa toda pelo fio em um pedaço. (Você não está ficando cansado de ouvir "não pode fazer isso", "não pode fazer isso"? Desculpe! Para citar um amigo: "Sempre que algo dá errado, eu sempre culpo a Microsoft". Esta pode não ser uma culpa da Microsoft, admito, mas a afirmação do meu amigo é completamente verdadeira.)

Voltando ao assunto: a melhor maneira para enviar uma `struct` através do fio é embalar cada campo, independentemente, e, em seguida, desempacotá-los para a `struct` quando chegarem ao outro lado.

Isso é muito trabalhoso, é o que você está pensando. Sim. Uma coisa que você pode fazer é escrever uma função auxiliar para ajudar a embalar os dados para você. Vai ser divertido! Realmente!

No livro *The Practice of Programming*[37] de Kernighan e Pike, eles implementam `printf()`-como funções chamadas `pack()` e `unpack()` que fazem exatamente isso. Eu teria um link para elas mas, aparentemente, essas funções não estão online com o resto dos fontes do livro.

(The Practice of Programming é uma excelente leitura. Zeus salva um gatinho cada vez que eu a recomendo.)

Neste ponto, largarei um ponteiro para o Protocol Buffers implementation in C[38] que eu nunca usei, mas parece completamente respeitável. Os programadores Python e Perl podem verificar as funções `pack()` e `unpack()` de sua linguagem para realizarem a mesma coisa. E o Java possui uma interface serializável que pode ser usada de maneira semelhante.

Mas se você quiser escrever seu próprio utilitário de empacotamento em C, o truque de K&P é usar listas de argumentos variáveis para criar funções como `printf()` para construir os pacotes. Aqui está a versão que eu criei [39] sozinho com base naquilo que espero ser suficiente para lhe dar uma idéia de como isso pode funcionar.

(Este código faz referência às funções `pack754()` acima. As funções `packi*()` operam de forma similar as da família `htons()`, exceto por elas embalarem array de `char` em vez de outro inteiro.)

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

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

/* packi32() -- armazena um int de 16 bits em um buffer
** de char (como htons ())
*/
void packi32(unsigned char *buf, unsigned long int i)
{
    *buf++ = i>>24; *buf++ = i>>16;
    *buf++ = i>>8;  *buf++ = i;
}

/* packi64() -- armazena um int de 64 bits em um buffer
** de char (como htonl())
*/
void packi64(unsigned char *buf, unsigned long long int
 i)
{
    *buf++ = i>>56; *buf++ = i>>48;
    *buf++ = i>>40; *buf++ = i>>32;
    *buf++ = i>>24; *buf++ = i>>16;
    *buf++ = i>>8;  *buf++ = i;
}

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

```

```
42      // alterar números unsigned para signed
43      if (i2 <= 0x7fffu) { i = i2; }
44      else { i = -1 - (unsigned int)(0xffffu - i2); }
45
46      return i;
47  }
48
49  /* unpacku16() -- descompacta um 16 bits unsigned de um
     buffer
50  ** de char (como ntohs())
51  */
52  unsigned int unpacku16(unsigned char *buf)
53  {
54      return ((unsigned int)buf[0]<<8) | buf[1];
55  }
56
57  /* unpacki32() -- descompacta um int de 32 bits de um
     buffer
58  ** de char (como ntohl())
59  */
60  long int unpacki32(unsigned char *buf)
61  {
62      unsigned long int i2 = ((unsigned long
     int)buf[0]<<24) |
63                             ((unsigned long
     int)buf[1]<<16) |
64                             ((unsigned long
     int)buf[2]<<8)  |
65                             buf[3];
66      long int i;
67
68      // alterar números unsigned para signed
69      if (i2 <= 0x7fffffffu) { i = i2; }
70      else { i = -1 - (long int)(0xffffffffu - i2); }
71
72      return i;
73  }
74
75  /* unpacku32() -- descompacta um 32 bits unsigned de um
     buffer
76  ** de char (como ntohl())
77  */
78  unsigned long int unpacku32(unsigned char *buf)
79  {
80      return ((unsigned long int)buf[0]<<24) |
```

```
81              ((unsigned long int)buf[1]<<16) |
82              ((unsigned long int)buf[2]<<8)  |
83              buf[3];
84  }
85
86  /* unpacki64() -- descompacta um int de 64 bits de um
     buffer
87  ** de char (como ntohl())
88  */
89  long long int unpacki64(unsigned char *buf)
90  {
91      unsigned long long int i2 = ((unsigned long long
     int)buf[0]<<56) |
92                                  ((unsigned long long
     int)buf[1]<<48) |
93                                  ((unsigned long long
     int)buf[2]<<40) |
94                                  ((unsigned long long
     int)buf[3]<<32) |
95                                  ((unsigned long long
     int)buf[4]<<24) |
96                                  ((unsigned long long
     int)buf[5]<<16) |
97                                  ((unsigned long long
     int)buf[6]<<8)  |
98                                  buf[7];
99      long long int i;
100
101     // alterar números unsigned para signed
102     if (i2 <= 0x7fffffffffffffffu) { i = i2; }
103     else { i = -1 -(long long int)(0xffffffffffffffffu -
     i2); }
104
105     return i;
106 }
107
108 /* unpacku64() -- descompacta um 64 bits unsigned de um
     buffer
109 ** de char (como ntohl())
110 */
111 unsigned long long int unpacku64(unsigned char *buf)
112 {
113     return ((unsigned long long int)buf[0]<<56) |
114            ((unsigned long long int)buf[1]<<48) |
115            ((unsigned long long int)buf[2]<<40) |
116            ((unsigned long long int)buf[3]<<32) |
```

```
117             ((unsigned long long int)buf[4]<<24) |
118             ((unsigned long long int)buf[5]<<16) |
119             ((unsigned long long int)buf[6]<<8)  |
120             buf[7];
121 }
122 
123 /*
124 ** pack() -- armazenar dados ditados pelo formato string
     no buffer
125 **
126 **   bits |signed   unsigned   float   string
127 **   -----+----------------------------------
128 **      8 |   c        C
129 **     16 |   h        H         f
130 **     32 |   l        L         d
131 **     64 |   q        Q         g
132 **      - |                              s
133 ** (O comprimento de um 16 bits unsigned é anexado
134 **  automaticamente às strings)
135 */
136 
137 unsigned int pack(unsigned char *buf, char *format, ...)
138 {
139     va_list ap;
140 
141     signed char c;              // 8-bit
142     unsigned char C;
143 
144     int h;                      // 16-bit
145     unsigned int H;
146 
147     long int l;                 // 32-bit
148     unsigned long int L;
149 
150     long long int q;            // 64-bit
151     unsigned long long int Q;
152 
153     float f;                    // floats
154     double d;
155     long double g;
156     unsigned long long int fhold;
157 
158     char *s;                    // strings
159     unsigned int len;
```

```
160
161     unsigned int size = 0;
162
163     va_start(ap, format);
164
165     for(; *format != '\0'; format++) {
166         switch(*format) {
167         case 'c': // 8-bit
168             size += 1;
169             c = (signed char)va_arg(ap, int); //
    promovido
170             *buf++ = c;
171             break;
172
173         case 'C': // 8-bit unsigned
174             size += 1;
175             C = (unsigned char)va_arg(ap, unsigned int);
    // promovido
176             *buf++ = C;
177             break;
178
179         case 'h': // 16-bit
180             size += 2;
181             h = va_arg(ap, int);
182             packi16(buf, h);
183             buf += 2;
184             break;
185
186         case 'H': // 16-bit unsigned
187             size += 2;
188             H = va_arg(ap, unsigned int);
189             packi16(buf, H);
190             buf += 2;
191             break;
192
193         case 'l': // 32-bit
194             size += 4;
195             l = va_arg(ap, long int);
196             packi32(buf, l);
197             buf += 4;
198             break;
199
200         case 'L': // 32-bit unsigned
201             size += 4;
```

```
202             L = va_arg(ap, unsigned long int);
203             packi32(buf, L);
204             buf += 4;
205             break;
206 
207         case 'q': // 64-bit
208             size += 8;
209             q = va_arg(ap, long long int);
210             packi64(buf, q);
211             buf += 8;
212             break;
213 
214         case 'Q': // 64-bit unsigned
215             size += 8;
216             Q = va_arg(ap, unsigned long long int);
217             packi64(buf, Q);
218             buf += 8;
219             break;
220 
221         case 'f': // float-16
222             size += 2;
223             f = (float)va_arg(ap, double); // promovido
224             fhold = pack754_16(f); // converte para IEEE
    754
225             packi16(buf, fhold);
226             buf += 2;
227             break;
228 
229         case 'd': // float-32
230             size += 4;
231             d = va_arg(ap, double);
232             fhold = pack754_32(d); // converte para IEEE
    754
233             packi32(buf, fhold);
234             buf += 4;
235             break;
236 
237         case 'g': // float-64
238             size += 8;
239             g = va_arg(ap, long double);
240             fhold = pack754_64(g); // converte para IEEE
    754
241             packi64(buf, fhold);
242             buf += 8;
```

```
243             break;
244 
245         case 's': // string
246             s = va_arg(ap, char*);
247             len = strlen(s);
248             size += len + 2;
249             packi16(buf, len);
250             buf += 2;
251             memcpy(buf, s, len);
252             buf += len;
253             break;
254         }
255     }
256 
257     va_end(ap);
258 
259     return size;
260 }
261 
262 /* unpack() -- descompacta os dados ditados pelo formato
263 ** string no buffer
264 **
265 **   bits |signed   unsigned   float   string
266 **   -----+----------------------------------
267 **      8 |   c        C
268 **     16 |   h        H         f
269 **     32 |   l        L         d
270 **     64 |   q        Q         g
271 **      - |                               s
272 ** (A string é extraída com base no comprimento
     armazenado,
273 ** mas 's' pode ser anexado com um comprimento máximo)
274 **
275 */
276 void unpack(unsigned char *buf, char *format, ...)
277 {
278     va_list ap;
279 
280     signed char *c;              // 8-bit
281     unsigned char *C;
282 
283     int *h;                     // 16-bit
284     unsigned int *H;
285 
```

```
286     long int *l;                // 32-bit
287     unsigned long int *L;
288
289     long long int *q;           // 64-bit
290     unsigned long long int *Q;
291
292     float *f;                   // floats
293     double *d;
294     long double *g;
295     unsigned long long int fhold;
296
297     char *s;
298     unsigned int len, maxstrlen=0, count;
299
300     va_start(ap, format);
301
302     for(; *format != '\0'; format++) {
303         switch(*format) {
304         case 'c': // 8-bit
305             c = va_arg(ap, signed char*);
306             if (*buf <= 0x7f) { *c = *buf;} // re-sign
307             else { *c = -1 - (unsigned char)(0xffu -
    *buf); }
308             buf++;
309             break;
310
311         case 'C': // 8-bit unsigned
312             C = va_arg(ap, unsigned char*);
313             *C = *buf++;
314             break;
315
316         case 'h': // 16-bit
317             h = va_arg(ap, int*);
318             *h = unpacki16(buf);
319             buf += 2;
320             break;
321
322         case 'H': // 16-bit unsigned
323             H = va_arg(ap, unsigned int*);
324             *H = unpacku16(buf);
325             buf += 2;
326             break;
327
328         case 'l': // 32-bit
```

```
329             l = va_arg(ap, long int*);
330             *l = unpacki32(buf);
331             buf += 4;
332             break;
333 
334         case 'L': // 32-bit unsigned
335             L = va_arg(ap, unsigned long int*);
336             *L = unpacku32(buf);
337             buf += 4;
338             break;
339 
340         case 'q': // 64-bit
341             q = va_arg(ap, long long int*);
342             *q = unpacki64(buf);
343             buf += 8;
344             break;
345 
346         case 'Q': // 64-bit unsigned
347             Q = va_arg(ap, unsigned long long int*);
348             *Q = unpacku64(buf);
349             buf += 8;
350             break;
351 
352         case 'f': // float
353             f = va_arg(ap, float*);
354             fhold = unpacku16(buf);
355             *f = unpack754_16(fhold);
356             buf += 2;
357             break;
358 
359         case 'd': // float-32
360             d = va_arg(ap, double*);
361             fhold = unpacku32(buf);
362             *d = unpack754_32(fhold);
363             buf += 4;
364             break;
365 
366         case 'g': // float-64
367             g = va_arg(ap, long double*);
368             fhold = unpacku64(buf);
369             *g = unpack754_64(fhold);
370             buf += 8;
371             break;
372 
```

```
373         case 's': // string
374             s = va_arg(ap, char*);
375             len = unpacku16(buf);
376             buf += 2;
377             if (maxstrlen > 0 && len >= maxstrlen) count
 = maxstrlen - 1;
378             else count = len;
379             memcpy(s, buf, count);
380             s[count] = '\0';
381             buf += len;
382             break;
383 
384         default:
385             if (isdigit(*format)) { // segue max str len
386                 maxstrlen = maxstrlen * 10 +
 (*format-'0');
387             }
388         }
389 
390         if (!isdigit(*format)) maxstrlen = 0;
391     }
392 
393     va_end(ap);
394 }
```

E aqui está um programa de demonstração [40] do código acima que embala alguns dados em `buf` e, em seguida, os descompacta em variáveis. Note que quando chamamos `unpack()` com um argumento em string (especificador de formato "s"), é aconselhável colocar uma contagem de comprimento máximo na frente para evitar uma saturação de buffer, por exemplo, "96s". Seja cauteloso ao descompactar dados recebidos pela rede—um usuário malicioso pode enviar pacotes mal construídas em um esforço para atacar seu sistema!

```
1  #include <stdio.h>
2  
3  // vários bits para tipos de ponto flutuante--
4  // varia para diferentes arquiteturas
5  typedef float float32_t;
6  typedef double float64_t;
7  
8  int main(void)
9  {
10     unsigned char buf[1024];
11     int8_t magic;
12     int16_t monkeycount;
```

```
13      int32_t altitude;
14      float32_t absurdityfactor;
15      char *s = "Great unmitigated Zot! You've found the
    Runestaff!";
16      char s2[96];
17      int16_t packetsize, ps2;
18
19      packetsize = pack(buf, "chhlsf", (int8_t)'B',
    (int16_t)0, (int16_t)37,
20              (int32_t)-5, s, (float32_t)-3490.6677);
21      packi16(buf+1, packetsize); // guarda tamanho do
    pacote no pacote
22
23      printf("packet é %" PRId32 " bytes\n", packetsize);
24
25      unpack(buf, "chhl96sf", &magic, &ps2, &monkeycount,
    &altitude, s2,
26          &absurdityfactor);
27
28      printf("'%c' %" PRId32" %" PRId16 " %" PRId32
29              " \"%s\" %f\n", magic, ps2, monkeycount,
30              altitude, s2, absurdityfactor);
31
32      return 0;
33  }
```

Se você trabalha com seu próprio código ou usa o de outra pessoa, é uma boa idéia ter um conjunto geral de rotinas de embalagem de dados para manter os bugs sob controle, ao invés de embalar cada bit manualmente a cada vez.

Ao embalar os dados, qual é o melhor formato a ser usado? Excelente questão. Felizmente, a RFC 4506 [41], o Padrão de Representação de Dados Externos, já define formatos binários para diferentes tipos, como tipos float, tipos int, arrays, raw data, etc. Eu sugiro que se conforme com isso, se você estiver trabalhando com os dados sozinho. Mas você não é obrigado a isso. A Polícia de Pacotes não estará mesmo à sua porta. Pelo menos, eu não *acredito* que estará.

Em qualquer caso, codificar os dados de uma forma ou de outra antes de enviá-los é a maneira certa de fazer as coisas!

Bases do encapsulamento de dados

O que realmente significa encapsular dados, de qualquer maneira? No caso mais simples, significa que você colocará um cabeçalho lá com algumas informações de identificação ou o comprimento do pacote, ou ambos.

Como deve ser seu cabeçalho? Bem, são apenas alguns dados binários que representam o que você acha necessário para concluir seu projeto.

Uau. Isso é vago.

Ok. Por exemplo, digamos que você tenha um programa de chat multi-usuário que use `SOCK_STREAM`. Quando um usuário digita ("diz") algo, dois pedaços de informação precisam ser transmitidos ao servidor: o que foi dito e quem disse.

Até aí tudo bem? "Qual é o problema?" você está perguntando.

O problema é que as mensagens podem ter comprimentos variados. Uma pessoa chamada "tom" pode dizer: "Olá", e outra pessoa chamada "Benjamin" pode dizer: "Ei, pessoal, o que aconteceu?"

Então você envia com `send()` todas essas coisas para os clientes quando chegarem a você. Seu fluxo de dados de saída se parece com isso:

```
t o m O l á B e n j a m i n E i , p e s s o a l , o q u e a c o n t
e c e u ?
```

E assim por diante. Como o cliente sabe quando uma mensagem termina e outra começa? Você poderia, se quisesse, fazer com que todas as mensagens tivessem o mesmo comprimento e apenas chamar `sendall()` que implementamos, acima. Mas isso desperdiça largura de banda! Nós não queremos executar `send()` com 1024 bytes apenas para que "Tom" possa dizer "Olá".

Por isso, *encapsulamos* os dados em uma pequena estrutura de cabeçalho e corpo. Tanto o cliente quanto o servidor sabem como compactar e descompactar (por vezes referido como "marshal" e "unmarshal") esses dados. Não olhe agora, mas estamos começando a definir um *protocolo* que descreve como um cliente e servidor comunicam-se!

Neste caso, vamos supor que o nome de usuário tenha um comprimento fixo de 8 caracteres, terminados com `'\0'`. E então vamos supor que os dados tenham comprimento variável, até um máximo de 128 caracteres. Vamos dar uma olhada em uma estrutura de pacote num exemplo que poderíamos usar nesta situação:

1. `len` (1 byte, sem sinal)—O comprimento total do pacote, contendo 8 bytes do user name e chat data.
2. `name` (8 bytes)—O nome do usuário, NUL-preenchido se necessário.
3. `chatdata` (*n*-bytes)—Os dados em si, não mais do que 128 bytes. O comprimento do pacote deve ser calculado como o comprimento dos dados mais 8 (o comprimento do campo do nome, acima).

Por que escolhi os limites de 8 bytes e 128 bytes para os campos? Eu os puxei para fora pelo fio, assumindo que seriam longos o suficiente. Talvez, porém, 8 bytes sejam muito restritivos para as suas necessidades, e você pode ter um campo de nome com 30 bytes, ou quantos queira que sejam. A escolha é sua.

Usando a definição de pacotes acima, o primeiro pacote consistiria nas seguintes informações (em hexadecimal e ASCII):

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

And the second is 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  ...
```

(O comprimento é armazenado em Network Byte Order, é claro. Neste caso, é apenas um byte, por isso não importa, mas em geral você desejará que todos os seus inteiros binários sejam armazenados em Network Byte Order em seus pacotes.)

Quando você está enviando esses dados, você deve estar seguro e usar um comando semelhante a `sendall()`, acima, para que você saiba que todos os dados são enviados, mesmo que sejam necessárias várias chamadas a `send()` para colocar tudo para fora.

Da mesma forma, quando você está recebendo esses dados, precisa fazer um pouco mais de trabalho. Para estar seguro, você deve assumir que pode receber um pacote parcial (como talvez recebamos "`18 42 65 6E 6A`" de Benjamin, acima, mas isso é tudo o que recebemos nesta chamada `recv()`). Precisamos chamar `recv()` repetidamente até que o pacote seja completamente recebido.

Mas como? Bem, nós sabemos o número de bytes que precisamos receber no total para que o pacote esteja completo, uma vez que o número é inserido na frente do pacote. Sabemos também que o tamanho máximo do pacote é 1+8+128, ou 137 bytes (porque é assim que definimos o pacote.)

Na verdade, existem algumas coisas que você pode fazer aqui. Como você sabe que cada pacote começa com um comprimento, você pode chamar `recv()` apenas para obter o tamanho do pacote. Então uma vez que você tenha isso, você pode chamá-la novamente especificando exatamente o comprimento restante do pacote (possivelmente repetidamente para obter todos os dados) até que você tenha o pacote completo. A vantagem deste método é que você só precisa de um buffer grande suficiente para um pacote, enquanto a desvantagem é que você precisa chamar `recv()` pelo menos duas vezes para obter todos os dados.

Outra opção é apenas chamar `recv()` e dizer que o valor que você está disposto a receber é o número máximo de bytes em um pacote. Então, o que quer que você receba, coloque-o na parte de trás de um buffer, e, finalmente, verifique se o pacote está completo. Claro, você pode pegar um pouco do próximo pacote, então você precisa ter espaço para isso.

O que você pode fazer é declarar uma matriz grande o suficiente para dois pacotes. Este é o seu array de trabalho onde você irá reconstruir os pacotes conforme eles chegam.

Cada vez que você receber dados com `recv()`, você os anexará ao buffer de trabalho e verificará se o pacote está completo. Ou seja, o número de bytes no buffer é maior ou igual ao comprimento especificado no cabeçalho (+1, porque o comprimento no cabeçalho não inclui o

byte para o próprio comprimento). Se o número de bytes no buffer for menor que 1, o pacote não está completo, obviamente. Você tem que fazer um case especial para isso, porém, desde que o primeiro byte é lixo e você não pode contar com ele para o comprimento correto do pacote.

Quando o pacote estiver completo, você poderá fazer com ele o que quiser. Use-o e remova-o do seu buffer de trabalho.

Ufa! Você ainda está fazendo malabarismos com isso na sua cabeça? Bem, aqui está mais complexidade: você pode ter lido e passado do fim de um pacote e lido parte do próximo em uma única chamada `recv()`. Ou seja, você tem um buffer de trabalho com um pacote completo e uma parte incompleta do próximo pacote! Maldito. (Mas foi por isso que você fez o seu buffer de trabalho suficientemente grande para conter *dois* pacotes—caso isso acontecesse!)

Uma vez que você saiba o tamanho do primeiro pacote do cabeçalho, e você está mantendo o controle do número de bytes no buffer de trabalho, você pode subtrair e calcular quantos dos bytes no buffer de trabalho pertencem ao segundo pacote (incompleto). Quando você já lidou com o primeiro, você pode removê-lo do buffer de trabalho e mover o segundo pacote parcial para frente no buffer para que ele esteja pronto para o próximo `recv()`.

(Alguns de vocês leitores notarão que mover o segundo pacote parcial para o início do buffer de trabalho leva tempo, e o programa pode ser codificado para não exigir isso usando um buffer circular. Infelizmente para o resto de vocês, uma discussão sobre buffers circulares está além do escopo deste artigo. Se você ainda está curioso, pegue um livro sobre estruturas de dados e siga a partir daí.)

Eu nunca disse que era fácil. Ok, eu disse que era fácil. E isso é: você só precisa praticar e muito em breve se tornará natural. Juro por Excalibur!

Pacotes Broadcast—Olá, mundo!

Até agora, este guia falou sobre o envio de dados de um host para um outro host. Mas é possível, insisto, que você possa, com a devida autoridade, enviar dados para vários hosts *ao mesmo tempo*!

Com UDP (somente UDP, não TCP) e IPv4 padrão, isso é feito através de um mecanismo chamado *broadcasting*. Com o IPv6, broadcasting não é suportado e você precisa recorrer à técnica frequentemente superior de *multicasting*, que, infelizmente, eu não estarei discutindo neste momento. Mas o suficiente para espiarmos o futuro—estamos presos no presente de 32 bits.

Mas espere! Você não pode simplesmente sair daqui e começar seu broadcasting de forma precipitada; Você precisa definir a opção de socket `SO_BROADCAST` antes de pode enviar um pacote broadcast para a rede. É como uma daqueles pequenas tampas de plástico que eles colocaram sobre o interruptor de lançamento do míssil! Isso é o quanto de poder você tem suas mãos!

Mas, falando sério, existe o perigo de usar pacotes broadcast, ou seja: todo sistema que recebe um pacote broadcast deve desfazer todas as camadas de encapsulamento de dados até que descubra-se a que porta os dados são destinados. E então entrega os dados ou os

descarta. Em ambos os casos, é muito trabalhoso para cada máquina que recebe o pacote broadcast, e como trafegam todos na rede local pode haver muitas máquinas fazendo trabalho desnecessário. Quando o jogo Doom apareceu pela primeira vez, isso era uma reclamação sobre seu código de rede.

Agora, há mais de uma maneira de esfolar um gato... espere um minuto. Existe realmente mais do que uma maneira de esfolar um gato? Que tipo de expressão é essa? E, da mesma forma, há mais de uma maneira de enviar um pacote broadcast. Então, para chegar à carne e às batatas da coisa toda: como você especifica o endereço de destino para uma mensagem de broadcast? Existem duas formas comuns:

1. Envie os dados para o endereço de broadcast de uma sub-rede específica. Esse é o número de rede da sub-rede com todas os bits um definidos para a parte de host do endereço. Por exemplo, em casa minha rede é `192.168.1.0`, a minha máscara de rede é `255.255.255.0`, então o último byte do endereço é meu número de host (porque os três primeiros bytes, de acordo com a máscara de rede, são o número da rede). Então, meu endereço de broadcast é `192.168.1.255`. No Unix, o comando `ifconfig` fornecerá todos esses dados. (Se você está curioso, a lógica bitwise para obter o seu endereço de broadcast é `network_number` OR (NOT `netmask`).) Você pode enviar este tipo de pacote broadcast para redes remotas, bem como para a sua rede local, mas você corre o risco de o pacote ser descartado pelo roteador de destino. (Se eles não o descartassem, algum smurf aleatório poderia começar a inundar a sua LAN com tráfego de broadcast.)

2. Envie os dados para o endereço de broadcast "global". Isso é `255.255.255.255`, também conhecido como `INADDR_BROADCAST`. Muitas máquinas realizam operações bitwise AND com o seu número de rede para o converter em um endereço de broadcast, mas algumas não. Varia. Roteadores não encaminham este tipo de pacote broadcast para fora da sua rede local, ironicamente.

Então, o que acontece se você tentar enviar dados para o endereço de broadcast sem antes definir a opção `SO_BROADCAST` no socket? Bem, vamos até o bom e velho `talker e listener` ver o que acontece.

```
$ 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
```

Sim, nem tudo funcionou... porque não definimos a opção `SO_BROADCAST` para o socket. Faça isso, e então você poderá executar `sendto()` em qualquer lugar!

Na verdade, essa é a única *diferença* entre um aplicativo UDP que pode transmitir e outro que não pode. Então, vamos pegar o antigo programa `talker` e adicionar uma seção que defina a opção `SO_BROADCAST` para o socket. Vamos chamar este programa `broadcaster.c` [42]:

```
1  /*
```

```c
/* broadcaster.c -- um "cliente" de datagrama como o talker.c, exceto
**                  que este faz broadcast
*/

#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 // a porta onde os usuários se conectarão

int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in their_addr; // informações de endereço do conector
    struct hostent *he;
    int numbytes;
    int broadcast = 1;
    //char broadcast = '1'; // se isso não funcionar, tente isso

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

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

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

```
40          exit(1);
41      }
42
43      // esta chamada é o que permite o envio de pacotes
    de broadcast:
44      if (setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST,
    &broadcast,
45          sizeof broadcast) == -1) {
46          perror("setsockopt (SO_BROADCAST)");
47          exit(1);
48      }
49
50      their_addr.sin_family = AF_INET;     // host byte
    order
51      their_addr.sin_port = htons(SERVERPORT); // short,
    network byte order
52      their_addr.sin_addr = *((struct in_addr
    *)he->h_addr);
53      memset(their_addr.sin_zero, '\0', sizeof
    their_addr.sin_zero);
54
55      if ((numbytes=sendto(sockfd, argv[2],
    strlen(argv[2]), 0,
56              (struct sockaddr *)&their_addr, sizeof
    their_addr)) == -1) {
57          perror("sendto");
58          exit(1);
59      }
60
61      printf("enviados %d bytes para %s\n", numbytes,
62          inet_ntoa(their_addr.sin_addr));
63
64      close(sockfd);
65
66      return 0;
67  }
```

O que há de diferente entre esta e uma situação de cliente/servidor UDP "normal"? Nada! (Com a exceção do cliente ter permissão para enviar pacotes broadcast nesse caso.) Dessa forma, vá em frente e execute o velho programa UDP `listener` em uma janela e o `broadcaster` em outra. Você deve agora ser capaz de fazer todos aqueles envios que falharam acima.

```
$ broadcaster 192.168.1.2 foo
enviados 3 bytes para 192.168.1.2
$ broadcaster 192.168.1.255 foo
```

```
enviados 3 bytes para 192.168.1.255
$ broadcaster 255.255.255.255 foo
enviados 3 bytes para 255.255.255.255
```

E você deve ver `listener` respondendo que recebeu os pacotes. (Se `listener` não responder, pode ser porque ele está vinculado a um endereço IPv6. Tente alterar `AF_UNSPEC` em `listener.c` para `AF_INET` para forçar IPv4).

Bem, isso é excitante. Mas agora ative o `listener` em outra máquina próxima a você na mesma rede para que você tenha duas cópias, uma em cada máquina, e execute `broadcaster` novamente com o seu endereço de broadcast... Ei! Ambos `listener` recebem o pacote embora você só tenha feito uma chamada a `sendto()`! Legal!

Se `listener` receber os dados enviados diretamente a ele, mas não os dados enviados ao endereço de broadcast, pode ser que você tenha um firewall em sua máquina local que esteja bloqueando os pacotes. (Sim, Pat e Bapper, obrigado por perceber antes de mim, era por isso que o meu código de exemplo não estava funcionando. Eu lhe disse que o mencionaria no guia, e você está aqui. Então *obrigado*.)

Mais uma vez, tenha cuidado com pacotes broadcast. Uma vez que cada máquina na LAN será forçada a lidar com o pacote quer ela receba com `recvfrom()` ou não, ele pode apresentar uma grande carga para o toda a rede de computação. Definitivamente eles devem ser usados com moderação e de forma adequada.

Dúvidas Frequentes

Onde posso obter os arquivos de cabeçalho?

Se você não os tiver em seu sistema, provavelmente não precisará deles. Verifique o manual da sua plataforma específica. Se você está construindo para Windows, você só precisa de `#include <winsock.h>`.

O que eu faço quando `bind()` relata "Endereço já em uso"?

Você precisa usar `setsockopt()` com a opção `SO_REUSEADDR` no socket de escuta. Confira a seção de `bind()` e a seção de `select()` para exemplos.

Como faço para obter uma lista de sockets abertos no sistema?

Use o `netstat`. Verifique suas páginas de `manual` para mais detalhes, mas você deve obter uma boa saída apenas digitando:

```
$ netstat
```

O único truque é determinar qual socket está associado a qual programa. `:-)`

Como posso visualizar a tabela de roteamento?

Execute o comando `route` (em `/sbin` na maioria dos Linuxes) ou o comando `netstat -r`.

Como posso executar os programas cliente e servidor se eu tiver apenas um computador? Não preciso de uma rede para escrever programas de rede?

Felizmente, praticamente todas as máquinas implementam um "dispositivo" de rede de loopback que fica no kernel e finge ser uma placa de rede. (Esta é a interface listada como "`lo`" na tabela de roteamento.)

Finja que você está conectado a uma máquina chamada "`cabra`". Execute o cliente em uma janela e o servidor em outra. Ou inicie o servidor em segundo plano ("`server &`") e execute o cliente na mesma janela. A conclusão do dispositivo de loopback é que você pode executar `cliente cabra` ou `cliente localhost` (Uma vez que "`localhost`" está provavelmente definido no seu arquivo `/etc/hosts`) e você terá o cliente conversando com o servidor sem uma rede!

Em suma, nenhuma alteração é necessária, em qualquer código, para o fazer funcionar em uma única máquina que não esteja em rede! Uhull!

Como posso saber se o lado remoto fechou a conexão?

Você pode saber verificando se `recv()` retornou `0`.

Como faço para implementar um utilitário "ping"? O que é ICMP? Onde posso encontrar mais informações sobre raw sockets e `SOCK_RAW`?

Todas as suas questões sobre raw sockets serão respondidas nos W. Richard Stevens' UNIX Network Programming books. Além disso, procure no subdiretório `ping/` no Stevens' UNIX Network Programming source code, disponível online [43].

Como posso alterar ou encurtar o tempo de espera em uma chamada à `connect()`?

Em vez de dar-lhe exatamente a mesma resposta que W. Richard Stevens lhe daria, eu apenas indicarei `lib/connect_nonb.c` in the UNIX Network Programming source code [44].

A essência disso é que você cria um descritor de socket com `socket()`, o configura para non-blocking, chama `connect()`, e se tudo correr bem `connect()` retornará `-1` imediatamente e `errno` será definido para `EINPROGRESS`. Em seguida você chama `select()` com o timeout que desejar, passando o descritor do socket nos sets de leitura e gravação. Se não expirar, significa que a chamada a `connect()` foi concluída. Neste ponto, você terá que usar `getsockopt()` com a opção `SO_ERROR` para obter o valor de retorno a partir da chamada `connect()`, que deve ser zero se não houver erro.

Finalmente, você provavelmente vai querer definir o socket de volta para blocking antes de iniciar a transferência de dados sobre ele.

Observe que isso tem a vantagem adicional de permitir que seu programa faça outra coisa enquanto está se conectando, também. Você poderia, por exemplo, definir o tempo limite para algo baixo, como 500 ms, e atualizar um indicador na tela a cada timeout, em seguida, chamar `select()` novamente. Quando você tiver chamado `select()` e excedido, digamos, 20 vezes,

você saberá que é hora de desistir da conexão.

Como eu disse, confira o source de Stevens para um exemplo perfeitamente excelente.

Como posso construir para Windows?

Primeiro, apague o Windows e instale um Linux ou um BSD. `};-)`. Não, na verdade, apenas consulte a seção construindo no Windows na introdução.

Como faço para construir para Solaris/SunOS? Eu continuo recebendo erros de linker quando tento compilar!

Os erros de linker acontecem porque ambientes Sun não compilam automaticamente com bibliotecas de socket. Consulte a seção construção para Solaris/SunOS na introdução para um exemplo de como fazer isso.

Por que `select()` segue caindo em um sinal?

Os sinais tendem a fazer com que as chamadas do sistema bloqueadas retornem `-1` com `errno` definido como `EINTR`. Quando você configura um manipulador de sinal com `sigaction()`, você pode definir o sinalizador `SA_RESTART`, que deve reiniciar a chamada de sistema depois que ela for interrompida.

Naturalmente, isso nem sempre funciona.

A minha solução favorita para isso envolve uma estrutura `goto`. Você sabe que isso irrita seus professores infinitamente, então vá em frente!

```
1  select_restart:
2  if ((err = select(fdmax+1, &readfds, NULL, NULL, NULL))
    == -1) {
3      if (errno == EINTR) {
4          // algum sinal acabou de nos interromper, então
    reinicie
5          goto select_restart;
6      }
7      // lide com o erro real aqui:
8      perror("select");
9  }
```

Claro, você não *necessita* usar `goto`, neste caso; Você pode usar outras estruturas para controle. Mas acho que `goto` é realmente mais limpa.

Como posso implementar um timeout em uma chamada a `recv()`?

Use `select()`! Ela permite que você especifique um parâmetro de tempo limite para os descritores de socket que você deseja ler. Ou, você poderia envolver toda a funcionalidade em uma única função, assim:

```c
#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 set do descritor de arquivo
    FD_ZERO(&fds);
    FD_SET(s, &fds);

    // configura a struct timeval para timeout
    tv.tv_sec = timeout;
    tv.tv_usec = 0;

    // aguarde até timeout ou receber os dados
    n = select(s+1, &fds, NULL, NULL, &tv);
    if (n == 0) return -2; // timeout!
    if (n == -1) return -1; // error

    // os dados devem estar aqui, então faça um recv()
 normal
    return recv(s, buf, len, 0);
}
.
.
.
// Chamada de amostra para recvtimeout():
n = recvtimeout(s, buf, sizeof buf, 10); // 10 seg
 timeout

if (n == -1) {
    // ocorreu um erro
    perror("recvtimeout");
}
else if (n == -2) {
    // ocorreu timeout
} else {
    // tenho alguns dados em buf
}
```

```
43  .
44  .
45  .
```

Observe que `recvtimeout()` retorna `-2` no caso de um timeout. Por que não retorna `0`? Bem, se você se lembra, um valor de retorno `0` em uma chamada a `recv()` significa que o lado remoto fechou a conexão. Então esse valor de retorno já é usado, e `-1` significa "erro", então eu escolhi `-2` como o meu indicador de timeout.

Como posso criptografar ou comprimir os dados antes de enviá-los através do socket?

Uma maneira fácil de criptografar é usar SSL (Secure sockets layer), mas isso está além do escopo deste guia. (Confira o projeto OpenSSL [45] para mais informações.)

Mas supondo que você deseja conectar ou implementar seu próprio compressor ou sistema de criptografia, é apenas uma questão de pensar em seus dados como uma sequência de etapas entre as duas extremidades. Cada etapa altera os dados de alguma forma.

1. servidor lê dados do arquivo (ou de qualquer lugar)
2. servidor criptografa/comprime os dados (você adiciona esta parte)
3. servidor envia os dados criptografados com `send()`

Agora o contrário:

1. cliente recebe os dados criptografados, com `recv()`
2. cliente decifra/descomprime os dados (você adiciona esta parte)
3. cliente grava os dados em arquivo (ou em qualquer lugar)

Se você for compactar e criptografar, lembre-se de compactar primeiro. `:-)`

Contando que o cliente desfaça corretamente o que o servidor faz, os dados ficarão bem no final, independentemente de quantos passos intermediários você adicione.

Então, tudo que você precisa fazer para usar meu código é encontrar o local entre onde os dados são lidos e onde são enviados (usando `send()`) através da rede, e colocar lá algum código que faça a criptografia.

O que é esse "`PF_INET`" que continuo vendo? Está relacionado com `AF_INET`?

Sim, sim é isso. Consulte a seção sobre `socket()` para mais detalhes.

Como eu posso escrever um servidor que aceite comandos shell de um cliente e os execute?

Para simplificar, vamos dizer que o cliente execute `connect()`, `send()` e `close()` na conexão (ou seja, não haverá chamadas de sistema subseqüentes sem que o cliente se conecte novamente.)

O processo que o cliente segue é o seguinte:

1. `connect()` conecta ao servidor

2. `send("/sbin/ls > /tmp/client.out")`
3. `close()` termina a conexão

Enquanto isso, o servidor está manipulando os dados e os executando:

1. `accept()` aceita a conexão do cliente
2. `recv(str)` recebe a string de comandos
3. `close()` termina a conexão
4. `system(str)` para executar o comando

Atenção! Ter o servidor executando o que o cliente diz é como dar acesso a um shell remoto e as pessoas podem fazer coisas na sua conta quando se conectam ao servidor. Por exemplo, no exemplo acima, e se o cliente envia "`rm -rf ~`"? Ele excluiria tudo na sua conta, isso é um absurdo!

Então seja prudente, e evite que o cliente use qualquer comando com exceção de um par de utilitários que você sabe que são seguros, como o utilitário `foobar`:

```
if (!strncmp(str, "foobar", 6)) {
    sprintf(sysstr, "%s > /tmp/server.out", str);
    system(sysstr);
}
```

Mas você ainda está inseguro, infelizmente: o que acontece se o cliente entra com "`foobar; rm -rf ~`"? A coisa mais segura a fazer é escrever uma pequena rotina que coloque um caractere de escape ("\") na frente de todos os caracteres não alfanuméricos (incluindo espaços, se for o caso) nos argumentos para o comando.

Como você pode ver, a segurança é um grande problema quando o servidor começa a executar o que o cliente envia.

Estou enviando uma enorme quantidade de dados, mas quando eu executo `recv()` ele recebe apenas 536 bytes ou 1460 bytes de cada vez. Mas se eu executo em minha máquina local, ele recebe todos os dados ao mesmo tempo. O que está acontecendo?

Você está atingindo o MTU—o tamanho máximo que o meio físico pode manipular. Na máquina local, você está usando o dispositivo de loopback que pode lidar até com 8K ou mais sem nenhum problema. Mas em Ethernet, que pode manipular apenas 1500 bytes com um cabeçalho, você atinge esse limite. Através de um modem, com 576 MTU (novamente, com cabeçalho), você atinge o limite ainda mais rapidamente.

Você precisa garantir que todos os dados estão sendo enviados, em primeiro lugar. (Veja a implementação da função `sendall()` para detalhes.) Uma vez que você tenha certeza disso, então você precisa chamar `recv()` em um loop até que todos os seus dados sejam lidos.

Leia a seção Bases do encapsulamento de dados para obter detalhes sobre o recebimento de pacotes completos de dados usando várias chamadas `recv()`.

Eu estou em um ambiente Windows e eu não possuo a chamada de sistema `fork()` ou

qualquer tipo de `struct sigaction`. O que fazer?

Elas podem estar em qualquer lugar, elas estarão em bibliotecas POSIX que podem ter sido fornecidas com o compilador. Como eu não tenho um ambiente Windows, eu realmente não posso dizer-lhe a resposta, mas eu me lembro que a Microsoft tem uma camada de compatibilidade POSIX e é aí onde estaria `fork()`. (E talvez até mesmo `sigaction`.)

Procure no help que veio com o VC++ por "fork" ou "POSIX" e veja se ele fornece alguma pista.

Se isso não funcionar, esqueça `fork()/sigaction` e use em substituiçao o equivalente em Win32: `CreateProcess()`. Eu não sei como usar `CreateProcess()`—é preciso milhões de argumentos, mas isso deve ser coberto na documentação que veio com VC++.

Estou atrás de um firewall—como faço para que as pessoas de fora do firewall saibam o meu endereço IP para que elas possam se conectar à minha máquina?

Infelizmente, o objetivo de um firewall é impedir que pessoas fora do firewall se conectem à máquinas dentro do firewall, portanto, permitir que isso ocorra é basicamente considerado uma violação de segurança.

Isto não quer dizer que tudo está perdido. Por um lado, você ainda pode usar `connect()` através do firewall, se ele estiver fazendo algum tipo de mascaramento ou NAT ou algo parecido. Basta projetar seus programas para que você seja sempre o único a iniciar a conexão, e tudo ocorrerá bem.

Se isso não for satisfatório, você pode pedir a seus administradores para que abram um buraco no firewall para que as pessoas possam se conectar a você. O firewall pode encaminhar pacotes para você através do software NAT, ou através de um proxy ou algo parecido.

Esteja ciente dee que um buraco no firewall não deve ser visto de forma leviana. Você precisa garantir que não concederá às pessoas más acesso à rede interna; se você é um novato, é muito mais difícil fazer software seguro do que você possa imaginar.

Não faça seu sysadmin ter raiva de mim. `;-)`

Como faço para escrever um packet sniffer? Como faço para colocar minha interface Ethernet em modo promíscuo?

Para aqueles que não sabem, quando uma placa de rede está em "modo promíscuo", ela encaminhará TODOS os pacotes para o sistema operacional, e não apenas aqueles que foram endereçados a esta máquina específica. (Estamos falando de endereços da camada Ethernet aqui, não endereços IP--mas como ethernet é de camada inferior a camada IP, todos os endereços IP são efetivamente encaminhados. Consulte a seção Baixo nível nonsense e Teoria de Rede para mais informações.)

Esta é a base para o funcionamento de um packet sniffer. Ela coloca a interface em modo promíscuo e, em seguida, o sistema operacional obtém cada pacote que passa pelo fio. Você terá um socket de algum tipo do qual você poderá ler esses dados.

Infelizmente, a resposta para a pergunta varia de acordo com a plataforma, mas se você busca no Google por, por exemplo, "windows promiscuous ioctl" você provavelmente chegará a algum

lugar. Para Linux, há o que parece ser uma thread útil no Stack Overflow [46], também.

Como posso definir um valor de timeout personalizado para um socket TCP ou UDP?

Depende de seu sistema. Você pode pesquisar na net por `SO_RCVTIMEO` e `SO_SNDTIMEO` (para uso com `setsockopt()`) para ver se o seu sistema suporta essa funcionalidade.

As páginas man Linux sugerem o uso de `alarm()` ou `setitimer()` como um substituto.

Como posso saber quais portas estão disponíveis para uso? Existe uma lista de números "oficiais" de portas?

Normalmente, isso não é um problema. Se você está escrevendo, digamos, um servidor web, então é uma boa ideia usar a bem conhecida porta 80 para o seu programa. Se você estiver escrevendo apenas o seu próprio servidor especializado, escolha uma porta aleatoriamente (mas maior que 1023) e experimente.

Se a porta já estiver em uso, você receberá um erro "Endereço já em uso" ao tentar `bind()`. Escolha outra porta. (É uma boa ideia permitir que o usuário do seu software especifique uma porta alternativa com um arquivo de configuração ou uma opção de linha de comando.)

Há uma lista de números de portas oficiais [47] mantida pelo Internet Assigned Numbers Authority (IANA). Só porque algo (acima de 1023) está nessa lista, não significa que você não possa usar a porta. Por exemplo, o DOOM da Id Software usa a mesma porta que "mdqs", de qualquer forma. Tudo o que importa é que ninguém mais *na mesma máquina* esteja usando essa porta quando você quiser usá-la.

Páginas de Manual

No mundo Unix, há uma série de manuais. Eles têm pequenas seções que descrevem funções individuais que você tem à sua disposição.

Claro, `manual` seria muito texto para digitar. Quero dizer, ninguém no mundo Unix, inclusive eu, gosta de digitar muito. Na verdade, eu poderia continuar e continuar longamente escrevendo sobre o quanto eu prefiro ser conciso, mas em vez disso, serei breve e não o aborrecerei com textos despropositados sobre quão incrivelmente breve eu prefiro ser em quase todas as circunstâncias em sua totalidade.

[Aplausos]

Obrigado. O que estou querendo dizer é que estas páginas são chamadas de "man pages" no mundo Unix, e eu incluí minha própria variante pessoal truncada aqui para o seu prazer de leitura. A coisa é, muitas destas funções são de uso muito mais geral do que estou mostrando, mas eu só apresentarei os usos relevantes para Internet Sockets Programming.

Mas espere! Isso não é tudo o que há de errado com minhas man pages:

- Elas estão incompletas e mostram apenas os conceitos básicos do guia.
- Existem muito mais páginas do que estas no mundo real.
- Elas são diferentes das que estão em seu sistema.
- Os arquivos de cabeçalho podem ser diferente para determinadas funções no seu

sistema.
- Os parâmetros das funções podem ser diferentes para determinadas funções no seu sistema.

Se você quiser a informação real, verifique suas man pages Unix locais digitando `man qualquer`, onde "qualquer" é algo em que você está incrivelmente interessado, como "`accept`". (Tenho certeza que o Microsoft Visual Studio tem algo semelhante em sua seção de ajuda. Mas o "man" é melhor porque é um byte mais conciso do que "help". Unix ganha novamente!)

Então, se elas são tão falhas, porque mesmo incluí-las no Guia? Bem, há algumas razões, mas as melhores são que (a)estas versões são voltadas especificamente para a programação de rede e são mais fáceis de digerir que as reais, e (b)estas versões contêm exemplos!

Oh! E falando dos exemplos, eu não costumo colocar toda a verificação de erros, porque realmente aumenta o comprimento do código. Mas você deve absolutamente fazer a verificação de erros praticamente sempre que fizer qualquer chamada de sistema, a menos que você esteja totalmente, 100%, certo de que não irá falhar, e você provavelmente deveria fazê-lo mesmo assim!

`accept()`

Aceita uma conexão de entrada em um socket de escuta

Sinopse

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

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

Descrição

Uma vez que você tenha passado pela dificuldade de conseguir um socket `SOCK_STREAM` e defini-lo para conexões de entrada com `listen()`, então você chama `accept()` para obter-se, na verdade, um novo descritor de socket para utilizar para comunicação posterior com o cliente recém-conectado.

O velho socket que você está usando para ouvir ainda está lá, e será usado para as chamadas `accept()` mais recentes.

Parâmetro	Descrição
`s`	O descritor de socket de `listen()`.
`addr`	Isso é preenchido com o endereço de quem se conecta a você.
`addrlen`	Isso é preenchido com `sizeof()` da estrutura retornada no parâmetro `addr`. Você pode com segurança ignorá-lo se você assumir que você está

Parâmetro	Descrição
	recebendo uma `struct sockaddr_in` de volta, você sabe o que é, porque esse é o tipo que você passou para `add`.

`accept()` normalmente bloqueará, e você pode usar `select()` para dar uma olhada no descritor de socket de escuta antes do tempo para ver se ele está "pronto para ler". Se sim, então há uma nova conexão esperando para ser aceita com `accept()`, Sim! Alternativamente, você pode definir a flag `O_NONBLOCK` no socket de escuta usando `fcntl()`, e, em seguida, ele nunca será bloqueado, preferindo retornar `-1` com `errno` definido para `EWOULDBLOCK`.

O descritor de socket retornado por `accept()` é funcional, aberto e conectado ao host remoto. Você precisa o fechar com `close()` ao terminar.

Valor de retorno

`accept()` retorna o descritor de socket recém-conectado, ou `-1` em caso de erro, com `errno` definido apropriadamente.

Exemplo

```
1  struct sockaddr_storage their_addr;
2  socklen_t addr_size;
3  struct addrinfo hints, *res;
4  int sockfd, new_fd;
5
6  // primeiro, carregar estruturas de endereço com
    getaddrinfo():
7
8  memset(&hints, 0, sizeof hints);
9  hints.ai_family = AF_UNSPEC;  // use IPv4 ou IPv6, o que
    for
10 hints.ai_socktype = SOCK_STREAM;
11 hints.ai_flags = AI_PASSIVE;     // preencha meu IP para
    mim
12
13 getaddrinfo(NULL, MYPORT, &hints, &res);
14
15 // crie um socket, faça bind com, e listen:
16
17 sockfd = socket(res->ai_family, res->ai_socktype,
    res->ai_protocol);
18 bind(sockfd, res->ai_addr, res->ai_addrlen);
19 listen(sockfd, BACKLOG);
20
```

```
21 // agora aceita uma conexão de entrada:
22
23 addr_size = sizeof their_addr;
24 new_fd = accept(sockfd, (struct sockaddr *)&their_addr,
    &addr_size);
25
26 // pronto para se comunicar no descritor de socket
    new_fd!
```

Veja também

socket(), getaddrinfo(), listen(), struct sockaddr_in

bind()

Associa um socket com um endereço IP e um número de porta

Sinopse

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

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

Descrição

Quando uma máquina remota deseja se conectar ao seu programa servidor, ela precisa de duas informações: o endereço IP e o número da porta. A chamada `bind()` permite que você faça exatamente isso.

Primeiro, você chama `getaddrinfo()` para carregar uma `struct sockaddr` com o endereço de destino e informações de porta. Então você chama `socket()` para obter um descritor de socket, e então você passa o socket e o endereço IP a `bind()`, e o endereço IP e a porta são magicamente (usando magia real) ligados ao socket!

Se você não sabe o seu endereço IP, ou sabe que só possui um endereço IP na máquina, ou não se importa com qual dos endereços IP da máquina se utiliza, você pode simplesmente passar a flag `AI_PASSIVE` no parâmetro `hints` para `getaddrinfo()`. O que isto faz é preencher parte do endereço IP de `struct sockaddr` com um valor especial que diz a `bind()` que deve preencher automaticamente o endereço IP.

O quê? Que valor especial é carregado no endereço IP da `struct sockaddr` para fazer com que ela preencha automaticamente com o endereço do host atual? Eu lhe direi, mas lembre-se que isto é apenas se você estiver preenchendo a `struct sockaddr` manualmente; Se não, use os resultados de `getaddrinfo()`, conforme acima. Em IPv4, o campo

`sin_addr.s_addr` da estrutura `struct sockaddr_in` está definido para `INADDR_ANY`. Em IPv6, o campo `sin6_addr` da estrutura `struct sockaddr_in6` é atribuído a partir da variável global `in6addr_any`. Ou, se você está declarando uma nova `struct in6_addr`, você pode inicializá-la para `IN6ADDR_ANY_INIT`.

Por fim, o parâmetro `addrlen` deve ser definido como `sizeof my_addr`.

Valor de retorno

Retorna zero em caso de sucesso, ou `-1` em caso de erro (e `errno` irá será definido em conformidade.)

Exemplo

```
// maneira moderna de fazer as coisas com getaddrinfo()

struct addrinfo hints, *res;
int sockfd;

// primeiro, carregue as estruturas de endereço com
 getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 ou IPv6, o que
 for
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // preencha meu IP para
 mim

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

// cria um socket:
// (percorra a lista vinculada "res" e verifique por
 erros!)

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

// bind para a porta que passamos para getaddrinfo():

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

```
// exemplo de empacotamento manual de uma estrutura,
 IPv4

```

```
struct sockaddr_in myaddr;
int s;

myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(3490);

// você pode especificar um endereço IP:
inet_pton(AF_INET, "63.161.169.137",
 &(myaddr.sin_addr));

// ou você pode deixar que ele selecione um
 automaticamente:
myaddr.sin_addr.s_addr = INADDR_ANY;

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&myaddr, sizeof myaddr);
```

Veja também

getaddrinfo(), socket(), struct sockaddr_in, struct in_addr

connect()

Conecta um socket a um servidor

Sinopse

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

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

Descrição

Uma vez que você construiu um descritor de socket com a chamada `socket()`, pode-se conectar esse socket a um servidor remoto usando a já bem nomeada chamada de sistema `connect()`. Tudo o que você precisa fazer é passar o descritor de socket e o endereço do servidor que você está interessado em conhecer melhor. (Ah, e o comprimento do endereço, que é normalmente passado para funções como esta.)

Normalmente, esta informação vem como resultado de uma chamada a `getaddrinfo()`, mas você pode preencher sua própria `struct sockaddr` se você quiser.

Se você ainda não chamou `bind()` no descritor de socket, ele é automaticamente ligado ao

seu endereço IP e a uma porta local aleatória. Isso geralmente é bom para você, se você não for um servidor, já que realmente não se importará com a porta local; Você só se importará com a porta remota, então você pode colocá-la no parâmetro `serv_addr`. Você *pode* chamar `bind()` se você realmente quiser que seu socket cliente esteja em um endereço IP e porta específicos, mas isso é muito raro.

Uma vez que o socket é conectado com `connect()`, você está livre para usar `send()` e `recv()` e trafegar dados sobre ele como mandar seu coração.

Nota especial: se você se conecta com `connect()` a um socket `SOCK_DGRAM` UDP em um host remoto, você pode usar `send()` e `recv()` bem como `sendto()` e `recvfrom()`. Se você quiser.

Valor de retorno

Retorna zero em caso de sucesso, ou `-1` em caso de erro (e `errno` irá ser definido em conformidade.)

Exemplo

```
1  // conecta à www.example.com, porta 80 (http)
2
3  struct addrinfo hints, *res;
4  int sockfd;
5
6  // primeiro, carregar estruturas de endereço com
    getaddrinfo():
7
8  memset(&hints, 0, sizeof hints);
9  hints.ai_family = AF_UNSPEC;  // use IPv4 ou IPv6, o que
    for
10 hints.ai_socktype = SOCK_STREAM;
11
12 // poderíamos colocar "80" em vez de "http" na próxima
    linha:
13 getaddrinfo("www.example.com", "http", &hints, &res);
14
15 // cria o socket:
16
17 sockfd = socket(res->ai_family, res->ai_socktype,
    res->ai_protocol);
18
19 // conecte-o ao endereço e à porta que passamos para
    getaddrinfo():
20
21 connect(sockfd, res->ai_addr, res->ai_addrlen);
```

Veja também

socket(), bind()

close()

Fecha um descritor de socket

Sinopse

```
#include <unistd.h>

int close(int s);
```

Descrição

Depois que você terminar de usar o socket para qualquer esquema demente que você tenha inventado e você não quiser mais usar `send()` ou `recv()` ou, na verdade, fazer *qualquer coisa* com o socket, você pode o fechar com `close()`, e ele será liberado, para nunca mais ser usado novamente.

O lado remoto pode saber que isso aconteceu de duas maneiras. Um: se o lado remoto chamadar `recv()`, ele retornará `0`. Dois: se o lado remoto chamar `send()`, receberá um sinal SIGPIPE e send() retornará `-1` e `errno` será definido para EPIPE.

Usuários de Windows: a função que você precisa usar chama-se `closesocket()`, não `close()`. Se você tentar usar `close()` em um descritor de socket, é possível que o Windows fique irritado... E você não gostaria dele irritado.

Valor de retorno

Retorna zero em caso de sucesso, ou `-1` em caso de erro (e `errno` será definido em conformidade.)

Exemplo

```
1 s = socket(PF_INET, SOCK_DGRAM, 0);
2 .
3 .
4 .
5 // um monte de coisas...*BRRRONNNN!*
6 .
7 .
8 .
9 close(s);  // não muito, realmente.
```

Veja também

socket(), shutdown()

getaddrinfo(), freeaddrinfo(), gai_strerror()

Obtém informações sobre um nome de host e/ou serviço e carrega uma `struct sockaddr` com o resultado.

Sinopse

```
#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) ou IPPROTO_TCP,
    IPPROTO_UDP

  socklen_t  ai_addrlen;     // tamanho de ai_addr
  char   *ai_canonname;      // nome canônico para nodename
  struct sockaddr  *ai_addr; // endereço binário
  struct addrinfo  *ai_next; // próxima estrutura na lista
    vinculada
};
```

Descrição

`getaddrinfo()` é uma excelente função que retornará informações sobre um nome de host específico (como o seu endereço IP) e carregar uma `struct sockaddr` para você, cuidando dos detalhes importantes (como se é IPv4 ou IPv6). Ela substitui as antigas funções `gethostbyname()` e `getservbyname()`. A descrição, abaixo, contém uma grande quantidade de informação que pode ser um pouco assustadora, mas o uso é bastante simples. Pode valer a pena conferir os exemplos primeiro.

O nome do host em que você estiver interessado vai no parâmetro `nodename`. O endereço pode ser um nome de host, como "www.example.com", ou um endereço IPv4 ou IPv6 (passado como string). Este parâmetro também pode ser `NULL` se você estiver usando a flag `AI_PASSIVE` (veja abaixo).

O parâmetro `servname` é basicamente o número da porta. Ele pode ser um número de porta (passada como string, como "80"), ou pode ser um nome de serviço, como "http" ou "tftp" ou "smtp" ou "pop", etc. Nomes de serviços bem conhecidos podem ser encontrados no IANA Port List [48] ou em seu arquivo `/etc/services`.

Por fim, para parâmetros de entrada, temos `hints`. Isto é realmente onde você começa a definir o que a função `getaddrinfo()` fará. Zere toda a estrutura antes da utilização, com `memset()`. Vamos dar uma olhada nos campos que você precisa configurar antes do uso.

O `ai_flags` pode ser configurado para uma variedade de coisas, mas aqui estão algumas das mais importantes. (Se podem especificar múltiplas flags por bitwise-ORing juntamente com o operador `|`.) Verifique a sua página man para a lista completa de flags.

`AI_CANONNAME` causa que `ai_canonname` do resultado se complete com o nome canônico (real) do host. `AI_PASSIVE` faz com que o endereço IP do resultado se complete com `INADDR_ANY` (IPv4) ou `in6addr_any` (IPv6); Isso faz com que uma chamada subsequente a `bind()` preencha automaticamente o endereço IP da `struct sockaddr` com o endereço do host atual. Isso é excelente para configurar um servidor quando você não deseja codificar o endereço.

Se você usar a flag `AI_PASSIVE`, então você pode passar `NULL` no `nodename` (já que `bind()` irá preenchê-lo para você mais tarde.)

Continuando com os parâmetros de entrada, você provavelmente vai querer definir `ai_family` para `AF_UNSPEC` que diz a `getaddrinfo()` para operar com ambos os endereços, IPv4 e IPv6. Você também pode restringir a um ou a outro com `AF_INET` ou `AF_INET6`.

Em seguida, o campo `socktype` deve ser definido como `SOCK_STREAM` ou `SOCK_DGRAM`, dependendo de qual tipo de socket se deseja.

Finalmente, apenas deixe `ai_protocol` definido em `0` para escolher automaticamente o seu tipo de protocolo.

Agora, depois meter todas essas coisas lá dentro, você pode *finalmente* fazer a chamada a `getaddrinfo()`!

É claro, este é o lugar onde a diversão começa. A `res` agora apontará para uma lista vinculada de `struct addrinfos`, e você pode percorrer esta lista para obter todos os endereços que correspondam ao que você passou com hints.

Agora, é possível obter alguns endereços que não funcionam por uma razão ou outra, de modo que o que a Linux man page faz é em loops percorrer a lista fazendo uma chamada a `socket()` e `connect()` (Ou `bind()` se você estiver configurando um servidor com a flag `AI_PASSIVE`) até obter êxito.

Finalmente, quando você terminar com a lista vinculada, você precisa chamar `freeaddrinfo()` para liberar memória (ou ela vazará, e algumas pessoas ficarão chateadas.)

Valor de retorno

Retorna zero em caso de sucesso, ou diferente de zero em caso de erro. Se ela retornar diferente de zero, você pode usar a função `gai_strerror()` para obter uma versão de impressão do código de erro no valor de retorno.

Exemplo

```
1  // código para um cliente se conectar a um servidor
2  // ou seja, um socket stream para www.example.com na
    porta 80 (http)
3  // seja IPv4 ou IPv6
4
5  int sockfd;
6  struct addrinfo hints, *servinfo, *p;
7  int rv;
8
9  memset(&hints, 0, sizeof hints);
10 hints.ai_family = AF_UNSPEC; // use AF_INET6 para forçar
    IPv6
11 hints.ai_socktype = SOCK_STREAM;
12
13 if ((rv = getaddrinfo("www.example.com", "http", &hints,
    &servinfo)) != 0) {
14     fprintf(stderr, "getaddrinfo: %s\n",
    gai_strerror(rv));
15     exit(1);
16 }
17
18 // percorre todos os resultados e conecta ao primeiro
    que puder
19 for(p = servinfo; p != NULL; p = p->ai_next) {
20     if ((sockfd = socket(p->ai_family, p->ai_socktype,
21             p->ai_protocol)) == -1) {
22         perror("socket");
23         continue;
24     }
25
26     if (connect(sockfd, p->ai_addr, p->ai_addrlen) ==
    -1) {
27         perror("connect");
28         close(sockfd);
```

```
        continue;
    }

    break; // se chegamos aqui, devemos ter conectado com sucesso
}

if (p == NULL) {
    // repetiu o final da lista sem conexão
    fprintf(stderr, "falha ao connect\n");
    exit(2);
}

freeaddrinfo(servinfo); // tudo feito com essa estrutura
```

```
// código para um servidor aguardar conexões
// ou seja, um socket stream na porta 3490, no IP deste host
// seja IPv4 ou IPv6.

int sockfd;
struct addrinfo hints, *servinfo, *p;
int rv;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // use AF_INET6 para forçar IPv6
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE; // use meu endereço IP

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

// loop por todos os resultados e vincular ao primeiro que pudermos
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; // se chegamos aqui, devemos ter conectado com sucesso
}

if (p == NULL) {
    // repetiu o final da lista sem êxito em realizar bind
    fprintf(stderr, "falha ao realizar bind com o socket\n");
    exit(2);
}

freeaddrinfo(servinfo); // tudo feito com essa estrutura
```

Veja também

gethostbyname(), getnameinfo()

gethostname()

Retorna o nome do sistema

Sinopse

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

int gethostname(char *name, size_t len);
```

Descrição

Seu sistema tem um nome. Todos eles têm. Essa é uma coisa um pouco mais de UNIXy do que o resto das coisas sobre redes das quais temos falado, mas ela ainda tem seus usos.

Por exemplo, você pode obter o seu nome de host, e depois chamar `gethostbyname()` para descobrir o seu endereço IP.

O parâmetro `name` deve apontar para um buffer que conterá o nome de host, e `len` é o

tamanho desse buffer em bytes. `gethostname()` não sobrescreve o final do buffer (pode retornar um erro, ou pode simplesmente parar de escrever), e terminará a string com NUL, se houver espaço para isso no buffer.

Valor de retorno

Retorna zero em caso de sucesso, ou -1 em caso de erro (e `errno` definido em conformidade.)

Exemplo

```
1 char hostname[128];
2
3 gethostname(hostname, sizeof hostname);
4 printf("Meu hostname: %s\n", hostname);
```

Veja também

gethostbyname()

gethostbyname(), gethostbyaddr()

Obtém um endereço IP de um nome de host ou vice-versa.

Sinopse

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

struct hostent *gethostbyname(const char *name); //
    DESCONTINUADA!
struct hostent *gethostbyaddr(const char *addr, int len, int
    type);
```

Descrição

POR FAVOR NOTE: estas duas funções são substituídas por `getaddrinfo()` e `getnameinfo()`! Em particular, `gethostbyname()` não funciona bem com o IPv6.

Essas funções são mapeadas entre os nomes de host e os endereços IP. Por exemplo, se você tem "www.example.com", você pode usar `gethostbyname()` para obter seu endereço IP e armazená-lo em uma `struct in_addr`.

Por outro lado, se você tem uma `struct in_addr` ou uma `struct in6_addr`, você pode usar `gethostbyaddr()` para recuperar o nome de host. `gethostbyaddr()` *é* compatível

com IPv6, mas você deve usar o mais novo e brilhante `getnameinfo()` em seu lugar.

(Se você tem uma string contendo um endereço IP no formato pontos-e-números da qual você deseja procurar o nome do host, seria melhor usar `getaddrinfo()` com a flag `AI_CANONNAME`.)

`gethostbyname()` recebe uma string como "www.yahoo.com", e retorna uma `struct hostent`, que contém toneladas de informações, incluindo o endereço IP. (Outras informações são o nome oficial do host, uma lista de aliases, o tipo de endereço, o comprimento dos endereços e a lista de endereços—é uma estrutura de uso geral que é muito fácil de usar para os nossos propósitos específicos, uma vez que você vê como fazer.)

`gethostbyaddr()` leva uma `struct in_addr` ou uma `struct in6_addr` e traz para você um nome de host correspondente (se houver), por isso é uma espécie de `gethostbyname()` reversa. Quanto aos parâmetros, embora `addr` seja um `char*`, você realmente deseja passar um ponteiro para uma `struct in_addr`. `len` deve ser `sizeof (struct in_addr)` e `type` deve ser `AF_INET`.

Então, o que é esta `struct hostent` que é retornada? Ela possui uma série de campos que contém informações sobre o host em questão.

Campo	Descrição
`char *h_name`	O nome real canônico do host.
`char **h_aliases`	Uma lista de aliases que podem ser acessados com arrays—o último elemento é `NULL`
`int h_addrtype`	O tipo de endereço do resultado, que realmente deve ser `AF_INET` para nossos propósitos.
`int comprimento`	O comprimento dos endereços em bytes, que é 4 para endereços IP (versão 4).
`char **h_addr_list`	Uma lista de endereços IP para este host. Embora esse seja um `char**`, é realmente um array de `struct in_addr*`s disfarçado. O último elemento do array é `NULL`.
`h_addr`	Um alias comumente definido para `h_addr_list[0]`. Se você apenas quiser qualquer endereço IP antigo para esse host (sim, eles podem ter mais de um) basta usar este campo.

Valor de retorno

Retorna um ponteiro para uma `struct hostent` resultante em caso de sucesso, ou `NULL` em caso de erro.

Em vez da `perror()` normal e todas essas coisas que você normalmente usaria para o relatório de erros, essas funções têm resultados paralelos na variável `h_errno`, que podem ser impressos usando-se as funções `herror()` ou `hstrerror()`. Elas funcionam como as clássicas funções `errno`, `perror()`, e `strerror()` com as quais você está acostumado.

Exemplo

```c
// ESTE É UM MÉTODO DESCONTINUADO PARA SE OBTER NOMES DE HOST
// use getaddrinfo() em vez de!

#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,"uso: ghbn hostname\n");
        return 1;
    }

    if ((he = gethostbyname(argv[1])) == NULL) { // obtém info. do host
        herror("gethostbyname");
        return 2;
    }

    // imprimir informações sobre este host:
    printf("O Nome oficial é: %s\n", he->h_name);
    printf("    Endereços IP: ");
    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;
}
```

```c
// ISTO FOI SUPERCEDIDO
// use getnameinfo() em vez de!
```

```
3
4  struct hostent *he;
5  struct in_addr ipv4addr;
6  struct in6_addr ipv6addr;
7
8  inet_pton(AF_INET, "192.0.2.34", &ipv4addr);
9  he = gethostbyaddr(&ipv4addr, sizeof ipv4addr, AF_INET);
10 printf("Nome de host: %s\n", he->h_name);
11
12 inet_pton(AF_INET6, "2001:db8:63b3:1::beef", &ipv6addr);
13 he = gethostbyaddr(&ipv6addr, sizeof ipv6addr,
    AF_INET6);
14 printf("Nome de host: %s\n", he->h_name);
```

Veja também

getaddrinfo(), getnameinfo(), gethostname(), errno, perror(), strerror(), struct in_addr

getnameinfo()

Procura informações do nome do host e do nome do serviço para uma determinada `struct sockaddr`.

Sinopse

```
#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);
```

Descrição

Esta função é o oposto de `getaddrinfo()`, isto é, esta função pega uma já carregada `struct sockaddr` e faz uma pesquisa de nome e nome de serviço nela. Ela substitui as antigas funções `gethostbyaddr()` e `getservbyport()`.

Você deve passar um ponteiro para uma `struct sockaddr` (que na realidade é provavelmente uma `struct sockaddr_in` ou `struct sockaddr_in6` que você tenha convertido) no parâmetro `sa`, e o comprimento dessa `struct` em `salen`.

O nome do host e nome do serviço resultantes serão escritos nas áreas apontadas pelos

parâmetros `host` e `serv`. Obviamente, você precisa especificar os comprimentos máximos desses buffers em `hostlen` e `servlen`.

Finalmente, existem várias flags você pode passar, mas aqui estão algumas boas. `NI_NOFQDN` fará com que `host` contenha apenas o nome do host, e não o nome completo do domínio. `NI_NAMEREQD` fará com que a função falhe se o nome não puder ser encontrado com uma pesquisa de DNS (se você não especificar esta flag e o nome não puder ser encontrado, `getnameinfo()` colocará uma versão de string do endereço IP em `host` no seu lugar.)

Como sempre, verifique as suas man pages locais para informações completas.

Valor de retorno

Retorna zero em caso de sucesso, ou diferente de zero em caso de erro. Se o valor de retorno é diferente de zero, pode ser passado para `gai_strerror()` para se obter uma string legível por humanos. Veja `getaddrinfo` para mais informações.

Exemplo

```
struct sockaddr_in6 sa; // poderia ser IPv4 se você quiser
char host[1024];
char service[20];

// fingir que sa está cheio de boas informações sobre o host e a porta...

getnameinfo(&sa, sizeof sa, host, sizeof host, service, sizeof service, 0);

printf("   host: %s\n", host);    // ex. "www.example.com"
printf("service: %s\n", service); // ex. "http"
```

Veja também

`getaddrinfo()`, `gethostbyaddr()`

`getpeername()`

Retorna informação de endereço sobre o lado remoto da conexão.

Sinopse

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

```
int getpeername(int s, struct sockaddr *addr, socklen_t *len);
```

Descrição

Uma vez que você tenha aceitado com `accept()` uma conexão remota, ou conectado com `connect()` a um servidor, você agora tem o que é conhecido como um *peer* (par). Seu par é simplesmente o computador ao qual você está conectado, identificado por um endereço IP e uma porta. Então...

`getpeername()` simplesmente retorna uma `struct sockaddr_in` preenchida com informações sobre a máquina a qual você está conectado.

Por que é chamado um "name"? Bem, há um monte de diferentes tipos de sockets, não apenas Internet Sockets, como estamos usando neste guia, e então "name" é um bom termo genérico que cobre todos os casos. No nosso caso, porém, "name" do peer (par) é o seu endereço IP e porta.

Embora a função retorne o tamanho do endereço resultante em `len`, é necessário pré-carregar `len` com o tamanho de `addr`.

Valor de retorno

Retorna zero em caso de sucesso, ou `-1` em caso de erro (e `errno` será definido em conformidade.)

Exemplo

```
1  // suponha que s é um socket conectado
2
3  socklen_t len;
4  struct sockaddr_storage addr;
5  char ipstr[INET6_ADDRSTRLEN];
6  int port;
7
8  len = sizeof addr;
9  getpeername(s, (struct sockaddr*)&addr, &len);
10
11 // lidar com IPv4 e IPv6:
12 if (addr.ss_family == AF_INET) {
13     struct sockaddr_in *s = (struct sockaddr_in
    *)&addr;
14     port = ntohs(s->sin_port);
15     inet_ntop(AF_INET, &s->sin_addr, ipstr, sizeof
    ipstr);
16 } else { // AF_INET6
```

```
17      struct sockaddr_in6 *s = (struct sockaddr_in6
    *)&addr;
18      port = ntohs(s->sin6_port);
19      inet_ntop(AF_INET6, &s->sin6_addr, ipstr, sizeof
    ipstr);
20  }
21
22  printf("Endereço IP do peer: %s\n", ipstr);
23  printf("Porta do peer      : %d\n", port);
```

Veja também

gethostname(), gethostbyname(), gethostbyaddr()

errno

Contém o código de erro da última chamada de sistema.

Sinopse

```
#include <errno.h>

int errno;
```

Descrição

Essa é a variável que contém informações de erro para muitas chamadas de sistema. Se você se lembra, coisas como `socket()` e `listen()` retornam `-1` em caso de erro, e elas definem o valor exato de `errno` para que você saiba especificamente que erro ocorreu.

O arquivo de cabeçalho `errno.h` lista um monte de nomes simbólicos constantes para erros, como `EADDRINUSE`, `EPIPE`, `ECONNREFUSED`, etc. Suas man pages locais lhe dirão quais códigos podem ser retornados como um erro, e você pode usá-los em tempo de execução para lidar com diferentes erros de diferentes maneiras.

Ou, mais comumente, você pode chamar `perror()` ou `strerror()` para obter uma versão humanamente legível do erro.

Uma coisa a notar, para você estusiasta do multithreading, é que na maioria dos sistemas `errno` é definido de forma thread-safe. (Ou seja, não é realmente uma variável global, mas se comporta exatamente como uma variável global faria em um ambiente single-threaded).

Valor de retorno

O valor da variável é o do erro mais recente a ter acontecido, que pode ser o código para o "sucesso" se a última ação for bem sucedida.

Exemplo

```
s = socket(PF_INET, SOCK_STREAM, 0);
if (s == -1) {
    perror("socket"); // ou use strerror()
}

tryagain:
if (select(n, &readfds, NULL, NULL) == -1) {
    // ocorreu um erro!!

    // se formos interrompidos, basta reiniciar a
 chamada select():
    if (errno == EINTR) goto tryagain;  // AAAA! goto!!!

    // caso contrário, é um erro mais sério:
    perror("select");
    exit(1);
}
```

Veja também

perror(), strerror()

fcntl()

Controla descritores de socket

Sinopse

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

int fcntl(int s, int cmd, long arg);
```

Descrição

Esta função é normalmente usada para realizar bloqueio de arquivos e outras coisas relacionadas, mas também possui algumas habilidades relacionadas a sockets que você pode ver ou usar de tempos em tempos.

O parâmetro `s` é o descritor de socket no qual você deseja operar, o `cmd` deve ser definido como `F_SETFL` e `arg` pode ser um dos seguintes comandos. (Como eu disse, há mais sobre `fcntl()` do que eu estou deixando aqui, mas eu estou tentando me manter voltado a

sockets.)

cmd	Descrição
O_NONBLOCK	Configura o socket para non-blocking. Veja a seção sobre blocking para mais detalhes.
O_ASYNC	Defina o socket para executar E/S assíncrona. Quando os dados estiverem prontos para serem recebidos com `recv()` no socket, o sinal SIGIO será aumentado. Isso é raro de se ver, e está além do escopo deste guia. E acho que está disponível apenas em determinados sistemas.

Valor de retorno

Retorna zero em caso de sucesso, ou `-1` em caso de erro (e `errno` será definido em conformidade.)

Diferentes usos da chamada de sistema `fcntl()` na verdade possuem valores de retorno diferentes, mas eu não os cobrirei aqui porque eles não são relacionados a sockets. Consulte a sua página man local de `fcntl()` para mais informações.

Exemplo

```
int s = socket(PF_INET, SOCK_STREAM, 0);

fcntl(s, F_SETFL, O_NONBLOCK);  // configura para non-
 blocking
fcntl(s, F_SETFL, O_ASYNC);     // configura E/S
 assíncrona
```

Veja também

Blocking, send()

htons(), htonl(), ntohs(), ntohl()

Converte tipos inteiros multi-byte de host byte order para network byte order.

Sinopse

```
#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);
```

Descrição

Apenas para fazer você realmente infeliz, computadores diferentes usam diferentes ordenações de bytes internamente para seus inteiros multi-byte (ou seja, qualquer número inteiro que seja maior que um `char`.) O resultado disso é que se você envia um `short int` de dois bytes com `send()` a partir de um ambiente Intel para um Mac (antes de ambos se tornarem ambientes Intel, também, quero dizer), o que um computador pensa é que o número é 1, o outro pensará que é o número 256, e vice-versa.

A maneira de contornar este problema é que todos deixam de lado suas diferenças e concordam que a Motorola e a IBM tinham razão, e a Intel fez isso da maneira estranha, e assim todos nós convertemos nossas ordenações de bytes para "big-endian" antes de enviá-las. Como a Intel usa máquinas "little-endian", é bem mais politicamente correto chamar nossa ordenação de bytes preferida de "Network Byte Order". Portanto, essas funções convertem de sua ordem de bytes nativa para ordem de bytes de rede e de volta.

(Isto significa que em Intel essas funções trocam todos os bytes, e em PowerPC elas não fazem nada porque os bytes já estão em Network Byte Order. Mas você deve sempre usá-las em seus códigos de qualquer maneira, uma vez que alguém possa querer construí-los em uma máquina Intel e ainda terá as coisas funcionando adequadamente.)

Note que os tipos envolvidos são de 32 bits (4 bytes, provavelmente `int`) e de 16 bits (dois bytes, muito provavelmente `short`). Máquinas de 64 bits podem ter um `htonll()` `ints` de 64bits, mas eu desconheço. Você só terá que escrever o seu próprio.

De qualquer forma, a maneira como essas funções trabalham é que primeiro você decide se está convertendo *de* host byte order (da sua máquina) ou de network byte order. Se "host", a primeira letra da função que você vai chamar é "h". Caso contrário, é "n" para "network". O meio do nome da função é sempre "to" porque você está convertendo de um "para" o outro, e a penúltima letra mostra o que você está convertendo *to*. A última letra é o tamanho dos dados, "s" para short, ou "l" para long. Assim:

Função	Descrição
`htons()`	host `to` network short
`htonl()`	host `to` network `long`
`ntohs()`	network `to` host short
`ntohl()`	network `to` host `long`

Valor de retorno

Cada função retorna o valor convertido.

Exemplo

```
1 uint32_t some_long = 10;
2 uint16_t some_short = 20;
3
```

```
uint32_t network_byte_order;

// converter e enviar
network_byte_order = htonl(some_long);
send(s, &network_byte_order, sizeof(uint32_t), 0);

some_short == ntohs(htons(some_short)); // essa
 expressão é verdadeira
```

inet_ntoa(), inet_aton(), inet_addr

Converte endereços IP de uma string de pontos-e-números para uma `struct in_addr` e de volta.

Sinopse

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

// TODAS ESTAS FORAM DESCONTINUADAS! Use inet_pton() ou
    inet_ntop()!

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);
```

Descrição

Estas funções são obsoletas porque não lidam com IPv6! Use `inet_ntop()` ou `inet_pton()` em vez disso! Elas estão incluídas aqui porque ainda podem ser encontradas na natureza.

Todas essas funções convertem de uma `struct in_addr` (parte de sua `struct sockaddr_in`, o mais provável) para uma string no formato de pontos-e-números (por exemplo, "192.168.5.10") e vice-versa. Se você tem um endereço IP passado na linha de comando ou algo assim, esta é a maneira mais fácil de obter uma `struct in_addr` para `connect()`, ou qualquer outra coisa. Se você precisar de mais energia, tente algumas das funções DNS, como `gethostbyname()` ou tente um *golpe de estado* em seu país local.

A função `inet_ntoa()` converte um endereço de rede de uma `struct in_addr` para uma string de formato pontos-e-números. O "n" em "ntoa" significa network, e o "a" significa ASCII por razões históricas (então isso é "Network To ASCII"—o sufixo "toa" tem um amigo análogo na biblioteca C chamada `atoi()` que converte uma cadeia de caracteres ASCII em um número inteiro.)

A função `inet_aton()` é o oposto, convertendo de uma string de pontos-e-números em uma `in_addr_t` (que é o tipo do campo `s_addr` na tua `struct in_addr`.)

Finalmente, a função `inet_addr()` é uma função antiga que faz basicamente a mesma coisa que `inet_aton()`. Está teoricamente obsoleta, mas você a verá muito e a polícia não virá buscá-lo se você a usar.

Valor de retorno

`inet_aton()` retorna diferente de zero se o endereço for válido, e retorna zero se o endereço for inválido.

`inet_ntoa()` retorna string de pontos-e-números em um buffer estático que é sobrescrito a cada chamada para a função.

`inet_addr()` retorna o endereço como um `in_addr_t`, ou `-1` se houver um erro. (Que é o mesmo resultado como se você tentasse converter a string "`255.255.255.255`", que é um endereço IP válido. É por isso que `inet_aton()` é melhor.)

Exemplo

```
1  struct sockaddr_in antelope;
2  char *some_addr;
3
4  inet_aton("10.0.0.1", &antelope.sin_addr); // armazena
    IP em antelope
5
6  some_addr = inet_ntoa(antelope.sin_addr); // retorna o
    IP
7  printf("%s\n", some_addr); // prints "10.0.0.1"
8
9  // e esta chamada é a mesma que a chamada inet_aton(),
    acima:
10 antelope.sin_addr.s_addr = inet_addr("10.0.0.1");
```

Veja também

`inet_ntop()`, `inet_pton()`, `gethostbyname()`, `gethostbyaddr()`

`inet_ntop()`, `inet_pton()`

Converte endereços IP para um formato humanamente legível e de volta.

Sinopse

```
#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);
```

Descrição

Essas funções permitem lidar com endereços IP legíveis convertendo-os em suas representações binárias para uso com várias funções e chamadas de sistema. O "n" significa "network" e "p" para "presentation". Ou "text presentation". Mas você pode pensar nisso como "printable". "ntop" é "network to printable". Vê?

Às vezes você não quer olhar para uma pilha de números binários ao olhar para um endereço IP. Você quer isso em uma boa forma para impressão, como `192.0.2.180` ou `2001:db8:8714:3a90::12`. Nesse caso, `inet_ntop()` é para você.

`inet_ntop()` usa a família de endereços no parâmetro `af` (ou `AF_INET` ou `AF_INET6`). O parâmetro `src` deve ser um ponteiro para uma `struct in_addr` ou `struct in6_addr` contendo o endereço que você deseja converter para uma string. Finalmente `dst` e `size` são o ponteiro para a string destino e o comprimento máximo dessa string.

Qual deve ser o comprimento máximo da string `dst`? Qual é o comprimento máximo para endereços IPv4 e IPv6? Felizmente há um par de macros para ajudá-lo. Os comprimentos máximos são: `INET_ADDRSTRLEN` e `INET6_ADDRSTRLEN`.

Outras vezes, você pode ter uma string contendo um endereço IP em formato legível, e você quer embalá-la em uma `struct sockaddr_in` ou `struct sockaddr_in6`. Nesse caso, a função oposta `inet_pton()` é o que você está procurando.

`inet_pton()` também usa uma família de endereços (ou `AF_INET` ou `AF_INET6`) no parâmetro `af`. O parâmetro `src` é um ponteiro para uma string contendo o endereço IP no formato imprimível. Por fim o parâmetro `dst` aponta para onde o resultado deve ser armazenado, que é provavelmente uma `struct in_addr` ou `struct in6_addr`.

Estas funções não fazem pesquisas de DNS—você precisará da função `getaddrinfo()` para isso.

Valor de retorno

`inet_ntop()` retorna o parâmetro `dst` em sucesso, ou `NULL` em caso de falha (e `errno` é definido).

`inet_pton()` retorna 1 em caso de sucesso. Isto retorna `-1` se houver erro (`errno` é definido), ou `0` se a entrada não é um endereço IP válido.

Exemplo

```c
// demonstração IPv4 de inet_ntop() e inet_pton()

struct sockaddr_in sa;
char str[INET_ADDRSTRLEN];

// armazene esse endereço IP em sa:
inet_pton(AF_INET, "192.0.2.33", &(sa.sin_addr));

// agora recupere e imprima
inet_ntop(AF_INET, &(sa.sin_addr), str,
 INET_ADDRSTRLEN);

printf("%s\n", str); // imprime "192.0.2.33"
```

```c
// demonstração IPv6 de inet_ntop() e inet_pton()
// (basicamente o mesmo, exceto por ter um monte de 6s
 ao redor)

struct sockaddr_in6 sa;
char str[INET6_ADDRSTRLEN];

// armazena esse endereço IP em sa:
inet_pton(AF_INET6, "2001:db8:8714:3a90::12",
 &(sa.sin6_addr));

// agora recupere e imprima
inet_ntop(AF_INET6, &(sa.sin6_addr), str,
 INET6_ADDRSTRLEN);

printf("%s\n", str); // imprima "2001:db8:8714:3a90::12"
```

```c
// Função auxiliar que você pode usar:

// Converte um endereço de struct sockaddr em uma
 string, IPv4 e IPv6:

char *get_ip_str(const struct sockaddr *sa, char *s,
 size_t maxlen)
{
    switch(sa->sa_family) {
        case AF_INET:
            inet_ntop(AF_INET, &(((struct sockaddr_in
 *)sa)->sin_addr),
                      s, maxlen);
```

```
            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;
}
```

Veja também

getaddrinfo()

listen()

Diga a um socket para ouvir as conexões recebidas.

Sinopse

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

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

Descrição

Você pode obter o seu descritor de socket (feito com a chamada de sistema `socket()`) e dizer-lhe para ouvir conexões de entrada. Isto é o que diferencia os servidores dos clientes, pessoal.

O parâmetro `backlog` pode significar algumas coisas diferentes dependendo do sistema em que você está, mas vagamente é quantas conexões pendentes você pode ter antes que o kernel comece a rejeitar as novas. Assim que as novas conexões entram, você deve ser rápido para aceitá-las com `accept()` para que backlog não seja alcançado. Experimente o definir com 10 ou mais, e seus clientes começarão a receber "Connection refused" sob carga pesada, configure-o maior.

Antes de chamar `listen()`, o servidor deve chamar `bind()` para se conectar a um número de porta específico. Esse número de porta (no endereço IP do servidor) será aquele ao qual os

clientes se conectam.

Valor de retorno

Retorna zero em caso de sucesso, ou -1 em caso de erro (e `errno` será definido em conformidade.)

Exemplo

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

// primeiro, carregue as estruturas de endereço com
 getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 ou IPv6, o que
 for
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // preencha meu IP para
 mim

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

// crie o socket:

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

// bind na porta em que passamos a getaddrinfo():

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

listen(sockfd, 10); // configura um socket servidor (que
 escuta)

// então tem um loop accept() aqui em algum lugar
```

Veja também

accept(), bind(), socket()

perror(), strerror()

Imprimir um erro como uma string legível por humanos.

Sinopse

```
#include <stdio.h>
#include <string.h>   // para strerror()

void perror(const char *s);
char *strerror(int errnum);
```

Descrição

Uma vez que tantas funções retornam -1 em caso de erro e definem o valor da variável `errno` como algum número, certamente seria bom se você pudesse facilmente imprimir isso em um formato que fizesse sentido para você.

Felizmente, `perror()` faz isso. Se você quiser que mais descrições sejam impressas antes do erro, você pode apontar o parâmetro `s` para ela (ou você pode deixar `s` como `NULL` e nada adicional será impresso.)

Em poucas palavras, esta função toma valores de `errno`, como `ECONNRESET`, e os imprime bem, como "Connection reset by peer."

A função `strerror()` é muito semelhante a `perror()`, exceto que retorna um ponteiro para a string da mensagem de erro para um determinado valor (você geralmente passa na variável `errno`.)

Valor de retorno

`strerror()` retorna um ponteiro para a string da mensagem de erro.

Exemplo

```
int s;

s = socket(PF_INET, SOCK_STREAM, 0);

if (s == -1) { // ocorreu algum erro
    // imprime "socket error: " + a mensagem de erro:
    perror("socket error");
}

// similarmente:
if (listen(s, 10) == -1) {
    // isso imprime "an error: " + a mensagem de erro de
 errno:
    printf("an error: %s\n", strerror(errno));
```

```
14 }
```

Veja também

errno

poll()

Teste para eventos em múltiplos sockets simultaneamente.

Sinopse

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

int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

Descrição

Esta função é muito semelhante a `select()` porque ambas monitoram eventos em conjuntos de descritores de arquivos, tais como dados de entrada prontos para `recv()`, sockets prontos para enviar dados com `send()`, dados out-of-band prontos para `recv()`, erros, etc.

A idéia básica é que você passe um array de `nfds` `struct pollfds` em `ufds`, juntamente com um tempo limite em milissegundos (1000 milissegundos por segundo.) O `timeout` pode ser negativo se você quiser esperar para sempre. Se nenhum evento acontece em qualquer dos descritores de socket até ao fim de timeout, `poll()` retornará.

Cada elemento no array de `struct pollfd` representa um descritor de socket e contém os seguintes campos:

```
struct pollfd {
    int fd;         // o descritor de socket
    short events;   // bitmap de eventos nos quais estamos
    interessados
    short revents;  // poll() retorna, bitmap de eventos que
    ocorreram
};
```

Antes de chamar `poll()`, carregue `fd` com o descritor de socket (se você definir `fd` para um número negativo, esta `struct pollfd` será ignorada e seu campo `revents` será definido para zero) e, em seguida, constroi-se o campo `events` por bitwise-ORing nas seguintes macros:

Macro	Descrição

Macro	Descrição
POLLIN	Avise-me quando os dados estiverem prontos para `recv()` neste socket.
POLLOUT	Avise-me quando eu puder enviar dados com `send()` para este socket sem blocking.
POLLPRI	Avise-me quando os dados out-of-band estiverem prontos para `recv()` neste socket.

Uma vez que a chamada `poll()` retorna, o campo `revents` será construído como bitwise-OR nos campos acima, dizendo a você em quais descritores realmente o evento ocorreu. Além disso, esses outros campos podem estar presentes:

Macro	Descrição
POLLERR	Ocorreu um erro neste socket.
POLLHUP	O lado remoto da conexão foi desligado.
POLLNVAL	Algo estava errado com o descritor de socket `fd`—Talvez não esteja inicializado?

Valor de retorno

Retorna o número de elementos em que ocorreram eventos no array `ufds`; Isso pode ser zero se timeout foi alcançado. Além disso retorna `-1` em caso de erro (e `errno` será definido em conformidade.)

Exemplo

```
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);

// finja que estamos conectados a um servidor neste momento
//connect(s1, ...)...
//connect(s2, ...)...

// configure o array de descritores de arquivo.
//
// neste exemplo, queremos saber quando há normais ou out-of-band
// dados prontos para recv()...
```

```
17
18  ufds[0].fd = s1;
19  ufds[0].events = POLLIN | POLLPRI; // checa por normal
     ou out-of-band
20
21  ufds[1].fd = s2;
22  ufds[1].events = POLLIN; // verifique apenas os dados
     normais
23
24  // espera por eventos nos sockets, timeout de 3,5
     segundos
25  rv = poll(ufds, 2, 3500);
26
27  if (rv == -1) {
28      perror("poll"); // ocorreu erro em poll()
29  } else if (rv == 0) {
30      printf("Timeout ocorreu!  Nenhum dado após 3.5
     segundos.\n");
31  } else {
32      // checar por eventos em s1:
33      if (ufds[0].revents & POLLIN) {
34          recv(s1, buf1, sizeof buf1, 0); // recebe dados
     normais
35      }
36      if (ufds[0].revents & POLLPRI) {
37          recv(s1, buf1, sizeof buf1, MSG_OOB); // dados
     out-of-band
38      }
39
40      // checar por eventos s2:
41      if (ufds[1].revents & POLLIN) {
42          recv(s1, buf2, sizeof buf2, 0);
43      }
44  }
```

Veja também

select()

recv(), recvfrom()

Recebe dados em um socket.

Sinopse

```
#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);
```

Descrição

Uma vez que você tenha um socket ativo e conectado, poderá ler dados de entrada vindos do lado remoto usando a `recv()` (para sockets TCP `SOCK_STREAM`) e `recvfrom()` (para sockets UDP `SOCK_DGRAM`).

Ambas as funções usam o descritor de socket `s`, um ponteiro para o buffer `buf`, o tamanho (em bytes) do buffer `len`, e um conjunto de `flags` que controlam como as funções funcionam.

Além disso, a `recvfrom()` recebe uma `struct sockaddr*`, `from` que lhe dirá de onde os dados vieram, e preencherá `fromlen` com o tamanho da `struct sockaddr`. (Você também deve inicializar `fromlen` para ser do tamanho de `from` ou `struct sockaddr`.)

Então, que flags maravilhosas você pode passar para essa função? Aqui estão algumas delas, mas você deve verificar suas páginas man locais para obter mais informação sobre o que é realmente suportado em seu sistema. Você pode realizar operações bitwise-or com elas, ou apenas definir `flags` para `0` se você quer que ela seja uma `recv()` regular.

Macro	Description
`MSG_OOB`	Recebe dados Out of Band. Esta é a forma como obter dados que foram enviados para você com a flag `MSG_OOB` em `send()`. Como receptor, você terá o sinal `SIGURG` levantado informando que há dados urgentes. No seu manipulador para esse sinal, você poderia chamar `recv()` com essa flag `MSG_OOB`.
`MSG_PEEK`	Se você quiser chamar `recv()` "apenas para fingir", você pode chamá-la com esta flag. Isso lhe dirá o que estará lhe esperando no buffer quando você chamar `recv()` "realmente" (ou seja *sem* a flag `MSG_PEEK`. É como uma pré-visualização para a próxima chamada `recv()`.
`MSG_WAITALL`	Diga a `recv()` para não retornar até que todos os dados especificados no parâmetro `len` sejam recebidos. Ele irá ignorar seus desejos em circunstâncias extremas, no entanto, como se um sinal interrompesse a chamada ou se algum erro ocorresse ou se o lado remoto fechasse a conexão, etc. Não fique bravo com isso.

Quando você chama `recv()`, ele irá bloquear até que haja alguns dados para ler. Se você não quiser blocking, defina o socket para non-blocking ou verifique com `select()` ou `poll()` para ver se há dados de entrada antes de chamar `recv()` ou `recvfrom()`.

Valor de retorno

Retorna o número de bytes realmente recebidos (que pode ser menos do que você solicitou no parâmetro `len`), ou `-1` em caso de erro (e `errno` definido em conformidade.)

Se o lado remoto fechar a conexão, `recv()` retornará `0`. Este é o método normal para a determinar se o lado remoto fechou a conexão. A normalidade é boa, rebelde!

Exemplo

```
// stream sockets e recv()

struct addrinfo hints, *res;
int sockfd;
char buf[512];
int byte_count;

// obter informações do host, criar o socket e conectá-
 lo
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 ou IPv6, o que
 for
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);

// Tudo certo! agora conectados, podemos receber alguns
 dados!
byte_count = recv(sockfd, buf, sizeof buf, 0);
printf("recv()'d %d bytes de dados em buf\n",
 byte_count);
```

```
// datagram sockets e recvfrom()

struct addrinfo hints, *res;
int sockfd;
int byte_count;
socklen_t fromlen;
struct sockaddr_storage addr;
char buf[512];
char ipstr[INET6_ADDRSTRLEN];

```

```
// obter informações do host, criar socket, bind para a
 porta 4950
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 ou IPv6, o que
 for
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);

// não precisa de accept(), apenas recvfrom():

fromlen = sizeof addr;
byte_count = recvfrom(sockfd, buf, sizeof buf, 0, &addr,
 &fromlen);

printf("recv()'d %d bytes de dados em buf\n",
 byte_count);
printf("do endereço IP %s\n",
    inet_ntop(addr.ss_family,
        addr.ss_family == AF_INET?
            ((struct sockadd_in *)&addr)->sin_addr:
            ((struct sockadd_in6 *)&addr)->sin6_addr,
        ipstr, sizeof ipstr);
```

Veja também

send(), sendto(), select(), poll(), Blocking

select()

Verifica se os descritores de sockets estão prontos para leitura/escrita.

Sinopse

```
#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);
```

Descrição

A função `select()` fornece a você uma maneira de verificar simultaneamente vários sockets para ver se eles têm dados esperando para serem lidos com `recv()`, ou se você pode enviar com `send()` dados a eles sem blocking, ou se alguma exceção ocorreu.

Você preenche seus sets de descritores de sockets usando as macros, como `FD_SET()`, acima. Uma vez que você tenha o set, você passa-o para a função como um dos seguintes parâmetros: `readfds` se você quiser saber quando qualquer um dos sockets no set está pronto para receber dados com `recv()`, `writefds` se qualquer um dos sockets estiver pronto para enviar dados com `send()`, e/ou `exceptfds` se você precisa saber quando uma exceção (erro) ocorre em qualquer um dos sockets. Qualquer um ou todos estes parâmetros podem ser `NULL` se você não estiver interessado nesses tipos de eventos. Após o retorno de `select()`, os valores nos sets serão alterados para mostrar quais estão prontos para leitura ou escrita, e quais possuem exceções.

O primeiro parâmetro, n é o descritor de socket com a numeração mais alta (eles são apenas `int`s, lembra?) mais um.

Por fim, a `struct timeval`, `timeout`, no final—isso permite que você informe a `select()` por quanto tempo verificar esses sets. Ela retornará após o timeout ou quando um evento ocorrer, o que ocorrer primeiro. A estrutura `struct timeval` possui dois campos: `tv_sec` é o número de segundos, ao qual é adicionado `tv_usec`, o número de microssegundos (1.000.000 microssegundos em um segundo.)

As macros auxiliares fazem o seguinte:

Macro	Description
`FD_SET(int fd, fd_set *set);`	Adiciona `fd` para o `set`.
`FD_CLR(int fd, fd_set *set);`	Remove `fd` do `set`.
`FD_ISSET(int fd, fd_set *set);`	Retorna true se `fd` está no `set`.
`FD_ZERO(fd_set *set);`	Limpar todas as entradas do `set`.

Nota para usuários Linux: A `select()` do Linux pode retornar "pronto-para-ler" e, em seguida, na verdade não estar pronta para ler, fazendo com que a chamada subsequente a `read()` gere blocking. Você pode contornar este problema definindo a flag `O_NONBLOCK` no socket receptor para que ele gere um erro com `EWOULDBLOCK`, em seguida, ignore-o se este erro ocorrer. Veja a man page `fcntl()` para mais informações sobre como configurar um socket para non-blocking.

Valor de retorno

Retorna o número de descritores no set em caso de sucesso, `0` se o tempo limite foi atingido, ou `-1` em caso de erro (e `errno` será definido em conformidade.) Além disso, os sets são modificados para mostrar quais sockets estão prontos.

Exemplo

```
int s1, s2, n;
fd_set readfds;
struct timeval tv;
char buf1[256], buf2[256];

// finja que estamos conectados a um servidor neste momento
//s1 = socket(...);
//s2 = socket(...);
//connect(s1, ...)...
//connect(s2, ...)...

// limpar o set antes do tempo
FD_ZERO(&readfds);

// adicione nossos descritores ao set
FD_SET(s1, &readfds);
FD_SET(s2, &readfds);

// desde que temos s2 segundos, é o "maior", então usamos isso para
// o parâmetro n em select()
n = s2 + 1;

// espere até que um socket tenha dados para recv() (timeout 10.5 secs)
tv.tv_sec = 10;
tv.tv_usec = 500000;
rv = select(n, &readfds, NULL, NULL, &tv);

if (rv == -1) {
    perror("select"); // ocorreu um erro em select()
} else if (rv == 0) {
    printf("Timeout ocorreu! Sem dados após 10.5 segundos.\n");
} else {
    // um ou ambos os descritores têm dados
    if (FD_ISSET(s1, &readfds)) {
        recv(s1, buf1, sizeof buf1, 0);
```

```
36      }
37      if (FD_ISSET(s2, &readfds)) {
38          recv(s2, buf2, sizeof buf2, 0);
39      }
40  }
```

Veja também

poll()

setsockopt(), getsockopt()

Ajusta várias opções para um socket.

Sinopse

```
#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);
```

Descrição

Sockets são completamente configuráveis. Na verdade, eles são tão configuráveis que nem mesmo cobrirei tudo aqui. Provavelmente é dependente do sistema, de qualquer maneira. Mas vou falar sobre o básico.

Obviamente, essas funções obtêm e definem determinadas opções em um socket. Em um ambiente Linux, todas as informações sobre sockets estão na página man de socket na seção 7. (Digite: "`man 7 socket`" para obter todas estas guloseimas.)

Quanto aos parâmetros, s é o socket em questão, level deve ser definido como SOL_SOCKET. Em seguida, defina `optname` para o nome que você está interessado. Novamente, veja sua página man para todas as opções, mas aqui estão algumas das mais divertidas:

optname	Descrição
SO_BINDTODEVICE	Vincula este socket a um nome de dispositivo simbólico como `eth0` em vez de usar `bind()` para vinculá-lo a um endereço IP. Digite o comando `ifconfig` em Unix para ver o nomes dos dispositivos.

optname	Descrição
SO_REUSEADDR	Permite que outros sockets façam `bind()` contra esta porta, a menos que haja um socket de escuta ativo já ligado à porta. Isso permite que você contorne as mensagens de erro "Endereço já em uso" ao tentar reiniciar o servidor após uma falha.
SOCK_DGRAM	Permite que sockets UDP datagram (SOCK_DGRAM) enviem e recebam pacotes para e do endereço de broadcast. Não faz nada—*NADA!!*—para sockets TCP stream! Hahaha!

Quanto ao parâmetro `optval`, geralmente é um ponteiro para um `int` indicando o valor em questão. Para booleanos, zero é falso, e diferente de zero é verdade. E isso é um fato absoluto, a menos que seja diferente em seu sistema. Se não houver um parâmetro a ser passado, `optval` pode ser `NULL`.

O parâmetro final, `optlen`, deve ser definido para o comprimento de `optval`, provavelmente `sizeof(int)`, mas varia dependendo da opção. Observe que, no caso de `getsockopt()`, esse é um ponteiro para um `socklen_t`, e especifica o objeto de tamanho máximo que será armazenado em `optval` (para evitar buffer overflows). E `getsockopt()` modificará o valor de `optlen` para refletir o número de bytes realmente definidos.

Aviso: em alguns sistemas (nomeadamente Sun e Windows), a option pode ser um `char` em vez de um `int`, e é definida como, por exemplo, um valor de caractere `'1'` em vez de um valor `int` 1. Mais uma vez, verifique as suas próprias man pages para obter mais informações com "`man setsockopt`" e "`man 7 socket`"!

Valor de retorno

Retorna zero em caso de sucesso, ou `-1` em caso de erro (e `errno` será definido em conformidade.)

Exemplo

```
int optval;
int optlen;
char *optval2;

// define SO_REUSEADDR em um socket para true (1):
optval = 1;
setsockopt(s1, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof
 optval);

// liga socket a nome de dispositivo (pode não funcionar
 em todos S.O):
optval2 = "eth1"; // 4 bytes long, então 4, abaixo:
setsockopt(s2, SOL_SOCKET, SO_BINDTODEVICE, optval2, 4);
```

```
12
13  // veja se a flag SO_BROADCAST está definida:
14  getsockopt(s3, SOL_SOCKET, SO_BROADCAST, &optval,
     &optlen);
15  if (optval != 0) {
16      print("SO_BROADCAST ativado em s3!\n");
17  }
```

Veja também

fcntl()

send(), sendto()

Envia dados através de um socket.

Sinopse

```
#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);
```

Descrição

Estas funções enviam dados para um socket. De um modo geral, `send()` é usada para sockets TCP `SOCK_STREAM` conectados, e `sendto()` é utilizada para sockets datagram UDP `SOCK_DGRAM` desconectados. Com os sockets não conectados, você deve especificar o destino de um pacote toda vez que você enviar um, e é por isso que os últimos parâmetros de `sendto()` definem para onde o pacote irá.

Com `send()` e `sendto()`, o parâmetro `s` é o socket, `buf` é um ponteiro para os dados que você deseja enviar, `len` é o número de bytes que você deseja enviar e `flags` permite que você especifique mais informações sobre como os dados devem ser enviados. Defina `flags` para zero se você quiser que sejam dados "normais". Aqui estão algumas das flags comumente usadas, mas verifique as suas man pages locais de `send()` para mais detalhes:

Macro	Descrição
`MSG_OOB`	Enviar dados como "out of band". TCP suporta isso, e é uma maneira de informar ao sistema receptor que esses dados têm uma maior prioridade que os dados normais. O receptor receberá o sinal `SIGURG`

Macro	Descrição
	e poderá receber esses dados sem antes receber todos os demais dados normais da fila.
MSG_DONTROUTE	Não envie esses dados através de um roteador, apenas mantenha-os localmente.
MSG_DONTWAIT	Se send() bloquear porque o tráfego de saída está obstruído, faça-o retornar EAGAIN. Isto é como um "ativar non-blocking somente para este send." Veja a seção sobre blocking para obter mais detalhes.
MSG_NOSIGNAL	Se você enviar com send() a um host remoto que já não esta executando recv(), você normalmente obterá o sinal SIGPIPE. A adição desta flag impede que o sinal se levante.

Valor de retorno

Retorna o número de bytes realmente enviados ou -1 em caso de erro (e errno será definido em conformidade.). Observe que o número de bytes realmente enviados pode ser menor que o número que você pediu para enviar! Veja a seção sobre manipulando send()s parcialmente para uma função auxiliar para contornar este problema.

Além disso, se o socket for fechado por qualquer um dos lados, o processo que chamar send() obterá o sinal SIGPIPE. (A menos que send() tenha sido chamado com a flag MSG_NOSIGNAL.)

Exemplo

```
int spatula_count = 3490;
char *secret_message = "The Cheese is in The Toaster";

int stream_socket, dgram_socket;
struct sockaddr_in dest;
int temp;

// primeiro com sockets TCP stream:

// assumir que sockets estão criados e conectados
//stream_socket = socket(...
//connect(stream_socket, ...

// converter para network byte order
temp = htonl(spatula_count);
// envia dados com send normalmente:
send(stream_socket, &temp, sizeof temp, 0);

// envia mensagem secreta out of band:
```

```
20  send(stream_socket, secret_message,
     strlen(secret_message)+1, MSG_OOB);
21
22  // agora com sockets UDP datagram:
23  //getaddrinfo(...
24  //dest = ... // assuma que "dest" contém o endereço do
     destino
25  //dgram_socket = socket(...
26
27  // envia mensagem secreta normalmente:
28  sendto(dgram_socket, secret_message,
     strlen(secret_message)+1, 0,
29          (struct sockaddr*)&dest, sizeof dest);
```

Veja também

recv(), recvfrom()

shutdown()

Para de enviar e ou receber em um socket.

Sinopse

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

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

Descrição

É isso aí! Nós temos isso! Se não quero mais permitir `send()`s no socket, mas eu ainda quero receber dados com `recv()` por ele! Ou vice-versa! Como posso fazer isso?

Quando você fecha um descritor de socket com `close()`, ele fecha ambos os lados do socket para leitura e escrita, e libera o descritor de socket. Se você quiser apenas para fechar um lado ou outro, você pode usar esta chamada `shutdown()`.

Quanto aos parâmetros, `s` é, obviamente, o socket no qual você quer executar esta ação, e a ação escolhida e que possa ser especificada estará no parâmetro `how`. How pode ser `SHUT_RD` para evitar mais `recv()`s, `SHUT_WR` para proibir mais `send()`s, ou `SHUT_RDWR` para ambos.

Note que `shutdown()` não libera o descritor de socket, então você ainda tem que fechar o socket com `close()`, mesmo que tenha sido completamente desligado.

Esta é uma chamada de sistema raramente usada.

Valor de retorno

Retorna zero em caso de sucesso, ou -1 em caso de erro (e `errno` será definido em conformidade.)

Exemplo

```
int s = socket(PF_INET, SOCK_STREAM, 0);

// ...faça alguns send() e outras coisas aqui...

// e agora que terminamos, não permita mais send()s:
shutdown(s, SHUT_WR);
```

Veja também

`close()`

socket()

Aloca um descritor de socket.

Sinopse

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

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

Descrição

Retorna um novo descritor de socket com o qual você pode fazer coisas de socket. Esta é geralmente a primeira chamada no processo enorme de escrever um programa com sockets, e você pode usar o resultado nas chamadas subsequentes como `listen()`, `bind()`, `accept()`, ou uma variedade de outras funções.

No uso normal, você obtém os valores para esses parâmetros de uma chamada `getaddrinfo()`, como é mostrado no exemplo abaixo. Mas você pode preenchê-los à mão se você realmente quiser.

Macro	Descrição
`domain`	`domain` descreve em que tipo de socket você está interessado. Isso pode, acredite, ser uma grande variedade de coisas, mas uma vez que este é um guia sobre sockets, ele será `PF_INET` para IPv4 e `PF_INET6` para o IPv6.

Macro	Descrição
`type`	Além disso, o parâmetro `type` pode ser uma série de coisas, mas você provavelmente o configurará como `SOCK_STREAM` para sockets TPC (`send()`, `recv()`) ou `SOCK_DGRAM` para rápidos e não confiáveis sockets UDP (`sendto()`, `recvfrom()`). (Outro tipo de socket interessante é `SOCK_RAW`, que pode ser usado para construir pacotes manualmente. É bem legal.)
`protocol`	Finalmente, o parâmetro `protocol` informa qual protocolo usar com um determinado tipo de socket. Como eu já disse, por exemplo, `SOCK_STREAM` usa TCP. Felizmente para você, ao usar `SOCK_STREAM` ou `SOCK_DGRAM`, você pode apenas definir o protocolo como 0, e ele usará o protocolo apropriado automaticamente. Caso contrário, você pode usar `getprotobyname()` para procurar o número de protocolo correto.

Valor de retorno

O novo descritor de socket para ser usado em chamadas subsequentes, ou `-1` em caso de erro (e `errno` será definido em conformidade.)

Exemplo

```
1  struct addrinfo hints, *res;
2  int sockfd;
3
4  // Primeiro, carregue as estruturas de endereço com
    getaddrinfo():
5
6  memset(&hints, 0, sizeof hints);
7  hints.ai_family = AF_UNSPEC;     // AF_INET, AF_INET6,
    ou AF_UNSPEC
8  hints.ai_socktype = SOCK_STREAM; // SOCK_STREAM ou
    SOCK_DGRAM
9
10 getaddrinfo("www.example.com", "3490", &hints, &res);
11
12 // crie um socket usando as informações obtidas com
    getaddrinfo():
13 sockfd = socket(res->ai_family, res->ai_socktype,
    res->ai_protocol);
```

Veja também

accept(), bind(), getaddrinfo(), listen()

`struct sockaddr` e companhia

Estruturas para a manipulação de endereços de internet.

Sinopse

```
#include <netinet/in.h>

// Todos os ponteiros para estruturas de endereços de socket são
// convertidos em ponteiros para este tipo antes do uso:

struct sockaddr {
    unsigned short    sa_family;    // família de endereço,
    AF_xxx
    char              sa_data[14];  // 14 bytes do endereço do
    protocolo
};


// sockets IPv4 AF_INET:

struct sockaddr_in {
    short            sin_family;   // ex. AF_INET, AF_INET6
    unsigned short   sin_port;     // ex. htons(3490)
    struct in_addr   sin_addr;     // veja struct in_addr,
    abaixo
    char             sin_zero[8];  // zere isso se você quiser
};

struct in_addr {
    unsigned long s_addr;          // carregar com inet_pton()
};


// sockets IPv6 AF_INET6:

struct sockaddr_in6 {
    u_int16_t       sin6_family;   // família de endereço,
    AF_INET6
    u_int16_t       sin6_port;     // número da porta, Network
    Byte Order
    u_int32_t       sin6_flowinfo; // informação de fluxo IPv6
    struct in6_addr sin6_addr;     // endereço IPv6
    u_int32_t       sin6_scope_id; // ID do escopo
```

```
};

struct in6_addr {
    unsigned char   s6_addr[16];   // carregar com inet_pton()
};


// Estrutura geral para guarda de endereços de sockets, grande
// o suficiente para struct sockaddr_in ou struct sockaddr_in6:

struct sockaddr_storage {
    sa_family_t  ss_family;     // família de endereço

    // tudo isso é preenchimento, implementação específica,
    ignore:
    char      __ss_pad1[_SS_PAD1SIZE];
    int64_t   __ss_align;
    char      __ss_pad2[_SS_PAD2SIZE];
};
```

Descrição

Estas são as estruturas básicas para todas as syscalls e funções que lidam com endereços da internet. Muitas vezes você usará `getaddrinfo()` para preencher estas estruturas, e, em seguida, você as lerá quando for necessário.

Na memória, a `struct sockaddr_in` e a `struct sockaddr_in6` partilham a mesma estrutura inicial que `struct sockaddr`, e você pode livremente converter o ponteiro de um tipo para outro, sem qualquer dano, exceto o possível fim do universo.

Estou apenas brincando na parte do fim-do-universo... se o universo terminar, quando você tentar um cast de `struct sockaddr_in*` para uma `struct sockaddr*`, eu prometo a você que é pura coincidência e você não deveria nem se preocupar com isso.

Então, com isso em mente, lembre-se que sempre que uma função disser que é necessária uma `struct sockaddr*` você pode realizar um cast com a sua `struct sockaddr_in*`, `struct sockaddr_in6*` ou `struct sockadd_storage*` para aquele tipo com facilidade e segurança.

`struct sockaddr_in` é a estrutura utilizada com endereços IPv4 (por exemplo, "192.0.2.10"). Ela mantém uma família de endereços (`AF_INET`), uma porta em `sin_port`, e um endereço IPv4 em `sin_addr`.

Há também este campo `sin_zero` em `struct sockaddr_in`, que algumas pessoas afirmam que deve ser definido para zero. Outras pessoas não afirmam nada sobre isso (a documentação Linux nem sequer menciona isso), e defini-lo como zero não parece ser realmente necessário. Então, se você quiser, defina-o para zero usando `memset()`.

Agora, essa `struct in_addr` é uma besta estranha em diferentes sistemas. Às vezes é uma `union` louca com todos os tipos de `#defines` e outras bobagens. Mas o que você deve fazer é usar apenas o campo `s_addr` nessa estrutura, porque muitos sistemas só implementam esse.

`struct sockadd_in6` e `struct in6_addr` são muito semelhantes, exceto que elas são usadas para IPv6.

`struct sockaddr_storage` é uma struct que você pode passar para `accept()` ou `recvfrom()` quando estiver tentando escrever um código independente de versão IP e não souber se o novo endereço será IPv4 ou IPv6. A estrutura `struct sockaddr_storage` é grande o suficiente para conter os dois tipos, ao contrário da pequena `struct sockaddr` original.

Exemplo

```
// 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);
```

Veja também

accept(), bind(), connect(), inet_aton(), inet_ntoa()

Mais Referências

Você chegou até aqui, e agora está gritando por mais! Onde mais você pode ir para aprender mais sobre tudo isso?

Livros

Para um livro de papel de celulose, old-school, segure-isso-em-suas-mãos, experimente alguns dos seguintes excelentes livros. Eles redirecionam para links afiliados com um livreiro popular, me oferecendo ótimos subornos. Se você está apenas se sentindo generoso, pode pagar uma doação para `beej@beej.us`. :-)

Unix Network Programming, volumes 1-2 por W. Richard Stevens. Publicado pela Addison-Wesley Professional e Prentice Hall. ISBNs para os volumes 1-2: 978-0131411555 [49], 978-0130810816 [50].

Internetworking with TCP/IP, volume I por Douglas E. Comer. Publicado pela Pearson. ISBN 978-0136085300 [51].

TCP/IP Illustrated, volumes 1-3 por W. Richard Stevens and Gary R. Wright. Publicado pela Addison Wesley. ISBNs para os volumes 1, 2, e 3 (e um conjunto com 3 volumes): 978-0201633467 [52], 978-0201633542 [53], 978-0201634952 [54], (978-0201776317 [55]).

TCP/IP Network Administration por Craig Hunt. Publicado pela O'Reilly & Associates, Inc. ISBN 978-0596002978 [56].

Advanced Programming in the UNIX Environment por W. Richard Stevens. Publicado por Addison Wesley. ISBN 978-0321637734 [57].

Web Referências

Na web:

BSD Sockets: A Quick And Dirty Primer [58] (Informações de programação do sistema Unix também!)

The Unix Socket FAQ[59]

TCP/IP FAQ[60]

The Winsock FAQ[61]

E aqui estão algumas páginas relevantes na Wikipédia:

Berkeley Sockets[62]

Internet Protocol (IP)[63]

Transmission Control Protocol (TCP)[64]

User Datagram Protocol (UDP)[65]

Client-Server[66]

Serialization[67] (empacotando e desempacotando dados)

RFCs

RFCs [68]—A sujeira real! Estes são documentos que descrevem números atribuídos, APIs de programação e protocolos usados na Internet. Incluí links para alguns deles aqui para sua diversão, então pegue um balde de pipoca e coloque seu boné do pensamento:

RFC 1 [69]—A primeiro RFC; Isso lhe dá uma idéia de como era a "Internet", assim como ela estava ganhando vida, e uma visão de como ela estava sendo projetada desde o zero. (Esta RFC é completamente obsoleta, obviamente!)

RFC 768[70] —The User Datagram Protocol (UDP)

RFC 791[71] —The Internet Protocol (IP)

RFC 793[72] —The Transmission Control Protocol (TCP)

RFC 854[73] —The Telnet Protocol

RFC 959[74] —File Transfer Protocol (FTP)

RFC 1350[75] —The Trivial File Transfer Protocol (TFTP)

RFC 1459[76] —Internet Relay Chat Protocol (IRC)

RFC 1918[77] —Address Allocation for Private Internets

RFC 2131[78] —Dynamic Host Configuration Protocol (DHCP)

RFC 2616[79] —Hypertext Transfer Protocol (HTTP)

RFC 2821[80] —Simple Mail Transfer Protocol (SMTP)

RFC 3330[81] —Special-Use IPv4 Addresses

RFC 3493[82] —Basic Socket Interface Extensions for IPv6

RFC 3542[83] —Advanced Sockets Application Program Interface (API) for IPv6

RFC 3849[84] —IPv6 Address Prefix Reserved for Documentation

RFC 3920[85] —Extensible Messaging and Presence Protocol (XMPP)

RFC 3977[86] —Network News Transfer Protocol (NNTP)

RFC 4193[87] —Unique Local IPv6 Unicast Addresses

RFC 4506[88] —External Data Representation Standard (XDR)

O IETF possuia uma boa ferramenta online para pesquisa e navegação entre RFCs [89].

---

1. https://www.linux.com/↩
2. https://bsd.org/↩
3. https://cygwin.com/↩
4. https://docs.microsoft.com/en-us/windows/wsl/about↩
5. https://tangentsoft.net/wskfaq/↩
6. http://www.catb.org/~esr/faqs/smart-questions.html↩
7. https://beej.us/guide/bgnet/examples/telnot.c↩
8. https://tools.ietf.org/html/rfc854↩
9. https://tools.ietf.org/html/rfc793↩
10. https://tools.ietf.org/html/rfc791↩
11. https://tools.ietf.org/html/rfc768↩
12. https://tools.ietf.org/html/rfc791↩
13. https://en.wikipedia.org/wiki/Vint_Cerf↩
14. https://en.wikipedia.org/wiki/ELIZA↩
15. https://www.iana.org/assignments/port-numbers↩
16. https://en.wikipedia.org/wiki/Doom_(1993_video_game)↩
17. https://en.wikipedia.org/wiki/Wilford_Brimley↩
18. https://tools.ietf.org/html/rfc1918↩
19. https://tools.ietf.org/html/rfc4193↩
20. https://www.iana.org/assignments/port-numbers↩
21. https://beej.us/guide/bgnet/examples/showip.c↩
22. https://tools.ietf.org/html/rfc1413↩
23. https://beej.us/guide/bgnet/examples/server.c↩
24. https://beej.us/guide/bgnet/examples/client.c↩
25. https://beej.us/guide/bgnet/examples/listener.c↩

26. https://beej.us/guide/bgnet/examples/talker.c↩
27. https://libevent.org/↩
28. https://beej.us/guide/bgnet/examples/poll.c↩
29. https://beej.us/guide/bgnet/examples/pollserver.c↩
30. https://libevent.org/↩
31. https://beej.us/guide/bgnet/examples/select.c↩
32. https://beej.us/guide/bgnet/examples/selectserver.c↩
33. https://en.wikipedia.org/wiki/Internet_Relay_Chat↩
34. https://beej.us/guide/bgnet/examples/pack.c↩
35. https://en.wikipedia.org/wiki/IEEE_754↩
36. https://beej.us/guide/bgnet/examples/ieee754.c↩
37. https://beej.us/guide/url/tpop↩
38. https://github.com/protobuf-c/protobuf-c↩
39. https://beej.us/guide/bgnet/examples/pack2.c↩
40. https://beej.us/guide/bgnet/examples/pack2.c↩
41. https://tools.ietf.org/html/rfc4506↩
42. https://beej.us/guide/bgnet/examples/broadcaster.c↩
43. http://www.unpbook.com/src.html↩
44. http://www.unpbook.com/src.html↩
45. https://www.openssl.org/↩
46. https://stackoverflow.com/questions/21323023/↩
47. https://www.iana.org/assignments/port-numbers↩
48. https://www.iana.org/assignments/port-numbers↩
49. https://beej.us/guide/url/unixnet1↩
50. https://beej.us/guide/url/unixnet2↩
51. https://beej.us/guide/url/intertcp1↩
52. https://beej.us/guide/url/tcpi1↩
53. https://beej.us/guide/url/tcpi2↩

54. https://beej.us/guide/url/tcpi3↩
55. https://beej.us/guide/url/tcpi123↩
56. https://beej.us/guide/url/tcpna↩
57. https://beej.us/guide/url/advunix↩
58. https://cis.temple.edu/~giorgio/old/cis307s96/readings/docs/sockets.html↩
59. https://developerweb.net/?f=70↩
60. http://www.faqs.org/faqs/internet/tcp-ip/tcp-ip-faq/part1/↩
61. https://tangentsoft.net/wskfaq/↩
62. https://en.wikipedia.org/wiki/Berkeley_sockets↩
63. https://en.wikipedia.org/wiki/Internet_Protocol↩
64. https://en.wikipedia.org/wiki/Transmission_Control_Protocol↩
65. https://en.wikipedia.org/wiki/User_Datagram_Protocol↩
66. https://en.wikipedia.org/wiki/Client-server↩
67. https://en.wikipedia.org/wiki/Serialization↩
68. https://www.rfc-editor.org/↩
69. https://tools.ietf.org/html/rfc1↩
70. https://tools.ietf.org/html/rfc768↩
71. https://tools.ietf.org/html/rfc791↩
72. https://tools.ietf.org/html/rfc793↩
73. https://tools.ietf.org/html/rfc854↩
74. https://tools.ietf.org/html/rfc959↩
75. https://tools.ietf.org/html/rfc1350↩
76. https://tools.ietf.org/html/rfc1459↩
77. https://tools.ietf.org/html/rfc1918↩
78. https://tools.ietf.org/html/rfc2131↩
79. https://tools.ietf.org/html/rfc2616↩
80. https://tools.ietf.org/html/rfc2821↩
81. https://tools.ietf.org/html/rfc3330↩

82. https://tools.ietf.org/html/rfc3493↩
83. https://tools.ietf.org/html/rfc3542↩
84. https://tools.ietf.org/html/rfc3849↩
85. https://tools.ietf.org/html/rfc3920↩
86. https://tools.ietf.org/html/rfc3977↩
87. https://tools.ietf.org/html/rfc4193↩
88. https://tools.ietf.org/html/rfc4506↩
89. https://tools.ietf.org/rfc/↩