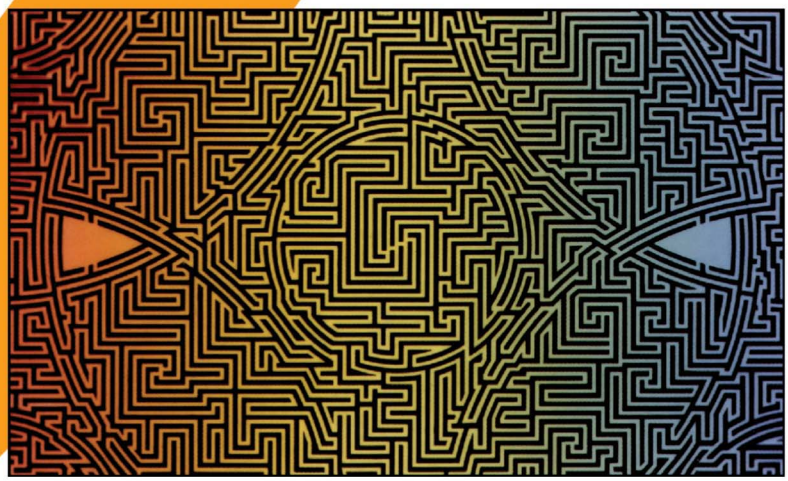
Programação de Rede UNIX®

API para soquetes de rede

VOLUME 1

TERCEIRA EDIÇÃO



W. RICHARD STEVENS
BILL FENNER
ANDREW M. RUDOFF

bookman®

Programação de Rede UNIX®

Sobre os Autores

W. RICHARD STEVENS, falecido em 1999, é o autor original de Programação de Rede Unix, Primeira e Segunda Edições, amplamente reconhecido como o texto clássico para rede UNIX.

BILL FENNER é Principal Technical Staff Member na AT&T Labs em Menlo Park, Califórnia, especializado em IP multicasting, gerenciamento de rede e medição. Ele é um dos Routing Area Directors do IETF, responsável pela aprovação de todos os documentos relacionados a roteamento publicados como RFCs.

ANDREW M. RUDOFF, engenheiro de *software* sênior na Sun Microsystems, é especializado em redes, sistemas operacionais, sistemas de arquivos e arquitetura de *software* de alta disponibilidade.

S845p Stevens, W. Richard.
Programação de rede Unix [recurso eletrônico] : API para soquetes de rede / W. Richad Stevens, Bill Fenner, Andrew M. Rudoff ; tradução Edson Furmankiewicz. – 3. ed. – Dados eletrônicos. – Porto Alegre : Bookman, 2008.

Editado também como livro impresso em 2005.
ISBN 978-85-7780-240-1

1. Computação – Redes - Unix. I. Fenner, Bill. II. Rudoff, Andrew M. III. Título.

CDU 004.7/.78

Catalogação na publicação: Mônica Ballejo Canto – CRB 10/1023.

W. RICHARD STEVENS
BILL FENNER
ANDREW M. RUDOFF

Programação de Rede UNIX®

API para soquetes de rede

TERCEIRA EDIÇÃO

Tradução:
Edson Furmankiewicz

Consultoria, supervisão e revisão técnica desta edição:
Dr. Taisy Silva Weber
Professora do Instituto de Informática da UFRGS

Versão impressa
desta obra: 2005

Bookman®

2008

Obra originalmente publicada sob o título
Unix Network Programming, Volume 1: The Sockets Networking API, 3rd Edition

ISBN 0-13-141155-1

Tradução autorizada do inglês do livro de autoria de W. Richard Stevens, Bill Fenner e Andrew M. Rudoff, publicado por Pearson Education, Inc., sob o selo Addison Wesley Professional.

Capa: *Mário Röhnelt*

Leitura final: *Sandro Andretta*

Supervisão editorial: *Arysinha Jacques Affonso*

Editoração eletrônica: *Laser House*

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

Para Rich.
Aloha nui loa.

Apresentação

Quando o texto original deste livro foi publicado em 1990, ele foi rapidamente reconhecido como a referência definitiva para as técnicas de programação de rede. Desde então, a arte de interconectar computadores mudou dramaticamente. Basta ver o endereço de retorno para comentários do texto original ("uunet!hsi!netbook"). (Quantos leitores reconhecerão isso como um endereço na rede discada UUCP que era comum na década de 1980?)

Atualmente, as redes UUCP são uma raridade e novas tecnologias, como redes sem fio, estão se tornando onipresentes! Com essas mudanças, foram desenvolvidos novos protocolos de rede e paradigmas de programação. Mas os programadores sentiram a falta de uma boa referência para aprenderem as complexidades dessas novas técnicas.

Para isso escrevemos este livro. Leitores com uma cópia "gasta" do original irão querer uma nova cópia para as técnicas de programação atualizadas e o novo material substancial que descreve os protocolos de última geração, como o IPv6. Todos irão querer este livro porque ele fornece uma excelente combinação de experiência prática, perspectiva histórica e um entendimento profundo que provém somente do envolvimento íntimo com o campo.

Muito aprendi com a leitura deste livro e, com certeza, você também vai aprender.

Sam Leffler

Prefácio

Introdução

Este livro é para aqueles que querem escrever programas que se comunicam entre si utilizando uma interface de programa de aplicação (*application programming interface* – API) conhecida como soquetes. Alguns leitores talvez já conheçam bem os soquetes, visto que esse modelo tornou-se sinônimo de programação de rede. Outros talvez precisem de uma introdução a eles a partir do zero. O objetivo deste livro é oferecer uma orientação sobre programação de rede tanto para iniciantes como para profissionais, para todos aqueles que desenvolvem novas aplicações rede ou que mantêm o código existente ou que simplesmente querem entender como os componentes de rede dos seus sistemas funcionam.

Todos os exemplos neste texto são códigos executáveis reais testados em sistemas Unix. Contudo, vários sistemas não-Unix suportam a API de soquetes e a maioria dos exemplos é independente de sistema operacional, como ocorre com os conceitos gerais que apresentaremos. Praticamente todo sistema operacional (SO) fornece várias aplicações de rede, como navegadores Web, clientes de correio eletrônico e servidores de compartilhamento de arquivos. Discutiremos o particionamento normal dessas aplicações em *cliente* e *servidor* e escreveremos nossos próprios pequenos exemplos de tais aplicações várias vezes neste livro.

A apresentação desse material de uma maneira orientada ao Unix tem o efeito colateral natural de fornecer um *background* sobre o próprio Unix e também sobre o TCP/IP. Sempre que um conhecimento mais extenso puder ser interessante, forneceremos ao leitor uma referência a outros textos. Assim, quatro textos serão comumente mencionados neste livro, aos quais atribuímos as seguintes abreviações:

- APUE: *Advanced Programming in the UNIX Environment* (Stevens, 1992)
- TCPv1: *TCP/IP Illustrated, Volume 1* (Stevens, 1994)
- TCPv2: *TCP/IP Illustrated, Volume 2* (Wright e Stevens, 1995)
- TCPv3: *TCP/IP Illustrated, Volume 3* (Stevens, 1996)

O TCPv2 contém um alto nível de detalhes intimamente relacionados ao material deste livro, uma vez que descreve e apresenta a implementação 4.4BSD real das funções de progra-

mação de rede para a API de soquetes (`socket`, `bind`, `connect`, e assim por diante). Se uma pessoa entender a implementação de um recurso, o uso desse recurso em uma aplicação fará mais sentido.

Modificações na Segunda Edição

Os soquetes estão em uso, mais ou menos na sua forma atual, desde a década de 1980 e é um tributo à sua concepção inicial o fato de continuarem a ser a escolha para APIs de redes. Portanto, pode ser uma surpresa aprender que várias coisas mudaram desde a segunda edição deste livro, publicada em 1998. As alterações que fizemos estão resumidas assim:

- Esta nova edição contém informações atualizadas sobre o IPv6, que, na época da segunda edição, era apenas uma proposta (*draft*) e evoluiu relativamente bastante.
- As descrições de funções e os exemplos foram atualizados para refletir a especificação POSIX mais recente (POSIX 1003.1-2001), também conhecida como *Single Unix Specification Version 3*.
- A abordagem da X/Open Transport Interface (XTI) foi descartada. Essa API não é mais comumente utilizada e mesmo a especificação POSIX mais recente não se preocupa em abordá-la.
- A abordagem do TCP para transações (T/TCP) foi descartada.
- Três capítulos foram adicionados para descrever um protocolo relativamente novo de transporte, o SCTP. Esse protocolo confiável, orientado a mensagens, fornece múltiplos fluxos entre as extremidades e o suporte no nível de transporte a multihoming. Originalmente, ele foi projetado para transporte de sinalização de telefonia pela Internet, mas fornece alguns recursos dos quais várias aplicações poderiam tirar proveito.
- Foi acrescentado um capítulo sobre *soquetes de gerenciamento de chave*, que podem ser utilizados com o Internet Protocol Security (IPSec) e outros serviços de segurança de rede.
- As máquinas usadas, bem como as versões das suas variantes do Unix, foram todas atualizadas, juntamente com os exemplos, para refletir como essas máquinas se comportam. Em muitos casos, os exemplos foram atualizados porque os fornecedores de SO corrigiram bugs ou acrescentaram recursos, mas, como poderíamos esperar, descobrimos novos bugs ocasionais aqui e ali. As máquinas utilizadas para testar os exemplos neste livro foram:
  - Apple Power PC executando um MacOS/X 10.2.6
  - HP PA-RISC executando HP-UX 11i
  - IBM Power PC executando AIX 5.1
  - Intel x86 executando FreeBSD 4.8
  - Intel x86 executando Linux 2.4.7
  - Sun SPARC executando FreeBSD 5.1
  - Sun SPARC executando Solaris 9

  Veja a Figura 1.16 para detalhes sobre como essas máquinas foram utilizadas.

O Volume 2 desta série, *Programação de rede Unix*, *Comunicações interprocesso*, desenvolve-se a partir do material apresentado aqui para abranger passagem de mensagem, sincronização, memória compartilhada e chamadas de procedimento remoto.

Utilização do livro

Este livro pode ser usado como um tutorial sobre programação de rede ou como uma referência para programadores experientes. Se utilizado como um tutorial ou uma aula introdutória

sobre programação de rede, a ênfase deve ser sobre a Parte 2, "Soquetes elementares" (Capítulos 3 a 11), seguida por quaisquer tópicos adicionais de interesse. A Parte 2 abrange as funções básicas de soquete tanto para TCP como para UDP, junto com SCTP, multiplexação de E/S, opções de soquete e conversões básicas de nomes e de endereços. O Capítulo 1 é de leitura obrigatória para todos, especialmente a Seção 1.4, que descreve algumas funções empacotadoras utilizadas por todo o livro. O Capítulo 2 e talvez o Apêndice A devem ser consultados conforme necessário, dependendo da formação do leitor. A maioria dos capítulos na Parte 3, "Soquetes avançados", pode ser lida de maneira independente dos demais capítulos nessa parte do livro.

Para auxiliar o uso deste livro como uma referência, há um índice completo, junto com resumos sobre artigos finais de onde encontrar descrições detalhadas sobre todas as funções e estruturas. Para ajudar a leitura desses tópicos em uma ordem aleatória, são fornecidas numerosas referências aos tópicos relacionados neste livro.

Disponibilidade do código-fonte e erratas

O código-fonte para todos os exemplos que serão apresentados está disponível na Web em `www.unpbook.com`. A melhor maneira de aprender programação de rede é pegar esses programas, modificá-los e aprimorá-los. Na verdade, escrever um código dessa forma é a *única maneira* de reforçar os conceitos e as técnicas. Vários exercícios também são fornecidos no final de cada capítulo e a maioria das respostas se encontra no Apêndice E.

Uma lista atualizada de erratas deste livro também está disponível no mesmo site da Web.

Agradecimentos

A primeira e a segunda edição deste livro foram escritas exclusivamente por W. Richard Stevens, que faleceu em 1º setembro de 1999. Seus livros estabeleceram um alto padrão e são considerados obras-primas extremamente legíveis, diligentemente detalhados e concisos. Ao fazerem esta revisão, os autores esforçaram-se para manter a qualidade e a abrangência completa das primeiras edições de W. Richard Stevens, assumindo quaisquer deficiências que possam ser encontradas.

O trabalho de um autor é diretamente proporcional ao suporte de seus familiares e amigos. Bill Fenner agradece a sua querida esposa, Peggy, e a Christopher Boyd, com quem ele divide a casa, por pouparem-no de todos os afazeres domésticos enquanto trabalhava neste projeto. Bill Fenner agradece também a seu amigo Jerry Winner, cujo apoio e estímulo foram inestimáveis. Da mesma forma, Andy Rudoff agradece especificamente a sua esposa, Ellen, e às filhas, Jo e Katie, pela compreensão e estímulo ao longo de todo o projeto. Nós simplesmente não poderíamos ter escrito este livro sem o apoio de vocês!

Randall Stewart, com a Cisco Systems, Inc., forneceu boa parte do material sobre SCTP e merece um reconhecimento especial por essa contribuição. A abordagem deste novo e interessante tópico simplesmente não existiria sem a sua ajuda.

O *feedback* de nossos revisores foi inestimável para descobrir erros, indicar áreas que exigiam mais explicações e sugerir melhorias ao texto e aos exemplos de código. Os autores agradecem a: James Carlson, Wu-Chang Feng, Rick Jones, Brian Kernighan, Sam Leffler, John McCann, Craig Metz, Ian Lance Taylor, David Schwartz e Gary Wright.

Várias pessoas e suas empresas foram além do esperado ao emprestarem um sistema ou fornecerem um *software* ou mesmo o acesso a um sistema – tudo utilizado para testar alguns exemplos neste livro.

- Jessie Haug, da IBM Austin, forneceu um sistema AIX e compiladores.
- Rick Jones e William Gilliam, da Hewlett-Packard, forneceram acesso a múltiplos sistemas executando HP-UP.

Trabalhar com o pessoal da Addison Wesley foi um verdadeiro prazer: Noreen Regina, Kathleen Caren, Dan DePasquale, Anthony Gemellaro e um agradecimento muito especial a nossa editora, Mary Franz.

Em uma tendência que Rich Stevens instituiu (mas contrária a modismos populares), produzimos uma cópia final deste livro utilizando o maravilhoso pacote Groff escrito por James Clark, criamos as ilustrações com o programa `gpic` (utilizando várias macros de Gary Wright), produzimos as tabelas com o programa `gtbl`, realizamos toda a indexação e fizemos o layout de página final. O programa `loom` de Dave Hansori e alguns scripts de Gary Wright foram utilizados para incluir o código-fonte neste livro. Um conjunto de scripts `awk` escritos por Jon Bentley e Brian Kernighan ajudou a produzir o índice final.

Os autores dão boas-vindas a mensagens de correio eletrônico de todos os leitores com comentários, sugestões ou correções de bugs.

Bill Fenner
Woodside, *California*

Andrew M. Rudoff
Boulder, *Colorado*

`authors@unpbook.com`
`http://www.unpbook.com`

Sumário

PARTE I

Introdução e TCP/IP

Capítulo 1

Introdução

1.1 Visão geral

Ao escrever programas que se comunicam por uma rede de computadores, deve-se primeiro criar um *protocolo*, um acordo sobre como esses programas irão comunicar-se. Antes de nos aprofundarmos nos detalhes sobre o *design* de um protocolo, devemos tomar decisões de alto nível sobre qual programa esperamos que inicie a comunicação e quando as respostas são esperadas. Por exemplo, um servidor Web em geral é pensado como um programa de longa execução (ou *daemon*) que envia mensagens de rede somente em resposta a solicitações recebidas da rede. O outro lado do protocolo é um cliente Web, como um navegador, que sempre inicia a comunicação com o servidor. Essa organização em *cliente* e *servidor* é utilizada pela maioria das aplicações de rede. Decidir que o cliente sempre inicia as solicitações tende a simplificar o protocolo e os próprios programas. Naturalmente, algumas aplicações mais complexas de rede também requerem uma comunicação de *retorno de chamada assíncrono (asynchronous callback)*, em que o servidor inicia uma mensagem para o cliente. Mas é bem mais comum que aplicações fixem-se no modelo básico cliente/servidor mostrado na Figura 1.1.

Os clientes normalmente se comunicam com um servidor de cada vez; embora usando um navegador na Web como exemplo, poderíamos nos comunicar com diferentes servidores Web, digamos, por um período de tempo de 10 minutos. Mas, da perspectiva do servidor, em qualquer ponto dado no tempo, é comum a um servidor estar em comunicação com múltiplos clientes. Mostramos isso na Figura 1.2. Mais adiante, abordaremos várias maneiras de um servidor tratar múltiplos clientes ao mesmo tempo.

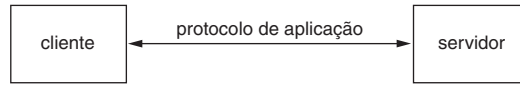


Figura 1.1 Aplicação de rede: cliente e servidor.

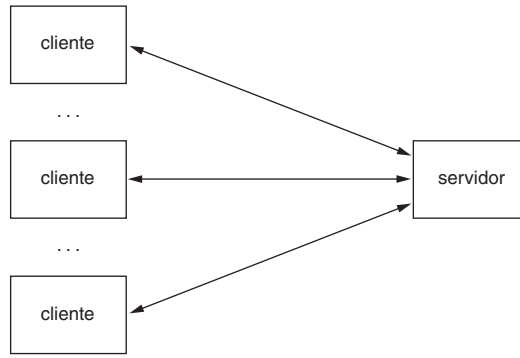


Figura 1.2 Um servidor tratando múltiplos clientes ao mesmo tempo.

A aplicação cliente e a aplicação servidor podem ser pensadas como se comunicando via um protocolo de rede, mas, na verdade, múltiplas camadas de protocolos de rede estão em geral envolvidas. Neste texto, vamos nos concentrar no conjunto de protocolos TCP/IP, também chamado conjunto de protocolos Internet (*Internet protocol suite*). Por exemplo, clientes e servidores Web comunicam-se utilizando o Transmission Control Protocol ou TCP. O TCP, por sua vez, utiliza o Internet Protocol, ou IP, e o IP comunica-se de alguma forma com a camada de enlace de dados. Se o cliente e o servidor estivessem na mesma Ethernet, teríamos o arranjo mostrado na Figura 1.3.

Ainda que o cliente e o servidor comuniquem-se utilizando um protocolo de aplicação, as camadas de transporte comunicam-se utilizando TCP. Observe que o fluxo real das informações entre o cliente e o servidor desce pela pilha de protocolos de um lado, através da rede, e sobe pela pilha de protocolos do outro lado. Também observe que o cliente e o servidor são em geral processos de usuários, enquanto os protocolos TCP e IP normalmente fazem parte da pilha de protocolos dentro do kernel. Rotulamos as quatro camadas no lado direito da Figura 1.3.

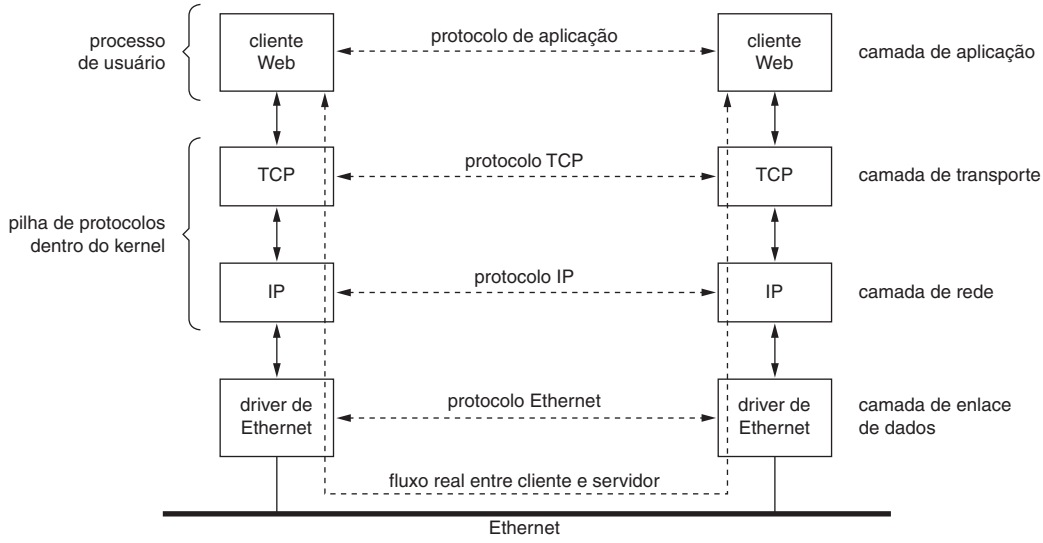


Figura 1.3 Cliente e servidor comunicando-se na mesma Ethernet usando TCP.

TCP e IP não são os únicos protocolos que discutiremos. Alguns clientes e servidores utilizam o User Datagram Protocol (UDP) em vez do TCP; discutiremos ambos em mais detalhes no Capítulo 2. Além disso, utilizamos o termo "IP", mas o protocolo, que tem sido utilizado desde o início da década de 1980, é oficialmente chamado *IP versão 4* (IPv4). Uma nova versão, o *IP versão 6* (IPv6), foi desenvolvida durante meados da década de 1990 e poderia potencialmente substituir o IPv4 nos próximos anos. Este texto abordará o desenvolvimento de aplicações de rede utilizando tanto o IPv4 como o IPv6. O Apêndice A fornece uma comparação entre o IPv4 e o IPv6, juntamente com outros protocolos que discutiremos.

O cliente e o servidor não precisam estar ligados na mesma *rede local* (Local Area Network – LAN), como mostramos na Figura 1.3. Por exemplo, na Figura 1.4, mostramos o cliente e o servidor em diferentes redes locais, ambas conectadas a uma *rede geograficamente distribuída* (Wide Area Network – WAN) utilizando *roteadores*.

Os roteadores são os blocos de construção das redes geograficamente distribuídas. A maior rede geograficamente distribuída da atualidade é a *Internet*. Muitas empresas constroem suas próprias WANs, as quais podem ou não estar conectadas à Internet.

O restante deste capítulo fornecerá uma introdução aos vários tópicos que serão abordados em detalhes mais adiante. Iniciaremos com um exemplo completo de um cliente TCP, simples, mas que demonstra várias chamadas de função e conceitos que encontraremos por todo o texto. Esse cliente funciona apenas com o IPv4 e mostraremos as alterações necessárias para funcionar com o IPv6. Uma solução mais adequada é escrever clientes e servidores independentes de protocolos, o que será discutido no Capítulo 11. Este capítulo também mostrará um servidor de TCP completo que funciona com o nosso cliente.

Para simplificar o nosso código, definiremos nossas próprias funções empacotadoras para a maioria das funções de sistema que usaremos neste texto. Podemos utilizar essas funções empacotadoras na maioria das vezes para verificar um erro, imprimir uma mensagem apropriada e terminar quando um erro ocorrer. Também mostraremos a rede de teste, hosts e roteadores utilizados na maioria dos exemplos que apresentaremos, juntamente com seus hostnames, endereços IP e sistemas operacionais.

Atualmente, muitas das discussões sobre Unix incluem o termo "X", que é o padrão que a maioria dos fornecedores tem adotado. Descreveremos a história do POSIX e a maneira co-

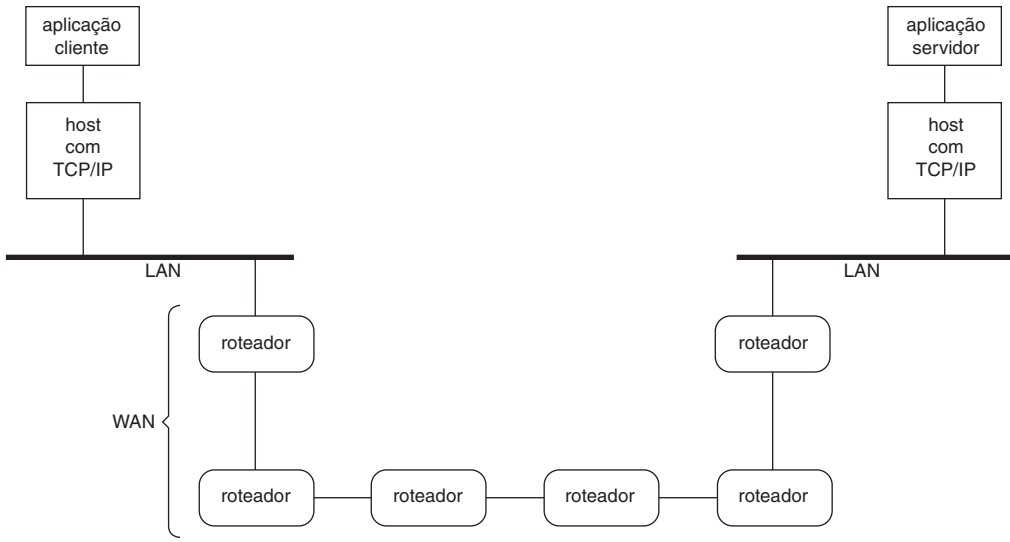


Figura 1.4 O cliente e o servidor em diferentes redes locais (LANs) conectados por meio de uma WAN.

mo ele afeta as interfaces de programas aplicativos (Applications Programming Interfaces – APIs) que serão citadas ao longo do texto, juntamente com os outros importantes desenvolvedores no campo dos padrões.

1.2 Um cliente de data/hora simples

Vamos considerar um exemplo específico para introduzir boa parte dos conceitos e termos que encontraremos por todo o livro. A Figura 1.5 é uma implementação de um cliente TCP de data/hora. Esse cliente estabelece uma conexão TCP com um servidor e o servidor simplesmente envia de volta a data e a hora atuais em um formato legível por humanos.

intro/daytimetcpcli.c

```
 1 #include "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int sockfd, n;
 6     char recvline[MAXLINE + 1];
 7     struct sockaddr_in servaddr;
 8     if (argc != 2)
 9         err_quit("usage: a.out <IPaddress>");
10     if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
11         err_sys("socket error");
12     bzero(&servaddr, sizeof(servaddr));
13     servaddr.sin_family = AF_INET;
14     servaddr.sin_port = htons(13); /* servidor de data/hora */
15     if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
16         err_quit("inet_pton error for %s", argv[1]);
17     if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
18         err_sys("connect error");
19     while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
20         recvline[n] = 0; /* termina com null */
21         if (fputs(recvline, stdout) == EOF)
22             err_sys("fputs error");
23     }
24     if (n < 0)
25         err_sys("read error");
26     exit(0);
27 }
```

intro/daytimetcpcli.c

Figura 1.5 Cliente TCP de data/hora.

Esse é o formato que utilizaremos para todo código-fonte neste livro. Cada linha não-vazia é numerada. O texto que descreve partes do código inclui o número das linhas inicial e final na margem esquerda, como será mostrado na próxima página. Às vezes, um parágrafo é precedido por um título descritivo, curto, em negrito, fornecendo um resumo do código sendo descrito.

As linhas horizontais no início e no final de um fragmento de código especificam o nome de arquivo do código-fonte: o arquivo `daytimetcpcli.c` no diretório `intro` para esse exemplo. Uma vez que o código-fonte de todos os exemplos está livremente disponível (consulte o Prefácio), isso permite localizar o arquivo-fonte apropriado. Compilar, executar e, especialmente, modificar esses programas é uma excelente maneira de aprender os conceitos de programação de rede. Portanto, sinta-se livre para isso.

Por todo o livro, utilizaremos notas recuadas, em corpo menor, como esta, para descrever os detalhes de implementação e os pontos históricos.

Se compilarmos e executarmos o programa no arquivo `a.out` padrão, teremos a seguinte saída:

```
solaris % a.out 206.168.112.96                    nossa entrada
Mon May 26 20:58:40 2003                          saída do programa
```

Sempre que exibirmos uma entrada e saída interativas, mostraremos nossa entrada digitada em **negrito** e a saída de computador em fonte monoespaçada, como esta. *Comentários serão adicionados do lado direito em itálico*. Sempre incluiremos o nome do sistema como parte do prompt de shell (`solaris`, nesse exemplo) para mostrar em qual host o comando foi executado. A Figura 1.16 mostra os sistemas utilizados para executar a maioria dos exemplos neste livro. Normalmente, os hostnames também descrevem o sistema operacional (SO).

Há muitos detalhes a serem considerados nesse programa de 27 linhas. Mencionaremos todos brevemente aqui, caso este seja seu primeiro contato com um programa de rede, e forneceremos informações adicionais sobre esses tópicos mais adiante.

Inclusão do nosso próprio cabeçalho

1 Incluímos nosso próprio cabeçalho, `unp.h`, que mostraremos na Seção D.1. Esse cabeçalho inclui diversos cabeçalhos de sistema necessários à maioria dos programas de rede e define várias constantes que utilizamos (por exemplo, `MAXLINE`).

Argumentos de linha de comando

2-3 Essa é a definição da função `main` junto com os argumentos de linha de comando. Escrevemos o código nesse texto adotando um compilador C do American National Standards Institute (ANSI) (também conhecido como compilador ISO C).

Criação do soquete TCP

10-11 A função `socket` cria um "*Internet* (`AF_INET`) *stream* (`SOCK_STREAM`) *socket*", um nome sofisticado para um soquete TCP. A função retorna um descritor do tipo *small integer* que podemos utilizar para identificar o soquete em todas as chamadas de função futuras (por exemplo, as chamadas a `connect` e `read` que se seguem).

A instrução `if` contém uma chamada à função `socket`, uma atribuição do valor de retorno à variável identificada `sockfd` e, então, um teste para saber se esse valor atribuído é menor que 0. Embora possamos dividir isso em duas instruções C,

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
```

é um estilo C comum combinar as duas linhas. O conjunto de parênteses em torno da chamada de função e atribuição é obrigatório, dadas as regras de precedência de C (o operador menor que tem uma precedência mais alta que a atribuição). Como uma questão de estilo de codificação, os autores sempre colocam um espaço entre os dois parênteses de abertura, como um indicador visual de que o lado esquerdo da comparação também é uma atribuição. (Esse estilo é copiado do código-fonte Minix [Tanenbaum 1987].) Utilizamos esse mesmo estilo na instrução `while` mais adiante no programa.

Encontraremos diferentes utilizações do termo "soquete". Primeiro, a API que estamos utilizando é chamada API de soquetes. No parágrafo precedente, referimo-nos a uma função identificada `socket` que faz parte da API de soquetes e também a um soquete TCP, que é sinônimo de uma extremidade de TCP.

Se a chamada a `socket` falhar, abortamos o programa chamando nossa própria função `err_sys`. Ela imprime nossa mensagem de erro juntamente com uma descrição do erro de

sistema que ocorreu (por exemplo, "Protocolo não suportado" é um dos possíveis erros de `socket`) e termina o processo. Essa função, e algumas outras nossas que iniciam com `err_`, são chamadas por todo o texto. Descreveremos essas funções na Seção D.3.

Especificação do endereço IP e da porta do servidor

12-16 Preenchemos uma estrutura de endereços de soquetes de Internet (uma estrutura `sockaddr_in` identificada como `servaddr`) com o endereço IP e o número da porta do servidor. Configuramos toda a estrutura como 0 utilizando `bzero`, configuramos a família de endereços como `AF_INET`, configuramos o número da porta como 13 (que é a porta bem-conhecida do servidor de data/hora em qualquer host TCP/IP que suporte esse serviço, como mostrado na Figura 2.18) e configuramos o endereço IP com o valor especificado como o primeiro argumento de linha de comando (`argv[1]`). Os campos de endereço IP e de número da porta nessa estrutura devem ter formatos específicos: chamamos a função de biblioteca `htons` ("host to network short") para converter o número de porta binário e a função de biblioteca `inet_pton` ("presentation to numeric") para converter o argumento de linha de comando ASCII (como `206.62.226.35` quando executamos esse exemplo) ao formato adequado.

> `bzero` não é uma função ANSI C. Ele é derivado do código de rede Berkeley inicial. Contudo, nós o utilizaremos por todo o texto, em vez da função ANSI C `memset`, porque `bzero` é mais fácil de lembrar (com somente dois argumentos) do que `memset` (com três argumentos). Quase todos os fornecedores que suportam a API de soquetes também fornecem `bzero`; se não, fornecemos uma definição de macro dele no nosso cabeçalho `unp.h`.
>
> De fato, o autor do TCPv3 cometeu o equívoco de inverter o segundo e o terceiro argumentos de `memset` em 10 ocorrências na primeira impressão. Um compilador C não pode capturar esse erro porque os dois argumentos são do mesmo tipo. (Na verdade, o segundo argumento é um `int` e o terceiro é `size_t`, que em geral é um `unsigned int`, mas os valores especificados, 0 e 16, respectivamente, ainda são aceitáveis para o outro tipo de argumento). A chamada a `memset` ainda funcionava, mas não fazia nada. O número de bytes a ser inicializado era especificado como 0. Os programas funcionavam porque, na verdade, somente algumas das funções de soquete requerem que os 8 bytes finais de uma estrutura de endereços de soquetes Internet sejam configurados como 0. Contudo, isso era um erro que poderia ser evitado utilizando `bzero`, pois trocar os dois argumentos para `bzero` sempre será capturado pelo compilador C se protótipos de função forem utilizados.
>
> Este pode ser seu primeiro contato com a função `inet_pton`. Ela é nova no IPv6 (que discutiremos em mais detalhes no Apêndice A). Códigos mais antigos utilizam a função `inet_addr` para converter uma string de pontos decimais ASCII ao formato correto, mas essa função tem inúmeras limitações que `inet_pton` corrige. Não se preocupe se seu sistema não suportar (ainda) essa função; forneceremos uma implementação dela na Seção 3.7.

Estabelecimento de uma conexão com o servidor

17-18 A função `connect`, quando aplicada a um soquete TCP, estabelece uma conexão TCP com o servidor especificado pela estrutura de endereços de soquetes apontada pelo segundo argumento. Também devemos especificar o comprimento da estrutura de endereços de soquetes como o terceiro argumento para `connect` e, para estruturas de endereços de soquetes de Internet, sempre deixamos o compilador calcular o comprimento utilizando o operador `sizeof` do C.

> No cabeçalho `unp.h`, utilizamos `define SA` para definir `SA` como `struct sockaddr`, isto é, uma estrutura genérica de endereços de soquetes. Cada vez que uma das funções de soquete requer um ponteiro para uma estrutura de endereços de soquetes, esse ponteiro deve ser convertido em um ponteiro para uma estrutura genérica de endereços de soquetes. Isso ocorre porque as funções de soquete são anteriores ao padrão ANSI C, portanto, o tipo de ponteiro `void *` não estava disponível no início da década de 1980, quando essas funções foram desenvolvidas. O problema é que `"struct sockaddr"` tem 15 caracteres e freqüente-

mente faz com que a linha do código-fonte se estenda além da borda direita da tela (ou página, no caso de um livro), por isso nós a abreviamos para SA. Discutiremos as estruturas genéricas de endereços de soquetes em mais detalhes ao explicar a Figura 3.3.

Leitura e exibição da resposta do servidor

19-25 Utilizamos `read` para obter a resposta do servidor e exibimos o resultado utilizando a função `fputs` de E/S-padrão. Devemos ter cuidado ao utilizar o TCP porque ele é um protocolo de fluxo de bytes (*byte-stream*) sem limites de registro. A resposta do servidor normalmente é uma string de 26 bytes na forma

```
Mon May 26 20:58:40 2003\r\n
```

no qual `\r` é o retorno de carro ASCII e `\n` é a quebra de linha ASCII. Com um protocolo de fluxo de bytes, esses 26 bytes podem ser retornados de várias maneiras: um único segmento de TCP contendo todos os 26 bytes de dados, 26 segmentos de TCP, cada um contendo 1 byte de dados, ou qualquer outra combinação que totalize 26 bytes. Normalmente, um único segmento contendo todos os 26 bytes de dados é retornado, mas com tamanhos maiores de dados não podemos supor que a resposta do servidor será retornada por um único `read`. Portanto, ao ler a partir de um soquete TCP, sempre precisamos codificar `read` em um loop e terminar o loop quando `read` retorna 0 (isto é, a outra extremidade fechou a conexão) ou um valor menor que 0 (um erro).

Nesse exemplo, o final do registro é denotado pelo servidor que fecha a conexão. Essa técnica também é utilizada pela versão 1.0 do Hypertext Transfer Protocol (HTTP). Outras técnicas estão disponíveis. Por exemplo, o Simple Mail Transfer Protocol (SMTP) marca o final de um registro com a seqüência de dois bytes de um retorno de carro ASCII seguido por uma quebra de linha ASCII. A chamada de procedimento remoto (Remote Procedure Call – RPC) da Sun e o Domain Name System (DNS) posicionam uma contagem binária que contém o tamanho do registro na frente de cada registro enviado utilizando o TCP. Aqui, o conceito importante é que o TCP por si só não fornece nenhum marcador de registro: se uma aplicação quiser delinear o final de registros, ela mesma deve fazer isso, e há algumas maneiras comuns para tal.

Término do programa

26 `exit` termina o programa. O Unix sempre fecha todos os descritores abertos quando um processo termina, assim nosso soquete TCP agora está fechado.

Como mencionamos, o texto trará mais detalhes sobre todos os pontos que acabamos de descrever.

1.3 Independência de protocolo

Nosso programa na Figura 1.5 é *dependente de protocolo* no IPv4. Alocamos e inicializamos uma estrutura `sockaddr_in`, configuramos a família dessa estrutura como `AF_INET` e especificamos o primeiro argumento para `socket` como `AF_INET`.

Para modificar o programa e fazer com que ele funcione com o IPv6, devemos modificar o código. A Figura 1.6 mostra uma versão que funciona com o IPv6, com as modificações destacadas em negrito.

intro/daytimetcpcliv6.c

```
1 #include "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     sockfd, n;
```

Figura 1.6 A versão da Figura 1.5 para o IPv6 (*continua*).

```
 6     char    recvline[MAXLINE + 1];
 7     struct sockaddr_in6 servaddr;

 8     if (argc != 2)
 9         err_quit("usage: a.out <IPaddress>");

10     if ( (sockfd = socket(AF_INET6, SOCK_STREAM, 0)) < 0)
11         err_sys("socket error");

12     bzero(&servaddr, sizeof(servaddr));
13     servaddr.sin6_family = AF_INET6;
14      servaddr.sin6_port = htons(13);     /* servidor de data/hora */
15     if (inet_pton(AF_INET6, argv[1], &servaddr.sin6_addr) <= 0)
16         err_quit("inet_pton error for %s", argv[1]);

17     if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
18         err_sys("connect error");

19     while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
20          recvline[n] = 0;    /* termina com null */
21         if (fputs(recvline, stdout) == EOF)
22             err_sys("fputs error");
23     }
24     if (n < 0)
25         err_sys("read error");

26     exit(0);
27 }
```

intro/daytimetcpcliv6.c

Figura 1.6 A versão da Figura 1.5 para o IPv6 (*continuação*).

Somente cinco linhas são alteradas, mas o que temos agora é um outro programa dependente de protocolo; desta vez, ele é dependente do IPv6. O melhor é torná-lo um programa *independente de protocolo*. A Figura 11.11 mostrará uma versão desse cliente, independente de protocolo, utilizando a função `getaddrinfo` (que é chamada por `tcp_connect`).

Uma outra deficiência nos nossos programas é que o usuário deve inserir o endereço IP do servidor como um número de pontos decimais (por exemplo, 206.168.112.219 para a versão do IPv4). Humanos trabalham melhor com nomes em vez de números (por exemplo, `www.unpbook.com`). No Capítulo 11, discutiremos as funções que convertem entre hostnames e endereços IP e entre nomes de serviço e portas. Adiamos a discussão dessas funções de propósito e continuaremos a utilizar endereços IP e números de porta de modo que saibamos exatamente o que entra nas estruturas de endereços de soquetes que devemos preencher e examinar. Isso também evita complicar nossa discussão sobre programação de rede com detalhes de mais outro conjunto de funções.

1.4 Tratamento de erro: funções empacotadoras

Em qualquer programa do mundo real, é essencial verificar se ocorre retorno de erro em *cada* chamada de função. Na Figura 1.5, verificamos erros em `socket`, `inet_pton`, `connect`, `read` e `fputs`; e, quando um erro ocorre, chamamos nossas próprias funções, `err_quit` e `err_sys`, para imprimir uma mensagem de erro e terminar o programa. Descobrimos que na maioria das vezes isso é o que queremos fazer. Ocasionalmente, queremos fazer algo além de terminar quando uma dessas funções retorna um erro, como na Figura 5.12, quando devemos verificar uma chamada de sistema interrompida.

Como terminar em um erro é um caso comum, podemos encurtar nossos programas definindo uma *função empacotadora* que realiza a chamada de função real, testa o valor de retorno e termina em caso de erro. A convenção que utilizamos é escrever o nome da função em letras maiúsculas, como em

```
sockfd = Socket(AF_INET, SOCK_STREAM, 0);
```

Nossa função empacotadora é mostrada na Figura 1.7.

lib/wrapsock.c

```
236 int
237 Socket(int family, int type, int protocol)
238 {
239     int n;

240     if ( (n = socket(family, type, protocol)) < 0)
241         err_sys("socket error");
242     return (n);
243 }
```

lib/wrapsock.c

Figura 1.7 Nossa função empacotadora para a função `socket`.

> *Sempre que você encontrar um nome de função no texto que começa com letra maiúscula, essa é uma das nossas funções empacotadoras. Ele chama uma função cujo nome é o mesmo, mas que começa com letra minúscula.*
>
> *Ao descrever o código-fonte apresentado no texto, sempre nos referimos à função de nível mais baixo sendo chamada (por exemplo,* `socket`*), não à função empacotadora (por exemplo,* `Socket`*).*

Embora essas funções empacotadoras talvez não sejam uma grande economia, ao discutirmos threads no Capítulo 26, descobriremos que funções de thread não configuram a variável `errno` padrão do Unix quando um erro ocorre; em vez disso, o valor de `errno` é o valor de retorno da função. Isso significa que, cada vez que chamamos uma das funções `pthread_`, devemos alocar uma variável, salvar o valor de retorno nessa variável e então configurar `errno` como esse valor antes de chamar `err_sys`. Para evitar encher o código com chaves, podemos utilizar o operador vírgula do C para combinar a atribuição a `errno` e a chamada de `err_sys` em uma única instrução, como a seguir:

```
int   n;

if ( (n = pthread_mutex_lock(&ndone_mutex)) != 0)
    errno = n, err_sys("pthread_mutex_lock error");
```

Alternativamente, poderíamos definir uma nova função de erro que recebe o número de erro do sistema como um argumento. Mas também podemos tornar esse fragmento de código muito mais fácil de ler, como simplesmente

```
Pthread_mutex_lock(&ndone_mutex);
```

definindo nossa própria função empacotadora, como mostrado na Figura 1.8.

lib/wrappthread.c

```
72 void
73 Pthread_mutex_lock(pthread_mutex_t *mptr)
74 {
75     int n;

76     if ( (n = pthread_mutex_lock(mptr)) == 0)
77         return;
78     errno = n;
79     err_sys("pthread_mutex_lock error");
80 }
```

lib/wrappthread.c

Figura 1.8 Nossa função empacotadora para `pthread_mutex_lock`.

Com uma codificação cuidadosa em C, poderíamos utilizar macros em vez de funções, fornecendo uma pequena eficiência de tempo de execução, mas essas funções empacotadoras raramente são o gargalo de desempenho de um programa.

Nossa escolha em escrever o primeiro caractere do nome de uma função com letra maiúscula está em obediência a um estilo. Muitos outros estilos foram considerados: prefixar o nome de função com um "e" (como feito na p. 182 de Kernighan e Pike [1984]), acrescentando "_e" ao nome de função e assim por diante. Nosso estilo parece ser o que menos distrai e, ao mesmo tempo, ainda fornece uma indicação visual de que alguma outra função está realmente sendo chamada.

Essa técnica tem também o benefício de verificar erros provenientes de funções cujos retornos de erros são freqüentemente ignorados: `close` e `listen`, por exemplo.

Daqui em diante, utilizaremos essas funções empacotadoras a menos que precisemos verificar e tratar um erro explícito de uma maneira diferente de terminar o processo. Não mostramos o código-fonte de todas as nossas funções empacotadoras, mas ele está livremente disponível (consulte o Prefácio).

Valor de `errno` do Unix

Quando um erro ocorre em uma função do Unix (como uma das funções de soquete), a variável global `errno` é configurada como um valor positivo que indica o tipo de erro e, normalmente, a função retorna –1. Nossa função `err_sys` verifica o valor de `errno` e imprime a string de mensagem de erro correspondente (por exemplo, "Connection timed out" ["Tempo limite da conexão expirado"] se `errno` for igual a `ETIMEDOUT`).

O valor de `errno` é configurado por uma função somente se um erro ocorrer. Seu valor é indefinido se a função não retornar um erro. Todos os valores de erro positivos são constantes com todos os nomes em letras maiúsculas iniciando com "E" e, normalmente, são definidos no cabeçalho `<sys/errno.h>`. Nenhum erro tem um valor de 0.

Armazenar `errno` em uma variável global não funciona com múltiplos threads que compartilham todas as variáveis globais. Discutiremos soluções para esse problema no Capítulo 26.

Por todo o livro, utilizaremos frases como "a função `connect` retorna `ECONNREFUSED`" como uma abreviação de que a função retorna um erro (em geral com um valor de retorno de –1), com `errno` configurado como a constante especificada.

1.5 Um servidor de data/hora simples

Podemos escrever uma versão simples de um servidor TCP de data/hora, que funcionará com o cliente na Seção 1.2. Para isso, utilizaremos as funções empacotadoras que descrevemos na seção anterior e mostraremos esse servidor na Figura 1.9.

Criação do soquete TCP

10 A criação do soquete TCP é idêntica à do código de cliente.

Vinculação de uma porta bem-conhecida do servidor ao soquete

11-15 A porta bem-conhecida do servidor (13 para o serviço de data/hora) é vinculada ao soquete preenchendo uma estrutura de endereços de soquetes de Internet e chamando `bind`. Especificamos o endereço IP como `INADDR_ANY`, que permite ao servidor aceitar uma conexão cliente em qualquer interface, se o host de servidor tiver múltiplas interfaces. Mais adiante, veremos como podemos restringir o servidor para aceitar uma conexão de cliente somente em uma única interface.

Conversão do soquete em soquete ouvinte

16 Chamando `listen`, o soquete é convertido em um soquete ouvinte, no qual as conexões entrantes de clientes serão aceitas pelo kernel. Estes três passos – `socket`, `bind` e `listen` –

são os passos normais de qualquer servidor TCP para preparar o que chamamos *descritor ouvinte* (nesse exemplo, `listenfd`).

A constante `LISTENQ` é do nosso cabeçalho `unp.h`. Ela especifica o número máximo de conexões cliente que o kernel irá enfileirar nesse descritor ouvinte. Discutiremos esse enfileiramento em mais detalhes na Seção 4.5.

intro/daytimetcpsrv.c

```
 1 #include "unp.h"
 2 #include <time.h>
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int listenfd, connfd;
 7     struct sockaddr_in servaddr;
 8     char buff[MAXLINE];
 9     time_t ticks;
10     listenfd = Socket(AF_INET, SOCK_STREAM, 0);
11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14     servaddr.sin_port = htons(13); /* servidor de data/hora */
15     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
16     Listen(listenfd, LISTENQ);
17     for ( ; ; ) {
18         connfd = Accept(listenfd, (SA *) NULL, NULL);
19         ticks = time(NULL);
20         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
21         Write(connfd, buff, strlen(buff));
22         Close(connfd);
23     }
24 }
```

intro/daytimetcpsrv.c

Figura 1.9 Servidor TCP de data/hora.

Aceitação da conexão cliente, envio de uma resposta

17-21 Normalmente, o processo servidor permanece em repouso na chamada a `accept`, esperando uma conexão cliente chegar e ser aceita. Uma conexão TCP utiliza o que é chamado de *handshake de três vias* para estabelecer uma conexão. Quando esse handshake é completado, `accept` retorna e o valor de retorno da função é um novo descritor (`connfd`) que é chamado *descritor conectado*. Esse novo descritor é utilizado na comunicação com o novo cliente. Um novo descritor é retornado por `accept` para cada cliente que se conecta ao nosso servidor.

> O estilo utilizado em todo o livro para um loop infinito é
>
> ```
> for ( ; ; ) {
>     . . .
> }
> ```

A data e a hora atual são retornadas pela função de biblioteca `time`, que retorna o número de segundos desde a Época Unix: 00:00:00 January 1, 1970, Coordinated Universal Time (UTC). A próxima função de biblioteca, `ctime`, converte esse valor de inteiro em uma string legível por humanos como

```
Mon May 26 20:58:40 2003
```

Um retorno de carro e uma quebra de linha são acrescentados à string por `snprintf`, e o resultado é gravado no cliente por `write`.

> Se você ainda não tiver o hábito de utilizar `snprintf` em vez de `sprintf` (mais antiga), agora é o momento de aprender. Chamadas a `sprintf` não podem verificar o estouro do buffer de destino. `snprintf`, por outro lado, requer que o segundo argumento tenha o tamanho do buffer de destino e esse buffer não irá estourar.
>
> `snprintf` foi uma adição relativamente tardia ao padrão ANSI C, introduzida na versão conhecida como *ISO C99*. Praticamente todos os fornecedores a distribuem como parte da biblioteca C padrão, e muitas versões gratuitas também estão disponíveis. Utilizamos `snprintf` por todo o texto e recomendamos sua utilização em vez de `sprintf` em todos os seus programas para melhorar a confiabilidade.
>
> É notável o número de invasões de rede por hackers que enviam dados para fazer com que o servidor chame `sprintf` e cause estouro de buffer. Outras funções que devemos ter cuidado são `gets`, `strcat` e `strcpy`, normalmente chamando `fgets`, `strncat` e `strncpy` em vez delas. Ainda melhores são as funções recentemente disponibilizadas `strlcat` e `strlpy`, que asseguram que o resultado seja uma string adequadamente terminada. Dicas adicionais sobre como escrever programas de rede seguros são oferecidas no Capítulo 23 de Garfinkel, Schwartz e Spafford (2003).

Término da conexão

22 O servidor fecha sua conexão com o cliente chamando `close`. Isso inicia a seqüência normal de término da conexão TCP: um FIN é enviado em cada direção e cada FIN é reconhecido pela outra extremidade. Discutiremos em mais detalhes o handshake de três vias do TCP e os quatro pacotes TCP utilizados para terminar uma conexão TCP na Seção 2.6.

Como ocorreu com o cliente na seção anterior, examinaremos esse servidor apenas brevemente, deixando todos os detalhes para mais adiante. Observe os seguintes pontos:

- Como ocorre com o cliente, o servidor é dependente do protocolo IPv4. Mostraremos uma versão independente de protocolo que utiliza a função `getaddrinfo` na Figura 11.13.
- Nosso servidor trata somente um cliente por vez. Se múltiplas conexões cliente chegarem aproximadamente ao mesmo tempo, o kernel irá enfileirá-las, até algum limite, e retorná-las a `accept` uma por vez. Esse servidor de data/hora, que requer chamar duas funções de biblioteca, `time` e `ctime`, é bem rápido. Mas, se o servidor levasse mais tempo para servir cada cliente (digamos alguns segundos ou um minuto), precisaríamos de alguma maneira sobrepor o serviço de um cliente a outro.

  O servidor que mostramos na Figura 1.9 é chamado *servidor iterativo* porque itera com cada cliente, um por vez. Há várias técnicas de escrever um servidor concorrente, um servidor que trata múltiplos clientes ao mesmo tempo. A mais simples para um servidor concorrente é chamar a função `fork` do Unix (Seção 4.7), criando um processo-filho para cada cliente. Outras são utilizar threads em vez de bifurcar (Seção 26.4) ou pré-bifurcar um número fixo de filhos quando o servidor inicia (Seção 30.6).
- Se iniciarmos um servidor como esse a partir de uma linha de comando de shell, é recomendável que o servidor execute durante muito tempo, uma vez que os servidores freqüentemente executam apenas enquanto o sistema estiver ativo. Isso requer a adição de código ao servidor para que ele execute corretamente como um *daemon* Unix*:* um processo que pode executar em segundo plano, não-vinculado a um terminal. Abordaremos esse processo na Seção 13.4.

1.6 Guia dos exemplos de cliente/servidor neste livro

Dois exemplos de cliente/servidor serão utilizados predominantemente por todo este livro para ilustrar as várias técnicas utilizadas na programação de rede:

- Um cliente/servidor de data/hora (que iniciamos nas Figuras 1.5, 1.6 e 1.9)
- Um cliente/servidor de eco (que iniciaremos no Capítulo 5)

Para fornecer um guia dos diferentes tópicos abrangidos neste texto, resumiremos os programas que serão desenvolvidos aqui e forneceremos o número da figura inicial e o número de página em que o código-fonte aparece. A Figura 1.10 lista as versões do cliente de data/hora, duas das quais já vimos. A Figura 1.11 lista as versões do servidor de data/hora. A Figura 1.12 lista as versões do cliente de eco e a Figura 1.13 lista as versões do servidor de eco.

Figura	Página	Descrição
1.5	30	TCP/IPv4, dependente de protocolo
1.6	33	TCP/IPv6, dependente de protocolo
11.4	298	TCP/IPv4, dependente de protocolo, chama `gethostbyname` e obtém `servbyname`
11.11	311	TCP, independente de protocolo, chama `getaddrinfo` e `tcp_connect`
11.16	317	UDP, independente de protocolo, chama `getaddrinfo` e `udp_client`
16.11	418	TCP, utiliza `connect` não-bloqueadora
31.8	782	TCP, dependente de protocolo, utiliza TPI em vez de soquetes
E.1	832	TCP, dependente de protocolo, gera `SIGPIPE`
E.5	835	TCP, dependente de protocolo, imprime o tamanho do buffer de recebimento do soquete e MSS
E.11	844	TCP, dependente de protocolo, permite hostname (`gethostbyname`) ou endereço IP
E.12	846	TCP, independente de protocolo, permite hostname (`gethostbyname`)

Figura 1.10 Diferentes versões do cliente de data/hora apresentadas neste livro.

Figura	Página	Descrição
1.9	33	TCP/IPv4, dependente de protocolo
11.13	314	TCP, independente de protocolo, chama `getaddrinfo` e `tcp_listen`
11.14	315	TCP, independente de protocolo, chama `getaddrinfo` e `tcp_listen`
11.19	320	UDP, independente de protocolo, chama `getaddrinfo` e `udp_server`
13.5	350	TCP, independente de protocolo, é executado como daemon independente
13.12	356	TCP, independente de protocolo, gerado a partir do daemon `inetd`

Figura 1.11 Diferentes versões do servidor de data/hora apresentadas neste livro.

1.7 Modelo OSI

Uma maneira comum de descrever as camadas em uma rede é utilizar o *modelo Open Systems Interconnection (OSI)* da International Organization for Standardization (ISO) para comunicações de computador. Esse é um modelo de sete camadas, que mostramos na Figura 1.14, junto com o mapeamento aproximado para o conjunto de protocolos Internet.

Consideramos as duas camadas inferiores do modelo OSI como o driver de dispositivo e o hardware de rede fornecidos com o sistema. Normalmente, não precisamos nos preocupar com essas camadas além de estarmos cientes de que algumas propriedades do enlace de dados, como a unidade de transferência máxima (*maximum transfer unit* – MTU) de 1.500 bytes da Ethernet, que descreveremos na Seção 2.11.

A camada de rede é tratada pelos protocolos IPv4 e IPv6, descritos no Apêndice A. As camadas de transporte que podemos escolher são TCP e UDP que descreveremos no Capítulo 2.

Figura	Página	Descrição
5.4	130	TCP/IPv4, dependente de protocolo
6.9	167	TCP, utiliza `select`
6.13	172	TCP, utiliza `select` e opera nos buffers
8.7	238	UDP/IPv4, dependente de protocolo
8.9	240	UDP, verifica o endereço do servidor
8.17	248	UDP chama `connect` para obter erros assíncronos
14.2	361	UDP, expira ao ler a resposta do servidor utilizando `SIGALRM`
14.4	363	UDP, expira ao ler a resposta do servidor utilizando `select`
14.5	364	UDP, expira ao ler a resposta do servidor utilizando `SO_RCVTIMEO`
15.4	391	Fluxo de domínio Unix, dependente de protocolo
15.6	392	Datagrama de domínio Unix, dependente de protocolo
16.3	408	TCP utiliza E/S sem bloqueio
16.10	415	TCP, utiliza dois processos (`fork`)
16.21	428	TCP, estabelece a conexão e então envia RST
14.15	378	TCP, utiliza `/dev/poll` para multiplexação
14.18	381	TCP, utiliza `kqueue` para multiplexação
20.5	498	UDP, transmite por broadcast com condição de corrida (*race condition*)
20.6	500	UDP, transmite por broadcast com condição de corrida (*race condition*)
20.7	501	UDP, transmite por broadcast, condição de corrida fixada utilizando `pselect`
20.9	503	UDP, transmite por broadcast, condição de corrida corrigida utilizando `sigsetjmp` e `siglongjm`
20.10	505	UDP, transmite por broadcast, condição de corrida corrigida utilizando IPC a partir do handler de sinal
22.6	554	UDP, confiável, utilizando tempo-limite, retransmissão e número de seqüência
26.2	625	TCP, utiliza dois threads
27.6	657	TCP/IPv4, especifica uma rota de origem
27.13	668	UDP/IPv6, especifica uma rota de origem

Figura 1.12 Diferentes versões do cliente de eco apresentadas neste livro.

Figura	Página	Descrição
5.2	123	TCP/IPv4, dependente de protocolo
5.12	139	TCP/IPv4, dependente de protocolo, colhe filhos terminados
6.21	178	TCP/IPv4, dependente de protocolo, utiliza `select`, um processo trata todos os clientes
6.25	186	TCP/IPv4, dependente de protocolo, utiliza `poll`, um processo trata todos os clientes
8.3	242	UDP/IPv4, dependente de protocolo
8.24	263	TCP e UDP/IPv4, dependente de protocolo, utiliza `select`
14.14	400	TCP, utiliza a biblioteca-padrão de E/S
15.3	417	Fluxo de domínio Unix, dependente de protocolo
15.5	418	Datagrama de domínio Unix, dependente de protocolo
15.15	431	Fluxo de domínio Unix, com passagem de credencial a partir do cliente
22.4	593	UDP, recebe o endereço de destino e a interface recebida; datagramas truncados
22.15	609	UDP, vincula todos os endereços de interface
25.4	668	UDP, utiliza E/S baseada em sinal
26.3	682	TCP, um thread por cliente
26.4	684	TCP, um thread por cliente, passagem de argumento portável
27.6	716	TCP/IPv4, imprime a rota de origem recebida
27.14	730	UDP/IPv6, imprime e inverte a rota de origem recebida
28.31	773	UDP, utiliza `icmpd` para receber erros assíncronos
E.15	943	UDP, vincula todos os endereços de interface

Figura 1.13 Diferentes versões do servidor de eco apresentadas neste livro.

Mostramos uma lacuna entre o TCP e o UDP na Figura 1.14 para indicar que é possível que uma aplicação pule a camada de transporte e utilize o IPv4 ou o IPv6 diretamente. Isso é chamado *soquete bruto*, que discutiremos no Capítulo 28.

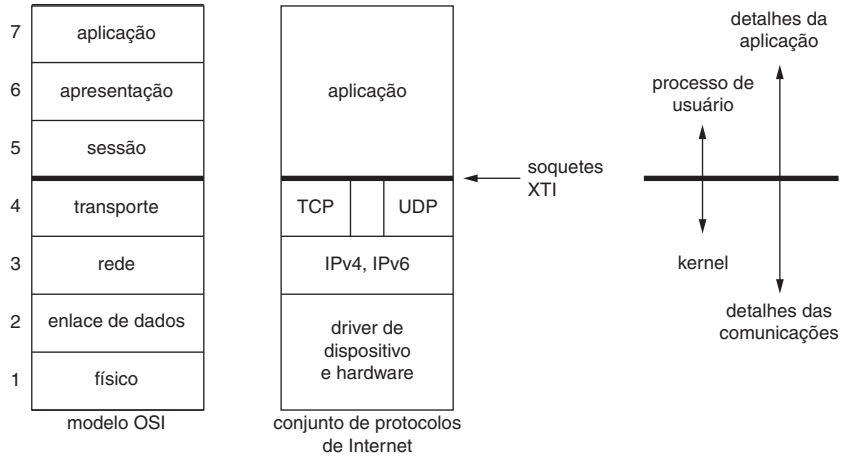


Figura 1.14 As camadas no modelo OSI e o conjunto de protocolos de Internet.

As três camadas superiores do modelo OSI são combinadas em uma única camada chamada aplicação. Esta é representada pelo cliente Web (navegador), cliente Telnet, servidor Web, servidor FTP ou qualquer que seja a aplicação que estejamos utilizando. Com os protocolos de Internet, raramente há alguma distinção entre as três camadas superiores do modelo OSI.

As interfaces de programação de soquetes descritas neste livro são provenientes das três camadas superiores (a "aplicação") para a camada de transporte. Esse é o nosso foco: como escrever aplicações com soquetes que utilizam TCP ou UDP. Já mencionamos soquetes brutos e, no Capítulo 29, veremos que podemos até mesmo pular completamente a camada de IP para ler e gravar nossos próprios quadros de camada de enlace de dados.

Por que os soquetes fornecem a interface proveniente das três camadas superiores do modelo OSI para a camada de transporte? Há duas razões para esse *design*, que observamos no lado direito da Figura 1.14. A primeira delas é que as três camadas superiores tratam todos os detalhes da aplicação (por exemplo, FTP, Telnet ou HTTP) e conhecem pouco sobre os detalhes da comunicação. As quatro camadas inferiores conhecem pouco sobre a aplicação, mas tratam todos os detalhes da comunicação: enviar dados, esperar reconhecimentos, colocar em seqüência os dados que chegam fora de ordem, calcular e confirmar somas de verificação e assim por diante. A segunda razão é que as três camadas superiores freqüentemente formam o que é chamado *processo de usuário*, enquanto as quatro camadas inferiores normalmente são fornecidas como parte do kernel do sistema operacional (SO). O Unix fornece essa separação entre o processo de usuário e o kernel, como ocorre com muitos outros sistemas operacionais contemporâneos. Portanto, a interface entre as camadas 4 e 5 é o lugar natural para construir a API.

1.8 História da rede BSD

A API de soquetes originou-se com o sistema 4.2BSD, distribuído em 1983. A Figura 1.15 mostra o desenvolvimento das várias distribuições do BSD, mencionando os desenvolvimentos de TCP/IP mais importantes. Algumas alterações na API de soquetes também aconteceram em 1990 com a distribuição do 4.3BSD Reno, quando os protocolos OSI foram incorporados ao kernel do BSD.

O caminho na figura do 4.2BSD ao 4.4BSD mostra as distribuições do Computer Systems Research Group (CSRG) em Berkeley, que exigiam que o destinatário tivesse uma licença do código-fonte para o Unix. Mas todo o código de rede, tanto o suporte de kernel (por

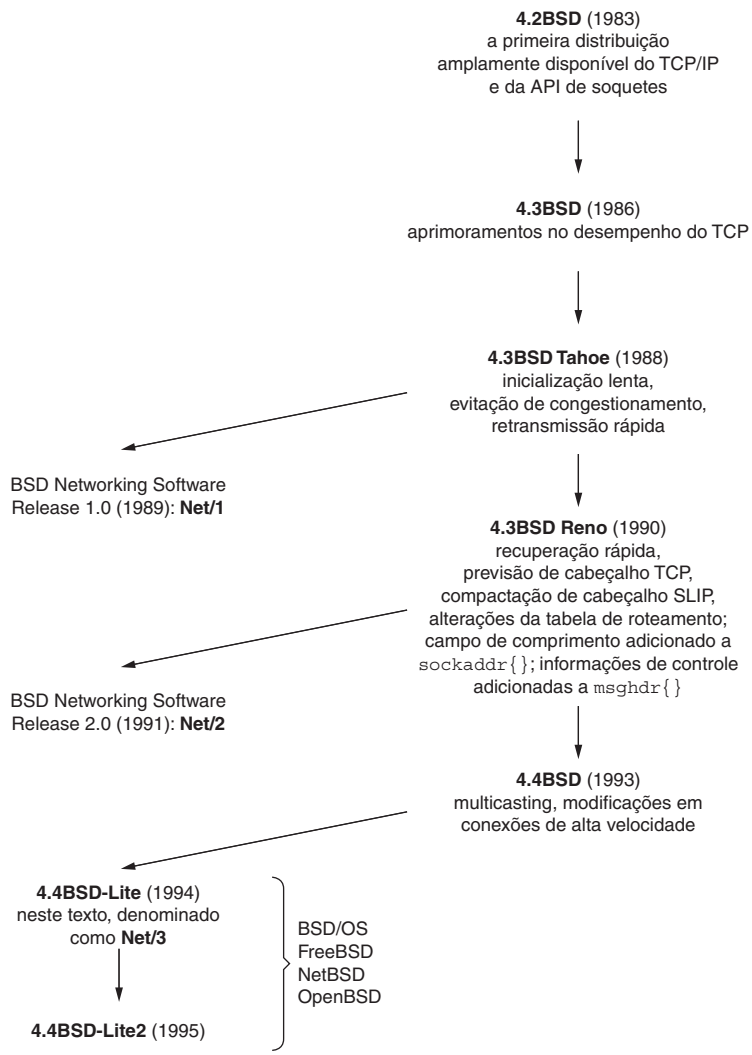


Figura 1.15 História das várias distribuições BSD.

exemplo, o TCP/IP e as pilhas dos protocolos de domínio Unix e a interface de soquete) como as aplicações (por exemplo, clientes e servidores Telnet e FTP), foi desenvolvido de maneira independente do código Unix derivado da AT&T. Portanto, começando em 1989, Berkeley forneceu a primeira das distribuições de rede BSD, que continha todo o código de rede e vários outras partes do sistema BSD que não foram restringidas pelo requisito de licença de código-fonte do Unix. Essas distribuições estavam "publicamente disponíveis" e por fim tornaram-se disponíveis via FTP anônimo para qualquer pessoa.

As distribuições finais do Berkeley foram o 4.4BSD-Lite, em 1994, e o 4.4BSD-Lite2, em 1995. Observamos que essas duas distribuições foram então utilizadas como a base para outros sistemas: BSD/OS, FreeBSD, NetBSD e OpenBSD, a maioria dos quais ainda está sendo ativamente desenvolvida e aprimorada. Informações adicionais sobre as várias distribuições do BSD, e sobre a história dos vários sistemas Unix em geral, podem ser encontradas no Capítulo 1 de McKusick *et al.* (1996).

Muitos sistemas Unix iniciaram com alguma versão do código de rede BSD, incluindo a API de soquetes, e referimo-nos a essas implementações como *implementações derivadas do Berkeley*. Várias versões comerciais do Unix são baseadas no System V Release 4 (SVR4). Algumas delas têm o código de rede derivado do Berkeley (por exemplo, o UnixWare 2.x), enquanto o código de rede em outros sistemas SVR4 foram derivados independentemente (por exemplo, o Solaris 2.x). Também observamos que o Linux, uma implementação popular e livremente disponível do Unix, *não* entra na classificação de derivado do Berkeley: o código de rede e a API de soquetes do Linux foram desenvolvidos a partir do zero.

1.9 Redes e hosts de teste

A Figura 1.16 mostra as várias redes e hosts utilizados nos exemplos deste livro. Para cada host, mostramos o SO e o tipo de hardware (uma vez que alguns sistemas operacionais são executados em mais de um tipo de hardware). O nome dentro de cada caixa é o hostname que aparece no texto.

A topologia mostrada na Figura 1.16 é interessante em consideração aos nossos exemplos, mas as máquinas estão amplamente espalhadas pela Internet e na prática a topologia física torna-se menos interessante. Em vez disso, conexões de redes privadas virtuais (Virtual Private Networks – VPNs) ou conexões de shell seguro (Secure Shell – SSH) fornecem a conectividade entre essas máquinas independentemente de onde elas residam fisicamente.

A notação "/24" indica o número de bits consecutivos iniciando do bit mais à esquerda do endereço utilizado para identificar a rede e a sub-rede. A Seção A.4 discutirá a notação */n* utilizada atualmente para especificar os limites de sub-rede.

> O nome real do SO Sun é SunOS 5.x e não Solaris 2.x, mas todos referem-se a ele como Solaris, o nome dado à soma do SO e outro software empacotado com o SO de base.

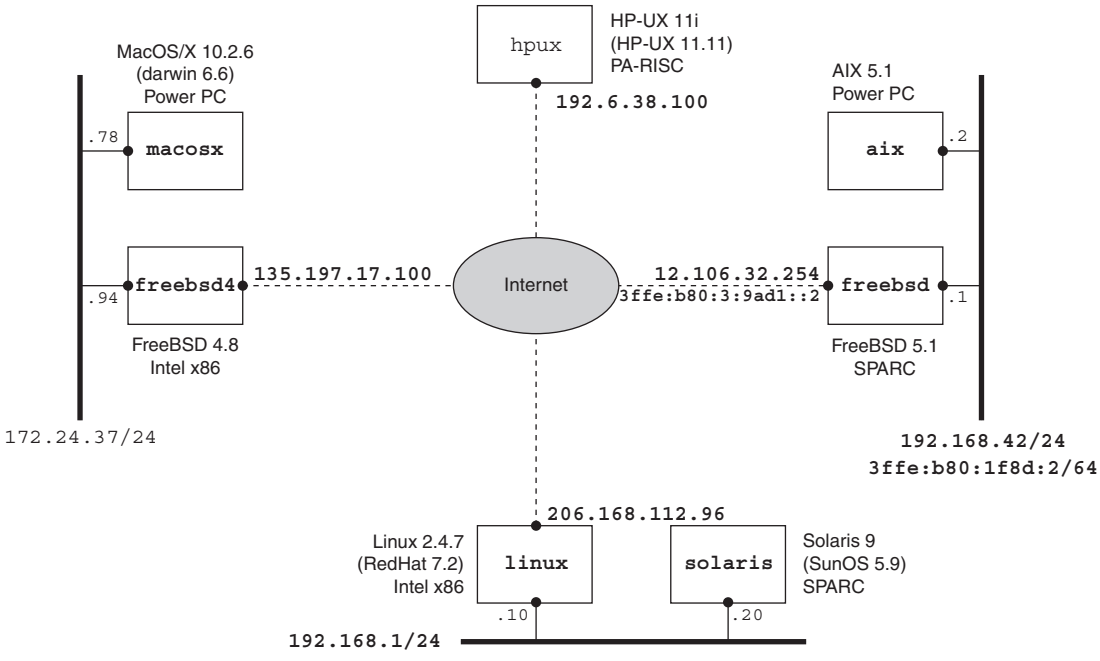


Figura 1.16 As redes e hosts utilizados na maioria dos exemplos deste livro.

Descobrindo a topologia de rede

Mostramos a topologia de rede na Figura 1.16 para os hosts utilizados nos exemplos deste livro, mas talvez você precise conhecer sua própria topologia de rede para executar os exemplos e exercícios na sua própria rede. Embora não haja nenhum padrão atual do Unix com relação à configuração e administração de rede, há dois comandos básicos fornecidos pela maioria dos sistemas Unix que podem ser utilizados para descobrir alguns detalhes sobre uma rede: `netstat` e `ifconfig`. Verifique esses comandos nas páginas do manual (man) do seu sistema para ver os detalhes sobre as informações que são geradas. Também esteja ciente de que alguns fornecedores colocam esses comandos em um diretório administrativo, como `/sbin` ou `/usr/sbin`, em vez do `/usr/bin` normal e esses diretórios talvez não estejam no caminho normal de pesquisa do shell (`PATH`).

1. `netstat -i` fornece informações sobre as interfaces. Também especificamos o flag `-n` para imprimir endereços numéricos, em vez de tentar encontrar os nomes das redes. Isso nos mostra as interfaces e seus nomes.

```
linux % netstat -ni
Kernel Interface table
Iface   MTU Met    RX-OK RX-ERR RX-DRP RX-OVR   TX-OK TX-ERR TX-DRP TX-OVR Flg
eth0   1500   0 49211085      0      0      0 40540958      0      0      0 BMRU
lo    16436   0 98613572      0      0      0 98613572      0      0      0 LRU
```

   A interface de loopback é chamada `lo` e a Ethernet é chamada `eth0`. O próximo exemplo mostra um host com suporte IPv6.

```
freebsd % netstat -ni
Name  Mtu Network       Address              Ipkts Ierrs     Opkts Oerrs Coll
hme0 1500 <Link#1>      08:00:20:a7:68:6b 29100435    35 46561488     0    0
hme0 1500 12.106.32/24  12.106.32.254     28746630     - 46617260     -    -
hme0 1500 fe80:1::a00:20ff:fea7:686b/64
                        fe80:1::a00:20ff:fea7:686b
                                              0     -         0     -    -
hme0 1500 3ffe:b80:1f8d:1::1/64
                        3ffe:b80:1f8d:1::1        0     -         0     -    -
hme1 1500 <Link#2>      08:00:20:a7:68:6b    51092     0     31537     0    0
hme1 1500 fe80:2::a00:20ff:fea7:686b/64
                        fe80:2::a00:20ff:fea7:686b
                                              0     -        90     -    -
hme1 1500 192.168.42    192.168.42.1         43584     -     24173     -    -
hme1 1500 3ffe:b80:1f8d:2::1/64
                        3ffe:b80:1f8d:2::1       78     -         8     -    -
lo0 16384 <Link#6>                           10198     0     10198     0    0
lo0 16384 ::1/128       ::1                     10     -        10     -    -
lo0 16384 fe80:6::1/64  fe80:6::1                0     -         0     -    -
lo0 16384 127           127.0.0.1            10167     -     10167     -    -
gif0 1280 <Link#8>                               6     0         5     0    0
gif0 1280 3ffe:b80:3:9ad1::2/128
                        3ffe:b80:3:9ad1::2       0     -         0     -    -
gif0 1280 fe80:8::a00:20ff:fea7:686b/64
                        fe80:8::a00:20ff:fea7:686b
                                              0     -         0     -    -
```

Nota: Quebramos algumas linhas mais longas para alinhar os campos de saída.

2. `netstat -r` mostra a tabela de roteamento, que é uma outra maneira de determinar as interfaces. Normalmente, especificamos o flag `-n` para imprimir endereços numéricos. Isso também mostra o endereço IP do roteador-padrão.

```
freebsd % netstat -nr
Routing tables
```

```
Internet:
Destination      Gateway              Flags    Refs     Use   Netif  Expire
default          12.106.32.1          UGSc       10    6877    hme0
12.106.32/24     link#1               UC          3       0    hme0
12.106.32.1      00:b0:8e:92:2c:00    UHLW        9       7    hme0    1187
12.106.32.253    08:00:20:b8:f7:e0    UHLW        0       1    hme0     140
12.106.32.254    08:00:20:a7:68:6b    UHLW        0       2     lo0
127.0.0.1        127.0.0.1            UH          1   10167     lo0
192.168.42       link#2               UC          2       0    hme1
192.168.42.1     08:00:20:a7:68:6b    UHLW        0      11     lo0
192.168.42.2     00:04:ac:17:bf:38    UHLW        2   24108    hme1     210

Internet6:
Destination                           Gateway              Flags  Netif  Expire
::/96                                 ::1                  UGRSc    lo0 =>
default   3ffe:b80:3:9ad1::1          UGSc                 gif0
::1                                   ::1                  UH       lo0
::ffff:0.0.0.0/96                     ::1                  UGRSc    lo0
3ffe:b80:3:9ad1::1                    3ffe:b80:3:9ad1::2   UH      gif0
3ffe:b80:3:9ad1::2                    link#8               UHL      lo0
3ffe:b80:1f8d::/48                    lo0                  USC      lo0
3ffe:b80:1f8d:1::/64                  link#1               UC      hme0
3ffe:b80:1f8d:1::1                    08:00:20:a7:68:6b    UHL      lo0
3ffe:b80:1f8d:2::/64                  link#2               UC      hme1
3ffe:b80:1f8d:2::1                    08:00:20:a7:68:6b    UHL      lo0
3ffe:b80:1f8d:2:204:acff:fe17:bf38    00:04:ac:17:bf:38    UHLW    hme1
fe80::/10                             ::1                  UGRSc    lo0
fe80::%hme0/64                        link#1               UC      hme0
fe80::a00:20ff:fea7:686b%hme0         08:00:20:a7:68:6b    UHL      lo0
fe80::%hme1/64                        link#2               UC      hme1
fe80::a00:20ff:fea7:686b%hme1         08:00:20:a7:68:6b    UHL      lo0
fe8o::%lo0/64                         fe80::1%lo0          Uc       lo0
fe8o::1%lo0                           link#6               UHL      lo0
fe80::%gif0/64                        link#8               UC      gif0
fe80::a00:20ff:fea7:686b%gif0         link#8               UHL      lo0
ff01::/32                             ::1                  U        lo0
ff02::/16                             ::1                  UGRS     lo0
ff02::%hme0/32                        link#1               UC      hme0
ff02::%hme1/32                        link#2               UC      hme1
ff02::%lo0/32                         ::1                  UC       lo0
ff02::%gif0/32                        link#8               UC      gif0
```

3. Dados os nomes de interface, executamos `ifconfig` para obter os detalhes de cada interface.

```
linux % ifconfig eth0
eth0      Link encap:Ethernet HWaddr 00:C0:9F:06:B0:E1
          inet addr:206.168.112.96 Bcast:206.168.112.127 Mask:255.255.255.128
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:49214397 errors:0 dropped:0 overruns:0 frame:0
          TX packets:40543799 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:1098069974 (1047.2 Mb) TX bytes:3360546472 (3204.8 Mb)
          Interrupt:11 Base address:0x6000
```

Isso mostra o endereço IP, a máscara de sub-rede e o endereço de broadcast. O flag `MULTICAST` é freqüentemente uma indicação de que o host suporta multicast. Algumas implementações fornecem um flag `-a`, que imprime informações sobre todas as interfaces configuradas.

4. Uma maneira de localizar o endereço IP de vários hosts na rede local é emitir um `ping` para o endereço de broadcast (que encontramos no passo anterior).

```
linux % ping -b 206.168.112.127
WARNING: pinging broadcast address
PING 206.168.112.127 (206.168.112.127) from 206.168.112.96 : 56(84) bytes of data.
64 bytes from 206.168.112.96: icmp_seq=0 ttl=255 time-241 usec
64 bytes from 206.168.112.40: icmp_seq=0 ttl=255 time=2.566 msec (DUP!)
64 bytes from 206.168.112.118: icmp_seq=0 ttl=255 time=2.973 msec (DUP!)
64 bytes from 206.168.112.14: icmp_seq=0 ttl=255 time=3.089 msec (DUP!)
64 bytes from 206.168.112.126: icmp_seq=0 ttl=255 time=3.200 msec (DUP!)
64 bytes from 206.168.112.71: icmp_seq=0 ttl=255 time=3.311 msec (DUP!)
64 bytes from 206.168.112.31: icmp_seq=0 ttl=64 time=3.541 msec (DUP!)
64 bytes from 206.168.112.7: icmp_seq=0 ttl=255 time=3.636 msec (DUP!)
...
```

1.10 Padrões Unix

No momento em que escrevíamos este livro, a atividade mais interessante de padronização do Unix estava sendo feita pelo The Austin Common Standards Revision Group (CSRG). Os esforços desse grupo produziram, *grosso modo*, 4 mil páginas de especificações abrangendo mais de 1.700 interfaces de programação (Josey, 2002). Essas especificações utilizam tanto a designação do IEEE POSIX como a designação Technical Standard do Open Group. O resultado final é que você provavelmente encontrará referências ao mesmo padrão com vários nomes: ISO/IEC 9945:2002, IEEE Std 1003.1-2001 e a Single Unix Specification Version 3, por exemplo. Aqui, iremos nos referir a esse padrão simplesmente como *Especificação POSIX*, exceto em seções, como esta, em que discutimos as especificidades dos vários padrões mais antigos.

A maneira mais fácil de adquirir uma cópia desse padrão consolidado é solicitá-la em CD-ROM ou acessá-la via a Web (gratuita). O ponto de partida para qualquer um desses métodos é

```
http://www.UNIX.org/version3
```

História do POSIX

POSIX é um acrônimo para Portable Operating System Interface. POSIX não é um padrão único, mas uma família de padrões sendo desenvolvida pelo Institute for Electrical and Electronics Engineers, Inc., normalmente denominado *IEEE*. Os padrões POSIX também foram adotados como padrões internacionais pela ISO e pela International Electrotechnical Commission (IEC), chamada ISO/IEC. Os padrões POSIX têm uma história interessante, que abordaremos apenas brevemente:

- O IEEE Std 1003.1-1988 (317 páginas) foi o primeiro padrão POSIX. Esse padrão especificou a interface da linguagem C em um kernel do tipo Unix e abrangia as seguintes áreas: processos primitivos (`fork`, `exec`, sinais e temporizadores), o ambiente de um processo (IDs de usuário e grupos de processos), arquivos e diretórios (todas as funções de E/S), E/S terminal, bancos de dados de sistema (arquivo de senha e arquivo de grupo) e os formatos de arquivos `tar` e `cpio`.

  > O primeiro padrão POSIX era, em 1986, uma versão para uso e avaliação conhecida como "IEEE-IX". O nome "POSIX" foi sugerido por Richard Stallman.

- O IEEE Std 1003.1-1990 (356 páginas) veio em seguida e também era conhecido como ISO/IEC 9945-1: 1990. Alterações mínimas foram feitas da versão de 1988 para a de 1990. Foi acrescentado ao título "Part 1: System Application Program Interface (API) [C Language]", indicando que esse padrão era a API da linguagem C.
- O IEEE Std 1003.2-1992 veio em seguida em dois volumes (aproximadamente 1.300 páginas). Seu título continha "Part 2: Shell and Utilities". Essa parte definiu o shell (baseado no shell do System V Bourne) e cerca de 100 utilitários (normalmente, programas executados a partir de um shell, desde `awk` e `basename` até `vi` e `yacc`). Por todo este livro, iremos nos referir a esse padrão como *POSIX.2*.

- O IEEE Std 1003.1b-1993 (590 páginas) era originalmente conhecido como IEEE P1003.4. Esse padrão foi uma atualização do padrão 1003.1-1990 para incluir as extensões de tempo real desenvolvidas pelo grupo de trabalho P1003.4. O padrão 1003.lb-1993 adicionou os seguintes itens ao padrão 1990: sincronização de arquivos, E/S assíncrona, semáforos, gerenciamento de memória (`mmap` e memória compartilhada), agendamento de execução, relógios e temporizadores e filas de mensagens.
- O IEEE Std 1003.1, 1996 Edition [IEEE 1996] (743 páginas) veio depois e incluía o 1003.1-1990 (a API básica), o 1003.1b-1993 (extensões de tempo real), o 1003.1c-1995 (`pthreads`) e o 1003.1i-1995 (correções técnicas ao 1003.1b). Esse padrão também foi chamado ISO/IEC 9945-1: 1996. Três capítulos sobre threads foram adicionados, junto com seções extras sobre sincronização de threads (mutexes e variáveis de condição), agendamento de threads e agendamento de sincronização. Por todo o livro, iremos nos referir a esse padrão como *POSIX.1*. Esse padrão também contém uma Apresentação declarando que o ISO/IEC 9945 consiste nas seguintes partes:

  - Parte 1: API de sistema (linguagem C)
  - Parte 2: Shell e utilitários
  - Parte 3: Administração de sistema (em desenvolvimento)

  As Partes 1 e 2 são o que chamamos POSIX.1 e POSIX.2.

  > Mais de um quarto das 743 páginas são um apêndice intitulado "Rationale and Notes", o qual contém as informações históricas e as razões pelos quais certos recursos foram incluídos ou omitidos. Freqüentemente, o raciocínio é tão informativo quanto o padrão oficial.

- IEEE Std 1003.1g: Interfaces independentes de protocolo (Protocol-independent interfaces – PII) tornaram-se, em 2000, um padrão aprovado. Até a introdução da Single Unix Specification Version 3, esse trabalho do POSIX era o tópico mais relevante abrangido neste livro. Esse é o padrão de API de interligação de redes e define duas APIs, que ele denomina Detailed Network Interfaces (DNIs) :

  1. DNI/Socket, baseada na API de soquetes do 4.4BSD
  2. DNI/XTI, baseada na especificação X/Open XPG4

  O trabalho nesse padrão iniciou no final da década de 1980 como o grupo de trabalho P1003.12 (posteriormente renomeado P1003.1g). Por todo o livro, iremos nos referir a esse padrão como *POSIX.1g*.

  O status atual dos vários padrões POSIX está disponível em

  ```
  http://www.pasc.org/standing/sd11.html
  ```

História do Open Group

O Open Group foi formado em 1996 pela consolidação da X/Open Company (fundada em 1984) e da Open Software Foundation (OSF, fundada em 1988). Ele é um consórcio internacional de fornecedores e clientes usuários finais dos setores industrial, governamental e acadêmico. Eis um breve relato dos padrões que esse consórcio produziu:

- O X/Open publicou o *X/Open Portability Guide*, número 3 (XPG3), em 1989.
- O número 4 foi publicado em 1992, seguido pelo número 4, versão 2, em 1994. Essa última versão também era conhecida como "Spec 1170", com o número mágico 1.170 sendo a soma do número de interfaces de sistema (926), do número de cabeçalhos (70) e do número de comandos (174). O nome mais recente para esse conjunto de especificações é "X/Open Single Unix Specification", embora também seja chamado "Unix 95".

- Em março de 1997, foi anunciada a versão 2 da Single Unix Specification. Os produtos que obedeciam a essa especificação foram chamados "Unix 98". Iremos nos referir a essa especificação simplesmente como "Unix 98" por todo o texto. O número de interfaces requerido pelo Unix 98 aumenta de 1.170 a 1.434, embora para uma estação de trabalho isso pule para 3.030, porque inclui o Common Desktop Environment (CDE) que, por sua vez, requer o X Window System e a interface com o usuário Motif. Os detalhes estão disponíveis em Josey (1997) e em `http://www.UNIX.org/version2`. Os serviços de rede que fazem parte do Unix 98 são definidos tanto para os soquetes como APIs XTI. Essa especificação é quase idêntica ao POSIX.1g.

  Infelizmente, o Unix 98 referia-se aos padrões de rede como XNS: Serviços de Rede do X/Open. A versão desse documento que define soquetes e XTI para o Unix 98 (Open Group, 1997) é chamada "XNS Issue 5". Na rede mundial, o XNS sempre foi uma abreviação para a arquitetura do Xerox Network System. Evitaremos essa utilização de XNS e iremos nos referir a esse documento do X/Open simplesmente como o padrão de API de rede do Unix 98 (Unix 98 network API standard).

Unificação de padrões

As breves histórias do POSIX e do Open Group continuam com a publicação do Austin Group da Single Unix Specification Version 3, como mencionado no começo desta seção. Fazer com que mais de 50 empresas concordassem sobre um único padrão certamente é um marco na história do Unix. Atualmente, a maioria dos sistemas Unix obedece a alguma versão do POSIX.1 ou do POSIX.2; muitos obedecem à Single Unix Specification Version 3.

Historicamente, a maioria dos sistemas Unix tem uma herança do Berkeley ou do System V, mas essas diferenças estão lentamente desaparecendo à medida que os fornecedores adotam os padrões. As principais diferenças ainda existentes têm a ver com administração de sistema, uma área que nenhum padrão atualmente resolve.

O foco deste livro é a Single Unix Specification Version 3 e, sobretudo, a API de soquetes. Sempre que possível, utilizaremos as funções-padrão.

Internet Engineering Task Force (IETF)

O Internet Engineering Task Force (IETF) é uma grande comunidade aberta e internacional de designers de rede, operadores, fornecedores e pesquisadores preocupados com a evolução da arquitetura da Internet e o bom funcionamento da Web. Ele está aberto a qualquer pessoa interessada.

Os processos-padrão de Internet estão documentados na RFC 2026 (Bradner, 1996). Os padrões Internet normalmente lidam com questões de protocolo e não com APIs de programação. Contudo, duas RFCs (RFC 3493 [Gilligan *et al.,* 2003] e RFC 3542 [Stevens *et al.,* 2003]) especificam a API de soquetes para o IPv6. Essas RFCs são RFCs informativas, não padrões, e foram produzidas para acelerar a instalação de aplicações portáveis pelos numerosos fornecedores que trabalham com as distribuições prévias do IPv6. Embora corpos de padrões tendam a levar muito tempo, muitas APIs foram padronizadas na Single Unix Specification Version 3.

1.11 Arquiteturas de 64 bits

Dos meados ao final da década de 1990, iniciou a tendência das arquiteturas e dos softwares de 64 bits. Uma das razões é o maior endereçamento dentro de um processo (isto é, ponteiros de 64 bits), para lidar com grandes quantidades de memória (mais de 2^{32} bytes). O modelo de programação comum para sistemas Unix de 32 bits existentes é chamado modelo *ILP32*, que denota que inteiros (I), inteiros longos (L) e ponteiros (P) ocupam 32 bits. O modelo que es-

tá se tornando predominante para sistemas Unix de 64 bits é chamado modelo *LP64*, significando que somente os inteiros longos (L) e os ponteiros (P) requerem 64 bits. A Figura 1.17 compara esses dois modelos.

Da perspectiva da programação, o modelo LP64 significa que não podemos supor que um ponteiro pode ser armazenado em um inteiro. Também devemos considerar o efeito do modelo LP64 sobre as APIs existentes.

O ANSI C inventou o tipo de dados `size_t`, utilizado, por exemplo, como o argumento para `malloc` (o número de bytes a alocar) e como o terceiro argumento para `read` e `write` (o número de bytes a ler ou gravar). Em um sistema de 32 bits, `size_t` tem um valor de 32 bits, mas, em um sistema de 64 bits, ele deve ter um valor de 64 bits, para tirar proveito do modelo de endereçamento maior. Isso significa que um sistema de 64 bits provavelmente conterá um `typedef` de `size_t` como um `unsigned long`. O problema da API de rede é que algumas propostas (*drafts*) do POSIX.1g especificaram que os argumentos de função contendo o tamanho das estruturas de um endereço de soquete têm o tipo de dados `size_t` (por exemplo, o terceiro argumento para `bind` e `connect`). Algumas estruturas XTI também tinham membros com um tipo de dados de `long` (por exemplo, as estruturas `t_info` e `t_opthdr`). Se essas estruturas tivessem sido deixadas como são, as duas mudariam de valores de 32 bits para valores de 64 bits quando o sistema Unix mudasse do modelo ILP32 para o LP64. Em ambos os casos, não há necessidade de um tipo de dados de 64 bits: o comprimento de uma estrutura de endereço de soquete é de no máximo algumas centenas de bytes, e o uso de `long` para os membros da estrutura XTI foi um equívoco.

A solução é utilizar tipos de dados projetados especificamente para tratar esses cenários. A API de soquetes utiliza o tipo de dados `socklen_t` para comprimentos das estruturas de endereço de soquete; e o XTI utiliza tipos de dados `t_scalar_t` e `t_uscalar_t`. A razão para não mudar esses valores de 32 bits para 64 bits é tornar mais fácil fornecer a compatibilidade binária no novo sistema de 64 bits para aplicações compiladas sob os sistemas de 32 bits.

Tipo de dados	Modelo ILP32	Modelo LP64
`char`	8	8
`short`	16	16
`int`	32	32
`long`	32	64
`pointer`	32	64

Figura 1.17 Comparação do número de bits para armazenar vários tipos de dados para os modelos ILP32 e LP64.

1.12 Resumo

A Figura 1.5 mostra um cliente TCP completo, embora simples, que busca a data e a hora atual a partir de um servidor especificado; e a Figura 1.9 mostra uma versão completa do servidor. Esses dois exemplos introduzem muitos dos termos e conceitos que serão amplamente expandidos no restante do livro.

Nosso cliente era dependente de protocolo no IPv4 e, em vez disso, o modificamos para utilizar o IPv6. Mas isso apenas nos forneceu um outro programa dependente de protocolo. No Capítulo 11, desenvolveremos algumas funções que nos permitirão escrever um código independente de protocolo, o que será importante à medida que a Internet começar a utilizar mais o IPv6.

Por todo o livro, utilizaremos as funções empacotadoras desenvolvidas na Seção 1.4 para reduzir o tamanho do nosso código e, mesmo assim, verificar um retorno de erro em cada chamada de função. Todas as nossas funções empacotadoras iniciam com uma letra maiúscula.

A Single Unix Specification Version 3, conhecida por vários outros nomes e, neste livro, simplesmente chamada *Especificação POSIX*, é a confluência de dois longos esforços de padrões, finalmente agrupados pelo Austin Group.

Os leitores interessados na história da rede Unix devem consultar Salus (1994) sobre uma descrição da história do Unix e Salus (1995) sobre a história do TCP/IP e da Internet.

Exercícios

1.1 Siga os passos no final da Seção 1.9 para descobrir informações sobre sua topologia de rede.

1.2 Obtenha o código-fonte para os exemplos neste texto (consulte o Prefácio). Compile e teste o cliente TCP de data/hora da Figura 1.5. Execute o programa algumas vezes, especificando cada vez um endereço IP diferente como o argumento da linha de comando.

1.3 Modifique o primeiro argumento para `socket` na Figura 1.5 para ser 9999. Compile e execute o programa. O que acontece? Descubra o valor de `errno` correspondente ao erro que é impresso. Como você pode encontrar informações adicionais sobre esse erro?

1.4 Modifique a Figura 1.5 posicionando um contador no loop `while`, contando o número de vezes que `read` retorna um valor maior que 0. Imprima o valor do contador antes de terminar. Compile e execute seu novo cliente.

1.5 Modifique a Figura 1.9 como a seguir. Primeiro, altere o número da porta atribuída ao membro `sin_port` de 13 para 9999. Em seguida, altere a única chamada a `write` para um loop que chama `write` para cada byte da string de resultado. Compile esse servidor modificado e inicie sua execução em segundo plano. Em seguida, modifique o cliente do exercício anterior (que imprime o contador antes de terminar), alterando o número da porta atribuída ao membro `sin_port` de 13 para 9999. Inicie esse cliente, especificando o endereço IP do host em que o servidor modificado está em execução como o argumento de linha de comando. Que valor é impresso como o contador do cliente? Se possível, também tente executar o cliente e o servidor em hosts diferentes.

Capítulo 2

Camada de Transporte: TCP, UDP e SCTP

2.1 Introdução

Este capítulo oferece uma visão geral sobre os protocolos no conjunto TCP/IP utilizados nos exemplos deste livro. Nosso objetivo é fornecer detalhes suficientes a partir de uma perspectiva de programação de rede para entender como utilizar os protocolos e fazer referências a descrições mais detalhadas sobre o *design*, a implementação e a história reais.

Este capítulo focaliza a camada de transporte: TCP, UDP e Stream Control Transmission Protocol (SCTP). A maioria das aplicações cliente/servidor utiliza TCP ou UDP. O SCTP é um protocolo mais recente, originalmente projetado para transporte de sinalização de telefonia pela Internet. Esses protocolos de transporte utilizam o protocolo IP de camada rede, IPv4 ou IPv6. Embora seja possível utilizar o IPv4 ou o IPv6 diretamente, pulando a camada de transporte, essa técnica, freqüentemente chamada *soquetes brutos (raw sockets)*, é utilizada bem menos. Portanto, temos uma descrição mais detalhada sobre o IPv4 e o IPv6, junto com o ICMPv4 e o ICMPv6, no Apêndice A.

O UDP é um protocolo de datagrama simples e não-confiável, enquanto o TCP é um protocolo confiável e sofisticado de fluxo de bytes. O SCTP é semelhante ao TCP como um protocolo de transporte confiável, mas também fornece limites de mensagem, suporte no nível de transporte para multihoming e uma maneira de minimizar o bloqueio de início de linha. Precisamos entender os serviços fornecidos por esses protocolos de transporte à aplicação, de modo que saibamos o que é tratado pelo protocolo e o que devemos tratar na aplicação.

Há recursos de TCP que, quando entendidos, tornam mais fácil a criação de clientes e servidores robustos. Além disso, quando entendemos esses recursos, torna-se mais fácil depurar nosso cliente e servidor utilizando as ferramentas comumente fornecidas, como a `netstat`. Abordaremos vários tópicos neste capítulo que entram nessa categoria: handshake de três vias do TCP, seqüência de término de conexão do TCP e estado TIME_WAIT do TCP; handshake de quatro vias do SCTP e término da conexão do SCTP; mais SCTP, TCP e UDP armazenados em buffer pela camada de soquete e assim por diante.

2.2 Visão geral

Embora o conjunto de protocolos seja chamado "TCP/IP", há outros membros dessa família além desses. A Figura 2.1 mostra uma visão geral desses protocolos.

Mostramos tanto o IPv4 como o IPv6 nessa figura. Movendo-se da direita para a esquerda, as cinco aplicações mais à direita estão utilizando o IPv6; discutiremos a constante `AF_INET6` no Capítulo 3, juntamente com a estrutura `sockaddr_in6`. As seis aplicações a seguir utilizam o IPv4.

A aplicação mais à esquerda, `tcpdump`, comunica-se diretamente com o enlace de dados utilizando um filtro de pacotes BSD (BSD packet filter – BPF) ou a interface de provedor de enlace de dados (datalink provider interface – DLPI). Marcamos a linha tracejada abaixo das nove aplicações à direita como a API, que normalmente são soquetes ou XTI. A interface para BPF ou DLPI não utiliza soquetes ou XTI.

> Há uma exceção a isso, que descreveremos em mais detalhes no Capítulo 28: o Linux fornece acesso ao enlace de dados utilizando um tipo de soquete especial chamado `SOCK_PACKET`.

Também observamos na Figura 2.1 que o programa `traceroute` utiliza dois soquetes: um para IP e outro para ICMP. No Capítulo 28, desenvolveremos as versões do IPv4 e do IPv6 tanto de `ping` como de `traceroute`.

Agora, descreveremos cada uma das caixas de protocolo nessa figura.

IPv4 O *Internet Protocol version 4,* IPv4, que costumamos denotar simplesmente como IP, tem sido o protocolo "burro de carga" do conjunto IP desde o início da década de

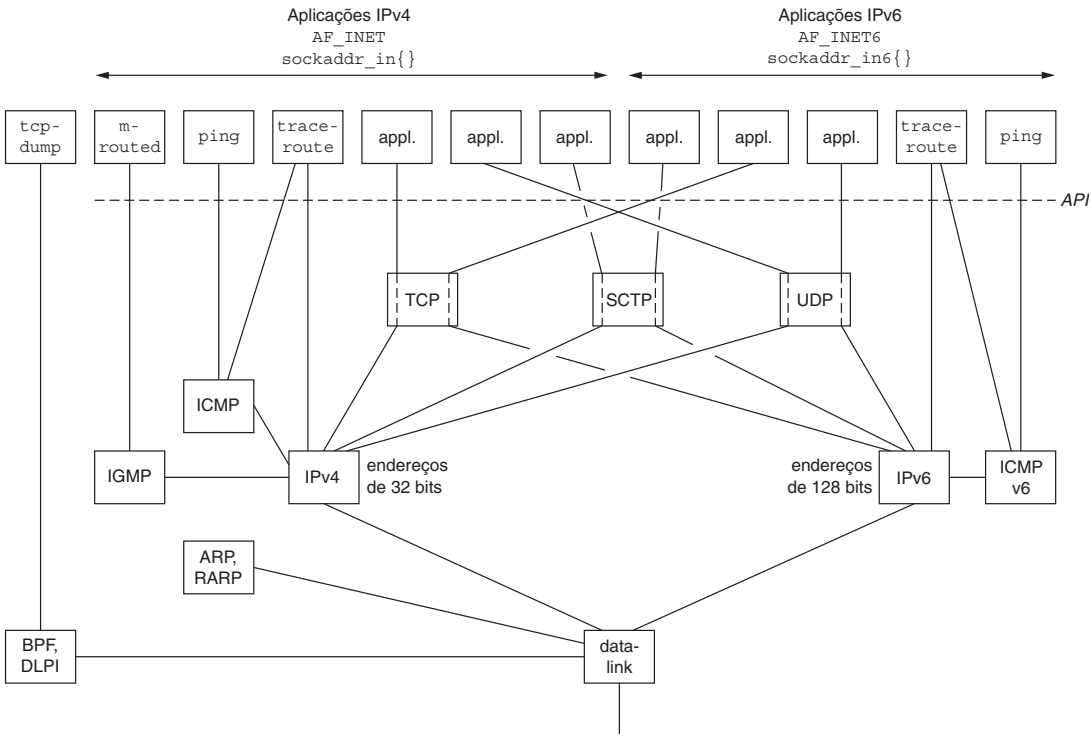


Figura 2.1 Visão geral sobre os protocolos TCP/IP.

1980. Ele utiliza endereços de 32 bits (Seção A.4). O IPv4 fornece serviço de entrega de pacote para TCP, UDP, SCTP, ICMP e IGMP.

IPv6 O *Internet Protocol version 6*, IPv6, foi projetado em meados da década de 1990 como um substituto do IPv4. A alteração mais importante é um endereço maior que abrange 128 bits (Seção A.5) para lidar com o crescimento explosivo da Internet na mesma década. O IPv6 fornece serviço de entrega de pacote para TCP, UDP, SCTP e ICMPv6.

Freqüentemente, utilizamos a palavra "*IP*" como um adjetivo, como em camada IP e endereço IP, quando a distinção entre IPv4 e IPv6 não é necessária.

TCP *Transmission Control Protocol*. O TCP é um protocolo orientado a conexão que fornece um fluxo de bytes full-duplex confiável aos seus usuários. Soquetes TCP são um exemplo de soquetes de fluxo (*stream sockets*). O TCP cuida dos detalhes como reconhecimentos, tempos-limite, retransmissões e assim por diante. A maioria dos programas aplicativos da Internet utiliza o TCP. Observe que o TCP pode utilizar o IPv4 ou o IPv6.

UDP *User Datagram Protocol*. O UDP é um protocolo sem conexão e os soquetes UDP são um exemplo de soquetes de datagrama (*datagram sockets*). Não há garantias de que os datagramas UDP alcancem seu destino final. Como ocorre com o TCP, o UDP pode utilizar o IPv4 ou o IPv6.

SCTP *Stream Control Transmission Protocol*. O SCTP é um protocolo orientado a conexão que fornece uma associação full-duplex confiável. A palavra "associação" é utilizada como uma referência a uma conexão no SCTP porque o SCTP é multihomed, envolvendo um conjunto de endereços IP e uma única porta para cada lado de uma associação. O SCTP fornece um serviço de mensagem, que mantém os limites de registros. Como ocorre com o TCP e o UDP, o SCTP pode utilizar o IPv4 ou o IPv6, mas também pode utilizá-los simultaneamente na mesma associação.

ICMP *Internet Control Message Protocol*. O ICMP trata erros e controla as informações entre roteadores e hosts. Normalmente, essas mensagens são geradas e processadas pelo próprio software de rede TCP/IP, não por processos de usuário, embora mostremos os programas `ping` e `traceroute`, que utilizam o ICMP. Às vezes nos referimos a esse protocolo como ICMPv4, para distingui-lo do ICMPv6.

IGMP *Internet Group Management Protocol*. O IGMP é utilizado com multicasting (Capítulo 21), que é opcional com o IPv4.

ARP *Address Resolution Protocol*. O ARP mapeia um endereço IPv4 para um endereço de hardware (como um endereço Ethernet). O ARP é normalmente utilizado em redes de broadcast como Ethernet, token ring e FDDI e não é necessário em redes ponto a ponto.

RARP *Reverse Address Resolution Protocol*. O RARP mapeia um endereço de hardware para um endereço IPv4. Às vezes ele é utilizado quando um nó sem disco é inicializado.

ICMPv6 *Internet Control Message Protocol version 6*. O ICMPv6 combina a funcionalidade do ICMPv4, do IGMP e do ARP.

BPF *BSD packet filter*. Essa interface fornece acesso à camada de enlace de dados. Normalmente, ela é encontrada em kernels derivados do Berkeley.

DLPI *Datalink provider interface*. Essa interface também fornece acesso à camada de enlace de dados. Normalmente ela é fornecida com o SVR4.

Cada protocolo de Internet é definido por um ou mais documentos chamados *Request for Comments (RFC)*, que são as especificações formais de protocolos. A solução ao Exercício 2.1 mostra como obter RFCs.

Utilizamos os termos *"host de IPv4/IPv6"* e *"host de pilha dual"* para indicar hosts que suportam tanto o IPv4 como o IPv6.

Detalhes adicionais sobre os próprios protocolos TCP/IP estão em TCPv1. A implementação 4.4BSD do TCP/IP é descrita no TCPv2.

2.3 User Datagram Protocol (UDP)

O UDP é um protocolo de camada de transporte simples. Ele é descrito na RFC 768 (Postel, 1980). A aplicação grava uma mensagem em um soquete UDP, que é então *encapsulada* em um *datagrama* UDP e, em seguida, encapsulada ainda mais como um datagrama IP e, finalmente, enviada ao seu destino. Não há garantias de que um datagrama UDP alcançará seu destino final, de que essa ordem será preservada pela rede ou de que datagramas cheguem somente uma vez.

O problema que encontramos com programação de rede que utiliza UDP é sua falta de confiabilidade. Se um datagrama alcançar seu destino final mas a soma de verificação detectar um erro, ou se o datagrama se perder na rede, ele não será entregue ao soquete UDP e não será retransmitido automaticamente. Se quisermos ter certeza de que um datagrama alcançará seu destino, podemos incorporar uma grande quantidade de recursos a nossa aplicação: reconhecimentos provenientes da outra extremidade, tempos-limite, retransmissões e assim por diante.

Cada datagrama UDP tem um comprimento. O comprimento de um datagrama é passado para a aplicação receptora junto com os dados. Já mencionamos o fato de que o TCP é um protocolo de *fluxo de bytes* (*byte-stream*), sem absolutamente nenhum limite de registro (Seção 1.2), o que difere do UDP.

Também dizemos que o UDP fornece um serviço *sem conexão*, uma vez que não é necessário haver nenhum relacionamento de longo prazo entre o cliente e servidor UDP. Por exemplo, um cliente UDP pode criar um soquete e enviar um datagrama a um determinado servidor e então enviar imediatamente um outro datagrama no mesmo soquete a um servidor diferente. De maneira semelhante, um servidor UDP pode receber vários datagramas em um único soquete UDP, cada um a partir de um cliente diferente.

2.4 Transmission Control Protocol (TCP)

O serviço fornecido pelo TCP para uma aplicação é diferente do serviço fornecido pelo UDP. O TCP é descrito na RFC 793 (Postel, 1981c) e atualizado pela RFC 1323 (Jacobson, Braden e Borman, 1992), RFC 2581 (Allman, Paxson e Stevens, 1999), RFC 2988 (Paxson e Allman, 2000) e RFC 3390 (Allman, Floyd e Partridge, 2002). Primeiro, o TCP fornece *conexões* entre clientes e servidores. Um cliente TCP estabelece uma conexão com um dado servidor, troca os dados com esse servidor na conexão e então termina a conexão.

O TCP também fornece *confiabilidade*. Quando o TCP envia os dados à outra extremidade, ele requer um reconhecimento em retorno. Se um reconhecimento não for recebido, o TCP retransmite automaticamente os dados e espera um período de tempo mais longo. Depois de algumas retransmissões, o TCP desistirá, com o tempo total gasto tentando enviar os dados em geral entre 4 e 10 minutos (dependendo da implementação).

> Observe que o TCP não garante que os dados serão recebidos pela outra extremidade, uma vez que isso é impossível. Ele entrega os dados para a outra extremidade se possível e notifica o usuário (desistindo das retransmissões e quebrando a conexão) se não for possível. Portanto, o TCP não pode ser descrito como um protocolo 100% confiável; ele fornece entrega confiável de dados *ou* uma notificação confiável de falha.

O TCP contém algoritmos para estimar o *tempo de viagem de ida e volta* (*round-trip time* – RTT) entre um cliente e um servidor dinamicamente de modo que ele saiba quanto tempo esperar por um reconhecimento. Por exemplo, o RTT em uma rede local (LAN) pode ser

de milissegundos e, em uma rede geograficamente distribuída (WAN), segundos. Além disso, o TCP estima continuamente o RTT uma dada conexão, porque o RTT é afetado pelas variações no tráfego de rede.

O TCP também *seqüencia* os dados associando um número de seqüência a cada byte que envia. Por exemplo, suponha que uma aplicação grave 2.048 bytes em um soquete TCP, fazendo com que o TCP envie dois segmentos, o primeiro contendo os dados com os números de seqüência 1-1.024 e o segundo contendo os dados com os números de seqüência 1.025-2.048. (Um *segmento* é a unidade de dados que o TCP passa para o IP). Se os segmentos chegarem fora de ordem, o TCP receptor reordenará os dois segmentos com base nos seus números de seqüência antes de passar os dados para a aplicação receptora. Se o TCP receber dados duplicados de um peer (digamos que este achou que um segmento foi perdido e o retransmitiu, quando na realidade o segmento não havia sido perdido, simplesmente a rede estava sobrecarregada), ele pode detectar que os dados foram duplicados (a partir dos números de seqüência) e descartar os dados duplicados.

> Não há confiabilidade fornecida pelo UDP. O UDP por si só não fornece nada como reconhecimentos, números de seqüência, estimativa de RTT, tempos-limite ou retransmissões. Se um datagrama UDP for duplicado na rede, duas cópias podem ser entregues para o host receptor. Além disso, se um cliente UDP enviar dois datagramas ao mesmo destino, eles podem ser reordenados pela rede e chegar fora de ordem. Aplicações UDP devem tratar todos esses casos, como mostraremos na Seção 22.5.

O TCP fornece um *controle de fluxo* e sempre informa ao seu peer exatamente quantos bytes de dados está disposto a aceitar do peer por vez. Isso é chamado *janela* anunciada. A janela é a quantidade de espaço atualmente disponível no buffer de recebimento, garantindo que o remetente não possa estourar o buffer. A janela muda dinamicamente ao longo do tempo: à medida que os dados são recebidos do remetente, seu tamanho diminui, mas como a aplicação receptora lê os dados do buffer, seu tamanho aumenta. É possível para a janela alcançar 0: quando o buffer TCP de recepção para um soquete está cheio e ela deve esperar a aplicação ler os dados do buffer antes de receber mais dados do peer.

> O UDP não fornece nenhum controle de fluxo. É fácil para um remetente de UDP rápido transmitir datagramas a uma taxa que o receptor de UDP não possa acompanhar, como mostraremos na Seção 8.13.

Por fim, uma conexão TCP é *full-duplex*. Isso significa que uma aplicação pode enviar e receber dados nas duas direções em uma dada conexão a qualquer momento. Isso significa que o TCP deve monitorar as informações sobre o estado como números de seqüência e tamanhos de janela de cada direção do fluxo de dados: envio e recebimento. Depois que uma conexão full-duplex é estabelecida, ela pode ser transformada em uma conexão simplex se desejado (consulte a Seção 6.6).

> O UDP pode ser full-duplex.

2.5 Stream Control Transmission Protocol (SCTP)

O SCTP oferece serviços semelhantes aos do UDP e do TCP. O SCTP é descrito na RFC 2960 (Stewart *et al.,* 2000) e atualizado pela RFC 3309 (Stone, Stewart e Otis, 2002). Uma introdução a ele está disponível na RFC 3286 (Ong e Yoakum, 2002). O SCTP fornece *associações* entre clientes e servidores e às aplicações, confiabilidade, seqüenciamento, controle de fluxo e transferência de dados full-duplex, como o TCP. A palavra *"associação"* é utilizada no SCTP em vez de "conexão" para evitar a conotação de que uma conexão envolve comunicação entre somente dois endereços IP. Uma associação refere-se a uma comunicação entre dois sistemas, o que pode envolver mais de dois endereços devido ao multihoming.

Diferentemente do TCP, o SCTP é *orientado a mensagem*. Ele fornece entrega seqüenciada de registros individuais. Como ocorre com o UDP, o comprimento de um registro escrito pelo remetente é passado para a aplicação receptora.

O SCTP pode fornecer múltiplos fluxos entre pontos finais de conexão, cada um com sua própria entrega seqüenciada confiável de mensagens. Uma mensagem perdida em um desses fluxos não bloqueia a entrega de mensagens em outros fluxos quaisquer. Essa abordagem é o contrário do TCP, em que uma perda em qualquer ponto no fluxo único de entrega de bytes bloqueia todos os dados futuros na conexão até que a perda seja reparada.

O SCTP também fornece um recurso de multihoming, que permite que uma única extremidade de SCTP suporte múltiplos endereços IP. Esse recurso pode fornecer aumento da robustez contra falhas de rede. Uma extremidade pode ter múltiplas conexões de rede redundantes, em que cada uma dessas redes tem uma conexão diferente à infra-estrutura da Internet. O SCTP pode contornar a falha de uma rede ou de caminho pela Internet alternando para outro endereço já associado com a associação SCTP.

> Robustez semelhante pode ser obtida no TCP com a ajuda de protocolos de roteamento. Por exemplo, conexões BGP dentro de um domínio (iBGP) freqüentemente utilizam endereços atribuídos a uma interface virtual dentro do roteador como os pontos finais da conexão TCP. O protocolo de roteamento do domínio assegura que, se houver uma rota entre dois roteadores, ela poderá ser utilizada, o que não seria possível se, por exemplo, os endereços utilizados pertencessem a uma interface que travasse. O recurso multihoming do SCTP permite que hosts sejam multihome, não apenas os roteadores, e que esse processo de multihome ocorra em diferentes provedores de serviço, o que o método TCP baseado em roteamento não pode permitir.

2.6 Estabelecimento e término de uma conexão TCP

Para ajudar a entender as funções `connect`, `accept` e `close` e também a depurar aplicações TCP que utilizam o programa `netstat`, devemos entender como as conexões TCP são estabelecidas e terminadas e o diagrama de transição de estado do TCP.

Handshake de três vias

O cenário a seguir ocorre quando uma conexão TCP é estabelecida:

1. O servidor deve estar preparado para aceitar uma conexão entrante. Isso normalmente é feito chamando `socket`, `bind` e `listen` e é chamado *abertura passiva (passive open)*.
2. O cliente requer uma *abertura ativa (active open)* chamando `connect`. Isso faz com que o TCP cliente envie um segmento de "sincronização" (SYN – synchronize), que diz ao servidor o número inicial de seqüência do cliente para os dados que o cliente enviará na conexão. Normalmente, não há nenhum dado enviado com o SYN; ele contém apenas um cabeçalho IP, um cabeçalho TCP e possíveis opções TCP (que discutiremos em breve).
3. O servidor deve reconhecer (ACK – acknowledge) o SYN do cliente e também enviar seu próprio SYN contendo o número da seqüência inicial dos dados que enviará na conexão. O servidor envia seu SYN e o ACK do cliente SYN em um único segmento.
4. O cliente deve reconhecer o SYN do servidor.

O número mínimo de pacotes requerido para essa troca é três; conseqüentemente, isso é chamado *handshake (aperto de mãos) de três vias* do TCP. Mostramos os três segmentos na Figura 2.2.

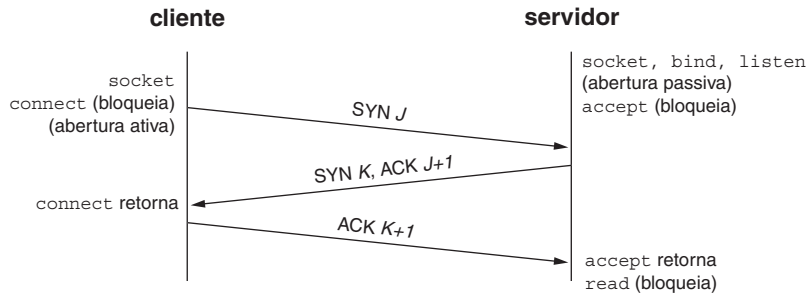


Figura 2.2 Handshake de três vias do TCP.

Mostramos o número de seqüência inicial do cliente como *J* e o número de seqüência inicial do servidor como *K*. O número de reconhecimento em um ACK é o próximo número de seqüência esperado para a extremidade transmitindo o ACK. Como um SYN ocupa um byte do espaço do número de seqüência, o número de reconhecimento no ACK de cada SYN é o número de seqüência inicial mais um. De maneira semelhante, o ACK de cada FIN é o número de seqüência do FIN mais um.

> Uma analogia cotidiana para estabelecer uma conexão TCP é o sistema de telefonia (Nemeth, 1997). A função `socket` equivale a ter um telefone para uso. `bind` informa a outras pessoas o seu número de telefone de modo que elas possam chamá-lo. `listen` ativa a campainha de modo que você ouça quando uma nova chamada chega. `connect` requer que saibamos e disquemos o número de telefone da outra pessoa. `accept` é quando a pessoa sendo chamada responde ao telefonema. Fazer com que a identidade do cliente retorne por `accept` (em que a identificação é o endereço IP e o número da porta do cliente) é semelhante a fazer com que o recurso ID do chamador mostre o número de telefone deste. Uma diferença, porém, é que `accept` retorna a identidade do cliente somente depois que a conexão foi estabelecida, ao passo que o recurso de identificação de chamada (caller ID) mostra o número de telefone do chamador antes de optarmos por responder ou não ao telefonema. Se o DNS for utilizado (Capítulo 11), ele fornecerá um serviço análogo a uma lista telefônica. `getaddrinfo` é semelhante a pesquisar o número de telefone de uma pessoa na agenda telefônica. `getnameinfo` seria equivalente a fazer uma pesquisa em uma agenda telefônica classificada por números de telefone, em vez de classificada por nome.

Opções TCP

Cada SYN pode conter opções TCP. As opções comumente utilizadas incluem as seguintes:

- Opção MSS. Com essa opção, o TCP que envia o SYN anuncia o tamanho máximo de seu segmento, a quantidade máxima de dados que está disposto a aceitar em cada segmento TCP, nessa conexão. O TCP que envia utiliza o valor de MSS do receptor como o tamanho máximo de um segmento que ele envia. Veremos como buscar e configurar essa opção TCP com a opção de soquete `TCP_MAXSEG` (Seção 7.9).
- Opção de dimensionamento de janela. A janela máxima que um TCP pode anunciar a outro TCP é 65.535, porque o campo correspondente no cabeçalho TCP ocupa 16 bits. Mas conexões de alta velocidade, comuns na Internet atual (45 Mbits/seg e mais rápidas, conforme descrito na RFC 1323 [Jacobson, Braden e Borman, 1992]), ou longos caminhos de retardo (enlaces de satélite) requerem uma janela maior para obter o maior throughput possível. Essa opção mais recente especifica que a janela anunciada no cabeçalho TCP deve ser dimensionada (deslocada à esquerda) por 0 a 14 bits, fornecendo uma janela má-

xima de quase um gigabyte (65.535×2^{14}). Os dois sistemas finais devem suportar essa opção para que o dimensionamento de janela seja utilizado em uma conexão. Veremos como afetar essa opção com a opção de soquete `SO_RCVBUF` (Seção 7.5).

> Para fornecer a interoperabilidade com implementações mais antigas que não suportam essa opção, são aplicadas as seguintes regras. O TCP pode enviar a opção com seu SYN como parte de uma abertura ativa (open ativo). Porém, pode dimensionar suas janelas somente se a outra extremidade também enviar a opção com seu SYN. De maneira semelhante, o TCP do servidor pode enviar essa opção somente se receber a opção com o SYN do cliente. Essa lógica assume que as implementações ignoram as opções que não entendem, o que é requerido e comum, mas, infelizmente, sem garantias para todas as implementações.

- Opção de timbre de data/hora (*timestamp*). Essa opção é necessária para conexões de alta velocidade para evitar possível corrupção de dados causada por segmentos antigos, retardados ou duplicados. Uma vez que é uma opção mais recente, é negociada de maneira semelhante à opção de dimensionamento de janela. Como programadores de rede, não há nada com que precisemos nos preocupar nessa opção.

Essas opções comuns são suportadas pela maioria das implementações. As duas últimas às vezes são chamadas "opções da RFC 1323", visto que essa RFC (Jacobson, Braden e Borman, 1992) especifica as opções. Elas também são chamadas "*long fat pipe options*" ("opções de pipe longo e gordo"), uma vez que uma rede com uma alta largura de banda ou um retardo longo é chamada *long fat pipe,* em inglês. O Capítulo 24 do TCPv1 contém outros detalhes sobre essas opções.

Término de conexão TCP

Embora sejam necessários três segmentos para estabelecer uma conexão, são necessários quatro para terminá-la.

1. Uma aplicação primeiro chama `close` e dizemos que essa extremidade realiza o fechamento ativo (*active close*). O TCP nessa extremidade envia um segmento FIN, o que significa que ele terminou de enviar os dados.
2. A outra extremidade que recebe o FIN realiza o fechamento passivo (*passive close*). O FIN recebido é reconhecido pelo TCP. A recepção do FIN também é passada para a aplicação como um fim de arquivo (depois que todos os dados que talvez já estejam enfileirados são recebidos pela aplicação), uma vez que a recepção do FIN significa que a aplicação não receberá nenhum dado adicional na conexão.
3. Posteriormente, a aplicação que recebeu o fim do arquivo irá chamar `close` para fechar o seu soquete. Isso faz com que seu TCP envie um FIN.
4. O TCP no sistema que recebe esse FIN final (a extremidade que fez o close ativo) reconhece o FIN.

Como um FIN e um ACK são requeridos em cada direção, quatro segmentos normalmente são exigidos. Utilizamos o qualificador "normalmente" porque, em alguns cenários, o FIN no Passo 1 é enviado com dados. Além disso, os segmentos nos Passos 2 e 3 são ambos provenientes da extremidade que realiza o close passivo e poderiam ser combinados em um segmento. Mostramos esses pacotes na Figura 2.3.

Um FIN ocupa um byte de espaço do número de seqüência como ocorre com um SYN. Portanto, o ACK de cada FIN é o número de seqüência do FIN mais um.

Entre os Passos 2 e 3 é possível que os dados fluam da extremidade que faz o close passivo à extremidade que faz o close ativo. Isso é chamado *half-close* ("meio fechamento") e será discutido em detalhes com a função `shutdown` na Seção 6.6.

O envio de cada FIN ocorre quando um soquete é fechado. Indicamos que a aplicação chama `close` para que isso aconteça, mas observe que quando um processo Unix termina, quer voluntariamente (chamando `exit` ou fazendo com que a função `main` retorne) ou involuntariamente (recebendo um sinal que termina o processo), todos os descritores abertos são fechados, o que também fará com que um FIN seja enviado a qualquer conexão TCP que ainda esteja aberta.

Embora mostremos o cliente na Figura 2.3 realizando o close ativo, uma das extremidades – o cliente ou o servidor – pode realizar o close ativo. Freqüentemente o cliente realiza o close ativo, mas com alguns protocolos (notavelmente com o HTTP/1.0) o servidor realiza o close ativo.

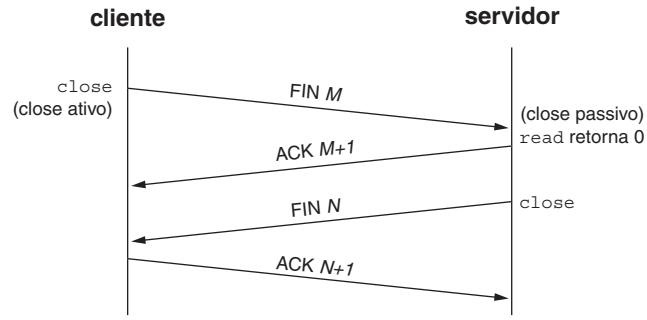


Figura 2.3 Pacotes trocados quando uma conexão TCP está fechada.

Diagrama de transição de estado TCP

A operação TCP com referência ao estabelecimento e término de uma conexão pode ser especificada com um *diagrama de transição de estado*. Mostramos isso na Figura 2.4.

Há 11 diferentes estados definidos para uma conexão, as regras de TCP ditam as transições de um estado para outro, com base no estado atual e no segmento recebido nesse estado. Por exemplo, se uma aplicação realizar um open ativo no estado CLOSED, o TCP envia um SYN e o novo estado é SYN_SENT. Se o TCP em seguida receber um SYN com um ACK, ele envia um ACK e o novo estado é ESTABLISHED. Esse estado final é onde ocorre a maior parte da transferência de dados.

A duas setas que partem do estado ESTABLISHED lidam com o término de uma conexão. Se uma aplicação chamar `close` antes de receber um FIN (um close ativo), a transição ocorre no estado FIN_WAIT_1. Mas, se uma aplicação receber um FIN enquanto estiver no estado ESTABLISHED (um close passivo), a transição ocorre no estado CLOSE_WAIT.

Indicamos as transições normais de cliente com uma linha escura sólida e as transições normais de servidor com uma linha escura tracejada. Também observe que há duas transições que não discutimos: um open simultâneo (quando as duas extremidades enviam SYNs quase ao mesmo tempo e os SYNs cruzam-se na rede) e um close simultâneo (quando as duas extremidades enviam FINs ao mesmo tempo). O Capítulo 18 do TCPv1 contém exemplos e uma discussão desses dois cenários, possíveis, mas raros.

Uma das razões para mostrar o diagrama de transição de estado é mostrar os 11 estados do TCP com seus nomes. Esses estados são exibidos pela ferramenta `netstat`, útil ao depurar aplicações cliente/servidor. Utilizaremos `netstat` para monitorar alterações de estado no Capítulo 5.

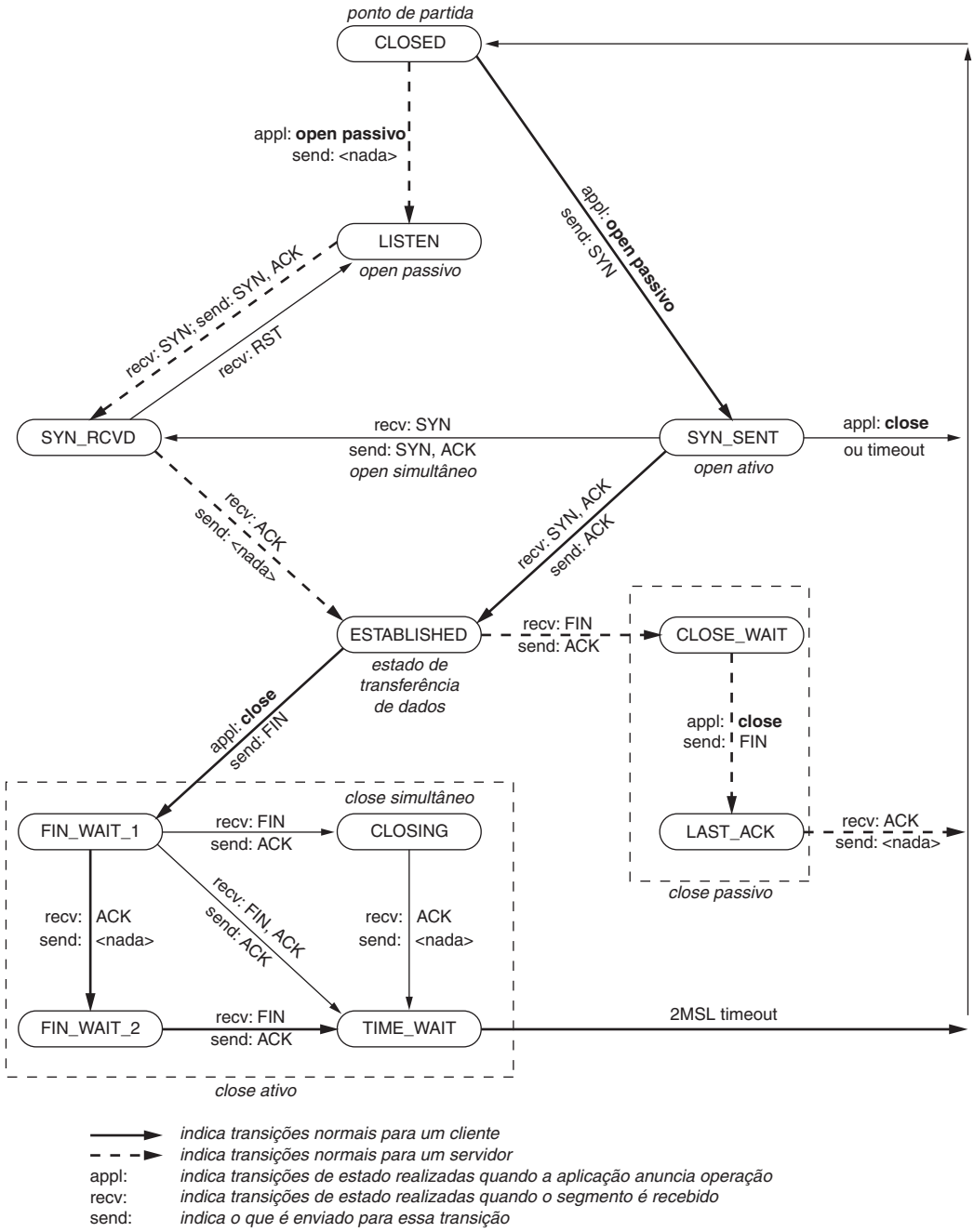


Figura 2.4 Diagrama de transição de estado TCP.

Observando os pacotes

A Figura 2.5 mostra a troca real de pacotes que acontece para uma conexão TCP completa: estabelecimento de conexão, transferência de dados e término de conexão. Também mostramos os estados de TCP pelos quais cada extremidade passa.

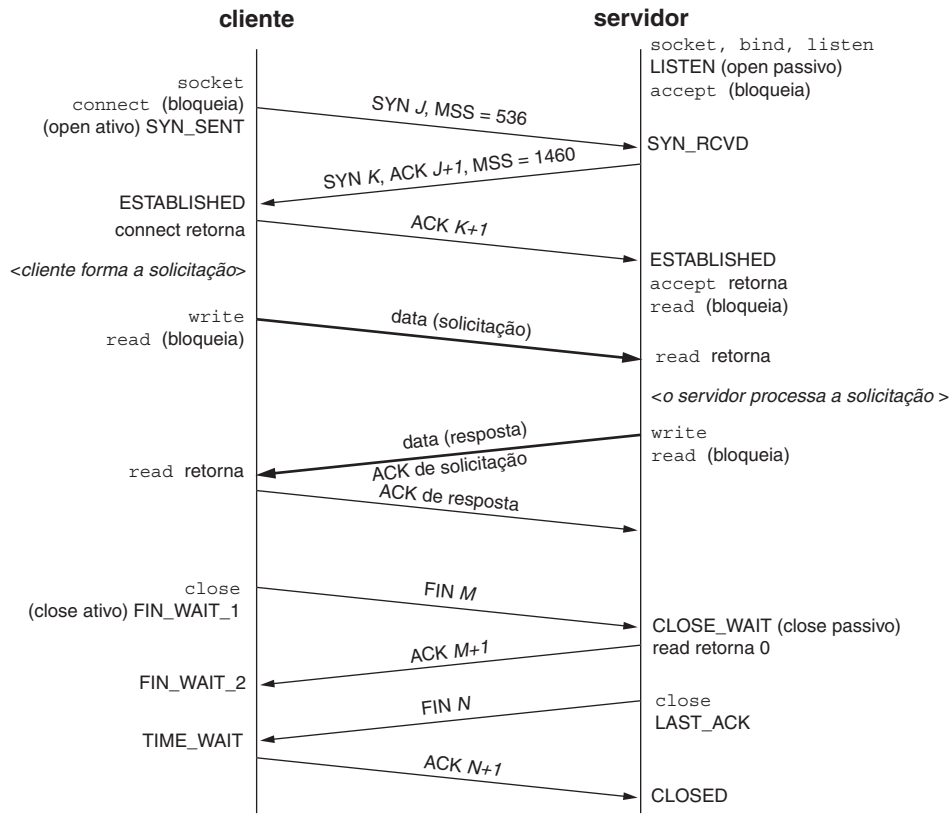


Figura 2.5 Troca de pacotes em uma conexão TCP.

O cliente nesse exemplo anuncia um MSS de 536 (indicando que ele implementa somente o tamanho mínimo de buffer de remontagem) e o servidor anuncia um MSS de 1.460 (típico para IPv4 em uma Ethernet). Não há problema se o MSS for diferente em cada direção (veja o Exercício 2.5).

Depois que uma conexão é estabelecida, o cliente forma e envia uma solicitação ao servidor. Supomos que essa solicitação se ajusta a um único segmento TCP (isto é, menos de 1.460 bytes dado o MSS anunciado do servidor). O servidor processa a solicitação e envia uma resposta e assumimos que a resposta cabe em um único segmento (menos de 536 nesse exemplo). Mostramos os dois segmentos de dados como setas mais escuras. Observe que o reconhecimento da solicitação do cliente é enviado junto com a resposta do servidor. Isso é chamado de "*piggybacking*" ("pegar uma carona") e normalmente acontecerá quando o tempo que o servidor leva para processar a solicitação e gerar a resposta for menor que 200 ms. Se o servidor levar mais tempo, digamos um segundo, veríamos o reconhecimento seguido pela resposta. (A dinâmica do fluxo de dados TCP é abrangida em detalhes nos Capítulos 19 e 20 do TCPv1.)

Em seguida, mostraremos os quatro segmentos que terminam a conexão. Observe que a extremidade que realiza o close ativo (nesse cenário, o cliente) entra no estado TIME_WAIT. Discutiremos esse estado na próxima seção.

É importante observar na Figura 2.5 que, se o propósito dessa conexão fosse enviar uma solicitação com um segmento e receber uma resposta com um segmento, haveria oito segmentos de overhead envolvidos ao utilizar o TCP. Se, em vez disso, o UDP fosse utilizado, somente dois pacotes seriam trocados: a solicitação e a resposta. Mas alternar de TCP para UDP re-

move toda a confiabilidade que o TCP garante à aplicação, empurrando muitos desses detalhes da camada de transporte (TCP) para a aplicação UDP. Outro recurso importante fornecido pelo TCP é o controle de congestionamento, que deve então ser tratado pela aplicação UDP. Contudo, é importante entender que várias aplicações são construídas utilizando UDP porque a aplicação troca poucos dados e o UDP evita o overhead do estabelecimento e término de uma conexão TCP.

2.7 Estado TIME_WAIT

Sem dúvida, um dos aspectos mais mal-entendidos do TCP com relação à programação de rede é seu estado TIME_WAIT. Podemos ver, na Figura 2.4, que a extremidade que realiza o close ativo passa por esse estado. A duração dessa extremidade nesse estado é duas vezes o *tempo de vida máximo do segmento* (maximum segment lifetime – MSL), às vezes chamado 2MSL.

Cada implementação do TCP deve escolher um valor para o MSL. O valor recomendado na RFC 1122 (Braden, 1989) é 2 minutos, embora implementações derivadas do Berkeley tradicionalmente utilizem, em vez disso, um valor de 30 segundos. Isso significa que a duração do estado TIME_WAIT está entre 1 e 4 minutos. O MSL é a quantidade máxima de tempo em que qualquer datagrama IP pode viver em uma rede. Sabemos que esse tempo é limitado porque cada datagrama contém um limite de hop de 8 bits (o campo IPv4 TTL na Figura A.1 e o campo de limite de hops do IPv6 na Figura A.2) com um valor máximo de 255. Embora isso seja um limite de hops e não um verdadeiro limite de tempo, assume-se que um pacote com o limite máximo de hops de 255 não pode existir em uma rede por mais que MSL segundos.

A maneira como um pacote é "perdido" em uma rede normalmente é o resultado de anomalias no roteamento. Uma queda de roteador ou um enlace parado entre os dois roteadores faz com que os protocolos de roteamento levem segundos ou minutos para se estabilizarem e encontrarem um caminho alternativo. Durante esse período de tempo, podem ocorrer loops no roteamento (o roteador A envia pacotes ao roteador B e B envia-os de volta a A) e pacotes podem ser capturados nesses loops. Neste ínterim, assumindo que o pacote perdido seja um segmento TCP, o tempo limite do TCP emissor expira e ele retransmite o pacote, que chega ao destino final por um caminho alternativo. Mas, posteriormente (até MSL segundos depois que o pacote iniciou sua viagem), o loop de roteamento é corrigido e o pacote que ficou perdido no loop é enviado ao destino final. Esse pacote original é chamado *duplicata perdida* (*lost duplicate*) ou *duplicata errante* (*wandering duplicate*). O TCP deve tratar essas duplicatas.

Há duas razões para o estado TIME_WAIT:

1. Implementar o término da conexão full-duplex do TCP de maneira confiável
2. Permitir que segmentos duplicados antigos expirem na rede

A primeira razão pode ser explicada examinando a Figura 2.5 e supondo que o ACK final é perdido. O servidor enviará novamente seu FIN final, portanto o cliente deve manter as informações do estado, permitindo que ele envie o ACK final novamente. Se o cliente não mantivesse essas informações, ele responderia com um RST (um tipo diferente de segmento TCP), que seria interpretado pelo servidor como um erro. Se o TCP estiver realizando todo o trabalho necessário para terminar as duas direções do fluxo de dados de maneira limpa para uma conexão (seu close full-duplex), ele deve então tratar corretamente a perda de qualquer um desses quatro segmentos. Esse exemplo também mostra por que a extremidade que realiza o close ativo é a que permanece no estado TIME_WAIT: porque essa extremidade é aquela que poderia precisar retransmitir o ACK final.

Para entender a segunda razão do estado TIME_WAIT, assuma que temos uma conexão TCP entre 12.106.32.254 na porta 1500 e 206.168.112.219 na porta 21. Essa conexão está fechada e, mais tarde, estabelecemos uma outra conexão entre os mesmos endereços IP e por-

tas: o 12.106.32.254 na porta 1500 e o 206.168.112.219 na porta 21. Essa última conexão é chamada *encarnação* da conexão anterior, uma vez que os endereços IP e as portas são iguais. O TCP deve evitar que duplicatas antigas em uma conexão reapareçam posteriormente e sejam erroneamente interpretadas como pertencentes a uma nova encarnação da mesma conexão. Para fazer isso, o TCP não iniciará uma nova encarnação de uma conexão que está atualmente no estado TIME_WAIT. Como a duração do estado TIME_WAIT é duas vezes o MSL, isso permite que MSL segundos de um pacote em uma direção sejam perdidos e que outros MSL segundos da resposta também sejam perdidos. Impondo essa regra, garantimos que, quando estabelecemos uma conexão TCP com sucesso, todas as duplicatas antigas provenientes de encarnações anteriores da conexão expiraram na rede.

> Há uma exceção a essa regra. Implementações derivadas do Berkeley iniciarão uma nova encarnação de uma conexão que está atualmente no estado TIME_WAIT se o SYN que chega tiver um número de seqüência "maior que" o número da seqüência final da encarnação anterior. As páginas 958 e 959 do TCPv2 discutem isso em mais detalhes. Isso exige que o servidor realize o close ativo, uma vez que o estado TIME_WAIT precisa existir na extremidade que recebe o próximo SYN. Essa capacidade é utilizada pelo comando `rsh`. A RFC 1185 (Jacobson, Braden e Zhang ,1990) discute algumas armadilhas ao fazer isso.

2.8 Estabelecimento e término de associação SCTP

O SCTP é orientado a conexão como ocorre com o TCP, assim ele também tem handshakes de estabelecimento e término de associação. Entretanto, os handshakes de SCTP são diferentes dos de TCP, portanto, vamos descrevê-los aqui.

Handshake de quatro vias

O cenário a seguir, semelhante ao TCP, ocorre quando uma associação SCTP é estabelecida:

1. O servidor deve estar preparado para aceitar uma associação entrante. Essa preparação normalmente ocorre chamando `socket`, `bind` e `listen` e é denominada *open passivo.*
2. O cliente emite um *open ativo* chamando `connect` ou enviando uma mensagem, que abre implicitamente a associação. Isso faz com que o SCTP cliente envie uma mensagem INIT ("initialization") para informar ao servidor a lista de endereços IP do cliente, o número de seqüência inicial, o tag de iniciação para identificar todos os pacotes nessa associação, o número de fluxos de saída que o cliente está solicitando e o número de fluxos de entrada que o cliente pode suportar.
3. O servidor reconhece a mensagem INIT do cliente com uma mensagem INIT-ACK, que contém a lista de endereços IP do servidor, o número de seqüência inicial, o tag de iniciação, o número de fluxos de saída que o servidor está solicitando, o número de fluxos de entrada que o servidor pode suportar e um cookie de estado. O cookie de estado contém toda a informação de estado que o servidor precisa para assegurar que a associação é válida, e é digitalmente assinado para assegurar sua validade.
4. O cliente ecoa o cookie de estado do servidor com uma mensagem COOKIE-ECHO. Essa mensagem também pode conter dados de usuário empacotados dentro do mesmo pacote.
5. O servidor reconhece que o cookie é correto e que a associação foi estabelecida com uma mensagem COOKIE-ACK. Essa mensagem também pode conter dados de usuário empacotados no mesmo pacote.

O número mínimo de pacotes requeridos para essa troca é quatro; portanto, esse processo é chamado *handshake de quatro vias* de SCTP. Uma descrição dos quatro segmentos é apresentada na Figura 2.6.

O handshake de quatro vias do SCTP é de várias maneiras semelhante ao handshake de três vias do TCP, exceto quanto à geração de cookie, que é parte integral do SCTP. A mensagem INIT transmite (junto com seus vários parâmetros) um tag de verificação, *Ta*, e um número de seqüência inicial, *J*. O tag *Ta* deve estar presente em cada pacote enviado pelo peer durante a vida da associação. O número de seqüência inicial é utilizado como o número de seqüência de partida para mensagens DATA denominadas *DATA chunks* ("blocos de dados"). O peer também escolhe um tag de verificação, *Tz*, que deve estar presente em cada um dos seus pacotes durante a vida da associação. Junto com o tag de verificação e o número de seqüência inicial, *K*, o receptor do INIT também envia um cookie, *C*. O cookie contém todo o estado necessário para configurar a associação SCTP, de modo que a pilha SCTP do servidor não precise manter as informações sobre o cliente que se associa. Mais detalhes sobre a configuração da associação SCTP podem ser encontrados no Capítulo 4 de Stewart e Xie (2001).

Na conclusão do handshake de quatro vias, cada extremidade escolhe um endereço primário de destino. O endereço primário de destino é utilizado como o destino-padrão ao qual os dados serão enviados na ausência de uma falha de rede.

O handshake de quatro vias é utilizado no SCTP para evitar uma forma de ataque de negação de serviço que discutiremos na Seção 4.5.

> O handshake de quatro vias do SCTP utilizando cookies formaliza um método de proteção contra esse ataque. Muitas implementações TCP utilizam um método semelhante; a grande diferença é que no TCP o estado de cookie deve ser codificado no número de seqüência inicial, que é apenas de 32 bits. O SCTP fornece um campo de comprimento arbitrário e requer segurança criptográfica para evitar ataques.

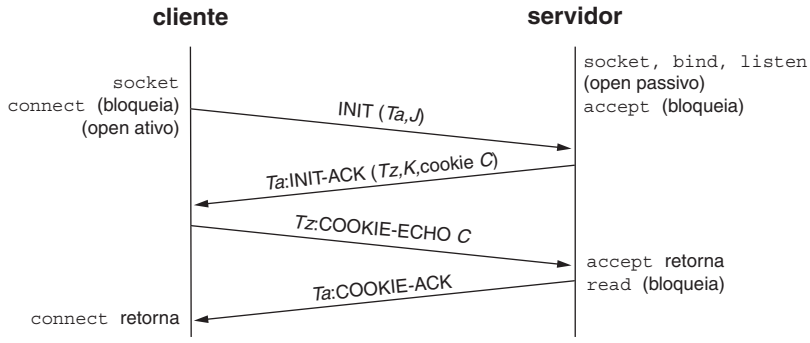


Figura 2.6 Handshake de quatro vias de SCTP.

Término de associação

Diferentemente do TCP, o SCTP não permite uma a associação "meio fechada" ("half-closed"). Quando uma extremidade desativa uma associação, a outra extremidade deve parar de enviar novos dados. O receptor da solicitação de desativação envia os dados que estavam enfileirados, se houver algum, e então completa a desativação. Mostramos essa troca na Figura 2.7.

O SCTP não tem um estado TIME_WAIT como o TCP, graças à utilização de tags de verificação. Todos os fragmentos são marcados com o tag trocado nos fragmentos INIT; um fragmento de uma conexão antiga chegará com um tag incorreto. Portanto, em vez de manter uma conexão completa em TIME_WAIT, o SCTP posiciona os valores dos tags de verificação em TIME_WAIT.

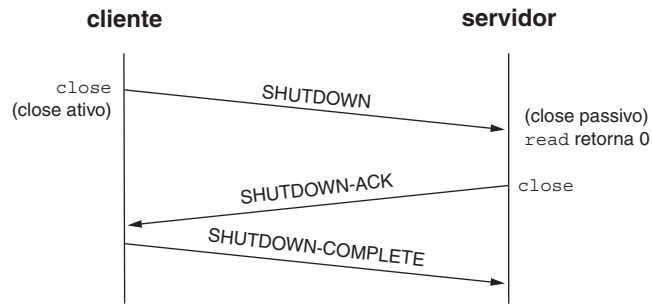


Figura 2.7 Pacotes trocados quando uma associação SCTP está fechada.

Diagrama de transição de estado SCTP

A operação de SCTP relacionada ao estabelecimento e término de uma associação pode ser especificada com *um diagrama de transição de estado*. Mostramos isso na Figura 2.8.

Como na Figura 2.4, as transições de um estado para outro na máquina de estado são ditadas pelas regras de SCTP, com base no estado atual e no fragmento recebido nesse estado. Por exemplo, se uma aplicação realizar um open ativo no estado CLOSED, o SCTP envia um INIT e o novo estado torna-se COOKIE-WAIT. Se, em seguida, o SCTP receber um INIT ACK, ele envia um COOKIE ECHO e o novo estado torna-se COOKIE-ECHOED. Se então o SCTP receber um COOKIE ACK, ele passa para o estado ESTABLISHED. É nesse estado final que ocorre a maior parte da transferência de dados, embora fragmentos de DATA possam ser enviados via *piggyback* juntamente com COOKIE ECHO e em fragmentos de COOKIE ACK.

A duas setas partindo do estado ESTABLISHED lidam com o término de uma associação. Se uma aplicação chamar `close` antes de receber SHUTDOWN (um close ativo), a transição ocorre para o estado SHUTDOWN-PENDING. Entretanto, se uma aplicação receber um SHUTDOWN no estado ESTABLISHED (um close passivo), a transição ocorre para o estado SHUTDOWN-RECEIVED.

Observando os pacotes

A Figura 2.9 mostra a troca real de pacotes que acontece para uma associação SCTP de exemplo: o estabelecimento da associação, a transferência de dados e o término da associação. Também mostramos os estados SCTP pelos quais cada extremidade passa.

Nesse exemplo, o cliente envia por *piggyback* o seu primeiro fragmento de dados junto com COOKIE ECHO e o servidor responde com dados em COOKIE ACK. Em geral, o COOKIE ECHO terá um ou mais fragmentos de DATA empacotados quando a aplicação utiliza o estilo de interface de um para muitos (discutiremos os estilos de interface de um para um e de um para muitos na Seção 9.2).

A unidade de informação dentro de um pacote SCTP é um "fragmento" ("chunk"). Um "fragmento" é autodescritivo e contém o tipo do fragmento, flags e o comprimento do fragmento. Essa abordagem facilita o empacotamento de fragmentos simplesmente combinando múltiplos fragmentos em um pacote SCTP de saída (detalhes sobre o empacotamento de fragmentos e os procedimentos normais de transmissão de dados podem ser encontrados no Capítulo 5 de (Stewart e Xie [2001]).

Opções SCTP

O SCTP utiliza parâmetros e fragmentos para facilitar os recursos opcionais. Novos recursos são definidos adicionando um desses dois itens e permitindo que regras de processamento

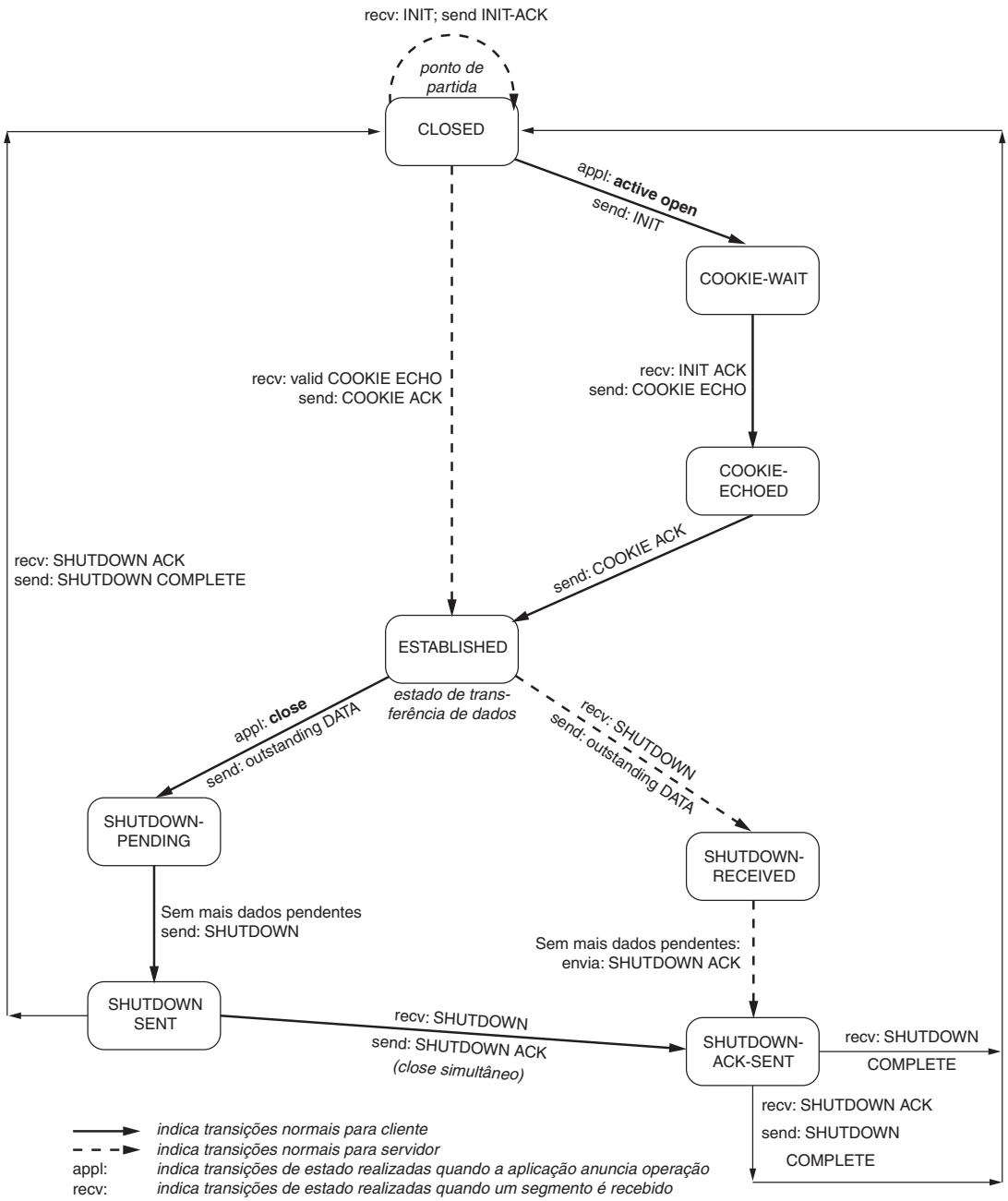


Figura 2.8 Diagrama de transição de estado SCTP.

SCTP normais informem parâmetros e fragmentos desconhecidos. Os dois bits superiores tanto do espaço de parâmetro como do espaço de fragmento ditam o que um receptor SCTP deve fazer com um parâmetro ou fragmento desconhecido (mais detalhes podem ser encontrados na Seção 3.1 de Stewart e Xie [2001]).

Atualmente, duas extensões para o SCTP estão em desenvolvimento:

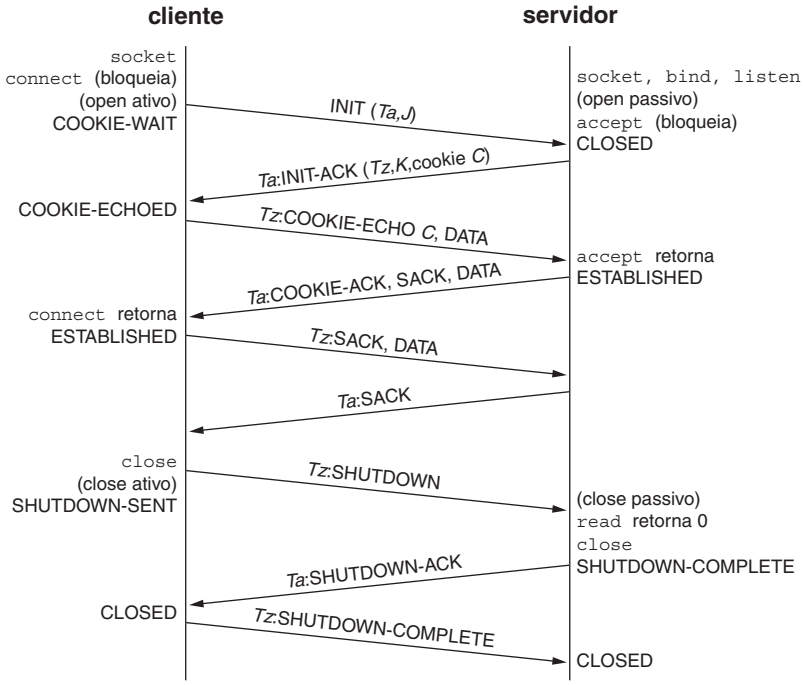


Figura 2.9 A troca de pacotes para a associação SCTP.

1. A extensão de endereço dinâmico, que permite que as extremidades SCTP cooperadoras adicionem e removam dinamicamente endereços IP de uma associação existente.
2. A extensão de confiabilidade parcial, que permite que extremidades SCTP cooperantes, sob orientação da aplicação, limitem a retransmissão de dados. Quando uma mensagem torna-se muito antiga para ser enviada (de acordo com a orientação da aplicação), ela será pulada e não mais enviada ao peer. Isso significa que não há garantias de que todos os dados chegarão à outra extremidade da associação.

2.9 Números de porta

A qualquer momento, múltiplos processos podem utilizar qualquer transporte dado: UDP, SCTP ou TCP. Todas as três camadas de transporte utilizam *números de porta,* um inteiro de 16 bits, para diferenciar entre esses processos.

Quando um cliente quer entrar em contato com um servidor, deve identificar o servidor com o qual quer se comunicar. TCP, UDP e SCTP definem um grupo de *portas bem-conhecidas* para identificar serviços bem-conhecidos. Por exemplo, cada implementação TCP/IP que suporta FTP atribui a porta bem-conhecida 21 (decimal) ao servidor FTP. Servidores TFTP (Trivial File Transfer Protocol) são atribuídos à porta UDP 69.

Os clientes, por outro lado, normalmente utilizam *portas efêmeras*, isto é, portas de vida curta. Esses números de porta costumam ser atribuídos automaticamente pelo protocolo de transporte ao cliente. Os clientes normalmente não se preocupam com o valor da porta efêmera; o cliente apenas precisa ter certeza de que a porta efêmera é única no host cliente. O código do protocolo de transporte garante essa unicidade.

A *Internet Assigned Numbers Authority* (IANA) mantém uma lista de atribuições de número de portas. Antigamente, as atribuições eram publicadas como RFCs; a RFC 1700 (Rey-

nolds e Postel, 1994) é a última nessa série. A RFC 3232 (Reynolds, 2002) fornece a localização do banco de dados on-line que substituiu a RFC 1700: `http://www.iana.org/`. Os números de porta são divididos em três intervalos:

1. As *portas bem-conhecidas:* 0 a 1023. Esses números de porta são controlados e atribuídos pela IANA. Quando possível, a mesma porta é atribuída a um dado serviço para TCP, UDP e SCTP. Por exemplo, a porta 80 é atribuída a um servidor Web, tanto para TCP como para UDP, embora todas as implementações atualmente utilizem somente o TCP.

   Quando a porta 80 foi atribuída, o SCTP ainda não existia. Novas atribuições de porta são feitas para todos os três protocolos e a RFC 2960 declara que todos os números de porta existentes de TCP devem ser válidos para o mesmo serviço que utiliza o SCTP.

2. As *portas registradas:* 1024 a 49151. Essas portas não são controladas pela IANA, mas ela registra e lista seu uso como uma conveniência à comunidade. Quando possível, a mesma porta é atribuída a um dado serviço tanto para TCP como para UDP. Por exemplo, as portas 6000 a 6063 são atribuídas a um servidor X Window para os dois protocolos, embora todas as implementações atualmente utilizem somente o TCP. O limite superior de 49151 para essas portas foi introduzido para permitir um intervalo de portas efêmeras; a RFC 1700 (Reynolds e Postel, 1994) lista o intervalo superior como 65535.

3. As portas *dinâmicas* ou *privadas*: 49152 a 65535. A IANA não informa nada sobre essas portas. Elas são o que chamamos portas *efêmeras*. (O número mágico 49152 é três quartos de 65536.)

A Figura 2.10 mostra essa divisão, junto com a alocação comum dos números de porta. Observamos os seguintes pontos nessa figura:

- Os sistemas Unix têm um conceito de *porta reservada*, que é qualquer porta menor que 1024. Essas portas podem ser atribuídas apenas a um soquete por um processo com privilégio apropriado. Todas as portas IANA bem-conhecidas são reservadas; conseqüentemente, o servidor que aloca essa porta (como o servidor FTP) deve ter privilégios de superusuário quando inicia.
- Historicamente, as implementações derivadas de Berkeley (iniciando com 4.3BSD) alocavam portas efêmeras no intervalo de 1024-5000. Isso era adequado no início da década de 1980, mas hoje é fácil encontrar um host que suporta mais de 3977 conexões a qualquer momento. Portanto, vários sistemas mais recentes alocam portas efêmeras de maneira dife-

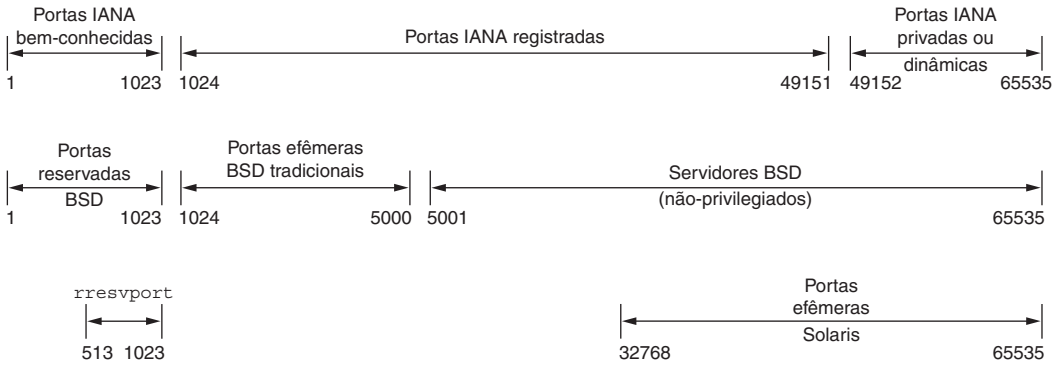


Figura 2.10 Alocação de números de porta.

rente para fornecer mais portas efêmeras, utilizando um intervalo de portas efêmeras definido pela IANA ou um intervalo maior (por exemplo, o Solaris mostrado na Figura 2.10).

> Como se veio a saber, o limite superior de 5.000 para portas efêmeras, que muitos sistemas mais antigos implementam, era um erro de digitação (Borman, 1997a) O limite deveria ter sido 50.000.

- Há alguns clientes (não servidores) que requerem uma porta reservada como parte da autenticação cliente/servidor: clientes `rlogin` e `rsh` são exemplos comuns. Esses clientes chamam a função de biblioteca `rresvport` para criar um soquete TCP e atribuir uma porta não-utilizada no intervalo 513-1023 ao soquete. Essa função normalmente tenta vincular a porta 1023 e, se isso falhar, ela tenta a 1022 e assim por diante, até que seja bem-sucedida ou falhe na porta 513.

  > Observe que as portas reservadas BSD e a função `rresvport` sobrepõem-se na metade superior das portas bem-conhecidas da IANA. Isso ocorre porque as portas bem-conhecidas da IANA paravam em 255. A RFC 1340 (uma "Assigned Numbers" RFC anterior), em 1992, começou a atribuir portas bem-conhecidas entre 256 e 1023. O documento "Assigned Numbers" anterior, a RFC 1060 de 1990, denominava as portas 256-1023 de *serviços padrão do Unix*. Há vários servidores derivados do Berkeley que selecionaram suas portas bem-conhecidas na década de 1980 iniciando em 512 (deixando as portas 256-511 intocadas). A função `rresvport` escolheu iniciar no topo do intervalo 512-1023 e trabalhar daí para baixo.

Par de soquetes

O par de soquetes para uma conexão TCP é uma quatro-tupla que define as duas extremidades da conexão: o endereço IP local, a porta local, o endereço IP externo e a porta externa. Um par de soquetes identifica cada conexão TCP de maneira única em uma rede. Para o SCTP, uma associação é identificada por um conjunto de endereços IP local, uma porta local, um conjunto de endereços IP externos e uma porta externa. Na sua forma mais simples, em que nenhuma extremidade é multihomed, isso resulta no mesmo par de soquetes de quatro-tupla sendo utilizado com o TCP. Entretanto, quando as duas extremidades de uma associação são multihomed, múltiplos conjuntos de quatro-tupla (com diferentes endereços IP, mas com os mesmos números de porta) podem então identificar a mesma associação.

Os dois valores que identificam cada extremidade, um endereço IP e um número de porta, são freqüentemente chamados *soquete*.

Podemos estender o conceito de um par de soquetes para o UDP, mesmo que o UDP seja sem conexão. Quando descrevermos as funções de soquete (`bind`, `connect`, `getpeername`, etc.), apontaremos quais funções especificam quais valores no par de soquetes. Por exemplo, `bind` deixa a aplicação especificar o endereço IP local e a porta local para soquetes TCP, UDP e SCTP.

2.10 Números da porta TCP e servidores concorrentes

Com um servidor concorrente, em que o loop do servidor principal gera um filho para tratar cada conexão nova, o que acontece se o filho continuar a utilizar o número de porta bem-conhecido ao servir uma longa solicitação? Vamos examinar uma seqüência típica. Primeiro, o servidor é iniciado no host `freebsd`, que é multihomed com os endereços IP 12.106.32.254 e 192.168.42.1, e o servidor faz um open passivo utilizando seu número de porta bem-conhecido (21, para esse exemplo). Agora, ele está esperando uma solicitação de cliente, que mostramos na Figura 2.11.

Utilizamos a notação {`*:21, *:*`} para indicar o par de soquetes do servidor. O servidor está esperando uma solicitação de conexão em qualquer interface local (o primeiro asterisco) na porta 21. O endereço IP externo e a porta externa não são especificados e os denotamos como * : *. Também chamamos isso de *soquete ouvinte*.

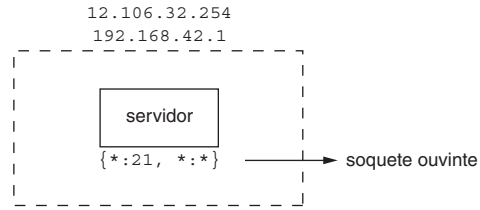


Figura 2.11 O servidor TCP com um open passivo na porta 21.

> Utilizamos dois-pontos para separar o endereço IP do número da porta, porque isso é o que o HTTP utiliza e é comumente visto em outras partes. O programa `netstat` utiliza um ponto para separar o endereço IP e a porta, mas isso às vezes torna-se confuso porque são utilizados pontos de fração decimal tanto nos nomes de domínio (`freebsd.unpbook.com.21`) como na notação de ponto decimal do IPv4 (`12.106.32.254.21`).

Quando especificamos o endereço IP local como um asterisco, ele é chamado caractere *curinga*. Se o host em que o servidor está em execução for multihomed (como nesse exemplo), o servidor pode especificar que quer aceitar somente conexões entrantes que chegam destinadas a uma interface local específica. Isso é uma escolha do tipo "uma ou qualquer" para o servidor. O servidor não pode especificar uma lista de múltiplos endereços. O endereço local curinga é a escolha "qualquer". Na Figura 1.9, o endereço curinga foi especificado configurando o endereço IP na estrutura de endereços de soquetes como `INADDR_ANY` antes de chamar `bind`.

Posteriormente, um cliente inicia no host com endereço IP 206.168.112.219 e executa um open ativo no endereço IP 12.106.32.254 do servidor. Supomos que a porta efêmera escolhida pelo TCP cliente seja 1500 nesse exemplo. Isso é mostrado na Figura 2.12. Abaixo do cliente mostramos seu par de soquetes.

Quando o servidor recebe e aceita a conexão do cliente, ele bifurca com `fork` para criar uma cópia de si próprio, deixando o filho tratar o cliente, como mostramos na Figura 2.13. (Descreveremos a função `fork` na Seção 4.7).

Nesse ponto, devemos fazer uma distinção entre o soquete ouvinte e o soquete conectado, ambos no host servidor. Observe que o soquete conectado utiliza a mesma porta local (21) que o soquete ouvinte. Também observe que no servidor multihomed o endereço local é preenchido para o soquete conectado (12.106.32.254) depois que a conexão é estabelecida.

O próximo passo assume que um outro processo cliente no host cliente solicita uma conexão no mesmo servidor. O código TCP no host cliente atribui ao novo soquete cliente um número de porta efêmero não utilizado, digamos 1501. Isso nos dá o cenário mostrado na Figura 2.14. No servidor, as duas conexões são distintas: o par de soquetes para a primeira conexão difere do par de soquetes para a segunda conexão porque o TCP do cliente escolhe uma porta não utilizada para a segunda conexão (1501).

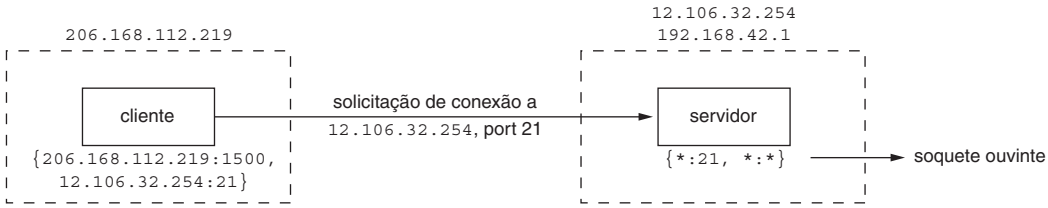


Figura 2.12 Solicitação de conexão do cliente ao servidor.

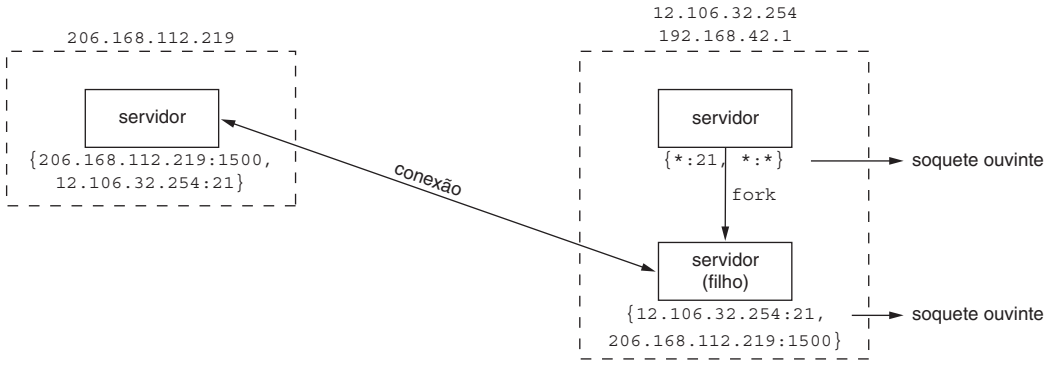


Figura 2.13 O servidor concorrente faz com que o filho trate do cliente.

Note, nesse exemplo, que o TCP não pode remover a multiplexação de segmentos entrantes examinando somente o número da porta de destino. O TCP deve examinar todos os quatro elementos no par de soquetes para determinar qual extremidade recebe um segmento entrante. Na Figura 2.14, temos três soquetes na mesma porta local (21). Se um segmento chegar de 206.168.112.219 porta 1500 destinado a 12.106.32.254 porta 21, ele é entregue ao primeiro filho. Se um segmento chegar de 206.168.112.219 porta 1501 destinado a 12.106.32.254 porta 21, ele é entregue ao segundo filho. Todos os outros segmentos TCP destinados à porta 21 são entregues ao servidor original com o soquete ouvinte.

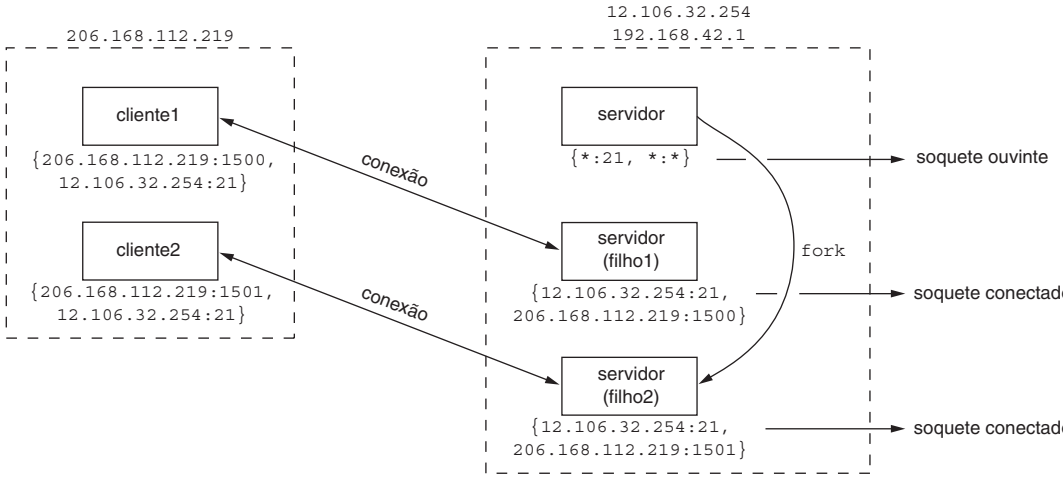


Figura 2.14 A segunda conexão de cliente no mesmo servidor.

2.11 Tamanhos e limitações de buffer

Certos limites afetam o tamanho de datagramas IP. Primeiro, descrevemos esses limites e então os agrupamos com referência à maneira como eles afetam os dados que uma aplicação pode transmitir.

- O tamanho máximo de um datagrama IPv4 é 65.535 bytes, incluindo o cabeçalho IPv4. Isso ocorre por causa do campo de 16 bits de comprimento total na Figura A.1.
- O tamanho máximo de um datagrama IPv6 é 65.575 bytes, incluindo o cabeçalho de 40 bytes do IPv6. Isso ocorre por causa do campo de 16 bits de comprimento do payload na Figura A.2. Observe que o campo de comprimento de payload do IPv6 não inclui o tamanho do cabeçalho IPv6, enquanto o campo de comprimento total do IPv4 inclui o tamanho de cabeçalho.

  O IPv6 tem uma opção jumbo payload, que estende o campo de comprimento de payload para 32 bits, mas essa opção é suportada somente em datalinks com um Maximum Transmission Unit (MTU) que exceda 65.535. (Isso é concebido para interconexões de host para host, como o HIPPI, que freqüentemente não tem nenhum MTU inerente.)
- Muitas redes têm um MTU que pode ser ditado pelo hardware. Por exemplo, o MTU da Ethernet é de 1.500 bytes. Outros datalinks, como links ponto a ponto que utilizam o Point-to-Point Protocol (PPP), têm um MTU configurável. Enlaces de SLIP mais antigos freqüentemente utilizavam um MTU de 1.006 ou 296 bytes.

  O MTU mínimo de enlace para o IPv4 é 68 bytes. Isso permite um cabeçalho IPv4 com tamanho máximo (20 bytes de cabeçalho fixo, 30 bytes de opções) e um fragmento de tamanho mínimo (o offset de fragmento ocorre em unidades de 8 bytes). O MTU mínimo de enlace para o IPv6 é de 1.280 bytes. O IPv6 pode executar em links com um MTU menor, mas requer fragmentação e remontagem específica do enlace para fazer com que o link pareça ter um MTU de pelo menos 1.280 bytes (RFC 2460 [Deering e Hinden, 1998]).
- O menor MTU no caminho entre dois hosts é chamado *MTU de caminho (path MTU)*. Atualmente, o MTU Ethernet de 1.500 bytes com freqüência é o MTU de caminho. O MTU do caminho não precisa ser o mesmo nas duas direções entre dois hosts quaisquer porque o roteamento na Internet muitas vezes é assimétrico (Paxson, 1996). Isto é, a rota de A para B pode diferir da rota de B para A.
- Quando um datagrama IP está para ser enviado para uma interface, se o seu tamanho exceder o MTU de enlace, a *fragmentação* é realizada tanto pelo IPv4 como pelo IPv6. Normalmente, os fragmentos não são *remontados* até alcançarem o destino final. Os hosts IPv4 realizam a fragmentação nos datagramas que eles geram e os roteadores IPv4 realizam a fragmentação nos datagramas que eles encaminham. Mas, com o IPv6, somente os hosts realizam a fragmentação nos datagramas que eles geram; os roteadores IPv6 não fragmentam datagramas que eles estão encaminhando.

  > Devemos ter cuidado com a nossa terminologia. Uma máquina rotulada como um roteador IPv6 pode de fato realizar a fragmentação, mas somente nos datagramas que o próprio roteador gera, nunca nos datagramas que ele está encaminhando. Quando essa máquina gera datagramas IPv6, ela na realidade está atuando como um host. Por exemplo, a maioria dos roteadores suporta o protocolo Telnet e esse protocolo é utilizado pelos administradores para configurar o roteador. Os datagramas IP gerados pelo servidor de Telnet do roteador são *gerados* pelo roteador, não encaminhados por ele.
  >
  > Você pode notar que há campos no cabeçalho IPv4 (Figura A.1) para tratar a fragmentação de IPv4, mas não há nenhum campo no cabeçalho IPv6 (Figura A.2) para fragmentação. Como a fragmentação é a exceção, em vez da regra, o IPv6 contém um cabeçalho de opção com as informações de fragmentação.
  >
  > Alguns firewalls, que normalmente atuam como roteadores, talvez remontem pacotes fragmentados para permitir a inspeção de conteúdo de todos os pacotes. Isso permite prevenir certos ataques com um custo de uma complexidade adicional no dispositivo de firewall. Também requer que o dispositivo de firewall faça parte do único caminho para a rede, reduzindo as oportunidades de redundância.

- Se o bit "don't fragment" (DF) for configurado no cabeçalho IPv4 (Figura A.1), ele especifica que esse datagrama não deve ser fragmentado, seja pelo host emissor ou por quaisquer roteadores. Um roteador que recebe um datagrama IPv4 com bit DF configurado cujo tamanho excede o MTU do enlace de saída gera uma mensagem de erro "destination unreachable, fragmentation needed but DF bit set" (destino inatingível, fragmentação necessária mas bit DF configurado) para o ICMPv4 (Figura A.15).

  Como os roteadores IPv6 não realizam fragmentação, há um bit DF implícito com cada datagrama IPv6. Quando um roteador IPv6 recebe um datagrama cujo tamanho excede o MTU do enlace de saída, ele gera uma mensagem de erro "packet too big" ("pacote muito grande") para o ICMPv6 (Figura A.16).

  O bit de DF do IPv4 e sua contraparte implícita do IPv6 pode ser utilizado para *descoberta do MTU do caminho* (RFC 1191 [Mogul e Deering, 1990] para IPv4 e RFC 1981 [McCann, Deering e Mogul, 1996] para IPv6). Por exemplo, se o TCP utilizar essa técnica com o IPv4, ele então envia todos os seus datagramas com o bit DF configurado. Se algum roteador intermediário retornar um erro ICMP "destination unreachable, fragmentation needed but DF bit set", o TCP diminui a quantidade de dados que ele envia por datagrama e retransmite. A descoberta do MTU do caminho é opcional com o IPv4, mas todas as implementações IPv6 ou suportam a descoberta do MTU do caminho ou sempre enviam utilizando o MTU mínimo.

  > A descoberta do MTU do caminho é problemática na Internet atualmente; muitos firewalls descartam todas as mensagens ICMP, incluindo a mensagem de fragmentação requerida, o que significa que o TCP nunca obtém o sinal que ele precisa para diminuir a quantidade de dados que está enviando. No momento em que este livro estava sendo escrito, havia um esforço inicial no IETF para definir um outro método para descoberta do MTU do caminho que não contasse com erros ICMP.

- O IPv4 e o IPv6 definem um *tamanho mínimo do buffer de remontagem*, o tamanho mínimo de datagrama que temos garantia de que qualquer implementação deve suportar. Para o IPv4, esse tamanho é de 576 bytes. O IPv6 aumenta esse valor para 1.500 bytes. Com o IPv4, por exemplo, não temos idéia se um dado destino pode ou não aceitar um datagrama de 577 bytes. Portanto, várias aplicações IPv4 que utilizam UDP (por exemplo, DNS, RIP, TFTP, BOOTP, SNMP) impedem que aplicações gerem datagramas IP e que excedam esse tamanho.
- O TCP tem um *tamanho máximo de segmento* (*maximum segment size* – MSS) que anuncia ao TCP peer a quantidade máxima de dados de TCP que o peer pode enviar por segmento. Vimos a opção MSS nos segmentos SYN na Figura 2.5. O objetivo do MSS é informar ao peer o valor real do tamanho do buffer de remontagem e tentar evitar a fragmentação. O MSS é freqüentemente configurado como o MTU da interface menos os tamanhos fixos dos cabeçalhos IP e TCP. Em uma Ethernet que utiliza o IPv4, esse valor seria 1.460 e, em uma Ethernet que utiliza o IPv6, seria 1.440. (O cabeçalho TCP tem 20 bytes para ambos, mas o cabeçalho IPv4 tem 20 bytes e o cabeçalho IPv6 tem 40 bytes.)

  O valor de MSS na opção TCP MSS é um campo de 16 bits, limitando o valor a 65.535. Isso é bom para o IPv4, uma vez que a quantidade máxima de dados de TCP em um datagrama IPv4 é 65.495 (65.535 menos os 20 bytes do cabeçalho IPv4 e menos os 20 bytes do cabeçalho TCP). Mas, com a opção jumbo payload do IPv6, é utilizada uma técnica diferente (RFC 2675 [Borman, Deering e Hinden, 1999]). Primeiro, a quantidade máxima de dados TCP em um datagrama IPv6 sem a opção jumbo payload é 65.515 (65.535 menos os 20 bytes do cabeçalho TCP). Portanto, o valor MSS de 65.535 é considerado um caso especial que designa "infinito". Esse valor é utilizado somente se a opção jumbo payload estiver sendo utilizada, o que requer um MTU que exceda 65.535. Se o TCP estiver utilizando a opção jumbo payload e receber um anúncio MSS de 65.535 do peer,

o limite nos tamanhos de datagrama que ele envia é simplesmente o MTU da interface. Se isso se revelar muito grande (isto é, há um link no caminho com um MTU menor), então a descoberta do MTU do caminho determinará o valor menor.

- O SCTP mantém um ponto de fragmentação com base no menor MTU do caminho encontrado para todos os endereços do peer. Esse menor tamanho de MTU é utilizado para dividir grandes mensagens de usuário em fragmentos menores que podem ser enviados em um datagrama IP. A opção de soquete SCTP_MAXSEG pode influenciar esse valor, permitindo ao usuário solicitar um ponto de fragmentação menor.

Saída TCP

Dados todos esses termos e definições, a Figura 2.15 mostra o que acontece quando uma aplicação grava dados em um soquete TCP.

Cada soquete TCP tem um buffer de envio, cujo tamanho podemos alterar \ com a opção de soquete `SO_SNDBUF` (Seção 7.5). Quando uma aplicação chama `write`, o kernel copia todos os dados do buffer da aplicação para o buffer de envio do soquete. Se não houver espaço suficiente no buffer do soquete para todos os dados da aplicação (ou o buffer da aplicação é maior que o buffer de envio do soquete, ou já existem dados no buffer de envio do soquete), o processo será colocado em repouso. Isso assume o padrão normal de um soquete bloqueador. (Discutiremos os soquetes não-bloqueadores no Capítulo 16.) O kernel não retornará de `write` até que o byte final no buffer da aplicação tenha sido copiado para o buffer de envio do soquete. Portanto, o retorno bem-sucedido de um `write` para um soquete TCP somente informa que podemos reutilizar nosso buffer de aplicação. Ele *não* informa se o TCP peer ou a aplicação peer recebeu os dados. (Discutiremos isso em mais detalhes com opção de soquete `SO_LINGER` na Seção 7.5.)

O TCP recebe os dados no buffer de envio do soquete e envia-os ao TCP peer com base em todas as regras de transmissão de dados TCP (Capítulos 19 e 20 do TCPv1). O TCP peer deve reconhecer os dados e, à medida que ACKs chegam do peer, somente então nosso TCP pode descartar os dados reconhecidos do buffer de envio do soquete. O TCP deve manter uma cópia dos nossos dados até que eles sejam reconhecidos pelo peer.

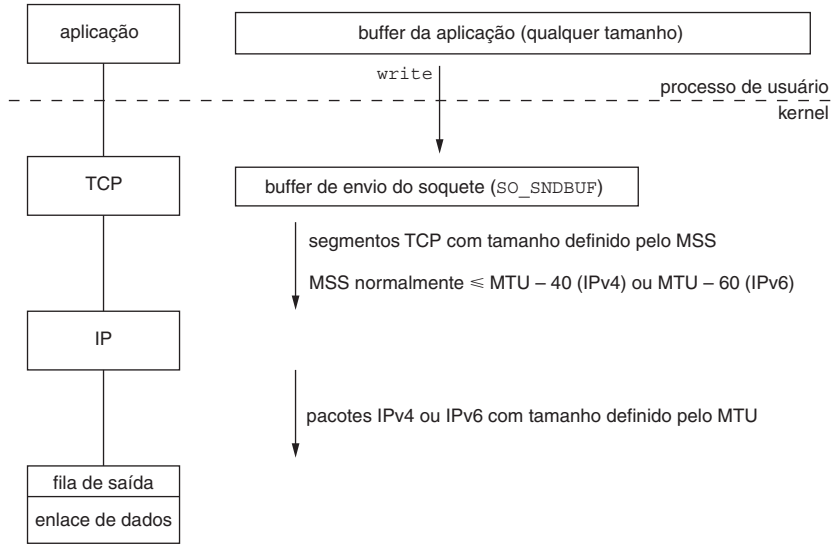


Figura 2.15 Os passos e os buffers envolvidos quando uma aplicação grava em um soquete TCP.

O TCP envia os dados ao IP em fragmentos de tamanho determinado pelo MSS ou em fragmentos menores, prefixando seu cabeçalho TCP para cada segmento, em que o MSS é o valor anunciado pelo peer, ou 536 se o peer não enviar uma opção MSS. O IP prefixa seu cabeçalho, pesquisa na tabela de roteamento o endereço IP de destino (a entrada correspondente na tabela de roteamento especifica a interface de saída) e passa o datagrama para o enlace de dados apropriado. O IP poderia realizar a fragmentação antes de passar o datagrama para o enlace de dados, mas, como dissemos anteriormente, um dos objetivos da opção MSS é tentar evitar fragmentação e as implementações mais recentes também utilizam a descoberta do MTU do caminho. Cada enlace de dados tem uma fila de saída, e se essa fila estiver cheia o pacote é descartado e um erro retorna para cima da pilha de protocolos: do enlace de dados para o IP e então do IP para o TCP. O TCP observará esse erro e tentará enviar o segmento mais tarde. A aplicação não é informada sobre essa condição transitória.

Saída UDP

A Figura 2.16 mostra o que acontece quando uma aplicação grava dados em um soquete UDP.

Dessa vez, mostramos o buffer de envio do soquete como uma caixa tracejada, pois ela realmente não existe. Um soquete UDP tem um tamanho de buffer de envio (que podemos alterar com a opção de soquete `SO_SNDBUF`, Seção 7.5), mas isso simplesmente é um limite superior quanto ao tamanho máximo do datagrama UDP que pode ser gravado no soquete. Se uma aplicação gravar um datagrama maior que o tamanho do buffer de envio do soquete, `EMSGSIZE` é retornado. Como o UDP é não-confiável, ele não precisa manter uma cópia dos dados da aplicação nem precisa de um buffer de envio real. (Normalmente, os dados da aplicação são copiados para um buffer de kernel de alguma forma à medida que ele passa pela pilha de protocolos, mas essa cópia é descartada pela camada de enlace de dados depois que os dados são transmitidos.)

O UDP simplesmente prefixa o seu cabeçalho de 8 bytes e passa o datagrama para o IP. O IPv4 ou o IPv6 prefixa seu cabeçalho, determina a interface de saída realizando a função de roteamento e então ou adiciona o datagrama à fila de saída de enlace de dados (se ele couber dentro do MTU) ou fragmenta o datagrama e adiciona cada fragmento à fila de saída de enla-

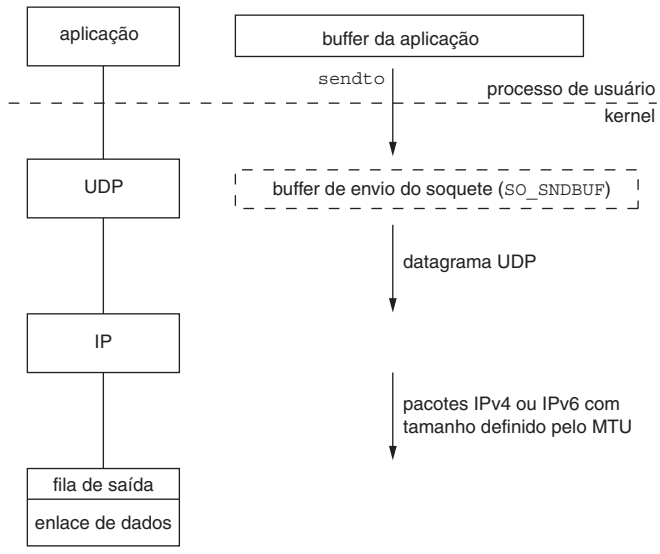


Figura 2.16 Os passos e os buffers envolvidos quando uma aplicação grava em um soquete UDP.

ce de dados. Se uma aplicação UDP enviar grandes datagramas (por exemplo, datagramas de 2.000 bytes), há uma probabilidade muito maior de fragmentação do que com o TCP, pois o TCP divide os dados da aplicação em fragmentos de tamanho definido pelo MSS, algo que não tem nenhuma contraparte no UDP.

O retorno bem-sucedido de um `write` para um soquete UDP informa que o datagrama ou todos os fragmentos deste foram adicionados à fila de saída de enlace de dados. Se não houver espaço na fila para o datagrama ou para um dos seus fragmentos, freqüentemente retorna `ENOBUFS` à aplicação.

> Infelizmente, algumas implementações não retornam esse erro, não fornecendo nenhuma indicação à aplicação de que o datagrama foi descartado sem mesmo ter sido transmitido.

Saída de SCTP

A Figura 2.17 mostra o que acontece quando uma aplicação grava dados em um soquete SCTP.

O SCTP, uma vez que é um protocolo confiável como TCP, possui um buffer de envio. Como ocorre com o TCP, uma aplicação pode alterar o tamanho desse buffer com a opção de soquete `SO_SNDBUF` (Seção 7.5). Quando a aplicação chama `write`, o kernel copia todos os dados do buffer da aplicação para o buffer de envio do soquete. Se não houver espaço suficiente no buffer do soquete para todos os dados da aplicação (o buffer de aplicação é maior que o buffer de envio do soquete ou já há dados no buffer de envio do soquete), o processo é colocado em repouso. Esse repouso assume o padrão normal de um soquete de bloqueio. (Discutiremos os soquetes não-bloqueadores no Capítulo 16.) O kernel não retornará de `write` até que o byte final do buffer da aplicação tenha sido copiado para o buffer de envio do soquete. Portanto, o retorno bem-sucedido de um `write` para um soquete SCTP somente informa ao remetente que ele pode reutilizar o buffer de aplicação. Ele *não* informa se um dos SCTP peer ou a aplicação peer recebeu os dados.

O SCTP recebe os dados no buffer de envio do soquete e envia-os ao SCTP peer com base nas regras de transmissão de dados do SCTP (para detalhes sobre a transferência de dados, veja o Capítulo 5 de Stewart e Xie [2001]). O SCTP que envia deve esperar um SACK em que

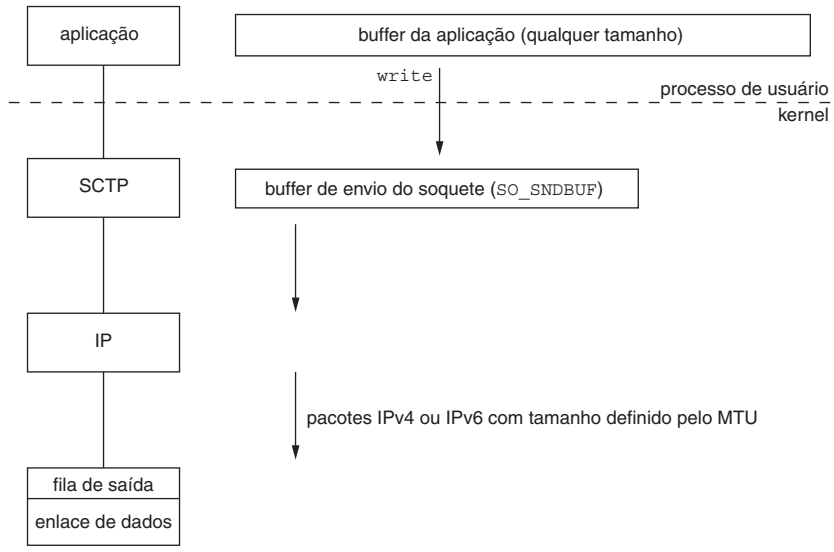


Figura 2.17 Os passos e os buffers envolvidos quando uma aplicação grava em um soquete SCTP.

o ponto cumulativo de reconhecimento passa os dados enviados antes que os eles possam ser removidos do buffer do soquete.

2.12 Serviços Internet-padrão

A Figura 2.18 lista os vários serviços-padrão fornecidos pela maioria das implementações TCP/IP. Observe que todos utilizam tanto o TCP como o UDP e que o número da porta é o mesmo para os dois protocolos.

Freqüentemente esses serviços são fornecidos pelo daemon `inetd` nos hosts Unix (Seção 13.5). Esses serviços-padrão fornecem um recurso de teste fácil utilizando o cliente Telnet-padrão. Por exemplo, os exemplos a seguir testam os servidores de data/hora e eco:

```
aix % telnet freebsd daytime
Trying 12.106.32.254...                 saída gerada pelo cliente de Telnet
Connected to freebsd.unpbook.com.       saída gerada pelo cliente de Telnet
Escape character is '^]'.               saída gerada pelo cliente de Telnet
Mon Jul 28 11:56:22 2003                saída gerada pelo servidor de data/hora
Connection closed by foreign host.      saída gerada pelo cliente de Telnet
                                          (o servidor fecha a conexão)

aix % telnet freebsd echo
Trying 12.106.32.254...                 saída gerada pelo cliente de Telnet
Connected to freebsd.unpbook.com.       saída gerada pelo cliente de Telnet
Escape character is '^]'.               saída gerada pelo cliente de Telnet
hello, world                            digitamos isso
hello, world                            e isso é ecoado de volta pelo servidor
^]                                      digitamos control e um colchete à direita para
                                          conversar com o cliente de Telnet
telnet> quit                            e dizemos ao cliente que concluímos
Connection closed.                      o cliente fecha a conexão desta vez
```

Nesses dois exemplos, digitamos o nome do host e o nome do serviço (`daytime` e `echo`). Os nomes desses serviços são mapeados para os números de porta mostrados na Figura 2.18 pelo arquivo `/etc/services`, como descreveremos na Seção 11.5.

Observe que, quando nos conectamos ao servidor `daytime`, o servidor realiza o close ativo, enquanto com o servidor `echo` é o cliente que realiza o close ativo. Lembre-se, da Figura 2.4, que a extremidade que realiza o close ativo é a que passa pelo estado TIME_WAIT.

Esses "serviços simples" são freqüentemente desativados por default nos sistemas modernos devido a ataques de negação de serviço e outros tipos de ataques de utilização de recursos contra eles.

Nome	porta TCP	porta UDP	RFC	Descrição
`echo`	7	7	862	O servidor retorna qualquer coisa que o cliente envia.
`discard`	9	9	863	O servidor descarta qualquer coisa que o cliente envia.
`daytime`	13	13	867	O servidor retorna a data e a hora em um formato legível por humanos.
`chargen`	19	19	864	O servidor TCP envia um fluxo ininterrupto de caracteres, até que a conexão seja terminada pelo cliente. O servidor UDP envia um datagrama contendo um número aleatório de caracteres (entre 0 e 512) todas as vezes que o cliente envia um datagrama.
`time`	37	37	868	O servidor retorna a data/hora como um número binário de 32 bits. Esse número representa o número de segundos a partir da meia-noite de 1º janeiro de 1900, UTC.

Figura 2.18 Serviços TCP/IP-padrão fornecidos pela maioria das implementações.

2.13 Uso de protocolo por aplicações Internet comuns

A Figura 2.19 resume o uso de protocolo por várias aplicações Internet comuns.

As duas primeiras aplicações, `ping` e `traceroute`, são aplicações de diagnóstico que utilizam o ICMP. `traceroute` constrói seus próprios pacotes UDP para envio e lê respostas de ICMP.

Os três protocolos de roteamento populares demonstram a variedade de protocolos de transporte utilizada pelos protocolos de roteamento. O OSPF utiliza IP diretamente, empregando um soquete bruto, enquanto o RIP utiliza UDP; e o BGP utiliza TCP.

As cinco aplicações seguintes são baseadas em UDP, seguidas por sete aplicações TCP e quatro que utilizam tanto o UDP como o TCP. As cinco últimas são aplicações de telefonia de IP que utilizam SCTP exclusivamente ou UDP, TCP ou SCTP opcionalmente.

Aplicação	IP	ICMP	UDP	TCP	SCTP
ping		•			
traceroute		•	•		
OSPF (protocolo de roteamento)	•				
RIP (protocolo de roteamento)			•		
BGP (protocolo de roteamento)				•	
BOOTP (protocolo de bootstrap)			•		
DHCP (protocolo de bootstrap)			•		
NTP (protocolo de data/hora)			•		
TFTP			•		
SNMP (gerenciamento de rede)			•		
SMTP (correio eletrônico)				•	
Telnet (login remoto)				•	
SSH (login remoto seguro)				•	
FTP				•	
HTTP (a Web)				•	
NNTP (notícias de rede)				•	
LPR (impressão remota)				•	
DNS			•	•	
NFS (sistema de arquivos de rede)			•	•	
Sun RPC			•	•	
DCE RPC			•	•	
IUA (ISDN sobre IP)					•
M2UA, M3UA (sinalização de telefonia SS7)					•
H.248 (controle de gateway de mídia)			•	•	•
H.323 (telefonia de IP)			•	•	•
SIP (telefonia de IP)			•	•	•

Figura 2.19 O uso de protocolo por várias aplicações Internet comuns.

2.14 Resumo

O UDP é um protocolo sem conexão, não-confiável e simples, enquanto o TCP é um protocolo orientado a conexão, confiável e complexo. O SCTP combina alguns recursos desses dois protocolos, fornecendo outros além daqueles encontrados no TCP. Embora a maioria das aplicações na Internet utilize TCP (Web, Telnet, FTP e correio eletrônico), todas necessitam de três camadas de transporte. Na Seção 22.4, discutiremos as razões em escolher UDP em vez de TCP. Na Seção 23.12, discutiremos as razões em escolher SCTP em vez de TCP.

O TCP estabelece conexões utilizando um handshake de três vias e termina conexões utilizando uma troca de quatro pacotes. Quando uma conexão TCP é estabelecida, ela passa do estado CLOSED para o estado ESTABLISHED, e quando é terminada, volta ao estado CLOSED. Há 11 estados em que uma conexão TCP pode estar e um diagrama de transição de estado fornece as regras sobre como passar de um estado a outro. Entender esse diagrama é essencial para diagnosticar problemas utilizando o comando `netstat` e entender o que acontece quando uma aplicação chama funções como `connect`, `accept` e `close`.

O estado TIME_WAIT do TCP é uma fonte ininterrupta de confusão para programadores de rede. Esse estado existe para implementar o término da conexão full-duplex do TCP (isto é, tratar a perda do ACK) e permitir que segmentos duplicados antigos expirem na rede.

O SCTP estabelece uma associação utilizando um handshake de quatro vias e termina as conexões utilizando uma troca de três pacotes. Quando uma associação SCTP é estabelecida, ela passa do estado CLOSED para o estado ESTABLISHED, e quando é terminada, volta ao estado CLOSED. Há oito estados em que uma associação SCTP pode estar e um diagrama de transição de estado fornece as regras sobre como passar de um estado a outro. O SCTP não precisa do estado TIME_WAIT como o TCP devido à utilização de tags de verificação.

Exercícios

2.1 Mencionamos as versões 4 e 6 do IP. O que aconteceu com a versão 5 e quais eram as versões 0, 1, 2 e 3? (*Dica:* Encontre o registro "Internet Protocol" da IANA. Sinta-se livre para pular para a solução se você não puder visitar `http://www.iana.org`.)

2.2 Onde você pesquisaria para encontrar informações adicionais sobre o protocolo ao qual é atribuída a versão 5 do IP?

2.3 Com a Figura 2.15, dissemos que o TCP assume um MSS de 536 se ele não receber uma opção MSS do peer. Por que esse valor é utilizado?

2.4 Desenhe uma figura como a Figura 2.5 para o cliente/servidor de data/hora no Capítulo 1, supondo que o servidor retorna os 26 bytes de dados em um único segmento TCP.

2.5 Uma conexão é estabelecida entre um host em uma Ethernet, cujo TCP anuncia um MSS de 1.460, e um host em uma Token ring, cujo TCP anuncia um MSS de 4.096. Nenhum host implementa a descoberta do MTU do caminho. Observando os pacotes, nunca vemos mais de 1.460 bytes de dados em uma das direções. Por quê?

2.6 Na Figura 2.19, dissemos que o OSPF utiliza o IP diretamente. Qual é o valor do campo de protocolo no cabeçalho IPv4 (Figura A.1) desses datagramas de OSPF?

2.7 Ao discutir a saída de SCTP, dissemos que o remetente de SCTP deve esperar pelo ponto de reconhecimento cumulativo para passar os dados enviados antes que eles possam ser liberados do buffer do soquete. Se um reconhecimento seletivo mostrar que os dados são reconhecidos além do ponto de reconhecimento cumulativo, por que os dados não podem ser liberados?

PARTE II

Soquetes Elementares

Capítulo 3

Introdução a Soquetes

3.1 Visão geral

Este capítulo inicia a descrição da API de soquetes. Começaremos com as estruturas de endereço de soquete, que serão encontradas em quase todos os exemplos neste capítulo. Essas estruturas podem ser passadas em duas direções: do processo para o kernel e do kernel para o processo. O último caso é um exemplo de argumento de valor-resultado; encontraremos outros exemplos desses argumentos por todo o capítulo.

As funções de conversão de endereço convertem entre uma representação de texto de um endereço e o valor binário que é colocado em uma estrutura de endereço de soquete. A maior parte do código IPv4 existente utiliza `inet_addr` e `inet_ntoa`, mas duas novas funções, `inet_pton` e `inet_ntop`, tratam tanto o IPv4 como o IPv6.

Um dos problemas com essas funções de conversão de endereços é que elas são dependentes do tipo de endereço sendo convertido: IPv4 ou IPv6. Desenvolveremos um conjunto de funções cujos nomes iniciam com `sock_` e que funcionam com as estruturas de endereço de soquete de uma maneira independente do protocolo. Utilizaremos essas funções por todo o capítulo para tornar nosso código independente do protocolo.

3.2 Estruturas de endereço de soquete

A maioria das funções de soquete requer um ponteiro para uma estrutura de endereço de soquete como um argumento. Cada conjunto de protocolos suportado define sua própria estrutura de endereço de soquete. Os nomes dessas estruturas iniciam com `sockaddr_` e terminam com um sufixo único para cada conjunto de protocolos.

Estrutura de endereço de soquete IPv4

Uma estrutura de endereço de soquete IPv4, comumente chamada "estrutura de endereço de soquete Internet", é chamada `sockaddr_in` e definida incluindo-se o cabeçalho `<netinet/in.h>`. A Figura 3.1 mostra a definição POSIX.

```
struct in_addr {
   in_addr_t   s_addr;         /* endereço IPv4 de 32 bits */
                               /* ordenado por byte de rede */
};

struct sockaddr_in {
   uint8_t        sin_len;     /* comprimento da estrutura (16) */
   sa_family_t    sin_family;  /* AF_INET */
   in_port_t      sin_port;    /* número da porta TCP ou UDP de 16 bits */
                               /* ordenado por byte de rede */
   struct in_addr sin_addr;    /* endereço IPv4 de 32 bits */
                               /* ordenado por byte de rede */
   char           sin_zero[8]; /* não-utilizado */
};
```

Figura 3.1 A estrutura de endereço de soquete de Internet (IPv4): `sockaddr_in`.

Há vários pontos que precisamos descrever sobre as estruturas de endereço de soquete em geral utilizando este exemplo:

- O membro length, `sin_len`, foi adicionado com o 4.3BSD-Reno, quando o suporte a protocolos OSI foi adicionado (Figura 1.15). Antes dessa distribuição, o primeiro membro era `sin_family`, que historicamente era um `unsigned short`. Nem todos os fornecedores suportam um campo de comprimento para estruturas de endereço de soquete e a especificação POSIX não requer esse membro. O tipo de dados que mostramos, `uint8_t`, é típico e os sistemas compatíveis com POSIX fornecem tipos de dados dessa forma (Figura 3.2). Ter um campo de comprimento simplifica o tratamento de estruturas de endereço de soquete de comprimento variável.
- Mesmo que o campo de comprimento esteja presente, nunca precisaremos configurá-lo e examiná-lo, exceto ao lidar com soquetes de roteamento (Capítulo 18). Ele é utilizado dentro do kernel pelas rotinas que lidam com as estruturas de endereço de soquete a partir de várias famílias de protocolos (por exemplo, o código da tabela de roteamento).

  As quatro funções de soquete que passam uma estrutura de endereço de soquete do processo para o kernel, `bind`, `connect`, `sendto` e `sendmsg`, atravessam a função `sockargs` nas implementações derivadas do Berkeley (página 452 do TCPv2). Essa função copia do processo a estrutura de endereço de soquete e configura explicitamente seu membro `sin_len` com o tamanho da estrutura que foi passada como um argumento para essas quatro funções. Todas as cinco funções de soquete que passam uma estrutura de endereço de soquete do kernel para o processo, `accept`, `recvfrom`, `recvmsg`, `getpeername` e `getsockname`, configuram o membro `sin_len` antes de retornar ao processo.

Tipo de dados	Descrição	Cabeçalho
`int8_t`	Inteiro de 8 bits com sinal	`<sys/types.h>`
`uint8_t`	Inteiro de 8 bits sem sinal	`<sys/types.h>`
`int16_t`	Inteiro de 16 bits com sinal	`<sys/types.h>`
`uint16_t`	Inteiro de 16 bits sem sinal	`<sys/types.h>`
`int32_t`	Inteiro de 32 bits com sinal	`<sys/types.h>`
`uint32_t`	Inteiro de 32 bits sem sinal	`<sys/types.h>`
`sa_family_t`	Família de endereços da estrutura de endereço de soquete	`<sys/socket.h>`
`socklen_t`	Comprimento da estrutura de endereço de soquete, normalmente `uint32_t`	`<sys/socket.h>`
`in_addr_t`	Endereço IPv4, normalmente `uint32_t`	`<netinet/in.h>`
`in_port_t`	Porta TCP ou UDP, normalmente `uint16_t`	`<netinet/in.h>`

Figura 3.2 Tipos de dados requeridos pela especificação POSIX.

Infelizmente, normalmente não há nenhum teste simples em tempo de compilação para determinar se uma implementação define um campo de comprimento para suas estruturas de endereço de soquete. No nosso código, testamos nossa própria constante `HAVE_SOCKADDR_SA_LEN` (Figura D.2), mas definir ou não essa constante requer tentar compilar um programa de teste simples que utilize esse membro de estrutura opcional e verificar se a compilação foi ou não bem-sucedida. Veremos, na Figura 3.4, que são requeridas as implementações IPv6 para definir `SIN6_LEN` se as estruturas de endereço de soquete tiverem um campo de comprimento. Algumas implementações IPv4 fornecem o campo de comprimento da estrutura de endereço de soquete para a aplicação com base em uma opção em tempo de compilação (por exemplo, `SOCKADDR_LEN`). Esse recurso fornece compatibilidade com programas mais antigos.

- A especificação POSIX requer somente três membros na estrutura: `sin_family`, `sin_addr` e `sin_port`. É aceitável que uma implementação compatível com POSIX defina membros adicionais de estrutura e isso é normal para uma estrutura de endereço de soquete Internet. Quase todas as implementações adicionam o membro `sin_zero` para que todas as estruturas de endereço de soquete tenham pelo menos 16 bytes de tamanho.
- Mostramos os tipos de dados POSIX para os membros `s_addr`, `sin_family` e `sin_port`. O tipo de dados `in_addr_t` deve ser um tipo inteiro sem sinal de pelo menos 32 bits, `in_port_t` deve ser um tipo inteiro sem sinal de pelo menos 16 bits e `sa_family_t` pode ser qualquer tipo inteiro sem sinal. Normalmente, o último é um inteiro sem sinal de 8 bits se a implementação suportar o campo de comprimento ou um inteiro de 16 bits sem sinal se o campo de comprimento não for suportado. A Figura 3.2 lista esses três tipos de dados definidos pelo POSIX, junto com alguns outros tipos de dados POSIX que encontraremos.
- Você também encontrará os tipos de dados `u_char`, `u_short`, `u_int` e `u_long`, todos sem sinal. A especificação POSIX define esses tipos de dados com uma observação de que eles são obsoletos. Eles são oferecidos por questão de retrocompatibilidade.
- Tanto o endereço IPv4 como o número da porta TCP ou UDP sempre são armazenados na estrutura na ordem de bytes de rede. Devemos estar cientes disso ao utilizar esses membros. Discutiremos a diferença entre a ordem de bytes do host e ordem de bytes da rede em mais detalhes na Seção 3.4.
- O endereço IPv4 de 32 bits pode ser acessado de duas maneiras diferentes. Por exemplo, se `serv` for definido como uma estrutura de endereço de soquete de Internet, então `serv.sin_addr` irá referenciar o endereço IPv4 de 32 bits como uma estrutura `in_addr`, enquanto `serv.sin_addr.s_addr` irá referenciar o mesmo endereço IPv4 de 32 bits como um `in_addr_t` (em geral, um inteiro de 32 bits sem sinal). Devemos ter certeza de que estamos referenciando o endereço IPv4 corretamente, sobretudo quando ele é utilizado como um argumento para uma função, porque os compiladores costumam passar estruturas de maneira diferente de como inteiros são passados.

  A razão pela qual o membro `sin_addr` é uma estrutura e não apenas uma `in_addr_t`, é histórica. As primeiras distribuições (4.2BSD) definiam a estrutura `in_addr` como uma `union` de várias estruturas, para permitir acesso a cada um dos 4 bytes e a ambos os valores de 16 bits contidos dentro do endereço IPv4 de 32 bits. Isso foi utilizado com os endereços das classes A, B e C para buscar os bytes apropriados do endereço. Mas, com o advento de sub-redes e então o desaparecimento de várias classes de endereço com endereçamento sem classe (Seção A.4), a necessidade de `union` desapareceu. Hoje, a maioria dos sistemas acabou com `union` e somente define `in_addr` como uma estrutura com um único membro `in_addr_t`.
- O membro `sin_zero` não é utilizado, mas sempre o configuramos como 0 ao preencher uma dessas estruturas. Por convenção, sempre configuramos a estrutura inteira como 0 antes de preenchê-la, não apenas o membro `sin_zero`.

Embora a maioria das utilizações da estrutura não exija que esse membro seja 0, ao vincular um endereço IPv4 não-curinga, esse membro devem ser 0 (páginas 731 e 732 do TCPv2).

- As estruturas de endereço de soquete são utilizadas somente em um dado host: a estrutura em si não é transmitida entre hosts diferentes, embora certos campos (por exemplo, o endereço IP e a porta) sejam utilizados para comunicação.

Estrutura de endereço de soquetes genérica

Uma estrutura de endereço de soquete *sempre* é passada por referência quando passada como um argumento para quaisquer funções de soquete. Mas, qualquer função de soquete que recebe um desses ponteiros como argumento deve considerar as estruturas de endereço de soquete de *qualquer* família de protocolo suportada.

Há um problema sobre como declarar o tipo de ponteiro que é passado. Com ANSI C, a solução é simples: `void *` é o tipo de ponteiro genérico. Mas as funções de soquete são anteriores ao ANSI C e a solução escolhida em 1982 foi definir uma estrutura de endereço de soquete genérica no cabeçalho `<sys/socket.h>`, que mostramos na Figura 3.3.

```
struct sockaddr {
uint8_t     sa_len;
sa_family_t sa_family;          /* família de endereços: AF_xxx valor */
char        sa_data[14];        /* endereço específico de protocolo */
};
```

Figura 3.3 A estrutura de endereço de soquete genérica: `sockaddr`.

As funções de soquete são então definidas como recebendo um ponteiro para a estrutura de endereço de soquete genérica, como mostrado aqui no protótipo de função ANSI C para a função `bind`:

```
int bind(int, struct sockaddr *, socklen_t);
```

Isso requer que quaisquer chamadas a essas funções devem adaptar o ponteiro da estrutura de endereço de soquete específica de protocolo em um ponteiro para uma estrutura de endereço de soquete genérica. Por exemplo,

```
struct sockaddr_in serv;       /* estrutura de endereço de soquete IPv4 */

/* preenche serv{} */

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

Se omitirmos a adaptação "`(struct sockaddr *)`" o compilador C irá gerar um aviso na forma "warning: passing arg 2 of 'bind' from incompatible pointer type" ("aviso: passando arg 2 de 'bind' a partir de tipo de ponteiro incompatível"), assumindo que os cabeçalhos do sistema têm um protótipo ANSI C para a função `bind`.

Do ponto de vista do programador da aplicação, a única utilização dessas estruturas de endereço de soquete genéricas é adaptar os ponteiros para estruturas específicas de protocolo.

> Lembre-se da Seção 1.2, quando no nosso cabeçalho `unp.h` definimos `SA` como a string "`struct sockaddr`" somente para encurtar o código que precisamos escrever para fazer a adaptação desses ponteiros.
>
> Da perspectiva do kernel, uma outra razão para utilizar ponteiros para estruturas de endereço de soquete genéricas como argumentos é que o kernel deve receber o ponteiro do chamador, adaptá-lo para uma `struct sockaddr *` e então examinar o valor de `sa_family` para determinar o tipo da estrutura. Mas, da perspectiva do programador da aplicação, seria mais simples se o tipo de ponteiro fosse `void *`, omitindo a necessidade de uma adaptação explícita.

Estrutura de endereço de soquete IPv6

O endereço de soquete IPv6 é definido incluindo o cabeçalho `<netinet/in.h>`; mostramos isso na Figura 3.4.

```
struct in6_addr {
uint8_t s6_addr[16];          /* endereço IPv6 de 128 bits */
                              /* ordenado por byte de rede */
};

#define SIN6_LEN        /* requerido para testes em tempo de compilação */

struct sockaddr_in6 {
uint8_t         sin6_len;     /* comprimento dessa struct (28) */
sa_family_t     sin6_family;  /* AF_INET6 */
in_port_t       sin6_port;    /* número de porta da camada de transporte */
                              /* ordenado por byte de rede */
uint32_t        sin6_flowinfo; /* informação de fluxo, não-definido */
struct in6_addr sin6_addr;    /* endereço IPv6 */
                              /* ordenado por byte de rede */
uint32_t        sin6_scope_id; /* conjunto de interfaces para um escopo */
};
```

Figura 3.4 Estrutura de endereço de soquete IPv6: `sockaddr_in6`.

> As extensões para a API de soquetes para IPv6 estão definidas na RFC 3493 (Gilligan *et all.*, 2003).

Observe os seguintes pontos sobre a Figura 3.4:

- A constante `SIN6_LEN` deve ser definida se o sistema suportar o membro comprimento (length) para estruturas de endereço de soquete.
- A família IPv6 é `AF_INET6`, ao passo que a família IPv4 é `AF_INET`.
- Os membros nessa estrutura são ordenados de modo que, se a estrutura `sockaddr_in6` for alinhada em 64 bits, assim será o membro `sin6_addr` de 128 bits. Em alguns processadores de 64 bits, os acessos a dados com valores de 64 bits são otimizados se armazenados em um limite de 64 bits.
- O membro `sin6_flowinfo` é dividido em dois campos:
  - Os 20 bits de ordem inferior são o rótulo do fluxo
  - Os 12 bits de ordem superior são reservados

  O campo de rótulo do fluxo (flow label) é descrito na Figura A.2. O uso do campo de rótulo de fluxo ainda é um tópico de pesquisa.
- O `sin6_scope_ id` identifica a zona de escopo em que um endereço com escopo é significativo, mais comumente um índice de interface para um endereço de enlace local (Seção A.5).

Nova estrutura de endereço de soquete genérica

Uma nova estrutura de endereço de soquete genérica foi definida como parte da API de soquetes IPv6, para superar algumas falhas da `struct sockaddr` existente. Diferentemente da `struct sockaddr`, a nova `struct sockaddr_storage` é suficientemente grande para manter qualquer tipo de endereço de soquete suportado pelo sistema. A estrutura `sockaddr_storage` é definida incluindo o cabeçalho `<netinet/in.h>`, que mostramos na Figura 3.5.

```
struct sockaddr_storage {
   uint8_t     ss_len;          /* comprimento dessa estrutura
                                   (dependente da implementação) */
   sa_family_t ss_family;       /* família de endereços: AF_xxx valor */
   /* elementos dependentes da implementação a fornecer:
    * a) alinhamento suficiente para atender aos requisitos de alinhamento de
    *    todos os tipos de endereço de soquete que o sistema suporta.
    * b) memória suficiente para armazenar qualquer tipo de endereço de
    *    soquete que o sistema suporta.
    */
};
```

Figura 3.5 A estrutura de endereço de soquete de armazenamento: `sockaddr_storage`.

O tipo `sockaddr_storage` fornece uma estrutura de endereço de soquete genérica que é diferente de `struct sockaddr` de duas maneiras:

a. Se qualquer estrutura de endereço de soquete que o sistema suporta tiver requisitos de alinhamento, a `sockaddr_storage` fornece o requisito mais estrito de alinhamento.

b. A `sockaddr_storage` é suficientemente grande para conter qualquer estrutura de endereço de soquete que o sistema suporta.

Observe que os campos da estrutura `sockaddr_storage` são opacos para o usuário, exceto por `ss_family` e `ss_len` (se presente). `sockaddr_storage` deve ser adaptada ou copiada para a estrutura de endereço de soquete apropriada para o endereço fornecido em `ss_family` a fim de acessar quaisquer outros campos.

Comparação entre estruturas de endereço de soquete

A Figura 3.6 mostra uma comparação entre as cinco estruturas de endereço de soquete que encontraremos neste capítulo: IPv4, IPv6, domínio Unix (Figura 15.1), enlace de dados (Figura 18.1) e armazenamento. Nessa figura, supomos que todas as estruturas de endereço de soquete contêm um campo de comprimento de um byte, que o campo família também ocupe um byte e que qualquer campo que deva ter pelo menos algum número de bits tenha exatamente esse número de bits.

Duas estruturas de endereço de soquete têm comprimento fixo, enquanto a estrutura do domínio Unix e a estrutura de enlace de dados têm comprimento variável. Para tratar estruturas de comprimento variável, sempre que passamos um ponteiro para uma estrutura de endereço de soquete como um argumento para uma das funções de soquete, passamos seu comprimento como um outro argumento. Mostramos o tamanho em bytes (para a implementação 4.4BSD) das estruturas de comprimento fixo abaixo de cada estrutura.

> A estrutura `sockaddr_un` em si não tem comprimento variável (Figura 15.1), mas o volume de informações – o nome de caminho dentro da estrutura – tem comprimento variável. Ao passar ponteiros para essas estruturas, devemos ter cuidado quanto à maneira como tratamos o campo de comprimento, tanto o campo de comprimento na própria estrutura de endereço de soquete (se suportado pela implementação) como o comprimento até e a partir do kernel.
>
> Essa figura mostra o estilo que seguimos por todo este capítulo: nomes de estrutura sempre são mostrados em uma fonte em negrito, seguida por chaves, como em **`sockaddr_in{}`**.
>
> Observamos anteriormente que o campo de comprimento foi adicionado a todas as estruturas de endereço de soquete com a distribuição 4.3BSD Reno. Se o campo de comprimento estivesse presente na distribuição original de soquetes, não haveria necessidade do argumento de comprimento para todas as funções de soquete: por exemplo, o terceiro argumento para `bind` e `connect`. Em vez disso, o tamanho da estrutura poderia estar contido no campo de comprimento da estrutura.

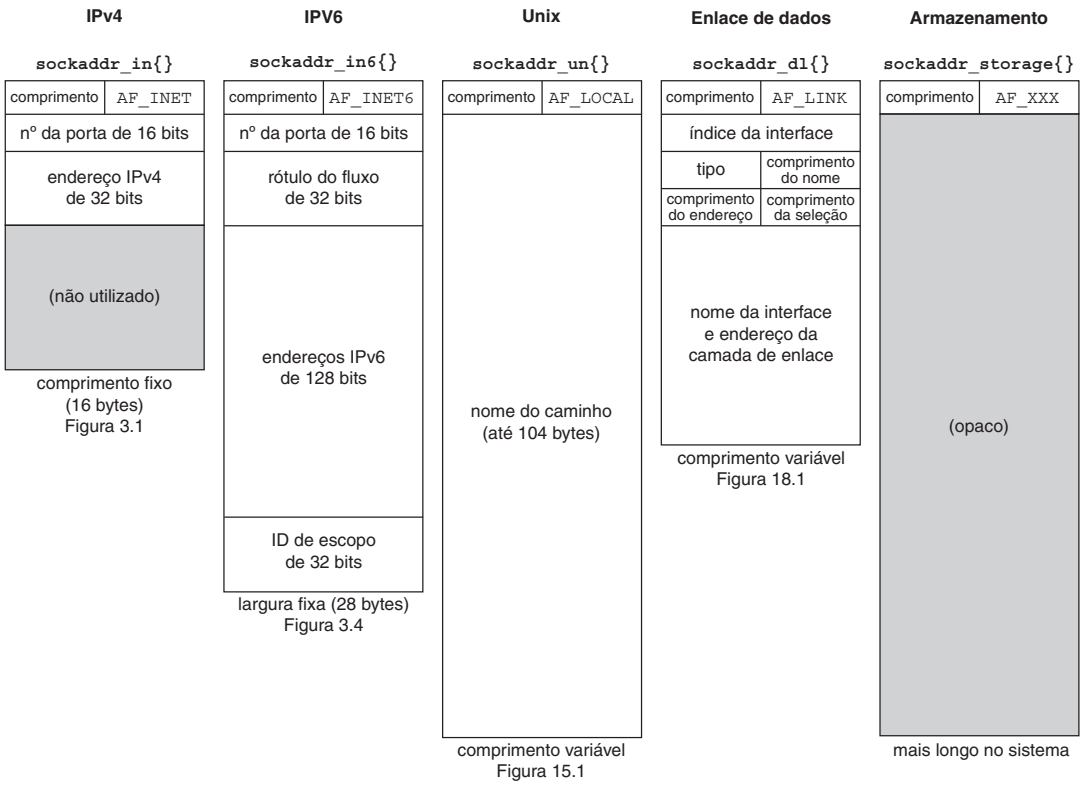


Figura 3.6 Comparação entre várias estruturas de endereço de soquete.

3.3 Argumentos valor-resultado

Mencionamos que quando uma estrutura de endereço de soquete é passada para qualquer função de soquete, ela sempre é passada por referência. Isto é, um ponteiro para a estrutura é passado. O comprimento da estrutura também é passado como um argumento. Mas a maneira como o comprimento é passado depende da direção em que a estrutura está sendo passada: do processo para o kernel ou vice-versa.

1. Três funções, `bind`, `connect` e `sendto`, passam uma estrutura de endereço de soquete do processo para o kernel. Um argumento para essas três funções é o ponteiro para a estrutura de endereço de soquete e um outro argumento é o tamanho em inteiro da estrutura, como em:

```
struct sockaddr_in serv;

/* preenche serv{} */
connect(sockfd, (SA *) &serv, sizeof(serv));
```

   Como o kernel recebe tanto o ponteiro como o tamanho da estrutura que este aponta, ele sabe exatamente quantos dados copiar do processo. A Figura 3.7 mostra esse cenário.

   Veremos no próximo capítulo que o tipo de dados para o tamanho de uma estrutura de endereço de soquete é na verdade `socklen_t` e não `int`, mas a especificação POSIX recomenda que `socklen_t` seja definido como `uint32_t`.

2. Quatro funções, `accept`, `recvfrom`, `getsockname` e `getpeername`, passam uma estrutura de endereço de soquete do kernel para o processo, a direção inversa do

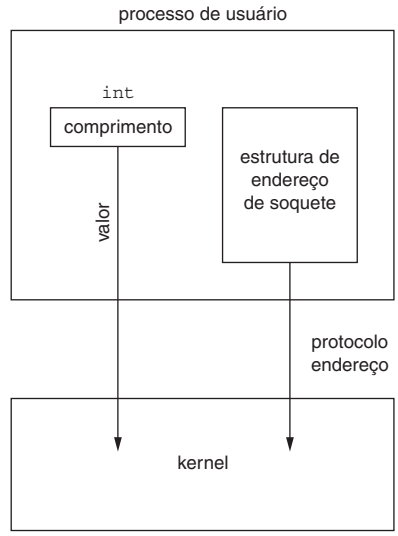


Figura 3.7 Estrutura de endereço de soquete passada do processo para o kernel.

cenário anterior. Dois dos argumentos para essas quatro funções são o ponteiro para a estrutura de endereço de soquete junto com um ponteiro para um inteiro contendo o tamanho da estrutura, como em:

```
struct sockaddr_un cli; /* domínio Unix */
socklen_t len;

len = sizeof(cli);      /* len é um valor */
getpeername(unixfd, (SA *) &cli, &len);
/* len pode ter mudado */
```

A razão pela qual o tamanho muda de um inteiro para se tornar um ponteiro para um inteiro é porque o tamanho é tanto um valor quando a função é chamada (ela informa ao kernel o tamanho da estrutura de modo que o kernel não escreva depois do final da estrutura ao preenchê-la) como um resultado quando a função retorna (ela informa ao processo o volume de informações que o kernel na verdade armazenou na estrutura). Esse tipo de argumento é chamado um argumento *valor-resultado*. A Figura 3.8 mostra esse cenário.

Veremos um exemplo de argumentos de valor-resultado na Figura 4.11.

> Temos discutido as estruturas de endereço de soquete sendo passadas entre o processo e o kernel. Para uma implementação como 4.4BSD, em que todas as funções de soquete são chamadas de sistema dentro do kernel, isso está correto. Mas em algumas implementações, notavelmente System V, funções de soquete são apenas funções de biblioteca que são executadas como parte de um processo normal de usuário. A maneira como essas funções fazem interface com a pilha de protocolos no kernel é um detalhe de implementação que normalmente não nos afeta. Contudo, por questão de simplicidade, continuaremos a discutir essas estruturas como sendo passadas entre o processo e o kernel por funções como `bind` e `connect`. (Veremos na Seção C.1 que as implementações System V de fato passam estruturas de endereço de soquete entre processos e o kernel, mas como parte de mensagens STREAMS.)
>
> Duas outras funções passam estruturas de endereço de soquete: `recvmsg` e `sendmsg` (Seção 14.5). Porém, veremos que o campo de comprimento não é um argumento de função, mas membro de estrutura.

Ao utilizar argumentos de valor-resultado para o comprimento das estruturas de endereço de soquete, se a estrutura de endereço de soquete tiver comprimento fixo (Figura 3.6), o va-

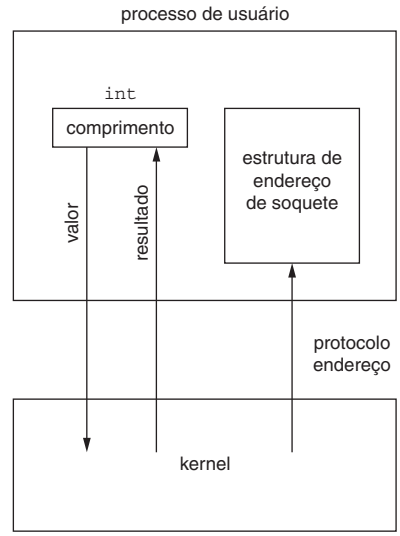


Figura 3.8 Estrutura de endereço de soquete passada do kernel para o processo.

lor retornado pelo kernel sempre terá esse tamanho fixo: por exemplo, 16 para uma `sockaddr_in` de IPv4 e 28 para uma `sockaddr_in6` de IPv6. Mas, com uma estrutura de endereço de soquete de comprimento variável (por exemplo, uma `sockaddr_un` de domínio Unix), o valor retornado pode ser menor que o tamanho máximo da estrutura (como veremos na Figura 15.2).

Com a programação de rede, o exemplo mais comum de um argumento de valor-resultado é o comprimento de uma estrutura de endereço de soquete retornada. Mas encontraremos outros argumentos de valor-resultado neste capítulo:

- Os três argumentos intermediários para a função `select` (Seção 6.3)
- O argumento de comprimento para a função `getsockopt` (Seção 7.2)
- Os membros `msg_namelen` e `msg_controllen` da estrutura `msghdr`, quando utilizada com `recvmsg` (Seção 14.5)
- O membro `ifc_len` da estrutura `ifconf` (Figura 17.2)
- O primeiro dos dois argumentos de comprimento para a função `sysctl` (Seção 18.4)

3.4 Funções de ordenamento de bytes

Considere um inteiro de 16 bits composto de 2 bytes. Há duas maneiras de armazenar os dois bytes na memória: com o byte de ordem inferior no endereço inicial, conhecida como ordem de bytes *little-endian,* ou com o byte de ordem superior no endereço inicial, conhecida como ordem de bytes *big-endian*. Mostramos esses dois formatos na Figura 3.9.

Nessa figura, mostramos os endereços de memória crescentes indo da direita para a esquerda na parte superior e da esquerda para a direita na parte inferior. Também mostramos o bit mais significativo (*most significant bit* – MSB) como o bit mais à esquerda do valor de 16 bits e o bit menos significativo (*least significant bit* – LSB) como o bit mais à direita.

> Os termos "little-endian" e "big-endian" indicam qual extremidade do valor de múltiplos bytes (a extremidade grande [*big end* em inglês] ou a pequena [*little end*]), é armazenada no endereço inicial do valor.

Infelizmente, não há nenhum padrão entre essas duas ordens de bytes e encontramos sistemas que utilizam ambos os formatos. Denominamos o ordenamento de bytes utilizado por

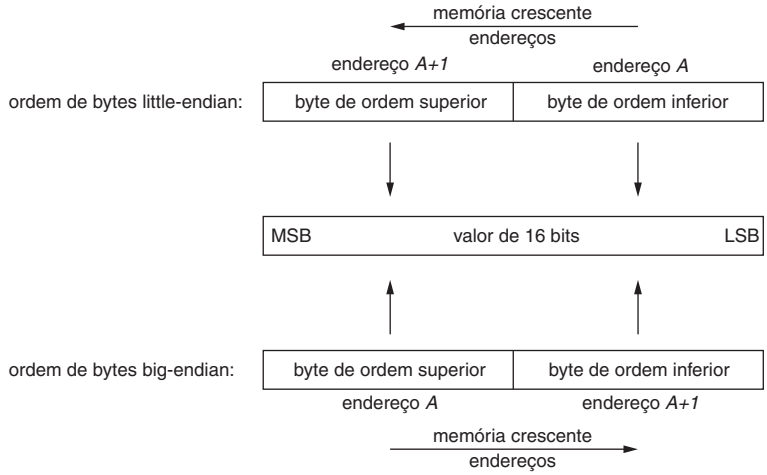


Figura 3.9 A ordem de bytes little-endian e a ordem de bytes big-endian para um inteiro de 16 bits.

um dado sistema como a ordem de bytes do host. O programa mostrado na Figura 3.10 imprime a ordem de bytes do host.

intro/byteorder.c

```
 1 #include      "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     union {
 6         short  s;
 7         char   c[sizeof(short)];
 8     } un;

 9     un.s = 0x0102;
10     printf("%s: ", CPU_VENDOR_OS);
11     if (sizeof(short) == 2) {
12         if (un.c[0] == 1 && un.c[1] == 2)
13             printf("big-endian\n");
14         else if (un.c[0] == 2 && un.c[1] == 1)
15             printf("little-endian\n");
16         else
17             printf("unknown\n");
18     } else
19         printf("sizeof(short) = %d\n", sizeof(short));

20     exit(0);
21 }
```

intro/byteorder.c

Figura 3.10 O programa para determinar a ordem de bytes do host.

Armazenamos o valor `0x0102` de dois bytes no inteiro curto e então examinamos os dois bytes consecutivos, `c[0]` (o endereço *A* na Figura 3.9) e `c[1]` (o endereço *A+1* na Figura 3.9), para determinar a ordem de bytes.

A string `CPU_VENDOR_OS` é determinada pelo programa GNU `autoconf` quando o software neste livro é configurado e identifica o tipo de CPU, o fornecedor e a distribuição do SO. Mostramos alguns exemplos aqui da saída desse programa quando da execução dos vários sistemas na Figura 1.16.

```
freebsd4 % byteorder
i386-unknown-freebsd4.8: little-endian

macosx % byteorder
powerpc-apple-darwin6.6: big-endian

freebsd5 % byteorder
sparc64-unknown-freebsd5.1: big-endian

aix % byteorder
powerpc-ibm-aix5.1.0.0: big-endian

hpux % byteorder
hppa1.1-hp-hpux11.11: big-endian

linux % byteorder
i586-pc-linux-gnu: little-endian

solaris % byteorder
sparc-sun-solaris2.9: big-endian
```

Discutimos o ordenamento de bytes de um inteiro de 16 bits; obviamente, a mesma discussão se aplica a um inteiro de 32 bits.

> Atualmente, há uma variedade de sistemas que podem alterar entre o ordenamento de bytes little-endian e big-endian, às vezes na reinicialização do sistema ou em tempo de execução.

Devemos lidar com essas diferenças no ordenamento de bytes como programadores de rede porque os protocolos de rede devem especificar uma ordem de bytes de rede. Por exemplo, em um segmento TCP, há um número de porta de 16 bits e um endereço IPv4 de 32 bits. A pilha de protocolos emissora e a pilha de protocolos receptora devem entrar em um acordo sobre a ordem em que os bytes desses campos de múltiplos bytes serão transmitidos. Os protocolos Internet utilizam o ordenamento de byte big-endian para esses inteiros de múltiplos bytes.

Em teoria, uma implementação poderia armazenar os campos em uma estrutura de endereço de soquete na ordem de bytes do host e então converter *para* e da ordem de bytes de rede ao mover os campos *para* e *dos* cabeçalhos de protocolo, de modo que não precisemos nos preocupar com esse detalhe. Mas, tanto o histórico como a especificação POSIX informam que certos campos nas estruturas de endereço de soquete devem ser mantidos na ordem de byte da rede. Nosso interesse, portanto, é converter entre a ordem de bytes do host e a ordem de bytes da rede. Utilizamos as quatro funções a seguir para converter entre essas duas ordens de bytes.

```
#include <netinet/in.h>

uint16_t htons (uint16_t valorDe16BitsDoHost);

uint32_t htonl (uint32_t valorDe16BitsDoHost);

                                        As duas retornam: o valor na ordem de bytes da rede

uint16_t ntohs (uint16_t valorDe32BitsDaRede);

uint32_t ntohl (uint32_t valorDe32BitsDaRede) ;

                                        As duas retornam: o valor na ordem de bytes do host
```

Nos nomes dessas funções, `h` significa *host*, `n` significa *network* (rede), `s` significa *short* (curto) e `l` significa *long* (longo). Os termos "curto" e "longo" são artefatos históricos da implementação Digital VAX do 4.2BSD. Devemos pensar no `s` como um valor de 16 bits (como um número de porta TCP ou UDP) e no `l` como um valor de 32 bits (como um endereço

IPv4). De fato, na Digital Alpha de 64 bits, um inteiro longo ocupa 64 bits, mesmo assim as funções `htonl` e `ntohl` operam sobre valores de 32 bits.

Ao utilizar essas funções, não nos preocupamos com os valores reais (big-endian ou little-endian) para a ordem de bytes do host e para a ordem de bytes da rede. O que devemos fazer é chamar a função apropriada para converter um dado valor entre a ordem de bytes do host e de rede. Nesses sistemas que têm o mesmo ordenamento de bytes dos protocolos de Internet (big-endian), essas quatro funções normalmente são definidas como macros nulas.

Discutiremos o problema de ordenamento de bytes, com relação aos dados contidos em um pacote de rede em oposição aos campos nos cabeçalhos de protocolo, na Seção 5.18 e no Exercício 5.8.

Ainda não definimos o termo "byte". Utilizamos esse termo para significar uma quantidade de 8 bits, uma vez que quase todos os sistemas de computadores atuais utilizam bytes de 8 bits. A maioria dos padrões Internet utiliza o termo *octeto* em vez de byte para indicar uma quantidade de 8 bits. Isso teve início nos primeiros estágios do TCP/IP porque boa parte do trabalho inicial era feita em sistemas como o DEC-10, que não utilizava bytes de 8 bits.

Uma outra convenção importante quanto aos padrões Internet é o ordenamento de bits. Em vários padrões Internet, você verá "figuras" de pacotes que são semelhantes à seguinte (esses são os primeiros 32 bits do cabeçalho IPv4 na RFC 791):

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Isso representa quatro bytes na ordem em que eles aparecem no cabo de rede; o bit mais à esquerda é o mais significativo. Entretanto, a numeração inicia com zero atribuído ao bit mais significativo. Essa é uma notação com a qual você deve se familiarizar para facilitar a leitura das definições de protocolos nas RFCs.

> Um erro comum de programação de rede na década de 1980 era desenvolver o código em estações de trabalho Sun (Motorola 68000s com big-endian) e esquecer de chamar qualquer uma dessas quatro funções. O código funcionava bem nessas estações de trabalho, mas não quando portado para máquinas com little-endian (como VAXes).

3.5 Funções de manipulação de bytes

Há dois grupos de funções que operam em campos multibyte, sem interpretar ou assumir que os dados são uma string em C terminada por um caractere nulo. Precisamos desses tipos de funções ao lidar com as estruturas de endereço de soquete porque precisamos manipular campos como endereços IP, que podem conter bytes de 0, mas não são strings de caractere em C. As funções que iniciam com `str` (string), definidas incluindo o cabeçalho `<string.h>`, lidam com strings de caracteres C terminadas por um caractere nulo.

As funções do primeiro grupo, cujo nome inicia com `b` (de byte), são do 4.2BSD e ainda são fornecidas por quase todos os sistemas que suportam funções de soquete. As do segundo grupo, cujo nome inicia com `mem` (de *memory*), são do padrão ANSI C e são fornecidas com qualquer sistema que suporta uma biblioteca ANSI C.

Primeiro mostramos as funções derivadas de Berkeley, apesar de que a única que utilizamos neste capítulo é `bzero`. (Utilizamos essa função porque ela tem somente dois argumentos e é mais fácil lembrar que a função `memset` de três argumentos, como explicado na página 8.) Você pode encontrar as outras duas funções, `bcopy` e `bcmp`, nas aplicações existentes.

```
#include <strings.h>

void bzero (void *dest, size_t nbytes);

void bcopy(const void *src, void *dest, size_t nbytes);

int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);
```

Retorna: 0 se igual, não-zero se desigual

> Esse é nosso primeiro encontro com o qualificador `const` ANSI C. Nas três utilizações aqui, ele indica que aquilo para o que o ponteiro aponta com essa qualificação, *src*, *ptr1* e *ptr2*, não é modificado pela função. Dito de uma outra maneira, a memória para a qual o ponteiro `const` aponta é lida, mas não modificada pela função.

`bzero` configura o número especificado de bytes como 0 no destino. Freqüentemente utilizamos essa função para inicializar uma estrutura de endereço de soquete como 0. `bcopy` move o número especificado de bytes da origem para o destino. `bcmp` compara duas strings de bytes arbitrárias. O valor de retorno é zero se as duas strings de byte forem idênticas; caso contrário, ela é não-zero.

As seguintes funções são funções ANSI C:

```
#include <string.h>

void *memset (void *dest, int c, size_t len);

void *memcpy(void *dest, const void *src, size_t nbytes);

int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);
```

Retorna: 0 se igual, <0 ou >0 se desigual (ver texto)

`memset` configura o número especificado de bytes como o valor de *c* no destino. `memcpy` é semelhante a `bcopy`, mas a ordem dos dois argumentos de ponteiro é trocada. `bcopy` trata corretamente campos sobrepostos, enquanto o comportamento de `memcpy` permanece indefinido se a origem e o destino se sobrepuserem. A função `memmove` ANSI C deve ser utilizada quando os campos se sobrepõem.

> Uma maneira de lembrar a ordem dos dois ponteiros para `memcpy` é lembrar que eles são escritos na mesma ordem, da esquerda para a direita, de uma instrução de atribuição em C:
>
> *dest = src;*
>
> Uma das maneiras de lembrar a ordem dos dois argumentos finais para `memset` é perceber que todas as funções `mem`*XXX* ANSI C requerem um argumento de comprimento que sempre é o argumento final.

`memcmp` compara duas strings de byte arbitrárias e retorna 0 se elas forem idênticas. Se não forem idênticas, o valor de retorno é maior que 0 ou menor que 0, dependendo se o primeiro byte desigual apontado por *ptr1* for maior ou menor que o byte correspondente apontado por *ptr2*. A comparação é feita supondo que os dois bytes desiguais são `unsigned chars`.

3.6 Funções `inet_aton`, `inet_addr` e `inet_ntoa`

Descreveremos dois grupos de funções de conversão de endereços nesta e na próxima seção. Elas convertem um endereço Internet entre strings ASCII (o que humanos preferem utilizar)

e valores binários ordenados de bytes de rede (valores que são armazenados em estruturas de endereço de soquete).

1. `inet_aton, inet_ntoa` e `inet_addr` convertem um endereço IPv4 de uma string na notação decimal separada por pontos (por exemplo, "206.168.112.96") em seu valor binário ordenado de bytes de rede de 32 bits. Você provavelmente encontrará essas funções em vários códigos existentes.
2. As funções mais recentes, `inet_pton` e `inet_ntop`, tratam de endereços IPv4 e IPv6. Descreveremos essas duas funções na próxima seção e as utilizaremos por todo este capítulo.

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

int inet_aton(const char *strptr, struct in_addr *addrptr);
```

Retorna: 1 se a string for válida, 0 em caso de erro

```
in_addr_t inet_addr (const char *strptr) ;
```

Retorna: endereço IPv4 ordenado por byte de rede de 32 bits; `INADDR_NONE` se ocorrer um erro

```
char *inet_ntoa(struct in_addr inaddr);
```

Retorna: ponteiro para a string na notação decimal separada por pontos

A primeira dessas, `inet_aton`, converte a string de caracteres C para a qual aponta *strptr* no seu valor binário ordenado por byte de rede de 32 bits, que é armazenado pelo ponteiro *addrptr*. Se bem-sucedida, 1 é retornado; caso contrário, 0 é retornado.

> Um recurso de `inet_aton` não documentado é se *addrptr* for um ponteiro nulo, a função ainda realiza sua validação da string de entrada, mas não armazena nenhum resultado.

`inet_addr` faz a mesma conversão, retornando o valor binário ordenado por byte de rede de 32 bits como o valor de retorno. O problema com essa função é que todos os 2^{32} possíveis valores binários são endereços IP válidos (0.0.0.0 a 255.255.255.255), mas a função retorna a constante `INADDR_NONE` (em geral, 32 bits com valor 1) em um erro. Isso significa que a string na notação decimal separada por pontos 255.255.255.255 (o endereço de broadcast limitado do IPv4, Seção 20.2) não pode ser tratada por essa função uma vez que seu valor binário parece indicar falha da função.

> Um problema potencial com `inet_addr` é que algumas páginas man afirmam que ela retorna –1 em um erro, em vez de `INADDR_NONE`. Isso pode levar a problemas, dependendo do compilador C, ao comparar o valor de retorno da função (um valor sem sinal) com uma constante negativa.

Hoje, `inet_addr` está obsoleto e qualquer novo código deve utilizar `inet_aton`. Melhor ainda é utilizar as funções mais recentes descritas na próxima seção, que tratam tanto o IPv4 como o IPv6.

A função `inet_ntoa` converte um endereço IPv4 ordenado de byte de rede binário de 32 bits na sua correspondente string na notação decimal separada por pontos. A string apontada pelo valor de retorno da função reside na memória estática. Isso significa que a função não é reentrante, o que discutiremos na Seção 11.18. Por fim, observe que essa função recebe uma estrutura como seu argumento, não um ponteiro para uma estrutura.

> As funções que recebem estruturas reais como argumentos são raras. É mais comum passar um ponteiro para a estrutura.

3.7 Funções `inet_pton` e `inet_ntop`

Essas duas funções são novas no IPv6 e funcionam tanto com endereços IPv4 como com IPv6. Utilizamos essas duas funções por todo este capítulo. As letras "p" e "n" significam *presentation* (apresentação) e *numeric* (numérico). O formato de apresentação para um endereço é freqüentemente uma string ASCII e o formato numérico é o valor binário que é colocado em uma estrutura de endereço de soquete.

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

int inet_pton (int family, const char *strptr, void *addrptr);
```

Retorna: 1 se OK, 0 se a entrada não estiver em um formato válido de apresentação, –1 em erro

```
const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len);
```

Retorna: ponteiro para resultado se OK, NULL em erro

O argumento *family* para as duas funções é um `AF_INET` ou `AF_INET6`. Se *family* não for suportada, as duas funções retornam um erro com `errno` configurado como `EAFNOSUPPORT`.

A primeira função tenta converter a string apontada por *strptr*, armazenando o resultado binário com o ponteiro *addrptr*. Se bem-sucedida, o valor de retorno é 1. Se a string de entrada não tiver um formato de apresentação válido para a *família* especificada, 0 é retornado.

`inet_ntop` faz a conversão inversa, de numérico (*addrptr*) para apresentação (*strptr*). O argumento *len* é o tamanho do destino, para evitar que a função estoure o buffer do chamador. Para ajudar a especificar esse tamanho, as duas definições a seguir são definidas incluindo o cabeçalho `<netinet/in.h>`:

```
#define INET_ADDRSTRLEN 16 /* para notação com ponto decimal do IPv4 */
#define INET6_ADDRSTRLEN 46 /* para string hexadecimal do IPv6 */
```

Se *len* for muito pequeno para manter o formato de apresentação resultante, incluindo o caractere nulo de término, um ponteiro nulo retorna e `errno` é configurado como `ENOSPC`.

O argumento *strptr* para `inet_ntop` não pode ser um ponteiro nulo. O chamador deve alocar e especificar o tamanho de memória para o destino. Se bem-sucedido, esse ponteiro é o valor de retorno da função.

A Figura 3.11 resume as cinco funções que descrevemos nesta e na seção anterior.

Exemplo

Mesmo que seu sistema ainda não inclua suporte para IPv6, você pode começar a utilizar essas funções mais recentes substituindo chamadas na forma

```
foo.sin_addr.s_addr = inet_addr(cp);
```

por

```
inet_pton(AF_INET, cp, &foo.sin_addr);
```

e substituindo chamadas na forma

```
ptr = inet_ntoa(foo.sin_addr);
```

por

```
char str[INET_ADDRSTRLEN];
ptr = inet_ntop(AF_INET, &foo.sin_addr, str, sizeof(str));
```

A Figura 3.12 mostra uma definição simples de `inet_pton` que suporta somente IPv4. De maneira semelhante, a Figura 3.13 mostra uma versão simples de `inet_ntop` que suporta somente IPv4.

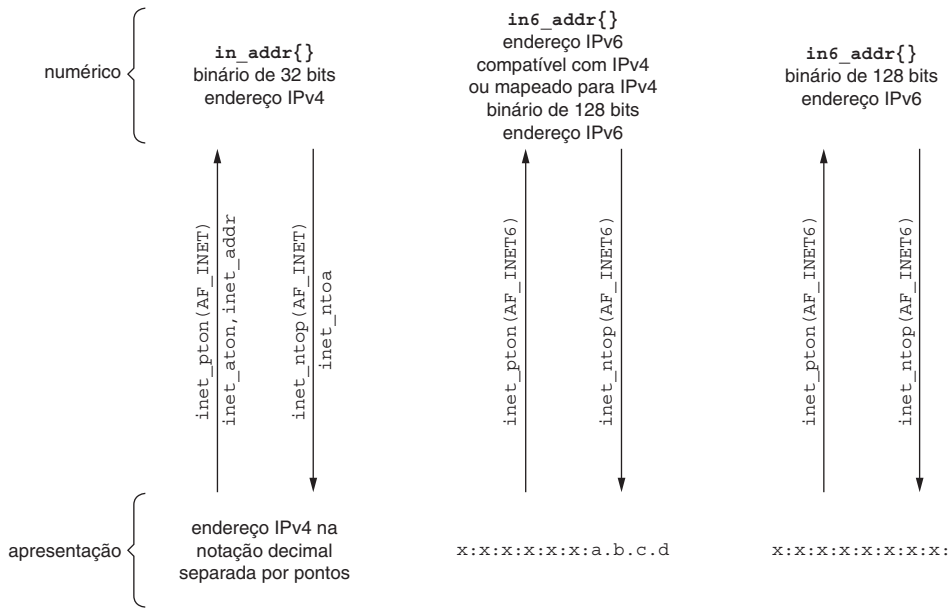


Figura 3.11 Resumo das funções de conversão de endereços.

libfree/inet_pton_ipv4.c

```
10 int
11 inet_pton(int family, const char *strptr, void *addrptr)
12 {
13     if (family == AF_INET) {
14         struct in_addr in_val;

15         if (inet_aton(strptr, &in_val)) {
16             memcpy(addrptr, &in_val, sizeof(struct in_addr));
17             return (1);
18         }
19         return(0);
20     }
21     errno = EAFNOSUPPORT;
22     return (-1);
23 }
```

libfree/inet_pton_ipv4.c

Figura 3.12 Versão simples de `inet_pton` que suporta somente IPv4.

libfree/inet_ntop_ipv4.c

```
 8 const char *
 9 inet_ntop(int family, const void *addrptr, char *strptr, size_t len)
10 {
11     const u_char *p = (const u_char *) addrptr;

12     if (family == AF_INET) {
13         char temp[INET_ADDRSTRLEN];

14         snprintf(temp, sizeof(temp), "%d.%d.%d.%d", p[0], p[1], p[2], p[3]);
15         if (strlen(temp) >= len) {
16             errno = ENOSPC;
17             return (NULL);
18         }
```

Figura 3.13 Versão simples de `inet_ntop` que suporta somente IPv4 (*continua*).

```
19          strcpy(strptr, temp);
20          return (strptr);
21      }
22      errno = EAFNOSUPPORT;
23      return (NULL);
24  }
```

libfree/inet_ntop_ipv4.c

Figura 3.13 Versão simples de `inet_ntop` que suporta somente IPv4 (*continuação*).

3.8 `sock_ntop` e funções relacionadas

Um problema básico com `inet_ntop` é que ele requer que o chamador passe um ponteiro para um endereço binário. Normalmente, esse endereço está contido em uma estrutura de endereço de soquete, exigindo que o chamador conheça o formato da estrutura e a família de endereços. Isto é, para utilizá-lo, devemos escrever um código da forma

```
struct sockaddr_in addr;

inet_ntop(AF_INET, &addr.sin_addr, str, sizeof(str));
```

para o IPv4, ou

```
struct sockaddr_in6 addr6;

inet_ntop(AF_INET6, &addr6.sin6_addr, str, sizeof(str));
```

para o IPv6. Isso torna nosso código dependente do protocolo.

Para resolver isso, escreveremos nossa própria função identificada como `sock_ntop` que recebe um ponteiro para uma estrutura de endereço de soquete, examina dentro da estrutura e chama a função apropriada para retornar o formato de apresentação do endereço.

```
#include "unp.h"

char *sock_ntop(const struct sockaddr *sockaddr, socklen_t addrlen);
```

Retorna: ponteiro não-nulo se OK, `NULL` em erro

> Essa é a notação que utilizamos para nossas próprias funções (funções de sistema não-padrão) neste livro: a caixa em torno do protótipo de função e do valor de retorno é tracejada. O cabeçalho incluído no início normalmente é o nosso cabeçalho `unp.h`.

sockaddr aponta para uma estrutura de endereço de soquete cujo comprimento é *addrlen*. A função utiliza seu próprio buffer estático para armazenar o resultado e um ponteiro desse buffer é o valor de retorno.

> Observe que utilizar um armazenamento estático para o resultado impede que a função seja *reentrante* ou *segura para thread*. Discutiremos outros detalhes disso na Seção 11.18. Tomamos essa decisão de *design* para essa função a fim de permitir facilmente chamá-la a partir dos exemplos simples neste livro.

O formato de apresentação está na notação com pontos decimais de um endereço IPv4 ou na forma de string hexadecimal de um endereço IPv6 cercado por colchetes, seguido por um terminador (utilizamos dois-pontos, semelhante à sintaxe URL), seguido pelo número decimal da porta e por um caractere nulo. Por isso, o tamanho do buffer deve ser pelo menos `INET_ADDRSTRLEN` mais 6 bytes para IPv4 (16 + 6 = 22) ou `INET6_ADDRSTRLEN` mais 8 bytes para IPv6 (46 + 8 = 54).

Mostramos o código-fonte somente para o caso `AF_INET` na Figura 3.14.

lib/sock_ntop.c

```
 5 char *
 6 sock_ntop(const struct sockaddr *sa, socklen_t salen)
 7 {
 8     char   portstr[8];
 9     static char str[128];        /* O domínio Unix é o maior */

10     switch (sa->sa_family) {
11     case AF_INET: {
12             struct sockaddr_in *sin = (struct sockaddr_in *) sa;

13             if (inet_ntop(AF_INET, &sin->sin_addr, str, sizeof(str)) == NULL)
14                return(NULL);
15             if (ntohs(sin->sin_port) != 0) {
16                snprintf(portstr, sizeof(portstr), ":%d",
17                         ntohs(sin->sin_port));
18                strcat(str, portstr);
19             }
20             return(str);
21         }
```

lib/sock_ntop.c

Figura 3.14 Nossa função `sock_ntop`.

Há algumas outras funções que definimos para operar nas estruturas de endereço de soquete que simplificarão a portabilidade do nosso código entre o IPv4 e o IPv6.

```
#include "unp.h"

int sock_bind_wild (int sockfd, int family);
```

Retorna: 0 se OK, –1 em erro

```
int sock_cmp_addr(const struct sockaddr * sockaddr1,
                  const struct sockaddr * sockaddr1, socklen_t addrlen) ;
```

Retorna: 0 se os endereços forem da mesma família e iguais, não-zero se diferentes

```
int sock_cmp_port(const struct sockaddr *sockaddr1,
                  const struct sockaddr *sockaddr2, socklen_t addrlen) ;
```

Retorna: 0 se os endereços forem da mesma família e se as portas forem iguais, não-zero se diferentes

```
int sock_get_port(const struct sockaddr *sockaddr, socklen_t addrlen);
```

Retorna: número da porta não-negativo para endereço IPv4 ou IPv6, –1 se diferente

```
char *sock_ntop_host(const struct sockaddr *sockaddr, socklen_t addrlen);
```

Retorna: ponteiro não-nulo se OK, `NULL` em erro

```
void sock_set_addr(const struct sockaddr *sockaddr, socklen_t addrlen, void *ptr);

void sock_set_port(const struct sockaddr *sockaddr, socklen_t addrlen, int port);

void sock_set_wild(struct sockaddr *sockaddr, socklen_t addrlen);
```

`sock_bind_wild` vincula o endereço curinga e uma porta efêmera a um soquete. `sock_cmp_addr` compara a parte do endereço de duas estruturas de endereço de soquete e

`sock_cmp_port` compara o número de porta de duas estruturas de endereço de soquete. `sock_get_port` retorna somente o número da porta e `sock_ntop_host` converte somente a parte host de uma estrutura de endereço de soquete no formato de apresentação (não no número da porta). `sock_set_addr` configura somente a parte de endereço de uma estrutura de endereço de soquete como o valor apontado por *ptr*; e `sock_set_port` configura somente o número da porta de uma estrutura de endereço de soquete. `sock_set_wild` configura a parte de endereço de uma estrutura de endereço de soquete como curinga. Como ocorre com todas as funções neste livro, fornecemos uma função empacotadora cujo nome inicia com "S" para todas as funções que retornam valores diferentes de `void` e normalmente chamamos a função empacotadora a partir dos nossos programas. Não mostramos o código-fonte para todas essas funções, mas ele está livremente disponível (consulte o Prefácio).

3.9 Funções `readn`, `writen` e `readline`

Os soquetes de fluxo (por exemplo, soquetes TCP) exibem um comportamento com as funções `read` e `write` que difere da E/S normal de arquivo. Uma função `read` ou `write` em um soquete de fluxo poderia gerar a entrada ou a saída de um número menor de bytes do que o solicitado, mas isso não é uma condição de erro. A razão disso é que os limites de buffer poderiam ser alcançados para o soquete no kernel. Tudo que é requerido para gerar saída ou entrada dos bytes restantes é fazer com que o chamador invoque a função `read` ou `write` novamente. Algumas versões do Unix também exibem esse comportamento ao gravar mais de 4.096 bytes em um pipe. Esse cenário sempre é uma possibilidade em um soquete de fluxo com `read`, mas normalmente é visto com `write` somente se o soquete for não-bloqueador. Contudo, sempre chamamos nossa função `writen` em vez de `write`, caso a implementação retorne uma contagem curta.

Fornecemos as três funções a seguir que utilizamos sempre que lemos de ou gravamos em um soquete de fluxo:

```
#include "unp.h"

ssize_t readn(int filedes, void *buff, size_t nbytes);

ssize_t writen(int filedes, const void *buff, size_t nbytes);

ssize_t readline(int filedes, void *buff, size_t maxlen);
```

Todas retornam: o número de bytes de leitura ou gravação, –1 em erro

A Figura 3.15 mostra a função `readn`, a Figura 3.16 mostra a função `writen` e a Figura 3.17 mostra a função `readline`.

lib/readn.c

```
 1 #include   "unp.h"

 2 ssize_t                       /* Lê n bytes a partir de um descritor. */
 3 readn(int fd, void *vptr, size_t n)
 4 {
 5     size_t nleft;
 6     ssize_t nread;
 7     char *ptr;

 8     ptr = vptr;
 9     nleft = n;
10     while (nleft > 0) {
```

Figura 3.15 Função `readn`: lê *n* bytes a partir de um descritor (*continua*).

```
11          if ( (nread = read(fd, ptr, nleft)) < 0) {
12              if (errno == EINTR)
13                  nread = 0;       /* e chama read() novamente */
14              else
15                  return(-1);
16          } else if (nread == 0)
17              break;               /* EOF */

18          nleft -= nread;
19          ptr += nread;
20      }
21      return(n - nleft);           /* retorna >= 0 */
22  }
```

lib/readn.c

Figura 3.15 Função `readn`: lê *n* bytes a partir de um descritor (*continuação*).

lib/writen.c

```
 1 #include   "unp.h"

 2 ssize_t                           /* Grava n bytes para um descritor. */
 3 writen(int fd, const void *vptr, size_t n)
 4 {
 5     size_t nleft;
 6     ssize_t nwritten;
 7     const char *ptr;

 8     ptr = vptr;
 9     nleft = n;
10     while (nleft > 0) {
11         if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
12             if (nwritten < 0 && errno == EINTR)
13                 nwritten = 0;     /* e chama write() novamente */
14             else
15                 return(-1);       /* erro */
16         }

17         nleft -= nwritten;
18         ptr += nwritten;
19     }
20     return(n);
21 }
```

lib/writen.c

Figura 3.16 função `writen`: grava *n* bytes para um descritor.

test/readline1.c

```
 1 #include   "unp.h"

 2 /* VERSÃO MUITO LENTA -- apenas um exemplo */
 3 ssize_t
 4 readline(int fd, void *vptr, size_t maxlen)
 5 {
 6     ssize_t n, rc;
 7     char c, *ptr;

 8     ptr = vptr;
 9     for (n = 1; n < maxlen; n++) {
10         again:
11             if ( (rc = read(fd, &c, 1)) == 1) {
12                 *ptr++ = c;
```

Figura 3.17 Função `readline`: lê uma linha de texto a partir de um descritor, um byte por vez (*continua*).

```
13              if (c == '\n')
14                  break;      /* nova linha é armazenada, como fgets() */
15          } else if (rc == 0) {
16              *ptr = 0;
17              return(n - 1);  /* EOF, n - 1 bytes são lidos */
18          } else {
19              if (errno == EINTR)
20                  goto again;
21              return(-1);     /* erro, errno configurado por read() */
22          }
23      }
24      *ptr = 0;               /* null termina como fgets() */
25      return(n);
26  }
```

test/readline1.c

Figura 3.17 Função `readline`: lê uma linha de texto a partir de um descritor, um byte por vez (*continuação*).

Nossas três funções procuram o erro `EINTR` (a chamada de sistema foi interrompida por um sinal capturado, o que discutiremos em mais detalhes na Seção 5.9) e continuam a ler ou gravar se o erro ocorrer. Tratamos o erro aqui, em vez de forçar o chamador a chamar `readn` ou `writen` novamente, uma vez que o propósito dessas três funções é evitar que o chamador trate uma contagem curta.

Na Seção 14.3, mencionaremos o fato de que o flag `MSG_WAITALL` pode ser utilizado com a função `recv` para substituir a necessidade de uma função `readn` separada.

Observe que nossa função `readline` chama a função `read` do sistema uma vez para cada byte de dados. Isso é muito ineficiente e foi a razão por que fizemos o comentário "MUITO LENTO" no código. Quando confrontado com o desejo de ler linhas de um soquete, é bem tentador recorrer à biblioteca-padrão de E/S (conhecida como "stdio"). Discutiremos essa abordagem em detalhes na Seção 14.8, mas ela pode ser um caminho perigoso. O mesmo armazenamento em buffer de stdio que resolve esse problema de desempenho cria vários problemas logísticos que podem levar a bugs bem ocultos na sua aplicação. A razão é que o estado dos buffers stdio não é exposto. Para explicar isso em mais detalhes, considere um protocolo baseado em texto entre um cliente e um servidor, em que vários clientes e servidores que utilizam esse protocolo podem ser implementados ao longo do tempo (realmente bem comum; por exemplo, há muitos navegadores e servidores Web que são escritos de maneira independente para a especificação HTTP). Boas técnicas de "programação defensiva" exigem que esses programas não apenas esperem que suas contrapartes sigam o protocolo de rede, mas também que verifiquem o tráfego inesperado de rede. Essas violações de protocolo devem ser informadas como erros para que os bugs sejam observados e corrigidos (bem como sejam detectadas tentativas maliciosas) e também para que as aplicações de rede possam se recuperar de problemas de tráfego e, se possível, continuem a funcionar. Utilizar stdio a fim de armazenar dados em buffer para melhor desempenho contraria esses objetivos uma vez que a aplicação não tem nenhuma maneira de informar se dados inesperados estão sendo mantidos nos buffers de stdio a qualquer dado momento.

Há muitos protocolos de rede baseados em texto, como o SMTP, o HTTP, o protocolo de conexão de controle de FTP e o finger. Portanto, o desejo de operar em linhas surge repetidas vezes. Mas o nosso conselho é pensar em termos de buffers e não de linhas. Escreva seu código para ler buffers de dados e, se uma linha é esperada, verifique se o buffer contém essa linha.

A Figura 3.18 mostra uma versão mais rápida da função `readline`, que utiliza seu próprio armazenamento em buffer em vez do armazenamento em buffer de stdio. Acima de tudo, o estado do buffer interno de `readline` é exposto, assim os chamadores têm visibilidade sobre

exatamente o que foi recebido. Mesmo com esse recurso, `readline` pode ser problemático, como veremos na Seção 6.3. Funções de sistema como `select` ainda não conhecem o buffer interno da `readline`, dessa forma um programa negligentemente escrito poderia facilmente se encontrar esperando dados já recebidos em `select` e armazenados nos buffers de `readline`. Nesse sentido, mesclar chamadas `readn` e `readline` não funcionará como o esperado a menos que `readn` seja modificada para também verificar o buffer interno.

lib/readline.c

```
 1 #include   "unp.h"
 2 static int read_cnt;
 3 static char *read_ptr;
 4 static char read_buf[MAXLINE];

 5 static ssize_t
 6 my_read(int fd, char *ptr)
 7 {
 8     if (read_cnt <= 0) {
 9         again:
10             if ( (read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
11                 if (errno == EINTR)
12                     goto again;
13                 return(-1);
14             } else if (read_cnt == 0)
15                 return(0);
16             read_ptr = read_buf;
17     }
18     read_cnt--;
19     *ptr = *read_ptr++;
20     return(1);
21 }
22 ssize_t
23 readline(int fd, void *vptr, size_t maxlen)
24 {
25     ssize_t n, rc;
26     char c, *ptr;
27     ptr = vptr;
28     for (n = 1; n < maxlen; n++) {
29         if ( (rc = my_read(fd, &c)) == 1) {
30             *ptr++ = c;
31             if (c == '\n')
32                 break;          /* nova linha é armazenada, como fgets() */
33         } else if (rc == 0) {
34             *ptr = 0;
35             return(n - 1);      /* EOF, n - 1 bytes são lidos */
36         } else
37             return(-1);         /* erro, errno configurado por read() */
38     }
39     *ptr = 0;                   /* null termina como fgets() */
40     return(n);
41 }
42 ssize_t
```

Figura 3.18 Melhor versão da função `readline` (*continua*).

```
43 readlinebuf(void **vptrptr)
44 {
45     if (read_cnt)
46         *vptrptr = read_ptr;
47     return(read_cnt);
48 }
```

lib/readline.c

Figura 3.18 Melhor versão da função `readline` (*continuação*).

2-21 A função interna `my_read` lê até um `MAXLINE` caracteres por vez e então os retorna, um por vez.

29 A única alteração na função `readline` em si é chamar `my_read` em vez de `read`.

42-48 Uma nova função, `readlinebuf`, expõe o estado interno do buffer de modo que os chamadores possam verificar e ver se outros dados foram recebidos além de uma única linha.

> Infelizmente, utilizando variáveis `static` em `readline.c` para manter as informações do estado entre chamadas sucessivas, as funções não são *reentrantes* nem *seguras para thread*. Discutiremos isso nas Seções 11.18 e 26.5. Desenvolveremos uma versão segura para thread utilizando dados específicos de thread na Figura 26.11.

3.10 Resumo

As estruturas de endereço de soquete são uma parte integral de cada programa de rede. Alocamos, preenchemos e passamos ponteiros para essas estruturas para várias funções de soquete. Às vezes passamos para uma função de soquete um ponteiro para uma dessas estruturas e ele preenche o conteúdo. Sempre passamos essas estruturas por referência (isto é, passamos um ponteiro para a estrutura, não a própria estrutura) e sempre passamos o tamanho da estrutura como um outro argumento. Quando uma função de soquete preenche uma estrutura, o comprimento também é passado por referência, de modo que seu valor possa ser atualizado pela função. Chamamos esses argumentos de valor-resultado.

As estruturas de endereço de soquete são autodefinidas porque elas sempre iniciam com um campo (a "família") que identifica a família de endereços contida na estrutura. Implementações mais recentes que suportam estruturas de endereço de soquete de comprimento variável também contêm um campo de comprimento no início, que contém o comprimento da estrutura inteira.

As duas funções que convertem endereços IP entre um formato de apresentação (o que escrevemos, como caracteres ASCII) e um formato numérico (o que é colocado em uma estrutura de endereço de soquete) são `inet_pton` e `inet_ntop`. Embora utilizemos essas duas funções nos próximos capítulos, elas são dependentes de protocolo. Uma técnica mais adequada é manipular as estruturas de endereço de soquete como objetos opacos, conhecendo somente o ponteiro para estrutura e seu tamanho. Utilizamos esse método para desenvolver um conjunto de funções de soquete que ajudou a tornar nossos programas independentes de protocolo. Completaremos o desenvolvimento das nossas ferramentas independentes de protocolo no Capítulo 11 com as funções `getaddrinfo` e `getnameinfo`.

Os soquetes TCP fornecem a uma aplicação um fluxo de bytes: não há nenhum marcador de registro. O valor de retorno de uma função `read` pode ser menor que o que solicitamos, mas isso não indica um erro. Para ajudar a ler e gravar um fluxo de bytes, desenvolvemos três funções: `readn`, `writen` e `readline`, que utilizaremos por todo o livro. Entretanto, programas de rede devem ser escritos para atuarem nos buffers em vez de em linhas.

Exercícios

3.1 Por que argumentos valor-resultado como o comprimento de uma estrutura de endereço de soquete precisam ser passados por referência?

3.2 Por que tanto as funções `readn` como `writen` copiam o ponteiro `void*` para um ponteiro `char*`?

3.3 As funções `inet_aton` e `inet_addr` têm sido tradicionalmente liberais naquilo que aceitam como uma string de endereço IPv4 na notação decimal separada por pontos: permitindo de um a quatro números separados por pontos decimais e que um `0x` inicial especifique um número hexadecimal ou que um 0 inicial especifique um número octal. (Experimente `telnet 0xe` para ver esse comportamento.) `inet_pton` é muito mais estrito com um endereço IPv4 e requer exatamente quatro números separados por três pontos decimais, com cada número como um número decimal entre 0 e 255. `inet_pton` não permite que um número na notação decimal separada por pontos seja especificado quando a família de endereços for `AF_INET6`, embora possamos argumentar que esses números devam ser permitidos e o valor de retorno deva ser o endereço o IPv6 mapeado para IPv4 para a string na notação decimal separada por pontos (Figura A.10).

Escreva uma nova função chamada `inet_pton_loose` que trate esses cenários: se a família de endereços for `AF_INET` e `inet_pton` retornar 0, chame `inet_aton` e verifique se ela é bem-sucedida. De maneira semelhante, se a família de endereço for `AF_INET6` e `inet_pton` retorna 0, chame `inet_aton`, e se ela for bem-sucedida, retorne o endereço IPv6 mapeado para IPv4.

Capítulo 4

Soquetes de TCP Elementares

4.1 Visão geral

Este capítulo descreve as funções de soquete básicas requeridas para escrever um cliente e um servidor TCP completos. Primeiro, descreveremos todas as funções de soquete básicas que utilizaremos e então desenvolveremos o cliente e o servidor no próximo capítulo. Trabalharemos com esse cliente e esse servidor por todo o livro, aprimorando-os várias vezes (Figuras 1.12 e 1.13).

Também descreveremos servidores concorrentes, uma técnica Unix comum para fornecer concorrência quando muitos clientes estão conectados ao mesmo servidor e ao mesmo tempo. Cada conexão de cliente faz com que o servidor bifurque um novo processo apenas para esse cliente. Neste capítulo, consideraremos somente o modelo de "um *processo* por cliente" utilizando `fork`, mas também iremos considerar um modelo diferente de "um *thread* por cliente" ao descrever threads no Capítulo 26.

A Figura 4.1 mostra a linha de tempo de um cenário típico que acontece entre um cliente e um servidor TCP. Primeiro, o servidor é iniciado; posteriormente, um cliente é iniciado e conectado ao servidor. Supomos que o cliente envia uma solicitação ao servidor, o servidor processa essa solicitação e envia uma resposta de volta ao cliente. Isso continua até que o cliente feche sua extremidade da conexão, o que envia uma notificação de fim de arquivo ao servidor. O servidor fecha então sua extremidade da conexão e termina ou espera uma nova conexão de cliente.

4.2 Função `socket`

Para realizar a E/S de rede, a primeira coisa que um processo deve fazer é chamar a função de soquete, especificando o tipo de protocolo de comunicação desejado (TCP utilizando IPv4, UDP utilizando IPv6, protocolo de fluxo de domínio do Unix, etc.).

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

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

Retorna: descritor não-negativo se OK, –1 em erro

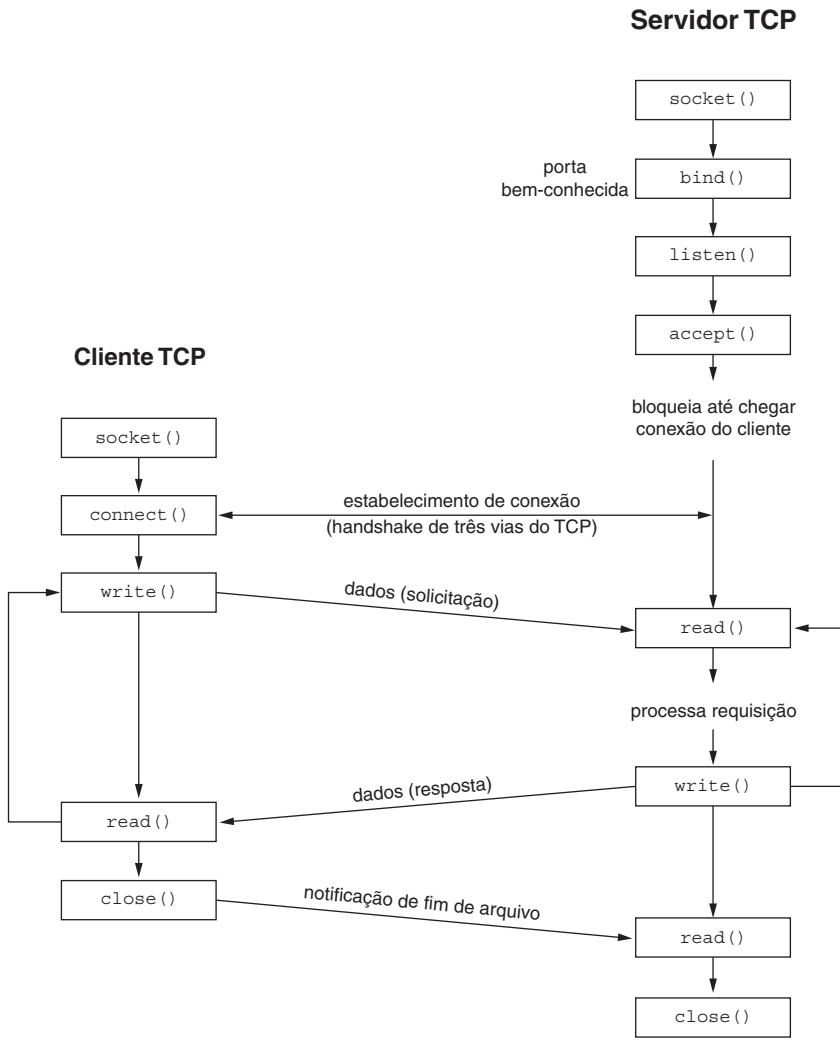


Figura 4.1 Funções de soquete para cliente/servidor TCP básico.

family especifica a *família* de protocolos e é uma das constantes mostradas na Figura 4.2. Esse argumento é freqüentemente referido como *domínio* em vez de *família*. O *type (tipo)* de soquete é uma das constantes mostradas na Figura 4.3. O argumento *protocol* da função de soquete deve ser configurado como o tipo de protocolo específico encontrado na Figura 4.4 ou 0 para selecionar o padrão do sistema para a combinação dada de *família* e *tipo*.

Nem todas as combinações de *família* e *tipo* de soquete são válidas. A Figura 4.5 mostra as combinações válidas, junto com os protocolos reais válidos para cada par. As caixas marcadas com "Sim" são válidas, mas não têm acrônimos úteis. As caixas em branco não são suportadas.

> Você também pode encontrar a correspondente constante `PF_xxx` como o primeiro argumento para `socket`. Discutiremos isso em mais detalhes no final desta seção.
>
> Observamos que você poderia encontrar `AF_UNIX` (o nome Unix histórico) em vez de `AF_LOCAL` (o nome `POSIX`), o que será comentado em mais detalhes no Capítulo 15.

família	Descrição
AF_INET	Protocolos IPv4
AF_INET6	Protocolos IPv6
AF_LOCAL	Protocolos de domínio Unix (Capítulo 15)
AF_ROUTE	Soquetes de roteamento (Capítulo 18)
AF_KEY	Soquete de chave (Capítulo 19)

Figura 4.2 Constantes da *família* de protocolos para a função `socket`.

tipo	Descrição
SOCK_STREAM	soquete de fluxo
SOCK_DGRAM	soquete de datagrama
SOCK_SEQPACKET	soquete de pacote seqüenciado
SOCK_RAW	soquete bruto

Figura 4.3 *Tipo* de soquete para a função `socket`.

Protocolo	Descrição
IPPROTO_TCP	Protocolo de transporte TCP
IPPROTO_UDP	Protocolo de transporte UDP
IPPROTO_SCTP	Protocolo de transporte SCTP

Figura 4.4 *Protocolo* de soquetes para `AF_INET` ou `AF_INET6`.

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP | SCTP	TCP | SCTP	Sim		
SOCK_DGRAM	UDP	UDP	Sim		
SOCK_SEQPACKET	SCTP	SCTP	Sim		
SOCK_RAW	IPv4	IPv6		Sim	Sim

Figura 4.5 Combinações de *família* e *tipo* para a função `socket`.

Há outros valores para os argumentos de *família* e *tipo*. Por exemplo, 4.4BSD suporta tanto `AF_NS` (os protocolos Xerox NS, freqüentemente chamados XNS) como `AF_ISO` (os protocolos OSI). De maneira semelhante, o tipo de `SOCK_SEQPACKET`, um soquete de pacote seqüenciado, é implementado tanto pelos protocolos Xerox NS como pelos OSI (descreveremos sua utilização com SCTP na Seção 9.2). Mas o TCP é um protocolo de fluxo de bytes e suporta somente os soquetes `SOCK_STREAM`.

O Linux suporta um novo tipo de soquete, `SOCK_PACKET`, que fornece acesso ao data-link, semelhante ao BPF e ao DLPI na Figura 2.1. Discutiremos esse soquete em mais detalhes no Capítulo 29.

O soquete de chave, `AF_KEY`, é o mais recente. Ele fornece suporte para segurança criptográfica. Semelhante à maneira como um soquete de roteamento (`AF_ROUTE`) é uma interface para a tabela de roteamento do kernel, o soquete de chave é uma interface para a tabela de chaves do kernel. Consulte o Capítulo 19 para detalhes.

Se bem-sucedida, a função `socket` retorna um valor `small integer` não-negativo, semelhante a um descritor de arquivo. Chamamos isso de *descritor de soquete* ou *sockfd*. Pa-

ra obter esse descritor de soquete, tudo que especificamos é uma família de protocolos (IPv4, IPv6 ou Unix) e o tipo de soquete (fluxo, datagrama ou bruto). Ainda não especificamos o endereço local de protocolo nem o endereço externo de protocolo.

`AF_xxx` versus `PF_xxx`

O prefixo "`AF`" significa "address family" ("família de endereços") e o prefixo "`PF`" significa "protocol family" ("família de protocolos"). Historicamente, a intenção era de que uma única família de protocolos poderia suportar múltiplas famílias de endereços e que o valor de `PF_` fosse utilizado para criar o soquete e o de `AF_` fosse utilizado nas estruturas de endereço de soquete. Mas, na realidade, uma família de protocolos que suporta múltiplas famílias de endereços nunca foi suportada e o cabeçalho `<sys/socket.h>` define o valor de `PF` para um dado protocolo como igual ao valor de `AF_` para esse protocolo. Embora não haja garantias de que essa igualdade entre os dois sempre será verdadeira, se uma alteração fosse feita nos protocolos existentes, uma grande quantidade de código existente quebraria. Para nos adaptarmos à prática da codificação existente, utilizamos somente a constante `AF_` neste capítulo, embora você possa encontrar o valor de `PF_`, principalmente em chamadas a `socket`.

> O exame de 137 programas que chamam `socket` na distribuição BSD/OS 2.1 mostra 143 chamadas que especificam o valor de `AF_` e somente 8 que especificam o valor de `PF_`.
>
> Historicamente, a razão para os conjuntos semelhantes de constantes com os prefixos `AF_` e `PF_` remonta à distribuição 4.1cBSD (Lanciani, 1996) e a uma versão da função `socket` que é anterior àquela que estamos descrevendo (que apareceu com o 4.2BSD). A versão 4.1cBSD de `socket` aceitava quatro argumentos, um dos quais era um ponteiro para uma estrutura `sockproto`. O primeiro membro dessa estrutura foi chamado de `sp_family` e seu valor era um dos valores de `PF_`. O segundo membro, `sp_protocol`, era um número de protocolo, semelhante ao terceiro argumento da função `socket` atual. Especificar essa estrutura era a única maneira de especificar a família de protocolos. Portanto, nesse sistema inicial, os valores de `PF` foram utilizados como tags de estrutura para especificar a família de protocolos na estrutura `sockproto` e os valores de `AF_` foram utilizados como tags de estrutura para especificar a família de endereços nas estruturas de endereço de soquete. A estrutura `sockproto` ainda está no 4.4BSD (páginas 626 e 627 do TCPv2), mas é utilizada somente internamente pelo kernel. A definição original tinha o comentário "protocol family" para o membro `sp_family`, mas isso foi alterado para "address family" no código-fonte do 4.4BSD.
>
> Para confundir ainda mais essa diferença entre as constantes `AF_` e `PF_`, a estrutura de dados do kernel do Berkeley que contém o valor que é comparado com o primeiro argumento de `socket` (o membro `dom_family` da estrutura `domain`, página 187 do TCPv2) tem o comentário de que ela contém um valor de `AF_`. Mas algumas estruturas `domain` dentro do kernel são inicializadas com o valor de `AF_` correspondente (página 192 do TCPv2), enquanto as outras são inicializadas com o valor de `PF_` (página 646 do TCPv2 e página 229 do TCPv3).
>
> Como uma outra nota histórica, a página man do 4.2BSD para `socket`, datada de julho de 1983, chama seu primeiro argumento *af* e lista os possíveis valores como constantes `AF_`.
>
> Por fim, observamos que o padrão `POSIX` especifica que o primeiro argumento para `socket` seja um valor de `PF_` e que o valor de `AF_` seja utilizado para uma estrutura de endereço de soquete. Mas ele define então somente um dos valores da família na estrutura `addrinfo` (Seção 11.6), concebido para utilização em uma chamada a `socket` ou em uma estrutura de endereço de soquete!

4.3 Função `connect`

A função `connect` é utilizada pelo cliente TCP para estabelecer uma conexão com um servidor TCP.

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

                                                    Retorna: 0 se OK, –1 em erro
```

sockfd é um descritor de soquete retornado pela função `socket`. O segundo e o terceiro argumentos são um ponteiro para estrutura de endereço de um soquete e seu tamanho, como descrito na Seção 3.3. A estrutura de endereço de soquete deve conter o endereço e o número de porta IP do servidor. Vimos um exemplo dessa função na Figura 1.5.

O cliente não tem de chamar `bind` (que descreveremos na próxima seção) antes de chamar `connect`: o kernel irá escolher tanto uma porta efêmera como o endereço IP de origem se necessário.

No caso de um soquete TCP, a função `connect` inicia um handshake de três vias do TCP (Seção 2.6). A função retorna somente quando a conexão é estabelecida ou se ocorrer um erro. Há diferentes retornos de erros possíveis.

1. Se o TCP cliente não receber nenhuma resposta para seu segmento SYN, `ETIMEDOUT` é retornado. O 4.4BSD, por exemplo, envia um SYN quando `connect` é chamada, outro depois de 6 segundos e mais outro depois de 24 segundos (página 828 do TCPv2). Se nenhuma resposta for recebida depois de um total de 75 segundos, o erro é retornado.

   Alguns sistemas fornecem um controle administrativo sobre esse tempo-limite; consulte o Apêndice E do TCPv1.

2. Se a resposta do servidor ao SYN do cliente for uma reinicialização (RST), isso indicará que nenhum processo está esperando conexões no host servidor na porta especificada (isto é, o processo servidor provavelmente não está em execução). Esse é um *erro irrecuperável* [*hard error*] e o erro `ECONNREFUSED` retorna ao cliente logo que o RST é recebido.

   Um RST é um tipo de segmento TCP enviado pelo TCP quando algo está errado. As três condições que geram um RST são: quando um SYN chega a uma porta que não tem nenhum servidor ouvinte (o que acabamos de descrever), quando o TCP quer abortar uma conexão existente e quando o TCP recebe um segmento para uma conexão que não existe. (O TCPv1 [páginas 246 a 250] contém informações adicionais.)

3. Se o SYN do cliente elicia um "destination unreachable" de ICMP a partir de algum roteador intermediário, isso é considerado um *erro leve* (*soft error*). O kernel cliente salva a mensagem, mas continua a enviar SYNs ao mesmo tempo entre cada SYN como no primeiro cenário. Se nenhuma resposta for recebida depois de algum período de tempo fixado (75 segundos para 4.4BSD), o erro ICMP salvo é retornado ao processo como `EHOSTUNREACH` ou `ENETUNREACH`. Também é possível que o sistema remoto não possa ser alcançado por nenhuma rota na tabela de encaminhamento do sistema local, ou que a chamada `connect` simplesmente retorne sem esperar.

   Muitos sistemas anteriores, como o 4.2BSD, abortavam incorretamente a tentativa de estabelecimento de conexão quando o erro "destination unreachable" de ICPM era recebido. Isso é errado porque esse erro de ICMP pode indicar uma condição transitória. Por exemplo, pode ser que a condição seja causada por um problema de roteamento que será corrigido.

   Observe que `ENETUNREACH` não está listado na Figura A.15, mesmo quando o erro indicar que a rede de destino está inacessível. Indicações de redes inacessíveis são consideradas obsoletas e as aplicações devem apenas tratar `ENETUNREACH` e `EHOSTUNREACH` como o mesmo erro.

Podemos ver essas diferentes condições de erro com o nosso cliente simples na Figura 1.5. Primeiro, especificamos o host local (127.0.0.1), que está em execução no servidor de data/hora e verificamos a saída.

```
solaris % daytimetcpcli 127.0.0.1
Sun Jul 27 22:01:51 2003
```

Para verificar um formato diferente à resposta retornada, especificamos um endereço IP de máquina diferente (nesse exemplo, o endereço IP da máquina HP-UX).

```
solaris % daytimetcpcli 192.6.38.100
Sun Jul 27 22:04:59 PDT 2003
```

Em seguida, especificamos um endereço IP que está na sub-rede local (192.168.1/24), mas o ID de host (100) é inexistente. Isto é, não há nenhum host na sub-rede com um ID de host igual a 100, portanto, quando o host cliente envia solicitações ARP (solicitando que esse host responda com seu endereço de hardware), ele nunca receberá uma resposta ARP.

```
solaris % daytimetcpcli 192.168.1.100
connect error: Connection timed out
```

Somente obtemos o erro depois da expiração de `connect` (cerca de quatro minutos com o Solaris 9). Observe que nossa função `err_sys` imprime a string legível por humanos associada com o erro `ETIMEDOUT`.

Nosso próximo exemplo é especificar um host (um roteador local) que não está em execução em um servidor de data/hora.

```
solaris % daytimetcpcli 192.168.1.5
connect error: Connection refused
```

O servidor responde imediatamente com um RST.

Nosso exemplo final especifica um endereço IP que não está acessível na Internet. Se observarmos os pacotes com `tcpdump`, veremos que um roteador a seis hops de distância retorna um erro de host inacessível de ICMP.

```
solaris % daytimetcpcli 192.3.4.5
connect error: No route to host
```

Como ocorre com o erro `ETIMEDOUT`, nesse exemplo, `connect` retorna o erro `EHOSTUNREACH` somente depois de esperar seu período de tempo especificado.

Em termos do diagrama de transição de estado TCP (Figura 2.4), `connect` move-se do estado CLOSED (o estado em que um soquete inicia quando é criado pela função `socket`) para o estado SYN_SENT e então, se bem-sucedido, para o estado ESTABLISHED. Se `connect` falhar, o soquete não será mais usável e deverá ser fechado. Não podemos chamar `connect` novamente no soquete. Na Figura 11.10, veremos que, quando chamamos `connect` em um loop, tentando cada endereço IP para um dado host até que um funcione, a cada vez que `connect` falhar, devemos fechar com `close` o descritor de soquete e chamar `socket` novamente.

4.4 Função `bind`

A função `bind` atribui um endereço de protocolo local a um soquete. Com os protocolos de Internet, o endereço de protocolo é a combinação de um endereço IPv4 de 32 bits ou de um endereço IPv6 de 128 bits, junto com um número de porta TCP ou UDP de 16 bits.

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

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

Retorna: 0 se OK, –1 em erro

> Historicamente, a descrição da página man de `bind` informava "`bind` assigns a name to an unnamed socket" ("`bind` atribui um nome a um soquete sem nome"). O uso do termo "nome" é confuso e lembra a conotação de nomes de domínio (Capítulo 11), como `foo.bar.com`. A função `bind` não tem nada a ver com nomes. `bind` atribui um endereço de protocolo a um soquete e o que esse endereço de protocolo significa depende do protocolo.

O segundo argumento é um ponteiro para um endereço específico de protocolo e o terceiro argumento é o tamanho dessa estrutura de endereço. Com o TCP, chamar `bind` permite especificar um número de porta, um endereço IP, ambos ou nenhum.

- Os servidores vinculam suas portas bem-conhecidas quando iniciam. Vimos isso na Figura 1.9. Se um cliente ou um servidor TCP não fizer isso, o kernel escolhe uma porta efêmera para o soquete quando `connect` ou `listen` é chamada. É normal que um cliente TCP deixe o kernel escolher uma porta efêmera, a menos que a aplicação requeira uma porta reservada (Figura 2.10), mas é raro um servidor TCP deixar o kernel escolher uma porta efêmera, uma vez que os servidores são conhecidos pelas suas portas bem-conhecidas.

  > Exceções a essa regra são os servidores RPC (Remote Procedure Call). Normalmente, eles deixam o kernel escolher uma porta efêmera para seus soquetes ouvintes desde que essa porta seja então registrada junto ao mapeador de portas RPC. Os clientes têm de contatar o mapeador de portas para obter a porta efêmera antes que eles possam chamar `connect` para o servidor. Isso também se aplica a servidores RPC que utilizam UDP.

- Um processo pode chamar `bind` para um endereço IP específico no seu soquete. O endereço IP deve pertencer a uma interface no host. Para um cliente TCP, isso atribui o endereço IP de origem que será utilizado pelos datagramas IP enviados ao soquete. Para um servidor TCP, isso restringe o soquete a receber conexões entrantes de cliente destinadas somente a esse endereço IP.

  Normalmente, um cliente TCP não chama `bind` para vincular um endereço IP ao seu soquete. O kernel escolhe o endereço IP de origem quando o soquete está conectado, com base na interface de saída que é utilizada, a qual, por sua vez, está baseada na rota requerida para alcançar o servidor (página 737 do TCPv2).

  Se um servidor TCP não vincular um endereço IP ao seu soquete, o kernel utilizará o endereço IP de destino do SYN do cliente como o endereço IP de origem do servidor (página 943 do TCPv2).

Como dissemos, chamar `bind` permite especificar o endereço IP, a porta, ambos ou nenhum. A Figura 4.6 resume os valores com que configuramos `sin_addr` e `sin_port` ou `sin6_addr` e `sin6_port`, dependendo do resultado desejado.

Se especificarmos um número de porta como 0, o kernel escolhe uma porta efêmera quando `bind` é chamada. Mas, se especificarmos um endereço IP curinga, o kernel não escolhe o endereço IP local até que o soquete esteja conectado (TCP) ou um datagrama seja enviado no soquete (UDP).

O processo especifica		Resultado
Endereço IP	**porta**	
Curinga	0	O kernel escolhe o endereço e a porta IP
Curinga	não-zero	O kernel escolhe o endereço IP, o processo especifica a porta
Endereço IP local	0	O processo especifica o endereço IP, o kernel escolhe a porta
Endereço IP local	não-zero	O processo especifica o endereço e a porta IP

Figura 4.6 Resultado ao especificar o endereço e/ou o número da porta IP para `bind`.

Com o IPv4, o endereço curinga é especificado pela constante `INADDR_ANY`, cujo valor normalmente é 0. Isso instrui o kernel a escolher o endereço IP. Vimos o uso disso na Figura 1.9 com a atribuição

```
struct sockaddr_in servaddr;

servaddr.sin_addr.s_addr = htonl(INADDR_ANY); /* curinga */
```

Embora isso funcione com o IPv4, no qual um endereço IP é um valor de 32 bits que pode ser representado como uma constante numérica simples (nesse caso, 0), não podemos utilizar essa técnica com o IPv6, uma vez que o endereço IPv6 de 128 bits é armazenado em uma estrutura. (Em C não podemos representar uma estrutura de constantes no lado direito de uma atribuição.) Para resolver esse problema, escrevemos

```
struct sockaddr_in6 serv;

serv.sin6_addr = in6addr_any; /* curinga */
```

O sistema aloca e inicializa a variável `in6addr_any` para a constante `IN6ADDR_ANY_INIT`. O cabeçalho `<netinet/in.h>` contém a declaração `extern` para `in6addr_any`.

O valor de `INADDR_ANY` (0) é o mesmo na ordem de bytes da rede ou do host, assim o uso de `htonl` não é realmente requerido. Mas, como todas as constantes `INADDR_` definidas pelo cabeçalho `<netinet/in.h>` são definidas na ordem de bytes do host, devemos utilizar `htonl` com qualquer uma dessas constantes.

Se instruirmos o kernel a escolher um número de porta efêmero para nosso soquete, observe que `bind` não retorna o valor escolhido. De fato, ele não pode retornar esse valor uma vez que o segundo argumento para `bind` tem o qualificador `const`. Para obter o valor da porta efêmera atribuído pelo kernel, devemos chamar `getsockname` para retornar o endereço de protocolo.

Um exemplo comum de um processo que vincula um endereço IP não-curinga a um soquete é um host que fornece servidores Web para múltiplas organizações (Seção 14.2 do TCPv3). Primeiro, cada organização tem seu próprio nome de domínio, como `www.`*`organization`*`.com`. Em seguida, cada nome de domínio da organização mapeia para um endereço IP diferente, mas em geral na mesma sub-rede. Por exemplo, se a sub-rede for 198.69.10, o primeiro endereço IP da organização poderia ser 198.69.10.128, o seguinte 198.69.10.129 e assim por diante. Todos esses endereços IP são mapeados (*aliased*) para uma única interface de rede (por exemplo, utilizando a opção `alias` do comando `ifconfig` no 4.4BSD), de modo que a camada IP aceitará datagramas entrantes destinados a qualquer um dos endereços mapeados (*aliased*). Por fim, uma cópia do servidor HTTP é inicializada para cada organização e cada cópia chama `bind` somente no endereço IP dessa organização.

> Uma técnica alternativa é executar um único servidor que vincula o endereço curinga. Quando uma conexão chega, o servidor chama `getsockname` para obter o endereço IP de destino do cliente que, na nossa discussão anterior, poderia ser 198.69.10.128, 198.69.10.129 e assim por diante. O servidor trata então a solicitação de cliente com base no endereço IP para o qual a conexão foi emitida.
>
> Uma vantagem em vincular um endereço IP não-curinga é que a desmultiplexação de um dado endereço IP de destino para um dado processo servidor é então feita pelo kernel.
>
> Devemos ter cuidado de distinguir entre a interface em que um pacote chega e o endereço IP de destino desse pacote. Na Seção 8.8, discutiremos o modelo de sistema final fraco e o modelo de sistema final forte. A maioria das implementações emprega o primeiro sistema, o que significa que não há problema se um pacote chegar com um endereço IP de destino que identifica uma outra interface além da interface em que ele chega. (Isso supõe um host multihomed.) O ato de vincular um endereço IP não-curinga restringe os datagramas que serão entregues ao soquete com base somente no endereço IP de destino. Esse ato não diz nada sobre a interface que chega, a menos que o host empregue o modelo de sistema final forte.

Um erro comum de `bind` é `EADDRINUSE` ("Endereço já em uso"). Discutiremos isso em mais detalhes na Seção 7.5, quando comentaremos as opções de soquete `SO_REUSEADDR` e `SO_REUSEPORT`.

4.5 Função `listen`

A função `listen` é chamada somente por um servidor TCP e realiza duas ações:

1. Quando um soquete é criado pela função `socket`, assume-se que é um soquete ativo, isto é, um soquete de cliente que emitirá uma `connect`. A função `listen` converte um soquete não-conectado em um soquete passivo, indicando que o kernel deve aceitar solicitações de conexões entrantes direcionadas a esse soquete. Em termos do diagrama de transição de estado TCP (Figura 2.4), a chamada a `listen` move o soquete do estado CLOSED para o estado LISTEN.
2. O segundo argumento para essa função especifica o número máximo de conexões que o kernel deve enfileirar para esse soquete.

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

int listen (int sockfd, int backlog);

                                                    Retorna: 1 se OK, –1 em erro
```

Essa função normalmente é chamada depois das duas funções `socket` e `bind` e deve ser chamada antes de chamar a função `accept`.

Para entender o argumento *backlog*, devemos observar que para um dado soquete ouvinte, o kernel mantém duas filas:

1. Uma fila de conexões não-completadas, que contém uma entrada para cada SYN que chegou de um cliente para o qual o servidor espera a conclusão do handshake de três vias do TCP. Esses soquetes estão no estado SYN_RCVD (Figura 2.4).
2. Uma fila de conexões completadas, que contém uma entrada para cada cliente com o qual o handshake de três vias do TCP foi completado. Esses soquetes estão no estado ESTABLISHED (Figura 2.4).

A Figura 4.7 representa essas duas filas para um dado soquete ouvinte.

Quando uma entrada é criada na fila não-completada, os parâmetros no soquete ouvinte são copiados para a conexão recém-criada. O mecanismo de criação de conexão é completamente automático; o processo servidor não é envolvido. A Figura 4.8 representa os pacotes trocados durante o estabelecimento de conexão com essas duas filas.

Quando um SYN chega de um cliente, o TCP cria uma nova entrada na fila não-completada e então responde com o segundo segmento do handshake de três vias: o SYN do servidor com um ACK ao SYN do cliente (Seção 2.6). Essa entrada permanecerá na fila não-completada até que o terceiro segmento do handshake de três vias chegue (o ACK do cliente ao SYN do servidor) ou até que a entrada expire. (Implementações derivadas do Berkeley têm um tempo-limite de 75 segundos para essas entradas não-completadas.) Se o handshake de três vias for completado normalmente, a entrada é movida da fila não-completada para o final da fila completada. Quando o processo chama `accept`, que descreveremos na próxima seção, a primeira entrada na fila completada retorna para o processo, ou, se a fila estiver vazia, o processo é colocado em repouso até que uma entrada seja colocada na fila completada.

Há vários pontos a considerar com relação ao tratamento dessas duas filas.

- O argumento *backlog* da função `listen` especificava historicamente o valor máximo da soma das duas filas.

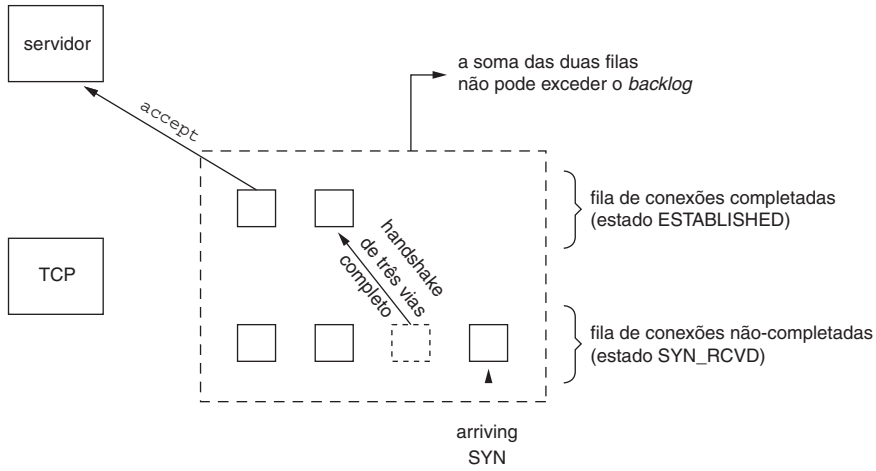


Figura 4.7 As duas filas mantidas pelo TCP para um soquete ouvinte.

Nunca houve uma definição formal sobre o que *backlog* significa. A página man do 4.2BSD diz que ele "define o comprimento máximo que a fila de conexões pendentes pode alcançar". Muitas páginas man e mesmo a especificação `POSIX` copiam essa definição literalmente, mas ela não diz se uma conexão pendente é uma no estado SYN_RCVD, uma no estado ESTABLISHED que ainda não foi aceita, ou ambas. A definição histórica nesse marcador é a implementação Berkeley, que remonta ao 4.2BSD e foi copiada por muitas outras.

- Implementações derivadas do Berkeley adicionam um "fator de tolerância" ("fudge factor") ao *backlog*: ele é multiplicado por 1,5 (página 257 do TCPv1 e página 462 do TCPv2). Por exemplo, o *backlog* comumente especificado em 5 realmente permite até 8 entradas enfileiradas nesses sistemas, como mostramos na Figura 4.10.

  A razão para adicionar esse fator de tolerância parece perdida na história (Joy, 1994). Mas, se considerarmos o *backlog* como especificando o número máximo de conexões completadas que o kernel irá enfileirar para um soquete ([Borman, 1997b], como discutido em breve), então a razão do fator de tolerância é levar em conta as conexões não-completadas na fila.

- Não especifique um *backlog* de 0, uma vez que diferentes implementações interpretam isso de maneira diferente (Figura 4.10). Se você não quiser que nenhum cliente se conecte ao seu soquete ouvinte, feche-o.

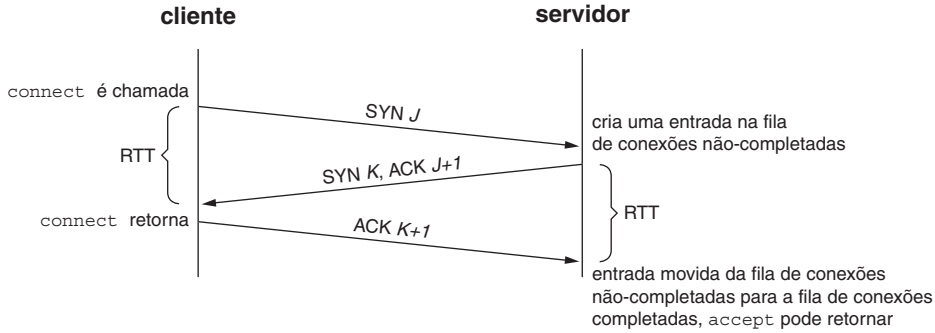


Figura 4.8 O handshake de três vias do TCP e as duas filas para um soquete ouvinte.

- Assumindo que o handshake de três vias é completado normalmente (isto é, nenhum segmento perdido e nenhuma retransmissão), uma entrada permanece na fila de conexões não-completadas por um RTT, qualquer que seja o valor entre um cliente e um servidor particular. A Seção 14.4 do TCPv3 mostra que, para um servidor Web, o RTT mediano entre muitos clientes e o servidor era 187 ms. (O mediano é freqüentemente utilizado para esse cálculo, uma vez que alguns valores grandes podem distorcer notavelmente a média.)
- Historicamente, o código de exemplo sempre mostra um *backlog* de 5, uma vez que esse era o valor máximo suportado pelo 4.2BSD. Isso era adequado na década de 1980, quando servidores ocupados tratavam somente algumas centenas de conexões por dia. Mas com o crescimento da World Wide Web (WWW), em que servidores ocupados tratam milhões de conexões por dia, esse número pequeno é completamente inadequado (páginas 187 a 192 do TCPv3). Servidores HTTP ocupados devem especificar um *backlog* muito maior e kernels mais recentes devem suportar valores maiores.

  Muitos sistemas atuais permitem ao administrador modificar o valor máximo do *backlog*.
- Um problema é: que valor a aplicação deve especificar para o *backlog*, visto que 5 é freqüentemente inadequado? Não há uma resposta fácil para essa questão. Servidores HTTP agora especificam um valor maior, mas se o valor especificado for uma constante no código-fonte, aumentar o valor da constante exigiria recompilar o servidor. Um outro método é assumir algum valor default, mas permitir que uma opção de linha de comando ou uma variável de ambiente sobrescreva o default. Sempre é aceitável especificar um valor maior que o suportado pelo kernel, uma vez que o kernel deve silenciosamente truncar o valor para o máximo que ele suporta, sem retornar um erro (página 456 do TCPv2).

  Podemos fornecer uma solução simples para esse problema modificando nossa função empacotadora para a função `listen`. A Figura 4.9 mostra o código real. Permitimos que a variável de ambiente `LISTENQ` sobrescreva o valor especificado pelo chamador.

lib/wrapsock.c

```
137 void
138 Listen(int fd, int backlog)
139 {
140     char *ptr;

141         /* pode sobrescrever o segundo argumento com a variável de ambiente */
142     if ( (ptr = getenv("LISTENQ")) != NULL)
143         backlog = atoi(ptr);

144     if (listen(fd, backlog) < 0)
145         err_sys("listen error");
146 }
```

lib/wrapsock.c

Figura 4.9 Função empacotadora para `listen` que permite a uma variável de ambiente especificar o *backlog*.

- Historicamente, manuais e livros têm informado que a razão para enfileirar um número fixo de conexões é tratar o caso de o processo servidor estar ocupado entre chamadas sucessivas a `accept`. Isso implica que, entre as duas filas, a completada normalmente deve ter mais entradas que a não-incompletada. Novamente, servidores Web ocupados mostraram que isso é falso. A razão para especificar um *backlog* grande é porque a fila de conexões não-completadas pode crescer à medida que chegam SYNs de cliente, esperando pela conclusão do handshake de três vias.
- Se as filas estiverem cheias quando um SYN de cliente chegar, o TCP ignora o SYN que chega (páginas 930 e 931 do TCPv2); ele não envia um RST. Isso ocorre porque a condição é considerada temporária e o TCP cliente retransmitirá seu SYN, na esperança de encontrar um lugar na fila em um futuro próximo. Se o TCP de servidor respondeu imediatamente

com um RST, o `connect` do cliente retornaria um erro, forçando a aplicação a tratar essa condição em vez de deixar que a retransmissão normal de TCP assuma. Além disso, o cliente não poderia diferenciar entre um RST em resposta a um SYN, o que significa "não há nenhum servidor nessa porta" *versus* "há um servidor nessa porta, mas suas filas estão cheias".

> Algumas implementações enviam um RST quando a fila está cheia. Esse comportamento é incorreto devido às razões declaradas anteriormente e, a menos que seu cliente precise interagir especificamente com esse servidor, é melhor ignorar tal possibilidade. A codificação para tratar esse caso reduz a robustez do cliente e impõe mais carga na rede no caso do RST normal, em que a porta realmente não tem nenhum servidor ouvindo nela.

- Os dados que chegam depois que o handshake de três vias se completa, mas antes de o servidor chamar `accept`, devem ser enfileirados pelo TCP do servidor, de acordo com o tamanho do buffer de recebimento do soquete conectado.

A Figura 4.10 mostra o número real de conexões enfileiradas fornecido aos diferentes valores do argumento *backlog* para os vários sistemas operacionais na Figura 1.16. Para sete sistemas operacionais diferentes há cinco colunas distintas, mostrando a variedade de interpretações sobre o que *backlog* significa!

O AIX e o MacOS têm o algoritmo tradicional do Berkeley; e o Solaris parece estar também muito próximo desse algoritmo. O FreeBSD somente adiciona um ao *backlog*.

> O programa para medir esses valores é mostrado na solução do Exercício 15.4.
>
> Como dissemos, historicamente o *backlog* especificava o valor máximo para a soma das duas filas. Em 1996, um novo tipo de ataque foi lançado na Internet, chamado *SYN flooding ("inundação de SYN")* (CERT, 1996b). O hacker escreve um programa para enviar SYNs à vítima a uma alta velocidade, preenchendo a fila de conexões não-completadas de uma ou mais portas TCP. (Utilizamos o termo *hacker* com o significado de invasor, como descrito em [Cheswick, Bellovin e Rubin, 2003]). Além disso, o endereço IP de origem de cada SYN é configurado como um número aleatório (isso é chamado *spoofing de IP*), de modo que o SYN/ACK do servidor não vá a nenhum lugar. Isso também impede que o servidor saiba o endereço real do hacker. Preenchendo a fila de conexões não-completadas com SYNs falsos, os SYNs legítimos não são enfileirados, fornecendo uma *negação de serviço* aos clientes legítimos. Há dois métodos comumente utilizados para tratar esses ataques, resumidos em Borman (1997b). Mas o que é mais interessante nessa nota é revisitar o que o *backlog* de `listen` realmente significa. Ele deve especificar o número máximo de cone-

backlog	Número máximo real de conexões enfileiradas				
	MacOS 10.2.6 AIX 5.1	Linux 2.4.7	HP-UX 11.11	FreeBSD 4.8 FreeBSD 5.1	Solaris 2.9
0	1	3	1	1	1
1	2	4	1	2	2
2	4	5	3	3	4
3	5	6	4	4	5
4	7	7	6	5	6
5	8	8	7	6	8
6	10	9	9	7	10
7	11	10	10	8	11
8	13	11	12	9	13
9	14	12	13	10	14
10	16	13	15	11	16
11	17	14	16	12	17
12	19	15	18	13	19
13	20	16	19	14	20
14	22	17	21	15	22

Figura 4.10 Número real de conexões enfileiradas para valores de *backlog*.

xões *completadas* para um dado soquete que o kernel irá enfileirar. O propósito de ter um limite para essas conexões completadas é fazer com que o kernel pare de aceitar novas solicitações de conexão para um dado soquete quando a aplicação não as está aceitando (por qualquer que seja a razão). Se um sistema implementar essa interpretação, como faz o BSD/OS 3.0, a aplicação não precisará especificar valores de *backlog* enormes simplesmente porque o servidor trata uma grande quantidade de solicitações de cliente (por exemplo, um servidor da Web ocupado) ou para fornecer proteção contra SYN Flooding. O kernel trata uma grande quantidade de conexões não-completadas, independentemente de elas serem legítimas ou provenientes de um hacker. Mas, mesmo com essa interpretação, ocorrem cenários em que o valor tradicional de 5 é inadequado.

4.6 Função `accept`

`accept` é chamada por um servidor TCP para retornar a próxima conexão completada a partir do início da fila de conexões completadas (Figura 4.7). Se a fila de conexões completadas estiver vazia, o processo é colocado em repouso (assumindo o default de um soquete bloqueador). Os argumentos *cliaddr* e *addrlen* são utilizados para retornar o endereço de protocolo do processo do peer conectado (o cliente). *addrlen* é um argumento valor-resultado (Seção 3.3): antes da chamada, configuramos o valor inteiro referenciado por **addrlen* como o tamanho da estrutura de endereço de soquete apontada por *cliaddr;* ao retornar, esse valor de inteiro contém o número real de bytes armazenado pelo kernel na estrutura de endereço de soquete.

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

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

Retorna: descritor não-negativo se OK, –1 em erro

Se `accept` for bem-sucedido, seu valor de retorno é um descritor novo em folha automaticamente criado pelo kernel. Esse novo descritor referencia a conexão TCP com o cliente. Ao discutir `accept`, denominamos o primeiro argumento para `accept` de *soquete ouvinte* (o descritor criado pelo soquete e então utilizado como o primeiro argumento tanto para `bind` como para `listen`) e denominamos de *soquete conectado* o valor de retorno de `accept`. É importante diferenciar esses dois soquetes. Normalmente, um dado servidor cria somente um soquete ouvinte, que então existe pelo tempo de vida do servidor. O kernel cria um soquete conectado para cada conexão de cliente que é aceita (isto é, para o qual o handshake de três vias de TCP é completado). Quando o servidor termina de atender um dado cliente, o soquete conectado é fechado.

Essa função retorna até três valores: um código de retorno inteiro que é um novo descritor de soquete ou uma indicação de erro, o endereço de protocolo do processo cliente (por meio do ponteiro *cliaddr*) e o tamanho desse endereço (por meio do ponteiro *addrlen*). Se não houver interesse em manter o endereço de protocolo do cliente retornado, configuramos *cliaddr* e *addrlen* como ponteiros nulos.

A Figura 1.9 mostra esses pontos. O soquete conectado é fechado a cada passagem pelo loop, mas o soquete ouvinte permanece aberto pelo tempo de vida do servidor. Também vemos que o segundo e o terceiro argumentos para `accept` são ponteiros nulos, uma vez que não temos interesse na identidade do cliente.

Exemplo: argumentos valor-resultado

Agora, mostraremos como tratar o argumento resultado de valor para `accept` modificando o código na Figura 1.9 para imprimir o endereço e a porta IP do cliente. Mostramos isso na Figura 4.11.

intro/daytimetcpsrv1.c

```
 1 #include   "unp.h"
 2 #include   <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int    listenfd, connfd;
 7     socklen_t len;
 8     struct sockaddr_in servaddr, cliaddr;
 9     char     buff[MAXLINE];
10     time_t   ticks;

11     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

12     bzero(&servaddr, sizeof(servaddr));
13     servaddr.sin_family = AF_INET;
14     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
15     servaddr.sin_port = htons(13);  /* servidor de data/hora */

16     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

17     Listen(listenfd, LISTENQ);

18     for ( ; ; ) {
19         len = sizeof(cliaddr);
20         connfd = Accept(listenfd, (SA *) &cliaddr, &len);
21         printf("connection from %s, port %d\n",
22                Inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof(buff)),
23                ntohs(cliaddr.sin_port));

24         ticks = time(NULL);
25         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
26         Write(connfd, buff, strlen(buff));

27         Close(connfd);
28     }
29 }
```

intro/daytimetcpsrv1.c

Figura 4.11 Servidor de data/hora que imprime o endereço e a porta IP do cliente.

Novas declarações

7-8 Definimos duas novas variáveis: `len`, que será uma variável de valor-resultado, e `cliaddr`, que conterá o endereço de protocolo do cliente.

Aceitação da conexão e impressão do endereço do cliente

19-23 Inicializamos `len` para o tamanho da estrutura de endereço de soquete e passamos um ponteiro para a estrutura `cliaddr` e um ponteiro para `len` como o segundo e o terceiro argumentos para `accept`. Chamamos `inet_ntop` (Seção 3.7) a fim de converter o endereço IP de 32 bits na estrutura de endereço de soquete para uma string ASCII de ponto decimal e chamamos `ntohs` (Seção 3.4) a fim de converter o número da porta de 16 bits proveniente da ordem de bytes da rede ou do host.

> Chamar `sock_ntop` em vez de `inet_ntop` tornaria nosso servidor mais independente de protocolo, mas esse servidor já é dependente do IPv4. Mostraremos uma versão independente de protocolo desse servidor na Figura 11.13.

Se executarmos nosso novo servidor e então executarmos nosso cliente no mesmo host, conectando nosso servidor duas vezes em seqüência, teremos a seguinte saída do cliente:

```
solaris % daytimetcpcli 127.0.0.1
Thu Sep 11 12:44:00 2003
```

```
solaris % daytimetcpcli 192.168.1.20
Thu Sep 11 12:44:09 2003
```

Primeiro especificamos o endereço IP do servidor como o endereço de loopback (127.0.0.1) e então como seu próprio endereço IP (192.168.1.20). Eis a saída de servidor correspondente:

```
solaris # daytimetcpsrv1
connection from 127.0.0.1, port 43388
connection from 192.168.1.20, port 43389
```

Observe o que acontece com o endereço IP do cliente. Como nosso cliente de data/hora (Figura 1.5) não chama `bind`, dissemos na Seção 4.4 que o kernel escolhe o endereço IP de origem com base na interface de saída que é utilizada. No primeiro caso, o kernel configura o endereço IP de origem como o endereço de loopback; no segundo caso, ele configura o endereço como o endereço IP da interface Ethernet. Também podemos ver nesse exemplo que a porta efêmera escolhida pelo kernel do Solaris é 43388 e então 43389 (lembre-se da Figura 2.10).

Como um ponto final, nosso prompt de shell para o script de servidor muda para o sinal de libra (#), o prompt comumente utilizado para o superusuário. Nosso servidor deve executar com privilégios de superusuário para chamar `bind` na porta reservada de 13. Se não tivermos privilégios de superusuário, a chamada para `bind` irá falhar:

```
solaris % daytimetcpsrv1
bind error: Permission denied
```

4.7 Funções `fork` e `exec`

Antes de discutir como escrever um servidor concorrente na próxima seção, devemos discutir a função `fork` do Unix. Essa função (incluindo suas variantes fornecidas por alguns sistemas) é a única maneira no Unix de criar um novo processo.

```
#include <unistd.h>

pid_t fork(void);
```

Retorna: 0 no filho, ID de processo do filho no pai, –1 em erro

Se você nunca viu essa função antes, a parte difícil para entender a função `fork` é que ela é chamada *uma* vez, mas retorna *duas* vezes. Ela retorna uma vez no processo chamador (denominado pai) com um valor de retorno que é o ID do processo recém-criado (o filho). Ela também retorna uma vez no filho, com um valor de retorno de 0. Conseqüentemente, o valor de retorno informa ao processo se ele é o pai ou o filho.

A razão por que `fork` retorna 0 no filho, em vez de o ID de processo do pai, é porque o filho tem somente um pai e sempre pode obter o ID de processo do pai chamando `getppid`. Um pai, por outro lado, pode ter um número qualquer de filhos e não há nenhuma maneira de obter os IDs de processo deles. Se um pai quiser monitorar os IDs de processo de todos os seus filhos, ele deve registrar os valores de retorno de `fork`.

Todos os descritores abertos no pai antes da chamada a `fork` são compartilhados com o filho depois que `fork` retorna. Veremos este recurso utilizado pelos servidores de rede: o pai chama `accept` e então chama `fork`. O soquete conectado é então compartilhado entre o pai e o filho. Normalmente, o filho então lê e grava o soquete conectado e o pai fecha o soquete conectado.

Há duas utilizações típicas de `fork`:

1. Um processo faz uma cópia de si próprio de modo que uma cópia possa tratar uma operação enquanto a outra cópia realiza uma outra tarefa. Isso é típico para servidores de rede. Veremos vários exemplos disso mais adiante neste capítulo.

2. Um processo quer executar um outro programa. Como a única maneira de criar um novo processo é chamando `fork`, o processo primeiro chama `fork` para fazer uma cópia de si próprio e então uma das cópias (em geral o processo-filho) chama `exec` (descrito a seguir) para substituir a si próprio pelo novo programa. Isso é típico para programas como shells.

A única maneira pela qual um arquivo de um programa executável em disco pode ser executado pelo Unix é fazer com que um processo existente chame uma das seis funções `exec`. (Freqüentemente iremos nos referir genericamente à função "`exec`" quando não importa qual das seis funções é chamada.) `exec` substitui a imagem do processo atual pelo novo arquivo de programa, e esse novo programa normalmente inicia na função `main`. O ID de processo não muda. Referimo-nos ao processo que chama `exec` como *processo chamador* (*calling process*) e ao novo programa que é executado como *novo programa.*

> Manuais e livros mais antigos referem-se incorretamente ao novo programa como *novo processo*, o que é errado, porque um novo processo não é criado.

As diferenças entre as seis funções `exec` são: (a) se o arquivo de programa a executar for especificado por um *nome de arquivo* ou por um *nome de caminho*; (b) se os argumentos do novo programa forem listados um a um ou referenciados por um array de ponteiros; e (c) se o ambiente do processo que chama for passado para o novo programa ou se um novo ambiente for especificado.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *) 0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
              /* (char *) 0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *) 0 */ );

int execvp(const char *filename, char *const argv[]);
```

Todas as seis funções retornam: –1 em erro, nenhum retorno em caso de sucesso

Essas funções retornam ao chamador somente se ocorrer um erro. Caso contrário, o controle passa para o início do novo programa, normalmente para a função `main`.

O relacionamento entre essas seis funções é mostrado na Figura 4.12. Normalmente, somente `execve` é uma chamada de sistema dentro do kernel e as outras cinco são funções de biblioteca que chamam `execve`.

Observe as diferenças a seguir entre essas seis funções:

1. As três funções na linha superior especificam cada string de argumento como um argumento separado para a função `exec`, com um ponteiro nulo terminando o número variável de argumentos. As três funções na segunda linha têm um array *argv*, contendo ponteiros para as strings de argumento. Esse array *argv* deve conter um ponteiro nulo para especificar seu final, uma vez que uma contagem não é especificada.
2. As duas funções na coluna mais à esquerda especificam um argumento de *nome de arquivo*. Esse argumento é convertido em um *nome de caminho* utilizando a variável de ambiente `PATH` atual. Se o argumento de *nome de arquivo* para `execlp` ou `execvp` contiver uma barra (/) em qualquer lugar na string, a variável `PATH` não é uti-

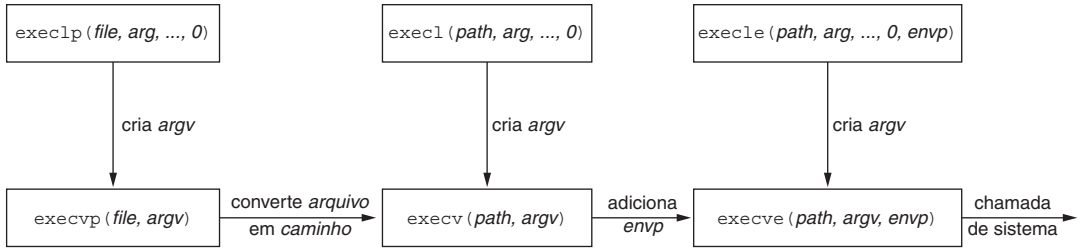


Figura 4.12 Relacionamento entre as seis funções `exec`.

lizada. As quatro funções nas duas colunas à direita especificam um argumento de *nome de caminho* completamente qualificado.

3. As quatro funções nas duas colunas à esquerda não especificam um ponteiro de ambiente explícito. Em vez disso, o valor atual da variável externa `environ` é utilizado para criar uma lista de ambientes que é passada para o novo programa. As duas funções na coluna mais à direita especificam uma lista de ambientes explícita. O array *envp* de ponteiros deve ser terminado por um ponteiro nulo.

Os descritores abertos no processo antes de chamar `exec` normalmente permanecem abertos por todo o `exec`. Utilizamos o qualificador "normalmente" porque isso pode ser desativado utilizando `fcntl` para configurar o flag de descritor `FD_CLOEXEC`. O servidor `inetd` utiliza esse recurso, como descreveremos na Seção 13.5.

4.8 Servidores concorrentes

O servidor na Figura 4.11 é um *servidor iterativo*. Para algo tão simples como um servidor de data/hora, isso é suficiente. Mas, se uma solicitação de cliente levar mais tempo para ser atendida, não é recomendável associar um único servidor a um cliente; queremos tratar múltiplos clientes ao mesmo tempo. A maneira mais simples de escrever *um servidor concorrente* no Unix é chamar `fork` criando um processo-filho para tratar cada cliente. A Figura 4.13 mostra a estrutura básica de um servidor concorrente típico.

```
pid_t pid;
int   listenfd, connfd;

listenfd = Socket( ... );

   /* preenche sockaddr_in{} com a porta bem-conhecida do servidor */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);

for ( ; ; ) {
    connfd = Accept(listenfd, ... );  /* provavelmente bloqueia */

    if ( (pid = Fork()) == 0) {
      Close(listenfd);      /* filho fecha o soquete ouvinte */
      doit(connfd);         /* processa a solicitação */
      Close(connfd);        /* feito com esse cliente */
      exit(0);              /* filho termina */
    }

    Close(connfd);          /* pai fecha o soquete conectado */
}
```

Figura 4.13 Estrutura básica de servidor concorrente típico.

Quando uma conexão é estabelecida, `accept` retorna, o servidor chama `fork` e o processo-filho atende o cliente (em `connfd`, o soquete conectado) enquanto o processo-pai espera uma outra conexão (em `listenfd`, o soquete ouvinte). O pai fecha o soquete conectado uma vez que o filho trata o novo cliente.

Na Figura 4.13, supomos que a função `doit` faz o que for requerido para atender o cliente. Quando essa função retorna, chamamos `close` para fechar explicitamente o soquete conectado ao filho. Isso não é requerido uma vez que a próxima instrução chama `exit` e parte do término de processo é fechar todos os descritores abertos pelo kernel. Incluir ou não essa chamada explícita para `close` é uma questão de preferência pessoal de programação.

Dissemos na Seção 2.6 que chamar `close` em um soquete TCP faz com que um FIN seja enviado, seguido pela seqüência normal de término de conexão TCP. Por que o `close` de `connfd` na Figura 4.13 realizado pelo pai não termina a conexão com o cliente? Para entender o que está acontecendo, devemos entender que todos os arquivos ou soquetes têm uma contagem de referências. A contagem de referências é mantida na entrada da tabela de arquivos (páginas 57 a 60 do APUE). Essa é uma contagem do número de descritores atualmente abertos que fazem referência a esse arquivo ou soquete. Na Figura 4.13, depois que `socket` retorna, a entrada de tabela de arquivo associada a `listenfd` tem uma contagem de referências de 1. Depois que `accept` retorna, a entrada de tabela de arquivo associada a `connfd` tem uma contagem de referências também de 1. Mas, depois que `fork` retorna, os dois descritores são compartilhados (isto é, duplicados) entre o pai e o filho, assim as entradas da tabela de arquivo associadas aos dois soquetes agora têm uma contagem de referências de 2. Portanto, quando o pai fecha `connfd`, ele apenas decrementa a contagem de referências de 2 para 1, e isso é tudo. A limpeza e a desalocação real do soquete não acontecem até que a contagem de referências alcance 0. Isso ocorrerá algum momento mais tarde quando o filho fechar `connfd`.

Também podemos visualizar os soquetes e a conexão que ocorre na Figura 4.13 como a seguir. Primeiro, a Figura 4.14 mostra o *status* do cliente e do servidor enquanto o servidor está bloqueado na chamada para `accept` e a solicitação de conexão chega do cliente.

Logo depois que `accept` retorna, temos o cenário mostrado na Figura 4.15. A conexão é aceita pelo kernel e um novo soquete, `connfd`, é criado. Esse é um soquete conectado e dados agora podem ser lidos e gravados na conexão.

O próximo passo no servidor concorrente é chamar `fork`. A Figura 4.16 mostra o *status* depois que `fork` retorna.

Observe que os dois descritores, `listenfd` e `connfd`, são compartilhados (duplicados) entre o pai e o filho.

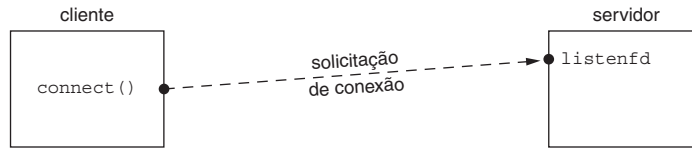


Figura 4.14 O *status* de cliente/servidor antes de a chamada a `accept` retornar.

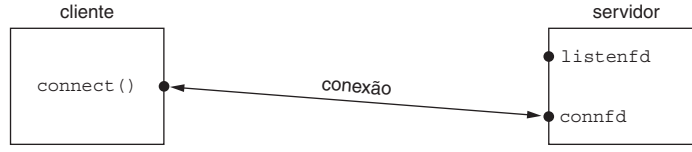


Figura 4.15 O *status* de cliente/servidor após o retorno de `accept`.

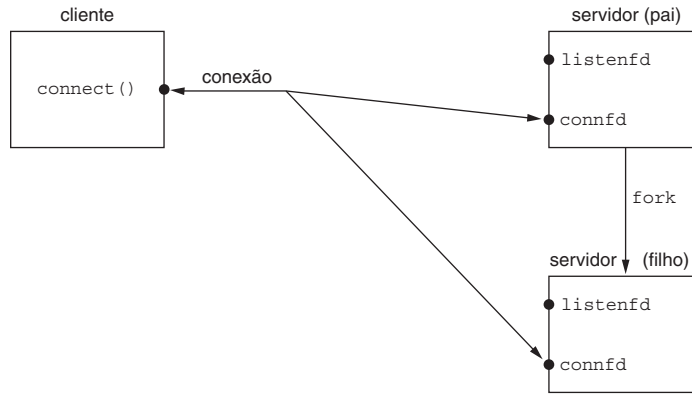


Figura 4.16 O *status* de cliente/servidor depois que `fork` retorna.

O próximo passo é fazer com que o pai feche o soquete conectado e o filho feche o soquete ouvinte. Isso é mostrado na Figura 4.17.

Esse é o estado final desejado dos soquetes. O filho trata a conexão com o cliente e o pai pode chamar `accept` novamente no soquete ouvinte, para tratar a próxima conexão de cliente.

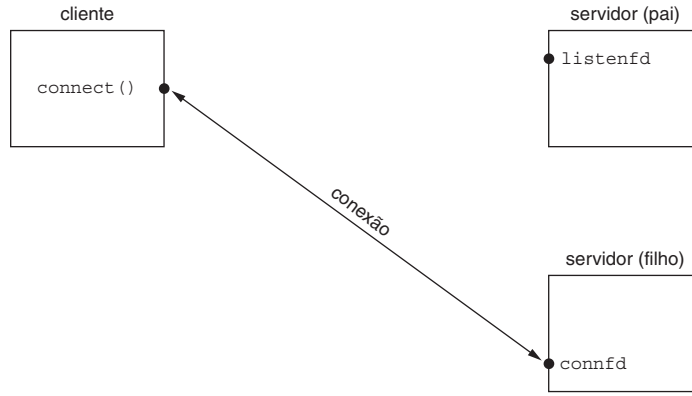


Figura 4.17 O *status* de cliente/servidor depois que pai e filho fecham os soquetes apropriados.

4.9 Função `close`

A função `close` normal do Unix também é utilizada para fechar um soquete e terminar uma conexão TCP.

```
#include <unistd.h>

int close(int sockfd);

                                                    Retorna: 0 se OK, –1 em erro
```

A ação default de `close` com um soquete TCP é marcar o soquete como fechado e retornar ao processo imediatamente. O descritor de soquete não é mais utilizável pelo processo:

ele não pode ser utilizado como um argumento para `read` ou `write`. Mas o TCP tentará enviar quaisquer dados que já estão enfileirados para serem enviados a outra extremidade e, depois que isso ocorre, acontece a seqüência normal de término da conexão TCP (Seção 2.6).

Na Seção 7.5, descreveremos a opção de soquete `SO_LINGER`, que permite alterar essa ação default com um soquete TCP. Nessa seção, também descreveremos o que uma aplicação TCP deve fazer para ter garantias de que a aplicação do peer recebeu quaisquer dados pendentes.

Contagens de referência do descritor

No final da Seção 4.8, mencionamos que quando o processo-pai no nosso servidor concorrente fecha o soquete conectado chamando `close`, isso apenas diminui a contagem de referências para o descritor. Como a contagem de referências ainda era maior que 0, essa chamada a `close` não iniciou a seqüência de término da conexão de quatro pacotes do TCP. Esse é o comportamento que queremos no nosso servidor concorrente com o soquete conectado que é compartilhado entre o pai e o filho.

Se realmente quisermos enviar um FIN em uma conexão TCP, a função `shutdown` pode ser utilizada (Seção 6.6) em vez de `close`. Descreveremos a razão disso na Seção 6.5.

Também devemos estar cientes do que acontece no nosso servidor concorrente se o pai não chamar `close` para cada soquete conectado retornado por `accept`. Primeiro, o pai em algum momento ficará sem descritores, pois normalmente há um limite quanto ao número de descritores que qualquer processo pode ter abertos a qualquer momento. Mas, sobretudo, nenhuma conexão de cliente será terminada. Quando o filho fecha o soquete conectado, sua contagem de referências passará de 2 para 1 e permanecerá em 1 uma vez que o pai nunca chama `close` para fechar o soquete conectado. Isso impedirá que a seqüência de término de conexão do TCP ocorra, e a conexão permanecerá aberta.

4.10 Funções `getsockname` e `getpeername`

Essas duas funções retornam o endereço local de protocolo associado a um soquete (`getsockname`) ou o endereço externo de protocolo associado a um soquete (`getpeername`).

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

int getsockname (int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);

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

As duas retornam: 0 se OK, –1 em erro

Observe que o argumento final para as duas funções é um argumento valor-resultado. Isto é, as duas funções preenchem a estrutura de endereço de soquete apontada por *localaddr* ou *peeraddr*.

> Mencionamos na nossa discussão sobre `bind` que o termo "nome" induz a erros. Essas duas funções retornam o endereço do protocolo associado a uma das duas extremidades de uma conexão de rede, o que para o IPv4 e o IPv6 é a combinação de um endereço e número de porta IP. Essas funções não têm nada a ver com nomes de domínio (Capítulo 11).

Essas duas funções são requeridas pelas seguintes razões:

- Depois que `connect` retorna com sucesso em um cliente TCP que não chama `bind`, `getsockname` retorna o endereço IP local e o número de porta local atribuído à conexão pelo kernel.

- Depois de chamar `bind` com um número de porta de 0 (instruindo o kernel a escolher o número da porta local), `getsockname` retorna o número da porta local que foi atribuído.
- `getsockname` pode ser chamado para obter a família de endereços de um soquete, como mostramos na Figura 4.19.
- Em um servidor TCP que vincula o endereço IP curinga chamando `bind` (Figura 1.9), depois que uma conexão é estabelecida com um cliente (`accept` retorna com sucesso), o servidor pode chamar `getsockname` para obter o endereço IP local atribuído à conexão. O argumento descritor do soquete nessa chamada deve ser o do soquete conectado e não o do soquete ouvinte.
- Quando `execed` é executado em um servidor pelo processo que chama `accept`, a única maneira como o servidor pode obter a identidade do cliente é chamando `getpeername`. Isso é o que acontece sempre que `inetd` (Seção 13.5) bifurca, chamando `fork`, e executa, chamando `exec`, um servidor TCP. A Figura 4.18 mostra esse cenário. `inetd` chama `accept` (caixa superior esquerda) e retorna dois valores: o descritor do soquete conectado, `connfd`, é o valor de retorno da função; e a pequena caixa que rotulamos "endereço do peer" (uma estrutura de endereço de soquete de Internet) contém o endereço e o número da porta IP do cliente. `fork` é chamado e um filho de `inetd` é criado. Como o filho inicia com uma cópia da imagem da memória do pai, a estrutura de endereço de soquete torna-se disponível a ele, assim como o é o descritor do soquete conectado (uma vez que os descritores são compartilhados entre o pai e o filho). Mas, quando o filho chama `exec` para executar um servidor real (digamos o servidor de Telnet que mostramos), a imagem de memória do filho é substituída pelo novo arquivo de programa do servidor de Telnet (isto é, a estrutura de endereço de soquete contendo o endereço do peer é perdida) e o descritor do soquete conectado permanece aberto por todo o `exec`. Uma das primeiras chamadas de função realizadas pelo servidor de Telnet é `getpeername` para obter o endereço e o número da porta IP do cliente.

Obviamente, o servidor de Telnet nesse exemplo final deve conhecer o valor de `connfd` quando ele inicia. Há duas maneiras comuns de fazer isso. Na primeira delas, o processo que chama `exec` pode formatar o número do descritor como uma string de caracteres e passá-lo como um argumento de linha de comando para o novo programa sendo executado. Alternativamente, uma convenção pode ser estabelecida para que um certo descritor sempre seja con-

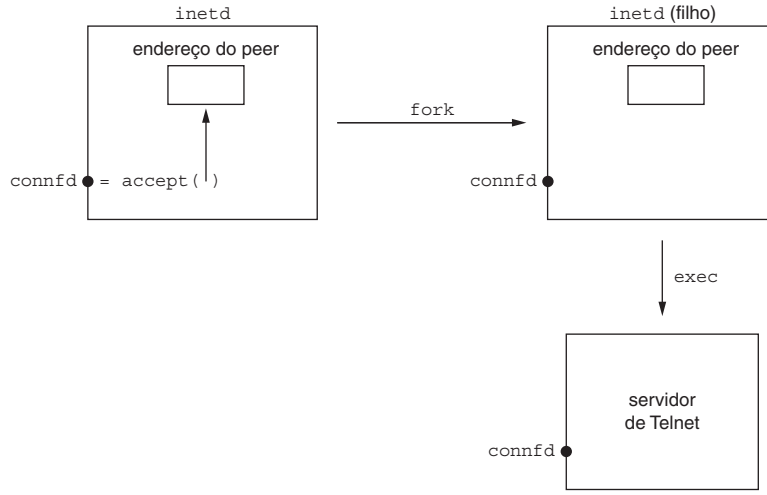


Figura 4.18 Exemplo de `inetd` gerando um servidor.

figurado como o soquete conectado antes de chamar `exec`. A outra é o que `inetd` faz, sempre configura os descritores 0, 1 e 2 como o soquete conectado.

Exemplo: obtendo a família de endereços de um soquete

A função `sockfd_to_family` mostrada na Figura 4.19 retorna a família de endereços de um soquete.

lib/sockfd_to_family.c

```
 1 #include   "unp.h"

 2 int
 3 sockfd_to_family(int sockfd)
 4 {
 5     struct sockaddr_storage ss;
 6     socklen_t len;

 7     len = sizeof(ss);
 8     if (getsockname(sockfd, (SA *) &ss, &len) < 0)
 9         return (-1);
10     return (ss.ss_family);
11 }
```

lib/sockfd_to_family.c

Figura 4.19 Retorna a família de endereços de um soquete.

Alocação de espaço para a maior estrutura de endereço de soquete

5 Como não sabemos que tipo de estrutura de endereço de soquete alocar, utilizamos um valor de `sockaddr_storage`, uma vez que ele pode armazenar qualquer estrutura de endereço de soquete suportado pelo sistema.

Chamada de `getsockname`

7-10 Chamamos `getsockname` e retornamos a família de endereços.

Como a especificação POSIX permite uma chamada a `getsockname` em um soquete desvinculado, essa função deve funcionar para qualquer descritor de soquete aberto.

4.11 Resumo

Todos os clientes e servidores iniciam com uma chamada a `socket`, retornando um descritor de soquete. Os clientes chamam então `connect` e os servidores chamam `bind`, `listen` e `accept`. Normalmente, os soquetes são fechados com a função `close` padrão, apesar de que veremos uma outra maneira de fazer isso com a função `shutdown` (Seção 6.6) e também examinaremos o efeito da opção de soquete `SO_LINGER` (Seção 7.5).

A maioria dos servidores TCP é concorrente, com o servidor chamando `fork` para cada conexão de cliente que ele trata. Veremos que a maioria dos servidores UDP é iterativa. Embora esses dois modelos tenham sido utilizados com sucesso por vários anos, no Capítulo 30 veremos outras opções de *design* de servidor que utilizam threads e processos.

Exercícios

4.1 Na Seção 4.4, afirmamos que as constantes `INADDR_` definidas pelo cabeçalho `<netinet/in.h>` estão na ordem de bytes do host. Como podemos afirmar isso?

4.2 Modifique a Figura 1.5 para chamar `getsockname` depois que `connect` retorna com sucesso. Imprima o endereço e a porta local IP atribuída ao soquete TCP utilizando `sock_ntop`. Em que intervalo (Figura 2.10) estão as portas efêmeras do seu sistema?

4.3 Em um servidor concorrente, suponha que o filho é executado primeiro depois da chamada a `fork`. O filho completa então o serviço do cliente antes que a chamada para `fork` retorne ao pai. O que acontece nas duas chamadas a `close` na Figura 4.13?

4.4 Na Figura 4.11, primeiro altere a porta do servidor de 13 para 9999 (de modo que não haja necessidade de privilégios de superusuário para iniciar o programa). Remova a chamada a `listen`. O que acontece?

4.5 Prossiga com o exercício anterior. Remova a chamada a `bind`, mas permita a chamada a `listen`. O que acontece?

Capítulo 5

Exemplo de Cliente/Servidor TCP

5.1 Visão geral

Agora, utilizaremos as funções elementares do capítulo anterior para escrever um exemplo de cliente/servidor TCP completo. Nosso exemplo simples é um servidor de eco que realiza os seguintes procedimentos:

1. O cliente lê uma linha de texto a partir da sua entrada-padrão e grava essa linha no servidor.
2. O servidor lê a linha a partir da sua entrada de rede e ecoa a linha de volta para o cliente.
3. O cliente lê a linha ecoada e a imprime na sua saída-padrão.

A Figura 5.1 representa esse cliente/servidor simples junto com as funções utilizadas para entrada e saída.

Mostramos duas setas entre o cliente e o servidor, mas isso é na realidade uma conexão TCP full-duplex. As funções `fgets` e `fputs` são provenientes da biblioteca de E/S-padrão, as funções `writen` e `readline` foram mostradas na Seção 3.9.

Desenvolveremos nossa própria implementação de um servidor de eco utilizando tanto o TCP como o UDP, embora a maioria das implementações TCP/IP forneça esse servidor (Seção 2.12). Também utilizaremos esse servidor com o nosso próprio cliente.

Um cliente/servidor que ecoa linhas de entrada é um exemplo válido, ainda que simples, de uma aplicação de rede. Todos os passos básicos requeridos para implementar qualquer cliente/servidor são ilustrados por meio desse exemplo. Para expandir esse exemplo para sua

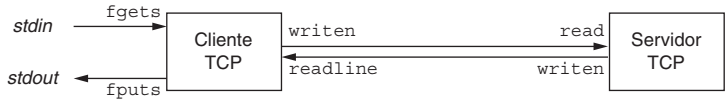


Figura 5.1 Cliente e servidor de eco simples.

própria aplicação, tudo o que você precisa fazer é alterar o que o servidor faz com a entrada que ele recebe dos seus clientes.

Além de executar nosso cliente e servidor no modo normal (digite uma linha e observe-o ecoar), examinaremos várias condições-limite desse exemplo: o que acontece quando o cliente e o servidor são iniciados; o que acontece quando o cliente termina normalmente; o que acontece para o cliente se o processo servidor terminar antes de o cliente concluir; o que acontece para o cliente se o host servidor parar; e assim por diante. Examinando todos esses cenários e entendendo o que acontece no nível de rede, e como isso aparece na API de soquetes, entenderemos mais sobre o que ocorre nesses níveis e como codificar nossas aplicações para tratar esses cenários.

Em todos esses exemplos, codificamos diretamente as constantes específicas dos endereços e portas de protocolos. Há duas razões para isso. A primeira delas é que devemos entender exatamente o que precisa ser armazenado nas estruturas de endereços específicas de protocolos. A segunda é que ainda não abordamos as funções de biblioteca que podem tornar esse processo mais portável. Essas funções serão discutidas no Capítulo 11.

Agora, observaremos as várias alterações que faremos tanto no cliente como no servidor nos capítulos sucessivos à medida que aprendermos mais sobre a programação de rede (Figuras 1.12 e 1.13).

5.2 Servidor de eco TCP: função `main`

Nossos cliente e servidor TCP seguem o fluxo de funções que diagramamos na Figura 4.1. Mostramos o programa do servidor concorrente na Figura 5.2.

Criação e vinculação de um soquete a uma porta bem-conhecida do servidor

9-15 Um soquete TCP é criado. Uma estrutura de endereço de soquete de Internet é preenchida com o endereço curinga (`INADDR_ANY`) e a porta bem-conhecida do servidor (`SERV_PORT`, definida como 9877 no nosso cabeçalho `unp.h`). Vincular o endereço curinga informa ao sistema que aceitaremos uma conexão destinada a qualquer interface local, caso o sistema seja multihomed. Nossa escolha do número da porta TCP tem por base a Figura 2.10. Ela deve ser maior que 1023 (não precisamos de uma porta reservada), maior que 5000 (para evitar conflito com as portas efêmeras alocadas por muitas implementações derivadas do Berkeley), menor que 49152 (para evitar conflito com o intervalo "correto" de portas efêmeras) e não deve conflitar com nenhuma porta registrada. O soquete é convertido em um soquete ouvinte por `listen`.

Esperando a conexão cliente se completar

17-18 O servidor bloqueia na chamada a `accept`, esperando que uma conexão de cliente complete.

Servidor concorrente

19-24 Para cada cliente, `fork` gera um filho e o filho trata o novo cliente. Como discutimos na Seção 4.8, o filho fecha o soquete ouvinte, e o pai fecha o soquete conectado. O filho chama então `str_echo` (Figura 5.3) para tratar o cliente.

tcpcliserv/tcpserv01.c

```
1 #include   "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int    listenfd, connfd;
6     pid_t  childpid;
7     socklen_t clilen;
```

Figura 5.2 O servidor de eco TCP (aprimorado na Figura 5.12) (*continua*).

```
 8     struct sockaddr_in cliaddr, servaddr;

 9     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

10     bzero(&servaddr, sizeof(servaddr));
11     servaddr.sin_family = AF_INET;
12     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
13     servaddr.sin_port = htons(SERV_PORT);

14     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

15     Listen(listenfd, LISTENQ);

16     for ( ; ; ) {
17         clilen = sizeof(cliaddr);
18         connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

19         if ( (childpid = Fork()) == 0) { /* processo-filho */
20            Close(listenfd);      /* fecha o soquete ouvinte */
21            str_echo(connfd);     /* processa a solicitação */
22            exit(0);
23         }
24         Close(connfd);           /* o pai fecha soquete conectado */
25     }
26 }
```

tcpcliserv/tcpserv01.c

Figura 5.2 O servidor de eco TCP (aprimorado na Figura 5.12) (*continuação*).

5.3 Servidor de eco TCP: função `str_echo`

A função `str_echo`, mostrada na Figura 5.3, realiza o processamento de servidor para cada cliente: ela lê e ecoa os dados do cliente de volta ao cliente.

Leitura de um buffer e eco do buffer

8-9 `read` lê dados a partir do soquete e a linha é ecoada de volta ao cliente por `writen`. Se o cliente fechar a conexão (o cenário normal), a recepção do FIN do cliente faz com que `read` do filho retorne 0. Isso faz a função `str_echo` retornar, o que termina o filho na Figura 5.2.

5.4 Cliente de eco TCP: função `main`

A Figura 5.4 mostra a função `main` do cliente TCP.

lib/str_echo.c

```
 1 #include   "unp.h"

 2 void
 3 str_echo(int sockfd)
 4 {
 5     ssize_t n;
 6     char  buf[MAXLINE];

 7   again:
 8     while ( (n = read(sockfd, buf, MAXLINE)) > 0)
 9       Writen(sockfd, buf, n);

10     if (n < 0 && errno == EINTR)
11       goto again;
12     else if (n < 0)
13       err_sys("str_echo: read error");
14 }
```

lib/str_echo.c

Figura 5.3 Função `str_echo`: ecoa os dados em um soquete.

tcpcliserv/tcpcli01.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    sockfd;
 6     struct sockaddr_in servaddr;
 7     if (argc != 2)
 8         err_quit("usage: tcpcli <IPaddress>");
 9     sockfd = Socket(AF_INET, SOCK_STREAM, 0);
10     bzero(&servaddr, sizeof(servaddr));
11     servaddr.sin_family = AF_INET;
12     servaddr.sin_port = htons(SERV_PORT);
13     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
14     Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
15     str_cli(stdin, sockfd);      /* faz tudo */
16     exit(0);
17 }
```

tcpcliserv/tcpcli01.c

Figura 5.4 Cliente de eco de TCP.

Criação do soquete, preenchimento da estrutura de endereço de soquete Internet

9-13 Um soquete TCP é criado e uma estrutura de endereço de soquete Internet é preenchida com o endereço e o número da porta IP do servidor. Recebemos o endereço IP do servidor a partir do argumento da linha de comando e a porta bem-conhecida do servidor (`SERV_PORT`) é proveniente do nosso cabeçalho `unp.h`.

Conexão a um servidor

14-15 `connect` estabelece a conexão com o servidor. A função `str_cli` (Figura 5.5) trata o restante do processamento de cliente.

5.5 Cliente de eco TCP: função `str_cli`

Essa função, mostrada na Figura 5.5, trata o loop do processamento de cliente: ela lê uma linha de texto a partir da entrada-padrão, grava-a no servidor, lê novamente o eco do servidor da linha e gera a saída da linha ecoada para a saída-padrão.

lib/str_cli.c

```
 1 #include   "unp.h"
 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     char   sendline[MAXLINE], recvline[MAXLINE];
 6     while (Fgets(sendline, MAXLINE, fp) != NULL) {
 7         Writen(sockfd, sendline, strlen(sendline));
 8         if (Readline(sockfd, recvline, MAXLINE) == 0)
 9            err_quit("str_cli: server terminated prematurely");
10         Fputs(recvline, stdout);
11     }
12 }
```

lib/str_cli.c

Figura 5.5 Função `str_cli`: loop do processamento de cliente.

Leitura de uma linha, gravação no servidor

6-7 `fgets` lê uma linha de texto e `writen` a envia ao servidor.

Leitura da linha ecoada a partir do servidor, gravação na saída-padrão

8-10 `readline` lê novamente a linha ecoada a partir do servidor e `fputs` a grava na saída-padrão.

Retorno para `main`

11-12 O loop termina quando `fgets` retorna um ponteiro nulo, o que ocorre quando encontra um fim de arquivo (end-of-file – EOF) ou um erro. Nossa função empacotadora `Fgets` verifica e aborta se ocorrer um erro, assim `Fgets` retorna um ponteiro nulo somente quando um fim de arquivo é encontrado.

5.6 Inicialização normal

Embora nosso exemplo TCP seja pequeno (cerca de 150 linhas de código para as duas funções `main`, `str_echo`, `str_cli`, `readline` e `writen`), é essencial entender como o cliente e o servidor são inicializados, terminados e, sobretudo, o que acontece quando algo sai errado: o host cliente pára, o processo do cliente pára, a conectividade de rede é perdida e assim por diante. Somente entendendo essas condições-limite, e suas interações com os protocolos TCP/IP, podemos escrever clientes e servidores robustos que podem tratar essas condições.

Primeiro, iniciamos o servidor em segundo plano no host `linux`.

```
linux % tcpserv01 &
[1] 17870
```

Quando o servidor é iniciado, ele chama `socket`, `bind`, `listen` e `accept`, bloqueando na chamada a `accept`. (Ainda não iniciamos o cliente.) Antes de iniciar o cliente, executamos o programa `netstat` para verificar o estado do soquete ouvinte do servidor.

```
linux % netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address         Foreign Address     State
tcp        0      0 *:9877                *:*                 LISTEN
```

Aqui mostramos somente a primeira linha de saída (o cabeçalho), mais a linha em que estamos interessados. Esse comando mostra o *status* de todos os soquetes no sistema, o que pode ser uma grande quantidade de saída. Devemos especificar o flag `-a` para verificar os soquetes ouvintes.

A saída é o que esperamos. Um soquete está no estado LISTEN com um curinga para o endereço IP local e uma porta local de 9877. `netstat` imprime um asterisco para um endereço IP de 0 (`INADDR_ANY`, o curinga) ou para uma porta de 0.

Em seguida, iniciamos o cliente no mesmo host, especificando o endereço IP do servidor como 127.0.0.1 (o endereço de loopback). Também poderíamos ter especificado o endereço IP normal do servidor (não-loopback).

```
linux % tcpcli01 127.0.0.1
```

O cliente chama `socket` e `connect`, esta última faz com que o handshake de três vias do TCP ocorra. Quando o handshake de três vias se completa, `connect` retorna no cliente e `accept` retorna no servidor. A conexão é estabelecida. Em seguida, ocorrem os passos a seguir:

1. O cliente chama `str_cli`, que será bloqueada na chamada a `fgets`, pois ainda não digitamos uma linha de entrada.

2. Quando `accept` retorna no servidor, ela chama `fork` e o filho chama `str_echo`. Essa função chama `readline`, que chama `read`, a qual é bloqueada enquanto espera que uma linha seja enviada a partir do cliente.
3. O servidor-pai, por outro lado, chama `accept` novamente e bloqueia enquanto espera a próxima conexão de cliente.

Temos três processos e todos estão em repouso (bloqueados): cliente, servidor-pai e servidor-filho.

> Quando o handshake de três vias se completa, listamos propositadamente primeiro o passo do cliente e então os passos do servidor. A razão disso pode ser vista na Figura 2.5: `connect` retorna quando o segundo segmento do handshake é recebido pelo cliente, mas `accept` não retorna até que o terceiro segmento do handshake seja recebido pelo servidor, metade do RTT depois que `connect` retorna.

Executamos propositadamente o cliente e o servidor no mesmo host porque essa é a maneira mais fácil de testar aplicações cliente/servidor. Como estamos executando o cliente e o servidor no mesmo host, `netstat` agora mostra duas linhas adicionais de saída, correspondentes à conexão TCP:

```
linux % netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address          Foreign Address      State
tcp        0      0 localhost:9877        localhost:42758      ESTABLISHED
tcp        0      0 localhost:42758       localhost:9877       ESTABLISHED
tcp        0      0 *:9877                *:*                  LISTEN
```

A primeira das linhas ESTABLISHED corresponde ao soquete do servidor-filho, uma vez que a porta local é 9877. A segunda das linhas ESTABLISHED é o soquete do cliente, uma vez que a porta local é 42758. Se estivéssemos executando o cliente e o servidor em hosts diferentes, o host cliente exibiria somente o soquete do cliente e o host servidor exibiria somente os dois soquetes do servidor.

Também podemos utilizar o comando `ps` para verificar o *status* e o relacionamento desses processos.

```
linux % ps -t pts/6 -o pid,ppid,tty,stat,args,wchan
  PID PPID TT       STAT COMMAND             WCHAN
22038 22036 pts/6    S    -bash               wait4
17870 22038 pts/6    S    ./tcpserv01         wait_for_connect
19315 17870 pts/6    S    ./tcpserv01         tcp_data_wait
19314 22038 pts/6    S    ./tcpcli01 127.0    read_chan
```

(Utilizamos argumentos bem específicos para `ps` somente para mostrar as informações relativas a essa discussão.) Nessa saída, executamos o cliente e o servidor a partir da mesma janela (`pts/6`, o que significa pseudoterminal de número 6). As colunas PID e PPID mostram os relacionamentos entre pai e filho. Podemos dizer que a primeira linha, `tcpserv01`, é o pai e a segunda linha, `tcpserv01`, é o filho, uma vez que o PPID do filho é o PID do pai. Além disso, o PPID do pai é o shell (`bash`).

A coluna STAT para todos os nossos três processos de rede é "S" (de "sleeping", dormindo), o que significa que o processo está dormindo (esperando algo). Quando um processo está em repouso, a coluna WCHAN especifica a condição. O Linux imprime `wait_for_connect` quando um processo é bloqueado em `accept`; ou `connect`, `tcp_data_wait` quando um processo é bloqueado no soquete de entrada ou de saída; ou `read_chan` quando um processo é bloqueado na E/S de terminal. Portanto, os valores de WCHAN para os nossos três processos de rede fazem sentido.

5.7 Término normal

Nesse ponto, a conexão é estabelecida e qualquer coisa que digitamos para o cliente é ecoado de volta.

```
linux % tcpcli01 127.0.0.1          mostramos essa linha anteriormente
hello, world                        agora digitamos isso
hello, world                        e a linha é ecoada
good bye
good bye
^D                                  Control-D é o nosso caractere terminal de EOF
```

Digitamos duas linhas, cada uma é ecoada e então digitamos nosso caractere terminal de EOF (Control-D), que termina o cliente. Se executarmos `netstat`, imediatamente teremos

```
linux % netstat -a | grep 9877
tcp        0      0 *:9877                  *:*                     LISTEN
top        0      0 localhost:42758         localhost:9877          TIME_WAIT
```

O lado do cliente da conexão (uma vez que a porta local é 42758) entra no estado TIME_WAIT (Seção 2.7) e o servidor ouvinte continua a esperar uma outra conexão de cliente. (Desta vez, redirecionamos a saída de `netstat` para `grep`, imprimindo somente as linhas com a nossa porta bem-conhecida do servidor. Fazer isso também remove a linha de cabeçalho.)

Podemos seguir os passos envolvidos no término normal do nosso cliente e servidor:

1. Quando digitamos nosso caractere de EOF, `fgets` retorna um ponteiro nulo e a função `str_cli` (Figura 5.5) retorna.
2. Quando `str_cli` retorna para a função `main` do cliente (Figura 5.4), ela termina chamando `exit`.
3. Parte do término de processo é o fechamento de todos os descritores abertos, para que o soquete cliente seja fechado pelo kernel. Isso envia um FIN ao servidor, ao qual o TCP servidor responde com um ACK. Essa é a primeira metade da seqüência de término da conexão TCP. Nesse ponto, o soquete do servidor está no estado CLOSE_WAIT e o soquete do cliente está no estado FIN_WAIT_2 (Figuras 2.4 e 2.5).
4. Quando o TCP servidor recebe o FIN, o servidor-filho é bloqueado em uma chamada a `readline` (Figura 5.3) e `readline` então retorna 0. Isso faz com que a função `str_echo` retorne para `main` do servidor-filho.
5. O servidor-filho termina chamando `exit` (Figura 5.2).
6. Todos os descritores abertos no servidor-filho são fechados. O fechamento do soquete conectado pelo filho faz com que ocorram os dois segmentos finais do término da conexão TCP: um FIN do servidor para o cliente e um ACK do cliente (Figura 2.5). Nesse ponto, a conexão é completamente terminada. O soquete do cliente entra no estado TIME_WAIT.
7. Por fim, o sinal `SIGCHLD` é enviado ao pai quando o servidor-filho termina. Isso ocorre nesse exemplo, mas não capturamos o sinal no nosso código e a ação default do sinal deve ser ignorada. Portanto, o filho entra no estado zumbi. Podemos verificar isso com o comando `ps`.

```
linux % ps -t pts/6 -o pid,ppid,tty,stat,args,wchan
  PID  PPID TT       STAT COMMAND             WCHAN
22038 22036 pts/6    S    -bash               read_chan
17870 22038 pts/6    S    ./tcpserv01         wait_for_connect
19315 17870 pts/6    Z    [tcpserv01 <defu    do_exit
```

O STAT do filho agora é `z` (para zumbi).

Precisamos limpar nossos processos zumbis, o que requer lidar com sinais do Unix. Na próxima seção, teremos uma visão geral sobre o tratamento de sinais.

5.8 Tratamento de sinal POSIX

Um *sinal* é uma notificação a um processo de que um evento ocorreu. Às vezes, os sinais são chamados *interrupções de software*. Os sinais normalmente ocorrem de maneira *assíncrona*. Isso significa que um processo não sabe antecipadamente e exatamente quando um sinal ocorrerá.

Sinais podem ser enviados:

- De um processo para outro (ou para o próprio processo)
- Do kernel para um processo

O sinal `SIGCHLD` que descrevemos no final da seção anterior é um sinal enviado pelo kernel, sempre que um processo termina, ao pai do processo que termina.

Cada sinal tem uma *disposição*, que também é chamada *ação* associada ao sinal. Configuramos a disposição de um sinal chamando a função `sigaction` (descrita em breve) e temos três opções para a disposição:

1. Podemos fornecer uma função que é chamada sempre que um sinal específico ocorre. Essa função é chamada *handler de sinal* e a ação é chamada *capturar* um sinal. Os dois sinais `SIGKILL` e `SIGSTOP` não podem ser capturados. Nossa função é chamada com um único argumento do tipo inteiro, que é o número do sinal, e a função não retorna nada. Seu protótipo de função é, portanto:

   ```
   void handler(int signo);
   ```

   Para a maioria dos sinais, chamar `sigaction` e especificar uma função a ser chamada quando o sinal ocorre é tudo o que é requerido para capturar um sinal. Mas veremos mais adiante que alguns sinais, `SIGIO`, `SIGPOLL` e `SIGURG`, requerem ações adicionais por parte do processo para capturar o sinal.
2. Podemos *ignorar* um sinal configurando sua disposição como `SIG_IGN`. Os dois sinais `SIGKILL` e `SIGSTOP` não podem ser ignorados.
3. Podemos configurar a *disposição-padrão* para um sinal configurando sua disposição como `SIG_DFL`. Normalmente, o padrão é terminar um processo ao recebimento de um sinal, com certos sinais também gerando uma imagem de memória do processo no seu diretório de trabalho atual. Há alguns sinais cuja disposição default é ser ignorada: `SIGCHLD` e `SIGURG` (enviados na chegada de dados fora da banda, Capítulo 24) são dois que encontraremos neste livro.

Função `signal`

A maneira como POSIX estabelece a disposição de um sinal é chamar a função `sigaction`. Isso, porém, torna-se complicado, uma vez que um dos argumentos para a função é uma estrutura que devemos alocar e preencher. Uma maneira mais fácil de configurar a disposição de um sinal é chamar a função `signal`. O primeiro argumento é o nome do sinal e o segundo é um ponteiro para uma função ou uma das constantes `SIG_IGN` ou `SIG_DFL`. Mas `signal` é uma função histórica anterior ao POSIX. Diferentes implementações fornecem diferentes semânticas de sinal quando ela é chamada, fornecendo retrocompatibilidade, ao passo que o POSIX descreve explicitamente a semântica quando `sigaction` é chamada. A solução é definir nossa própria função identificada como `signal` que chame apenas a função POSIX `sigaction`. Isso fornece uma interface simples com a semântica POSIX desejada. Incluímos essa função na nossa própria biblioteca, por exemplo, junto com as nossas funções `err_XXX` e as nossas funções empacotadoras, que especificamos ao construir qualquer um dos nossos programas neste livro. Essa função é mostrada na Figura 5.6 (a função empacotadora correspondente, `Signal`, não é mostrada aqui, uma vez que ela seria a mesma quer chamasse nossa função ou uma função `signal` distribuída pelo fornecedor).

lib/signal.c

```
 1 #include   "unp.h"
 2 Sigfunc *
 3 signal(int signo, Sigfunc *func)
 4 {
 5     struct sigaction act, oact;
 6     act.sa_handler = func;
 7     sigemptyset(&act.sa_mask);
 8     act.sa_flags = 0;
 9     if (signo == SIGALRM) {
10 #ifdef SA_INTERRUPT
11         act.sa_flags |= SA_INTERRUPT;        /* SunOS 4.x */
12 #endif
13     } else {
14 #ifdef SA_RESTART
15         act.sa_flags |= SA_RESTART; /* SVR4, 4.4BSD */
16 #endif
17     }
18     if (sigaction(signo, &act, &oact) < 0)
19         return(SIG_ERR);
20     return(oact.sa_handler);
21 }
```

lib/signal.c

Figura 5.6 A função `signal` que chama a função POSIX `sigaction`.

Simplificação do protótipo de função utilizando `typedef`

2-3 O protótipo normal da função para `signal` é complicado pelo nível de parênteses aninhados.

```
void (*signal(int signo, void (*func)(int)))(int);
```

Para simplificar isso, definimos o tipo `Sigfunc` no nosso cabeçalho `unp.h` como

```
typedef void Sigfunc(int);
```

declarando que handlers de sinais são funções com um argumento de inteiro e a função retorna nada (`void`). O protótipo de função torna-se então

```
Sigfunc *signal(int signo, Sigfunc *func);
```

Um ponteiro para um sinal que trata a função é o segundo argumento da função, bem como o valor de retorno da função.

Configuração do handler

6 O membro `sa_handler` da estrutura `sigaction` é configurado com o argumento de *func*.

Configuração da máscara de sinal para o handler

7 O POSIX permite especificar um conjunto de sinais que será *bloqueado* quando nosso handler de sinal é chamado. Qualquer sinal que não é bloqueado pode ser *entregue* para um processo. Configuramos o membro `sa_mask` como o conjunto vazio, o que significa que nenhum sinal adicional será bloqueado enquanto nosso handler de sinal estiver em execução. O POSIX garante que o sinal sendo capturado sempre é bloqueado enquanto seu handler estiver em execução.

Configuração do flag `SA_RESTART`

8-17 `SA_RESTART` é um flag opcional. Quando o flag é ligado, uma chamada de sistema interrompida por esse sinal será automaticamente reiniciada pelo kernel. (Discutiremos em mais detalhes as chamadas de sistema interrompidas na próxima seção, quando continuarmos com o nosso exemplo.) Se o sinal sendo capturado não for `SIGALRM`, especificaremos o flag

SA_RESTART, se definido. (A razão especial para mencionar o SIGALRM é que o propósito de gerar esse sinal normalmente é impor um tempo-limite sobre uma operação de E/S, como mostraremos na Seção 14.2; nesse caso, queremos que a chamada de sistema bloqueada seja interrompida pelo sinal.) Alguns sistemas mais antigos, notavelmente o SunOS 4.x, reiniciam automaticamente uma chamada de sistema interrompida por default e então definem o complemento desse flag como SA_INTERRUPT. Se esse flag estiver definido, nós o ligamos se o sinal sendo capturado for SIGALRM.

Chamada de sigaction

18-20 Chamamos sigaction e então retornamos a ação antiga ao sinal como o valor de retorno da função signal.

Por todo o livro, utilizaremos a função signal da Figura 5.6.

Semântica de sinal POSIX

Resumimos os seguintes pontos sobre tratamento de sinal em um sistema compatível com POSIX:

- Depois que um handler de sinal é instalado, ele permanece instalado. (Sistemas mais antigos removiam o handler de sinal todas as vezes que ele era executado.)
- Enquanto um handler de sinal estiver em execução, o sinal sendo entregue é bloqueado. Além disso, quaisquer sinais adicionais especificados no conjunto de sinais sa_mask passado para sigaction quando o handler é instalado também serão bloqueados. Na Figura 5.6, configuramos sa_mask como o conjunto vazio, o que significa que nenhum sinal adicional é bloqueado além do sinal sendo capturado.
- Se um sinal for gerado uma ou mais vezes enquanto está bloqueado, ele normalmente é entregue somente uma vez depois que o sinal é desbloqueado. Isto é, por default, os sinais do Unix não são *enfileirados*. Veremos um exemplo disso na próxima seção. O padrão de tempo real do POSIX, 1003.1b, define alguns sinais confiáveis que são enfileirados, mas nós não os utilizamos neste livro.
- É possível bloquear e desbloquear seletivamente um conjunto de sinais utilizando a função sigprocmask. Isso permite proteger uma região crítica do código evitando que certos sinais sejam capturados enquanto essa região estiver em execução.

5.9 Tratamento de sinais SIGCHLD

O propósito do estado zumbi é manter as informações sobre o filho para que o pai busque-as posteriormente. Essas informações incluem o ID do processo-filho, seu *status* de término e as informações sobre a utilização de recursos pelo filho (tempo de CPU, memória, etc.). Se um processo terminar, e se tiver filhos no estado zumbi, o ID do processo-pai em todos os filhos no estado zumbi é configurado como 1 (o processo init), o qual herdará e limpará os processos-filhos (isto é, init irá esperá-los – wait–, o que remove os zumbis). Alguns sistemas Unix mostram a coluna COMMAND para um processo zumbi como <defunct>.

Tratando zumbis

Obviamente, não queremos deixar zumbis por aí. Eles ocupam espaço no kernel e, em conseqüência disso, podemos ficar sem processos. Sempre que chamamos fork para gerar filhos, devemos chamar wait para impedir que eles se tornem zumbis. Para fazer isso, estabelecemos um handler de sinal para capturar SIGCHLD e, dentro do handler, chamamos wait. (Descreveremos as funções wait e waitpid na Seção 5.10.) Estabelecemos o handler de sinal adicionando a chamada de função

```
Signal (SIGCHLD, sig_chld);
```

na Figura 5.2, depois da chamada a `listen`. (Isso deve ser feito antes de chamarmos `fork` para gerar o primeiro filho, e precisa ser feito somente uma vez.) Em seguida, definimos o handler de sinal, a função `sig_chld`, que mostramos na Figura 5.7.

tcpcliserv/sigchldwait.c

```
 1 #include   "unp.h"
 2 void
 3 sig_chld(int signo)
 4 {
 5     pid_t  pid;
 6     int       stat;
 7     pid = wait(&stat);
 8     printf("child %d terminated\n", pid);
 9     return;
10 }
```

tcpcliserv/sigchldwait.c

Figura 5.7 A versão do handler de sinal `SIGCHLD` que chama `wait` (aprimorada na Figura 5.11).

> *Aviso:* Chamar funções E/S-padrão, como `printf`, em um handler de sinal não é recomendável, pelas razões que discutiremos na Seção 11.18. Chamamos `printf` aqui como uma ferramenta de diagnóstico para verificar quando o filho termina.
>
> No System V e no Unix 98, o filho de um processo não se torna um zumbi se o processo configurar a disposição de `SIGCHLD` como `SIG_IGN`. Infelizmente, isso funciona somente no System V e no Unix 98. POSIX declara explicitamente que esse comportamento não é especificado. A maneira portável de tratar zumbis é capturar `SIGCHLD` e chamar `wait` ou `waitpid`.

Se compilarmos esse programa – a Figura 5.2 com a chamada a `Signal`, com o nosso handler `sig_chld` – do Solaris 9 e utilizarmos a função `signal` da biblioteca de sistema (não a nossa versão da Figura 5.6), teremos o seguinte:

```
solaris % tcpserv02 &                          inicia o servidor em segundo plano
[2]     16939
solaris % tcpcli01 127.0.0.1                   então inicia o cliente no primeiro plano
hi there                                       digitamos isso
hi there                                       e isso é ecoado
^D                                             digitamos nosso caractere de EOF
child 16942 terminated                         saída gerada por printf no handler de sinal
accept error: Interrupted system call          a função main aborta
```

A seqüência de passos é a seguinte:

1. Terminamos o cliente digitando nosso caractere de EOF. O TCP cliente envia um FIN ao servidor, que responde com um ACK.
2. A recepção de FIN entrega um EOF à `readline` pendente do filho. O filho termina.
3. O pai é bloqueado na sua chamada a `accept` quando o sinal `SIGCHLD` é entregue. A função `sig_chld` é executada (nosso handler de sinal), `wait` busca o PID do filho e o *status* de término e `printf` é chamado a partir do handler de sinal. O handler de sinal retorna.
4. Como o sinal foi capturado pelo pai enquanto este estava bloqueado em uma chamada de sistema lenta (`accept`), o kernel faz com que `accept` retorne um erro de `EINTR` (chamada de sistema interrompida). O pai não trata esse erro (Figura 5.2), portanto ele aborta.

O propósito desse exemplo é mostrar que, ao escrever programas de rede que capturam sinais, devemos estar cientes e tratar as chamadas de sistema interrompidas. Nesse exemplo

específico, em execução no Solaris 9, a função `signal` fornecida na biblioteca C-padrão não faz com que uma chamada de sistema interrompida seja automaticamente reiniciada pelo kernel. Isto é, o flag `SA_RESTART` que configuramos na Figura 5.6 não é configurado pela função `signal` na biblioteca de sistema. Alguns sistemas reiniciam automaticamente a chamada de sistema interrompida. Se executarmos o mesmo exemplo no 4.4BSD, utilizando a versão da função `signal` da biblioteca deste, o kernel reiniciará a chamada de sistema interrompida e `accept` não retornará um erro. Tratar esse problema potencial entre diferentes sistemas operacionais é uma das razões pelas quais definimos nossa própria versão da função `signal` que utilizamos por todo este livro (Figura 5.6).

Como parte das convenções de codificação utilizadas neste texto, sempre codificamos um `return` explícito nos nossos handlers de sinal (Figura 5.7), mesmo que alcançar o final da função tenha o mesmo efeito para uma função que retorna `void`. Ao ler o código, a instrução de retorno desnecessária atua como um lembrete de que o retorno pode interromper uma chamada de sistema.

Tratando chamadas de sistema interrompidas

Utilizamos o termo "chamada de sistema lenta" para descrever `accept` e também para qualquer chamada de sistema que possa ser bloqueada eternamente. Isto é, a chamada de sistema nunca precisa retornar. A maioria das funções de rede entra nessa categoria. Por exemplo, não há garantias de que uma chamada do servidor para `accept` irá retornar, se não houver algum cliente que irá se conectar ao servidor. De maneira semelhante, a chamada do nosso servidor a `read` na Figura 5.3 nunca retornará se o cliente nunca enviar uma linha para o servidor ecoar. Outros exemplos de chamadas de sistema lentas são leituras e gravações de pipes e dispositivos de terminal. Uma exceção notável é a E/S de disco, que normalmente retorna ao chamador (supondo que não há nenhuma falha catastrófica de hardware).

A regra básica que se aplica aqui é que, quando um processo é bloqueado em uma chamada de sistema lenta *e* o processo captura um sinal *e* o handler de sinal retorna, a chamada de sistema *pode* retornar um erro de `EINTR`. *Alguns* kernels reiniciam automaticamente *algumas* chamadas de sistema interrompidas. Para portabilidade, quando escrevemos um programa que captura sinais (a maioria dos servidores concorrentes captura `SIGCHLD`), devemos estar preparados para o fato de que chamadas de sistema lentas retornam `EINTR`. Problemas de portabilidade são causados pelos qualificadores "pode" e "alguns" que eram utilizados anteriormente e pelo fato de que o suporte para o flag `POSIX_SA_RESTART` é opcional. Mesmo se uma implementação suportar o flag `SA_RESTART`, nem todas as chamadas de sistema interrompidas podem ser automaticamente reiniciadas. Por exemplo, a maioria das implementações derivadas do Berkeley nunca reinicia `select` automaticamente e algumas dessas implementações nunca reiniciam `accept` ou `recvfrom`.

Para tratar uma `accept` interrompida, alteramos a chamada para `accept` na Figura 5.2, o começo do loop `for`, para o seguinte:

```
for ( ; ; ) {
    clilen = sizeof(cliaddr);
    if ( (connfd = accept(listenfd, (SA *) &cliaddr, &clilen)) < 0) {
      if (errno == EINTR)
         continue;        /* de volta a for() */
      else
         err_sys("accept error");
   }
```

Observe que chamamos `accept` e não nossa função empacotadora `Accept`, uma vez que nós mesmos devemos tratar a falha da função.

O que estamos fazendo nesse fragmento de código é reiniciar a chamada de sistema interrompida. Isso é bom para `accept`, junto com funções como `read`, `write`, `select` e `open`.

Mas há uma função que não podemos reiniciar: `connect`. Se essa função retornar `EINTR`, não podemos chamá-la novamente, pois fazer isso retornará um erro imediato. Quando `connect` é interrompida por um sinal capturado e não é automaticamente reiniciada, devemos chamar `select` para esperar que a conexão complete, como descreveremos na Seção 16.3.

5.10 Funções `wait` e `waitpid`

Na Figura 5.7, chamamos a função `wait` para tratar o filho terminado.

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

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

As duas retornam: ID de processo se OK, 0 ou –1 em erro

`wait` e `waitpid` retornam ambas dois valores: o valor de retorno da função é o ID do processo-filho terminado; e o *status* de término do filho (um inteiro) é retornado pelo ponteiro *statloc*. Há três macros que podemos chamar que examinam o *status* de terminação, e nos informam se o filho foi terminado de maneira normal, eliminado por um sinal ou apenas parado pelo controle de trabalho. Macros adicionais permitem então buscar o *status* de saída do filho, ou o valor do sinal que o eliminou, ou o valor de sinal do controle de job que o parou. Utilizaremos as macros `WIFEXITED` e `WEXITSTATUS` na Figura 15.10 para esse propósito.

Se não houver nenhum filho terminado para o processo que chama `wait`, mas o processo tem um ou mais filhos que ainda estão em execução, então `wait` bloqueia até que um dos primeiros filhos existentes termine.

`waitpid` fornece mais controle sobre qual processo esperar e se devemos ou não bloqueá-lo. Primeiro, o argumento *pid* permite especificar o ID de processo que queremos esperar. Um valor de –1 diz para esperar que o primeiro dos nossos filhos termine. (Há outras opções, que lidam com IDs de grupos de processos, mas não precisamos delas neste livro.) Os argumentos *opções* permitem especificar opções adicionais. A opção mais comum é `WNOHANG`. Essa opção informa o kernel a não bloquear se não houver nenhum filho terminado.

Diferença entre `wait` e `waitpid`

Agora ilustraremos a diferença entre as funções `wait` e `waitpid` quando utilizadas para limpar filhos terminados. Para fazer isso, modificamos nosso cliente TCP como mostrado na Figura 5.9. O cliente estabelece cinco conexões com o servidor e então utiliza somente a primeira (*`sockfd [0]`*) na chamada a `str_cli`. O propósito de estabelecer múltiplas conexões é gerar múltiplos filhos a partir do servidor concorrente, como mostrado na Figura 5.8.

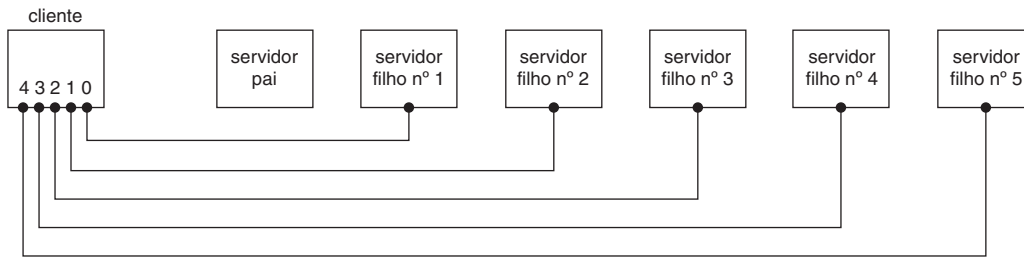


Figura 5.8 O cliente com cinco conexões estabelecidas ao mesmo servidor concorrente.

tcpcliserv/tcpcli04.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    i, sockfd[5];
 6     struct sockaddr_in servaddr;
 7     if (argc != 2)
 8         err_quit("usage: tcpcli <IPaddress>");
 9     for (i = 0; i < 5; i++) {
10         sockfd[i] = Socket(AF_INET, SOCK_STREAM, 0);
11         bzero(&servaddr, sizeof(servaddr));
12         servaddr.sin_family = AF_INET;
13         servaddr.sin_port = htons(SERV_PORT);
14         Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
15         Connect(sockfd[i], (SA *) &servaddr, sizeof(servaddr));
16     }
17     str_cli(stdin, sockfd[0]);   /* faz tudo */
18     exit(0);
19 }
```

tcpcliserv/tcpcli04.c

Figura 5.9 O cliente TCP que estabelece cinco conexões com o servidor.

Quando o cliente termina, todos os descritores abertos são fechados automaticamente pelo kernel (não chamamos `close`, somente `exit`) e todas as cinco conexões são terminadas quase ao mesmo tempo. Isso faz com que cinco FINs sejam enviados, um em cada conexão, o que, por sua vez, faz com que todos os cinco filhos do servidor terminem quase ao mesmo tempo. Isso faz com que cinco sinais `SIGCHLD` sejam entregues para o pai quase ao mesmo tempo, o que mostramos na Figura 5.10.

É essa entrega de múltiplas ocorrências do mesmo sinal que causa o problema que veremos em seguida.

Primeiro, executamos o servidor no segundo plano e então o nosso novo cliente. Nosso servidor é a Figura 5.2, modificada para chamar `signal` para estabelecer a Figura 5.7 como um handler de sinal para `SIGCHLD`.

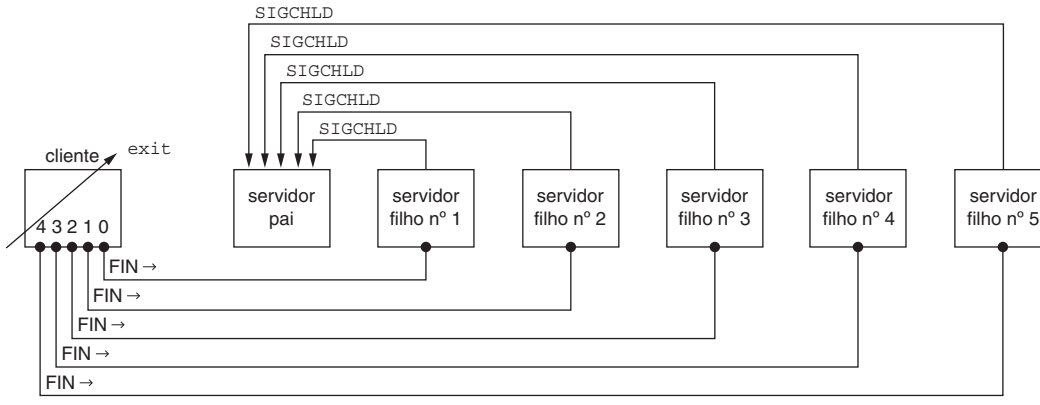


Figura 5.10 O cliente termina, fechando todas as cinco conexões, terminando todos os cinco filhos.

```
linux % tcpserv03 &
[1] 20419
linux % tcpcli04 127.0.0.1
hello                                   digitamos isso
hello                                   e isso é ecoado
^D                                      então digitamos nosso caractere de EOF
child 20426 terminated                  saída gerada pelo servidor
```

A primeira coisa que observamos é que é gerada a saída de somente um `printf`, quando esperávamos que todos os cinco filhos tivessem terminado. Se executarmos `ps`, veremos que os outros quatro filhos ainda existem como zumbis.

```
  PID TTY          TIME CMD
20419 pts/6    00:00:00 tcpserv03
20421 pts/6    00:00:00 tcpserv03 <defunct>
20422 pts/6    00:00:00 tcpserv03 <defunct>
20423 pts/6    00:00:00 tcpserv03 <defunct>
```

Estabelecer um handler de sinal e chamar `wait` a partir dele não é suficiente para impedir zumbis. O problema é que todos os cinco sinais são gerados antes de o handler de sinal ser executado e este é executado somente uma vez porque sinais Unix normalmente não são enfileirados. Além disso, esse problema é não-determinístico. No exemplo que acabamos de executar, com o cliente e o servidor no mesmo host, o handler de sinal é executado uma vez, deixando quatro zumbis. Mas, se executamos o cliente e o servidor em hosts diferentes, o handler de sinal normalmente é executado duas vezes: uma vez como resultado do primeiro sinal sendo gerado e, considerando que os outros quatro sinais ocorrem enquanto o handler de sinal está executando, o handler é chamado somente mais uma vez. Isso deixa três zumbis. Mas, às vezes, provavelmente dependendo do timing dos FINs chegando ao host servidor, o handler de sinal é executado três ou mesmo quatro vezes.

A solução correta é chamar `waitpid` em vez de `wait`. A Figura 5.11 mostra a versão de nossa função `sig_chld` que trata `SIGCHLD` corretamente. Essa versão funciona porque chamamos `waitpid` dentro de um loop, buscando o *status* de qualquer um de nossos filhos que terminou. Devemos especificar a opção `WNOHANG`: isso instrui `waitpid` a não bloquear se houver filhos executando que ainda não terminaram. Na Figura 5.7, não podemos chamar `wait` em um loop, porque não há nenhuma maneira de impedir que `wait` bloqueie se houver filhos executando que ainda não terminaram.

A Figura 5.12 mostra a versão final de nosso servidor. Ele trata corretamente um retorno de `EINTR` proveniente de `accept` e estabelece um handler de sinal (Figura 5.11) que chama `waitpid` para todos os filhos terminados.

tcpcliserv/sigchldwaitpid.c

```
 1 #include   "unp.h"
 2 void
 3 sig_chld(int signo)
 4 {
 5     pid_t pid;
 6     int       stat;
 7     while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
 8         printf("child %d terminated\n", pid);
 9     return;
10 }
```

tcpcliserv/sigchldwaitpid.c

Figura 5.11 A versão final (correta) da função `sig_chld` que chama `waitpid`.

tcpcliserv/tcpserv04.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    listenfd, connfd;
 6     pid_t  childpid;
 7     socklen_t clilen;
 8     struct sockaddr_in cliaddr, servaddr;
 9     void   sig_chld(int);
10     listenfd = Socket(AF_INET, SOCK_STREAM, 0);
11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14     servaddr.sin_port = htons(SERV_PORT);
15     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
16     Listen(listenfd, LISTENQ);
17     Signal(SIGCHLD, sig_chld);   /* deve chamar waitpid() */
18     for ( ; ; ) {
19         clilen = sizeof(cliaddr);
20         if ( (connfd = accept(listenfd, (SA *) &cliaddr, &clilen)) < 0) {
21             if (errno == EINTR)
22                 continue;          /* de volta a for() */
23             else
24                 err_sys("accept error");
25         }
26         if ( (childpid = Fork()) == 0) { /* processo-filho */
27             Close(listenfd);       /* fecha o soquete ouvinte */
28             str_echo(connfd);      /* processa a solicitação */
29             exit(0);
30         }
31         Close(connfd);             /* o pai fecha soquete conectado */
32     }
33 }
```

tcpcliserv/tcpserv04.c

Figura 5.12 A versão final (correta) do servidor de TCP que trata um erro de `EINTR` proveniente de `accept`.

O propósito desta seção foi demonstrar três cenários que podemos encontrar com programação de rede:

1. Devemos capturar o sinal `SIGCHLD` quando bifurcando (`fork`) processos-filhos.
2. Devemos tratar chamadas de sistema interrompidas quando capturamos sinais.
3. Um handler `SIGCHLD` deve ser codificado corretamente utilizando `waitpid` para impedir que qualquer zumbi seja deixado para trás.

A versão final de nosso servidor de TCP (Figura 5.12), juntamente com o handler `SIGCHLD` na Figura 5.11, trata todos esses três cenários.

5.11 Conexão abortada antes de `accept` retornar

Há outra condição semelhante ao exemplo da chamada de sistema interrompida da seção anterior que pode fazer com que `accept` retorne um erro não-fatal, caso em que devemos sim-

plesmente chamar `accept` novamente. A seqüência de pacotes mostrada na Figura 5.13 foi vista em servidores ocupados (servidores da Web tipicamente ocupados).

Aqui, o handshake de três vias é completado, a conexão é estabelecida e então o TCP cliente envia um RST (reinicialização). No lado do servidor, a conexão é enfileirada pelo seu TCP, esperando que o processo servidor chame `accept` quando o RST chega. Posteriormente, o processo servidor chama `accept`.

> Uma maneira fácil de simular esse cenário é iniciar o servidor, fazer com que ele chame `socket`, `bind` e `listen` e então pausar durante um curto período de tempo antes de chamar `accept`. Enquanto o processo servidor está em repouso, inicie o cliente e faça com que ele chame `socket` e `connect`. Logo que `connect` retorna, configure a opção de soquete `SO_LINGER` para gerar o RST (que descreveremos na Seção 7.5 e do qual mostraremos um exemplo na Figura 16.21) e termine.

Infelizmente, o que acontece com a conexão abortada depende da implementação. Implementações derivadas do Berkeley tratam a conexão abortada completamente dentro do kernel; o processo servidor nunca a vê. A maioria das implementações SVR4, porém, retorna um erro ao processo como retorno de `accept`; e o erro depende da implementação. Essas implementações SVR4 retornam um `errno` de `EPROTO` ("erro de protocolo"), mas o POSIX especifica que o retorno deve ser `ECONNABORTED` ("o software causou o abortamento da conexão"). A razão dessa alteração POSIX é que `EPROTO` também retorna quando alguns eventos fatais relacionados ao protocolo ocorrem nos subsistemas de fluxos. Retornar o mesmo erro para um abortamento não-fatal de uma conexão estabelecida pelo cliente torna impossível ao servidor saber se deve ou não chamar `accept` novamente. No caso do erro `ECONNABORTED`, o servidor pode ignorar o erro e simplesmente chamar `accept` novamente.

> Os passos envolvidos em kernels derivados do Berkeley que nunca passam esse erro para o processo podem ser seguidos no TCPv2. O RST é processado na página 964, fazendo com que `tcp_close` seja chamado. Essa função chama `in_pcbdetach` na página 897 que, por sua vez, chama `sofree` na página 719. `sofree` (página 473) descobre que o soquete sendo abortado continua na fila de conexões completadas do soquete ouvinte, remove o soquete da fila e o libera. Quando o servidor se envolve em uma chamada a `accept`, ele nunca saberá se uma conexão completada foi então removida da fila.

Retornaremos a essas conexões abortadas na Seção 16.6 e veremos como elas podem resultar em um problema quando combinadas com `select` e um soquete ouvinte no modo bloqueador normal.

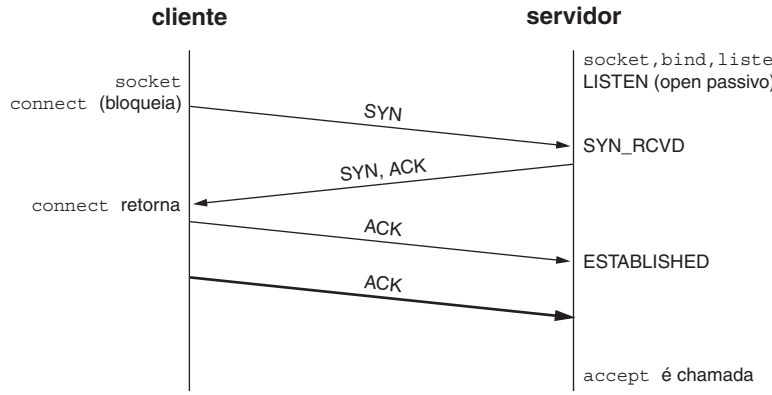


Figura 5.13 Recebendo um RST para uma conexão ESTABLISHED antes de `accept` ser chamada.

5.12 Término do processo servidor

Agora, iniciaremos nosso cliente/servidor e então eliminaremos o processo servidor-filho. Isso simula o travamento do processo servidor, de modo que possamos ver o que acontece ao cliente. (Devemos ter o cuidado de distinguir entre o travamento do processo servidor, que descreveremos em breve, e o travamento do host servidor, que descreveremos na Seção 5.14.) Ocorrem os seguintes passos:

1. Iniciamos o servidor e o cliente e digitamos uma linha para o cliente verificar se tudo está ok. Normalmente, essa linha é ecoada pelo servidor-filho.
2. Localizamos o ID do processo servidor-filho e o eliminamos. Como parte do término de processo, todos os descritores abertos no filho são fechados. Isso faz com que um FIN seja enviado ao cliente e o TCP cliente responde com um ACK. Essa é a primeira metade do término da conexão TCP.
3. O sinal `SIGCHLD` é enviado ao servidor-pai e tratado corretamente (Figura 5.12).
4. Nada acontece no cliente. O TCP cliente recebe o FIN a partir do TCP servidor e responde com um ACK, mas o problema é que o processo cliente é bloqueado na chamada a `fgets` esperando uma linha do terminal.
5. Executar `netstat` nesse ponto mostra o estado dos soquetes.

```
linux % netstat -a | grep 9877
tcp        0      0 *:9877                  *:*                     LISTEN
tcp        0      0 localhost:9877          localhost:43604         FIN_WAIT2
tcp        1      0 localhost:43604         localhost:9877          CLOSE_WAIT
```

Na Figura 2.4, vemos que ocorreu metade da seqüência de término da conexão TCP.

6. Ainda podemos digitar uma linha de entrada para o cliente. Eis o que acontece no cliente que inicia a partir do Passo 1:

```
linux % tcpcli01 127.0.0.1        inicia o cliente
hello                             a primeira linha que digitamos
hello                             é ecoada corretamente
                                  aqui eliminamos o servidor-filho no host servidor
another line                      então digitamos uma segunda linha para o cliente
str_cli: server terminated prematurely
```

   Quando digitamos "another line", `str_cli` chama `writen` e o TCP cliente envia os dados ao servidor. Isso é permitido pelo TCP porque a recepção do FIN pelo TCP cliente indica apenas que o processo servidor fechou a sua extremidade da conexão e não enviará outros dados. A recepção do FIN *não* informa ao TCP cliente que o processo servidor terminou (o que, nesse caso, ocorreu). Abordaremos isso novamente na Seção 6.6 ao discutirmos o *halfclose* ("meio fechamento") do TCP.

   Quando o TCP servidor recebe os dados do cliente, ele responde com um RST uma vez que o processo com esse soquete aberto foi terminado. Podemos verificar se o RST foi enviado observando os pacotes com `tcpdump`.

7. O processo cliente não verá o RST porque ele chama `readline` logo após a chamada a `writen` e `readline` retorna 0 (EOF) imediatamente por causa do FIN que foi recebido no Passo 2. Nosso cliente não espera receber um EOF nesse ponto (Figura 5.5), portanto, ele fecha com a mensagem de erro "server terminated prematurely" ("servidor terminado prematuramente").
8. Quando o cliente termina (chamando `err_quit` na Figura 5.5), todos os seus descritores abertos são fechados.

   O que descrevemos também depende do *timing* do exemplo. A chamada do cliente a `readline` pode acontecer antes ou depois que o RST do servidor é recebido pelo cliente. Se

> `readline` ocorrer antes de o RST ser recebido, como mostramos no nosso exemplo, o resultado é um EOF inesperado no cliente. Mas, se o RST chegar primeiro, o resultado é um retorno de erro ECONNRESET ("Conexão reinicializada pelo peer") de `readline`.

O problema nesse exemplo é que o cliente é bloqueado na chamada a `fgets` quando o FIN chega no soquete. O cliente está realmente trabalhando com dois descritores – o soquete e a entrada de usuário – e, em vez de bloquear na entrada de somente uma das duas origens (como `str_cli` está atualmente codificado), ele deve bloquear na entrada de qualquer das duas origens. De fato, esse é um dos propósitos das funções `select` e `poll`, que descreveremos no Capítulo 6. Quando recodificamos a função `str_cli` na Seção 6.4, depois que eliminamos o servidor-filho, o cliente é notificado sobre o FIN recebido.

5.13 Sinal `SIGPIPE`

O que acontece se o cliente ignorar o retorno de erro de `readline` e gravar mais dados no servidor? Isso pode acontecer, por exemplo, se o cliente precisar realizar duas gravações no servidor antes de ler qualquer coisa novamente, com a primeira gravação evocando o RST.

A regra que se aplica é: quando um processo grava em um soquete que recebeu um RST, o sinal `SIGPIPE` é enviado ao processo. A ação-padrão desse sinal é terminar o processo, assim o processo deve capturar o sinal para evitar ser terminado de maneira involuntária.

Se o processo capturar e retornar o sinal do handler de sinal, ou ignorar o sinal, a operação de gravação retorna `EPIPE`.

> Uma pergunta feita com freqüência (*frequently asked question* – FAQ) na Usenet é como obter esse sinal na primeira gravação e não na segunda. Isso não é possível. Seguindo nossa discussão anterior, a primeira gravação evoca o RST e a segunda evoca o sinal. É certo gravar em um soquete que recebeu um FIN, mas é um erro gravar em um soquete que recebeu um RST.

Para ver o que acontece com `SIGPIPE`, modificamos nosso cliente como mostrado na Figura 5.14.

tcpcliserv/str_cli11.c

```
 1 #include   "unp.h"

 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     char   sendline[MAXLINE], recvline[MAXLINE];

 6     while (Fgets(sendline, MAXLINE, fp) != NULL) {

 7         Writen(sockfd, sendline, 1);
 8         sleep(1);
 9         Writen(sockfd, sendline + 1, strlen(sendline) - 1);

10         if (Readline(sockfd, recvline, MAXLINE) == 0)
11            err_quit("str_cli: server terminated prematurely");

12         Fputs(recvline, stdout);
13     }
14 }
```

tcpcliserv/str_cli11.c

Figura 5.14 `str_cli` que chama `writen` duas vezes.

7-9 Tudo que alteramos é para chamar `writen` duas vezes: na primeira vez o primeiro byte de dados é gravado no soquete, seguido por uma pausa de um segundo, seguido pelo restante da linha. A intenção é que o primeiro `writen` elicie o RST e que então o segundo `writen` gere `SIGPIPE`.

Se executarmos o cliente no nosso host Linux, obteremos:

```
linux % tcpcli11 127.0.0.1
hi there                                    digitamos essa linha
hi there                                    que é ecoada pelo servidor
                                            aqui eliminamos o servidor-filho
bye                                         então digitamos essa linha
Broken pipe                                 isso é impresso pelo shell
```

Iniciamos o cliente, digitamos uma linha, verificamos essa linha ecoada corretamente e então terminamos o servidor-filho no host servidor. Em seguida, digitamos uma outra linha ("bye") e o Shell nos informa que o processo morreu com um sinal `SIGPIPE` (alguns shells não imprimem nada quando um processo morre sem fazer um dump da memória, mas o shell que estamos utilizando para esse exemplo, `bash`, informa o que queremos saber).

A maneira recomendável de tratar `SIGPIPE` depende do que a aplicação quer fazer quando isso ocorre. Se não houver nada especial a fazer, configurar a disposição de sinal para `SIG_IGN` é então fácil, assumindo que as operações de saída subseqüentes irão capturar o erro de `EPIPE` e terminar. Se ações especiais forem necessárias quando o sinal ocorre (talvez gravar em um arquivo de log), o sinal deve então ser capturado e quaisquer ações desejadas podem ser realizadas no handler de sinal. Esteja ciente, porém, de que se múltiplos soquetes estiverem em utilização, a entrega do sinal não informará qual deles encontrou o erro. Se precisarmos saber qual `write` causou o erro, devemos então ignorar o sinal ou retornar a partir do handler de sinal e tratar `EPIPE` a partir de `write`.

5.14 Travamento do host servidor

Esse cenário testará para ver o que acontece quando o host servidor pára. Para simular isso, devemos executar o cliente e o servidor em hosts diferentes. Em seguida, iniciamos o servidor e o cliente, digitamos uma linha no cliente para verificar se a conexão está ativa, desconectamos o host servidor da rede e digitamos uma outra linha no cliente. Isso também abrange o cenário de o host servidor estar inacessível quando o cliente envia dados (isto é, algum roteador intermediário pára depois que a conexão foi estabelecida).

Ocorrem os seguintes passos:

1. Quando o host servidor pára, nada é enviado nas conexões de rede existentes. Isto é, estamos assumindo que o host trava e não que seja desativado por um operador (o que discutiremos na Seção 5.16).
2. Digitamos uma linha de entrada para o cliente, a qual é gravada por `writen` (Figura 5.5) e enviada pelo TCP cliente como um segmento de dados. O cliente então bloqueia a chamada a `readline`, esperando a resposta ecoada.
3. Se observarmos a rede com `tcpdump`, veremos que o TCP cliente retransmite continuamente o segmento de dados, tentando receber um ACK do servidor. A Seção 25.11 do TCPv2 mostra um padrão típico para retransmissões de TCP: as implementações derivadas do Berkeley retransmitem o segmento de dados 12 vezes, esperando cerca de 9 minutos antes de desistir. Quando o TCP cliente por fim desiste (assumindo que o host servidor não foi reinicializado durante esse período, ou se o host servidor não travou, mas estava inacessível na rede, assumindo que o host ainda esteja inacessível), um erro é retornado para o processo cliente. Como o cliente é bloqueado na chamada a `readline`, ele retorna um erro. Supondo que o host servidor travou e não há absolutamente nenhuma resposta aos segmentos de dados do cliente, o erro é `ETIMEDOUT`. Mas, se algum roteador intermediário determinou que o host servidor estava inacessível e respondeu com uma mensagem "destination unreachable" de ICMP, o erro é `EHOSTUNREACH` ou `ENETUNREACH`.

Embora nosso cliente descubra (por fim) que o peer está fora do ar ou inacessível, há momentos em que queremos detectar isso mais rapidamente do que ter de esperar 9 minutos. A solução é impor um tempo-limite na chamada a `readline`, que discutiremos na Seção 14.2.

O cenário que acabamos de discutir detecta que o host servidor travou somente quando enviamos dados a ele. Se quisermos detectar o travamento do host servidor mesmo se não estivermos ativamente enviando-lhe dados, uma outra técnica é requerida. Discutiremos a opção de soquete `SO_KEEPALIVE` na Seção 7.5.

5.15 Travamento e reinicialização do host servidor

Nesse cenário, estabeleceremos uma conexão entre o cliente e o servidor e então suporemos que o host servidor trava e reinicializa. Na seção anterior, o host servidor estava parado quando lhe enviamos dados. Aqui, o deixaremos reinicializar antes de enviar os dados. A maneira mais fácil de simular isso é estabelecer a conexão, desconectar o servidor da rede, desativar (shut down) o host servidor e então reinicializá-lo e, em seguida, reconectar o host servidor à rede. Não queremos que o cliente veja a desativação do host servidor (que discutiremos na Seção 5.16).

Como afirmado na seção anterior, se o cliente não estiver ativamente enviando dados ao servidor quando o host servidor trava, ele não estará ciente de que o host servidor travou. (Isso supõe que não estamos utilizando a opção de soquete `SO_KEEPALIVE`.) Ocorrem os seguintes passos:

1. Iniciamos o servidor e então o cliente. Digitamos uma linha para verificar se a conexão foi estabelecida.
2. O host servidor trava e reinicializa.
3. Digitamos uma linha de entrada para o cliente, que é enviada como um segmento de dados TCP ao host servidor.
4. Quando o host servidor reinicializa depois do travamento, seu TCP perde todas as informações sobre as conexões que existiam antes do travamento. Portanto, o TCP servidor responde ao segmento de dados recebido do cliente com um RST.
5. Nosso cliente é bloqueado na chamada a `readline` quando o RST é recebido, fazendo com que `readline` retorne o erro `ECONNRESET`.

Se for importante ao nosso cliente detectar o travamento do host servidor, mesmo se não estiver ativamente enviando dados, alguma outra técnica (como a opção de soquete `SO_KEEPALIVE` ou alguma função de monitoração de cliente/servidor, como *heartbeat*) é requerida.

5.16 Desligamento do host servidor

As duas seções anteriores discutiram o travamento do host servidor ou sua inacessibilidade pela rede. Agora iremos considerar o que acontece se o host servidor for desativado por um operador enquanto nosso processo servidor está em execução nele.

Quando um sistema Unix é desligado, normalmente o processo `init` envia o sinal `SIGTERM` para todos os processos (podemos capturar esse sinal), espera algum período de tempo fixo (freqüentemente entre 5 e 20 segundos) e então envia o sinal `SIGKILL` (que não podemos capturar) a quaisquer processos que ainda estejam em execução. Isso fornece a todos os processos em execução um curto período de tempo para limpar e terminar. Se não capturarmos `SIGTERM` e terminarmos, nosso servidor será terminado pelo sinal `SIGKILL`. Quando o processo termina, todos os descritores abertos são fechados e então seguimos a mesma seqüência de passos discutida na Seção 5.12. Como afirmado nesta seção, devemos utilizar a função `select` ou `poll` no nosso cliente para fazer com que ele detecte o término do processo servidor logo que ele ocorre.

5.17 Resumo do exemplo de TCP

Antes que quaisquer cliente e servidor TCP possam se comunicar entre si, cada extremidade deve especificar o par de soquetes da conexão: o endereço IP local, a porta local, o endereço IP externo e a porta externa. Na Figura 5.15, mostramos esses quatro valores como marcadores. Essa figura é da perspectiva do cliente. O endereço IP e a porta externos devem ser especificados pelo cliente na chamada a `connect`. Normalmente, os dois valores locais são escolhidos pelo kernel como parte da função `connect`. O cliente tem a opção de especificar um ou ambos os valores locais, chamando `bind` antes de se conectar, mas isso não é comum.

Como mencionamos na Seção 4.10, o cliente pode obter os dois valores locais escolhidos pelo kernel chamando `getsockname` depois que a conexão é estabelecida.

A Figura 5.16 mostra os mesmos quatro valores, mas da perspectiva do servidor.

A porta local (a porta bem-conhecida do servidor) é especificada por `bind`. Normalmente, o servidor também especifica o endereço IP curinga nessa chamada. Se o servidor vincular o endereço IP curinga a um host multihomed, ele poderá determinar o endereço IP local chamando `getsockname` depois que a conexão é estabelecida (Seção 4.10). Os dois valores externos são retornados ao servidor por `accept`. Como mencionamos na Seção 4.10, se um outro programa é executado (por meio da chamada a `exec`) pelo servidor que chama `accept`, esse programa pode chamar `getpeername` para determinar o endereço e porta IP do cliente, se necessário.

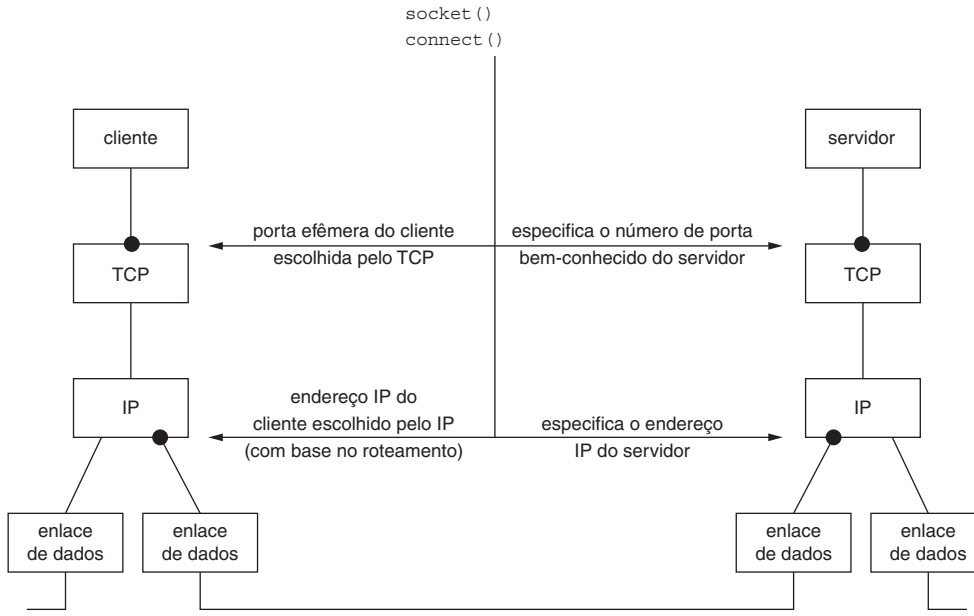


Figura 5.15 Resumo do TCP cliente/servidor da perspectiva do cliente.

5.18 Formato de dados

No nosso exemplo, o servidor nunca examina a solicitação que recebe do cliente. O servidor simplesmente lê todos os dados, inclui a nova linha e a envia de volta ao cliente, procurando somente ela. Isso é uma exceção, não a regra; normalmente devemos nos preocupar com o formato dos dados trocados entre o cliente e o servidor.

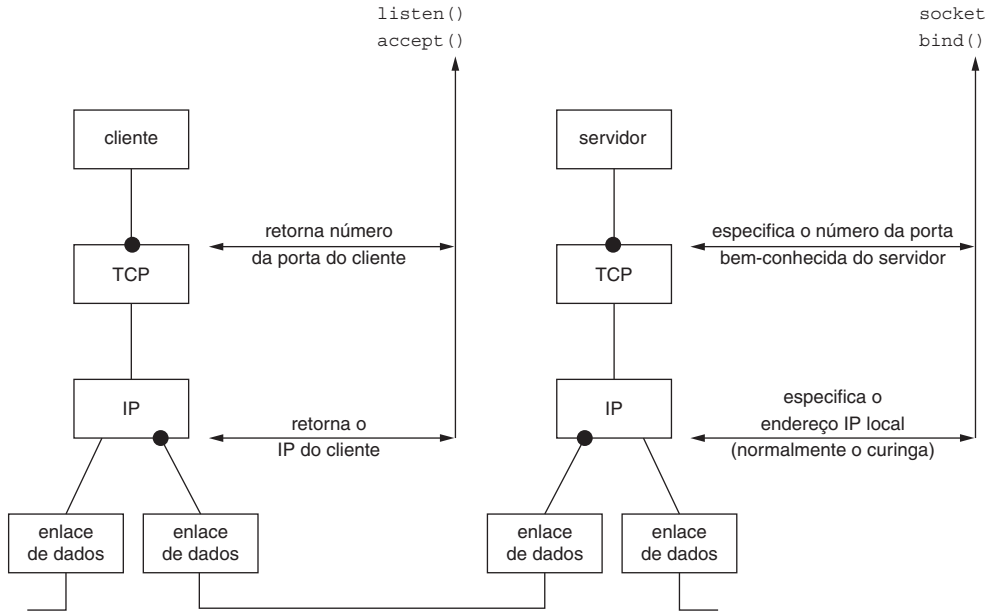


Figura 5.16 Resumo do TCP cliente/servidor da perspectiva do servidor.

Exemplo: passando strings de texto entre o cliente e o servidor

Vamos modificar o nosso servidor para que continue a ler uma linha de texto a partir do cliente, mas agora o servidor espera que a linha contenha dois inteiros separados por espaço em branco, e então retorna a soma desses dois inteiros. As funções `main` do nosso cliente e do servidor permanecem as mesmas, assim como nossa função `str_cli`. O que muda é a nossa função `str_echo`, que mostramos na Figura 5.17.

tcpcliserv/str_echo08.c

```
 1 #include   "unp.h"
 2 void
 3 str_echo(int sockfd)
 4 {
 5     long    arg1, arg2;
 6     ssize_t n;
 7     char    line[MAXLINE];
 8     for ( ; ; ) {
 9         if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
10            return;        /* conexão fechada pela outra extremidade */
11         if (sscanf(line, "%ld%ld", &arg1, &arg2) == 2)
12            snprintf(line, sizeof(line), "%ld\n", arg1 + arg2);
13         else
14            snprintf(line, sizeof(line), "input error\n");
15         n = strlen(line);
16         Writen(sockfd, line, n);
17     }
18 }
```

tcpcliserv/str_echo08.c

Figura 5.17 A função `str_echo` que adiciona dois números.

11-14 Chamamos `sscanf` para converter os dois argumentos provenientes das strings de texto em inteiros longos e então `snprintf` é chamada para converter o resultado em uma string de texto.

Esses novos cliente e servidor funcionam bem, independentemente do ordenamento de bytes dos hosts cliente e servidor.

Exemplo: passando estruturas binárias entre o cliente e o servidor

Agora, modificamos nossos cliente e servidor para passar valores binários pelo soquete, em vez de strings de texto. Veremos que isso não funciona quando o cliente e o servidor são executados em hosts com diferentes ordens de bytes, ou em hosts que não concordam com o tamanho de um inteiro longo (Figura 1.17).

As funções `main` de nossos cliente e servidor não mudam. Definimos uma estrutura para os dois argumentos, uma outra estrutura para o resultado e colocamos as duas definições no nosso cabeçalho `sum.h`, mostrado na Figura 5.18. A Figura 5.19 mostra a função `str_cli`.

tcpcliserv/sum.h

```
1 struct args {
2    long     arg1;
3    long     arg2;
4 };
5 struct result {
6    long     sum;
7 };
```

tcpcliserv/sum.h

Figura 5.18 Cabeçalho `sum.h`.

tcpcliserv/str_cli09.c

```
 1 #include   "unp.h"
 2 #include   "sum.h"
 3 void
 4 str_cli(FILE *fp, int sockfd)
 5 {
 6    char   sendline[MAXLINE];
 7    struct args args;
 8    struct result result;
 9    while (Fgets(sendline, MAXLINE, fp) != NULL) {
10        if (sscanf(sendline, "%ld%ld", &args.arg1, &args.arg2) != 2) {
11            printf("invalid input: %s", sendline);
12            continue;
13        }
14        Writen(sockfd, &args, sizeof(args));
15        if (Readn(sockfd, &result, sizeof(result)) == 0)
16            err_quit("str_cli: server terminated prematurely");
17        printf("%ld\n", result.sum);
18    }
19 }
```

tcpcliserv/str_cli09.c

Figura 5.19 A função `str_cli` que envia dois inteiros binários ao servidor.

10-14 `sscanf` converte os dois argumentos provenientes de strings de texto em binário e chamamos `writen` para enviar a estrutura ao servidor.

15-17 Chamamos `readn` para ler a resposta e imprimimos o resultado utilizando `printf`.

A Figura 5.20 mostra nossa função `str_echo`.

tcpcliserv/str_echo09.c

```
 1 #include   "unp.h"
 2 #include   "sum.h"

 3 void
 4 str_echo(int sockfd)
 5 {
 6     ssize_t n;
 7     struct args args;
 8     struct result result;

 9     for ( ; ; ) {
10         if ( (n = Readn(sockfd, &args, sizeof(args))) == 0)
11             return;        /* conexão fechada pela outra extremidade */

12         result.sum = args.arg1 + args.arg2;
13         Writen(sockfd, &result, sizeof(result));
14     }
15 }
```

tcpcliserv/str_echo09.c

Figura 5.20 A função `str_echo` que adiciona dois inteiros binários.

9-14 Lemos os argumentos chamando `readn`, calculamos e armazenamos a soma e chamamos `writen` para enviar de volta a estrutura de resultado.

Se executarmos o cliente e o servidor em duas máquinas com a mesma arquitetura, digamos duas máquinas SPARC, tudo funciona bem. Eis a interação de cliente:

```
solaris % tcpcli09 12.106.32.254
11 22                                   digitamos isso
33                                      essa é a resposta do servidor
-11 -44
-55
```

Mas quando o cliente e o servidor estão em duas máquinas com arquiteturas diferentes (digamos que o servidor está em um `freebsd` de um sistema SPARC baseado no ordenamento big-endian e o cliente está em um `linux` de um sistema Intel baseado no ordenamento little-endian), isso não funciona.

```
linux % tcpcli09 206.168.112.96
1 2                                     digitamos isso
3                                       e isso funciona
-22 -77                                 então digitamos isso
-16777314                               e não funciona
```

O problema é que os dois inteiros binários são enviados pelo soquete no formato little-endian pelo cliente, mas interpretados como inteiros no formato big-endian pelo servidor. Vemos que isso parece funcionar para inteiros positivos, mas falha para inteiros negativos (veja o Exercício 5.8). Há, na realidade, três problemas potenciais com esse exemplo:

1. Diferentes implementações armazenam números binários em diferentes formatos. Os formatos mais comuns são big-endian e little-endian, como descrevemos na Seção 3.4.
2. Diferentes implementações podem armazenar o mesmo tipo de dados em C de maneira diferente. Por exemplo, a maioria dos sistemas Unix de 32 bits utiliza 32 bits para um `long`, mas sistemas de 64 bits em geral utilizam 64 bits para o mesmo tipo de dados (Figura 1.17). Não há garantias de que um `short`, `int` ou `long` tenha um determinado tamanho.

3. Diferentes implementações empacotam estruturas de maneira diferente, dependendo do número de bits utilizado para os vários tipos de dados e das restrições de alinhamento da máquina. Portanto, nunca é sensato enviar estruturas binárias por um soquete.

Há duas soluções comuns para esse problema de formato de dados:

1. Passe todos os dados numéricos como strings de texto. Isso é o que fizemos na Figura 5.17. Essa solução supõe que os dois hosts têm o mesmo conjunto de caracteres.
2. Defina explicitamente os formatos binários dos tipos de dados suportados (número de bits, formato big ou little-endian) e passe todos os dados entre o cliente e o servidor nesse formato. Pacotes RPC normalmente utilizam essa técnica. A RFC 1832 (Srinivasan, 1995) descreve o padrão *External Data Representation* (XDR) utilizado com o pacote Sun RPC.

5.19 Resumo

A primeira versão do nosso cliente/servidor de eco totalizou aproximadamente 150 linhas (incluindo as funções `readline` e `writen`), mas mesmo assim forneceu uma grande quantidade de detalhes para examinar. O primeiro problema que encontramos foram os filhos zumbis e capturamos o sinal `SIGCHLD` para tratar isso. Nosso handler de sinal então chamou `waitpid` e demonstramos que precisamos chamar essa função em vez da função `wait` mais antiga, uma vez que sinais do Unix não são enfileirados. Isso nos levou a alguns detalhes sobre o tratamento de sinal POSIX (informações adicionais sobre esse tópico são fornecidas no Capítulo 10 de APUE).

O problema seguinte que encontramos foi o fato de o cliente não ter sido notificado quando o processo servidor terminou. Vimos que o nosso TCP do cliente foi notificado, mas não recebemos essa notificação, uma vez que estávamos bloqueados, esperando pela entrada de usuário. Utilizaremos a função `select` ou `poll` no Capítulo 6 para tratar esse cenário, esperando que qualquer um dos múltiplos descritores esteja pronto, em vez de bloquear um único descritor.

Também descobrimos que se o host servidor parar, não iremos detectar isso até que o cliente envie dados ao servidor. Algumas aplicações devem estar cientes desse fato com brevidade; na Seção 7.5, examinaremos a opção de soquete `SO_KEEPALIVE`.

Nosso exemplo simples trocou linhas de texto, o que não foi um problema uma vez que o servidor nunca examinou as linhas que ele ecoou. Enviar dados numéricos entre o cliente e o servidor pode levar a uma nova série de problemas, como mostrado.

Exercícios

5.1 Construa o servidor TCP das Figuras 5.2 e 5.3 e o cliente TCP das Figuras 5.4 e 5.5. Inicie o servidor e então o cliente. Digite algumas linhas para verificar se o cliente e o servidor estão funcionando. Termine o cliente digitando o seu caractere de EOF e anote a data/hora. Utilize `netstat` no host cliente para verificar se a extremidade cliente da conexão passa pelo estado TIME_WAIT. Execute `netstat` a cada cinco segundos aproximadamente para verificar quando o estado TIME_WAIT termina. Qual é o MSL para essa implementação?

5.2 O que acontece com nosso cliente/servidor de eco se executarmos o cliente e redirecionarmos a entrada-padrão para um arquivo binário?

5.3 Qual é a diferença entre o nosso cliente/servidor de eco e utilizar o cliente de Telnet para nos comunicarmos com nosso servidor de eco?

5.4 No nosso exemplo na Seção 5.12, verificamos que os dois primeiros segmentos do término da conexão são enviados (o FIN do servidor que é então reconhecido com um ACK pelo cliente) examinando os estados de soquete utilizando `netstat`. Os dois segmentos finais são trocados (um FIN do cliente que é reconhecido com um ACK pelo servidor)? Se sim ou não, por quê?

5.5 O que acontece no exemplo esboçado na Seção 5.14 se entre os Passos 2 e 3 reiniciarmos nossa aplicação de servidor no host servidor?

5.6 Para verificar se o que reivindicamos acontece com `SIGPIPE` na Seção 5.13, modifique a Figura 5.4 como a seguir. Escreva um handler de sinal para `SIGPIPE` que imprima apenas uma mensagem e retorne. Estabeleça esse handler de sinal antes de chamar `connect`. Altere o número de porta do servidor para 13, o servidor de data/hora. Quando a conexão é estabelecida, utilize `sleep` para fazê-la dormir por dois segundos, utilize `write` para gravar alguns bytes no soquete, utilize `sleep` para fazê-la dormir por outros dois segundos e utilize `write` para gravar mais alguns bytes no soquete. Execute o programa. O que acontece?

5.7 O que acontece na Figura 5.15 se o endereço IP do host servidor especificado pelo cliente na sua chamada a `connect` for o endereço IP associado ao datalink mais à direita no servidor, em vez do endereço IP associado ao datalink mais à esquerda no servidor?

5.8 Na nossa saída de exemplo na Figura 5.20, quando o cliente e o servidor estavam em sistemas com diferentes sistemas de ordenamento de bytes, o exemplo funcionou para números pequenos positivos, mas não para números pequenos negativos. Por quê? *(Dica:* Desenhe uma figura dos valores trocados no soquete, semelhante à Figura 3.9.)

5.9 No nosso exemplo nas Figuras 5.19 e 5.20, podemos resolver o problema do ordenamento de bytes fazendo o cliente converter os dois argumentos na ordem de bytes da rede utilizando `htonl`, fazendo o servidor então chamar `ntohl` em cada argumento antes de fazer a adição e então realizando uma conversão semelhante no resultado?

5.10 O que acontece nas Figuras 5.19 e 5.20 se o cliente estiver em um SPARC que armazena um `long` de 32 bits, mas o servidor estiver em um Digital Alfa que armazena um `long` de 64 bits? Isso muda se o cliente e o servidor forem trocados entre esses dois hosts?

5.11 Na Figura 5.15, dizemos que o endereço IP de cliente é escolhido pelo IP com base no roteamento. O que isso significa?

Capítulo 6

Multiplexação de E/S: As Funções `Select` e `Poll`

6.1 Visão geral

Na Seção 5.12, vimos nosso cliente TCP tratando duas entradas ao mesmo tempo: a entrada-padrão e um soquete TCP. Encontramos um problema quando o cliente foi bloqueado em uma chamada a `fgets` (na entrada-padrão) e o processo servidor foi eliminado. O TCP de servidor enviou corretamente um FIN ao TCP de cliente, mas como o processo cliente foi bloqueado na leitura da entrada-padrão, ele nunca viu o EOF até ler do soquete (possivelmente bem mais tarde). O que precisamos é a capacidade de dizer ao kernel que queremos ser notificados se uma ou mais condições de E/S estiverem prontas (isto é, a entrada está pronta para ser lida ou o descritor é capaz de receber outras saídas). Essa capacidade é chamada *multiplexação de E/S* e é fornecida pelas funções `select` e `poll`. Também discutiremos uma variação do POSIX mais recente chamada `pselect`.

> Alguns sistemas fornecem maneiras mais avançadas para que os processos esperem por uma lista de eventos. Um *dispositivo de poll* é um dos mecanismos fornecidos em diferentes formas por diferentes fornecedores. Esse mecanismo será descrito no Capítulo 14.

Em geral, a multiplexação de E/S é utilizada em aplicações de rede nos seguintes cenários:

- A multiplexação de E/S deve ser utilizada quando um cliente trata múltiplos descritores (normalmente, entrada interativa e um soquete de rede). Esse é o cenário que descrevemos anteriormente.
- É possível, mas raro, que o cliente trate múltiplos soquetes ao mesmo tempo. Mostraremos um exemplo disso utilizando a função `select` na Seção 16.5 no contexto de um cliente Web.
- Se um servidor TCP tratar tanto um soquete ouvinte como vários soquetes conectados, normalmente a multiplexação de E/S é utilizada, como mostraremos na Seção 6.8.
- Se um servidor trata tanto TCP como UDP, a multiplexação de E/S é normalmente utilizada. Mostraremos um exemplo disso na Seção 8.15.

- Se um servidor tratar múltiplos serviços e talvez múltiplos protocolos (por exemplo, o daemon `inetd`, que descreveremos na Seção 13.5), a multiplexação de E/S é normalmente utilizada.

A multiplexação de E/S não está limitada à programação de rede. Muitas aplicações não-triviais descobrem que precisam dessas técnicas.

6.2 Modelos de E/S

Antes de descrever `select` e `poll`, precisamos voltar e reexaminar um contexto maior, analisando as diferenças básicas entre os cinco modelos de E/S disponíveis sob Unix:

- E/S bloqueadora
- E/S não-bloqueadora
- Multiplexação de E/S (`select` e `poll`)
- E/S baseada em sinal (`SIGIO`)
- E/S assíncrona (funções `aio_` do POSIX)

Talvez você queira ler esta seção rapidamente e então voltar a ela à medida que encontrar diferentes modelos E/S descritos em mais detalhes nos capítulos posteriores.

Como mostraremos em todos os exemplos nesta seção, normalmente há duas fases distintas para uma operação de entrada:

1. Esperar que os dados estejam prontos
2. Copiar os dados do kernel para o processo

Para uma operação de entrada em um soquete, o primeiro passo normalmente envolve esperar que os dados cheguem na rede. Quando o pacote chega, ele é copiado para um buffer dentro do kernel. O segundo passo é copiar esses dados do buffer do kernel para o nosso buffer de aplicação.

Modelo de E/S bloqueadora

O modelo mais predominante para E/S é o *modelo de E/S bloqueadora*, que utilizamos em todos os exemplos mostrados até agora. Por default, todos os soquetes são bloqueadores. Utilizando um soquete de datagrama nos nossos exemplos, temos o cenário mostrado na Figura 6.1.

Utilizamos o UDP nesse exemplo, em vez do TCP, porque com o UDP o conceito de dados "prontos" para leitura é simples: um datagrama inteiro foi ou não recebido. Com o TCP isso se torna mais complicado, uma vez que entram em cena variáveis adicionais como o limite inferior do soquete.

Nos exemplos nesta seção, também nos referimos a `recvfrom` como uma chamada de sistema porque estamos diferenciando entre a nossa aplicação e o kernel. Independentemente de como `recvfrom` é implementado (como uma chamada de sistema em um kernel derivado do Berkeley ou como uma função que invoca a chamada de sistema `getmsg` no kernel de um System V), normalmente há uma alternação de executar na aplicação para executar no kernel, seguida posteriormente por um retorno à aplicação.

Na Figura 6.1, o processo chama `recvfrom` e a chamada de sistema não retorna até o datagrama chegar e ser copiado para o nosso buffer de aplicação, ou se ocorrer um erro. O erro mais comum é a chamada de sistema sendo interrompida por um sinal, como descrevemos na Seção 5.9. Dizemos que o nosso processo está *bloqueado* o tempo todo desde quando ele chama `recvfrom` até que ele retorne. Quando `recvfrom` retorna com sucesso, nossa aplicação processa o datagrama.

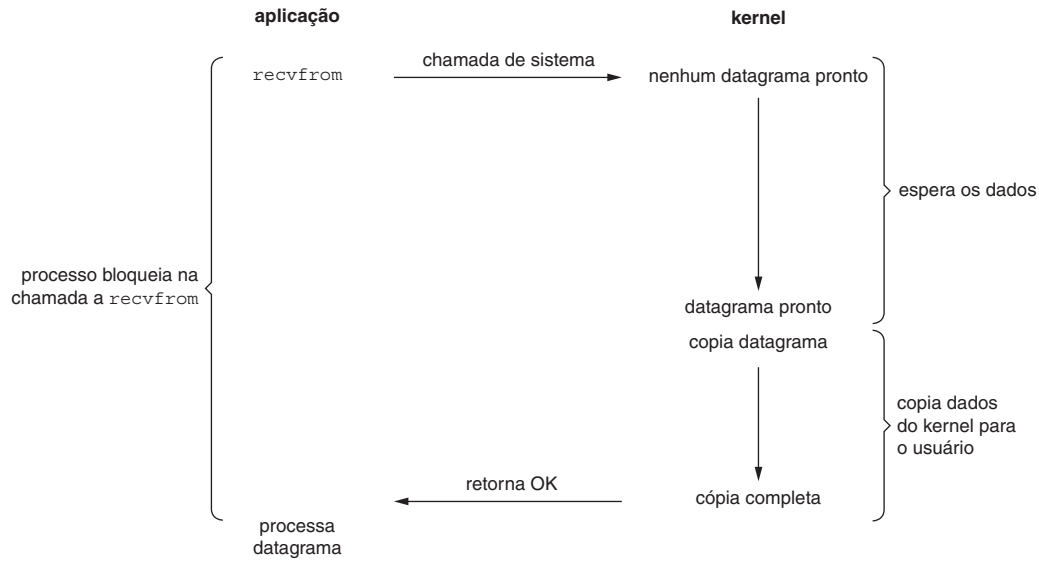


Figura 6.1 Modelo de E/S bloqueadora.

Modelo de E/S não-bloqueadora

Quando configuramos um soquete como não-bloqueador, estamos dizendo ao kernel "quando uma operação de E/S que eu solicito não puder ser completada sem colocar o processo para dormir, não coloque o processo para dormir, mas, em vez disso, retorne um erro". Descreveremos a E/S não-bloqueadora no Capítulo 16, mas a Figura 6.2 mostra um resumo do exemplo que estamos considerando.

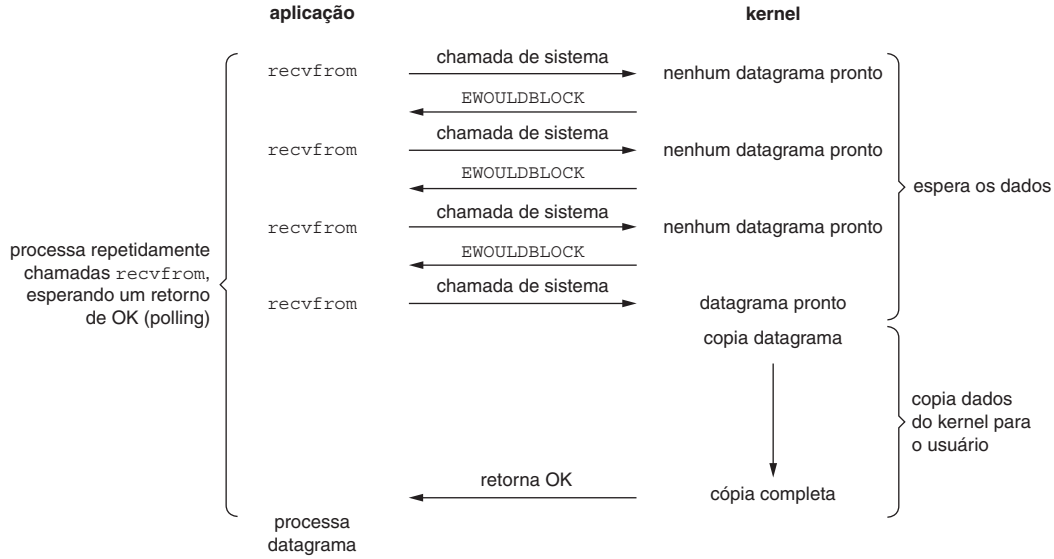


Figura 6.2 Modelo de E/S não-bloqueadora.

Nas três primeiras vezes que chamamos `recvfrom`, não há nenhum dado a retornar, assim o kernel imediatamente retorna um erro de `EWOULDBLOCK`. Na quarta vez que chamamos `recvfrom`, um datagrama está pronto, ele é copiado para o nosso buffer de aplicação e `recvfrom` retorna com sucesso. Então, processamos os dados.

Quando uma aplicação está em um loop que chama `recvfrom` em um descritor não-bloqueador como esse, ela é chamada *polling*. A aplicação consulta continuamente o kernel para verificar se alguma operação está pronta. Isso freqüentemente é um desperdício de tempo de CPU, mas esse modelo é ocasionalmente encontrado, em geral em sistemas dedicados a uma função.

Modelo de multiplexação de E/S

Com a *multiplexação de E/S*, chamamos `select` ou `poll` e bloqueamos uma dessas duas chamadas de sistema, em vez de bloquear a chamada de sistema E/S real. A Figura 6.3 é um resumo do modelo de multiplexação de E/S.

Bloqueamos uma chamada para `select`, esperando que o soquete de datagrama seja legível. Quando `select` retorna que o soquete tornou-se legível, chamamos então `recvfrom` para copiar o datagrama para o nosso buffer de aplicação.

Comparando a Figura 6.3 com a Figura 6.1, aparentemente não há nenhuma vantagem, pelo contrário, uma pequena desvantagem, porque utilizar `select` requer duas chamadas de sistema em vez de uma. Mas a vantagem em utilizar `select`, que veremos mais adiante neste capítulo, é que podemos esperar por mais de um descritor estar pronto.

> Um outro modelo de E/S estreitamente relacionado é utilizar múltiplos threads com a E/S bloqueadora. Esse modelo assemelha-se bastante ao descrito anteriormente, exceto que, em vez de utilizar `select` para bloquear múltiplos descritores de arquivo, o programa utiliza múltiplos threads (um por descritor de arquivo) e cada thread tem então a liberdade de invocar chamadas de sistema bloqueadoras como `recvfrom`.

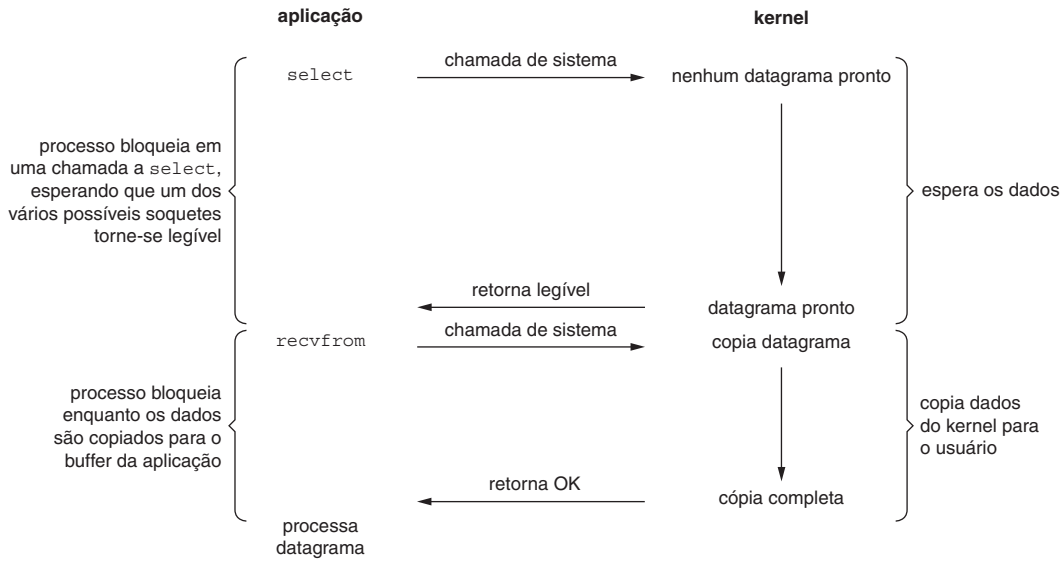


Figura 6.3 Modelo de multiplexação de E/S.

Modelo de E/S baseada em sinal

Também podemos utilizar sinais, informando o kernel a notificar com o sinal `SIGIO` quando o descritor está pronto. Chamamos isso *E/S baseada em sinal* e mostramos um resumo na Figura 6.4.

Primeiro ativamos o soquete para a E/S baseada em sinal (da maneira como descreveremos na Seção 25.2) e instalamos um handler de sinal utilizando a chamada de sistema `sigaction`. O retorno dessa chamada de sistema é imediato e o nosso processo continua; ele não é bloqueado. Quando o datagrama está pronto para ser lido, é gerado o sinal `SIGIO` para o nosso processo. Podemos ler o datagrama a partir do handler de sinal chamando `recvfrom` e então notificamos o loop principal de que os dados estão prontos para serem processados (isso é o que faremos na Seção 25.3), ou podemos notificar o loop principal e deixar que ele leia o datagrama.

Independentemente da maneira como tratamos o sinal, a vantagem desse modelo é que não somos bloqueados enquanto esperamos o datagrama chegar. O loop principal pode continuar a executar e simplesmente esperar ser notificado pelo handler de sinal de que os dados estão prontos para serem processados ou de que o datagrama está pronto para ser lido.

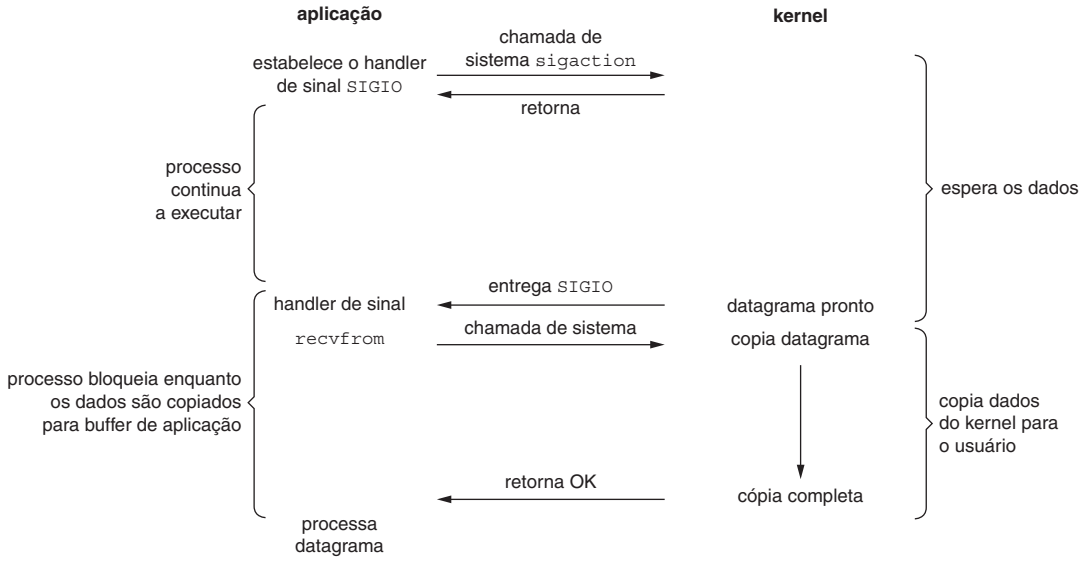


Figura 6.4 Modelo de E/S baseada em sinal.

Modelo de E/S assíncrona

A E/S assíncrona é definida pela especificação POSIX e as várias diferenças nas funções de *tempo real* que apareceram nos vários padrões foram reconciliadas e agrupadas para formar a especificação POSIX. Em geral, essas funções funcionam informando o kernel a iniciar a operação e notificar quando ela toda (incluindo a cópia dos dados do kernel para o nosso buffer) está completa. A principal diferença entre esse modelo e o modelo de E/S baseada em sinal, na seção anterior, é que com a E/S baseada em sinal o kernel informa quando uma operação de E/S pode ser *iniciada*, mas com a E/S assíncrona o kernel informa quando uma operação de E/S está *completa*. Mostramos um exemplo na Figura 6.5.

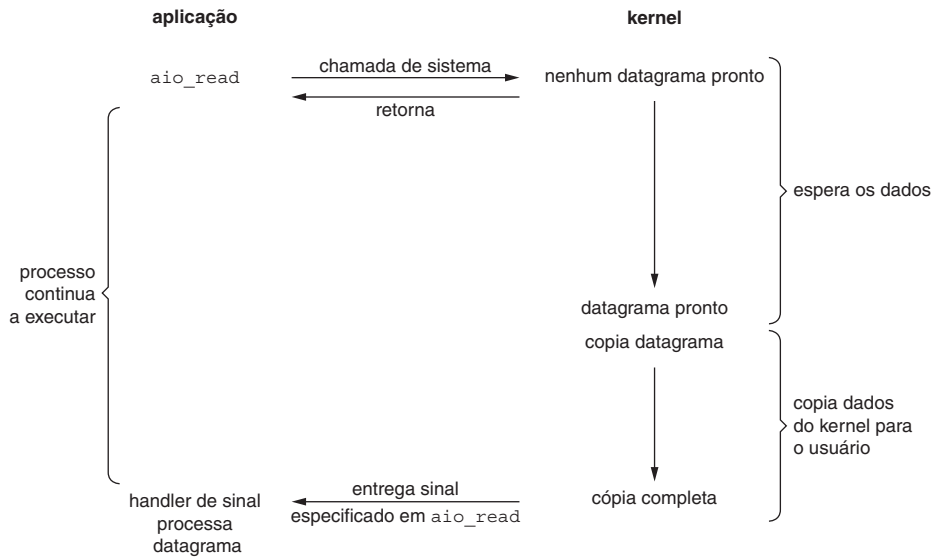


Figura 6.5 Modelo de E/S assíncrona.

Chamamos `aio_read` (as funções E/S assíncronas do POSIX iniciam com `aio_` ou `lio_`) e passamos para o kernel o descritor, o ponteiro do buffer, o tamanho do buffer (os mesmos três argumentos para `read`), o deslocamento no arquivo (semelhante a `lseek`) e como notificar quando a operação inteira está completa. Essa chamada de sistema retorna imediatamente e o nosso processo não é bloqueado enquanto esperamos que a E/S complete. Supomos nesse exemplo que solicitamos ao kernel gerar algum sinal quando a operação está completa. Esse sinal não é gerado até que os dados tenham sido copiados para nosso buffer de aplicação, que é diferente do modelo de E/S baseada em sinal.

> No momento em que este livro estava sendo escrito, poucos sistemas suportavam a E/S assíncrona do POSIX. Não temos certeza, por exemplo, se os sistemas irão suportá-la para soquetes. Utilizamos o modelo de E/S assíncrona aqui como exemplo comparativo ao modelo de E/S baseada em sinal.

Comparação dos modelos de E/S

A Figura 6.6 é uma comparação dos cinco diferentes modelos de E/S. Essa figura mostra que a principal diferença entre os quatro primeiros modelos é a primeira fase, uma vez que a segunda fase é a mesma em todos eles: o processo é bloqueado em uma chamada a `recvfrom` enquanto os dados são copiados do kernel para o buffer do chamador. A E/S assíncrona, porém, trata ambas as fases e é diferente das quatro primeiras.

E/S síncrona *versus* E/S assíncrona

O POSIX define esses dois termos como:

- Uma operação de E/S síncrona faz com que o processo solicitante seja bloqueado até que ela se complete.
- Uma operação de E/S assíncrona não faz com que o processo solicitante seja bloqueado.

Utilizando essas definições, todos os quatro primeiros modelos (E/S bloqueadora, E/S não-bloqueadora, multiplexação de E/S e E/S baseada em sinal) são síncronos porque a ope-

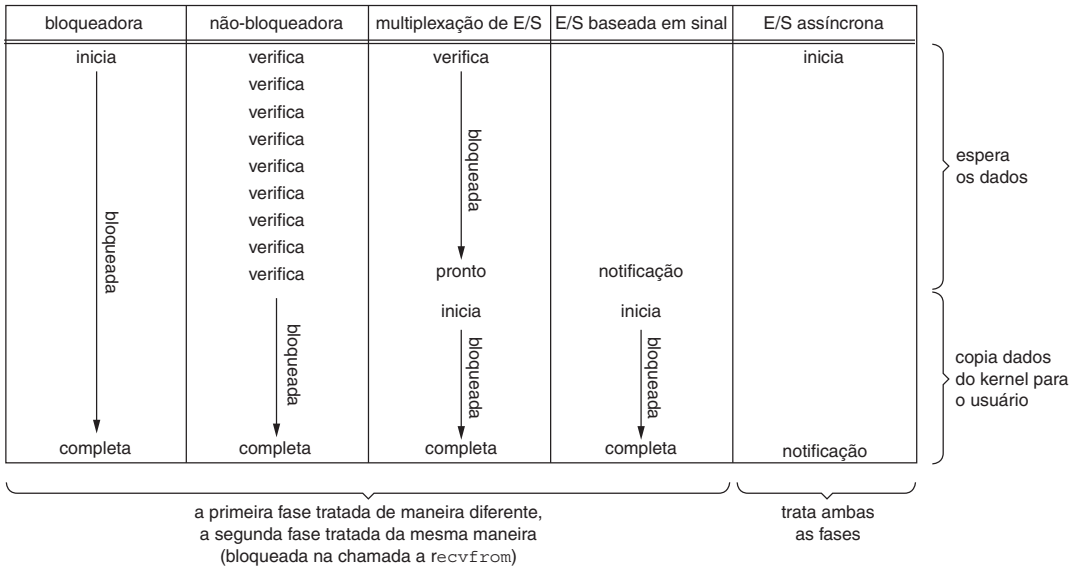


Figura 6.6 Comparação entre os cinco modelos de E/S.

ração de E/S real (`recvfrom`) bloqueia o processo. Somente o modelo de E/S assíncrona corresponde à definição da E/S assíncrona.

6.3 Função `select`

Essa função permite ao processo instruir o kernel a esperar que ocorra qualquer um de múltiplos eventos e ativar o processo somente quando um ou mais desses eventos ocorre ou quando um período de tempo especificado passou.

Como exemplo, podemos chamar `select` e dizer ao kernel para retornar somente quando:

- Qualquer um dos descritores no conjunto {1, 4, 5} está pronto para ler
- Qualquer um dos descritores no conjunto {2, 7} está pronto para gravar
- Qualquer um dos descritores no conjunto {1, 4} tem uma condição de exceção pendente
- 10,2 segundos se passaram

Isto é, dizemos ao kernel em quais descritores estamos interessados (para ler, gravar ou uma condição de exceção) e quanto tempo esperar. Os descritores em que estamos interessados não são restritos a soquetes; qualquer descritor pode ser testado utilizando `select`.

> Implementações derivadas do Berkeley sempre permitiram multiplexação de E/S com qualquer descritor. Originalmente, o SVR3 limitava a multiplexação de E/S a descritores que eram dispositivos STREAMS (Capítulo 31), mas essa limitação foi removida com o SVR4.

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

#include <sys/time.h>

int select (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
            const struct timeval *timeout);
```

Retorna: contagem positiva de descritores prontos, 0 no tempo limite, –1 em erro

Iniciamos nossa descrição dessa função com seu argumento final, que informa ao kernel quanto tempo esperar para que um dos descritores especificados torne-se pronto. Uma estrutura `timeval` especifica o número de segundos e de microssegundos.

```
struct timeval {
   long  tv_sec;        /* segundos */
   long  tv_usec;       /* microssegundos */
}%
```

Há três possibilidades:

1. Esperar eternamente – retorna somente quando um dos descritores especificados está pronto para a E/S. Para isso, especificamos o argumento *timeout* como um ponteiro nulo.
2. Esperar até um período fixo de tempo – retorna quando um dos descritores especificados está pronto para a E/S, mas não espera além do número de segundos e microssegundos especificados na estrutura `timeval` apontada pelo argumento de tempo limite *timeout*.
3. Não esperar absolutamente – retorna imediatamente após verificar os descritores. Isso é chamado *polling*. Para especificar isso, o argumento *timeout* deve apontar para uma estrutura `timeval` e o valor do timer (o número de segundos e microssegundos especificados pela estrutura) deve ser 0.

A espera nos dois primeiros cenários normalmente é interrompida se o processo capturar um sinal e retornar do handler de sinal.

> Kernels derivados do Berkeley nunca reiniciam `select` automaticamente (página 527 do TCPv2), enquanto o SVR4 irá reiniciar se o flag `SA_RESTART` for especificado quando o handler de sinal for instalado. Isso significa que, para portabilidade, devemos estar preparados para que `select` retorne um erro de `EINTR` se estivermos capturando sinais.

Embora a estrutura `timeval` permita especificar uma solução em microssegundos, a solução real suportada pelo kernel freqüentemente é mais grosseira. Por exemplo, muitos kernels do Unix arredondam o valor de tempo-limite até um múltiplo de 10 ms. Há também uma latência de agendamento envolvida, o que significa que demora algum tempo depois de o timer expirar para o kernel agendar a execução desse processo.

> Em alguns sistemas, `select` irá falhar com `EINVAL` se o campo `tv_sec` no tempo-limite for mais de 100 milhões de segundos. Naturalmente, esse é um tempo-limite muito grande (mais de três anos) e possivelmente não é muito útil, mas a particularidade é que a estrutura `timeval` pode representar valores que não são suportados por `select`.

O qualificador `const` no argumento de tempo-limite *timeout* significa que ele não é modificado por `select` no retorno. Por exemplo, se especificarmos um limite de tempo de 10 segundos e `select` retornar antes de o timer expirar com um ou mais descritores prontos ou com um erro de `EINTR`, a estrutura `timeval` não será atualizada com o número de segundos remanescente quando a função retorna. Se quisermos conhecer esse valor, devemos obter a data/hora do sistema antes de chamar `select` e então novamente quando ela retorna e subtrair os dois (qualquer programa robusto levará em conta o fato de que a data/hora do sistema pode ser ajustada pelo administrador ou, ocasionalmente, por um daemon como `ntpd`).

> Algumas versões do Linux modificam a estrutura `timeval`. Portanto, para portabilidade, assuma que a estrutura `timeval` não é definida no retorno e inicialize-a antes de cada chamada a `select`. O `POSIX` especifica o qualificador `const`.

Os três argumentos intermediários, *readset, writeset* e *exceptset*, especificam os descritores que queremos que o kernel teste para ler, gravar e para condições de exceção. Há somente duas condições de exceção atualmente suportadas:

1. A chegada de dados fora da banda para um soquete. Descreveremos isso em mais detalhes no Capítulo 24.
2. A presença de informações sobre o *status* de controle a serem lidas a partir do lado mestre de um pseudoterminal que foi colocado no modo de pacote. Não discutiremos os pseudoterminais neste livro.

Um problema de *design* é como especificar um ou mais valores descritores para cada um desses três argumentos. `select` utiliza conjuntos de descritores, em geral um array de inteiros, com cada bit em cada inteiro correspondendo a um descritor. Por exemplo, utilizando inteiros de 32 bits, o primeiro elemento do array corresponde aos descritores de 0 a 31, o segundo elemento do array corresponde aos descritores de 32 a 63, e assim por diante. Todos os detalhes da implementação são irrelevantes à aplicação e são ocultados no tipo de dados `fd_set` e nas quatro macros a seguir:

```
void FD_ZERO(fd_set *fdset);           /* limpa todos os bits em fdset */
void FD_SET(int fd, fd_set *fdset);    /* ativa o bit para fd em fdset */
void FD_CLR(int fd, fd_set *fdset);    /* desativa o bit para fd em fdset */
int FD_ISSET(int fd, fd_set *fdset);   /* o bit para fd em fdset está ativo ?*/
```

Alocamos um conjunto de descritores do tipo de dados `fd_set`, configuramos e testamos os bits no conjunto utilizando essas macros e também as atribuímos a um outro conjunto de descritores por meio de um sinal de igual (=) em C.

> O que estamos descrevendo, um array de inteiros utilizando um bit por descritor, é somente uma das possíveis maneiras de implementar `select`. Contudo, é comum referir-se aos descritores individuais dentro de um conjunto de descritores como *bits*, como em "ative o bit para o descritor ouvinte no conjunto de leitura".
>
> Veremos na Seção 6.10 que a função `poll` utiliza uma representação completamente diferente: um array de comprimento variável de estruturas com uma estrutura por descritor.

Por exemplo, para definir uma variável do tipo `fd_set` e então ativar os bits para os descritores 1, 4 e 5, escrevemos:

```
fd_set   rset;

FD_ZERO(&rset);     /* inicializa o conjunto: todos os bits desativados */
FD_SET(1, &rset);   /* ativa o bit para fd 1 */
FD_SET(4, &rset);   /* ativa o bit para fd 4 */
FD_SET(5, &rset);   /* ativa o bit para fd 5 */
```

É importante inicializar o conjunto, uma vez que resultados imprevisíveis podem ocorrer se o conjunto for alocado como uma variável automática e não-inicializado.

Qualquer um dos três argumentos intermediários para `select`, `readset`, `writeset` ou `exceptset` pode ser especificado como um ponteiro nulo se não estivermos interessados nessa condição. De fato, se todos os três ponteiros forem nulos, temos então um timer com precisão mais alta do que a função `sleep` normal do Unix (que dorme por múltiplos de um segundo). A função `poll` fornece uma funcionalidade semelhante. As Figuras C.9 e C.10 da APUE mostram uma função `sleep_us` implementada com `select` e `poll` que dorme por múltiplos de um microssegundo.

O argumento *maxfdp1* especifica o número de descritores a ser testado. Seu valor é o descritor máximo a ser testado mais um (por isso, o nosso nome de *maxfdp1*). Os descritores 0, 1, 2, até e incluindo *maxfdp1-1*, são testados.

A constante `FD_SETSIZE`, definida incluindo `<sys/select.h>`, é o número de descritores no tipo de dados `fd_set`. Seu valor freqüentemente é 1024, mas poucos programas utilizam esse número de descritores. O argumento *maxfdp1* força-nos a calcular o descritor maior em que estamos interessados e então informa ao kernel esse valor. Por exemplo, dado o código anterior que ativa os indicadores para os descritores 1, 4 e 5, o valor de *maxfdp1* é 6.

A razão pela qual é 6 e não 5 é que estamos especificando o número de descritores, não o maior valor, e os descritores iniciam em 0.

> A razão por que esse argumento existe, junto com o fardo do cálculo de seu valor, é puramente para eficiência. Embora cada `fd_set` tenha espaço para muitos descritores, em geral 1.024, esse número é bem maior que o número utilizado por um processo típico. O kernel ganha eficiência não copiando partes desnecessárias do conjunto de descritores entre ele e o processo e não testando bits que são sempre 0 (Seção 16.13 do TCPv2).

`select` modifica os conjuntos de descritores apontados pelos ponteiros *readset*, *writeset* e *exceptset*. Esses três argumentos são de valor-resultado. Quando chamamos a função, especificamos os valores dos descritores em que estamos interessados e, no retorno, o resultado indica quais descritores estão prontos. Utilizamos a macro `FD_ISSET` no retorno para testar um descritor específico em uma estrutura `fd_set`. Qualquer descritor que não está pronto no retorno terá seu bit correspondente zerado no conjunto de descritores. Para tratar isso, ativamos todos os bits em que estamos interessados em todos os conjuntos de descritores cada vez que chamamos `select`.

> Os dois erros de programação mais comuns ao utilizar `select` é esquecer de adicionar um ao maior número do descritor e esquecer que os conjuntos de descritores são argumentos valor-resultado. O segundo erro resulta em `select` sendo chamada com um bit configurado como 0 no conjunto de descritores, quando pensamos que o bit é 1.

O valor de retorno dessa função indica o número total de bits que estão prontos em todos os conjuntos de descritores. Se o valor de timer expirar antes que qualquer um dos descritores esteja pronto, um valor 0 é retornado. Um valor de retorno de –1 indica um erro (o que pode acontecer, por exemplo, se a função for interrompida por um sinal capturado).

> As distribuições prévias do SVR4 tinham um bug na sua implementação de `select`: se o mesmo bit estivesse em múltiplos conjuntos (por exemplo, um descritor pronto tanto para leitura como para gravação), ele seria contado somente uma vez. As distribuições atuais corrigiram esse bug.

Sob quais condições um descritor está pronto?

Temos conversado sobre esperar que um descritor esteja pronto para E/S (leitura ou gravação) ou ter uma condição de exceção pendente nele (dados fora da banda). Embora para os descritores a capacidade de serem legíveis e graváveis seja óbvia, como arquivos normais, devemos ser mais específicos sobre as condições que fazem com que `select` retorne "ready" para soquetes (Figura 16.52 de TCPv2).

1. Um soquete está pronto para leitura se qualquer uma das quatro condições a seguir for verdadeira:
   a. O número de bytes de dados no buffer de recebimento de soquete é maior ou igual ao tamanho atual da marca de menor nível do buffer de recebimento de soquete. Uma operação de leitura no soquete não bloqueará e retornará um valor maior que 0 (isto é, os dados que estão prontos para serem lidos). Podemos configurar essa marca de menor nível (*low-water mark*) utilizando a opção de soquete `SO_RCVLOWAT`. Ela estabelece o padrão de 1 para TCP e soquetes UDP.
   b. A metade leitura da conexão está fechada (isto é, uma conexão TCP que recebeu um FIN). Uma operação de leitura no soquete não bloqueará e retornará 0 (isto é, EOF).
   c. O soquete é um soquete ouvinte e o número de conexões completadas não é zero. Uma `accept` no soquete ouvinte normalmente não será bloqueada, embora iremos descrever uma condição de sincronização na Seção 16.6 sob a qual `accept` pode ser bloqueada.

d. Um erro de soquete está pendente. Uma operação de leitura no soquete não será bloqueada e retornará um erro (–1) com `errno` configurado como a condição de erro específica. Esses erros pendentes também podem ser buscados e limpos chamando `get_sockopt` e especificando a opção de soquete `SO_ERROR`.

2. Um soquete está pronto para gravar se qualquer uma das quatro condições a seguir for verdadeira:

a. O número de bytes do espaço disponível no buffer de envio do soquete é maior ou igual ao tamanho atual da marca de menor nível do buffer de envio do soquete *e*: (i) o soquete está conectado ou (ii) o soquete não requer uma conexão (por exemplo, UDP). Isso significa que, se configurarmos o soquete como não-bloqueador (Capítulo 16), uma operação de gravação não será bloqueada e retornará um valor positivo (por exemplo, o número de bytes aceito pela camada de transporte). Podemos configurar essa marca de menor nível utilizando a opção de soquete `SO_SNDLOWAT`. Essa marca de menor nível normalmente assume o default de 2048 para soquetes TCP e UDP.

b. A metade gravação da conexão está fechada. Uma operação de gravação no soquete irá gerar `SIGPIPE` (Seção 5.12).

c. Um soquete que utiliza uma `connect` não-bloqueadora completou a conexão ou `connect` falhou.

d. Um erro de soquete está pendente. Uma operação de gravação no soquete não será bloqueada e retornará um erro (–1) com `errno` configurado como a condição de erro específica. Esses erros pendentes também podem ser buscados e limpos chamando `getsockopt` com a opção de soquete `SO_ERROR`.

3. Um soquete tem uma condição de exceção pendente se houver dados fora da banda para o soquete ou se ele ainda estiver na marca de fora da banda. (Descreveremos os dados fora da banda no Capítulo 24.)

> Nossas definições de "legível" e "gravável" são tomadas diretamente de `soreadable` e `sowriteable` das macros do kernel nas páginas 530 e 531 do TCPv2. De maneira semelhante, nossa definição de "condição de exceção" para um soquete provém da função `soo_select` nessas mesmas páginas.

Observe que, quando um erro ocorre em um soquete, ele é marcado como tanto legível como gravável por `select`.

O propósito do recebimento e envio de marcas de menor nível é fornecer à aplicação controle sobre o número de dados que deve estar disponível para leitura ou quanto espaço deve estar disponível para gravação antes de `select` retornar um *status* de legível ou gravável. Por exemplo, se soubermos que a nossa aplicação não tem nada de produtivo a fazer a menos que pelo menos 64 bytes de dados estejam presentes, poderemos configurar a marca de menor nível de recebimento como 64 para evitar que `select` nos acorde se menos de 64 bytes estiverem prontos para leitura.

Enquanto a marca de menor nível de envio para um soquete UDP for menor que o tamanho do buffer de envio (que sempre deve ser o relacionamento default), o soquete UDP sempre será gravável, uma vez que uma conexão não é requerida.

A Figura 6.7 resume as condições recém-descritas que fazem com que um soquete esteja pronto para `select`.

Número máximo de descritores para `select`

Dissemos anteriormente que a maioria das aplicações não utiliza uma grande quantidade de descritores. É raro, por exemplo, encontrar uma aplicação que utiliza centenas de descritores. Mas essas aplicações existem, e elas freqüentemente utilizam `select` para multiplexar os descritores. Quando `select` foi originalmente projetada, o SO normalmente tinha um limi-

Condição	Legível?	Gravável?	Exceção?
Dados para ler	•		
Metade leitura da conexão fechada	•		
Nova conexão pronta para o soquete ouvinte	•		
Espaço disponível para gravação		•	
Metade gravação da conexão fechada		•	
Erro pendente	•	•	
Dados TCP fora da banda			•

Figura 6.7 O resumo das condições que fazem com que um soquete esteja pronto para `select`.

te superior quanto ao número máximo de descritores por processo (o limite do 4.2BSD era 31) e `select` simplesmente utilizava esse mesmo limite. Mas as versões atuais do Unix permitem um número praticamente ilimitado de descritores por processo (freqüentemente limitado somente pela quantidade de memória e por quaisquer limites administrativos), portanto, a pergunta é: como isso afeta `select`?

Muitas implementações têm declarações semelhantes à seguinte, que são tiradas do cabeçalho `<sys/types.h>` do 4.4BSD:

```
/*
 * Select utiliza máscaras de bit de descritores de arquivo em longos.
 * Essas macros manipulam tais campos de bit (as macros para sistemas
 * de arquivo utilizam chars).
 * FD_SETSIZE pode ser definido pelo usuário, mas o default aqui deve ser
 * suficiente para a maioria dos casos.
 */
#ifndef FD_SETSIZE
#define FD_SETSIZE      256
#endif
```

Isso nos faz pensar que podemos simplesmente definir `FD_SETSIZE`, usando `#define FD_SETSIZE`, como algum valor maior antes de incluir esse cabeçalho para aumentar o tamanho dos conjuntos de descritores utilizados por `select`. Infelizmente, isso normalmente não funciona.

> Para ver o que há de errado, observe que a Figura 16.53 do TCPv2 declara três conjuntos de descritores dentro do kernel e também utiliza a definição do kernel de `FD_SETSIZE` como o limite superior. A única maneira de aumentar o tamanho dos conjuntos de descritores é aumentar o valor de `FD_SETSIZE` e então recompilar o kernel. Alterar o valor sem recompilar o kernel não é adequado.

Alguns fornecedores alteram suas implementações de `select` para permitir que o processo defina `FD_SETSIZE` como um valor maior que o default. BSD/OS alterou a implementação do kernel para permitir conjuntos maiores de descritores e também para fornecer quatro novas macros `FD_`*`xxx`* para alocar e manipular dinamicamente esses conjuntos maiores. Do ponto de vista da portabilidade, porém, tenha cuidado ao utilizar grandes conjuntos de descritores.

6.4 Função `str_cli` (revisitada)

Agora, podemos regravar nossa função `str_cli` da Seção 5.5, dessa vez utilizando `select`; somos então notificados logo que o processo servidor termina. O problema com essa versão anterior foi o fato de que poderíamos ser bloqueados na chamada para `fgets` quando algo acontecia no soquete. Nossa nova versão, em vez disso, bloqueia na chamada para `select`, esperando a entrada-padrão ou que o soquete esteja legível. A Figura 6.8 mostra as várias condições tratadas pela nossa chamada para `select`.

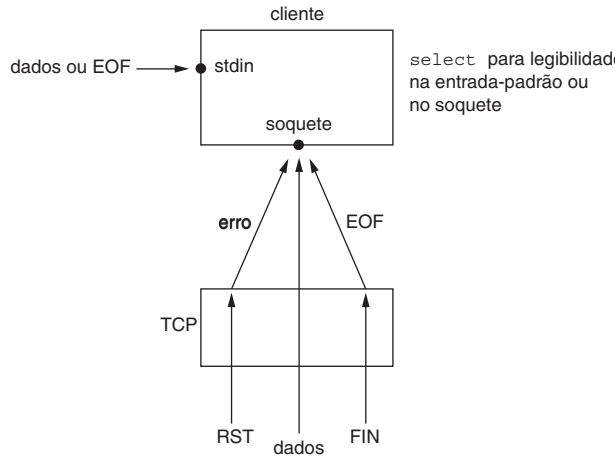


Figura 6.8 Condições tratadas por `select` em `str_cli`.

Três condições são tratadas com o soquete:

1. Se o TCP de peer enviar dados, o soquete torna-se legível e `read` retorna maior que 0 (isto é, o número de bytes de dados).
2. Se o TCP de peer enviar um FIN (o processo do peer termina), o soquete torna-se legível e read retorna 0 (EOF).
3. Se o TCP de peer enviar um RST (o host de peer parou e reinicializou), o soquete torna-se legível, `read` retorna –1 e `errno` contém o código de erro específico.

A Figura 6.9 mostra o código-fonte dessa nova versão.

select/strcliselect01.c

```
 1 #include   "unp.h"

 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     int    maxfdp1;
 6     fd_set rset;
 7     char   sendline[MAXLINE], recvline[MAXLINE];

 8     FD_ZERO(&rset);
 9     for ( ; ; ) {
10         FD_SET(fileno(fp), &rset);
11         FD_SET(sockfd, &rset);
12         maxfdp1 = max(fileno(fp), sockfd) + 1;
13         Select(maxfdp1, &rset, NULL, NULL, NULL);

14         if (FD_ISSET(sockfd, &rset)) {  /* o soquete é legível */
15            if (Readline(sockfd, recvline, MAXLINE) == 0)
16                err_quit("str_cli: server terminated prematurely");
17            Fputs(recvline, stdout);
18         }

19         if (FD_ISSET(fileno(fp), &rset)) {  /* a entrada é legível */
20            if (Fgets(sendline, MAXLINE, fp) == NULL)
21                return;            /* tudo feito */
22            Writen(sockfd, sendline, strlen(sendline));
23         }
24     }
25 }
```

select/strcliselect01.c

Figura 6.9 Implementação da função `str_cli` que utiliza `select` (aprimorada na Figura 6.13).

Chamada a `select`

8-13 Precisamos somente de um conjunto de descritores – para verificar a legibilidade. Esse conjunto é inicializado por `FD_ZERO` e então dois bits são ativados utilizando `FD_SET`: o bit correspondente ao ponteiro de arquivo da E/S-padrão, `fp`, e o bit correspondente ao soquete, `sockfd`. A função `fileno` converte um ponteiro de arquivo de E/S-padrão no seu descritor correspondente. `select` (e `poll`) funciona somente com descritores.

`select` é chamada depois de calcular o valor máximo dos dois descritores. Na chamada, o ponteiro de configuração de gravação e o ponteiro de configuração de exceção são nulos. O argumento final (o limite de tempo) também é um ponteiro nulo uma vez que queremos que a chamada seja bloqueada até que algo esteja pronto.

Tratamento do soquete legível

14-18 Se, no retorno de `select`, o soquete estiver legível, a linha ecoada é lida com `readline` e a saída por `fputs`.

Tratamento da entrada legível

19-23 Se a entrada-padrão estiver legível, uma linha é lida por `fgets` e gravada no soquete utilizando `writen`.

Observe que as quatro funções de E/S são utilizadas da mesma maneira como na Figura 5.5, `fgets`, `writen`, `readline` e `fputs`, mas a ordem de fluxo dentro da função mudou. Em vez do fluxo da função ser direcionado pela chamada a `fgets`, agora ele é direcionado pela chamada a `select`. Apenas com algumas linhas adicionais de código na Figura 6.9, comparado à Figura 5.5, aumentamos significativamente a robustez do nosso cliente.

6.5 Entrada em lote e armazenamento em buffer

Infelizmente, nossa função `str_cli` ainda não está correta. Primeiro, voltemos à nossa versão original, a Figura 5.5. Ela opera no modo "pare e espere", que é adequado à utilização interativa: ela envia uma linha ao servidor e então espera a resposta. Esse período de tempo é um RTT mais o tempo de processamento do servidor (próximo de 0 para um servidor de eco simples). Portanto, podemos estimar quanto tempo levará para que um dado número de linhas seja ecoado se soubermos o RTT entre o cliente e o servidor.

O programa `ping` é uma maneira fácil de medir RTTs. Se executarmos `ping` para o host `connix.com` do nosso host `solaris`, a média de RTT em 30 segmentos é de 175 ms. A página 89 do TCPv1 mostra que essas medidas de `ping` são para um datagrama IP cujo comprimento tem 84 bytes. Se selecionarmos as 2 mil primeiras linhas do arquivo `termcap` do Solaris, o tamanho do arquivo resultante será de 98.349 bytes, para uma média de 49 bytes por linha. Se adicionarmos os tamanhos do cabeçalho IP (20 bytes) e do cabeçalho TCP (20), o segmento TCP médio terá aproximadamente 89 bytes, quase o mesmo tamanho do pacote `ping`. Portanto, podemos estimar que o período de tempo total será de aproximadamente 350 segundos para 2 mil linhas (2.000 x 0,175 seg). Se executarmos nosso cliente de eco de TCP do Capítulo 5, o tempo real será de cerca de 354 segundos, bem próximo da nossa estimativa.

Se considerarmos a rede entre o cliente e o servidor como um pipe full-duplex, com solicitações do cliente para o servidor e respostas na direção inversa, a Figura 6.10 então mostrará nosso modo de "parar e esperar".

Uma solicitação é enviada pelo cliente no tempo 0, e assumimos um RTT de 8 unidades de tempo. A resposta enviada no tempo 4 é recebida no tempo 7. Também assumimos que não há nenhum servidor processando a data/hora e que o tamanho da solicitação é o mesmo da resposta. Mostramos somente os pacotes de dados entre o cliente e servidor, ignorando os reconhecimentos de TCP que também passam pela rede.

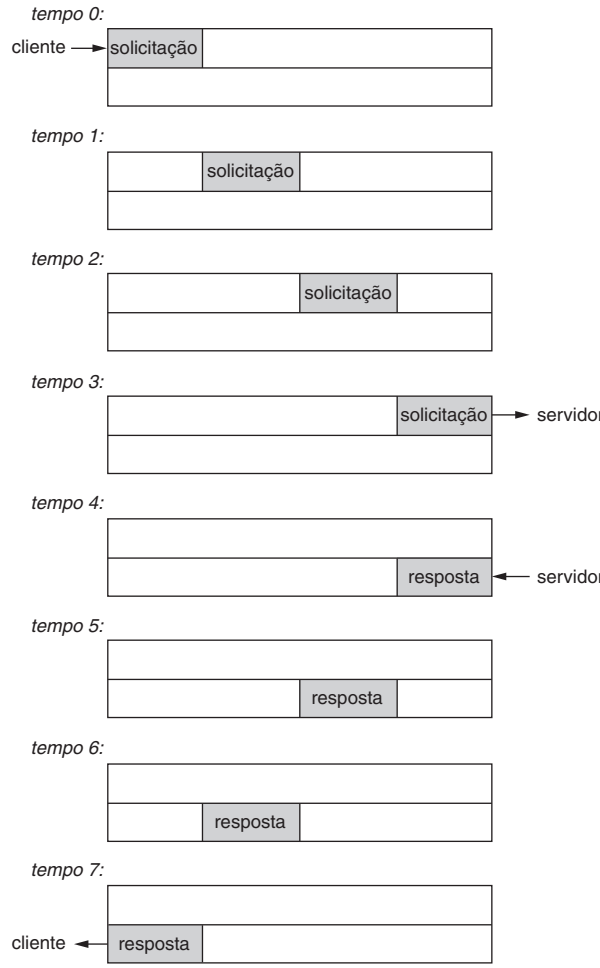


Figura 6.10 Linha do tempo do modo de "parar e esperar": entrada interativa.

Como há um retardo entre o envio de um pacote e o recebimento desse pacote na outra extremidade do pipe e como o pipe é full-duplex, nesse exemplo, estamos utilizando apenas um oitavo da capacidade do pipe. Esse modo de "parar e esperar" é adequado para entrada interativa, mas como nosso cliente lê a partir da entrada-padrão e grava na saída-padrão, e uma vez que é trivial sob os shells do Unix redirecionar a entrada e a saída, podemos facilmente executar nosso cliente em um modo de lote. Contudo, quando redirecionamos a entrada e a saída, o arquivo de saída resultante sempre é menor que o arquivo de entrada (e esses arquivos deveriam ser idênticos para um servidor de eco).

Para ver o que está acontecendo, perceba que, em um modo de lote, podemos continuar a enviar solicitações tão rapidamente quanto a rede pode aceitá-las. O servidor processa essas solicitações e envia de volta as respostas na mesma taxa. Isso resulta no preenchimento do pipe no tempo 7, como mostrado na Figura 6.11.

Aqui supomos que, depois de enviar a primeira solicitação, enviamos imediatamente uma outra e então outra. Também supomos que podemos continuar a enviar solicitações tão rapi-

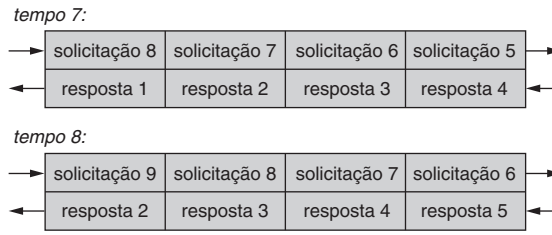


Figura 6.11 Preenchendo o pipe entre o cliente e o servidor: modo de lote.

damente quanto a rede possa aceitá-las, junto com processamento tão rápido das respostas quanto a rede possa fornecê-las.

> Há várias sutilezas ao lidar com fluxo de dados em volume do TCP que ignoramos aqui, como seu algoritmo de inicialização lento, que limita a taxa com que os dados são enviados em uma conexão nova ou desocupada e os ACKs que retornam. Tudo isso é discutido no Capítulo 20 do TCPv1.

Para ver o problema com a nossa função `str_cli` revisitada na Figura 6.9, assuma que o arquivo de entrada contém somente nove linhas. A última linha é enviada no tempo 8, como mostrado na Figura 6.11. Mas não podemos fechar a conexão depois de escrever essa solicitação porque ainda há outras solicitações e respostas no pipe. A causa desse problema é o nosso tratamento de um EOF na entrada: a função retorna para a função `main`, que então termina. Mas, em um modo de lote, um EOF na entrada não implica que terminamos a leitura do soquete; talvez ainda haja solicitações a caminho do servidor, ou respostas retornando deste.

O que precisamos é de uma maneira de fechar uma metade da conexão TCP. Isto é, queremos enviar um FIN ao servidor, informando que concluímos o envio de dados, mas deixar o descritor de soquete aberto para leitura. Isso é feito com a função `shutdown`, descrita na próxima seção.

Em geral, armazenar em buffer para melhorar o desempenho adiciona complexidade a uma aplicação de rede e o código na Figura 6.9 sofre dessa complexidade. Considere o caso em que várias linhas de entrada estão disponíveis a partir da entrada-padrão. `select` fará com que o código na linha 20 leia a entrada utilizando `fgets` e isso, por sua vez, irá ler as linhas disponíveis em um buffer utilizado por stdio. Mas, `fgets` somente retorna uma única linha e deixa quaisquer dados remanescentes residindo no buffer de stdio. O código na linha 22 da Figura 6.9 grava essa única linha no servidor e então `select` é chamada novamente para esperar por mais trabalho, mesmo se houver linhas adicionais a serem consumidas no buffer de stdio. A razão disso é que `select` não conhece nada sobre os buffers utilizados por stdio – ela somente mostrará legibilidade do ponto de vista da chamada de sistema `read`, não de chamadas como `fgets`. Por essa razão, mesclar stdio e `select` é considerado muito propenso a erros e somente deve ser feito com bastante cuidado.

O mesmo problema existe com a chamada a `readline` no exemplo da Figura 6.9. Em vez de os dados serem ocultados de `select` em um buffer de stdio, eles são ocultados no buffer de `readline`. Lembre-se de que na Seção 3.9 fornecemos uma função que dá visibilidade ao buffer de `readline`, portanto, uma possível solução é modificar nosso código para utilizar essa função *antes* de chamar `select` para ver se os dados já foram lidos, mas não consumidos. Mas, de novo, a complexidade foge de controle rapidamente quando temos de tratar o caso em que o buffer de `readline` contém uma linha parcial (o que significa que ainda precisamos ler mais), bem como quando ele contém uma ou mais linhas completas (que podemos consumir).

Resolveremos essas preocupações com relação ao armazenamento em buffer na versão aprimorada de `str_cli` mostrada na Seção 6.7.

6.6 Função `shutdown`

A maneira normal de terminar uma conexão de rede é chamar a função `close`. Mas há duas limitações com `close` que podem ser evitadas com a função `shutdown`:

1. A função `close` decrementa a contagem de referências ao descritor e fecha o soquete somente se a contagem alcançar 0. Discutimos isso na Seção 4.8. Com `shutdown`, podemos iniciar a seqüência normal de término da conexão TCP (os quatro segmentos que iniciam com um FIN na Figura 2.5), independentemente da contagem de referências.
2. `close` termina ambas as direções da transferência de dados, leitura e gravação. Como uma conexão TCP é full-duplex, há momentos em que queremos informar à outra extremidade que concluímos o envio, mesmo que essa extremidade possa ter outros dados a serem enviados. Esse é o cenário que encontramos na seção anterior com a entrada em lote da nossa função `str_cli`. A Figura 6.12 mostra as chamadas de função típicas nesse cenário.

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

int shutdown(int sockfd, int howto);

                                                    Retorna: 0 se OK, –1 em erro
```

A ação da função depende do valor do argumento *howto.*

`SHUT_RD` A metade leitura da conexão é fechada. Nenhum outro dado pode ser recebido no soquete e quaisquer dados atualmente no buffer de recebimento de soquete são descartados. O processo não pode mais invocar nenhuma das funções `read` no soquete. Quaisquer dados recebidos depois dessa chamada para um soquete TCP são reconhecidos e então silenciosamente descartados.

Por default, tudo que for gravado em um soquete de roteamento (Capítulo 18) volta como uma possível entrada para todos os soquetes de roteamento no host. Alguns programas chamam `shutdown` com um segundo argumento de `SHUT_RD` para evitar a cópia de

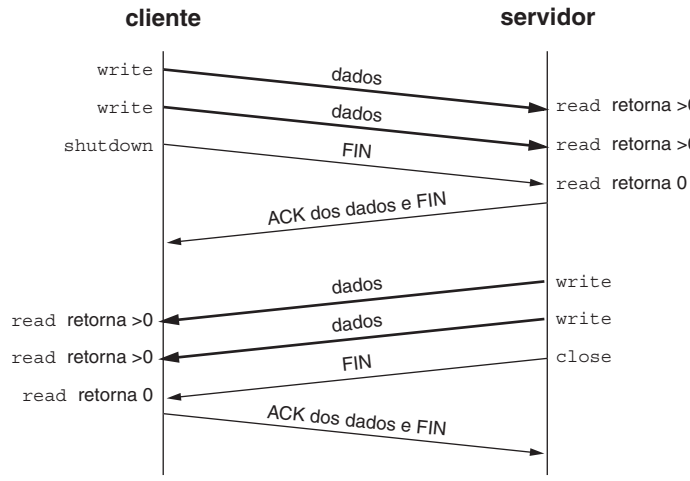


Figura 6.12 Chamando `shutdown` para fechar metade de uma conexão TCP.

loopback. Uma maneira alternativa de evitar essa cópia de loopback é limpar a opção de soquete SO_USELOOPBACK.

SHUT_WR A metade gravação da conexão é fechada. No caso do TCP, isso é chamado *half-close ("meio fechamento")* (Seção 18.5 do TCPv1). Quaisquer dados atualmente no buffer de envio do soquete serão enviados, seguidos por uma seqüência normal de término de conexão do TCP. Como mencionamos antes, esse fechamento da metade gravação é feito independentemente de a contagem de referências ao descritor de soquete ser atualmente maior que 0 ou não. O processo não pode mais emitir nenhuma das funções write no soquete.

SHUT_RDWR A metade leitura e a metade gravação da conexão são ambas fechadas. Isso é equivalente a chamar shutdown duas vezes: primeiro com SHUT_RD e então com SHUT_WR.

A Figura 7.12 resumirá as diferentes possibilidades disponíveis para o processo chamando shutdown e close. A operação de close depende do valor da opção do soquete SO_LINGER.

Os três nomes de SHUT_*xxx* são definidos pela especificação POSIX. Valores típicos para o argumento howto que você encontrará serão 0 (fecha a metade leitura), 1 (fecha a metade gravação) e 2 (fecha a metade leitura e a metade gravação).

6.7 Função str_cli (revisitada novamente)

A Figura 6.13 mostra nossa versão revisada (e corrigida) da função str_cli. Essa versão utiliza select e shutdown. A primeira notifica logo que o servidor fechar a extremidade da sua conexão e a última permite tratar a entrada em lote corretamente. Essa versão também acaba com código centrado na linha e, em vez disso, opera em buffers, eliminando a complexidade das preocupações levantadas na Seção 6.5.

select/strcliselect02.c

```
 1 #include   "unp.h"
 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     int maxfdp1, stdineof;
 6     fd_set rset;
 7     char buf[MAXLINE];
 8     int n;
 9     stdineof = 0;
10     FD_ZERO(&rset);
11     for ( ; ; ) {
12         if (stdineof == 0)
13             FD_SET(fileno(fp), &rset);
14         FD_SET(sockfd, &rset);
15         maxfdp1 = max(fileno(fp), sockfd) + 1;
16         Select(maxfdp1, &rset, NULL, NULL, NULL);
17         if (FD_ISSET(sockfd, &rset)) { /* o soquete é legível */
18             if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
19                 if (stdineof == 1)
20                     return; /* término normal */
21                 else
22                     err_quit("str_cli: server terminated prematurely");
23             }
24             Write(fileno(stdout), buf, n);
```

Figura 6.13 A função str_cli utilizando select que trata EOF corretamente (*continua*).

```
25          }
26          if (FD_ISSET(fileno(fp), &rset)) { /* a entrada é legível */
27              if ( (n = Read(fileno(fp), buf, MAXLINE)) == 0) {
28                  stdineof = 1;
29                  Shutdown(sockfd, SHUT_WR); /* envia FIN */
30                  FD_CLR(fileno(fp), &rset);
31                  continue;
32              }
33              Writen(sockfd, buf, n);
34          }
35      }
36  }
```

select/strcliselect02.c

Figura 6.13 A função `str_cli` utilizando `select` que trata EOF corretamente (*continuação*).

5-8 `stdineof` é um novo flag inicializado como 0. Contanto que esse flag seja 0, todas as vezes em torno do loop principal, selecionamos a entrada-padrão para legibilidade.

17-25 Quando lemos o EOF no soquete, se já tivermos encontrado um EOF na entrada-padrão, esse será o término normal, a função retorna. Mas, se ainda não tivermos encontrado um EOF na entrada-padrão, o processo servidor termina prematuramente. Agora, chamamos `read` e `write` para operar em buffers em vez de em linhas e permitir que `select` trabalhe por nós como o esperado.

26-34 Quando encontramos o EOF na entrada-padrão, nosso novo flag, `stdineof`, é ligado e chamamos `shutdown` com um segundo argumento `SHUT_WR` para enviar o FIN. Aqui, além disso, mudamos para operar em buffers em vez de em linhas, utilizando `read` e `writen`.

Não concluímos nossa função `str_cli`. Desenvolveremos uma versão com uma E/S não-bloqueadora na Seção 16.2 e uma versão com threads na Seção 26.3.

6.8 Servidor de eco TCP (revisitado)

Podemos revisitar nosso servidor de eco TCP nas Seções 5.2 e 5.3 e reescrevê-lo como um único processo que utiliza `select` para tratar qualquer número de clientes, em vez de bifurcar um filho por cliente. Antes de mostrar o código, vamos examinar as estruturas de dados que utilizaremos para monitorar os clientes. A Figura 6.14 mostra o estado do servidor antes de o primeiro cliente ter estabelecido uma conexão.

O servidor tem um único descritor ouvinte, que mostramos como um ponto na Figura 6.14.

O servidor mantém somente um conjunto de descritores de leitura, que mostramos na Figura 6.15. Supomos que o servidor é iniciado no primeiro plano, assim os descritores 0, 1 e 2 são configurados como entrada-padrão, saída e erro. Portanto, o primeiro descritor disponível para o soquete ouvinte é 3. Também mostramos um array de inteiros chamado *client* que contém o descritor do soquete conectado para cada cliente. Todos os elementos nesse array são inicializados com –1.

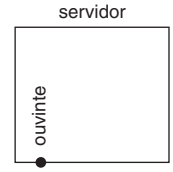


Figura 6.14 O servidor TCP antes de o primeiro cliente ter estabelecido uma conexão.

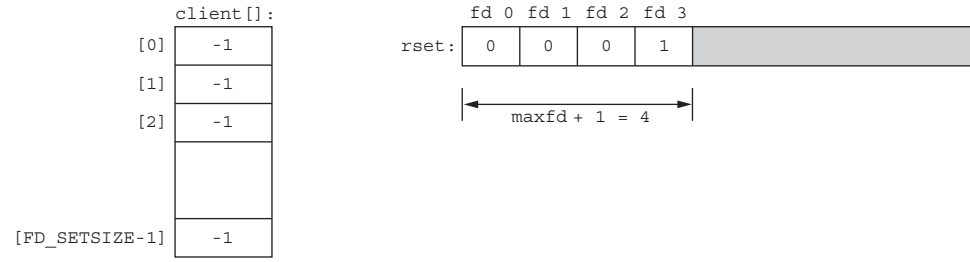


Figura 6.15 As estruturas de dados para o servidor TCP com somente um soquete ouvinte.

A única entrada não-zero no conjunto de descritores é a entrada para os soquetes ouvintes e o primeiro argumento para `select` será 4.

Quando o primeiro cliente estabelece uma conexão com o nosso servidor, o descritor ouvinte torna-se legível e o nosso servidor chama `accept`. O novo descritor conectado retornado por `accept` será 4, dadas as suposições desse exemplo. A Figura 6.16 mostra a conexão do cliente ao servidor.

Desse ponto em diante, nosso servidor deve lembrar do novo soquete conectado no seu array `client` e o soquete conectado deve ser adicionado ao conjunto de descritores. Essas estruturas de dados atualizadas são mostradas na Figura 6.17.

Posteriormente, um segundo cliente estabelece uma conexão e temos o cenário mostrado na Figura 6.18.

O novo soquete conectado (que assumimos ser 5) deve ser lembrado, resultando nas estruturas de dados mostradas na Figura 6.19.

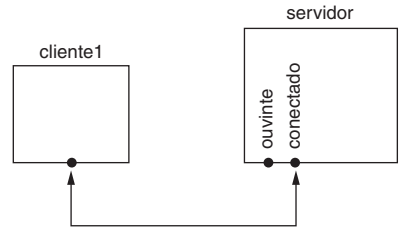


Figura 6.16 O servidor TCP depois que o primeiro cliente estabelece a conexão.

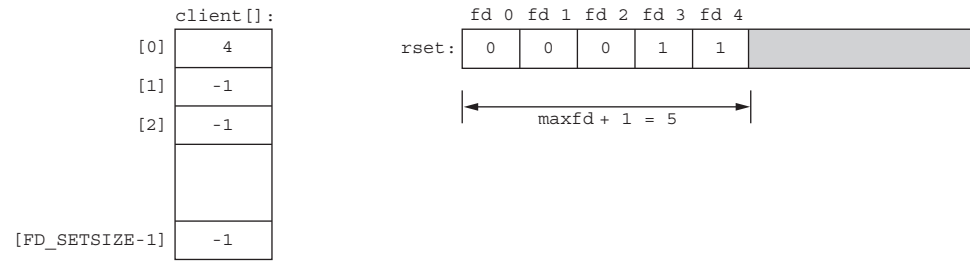


Figura 6.17 As estruturas de dados depois que a conexão do primeiro cliente é estabelecida.

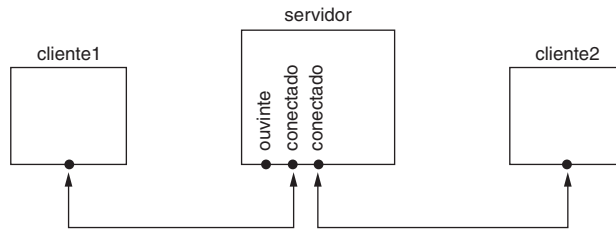


Figura 6.18 O servidor TCP depois que a conexão do segundo cliente é estabelecida.

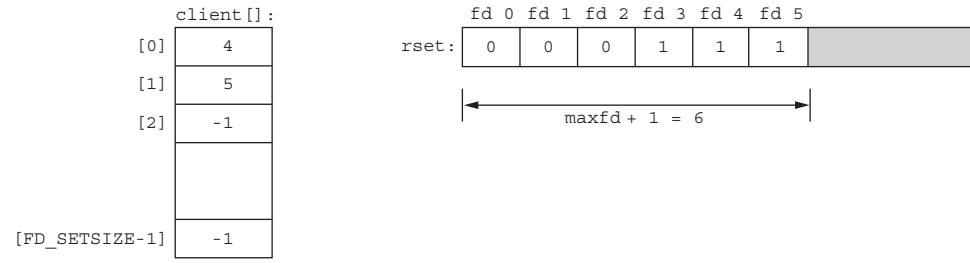


Figura 6.19 As estruturas de dados depois que a conexão do segundo cliente é estabelecida.

Em seguida, assumimos que o primeiro cliente termina sua conexão. O TCP cliente envia um FIN, o que faz o descritor 4 no servidor ficar legível. Quando nosso servidor lê esse soquete conectado, `read` retorna 0. Então, fechamos esse soquete e atualizamos nossas estruturas de dados de maneira correspondente. O valor de `client[0]` é configurado como –1 e o descritor 4 no conjunto de descritores é configurado como 0. Isso é mostrado na Figura 6.20. Observe que o valor de `maxfd` não muda.

Em resumo, à medida que os clientes chegam, registramos seus descritores de soquetes conectados na primeira entrada disponível no array `client` (isto é, a primeira entrada com um valor de –1). Também devemos adicionar o soquete conectado ao conjunto de descritores de leitura. A variável `maxi` é o índice mais alto no array `client` atualmente em utilização e a variável `maxfd` (mais um) é o valor atual do primeiro argumento para `select`. O único limite quanto ao número de clientes que esse servidor pode tratar é o mínimo dos dois valores `FD_SETSIZE` e o número máximo de descritores permitidos nesse processo pelo kernel (que discutiremos no final da Seção 6.3).

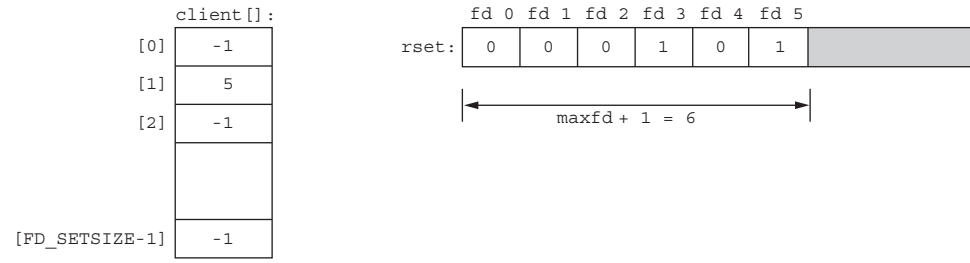


Figura 6.20 As estruturas de dados depois que o primeiro cliente termina sua conexão.

A Figura 6.21 mostra a primeira metade dessa versão do servidor.

tcpcliserv/tcpservselect01.c

```
 1 #include    "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     i, maxi, maxfd, listenfd, connfd, sockfd;
 6     int     nready, client[FD_SETSIZE];
 7     ssize_t n;
 8     fd_set  rset, allset;
 9     char    buf[MAXLINE];
10     socklen_t clilen;
11     struct sockaddr_in cliaddr, servaddr;
12     listenfd = Socket(AF_INET, SOCK_STREAM, 0);
13     bzero(&servaddr, sizeof(servaddr));
14     servaddr.sin_family = AF_INET;
15     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
16     servaddr.sin_port = htons(SERV_PORT);
17     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
18     Listen(listenfd, LISTENQ);
19     maxfd = listenfd;            /* inicializa */
20     maxi = -1;                   /* índice para o array client[] */
21     for (i = 0; i < FD_SETSIZE; i++)
22         client[i] = -1;          /* -1 indica entrada disponível */
23     FD_ZERO(&allset);
24     FD_SET(listenfd, &allset);
```

tcpcliserv/tcpservselect01.c

Figura 6.21 Servidor TCP utilizando um único processo e `select`: inicialização.

Criação de um soquete ouvinte e inicialização para `select`

12-24 Os passos para criar o soquete ouvinte são os mesmos vistos anteriormente: `socket`, `bind` e `listen`. Inicializamos nossas estruturas de dados assumindo que o único descritor que selecionaremos com `select` inicialmente é o soquete ouvinte.

A última metade da função é mostrada na Figura 6.22.

tcpcliserv/tcpservselect01.c

```
25     for ( ; ; ) {
26         rset = allset;       /* atribuição de estrutura */
27         nready = Select(maxfd + 1, &rset, NULL, NULL, NULL);
28         if (FD_ISSET(listenfd, &rset)) {     /* nova conexão de cliente */
29            clilen = sizeof(cliaddr);
30            connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
31            for (i = 0; i < FD_SETSIZE; i++)
32                if (client[i] < 0) {
33                   client[i] = connfd;   /* salva o descritor */
34                   break;
35               }
36            if (i == FD_SETSIZE)
37               err_quit("too many clients");
38            FD_SET(connfd, &allset);   /* adiciona novo descritor ao conjunto */
```

Figura 6.22 O servidor TCP utilizando um único processo e o loop `select` (*continua*).

```
39              if (connfd > maxfd)
40                  maxfd = connfd;       /* para select */
41              if (i > maxi)
42                  maxi = i;         /* índice máximo para o array client[] */
43              if (--nready <= 0)
44                  continue;             /* sem mais descritores legíveis */
45          }
46          for (i = 0; i <= maxi; i++) {/* verifica os clientes quanto a dados */
47              if ( (sockfd = client[i]) < 0)
48                  continue;
49              if (FD_ISSET(sockfd, &rset)) {
50                  if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
51                            /* conexão fechada pelo cliente */
52                      Close(sockfd);
53                      FD_CLR(sockfd, &allset);
54                      client[i] = -1;
55                  } else
56                      Writen(sockfd, buf, n);
57                  if (--nready <= 0)
58                      break;            /* sem mais descritores legíveis */
59              }
60          }
61      }
62  }
```

tcpcliserv/tcpservselect01.c

Figura 6.22 O servidor TCP utilizando um único processo e o loop `select` (*continuação*).

Bloqueio em `select`

26-27 `select` espera que algo aconteça: o estabelecimento de uma nova conexão de cliente ou a chegada de dados, um FIN, ou um RST em uma conexão existente.

Aceitação de novas conexões com `accept`

28-45 Se o soquete ouvinte está legível, uma nova conexão foi estabelecida. Chamamos `accept` e atualizamos nossas estruturas de dados de maneira correspondente. Utilizamos a primeira entrada não utilizada no array `client` para registrar o soquete conectado. O número de descritores prontos é diminuído e, se for 0, podemos evitar o próximo loop `for`. Isso permite utilizar o valor de retorno de `select` para evitar verificar os descritores que não estão prontos.

Verificação de conexões existentes

46-60 Um teste é feito para cada conexão de cliente existente quer seu descritor esteja ou não no conjunto de descritores retornado por `select`. Se estiver, uma linha é lida do cliente e ecoada de volta para o mesmo. Se o cliente fechar a conexão, `read` retorna 0 e atualizamos nossas estruturas de dados de maneira correspondente.

Nunca diminuimos o valor de `maxi`, mas poderíamos verificar essa possibilidade todas as vezes que um cliente fecha sua conexão.

Esse servidor é mais complicado do que aquele mostrado nas Figuras 5.2 e 5.3, mas ele evita todos os overheads da criação de um novo processo para cada cliente e é um bom exemplo de `select`. Contudo, na Seção 16.6, descreveremos um problema relacionado a esse servidor que é facilmente corrigido tornando o soquete ouvinte não-bloqueador e então verificando, e ignorando, alguns erros em `accept`.

Ataques de negação de serviço

Infelizmente, há um problema com relação ao servidor recém-mostrado. Considere o que acontece se um cliente malicioso conectar-se ao servidor, enviar um byte de dados (que não seja um caractere de nova linha) e então for dormir. O servidor chamará `read`, que irá ler o único byte de dados do cliente e então bloquear na próxima chamada a `read`, esperando mais dados desse cliente. O servidor é então bloqueado ("travado" pode ser um melhor termo) por esse cliente e não atenderá outros clientes (conexões novas de cliente ou dados de clientes existentes) até que o cliente malicioso envie um caractere de nova linha ou termine.

Aqui, o conceito básico é que, quando um servidor está tratando múltiplos clientes, ele *nunca* pode bloquear em uma chamada de função relacionada a um único cliente. Fazer isso pode travar o servidor e negar o serviço a todos os outros clientes. Isso é chamado de ataque de *negação de serviço*. Esse tipo de ataque faz algo ao servidor que impede que ele atenda outros clientes legítimos. Possíveis soluções são: (i) utilizar uma E/S não-bloqueadora (Capítulo 16), (ii) fazer com que cada cliente seja atendido por um thread separado de controle (por exemplo, gerar um processo ou um thread para atender cada cliente) ou (iii) impor um tempo-limite sobre as operações de E/S (Seção 14.2).

6.9 Função `pselect`

A função `pselect` foi criada pelo POSIX e agora é suportada por muitas variantes do Unix. `pselect` contém duas modificações em relação à função `select` normal:

```
#include <sys/select.h>
#include <signal.h>
#include <time.h>

int pselect(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
            const struct timespec *timeout, const sigset_t *sigmask) ;
```

Retorna: contagem de descritores prontos, 0 no tempo limite, –1 em erro

1. `pselect` utiliza a estrutura `timespec`, uma outra criação do POSIX, em vez da estrutura `timeval`.

   ```
   struct timespec {
     time_t tv_sec;      /* segundos */
     long tv_nsec;       /* nanossegundos */
   };
   ```

   A diferença entre essas duas estruturas está no segundo membro: o membro `tv_nsec` da estrutura mais recente especifica nanossegundos, enquanto o membro `tv_usec` da estrutura mais antiga especifica microssegundos.

2. `pselect` adiciona um sexto argumento: um ponteiro para uma máscara de sinal. Esse ponteiro permite que o programa desative a entrega de certos sinais, testa algumas variáveis globais configuradas pelos handlers para esses sinais, agora desativados, e então chama `pselect`, instruindo-a a reiniciar a máscara de sinal.

Quanto ao segundo ponto, considere o exemplo a seguir (discutido nas páginas 308 e 309 do APUE). O handler de sinal do nosso programa para `SIGINT` configura apenas o `intr_flag` global e retorna. Se nosso processo estiver bloqueado em uma chamada a `select`, o retorno do handler de sinal faz com que a função retorne com `errno` configurado como `EINTR`. Mas se `select` for chamada, o código se parece com isto:

```
if (intr_flag)
    handle_intr();       /* trata o sinal */
if ( (nready = select( ... )) < 0) {
   if (errno == EINTR) {
       if (intr_flag)
         handle_intr();
   }
   ...
}
```

O problema é que entre o teste de `intr_flag` e a chamada a `select`, se o sinal ocorrer, ele será perdido se `select` bloquear eternamente. Com `pselect`, agora podemos codificar esse exemplo de maneira mais confiável como

```
sigset_t newmask, oldmask, zeromask;

sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

sigprocmask(SIG_BLOCK, &newmask, &oldmask); /* bloqueia SIGINT */
if (intr_flag)
    handle_intr();    /* trata o sinal */
if ( (nready = pselect( ... , &zeromask)) < 0) {
    if (errno == EINTR) {
      if (intr_flag)
         handle_intr();
   }
   ...
}
```

Antes de testar a variável `intr_flag`, bloqueamos `SIGINT`. Quando `pselect` é chamada, ela substitui a máscara de sinal do processo por um conjunto vazio (isto é, `zeromask`) e então verifica os descritores, possivelmente indo dormir. Mas, quando `pselect` retorna, a máscara de sinal do processo é reinicializada com seu valor antes de `pselect` ter sido chamada (isto é, `SIGINT` é bloqueado).

Discutiremos `pselect` em mais detalhes e mostraremos um exemplo dela na Seção 20.5. Utilizaremos `pselect` na Figura 20.7 e mostraremos uma implementação simples, embora incorreta, de `pselect` na Figura 20.8.

> Há uma outra pequena diferença entre as duas funções `select`. O primeiro membro da estrutura `timeval` é um inteiro longo com sinal, enquanto o primeiro membro da estrutura `timespec` é uma `time_t`. O longo com sinal na primeira estrutura também deve ser um `time_t`, mas não foi alterado retroativamente para evitar a quebra do código existente. A novíssima função, porém, poderia fazer essa alteração.

6.10 Função `poll`

A função `poll`, originada com o SVR3, estava inicialmente limitada a dispositivos STREAMS (Capítulo 31). O SVR4 removeu essa limitação, permitindo à função `poll` funcionar com qualquer descritor. `poll` fornece uma funcionalidade semelhante a `select`, mas fornece informações adicionais ao lidar com dispositivos STREAMS.

```
#include <poll.h>

int poll(struct pollfd *fdarray, unsigned long nfds, int timeout);
```

Retorna: contagem de descritores prontos, 0 no tempo limite, –1 em erro

O primeiro argumento é um ponteiro para o primeiro elemento de um array de estruturas. Cada elemento do array é uma estrutura `pollfd` que especifica as condições a serem testadas para um dado descritor, `fd`.

```
struct pollfd {
   int   fd;           /* descritor a verificar */
   short events;       /* eventos de interesse em fd */
   short revents;      /* eventos que ocorreram em fd */
};
```

As condições a serem testadas são especificadas pelo membro `events` e a função retorna o *status* para esse descritor no membro `revents` correspondente. (Ter duas variáveis por descritor, uma um valor e outra um resultado, evita argumentos valor-resultado. Lembre-se de que os três argumentos intermediários para `select` são valor-resultado.) Cada um desses dois membros é composto de um ou mais bits que especificam uma certa condição. A Figura 6.23 mostra a constante utilizada para especificar o flag `events` e para testar o flag `revents` contra.

Dividimos essa figura em três seções: as quatro primeiras constantes lidam com entrada, as três seguintes lidam com saída e as três últimas lidam com erros. Observe que as últimas três não podem ser configuradas em `events`, mas sempre retornam em `revents` quando a condição correspondente existe.

Há três classes de dados identificadas por poll: *normal*, *banda de prioridade* e *alta prioridade*. Esses termos provêm das implementações baseadas em STREAMS (Figura 31.5).

> `POLLIN` poder ser definido como o OU lógico de `POLLRDNORM` e `POLLRDBAND`. A constante `POLLIN` é proveniente das implementações SVR3 anteriores às bandas de prioridade no SVR4, assim ela permanece por retrocompatibilidade. De maneira semelhante, `POLLOUT` é equivalente a `POLLWRNORM`, com a primeira sendo anterior à última.

Com referência aos soquetes TCP e UDP, as condições a seguir fazem com que `poll` retorne o *revent* especificado. Infelizmente, o POSIX deixa muitas brechas (isto é, maneiras opcionais de retornar a mesma condição) na sua definição de `poll`.

- Todos os dados TCP e UDP regulares são considerados normais.
- Dados fora da banda do TCP (Capítulo 24) são considerados de banda prioritária.
- Quando a metade leitura de uma conexão TCP é fechada (por exemplo, um FIN é recebido), isso também é considerado dado normal e uma operação de leitura subseqüente retornará 0.

Constante	Entrada para *events*?	Resultado de *revents*?	Descrição
`POLLIN`	•	•	Dados normais ou prioritários na banda podem ser lidos
`POLLRDNORM`	•	•	Dados normais podem ser lidos
`POLLRDBAND`	•	•	Dados prioritários na banda podem ser lidos
`POLLPRI`	•	•	Dados de alta prioridade podem ser lidos
`POLLOUT`	•	•	Dados normais podem ser gravados
`POLLWRNORM`	•	•	Dados normais podem ser gravados
`POLLWRBAND`	•	•	Dados prioritários na banda podem ser gravados
`POLLERR`		•	Ocorreu erro
`POLLHUP`		•	Ocorreu hangup
`POLLNVAL`		•	O descritor não é um arquivo aberto

Figura 6.23 *Events* de entrada e *revents* retornados para `poll`.

- A presença de um erro para uma conexão TCP pode ser considerada como dados normais ou um erro (POLLERR). Em qualquer caso, uma read subseqüente retornará –1 com errno configurado com o valor apropriado. Isso trata condições como a recepção de um RST ou um tempo-limite excedido.
- A disponibilidade de uma nova conexão em um soquete ouvinte pode ser considerada como dados normais ou prioritários. A maioria das implementações considera isso como dados normais.
- A conclusão de uma connect não-bloqueadora é considerada para tornar um soquete gravável.

O número de elementos no array de estruturas é especificado pelo argumento *nfds*.

> Historicamente, esse argumento tem sido um unsigned long, o que parece excessivo. Um unsigned int seria adequado. O Unix 98 define um novo tipo de dados para esse argumento: nfds_t.

O argumento *timeout* (tempo-limite) especifica quanto tempo a função deve esperar antes de retornar. Um valor positivo especifica o número de milissegundos a esperar. A Figura 6.24 mostra os possíveis valores do argumento *timeout*.

A constante INFTIM é definida como um valor negativo. Se o sistema não fornece um timer com exatidão de milissegundos, o valor é arredondado para o valor suportado mais próximo.

> A especificação POSIX requer que INFTIM seja definido incluindo <poll.h>, mas muitos sistemas ainda o definem em <sys/stropts.h>.
>
> Como ocorre com select, qualquer tempo-limite configurado para poll é limitado pela resolução de clock da implementação (freqüentemente 10 ms).

O valor de retorno de poll é –1 se ocorreu um erro, 0 se nenhum descritor estiver pronto antes de o timer expirar; caso contrário, ele é o número de descritores que tem um membro revents não-zero.

Se não houver interesse em um descritor particular, apenas configuramos o membro fd da estrutura pollfd como um valor negativo. Então o membro events é ignorado e o membro revents é configurado como 0 no retorno.

Lembre-se da nossa discussão no final da Seção 6.3 sobre FD_SETSIZE e sobre o número máximo de descritores por conjunto de descritores *versus* o número máximo de descritores por processo. Esse problema não ocorre com poll uma vez que é responsabilidade do chamador alocar um array de estruturas pollfd e então informar ao kernel o número de elementos no array. Não há nenhum tipo de dados de tamanho fixo semelhante a fd_set que o kernel conheça.

> A especificação POSIX requer tanto select como poll. Mas, atualmente, da perspectiva da portabilidade, mais sistemas suportam select do que poll. Além disso, o POSIX define pselect, uma versão aprimorada de select que trata do bloqueio de sinal e fornece uma resolução aumentada de data/hora. Nada semelhante é definido para poll.

valor de *timeout*	Descrição
INFTIM	Espera eternamente
0	Retorna imediatamente, não bloqueia
> 0	Espera o número especificado de milissegundos

Figura 6.24 Valores de *timeout* para poll.

6.11 Servidor de eco TCP (revisitado novamente)

Agora, refazemos nosso servidor de eco TCP na Seção 6.8 utilizando `poll` em vez de `select`. Na versão anterior com `select`, tínhamos de alocar um array `client` e um conjunto de descritores identificados como `rset` (Figura 6.15). Com `poll`, devemos alocar um array de estruturas `pollfd` para manter as informações do cliente em vez de alocar um outro array. Tratamos o membro `fd` desse array da mesma maneira como tratamos o array `client` na Figura 6.15: um valor de –1 significa que a entrada não está em utilização; caso contrário, ele é o valor do descritor. Lembre-se de que, na seção anterior, qualquer entrada no array de estruturas `pollfd` passadas para `poll` com um valor negativo para o membro `fd` é simplesmente ignorada.

A Figura 6.25 mostra a primeira metade do nosso servidor.

tcpcliserv/tcpservpoll01.c

```
 1 #include   "unp.h"
 2 #include   <limits.h>            /* para OPEN_MAX */

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int    i, maxi, listenfd, connfd, sockfd;
 7     int    nready;
 8     ssize_t n;
 9     char   buf[MAXLINE];
10     socklen_t clilen;
11     struct pollfd client[OPEN_MAX];
12     struct sockaddr_in cliaddr, servaddr;

13     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

14     bzero(&servaddr, sizeof(servaddr));
15     servaddr.sin_family = AF_INET;
16     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
17     servaddr.sin_port = htons(SERV_PORT);

18     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

19     Listen(listenfd, LISTENQ);

20     client[0].fd = listenfd;
21     client[0].events = POLLRDNORM;
22     for (i = 1; i < OPEN_MAX; i++)
23         client[i].fd = -1;      /* -1 indica entrada disponível */
24     maxi = 0;                   /* índice máximo para o array client[] */
```

tcpcliserv/tcpservpoll01.c

Figura 6.25 A primeira metade do servidor TCP utilizando `poll`.

Alocação do array de estruturas `pollfd`

11 Declaramos os elementos `OPEN_MAX` no nosso array de estruturas `pollfd`. É difícil determinar o número máximo de descritores que um processo pode manter aberto em um momento qualquer. Encontraremos esse problema novamente na Figura 13.4. Uma maneira é chamar a função `sysconf` do POSIX com um argumento de `SC_ OPEN_MAX` (como descrito nas páginas 42 a 44 de APUE) e então alocar dinamicamente um array do tamanho apropriado. Mas um dos possíveis retornos de `sysconf` é "indeterminado", o que significa que ainda temos de supor um valor. Aqui, apenas utilizamos a constante `OPEN_MAX` do POSIX.

Inicialização

20-24 Utilizamos a primeira entrada no array `client` para o soquete ouvinte e configuramos o descritor para as entradas restantes como –1. Também configuramos o evento `POLLRDNORM` para esse descritor para ser notificado por `poll` quando uma nova conexão está pronta para ser

aceita. A variável `maxi` contém o maior índice do array `client` atualmente em utilização.

A segunda metade da nossa função é mostrada na Figura 6.26.

tcpcliserv/tcpservpoll01.c

```
25    for ( ; ; ) {
26       nready = Poll(client, maxi + 1, INFTIM);

27       if (client[0].revents & POLLRDNORM) { /* nova conexão de cliente */
28          clilen = sizeof(cliaddr);
29          connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

30          for (i = 1; i < OPEN_MAX; i++)
31              if (client[i].fd < 0) {
32                client[i].fd = connfd;    /* salva o descritor */
33                break;
34             }
35          if (i == OPEN_MAX)
36             err_quit("too many clients");

37          client[i].events = POLLRDNORM;
38          if (i > maxi)
39             maxi = i;      /* índice máximo para o array client[] */

40          if (--nready <= 0)
41             continue;      /* sem mais descritores legíveis */
42       }

43       for (i = 1; i <= maxi; i++) { /* verifica os clientes quanto a dados */
44          if ( (sockfd = client[i].fd) < 0)
45             continue;
46          if (client[i].revents & (POLLRDNORM | POLLERR)) {
47             if ( (n = read(sockfd, buf, MAXLINE)) < 0) {
48                if (errno == ECONNRESET) {
49                      /* conexão redefinida pelo cliente */
50                   Close(sockfd);
51                   client[i].fd = -1;
52                } else
53                   err_sys("read error");
54             } else if (n == 0) {
55                   /* conexão fechada pelo cliente */
56                Close(sockfd);
57                client[i].fd = -1;
58             } else
59                Writen(sockfd, buf, n);

60             if (--nready <= 0)
61                break;       /* sem mais descritores legíveis */
62          }
63       }
64    }
65 }
```

tcpcliserv/tcpservpoll01.c

Figura 6.26 A segunda metade do servidor TCP utilizando `poll`.

Chamada a `poll`, verificação da nova conexão

26-42 Chamamos poll para esperar uma nova conexão ou os dados na conexão existente. Quando uma nova conexão é aceita, encontramos a primeira entrada disponível no array `client` procurando-a com um descritor negativo. Observe que iniciamos a pesquisa com o índice de 1, uma vez que `client[0]` é utilizado para o soquete ouvinte. Quando uma entrada disponível é encontrada, salvamos o descritor e configuramos o evento `POLLRDNORM`.

Verificação dos dados em uma conexão existente

43-63 Os dois eventos de retorno que verificamos são `POLLRDNORM` e `POLLERR`. Ainda não configuramos o segundo destes no membro `events` porque ele sempre retorna quando a condição é verdadeira. A razão pela qual verificamos `POLLERR` é porque algumas implementações retornam esse evento quando um RST é recebido para uma conexão, enquanto os outros apenas retornam `POLLRDNORM`. Em qualquer caso, chamamos `read` e, se ocorreu um erro, ela retornará um erro. Quando uma conexão existente é terminada pelo cliente, apenas configuramos o membro `fd` como –1.

6.12 Resumo

Há cinco diferentes modelos para E/S fornecidos pelo Unix:

- Bloqueadores
- Não-bloqueadores
- Multiplexação de E/S
- E/S baseada em sinal
- E/S assíncrona

O padrão é E/S bloqueadora, que também é a mais comumente utilizada. Neste capítulo, abordamos a multiplexação de E/S; nos próximos, discutiremos a E/S não-bloqueadora e a E/S baseada em sinais. A E/S assíncrona verdadeira é definida pela especificação POSIX, mas há poucas implementações.

A função mais comumente utilizada para multiplexação de E/S é `select`. Informamos à função `select` quais descritores nos interessam (para leitura, gravação e exceções), a quantidade máxima de tempo de espera e o número máximo de descritores (mais um). A maioria das chamadas a `select` especifica a legibilidade e observamos que a única condição de exceção ao lidar com soquetes é a chegada de dados fora da banda (Capítulo 24). Como `select` fornece um limite de tempo sobre quanto tempo uma função permanece bloqueada, utilizaremos esse recurso na Figura 14.3 para impor um limite de tempo em uma operação de entrada.

Utilizamos nosso cliente de eco em um modo de lote com `select` e descobrimos que, mesmo que o final da entrada de usuário seja encontrado, os dados ainda podem estar no pipe para ou do servidor. Tratar desse cenário requer a função `shutdown`, que nos permite tirar proveito do recurso half-close do TCP.

Os perigos de mesclar o armazenamento em buffer de stdio (bem como nosso próprio armazenamento em buffer de `readline`) com `select` fizeram com que produzíssemos versões do cliente e do servidor de eco que operavam em buffers em vez de em linhas.

O POSIX define a função `pselect`, que aumenta a precisão de data/hora de microssegundos para nanossegundos e recebe um novo argumento que é um ponteiro para um conjunto de sinais. Isso permite evitar condições de corrida (*race conditions*) quando os sinais estão sendo capturados, o que discutiremos em mais detalhes na Seção 20.5.

A função `poll` do System V fornece uma funcionalidade semelhante à de `select` e também informações adicionais sobre os dispositivos STREAMS. O POSIX requer tanto `select` como `poll`, mas a primeira é utilizada com mais freqüência.

Exercícios

6.1 Dissemos que um conjunto de descritores pode ser atribuído a outro conjunto de descritores por meio de um sinal de igual em C. Como isso é feito se um conjunto de descritores for um array de inteiros? *(Dica:* Examine o cabeçalho `<sys/select.h>` ou `<sys/types.h>` do seu sistema.)

6.2 Ao descrever as condições para as quais `select` retorna "writable" na Seção 6.3, por que precisamos do qualificador de que o soquete tinha de ser não-bloqueador para que uma operação de gravação retornasse um valor positivo?

6.3 O que aconteceria na Figura 6.9 se prefixássemos a palavra "`else`" antes de "`if`" na linha 19?

6.4 No nosso exemplo na Figura 6.21, adicione um código para permitir que o servidor seja capaz de utilizar tantos descritores quanto atualmente permitido pelo kernel. *(Dica:* Examine a função `setrlimit`.)

6.5 Vejamos o que acontece quando o segundo argumento para `shutdown` é `SHUT_RD`. Inicie com o cliente TCP na Figura 5.4 e faça as seguintes alterações: altere o número da porta de `SERV_PORT` para 19, o servidor `chargen` (Figura 2.18); então, substitua a chamada a `str_cli` por uma chamada à função `pause`. Execute esse programa especificando o endereço IP de um host local que execute o servidor `chargen`. Observe os pacotes com uma ferramenta como `tcpdump` (Seção C.5). O que acontece?

6.6 Por que uma aplicação chamaria `shutdown` com um argumento de `SHUT_RDWR` em vez de simplesmente chamar `close`?

6.7 O que acontece na Figura 6.22 quando o cliente envia um RST para terminar a conexão?

6.8 Recodifique a Figura 6.25 para chamar `sysconf` a fim de determinar o número máximo de descritores e alocar o array `client` de maneira correspondente.

Capítulo 7

Opções de Soquete

7.1 Visão geral

Há várias maneiras de obter e configurar as opções que afetam um soquete:

- As funções `getsockopt` e `setsockopt`
- A função `fcntl`
- A função `ioctl`

Este capítulo inicia com a discussão das funções `setsockopt` e `getsockopt`, seguida por um exemplo que imprime o valor default de todas as opções e então uma descrição detalhada de todas as opções de soquete. Dividimos as descrições detalhadas nas seguintes categorias: genéricas, IPv4, IPv6, TCP e SCTP. Essa discussão detalhada pode ser pulada em uma primeira leitura deste capítulo e em seções individuais relacionadas quando necessário. Algumas opções serão discutidas em detalhes em outro capítulo, como as opções de multicasting do IPv4 e do IPv6, que descreveremos com o multicasting na Seção 21.6.

Também descreveremos a função `fcntl`, porque é a maneira como o POSIX configura um soquete para uma E/S não-bloqueadora, uma E/S baseada em sinal e para configurar o proprietário de um soquete. Deixaremos a função `ioctl` para o Capítulo 17.

7.2 Funções `getsockopt` e `setsockopt`

Essas duas funções são aplicadas somente a soquetes.

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

int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t *optlen);

int setsockopt(int sockfd, int level, int optname, const void *optval,
               socklen_t optlen);
```

Ambas retornam: 0 se OK, –1 em erro

sockfd deve referenciar um descritor de soquete aberto. *level* especifica o código no sistema que interpreta a opção: o código de soquete geral ou algum código específico de protocolo (por exemplo, IPv4, IPv6, TCP ou SCTP).

optval é um ponteiro para uma variável a partir da qual o novo valor da opção é buscado por `setsockopt` ou no qual o valor atual da opção é armazenado por `getsockopt`. O tamanho dessa variável é especificado pelo argumento final, como um valor para `setsockopt` e como um valor-resultado para `getsockopt`.

As Figuras 7.1 e 7.2 resumem as opções que podem ser consultadas por `getsockopt` ou configuradas por `setsockopt`. A coluna "Tipo de dados" mostra o tipo de dados ao qual o ponteiro *optval* deve apontar para cada opção. Utilizamos a notação de duas chaves para indicar uma estrutura, como em `linger{}` com o significado de `struct linger`.

level	*optname*	`get`	`set`	Descrição	Flag	Tipo de dados
SOL_SOCKET	SO_BROADCAST	•	•	Permite envio de datagramas de broadcast	•	int
	SO_DEBUG	•	•	Ativa o monitoramento da depuração	•	int
	SO_DONTROUTE	•	•	Pula a pesquisa da tabela de roteamento	•	int
	SO_ERROR	•		Obtém erro pendente e limpa		int
	SO_KEEPALIVE	•	•	Testa periodicamente se a conexão ainda está ativa	•	int
	SO_LINGER	•	•	Espera o fechamento se houver dados a enviar		linger{}
	SO_OOBINLINE	•	•	Deixa em linha dados fora da banda recebidos	•	int
	SO_RCVBUF	•	•	Recebe tamanho do buffer		int
	SO_SNDBUF	•	•	Envia tamanho do buffer		int
	SO_RCVLOWAT	•	•	Recebe marca de menor nível do buffer		int
	SO_SNDLOWAT	•	•	Envia marca de menor nível do buffer		int
	SO_RCVTIMEO	•	•	Recebe tempo-limite		timeval{}
	SO_SNDTIMEO	•	•	Envia tempo-limite		timeval{}
	SO_REUSEADDR	•	•	Permite reutilizar endereço local	•	int
	SO_REUSEPORT	•	•	Permite reutilizar porta local	•	int
	SO_TYPE	•		Obtém tipo de soquete		int
	SO_USELOOPBACK	•	•	Soquete de roteamento obtém cópia do que ele envia	•	int
IPPROTO_IP	IP_HDRINCL	•	•	Cabeçalho IP incluído com dados	•	int
	IP_OPTIONS	•	•	Opções de cabeçalho IP		(ver texto)
	IP_RECVDSTADDR	•	•	Retorna o endereço IP de destino	•	int
	IP_RECVIF	•	•	Retorna o índice de interface recebida	•	int
	IP_TOS	•	•	Tipo de serviço e precedência		int
	IP_TTL	•	•	TTL		int
	IP_MULTICAST_IF	•	•	Especifica a interface de saída		in_addr{}
	IP_MULTICAST_TTL	•	•	Especifica o TTL de saída		u_char

Figura 7.1 Resumo das opções de soquete e de soquete da camada IP para `getsockopt` e `setsockopt` (*continua*).

level	*optname*	get	set	Descrição	Flag	Tipo de dados
	IP_MULTICAST_LOOP	•	•	Especifica o loopback		u_char
	IP_{ADD,DROP}_MEMBERSHIP		•	Ingressa em ou deixa grupo multicast		ip_mreq{}
	IP_{BLOCK,UNBLOCK}_SOURCE		•	Bloqueia ou desbloqueia origem de multicast		ip_mreq_source{}
	IP_{ADD,DROP}_SOURCE_MEMBERSHIP		•	Ingressa em ou deixa multicast específico da origem		ip_mreq_source{}
IPPROTO_ICMPV6	ICMP6_FILTER	•	•	Especifica que tipos de mensagem ICMPv6 podem passar		icmp6_filter{}
IPPROTO_IPV6	IPV6_CHECKSUM	•	•	Deslocamento do campo de soma de verificação para soquetes brutos		int
	IPV6_DONTFRAG	•	•	Descarta em vez de fragmentar pacotes grandes	•	int
	IPV6_NEXTHOP	•	•	Especifica endereço do próximo hop		sockaddr_in6{}
	IPV6_PATHMTU	•		Recupera MTU do caminho atual		ip6_mtuinfo{}
	IPV6_RECVDSTOPTS	•	•	Recebe opções de destino	•	int
	IPV6_RECVHOPLIMIT	•	•	Recebe limite do hop de unicast	•	int
	IPV6_RECVHOPOPTS	•	•	Recebe opções hop por hop	•	int
	IPV6_RECVPATHMTU	•	•	Recebe MTU do caminho	•	int
	IPV6_RECVPKTINFO	•	•	Recebe informações de pacote	•	int
	IPV6_RECVRTHDR	•	•	Recebe rota de origem	•	int
	IPV6_RECVTCLASS	•	•	Recebe classe de tráfego	•	int
	IPV6_UNICAST_HOPS	•	•	Limite de hop de unicast default		int
	IPV6_USE_MIN_MTU	•	•	Utiliza MTU mínimo	•	int
	IPV6_V6ONLY	•	•	Desativa compatibilidade com v4	•	int
	IPV6_*XXX*	•	•	Dados auxiliares persistentes		(ver texto)
	IPV6_MULTICAST_IF	•	•	Especifica a interface de saída		u_int
	IPV6_MULTICAST_HOPS	•	•	Especifica limite de hop de saída		int
	IPV6_MULTICAST_LOOP	•	•	Especifica o loopback	•	u_int
	IPV6_JOIN_GROUP		•	Ingressa em grupo de multicast		ipv6_mreq{}
	IPV6_LEAVE_GROUP		•	Deixa grupo multicast		ipv6_mreq{}
IPPROTO_IP or	MCAST_JOIN_GROUP		•	Ingressa em grupo de multicast		group_req{}
IPPROTO_IPV6	MCAST_LEAVE_GROUP		•	Deixa grupo multicast		group_source_req{}
	MCAST_BLOCK_SOURCE		•	Bloqueia origem de multicast		group_source_req{}
	MCAST_UNBLOCK_SOURCE		•	Desbloqueia origem de multicast		group_source_req{}
	MCAST_JOIN_SOURCE_GROUP		•	Ingressa em multicast de origem específica		group_source_req{}
	MCAST_LEAVE_SOURCE_GROUP		•	Deixa multicast de origem específica		group_source_req{}

Figura 7.1 Resumo das opções de soquete e das opções de soquete da camada IP para `getsockopt` e `setsockopt` (*continuação*).

level	*optname*	get	set	Descrição	Flag	Tipo de dados
IPPROTO_TCP	TCP_MAXSEG	•	•	Tamanho máximo de segmento TCP		int
	TCP_NODELAY	•	•	Desativa algoritmo de Nagle	•	int
IPPROTO_SCTP	SCTP_ADAPTION_LAYER	•	•	Indicação da camada de adaptação		sctp_setadaption{}
	SCTP_ASSOCINFO	†	•	Examina e configura informações de associação		sctp_assocparams{}
	SCTP_AUTOCLOSE	•	•	Operação de autofechamento		int
	SCTP_DEFAULT_SEND_PARAM	•	•	Parâmetros de envio default		sctp_sndrcvinfo{}
	SCTP_DISABLE_FRAGMENTS	•	•	Fragmentação de SCTP	•	int
	SCTP_EVENTS	•	•	Notificação de eventos de interesse		sctp_event_subscribe{}
	SCTP_GET_PEER_ADDR_INFO	†		Recupera *status* de endereço do peer		sctp_paddrinfo{}
	SCTP_I_WANT_MAPPED_V4_ADDR	•	•	Endereços v4 mapeados	•	int
	SCTP_INITMSG	•	•	Parâmetros INIT default		sctp_initmsg{}
	SCTP_MAXBURST	•	•	Tamanho máximo de rajada		int
	SCTP_MAXSEG	•	•	Tamanho máximo de fragmentação		int
	SCTP_NODELAY	•	•	Desativa algoritmo de Nagle	•	int
	SCTP_PEER_ADDR_PARAMS	†	•	Parâmetros de endereço do peer		sctp_paddrparams{}
	SCTP_PRIMARY_ADDR	†	•	Endereço primário de destino		sctp_setprim{}
	SCTP_RTOINFO	†	•	Informações de RTO		sctp_rtoinfo{}
	SCTP_SET_PEER_PRIMARY_ADDR		•	Endereço de destino primário do peer		sctp_setpeerprim{}
	SCTP_STATUS	†		Obtém *status* de associação		sctp_status{}

Figura 7.2 Resumo das opções de soquete da camada de transporte.

Há dois tipos básicos de opções: opções binárias que ativam ou desativam um certo recurso (flags) e opções que buscam e retornam valores específicos que podemos configurar ou examinar (valores). A coluna rotulada "Flag" especifica se a opção é de flag. Ao chamar `getsockopt` para essas opções de flag, **optval* é um inteiro. O valor retornado em **optval* será zero se a opção estiver desativada ou não-zero se estiver ativada. De maneira semelhante, `setsockopt` requer um **optval* valor não-zero para ativar a opção e um valor zero para desativá-la. Se a coluna de "Flag" não contiver um "•", a opção é então utilizada para passar um valor do tipo de dados especificado entre o processo usuário e o sistema.

As seções subseqüentes deste capítulo fornecerão detalhes adicionais sobre as opções que afetam um soquete.

7.3 Verificando se uma opção é suportada e obtendo o default

Agora, escreveremos um programa para verificar se a maioria das opções definidas nas Figuras 7.1 e 7.2 é suportada; se for, deve imprimir seus valores default. A Figura 7.3 contém as declarações do nosso programa.

Declaração de `union` de possíveis valores

3-8 Nossa união contém um membro para cada possível valor de retorno de `getsockopt`.

Definição dos protótipos de função

9-12 Definimos protótipos de função para quatro funções que são chamadas para imprimir o valor de uma dada opção de soquete.

Definição da estrutura e inicialização do array

13-52 Nossa estrutura `sock_opts` contém todas as informações necessárias para chamar `getsockopt` para cada opção de soquete e então imprimir seu valor atual. O membro final, `opt_val_str`, é um ponteiro para uma das nossas quatro funções que imprimirá o valor da opção. Alocamos e inicializamos um array para essas estruturas, um elemento para cada opção de soquete.

> Nem todas as implementações suportam todas as opções de soquete. A maneira de determinar se uma dada opção é suportada é utilizar um `#ifdef` ou um `#if defined`, como mostramos para `SO_REUSEPORT`. Para sermos completos, *todos* os elementos do array devem ser compilados de maneira semelhante àquilo que mostramos para `SO_REUSEPORT`, mas omitimos esses elementos porque os `#ifdef` somente alongam o código que mostramos e não acrescentam nada à discussão.

sockopt/checkopts.c

```
 1 #include   "unp.h"
 2 #include   <netinet/tcp.h>       /* para definições TCP_xxx*/

 3 union val {
 4     int                 i_val;
 5     long                l_val;
 6     struct linger       linger_val;
 7     struct timeval      timeval_val;
 8 } val;

 9 static char *sock_str_flag(union val *, int);
10 static char *sock_str_int(union val *, int);
11 static char *sock_str_linger(union val *, int);
12 static char *sock_str_timeval(union val *, int);

13 struct sock_opts {
14     const char    *opt_str;
15     int      opt_level;
16     int      opt_name;
17     char  *(*opt_val_str)(union val *, int);
18 } sock_opts[] = {
19     { "SO_BROADCAST",      SOL_SOCKET, SO_BROADCAST,    sock_str_flag },
20     { "SO_DEBUG",          SOL_SOCKET, SO_DEBUG,        sock_str_flag },
21     { "SO_DONTROUTE",      SOL_SOCKET, SO_DONTROUTE,    sock_str_flag },
22     { "SO_ERROR",          SOL_SOCKET, SO_ERROR,        sock_str_int },
23     { "SO_KEEPALIVE",      SOL_SOCKET, SO_KEEPALIVE,    sock_str_flag },
24     { "SO_LINGER",         SOL_SOCKET, SO_LINGER,       sock_str_linger },
25     { "SO_OOBINLINE",      SOL_SOCKET, SO_OOBINLINE,    sock_str_flag },
26     { "SO_RCVBUF",         SOL_SOCKET, SO_RCVBUF,       sock_str_int },
27     { "SO_SNDBUF",         SOL_SOCKET, SO_SNDBUF,       sock_str_int },
28     { "SO_RCVLOWAT",       SOL_SOCKET, SO_RCVLOWAT,     sock_str_int },
29     { "SO_SNDLOWAT",       SOL_SOCKET, SO_SNDLOWAT,     sock_str_int },
30     { "SO_RCVTIMEO",       SOL_SOCKET, SO_RCVTIMEO,     sock_str_timeval },
31     { "SO_SNDTIMEO",       SOL_SOCKET, SO_SNDTIMEO,     sock_str_timeval },
32     { "SO_REUSEADDR",      SOL_SOCKET, SO_REUSEADDR,    sock_str_flag },
33 #ifdef SO_REUSEPORT
34     { "SO_REUSEPORT",      SOL_SOCKET, SO_REUSEPORT,    sock_str_flag },
35 #else
36     { "SO_REUSEPORT",      0,          0,               NULL },
37 #endif
38     { "SO_TYPE",           SOL_SOCKET, SO_TYPE,         sock_str_int },
39     { "SO_USELOOPBACK",    SOL_SOCKET, SO_USELOOPBACK,  sock_str_flag },
40     { "IP_TOS",            IPPROTO_IP, IP_TOS,          sock_str_int },
41     { "IP_TTL",            IPPROTO_IP, IP_TTL,          sock_str_int },
```

Figura 7.3 Declarações do nosso programa para verificar as opções de soquete (*continua*).

```
42    { "IPV6_DONTFRAG",     IPPROTO_IPV6,IPV6_DONTFRAG, sock_str_flag },
43    { "IPV6_UNICAST_HOPS", IPPROTO_IPV6,IPV6_UNICAST_HOPS,sock_str_int },
44    { "IPV6_V6ONLY",       IPPROTO_IPV6,IPV6_V6ONLY,   sock_str_flag },
45    { "TCP_MAXSEG",        IPPROTO_TCP,TCP_MAXSEG",    sock_str_int },
46    { "TCP_NODELAY",       IPPROTO_TCP,TCP_NODELAY,    sock_str_flag },
47    { "SCTP_AUTOCLOSE",    IPPROTO_SCTP,SCTP_AUTOCLOSE,sock_str_int },
48    { "SCTP_MAXBURST",     IPPROTO_SCTP,SCTP_MAXBURST, sock_str_int },
49    { "SCTP_MAXSEG",       IPPROTO_SCTP,SCTP_MAXSEG,   sock_str_int },
50    { "SCTP_NODELAY",      IPPROTO_SCTP,SCTP_NODELAY,  sock_str_flag },
51    { NULL,                0,          0,              NULL }
52 };
```

sockopt/checkopts.c

Figura 7.3 Declarações do nosso programa para verificar as opções de soquete (*continuação*).

A Figura 7.4 mostra nossa função `main`.

sockopt/checkopts.c

```
53 int
54 main(int argc, char **argv)
55 {
56     int    fd;
57     socklen_t len;
58     struct sock_opts *ptr;

59     for (ptr = sock_opts; ptr->opt_str != NULL; ptr++) {
60          printf("%s: ", ptr->opt_str);
61          if (ptr->opt_val_str == NULL)
62            printf("(undefined)\n");
63          else {
64            switch (ptr->opt_level) {
65            case SOL_SOCKET:
66            case IPPROTO_IP:
67            case IPPROTO_TCP:
68                fd = Socket(AF_INET, SOCK_STREAM, 0);
69                break;
70 #ifdef IPV6
71            case IPPROTO_IPV6:
72                 fd = Socket(AF_INET6, SOCK_STREAM, 0);
73                 break;
74 #endif
75 #ifdef IPPROTO_SCTP
76            case IPPROTO_SCTP:
77                fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
78                break;
79 #endif
80            default:
81                err_quit("Can't create fd for level %d\n", ptr->opt_level);
82            }

83            len = sizeof(val);
84            if (getsockopt(fd, ptr->opt_level, ptr->opt_name,
85                           &val, &len) == -1) {
86                err_ret("getsockopt error");
87            } else {
88                printf("default = %s\n", (*ptr->opt_val_str)(&val, len));
89            }
90            close(fd);
91        }
92     }
93     exit(0);
94 }
```

sockopt/checkopts.c

Figura 7.4 Função `main` para verificar todas as opções de soquete.

Percorrendo todas as opções

59-63 Percorremos todos os elementos no nosso array. Se o ponteiro `opt_val_str` for nulo, a opção não é definida pela implementação (o que mostramos para `SO_REUSEPORT`).

Criação do soquete

63-82 Criamos um soquete no qual experimentamos a opção. Para experimentar um soquete, IPv4 e opções de soquete na camada de TCP, utilizamos um soquete TCP do IPv4. Para experimentar opções de soquete na camada de IPv6, utilizamos um soquete TCP do IPv6, e para experimentar as opções de soquete na camada de SCTP, utilizamos um soquete SCTP do IPv4.

Chamada a `getsockopt`

83-87 Chamamos `getsockop`, mas não terminamos se um erro retornar. Muitas implementações definem alguns nomes de opções de soquetes apesar de eles não suportarem a opção. Opções não suportadas devem evocar um erro de `ENOPROTOOPT`.

Impressão do valor default da opção

88-89 Se `getsockopt` retornar com sucesso, chamamos nossa função para converter o valor de opção em uma string e imprimi-la.

Na Figura 7.3, mostramos quatro protótipos de função, um para cada tipo de valor de opção que retorna. A Figura 7.5 mostra uma dessas quatro funções, `sock_str_flag`, que imprime o valor de uma opção de flag. As outras três funções são semelhantes.

sockopt/checkopts.c

```
 95 static char strres[128];

 96 static char *
 97 sock_str_flag(union val *ptr, int len)
 98 {
 99     if (len != sizeof(int))
100         snprintf(strres, sizeof(strres), "size (%d) not sizeof(int)", len);
101     else
102         snprintf(strres, sizeof(strres),
103                  "%s", (ptr->i_val == 0) ? "off" : "on");
104     return(strres);
105 }
```

sockopt/checkopts.c

Figura 7.5 Função `sock_str_flag`: converte a opção flag em uma string.

99-104 Lembre-se de que o argumento final para `getsockopt` é um argumento valor-resultado. A primeira verificação que fazemos é se o tamanho do valor retornado por `getsockopt` é o esperado. A string retornada é `off` ou `on`, dependendo se o valor da flag de opção for zero ou não-zero, respectivamente.

Executar esse programa no FreeBSD 4.8 com patches de KAME SCTP gera a saída a seguir:

```
freebsd % checkopts
SO_BROADCAST: default = off
SO_DEBUG: default = off
SO_DONTROUTE: default = off
SO_ERROR: default = 0
SO_KEEPALIVE: default = off
SO_LINGER: default = l_onoff = 0, l_linger = 0
SO_OOBINLINE: default = off
SO_RCVBUF: default = 57344
SO_SNDBUF: default = 32768
```

```
SO_RCVLOWAT: default = 1
SO_SNDLOWAT: default = 2048
SO_RCVTIMEO: default = 0 sec, 0 usec
SO_SNDTIMEO: default = 0 sec, 0 usec
SO_REUSEADDR: default = off
SO_REUSEPORT: default = off
SO_TYPE: default = 1
SO_USELOOPBACK: default = off
IP_TOS: default = 0
IP_TTL: default = 64
IPV6_DONTFRAG: default = off
IPV6_UNICAST_HOPS: default = -1
IPV6_V60NLY: default = off
TCP_MAXSEG: default = 512
TCP_NODELAY: default = off
SCTP_AUTOCLOSE: default = 0
SCTP_MAXBURST: default = 4
SCTP_MAXSEG: default = 1408
SCTP_NODELAY: default = off
```

O valor de 1 retornado para a opção `SO_TYPE` corresponde a `SOCK_STREAM` nessa implementação.

7.4 Estados do soquete

Algumas opções de soquete têm considerações de temporização sobre quando configurar ou buscar a opção *versus* o estado do soquete. Mencionamos essas considerações com as opções afetadas.

As opções de soquete a seguir são herdadas por um soquete TCP conectado a partir do soquete ouvinte (páginas 462 e 463 do TCPv2): `SO_DEBUG`, `SO_DONTROUTE`, `SO_KEEPALIVE`, `SO_LINGER`, `SO_OOBINLINE`, `SO_RCVBUF`, `SO_RCVLOWAT`, `SO_SNDBUF`, `SO_SNDLOWAT`, `TCP_MAXSEG` e `TCP_NODELAY`. Isso é importante com o TCP porque o soquete conectado não é retornado a um servidor por `accept` até que o handshake de três vias seja completado pela camada de TCP. Para assegurar que uma dessas opções de soquete seja configurada para o soquete conectado quando o handshake de três vias se completa, devemos configurá-la para o soquete ouvinte.

7.5 Opções de soquete genéricas

Iniciamos com uma discussão das opções de soquete genéricas. Essas opções são independentes de protocolo (isto é, são tratadas pelo código que é independente de protocolo dentro do kernel, não por um módulo particular de protocolo como o IPv4), mas algumas se aplicam apenas a certos tipos de soquetes. Por exemplo, apesar de a opção de soquete `SO_BROADCAST` ser denominada "genérica", ela se aplica somente a soquetes de datagrama.

Opção de soquete `SO_BROADCAST`

Essa opção ativa ou desativa a capacidade do processo de enviar mensagens de broadcast. O broadcasting é suportado somente em soquetes de datagrama e nas redes que suportam o conceito de mensagem de broadcast (por exemplo, Ethernet, token ring, etc.). Você não pode utilizar broadcast em um enlace ponto a ponto ou em qualquer protocolo de transporte baseado em conexão como SCTP ou TCP. Discutiremos o broadcasting em mais detalhes no Capítulo 20.

Uma vez que uma aplicação deve configurar essa opção de soquete antes de enviar um datagrama de broadcast, ela impede que um processo envie um broadcast se não foi projetada para isso. Por exemplo, uma aplicação UDP talvez receba o endereço IP de destino como um argumento de linha de comando, mas ela não tem por finalidade fazer com que um usuário digite um endereço de broadcast. Em vez de forçá-la a tentar determinar se um dado endereço é ou não de broadcast, o teste ocorre no kernel: se o endereço de destino for de broadcast e se essa opção de soquete não estiver ativada, `EACCES` é retornado (página 233 do TCPv2).

Opção de soquete `SO_DEBUG`

Essa opção é suportada somente pelo TCP. Quando ativada para um soquete TCP, o kernel monitora as informações detalhadas sobre todos os pacotes enviados ou recebidos pelo TCP para o soquete. Esses pacotes são mantidos em um buffer circular dentro do kernel que pode ser examinado com o programa `trpt`. As páginas 916 a 920 do TCPv2 fornecem detalhes adicionais e um exemplo que utiliza essa opção.

Opção de soquete `SO_DONTROUTE`

Essa opção especifica que os pacotes de saída devem pular os mecanismos normais de roteamento do protocolo subjacente. Por exemplo, com o IPv4 o pacote é direcionado à interface local apropriada, como especificado pelas partes rede e sub-rede do endereço de destino. Se a interface local não puder ser determinada a partir do endereço de destino (por exemplo, o destino não está na outra extremidade de um enlace ponto a ponto, ou não está em uma rede compartilhada), `ENETUNREACH` é retornado.

O equivalente dessa opção também pode ser aplicado a datagramas individuais utilizando o flag `MSG_DONTROUTE` com as funções `send`, `sendto` ou `sendmsg`.

Essa opção é freqüentemente utilizada por daemons de roteamento (por exemplo, `routed` e `gated`) para pular a tabela de roteamento e forçar um pacote a ser enviado a uma interface particular.

Opção de soquete `SO_ERROR`

Quando um erro ocorre em um soquete, o módulo de protocolo em um kernel derivado de Berkeley configura uma variável identificada `so_error` nesse soquete para um dos valores `E`*xxx*-padrão do Unix. Isso é chamado de *erro pendente* para o soquete. O processo pode ser imediatamente notificado sobre o erro de uma destas duas maneiras:

1. Se o processo estiver bloqueado em uma chamada para `select` no soquete (Seção 6.3), para legibilidade ou capacidade de ser gravável, `select` retorna com uma ou ambas as condições indicadas.
2. Se o processo utilizar uma E/S baseada em sinal (Capítulo 25), o sinal `SIGIO` é gerado para o processo ou grupo de processos.

O processo pode então obter o valor de `so_error` buscando a opção de soquete `SO_ERROR`. O valor do tipo inteiro retornado por `getsockopt` é o erro pendente para o soquete. O valor de `so_error` é então reinicializado para 0 pelo kernel (página 547 do TCPv2).

Se `so_error` for não-zero quando o processo chama `read` e se não houver nenhum dado a retornar, `read` retorna –1 com `errno` configurado como o valor de `so_error` (página 516 do TCPv2). O valor de `so_error` é então reinicializado para 0. Se houver dados enfileirados para o soquete, esses dados retornam por `read` em vez da condição de erro. Se `so_er-`

ror for não-zero quando o processo chamar write, –1 retorna com errno configurado como o valor de so_error (página 495 do TCPv2) e so_ error é reinicializado para 0.

> Há um bug no código mostrado na página 495 do TCPv2, pois so_error não é reinicializado para 0. Isso foi corrigido na maioria das distribuições modernas. Sempre que o erro pendente para um soquete é retornado, ele deve ser reinicializado para 0.

Essa é a primeira opção de soquete que encontramos que pode ser buscada, mas ela não pode ser configurada.

Opção de soquete SO_KEEPALIVE

Se a opção keep-alive é ativada para um soquete TCP e nenhum dado é trocado através do soquete em uma direção qualquer por duas horas, o TCP envia automaticamente uma keep-alive para investigar o peer. Essa investigação é um segmento de TCP ao qual o peer deve responder. Um dos três cenários resulta em:

1. O peer responde com o ACK esperado. A aplicação não é notificada (desde que tudo esteja ok). O TCP enviará outra investigação depois de mais duas horas de inatividade.
2. O peer responde com um RST, o que diz ao TCP local que o host peer travou e reinicializou. O erro pendente do soquete é configurado como ECONNRESET e o soquete é fechado.
3. Não há nenhuma resposta do peer à investigação keep-alive. TCPs derivados do Berkeley enviam 8 investigações adicionais, separadas por 75 segundos, tentando extrair uma resposta. O TCP desistirá se não houver nenhuma resposta no período de 11 minutos e 15 segundos depois de enviar a primeira investigação.

   > O HP-UX 11 trata a investigação keep-alive da mesma maneira como trataria dados, enviando a segunda investigação depois de um tempo-limite de retransmissão e dobrando o tempo-limite a cada pacote até o intervalo máximo configurado, com um default de 10 minutos.

   Se não houver absolutamente nenhuma resposta às investigações keep-alive do TCP, o erro pendente do soquete é configurado como ETIMEDOUT e o soquete é fechado. Mas, se o soquete receber um erro de ICMP em resposta a uma das investigações keep-alive, o erro correspondente (Figuras A.15 e A.16) retorna (e o soquete ainda é fechado). Um erro de ICMP comum nesse cenário é "host inacessível", indicando que o host do peer está inacessível; nesse caso, o erro pendente é configurado como EHOSTUNREACH. Isso pode ocorrer devido a uma falha de rede ou porque o host remoto parou e o roteador do último hop detectou a parada.

O Capítulo 23 do TCPv1 e as páginas 828 a 831 do TCPv2 contêm detalhes adicionais sobre a opção keep-alive.

Sem dúvida, a pergunta mais comum quanto a essa opção é se os parâmetros de timing podem ser modificados (normalmente, para reduzir o período de duas horas de inatividade a algum valor menor). O Apêndice E do TCPv1 discute como alterar esses parâmetros de timing para vários kernels, mas esteja ciente de que a maioria dos kernels mantém esses parâmetros individualmente por kernel, não individualmente por soquete, portanto, alterar o período de inatividade de 2 horas para 15 minutos, por exemplo, afetará todos os soquetes no host que ativam essa opção. Entretanto, essas perguntas normalmente são resultado de uma má compreensão do propósito dessa opção.

O propósito dessa opção é detectar se o host peer trava ou torna-se inacessível (por exemplo, quedas de conexão por modem em linha discada, falta de energia, etc.). Se o processo do peer travar, seu TCP enviará um FIN através da conexão, o que podemos detectar facilmente com select. (Essa é a razão por que utilizamos select na Seção 6.4.) Também note que,

se não houver nenhuma resposta a quaisquer investigações keep-alive (cenário 3), não há nenhuma garantia de que o host peer travou e de que o TCP pode terminar tranqüilamente uma conexão válida. Isso poderia ocorrer se algum roteador intermediário travasse por 15 minutos; e casualmente esse período de tempo supera completamente os 11 minutos e 15 segundos do período de investigação do nosso host. De fato, essa função talvez seja mais apropriadamente denominada "make-dead" ("torne morto") em vez de "keep-alive" ("mantenha vivo"), uma vez que pode terminar conexões vivas.

Essa opção normalmente é utilizada pelos servidores, embora os clientes também possam utilizá-la. Os servidores a utilizam porque passam a maior parte do tempo bloqueados esperando a entrada na conexão TCP, isto é, esperando uma solicitação de cliente. Mas, se a conexão do host cliente cair, sua energia for desligada ou a conexão travar, o processo de servidor nunca saberá disso e o servidor irá esperar continuamente uma entrada que pode nunca chegar. Isso é chamado de *conexão meio aberta* (*half open*). A opção keep-alive irá detectar e terminar essas conexões meio abertas.

Alguns servidores, notavelmente os FTP, fornecem um tempo-limite da aplicação, freqüentemente da ordem de minutos. Isso é feito pela própria aplicação, normalmente em torno de uma chamada para `read`, que lê o próximo comando do cliente. Esse tempo-limite (*timeout*) não envolve essa opção de soquete. Isso freqüentemente é um método mais apropriado de eliminar conexões para clientes ausentes, uma vez que a aplicação tem controle total se ela mesmo implementar o tempo-limite.

> O SCTP tem um mecanismo de *heartbeat* ("sinal de vida") semelhante ao mecanismo keep-alive do TCP. O mecanismo de heartbeat é controlado pelos parâmetros da opção de soquete `SCTP_SET_PEER_ADDR_PARAMS`, discutida mais adiante neste capítulo, em vez da opção de soquete `SO_KEEPALIVE`. As configurações feitas por `SO_KEEPALIVE` em um soquete SCTP são ignoradas e não afetam o mecanismo de heartbeat do SCTP.

A Figura 7.6 resume os vários métodos que precisamos detectar quando algo acontece na outra extremidade de uma conexão TCP. Quando dizemos "utilizar `select` para legibilidade", queremos dizer chamar `select` para testar se um soquete é legível.

Cenário	Processo do peer trava	Host peer trava	Host peer está inacessível
Nosso TCP está ativamente enviando dados	O TCP do peer envia um FIN, que podemos detectar imediatamente utilizando `select` para legibilidade. Se o TCP enviar um outro segmento, o TCP do peer responde com um RST. Se a aplicação tentar gravar no soquete depois que o TCP recebeu um RST, nossa implementação do soquete nos envia `SIGPIPE`.	O tempo-limite do nosso TCP irá expirar e o nosso erro pendente do soquete será configurado como `ETIMEDOUT`.	O tempo-limite do nosso TCP irá expirar e o erro pendente do nosso soquete será configurado como `EHOSTUNREACH`.
Nosso TCP está ativamente recebendo dados	O TCP do peer enviará um FIN, que interpretaremos como um EOF (possivelmente prematuro).	Iremos parar de receber dados.	Iremos parar de receber dados.
A conexão está inativa, keep-alive configurado	O TCP do peer envia um FIN, que podemos detectar imediatamente utilizando `select` para legibilidade.	Nove investigações keep-alive são enviadas depois de duas horas de inatividade e o erro pendente do nosso soquete é então configurado como `ETIMEDOUT`.	Nove investigações keep-alive são enviadas depois de duas horas de inatividade e o erro pendente do nosso soquete é então configurado como `EHOSTUNREACH`.
A conexão está inativa, keep-alive não está configurado	O TCP do peer envia um FIN, que podemos detectar imediatamente utilizando `select` para legibilidade.	(Nada)	(Nada)

Figura 7.6 Maneiras de detectar várias condições de TCP.

Opção de soquete SO_LINGER

Essa opção especifica como a função close opera em um protocolo orientado para conexão (por exemplo, no TCP e no SCTP, mas não no UDP). Por default, close retorna imediatamente, mas, se houver quaisquer dados remanescentes no buffer de envio do soquete, o sistema tentará entregar os dados para o peer.

A opção de soquete SO_LINGER permite alterar esse default. Essa opção requer que a estrutura a seguir seja passada entre o processo usuário e o kernel. Ela é definida incluindo <sys/socket.h>.

```
struct linger {
   int l_onoff;   /* 0=off, não-zero=on */
   int l_linger;  /* tempo de subsistência, o Posix especifica unidades
                     como segundos */
};
```

Chamar setsockopt leva a um dos três seguintes cenários, dependendo dos valores dos dois membros da estrutura:

1. Se l_onoff for zero, a opção está desativada. O valor de l_linger é ignorado e o default do TCP, discutido anteriormente, é aplicado: close retorna imediatamente.
2. Se l_onoff não for zero e l_linger for, o TCP aborta a conexão quando ela é fechada (páginas 1019 a 1020 do TCPv2). Isto é, o TCP descarta quaisquer dados que ainda permanecem no buffer de envio do soquete e envia um RST ao peer, não a seqüência normal de término da conexão de quatro pacotes (Seção 2.6). Mostraremos um exemplo disso na Figura 16.21. Isso evita o estado TIME_WAIT do TCP, mas, ao mesmo tempo, deixa aberta a possibilidade de uma outra "encarnação" dessa conexão ser criada dentro de 2MSL segundos (Seção 2.7) e faz com que segmentos duplicados antigos na conexão recém-terminada sejam incorretamente entregues para a nova encarnação.

   O SCTP também fará um fechamento fracassado do soquete enviando um bloco ABORT ao peer (consulte a Seção 9.2 de Stewart e Xie [2001]) se l_onoff for não-zero e l_linger for zero.

   > Postagens ocasionais na Usenet defendem o uso desse recurso somente para evitar o estado TIME_WAIT e para reiniciar um servidor ouvinte mesmo se as conexões ainda estiverem em utilização na porta bem-conhecida do servidor. Isso NÃO deve ser feito e pode levar a corrupção de dados, como detalhado na RFC 1337 (Braden, 1992). Em vez disso, a opção de soquete SO_REUSEADDR sempre deve ser utilizada no servidor antes da chamada a bind, como descreveremos em breve. O estado TIME_WAIT está aí para nos ajudar (isto é, deixar que segmentos duplicados antigos expirem na rede). Em vez de tentar evitar esse estado, devemos entendê-lo (Seção 2.7).
   >
   > Há certas circunstâncias que garantem a utilização desse recurso para enviar um fechamento abortivo. Um exemplo é um servidor de terminal R-232, que poderia travar eternamente no estado CLOSE_WAIT tentando entregar os dados para uma porta de terminal emperrada, mas redefiniria adequadamente essa porta emperrada se obtivesse um RST para descartar os dados pendentes.

3. Se l_onoff e l_linger forem não-zero, o kernel então irá esperar o momento em que o soquete é fechado (página 472 do TCPv2). Isto é, se ainda houver algum dado remanescente no buffer de envio do soquete, o processo é colocado para dormir até que: (i) todos os dados sejam enviados e reconhecidos pelo TCP do peer, ou (ii) o tempo de espera expire. Se o soquete foi configurado como não bloqueador (Capítulo 16), ele não irá esperar que o close se complete, mesmo que o tempo de espera seja não-zero. Ao utilizar esse recurso da opção SO_LINGER, é importante que a aplicação verifique o valor de retorno de close, porque se o tempo de espera expi-

rar antes que os dados remanescentes sejam enviados e reconhecidos, `close` retorna `EWOULDBLOCK` e quaisquer dados remanescentes no buffer de envio são descartados.

Agora, precisamos verificar exatamente quando um `close` em um soquete retorna de acordo com os vários cenários que examinamos. Supomos que o cliente grava os dados no soquete e então chama `close`. A Figura 7.7 mostra a situação default.

Supomos que, quando os dados do cliente chegam, o servidor está temporariamente ocupado, de modo que os dados são adicionados ao buffer de recebimento de soquete pelo seu TCP. De maneira semelhante, o próximo segmento, o FIN do cliente, também é adicionado ao buffer de recebimento de soquete (qualquer que seja a maneira como a implementação registra o fato de que um FIN foi recebido na conexão). Mas, por default, o `close` do cliente retorna imediatamente. Como mostramos nesse cenário, o `close` do cliente pode retornar antes de o servidor ler os dados remanescentes no seu buffer de recebimento de soquete. Portanto, é possível que o host servidor trave antes de a aplicação servidora ler esses dados remanescentes, e a aplicação cliente nunca saberá.

O cliente pode configurar a opção de soquete `SO_LINGER`, especificando algum tempo de espera positivo. Quando isso ocorre, o `close` do cliente não retorna até que todos os dados do cliente e seu FIN tenham sido reconhecidos pelo servidor TCP. Mostramos isso na Figura 7.8.

Mas continuamos com o mesmo problema da Figura 7.7: o host servidor pode travar antes de a aplicação servidora ler seus dados remanescentes e a aplicação cliente nunca saberá. Pior, a Figura 7.9 mostra o que pode acontecer quando a opção `SO_LINGER` é configurada como um valor muito baixo.

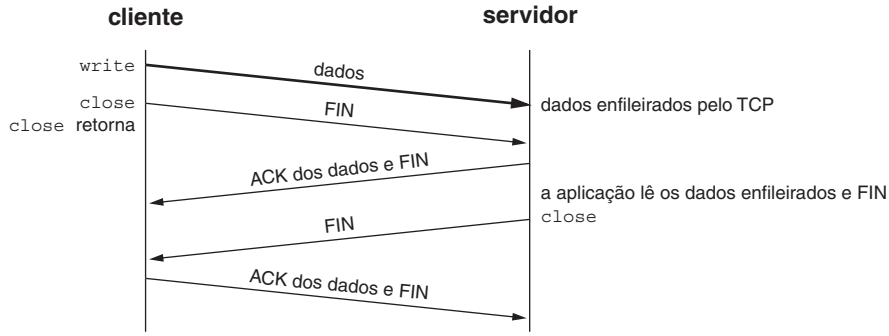


Figura 7.7 Operação default de `close`: retorna imediatamente.

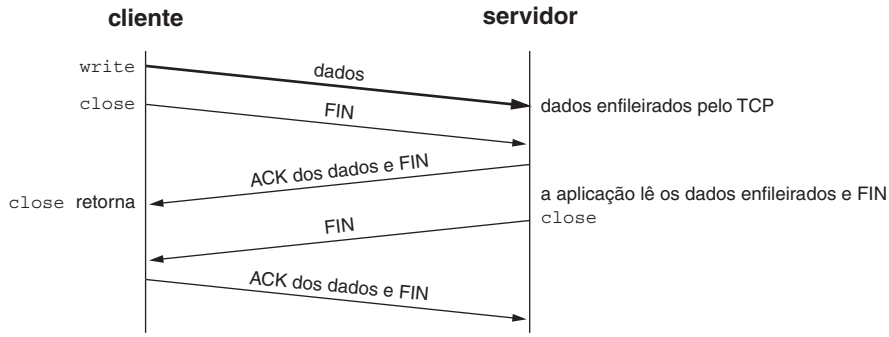


Figura 7.8 `close` com a opção de soquete `SO_LINGER` configurada e `l_linger` com um valor positivo.

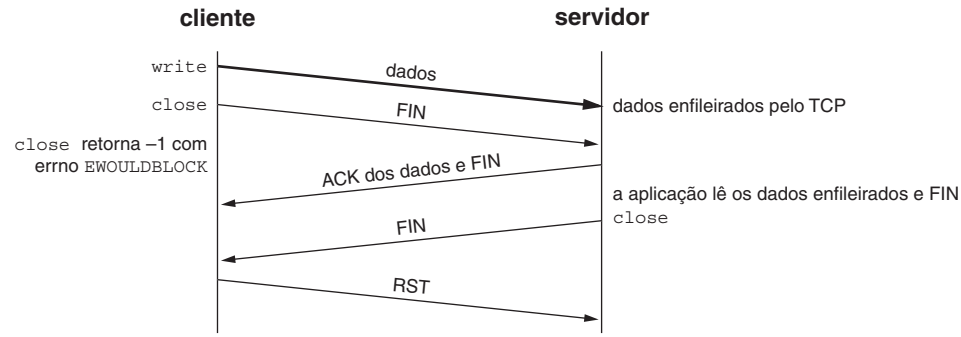


Figura 7.9 `close` com a opção de soquete `SO_LINGER` configurada e `l_linger` com um pequeno valor positivo.

Aqui, o princípio básico é que um retorno bem-sucedido de `close`, com a opção de soquete `SO_LINGER` ativada, informa apenas que os dados que enviamos (e o nosso FIN) foram reconhecidos pelo TCP do peer. Isso *não* informa se a aplicação do peer leu os dados. Se não ativamos a opção de soquete `SO_LINGER`, não saberemos se o TCP do peer reconheceu os dados.

Uma maneira de o cliente saber se o servidor leu seus dados é chamar `shutdown` (com o segundo argumento igual a `SHUT_WR`), em vez de chamar `close` e esperar que o peer também chame `close` para fechar a extremidade da sua conexão. Mostramos esse cenário na Figura 7.10.

Comparando essa figura às Figuras 7.7 e 7.8, vemos que, quando fechamos a extremidade da nossa conexão, dependendo da função chamada (`close` ou `shutdown`) e se a opção de soquete `SO_LINGER` estiver ativada, o retorno pode ocorrer em três momentos diferentes:

1. `close` retorna imediatamente, sem absolutamente esperar (o default; Figura 7.7).
2. `close` espera até que o ACK do nosso FIN tenha sido recebido (Figura 7.8).
3. `shutdown` seguido por um `read` espera até recebermos o FIN do peer (Figura 7.10).

Uma outra maneira de saber se a aplicação do peer leu nossos dados é utilizar um *reconhecimento no nível de aplicação* ou *ACK de aplicação*. Por exemplo, a seguir, o cliente envia seus dados ao servidor e então chama `read` para um dos bytes de dados:

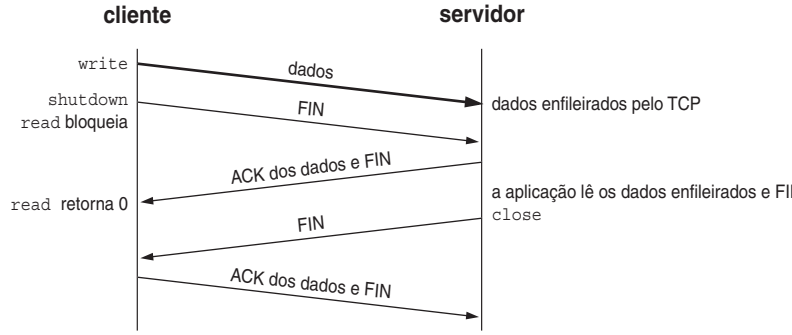


Figura 7.10 Utilizando `shutdown` para saber se esse peer recebeu nossos dados.

```
char ack;

Write(sockfd, data, nbytes);      /* dados do cliente para o servidor */
n = Read(sockfd, &ack, 1);        /* espera ACK no nível da aplicação */
```

O servidor lê os dados no cliente e então envia de volta o ACK de um byte gerado no nível de aplicação:

```
nbytes = Read(sockfd, buff, sizeof(buff));     /* dados do cliente */
         /* o servidor verifica se ele recebeu a quantidade
            correta de dados do cliente */
Write(sockfd, "" , 1);        /* ACK do servidor de volta para cliente */
```

Temos a garantia de que, quando `read` no cliente retornar, o processo servidor leu os dados que enviamos. (Isso supõe que ou o servidor sabe quantos dados o cliente está enviando ou há algum marcador de fim de registro definido pela aplicação, que não mostramos aqui.) Aqui, o ACK no nível de aplicação é um byte em 0, mas o conteúdo desse byte poderia ser utilizado para sinalizar outras condições do servidor ao cliente. A Figura 7.11 mostra a possível troca de pacotes.

A Figura 7.12 resume as duas possíveis chamadas para `shutdown` e as três possíveis chamadas para `close` e o efeito em um soquete TCP.

Opção de soquete `SO_OOBINLINE`

Quando essa opção está ativada, os dados fora da banda são colocados na fila normal de entrada (isto é, em linha). Quando isso ocorre, o flag `MSG_OOB` para as funções de recebimento não pode ser utilizado para ler os dados fora da banda. Discutiremos dados fora da banda em mais detalhes no Capítulo 24.

Opções de soquete `SO_RCVBUF` e `SO_SNDBUF`

Cada soquete tem um buffer de envio e um buffer de recebimento. Descrevemos a operação de buffers de envio com TCP, UDP e SCTP nas Figuras 2.15, 2.16 e 2.17.

Os buffers de recebimento são utilizados por TCP, UDP e SCTP para manter os dados recebidos até que eles sejam lidos pela aplicação. Com o TCP, o espaço disponível no buffer de recebimento do soquete limita a janela que o TCP pode anunciar para a outra extremidade. O

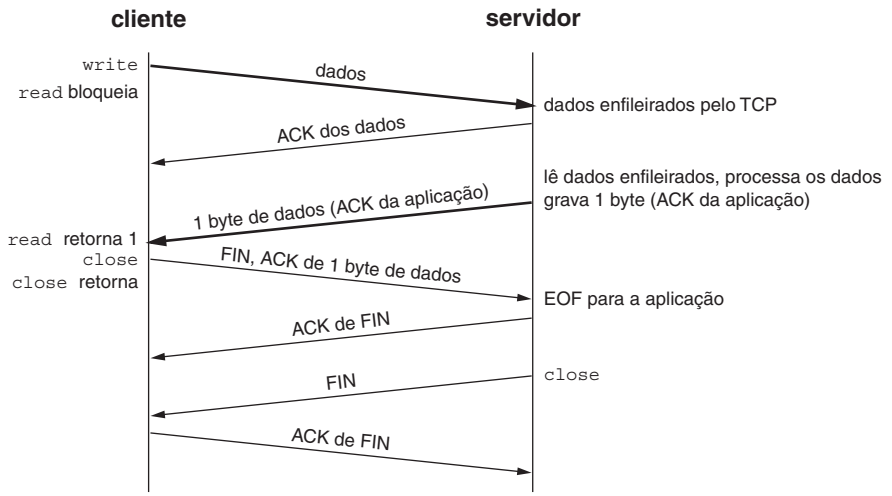


Figura 7.11 ACK da aplicação.

Função	Descrição
`shutdown, SHUT_RD`	Mais nenhum recebimento pode ser emitido no soquete; o processo ainda pode enviar no soquete; buffer de recebimento do soquete descartado; mais dados recebidos são descartados pelo TCP (Exercício 6.5); nenhum efeito sobre o buffer de envio do soquete.
`shutdown, SHUT_WR`	Mais nenhum envio pode ser emitido no soquete; o processo ainda pode receber no soquete; conteúdo do buffer de envio do soquete enviado para outra extremidade, seguido por terminação normal da conexão TCP (FIN); nenhum efeito sobre o buffer de recebimento do soquete.
`close, l_onoff = 0` (default)	Mais nenhum envio ou recebimento pode ser emitido no soquete; conteúdo do buffer de envio do soquete enviado para outra extremidade. Se contagem de referências ao descritor tornar-se 0: terminação normal de conexão TCP (FIN) enviada após os dados no buffer de envio e no buffer de recebimento do soquete serem descartados.
`close, l_onoff = 1` `l_linger = 0`	Mais nenhum recebimento ou envio pode ser emitido no soquete. Se contagem de referências ao descritor tornar-se 0: RST enviado para outra extremidade; estado de conexão configurado como CLOSED (estado TIME_WAIT); buffer de envio do soquete e buffer de recebimento do soquete descartados.
`close, l_onoff = 1` `l_linger != 0`	Mais nenhum envio ou recebimento pode ser emitido no soquete; conteúdo do buffer de envio do soquete enviado para outra extremidade. Se contagem de referências ao descritor tornar-se 0: terminação normal de conexão de TCP (FIN) enviada após os dados no buffer de envio; buffer de recebimento do soquete descartado; e se tempo de subsistência expirar antes de a conexão ser fechada, `close` retorna `EWOULDBLOCK`.

Figura 7.12 O resumo dos cenários `shutdown` e `SO_LINGER`.

buffer de recebimento do soquete de TCP não pode estourar porque o peer não tem permissão para enviar dados além do limite da janela anunciada. Esse é o controle de fluxo do TCP e, se o peer ignorar a janela anunciada e enviar dados além do limite desta, o TCP receptor irá descartá-los. Com o UDP, porém, quando um datagrama que não cabe no buffer de recebimento do soquete chega, é logo descartado. Lembre-se de que no UDP não há nenhum controle de fluxo: é fácil para um emissor rápido sobrecarregar um receptor mais lento, fazendo com que os datagramas sejam descartados pelo UDP do receptor, como mostraremos na Seção 8.13. De fato, um emissor rápido pode sobrecarregar sua própria interface de rede, fazendo com que datagramas sejam descartados pelo próprio emissor.

Essas duas opções de soquete permitem alterar os tamanhos default. Os valores default diferem amplamente entre implementações. Implementações derivadas do Berkeley mais antigas assumiriam um default para buffers de envio e de recebimento de 4.096 bytes para o TCP, mas sistemas mais recentes utilizam valores maiores, entre 8.192 e 61.440 bytes. O tamanho do buffer de envio do UDP freqüentemente assume um valor default em torno de 9.000 bytes se o host suportar NFS e o tamanho do buffer de recebimento do UDP freqüentemente assume um valor default em torno de 40.000 bytes.

Ao configurar o tamanho do buffer de recebimento do soquete do TCP, a ordem de chamadas de função é importante. Isso ocorre devido à opção de dimensionamento da janela de TCP (Seção 2.6), trocada com o peer nos segmentos SYN quando a conexão é estabelecida. Para um cliente, isso significa que a opção de soquete `SO_RCVBUF` deve ser configurada antes de chamar `connect`. Para um servidor, isso significa que a opção de soquete deve ser configurada para o soquete ouvinte antes de chamar `listen`. Configurar essa opção para o soquete conectado não terá qualquer efeito na possível opção de dimensionamento de janela porque `accept` não retorna com o soquete conectado até que o handshake de três vias do TCP seja completado. Essa é a razão pela qual essa opção deve ser configurada para o soquete ouvinte. (Os tamanhos dos buffers de soquete sempre são herdados do soquete ouvinte pelo soquete conectado recém-criado: páginas 462 e 463 do TCPv2.)

O tamanho dos buffers de soquete TCP devem ser pelo menos quatro vezes o MSS da conexão. Se estivermos lidando com transferência de dados unidirecional, como uma transferência de arquivos em uma direção, quando dizemos “tamanho dos buffers de soquete”, temos em mente o tamanho do buffer de envio do soquete no host emissor e o tamanho do buffer de

recebimento de soquete no host receptor. Para transferência de dados bidirecional, temos em mente ambos os tamanhos de buffer de soquete no emissor e ambos os tamanhos de buffer de soquete no receptor. Com tamanhos default de buffer típicos de 8.192 bytes ou maiores e um MSS típico de 512 ou 1.460, esse requisito normalmente é satisfeito.

> O mínimo MSS múltiplo de quatro é resultado da maneira como funciona o algoritmo de recuperação rápida do TCP. O emissor TCP utiliza três reconhecimentos duplicados para detectar o fato de que um pacote foi perdido (RFC 2581 [Allman, Paxson e Stevens, 1999]). O receptor envia um reconhecimento duplicado a cada segmento que ele recebe depois de um segmento perdido. Se o tamanho de janela for menor que quatro segmentos, não poderá haver três reconhecimentos duplicados, portanto, o algoritmo de recuperação rápida não pode ser invocado.

Para evitar desperdiçar espaço potencial em buffer, os tamanhos do buffer de soquete TCP também devem ser um múltiplo par do MSS para a conexão. Algumas implementações tratam esse detalhe para a aplicação, arredondando para cima o tamanho do buffer de soquete depois que a conexão é estabelecida (página 902 do TCPv2). Essa é uma outra razão para configurar as duas opções de soquete antes de estabelecer uma conexão. Por exemplo, utilizando o tamanho default do 4.4BSD de 8.192 e assumindo uma Ethernet com um MSS de 1.460, os dois buffers de soquete são arredondados para 8.760 (6 x 1.460) quando a conexão é estabelecida. Esse não é um requisito crucial; o espaço adicional no buffer de soquete acima do múltiplo de MSS simplesmente não é utilizado.

Uma outra consideração ao configurar os tamanhos do buffer de soquete tem a ver com o desempenho. A Figura 7.13 mostra uma conexão TCP entre duas extremidades (que chamamos de *pipe*) com uma capacidade de oito segmentos.

Mostramos quatro segmentos de dados na parte superior e quatro ACKs na parte inferior. Mesmo se houver somente quatro segmentos de dados no pipe, o cliente deverá ter um buffer de envio com capacidade de pelo menos oito segmentos, porque o TCP do cliente deve manter uma cópia de cada segmento até que o ACK seja recebido do servidor.

> Estamos ignorando alguns detalhes aqui. Primeiro, o algoritmo de início lento do TCP limita a taxa em que os segmentos são inicialmente enviados em uma conexão inativa. Em seguida, o TCP freqüentemente reconhece segmentos alternados, não todos os segmentos como mostramos. Todos esses detalhes são abordados nos Capítulos 20 e 24 do TCPv1.

O que é importante é entender o conceito do pipe full-duplex, sua capacidade e como isso se relaciona com os tamanhos de buffer de soquete nas duas extremidades da conexão. A capacidade do pipe é chamada de *produto de retardo de largura de banda,* que calculamos multiplicando a largura de banda (em bits/seg) pelo RTT (em segundos), convertendo o resultado de bits para bytes. O RTT é facilmente medido com o programa `ping`.

A largura de banda é o valor correspondente ao enlace mais lento entre as duas extremidades e deve ser de alguma maneira conhecida. Por exemplo, uma linha T1 (1.536.000 bits/seg) com um RTT de 60 ms dá a um produto de retardo de largura de banda 11.520 bytes. Se os tamanhos de buffer de soquete forem menores que isso, o pipe não permanecerá cheio e o desempenho será menor que o esperado. Grandes buffers de soquete são requeridos quando a largura de banda aumenta (por exemplo, linhas T3 a 45 Mbits/seg) ou quando o RTT torna-se grande (por exemplo, enlaces de satélite com um RTT em torno de 500 ms). Se o produto de retardo de largura de banda exceder o tamanho normal máximo da janela de TCP

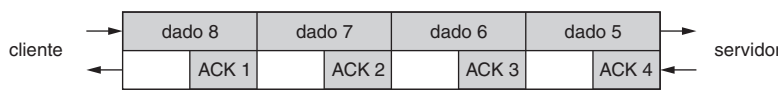


Figura 7.13 Conexão TCP (*pipe*) com uma capacidade de oito segmentos.

(65.535 bytes), as duas extremidades também precisam das opções de pipe "longo e gordo" que mencionamos na Seção 2.6.

> A maioria das implementações tem um limite superior para os tamanhos dos buffers de envio e recebimento de soquete, o qual, às vezes, pode ser modificado pelo administrador. Implementações derivadas do Berkeley mais antigas tinham um limite superior rígido de aproximadamente 52.000 bytes, mas implementações mais recentes têm um limite default de 256.000 bytes ou mais, e isso normalmente pode ser aumentado pelo administrador. Infelizmente, não há uma maneira simples de uma aplicação determinar esse limite. O POSIX define a função `fpathconf`, que a maioria das implementações suporta, e utilizando a constante `_PC_SOCK_MAXBUF` como o segundo argumento, podemos recuperar o tamanho máximo dos buffers de soquete. Alternativamente, uma aplicação pode tentar configurar os buffers de soquete com o valor desejado e, se isso falhar, cortar o valor pela metade e tentar novamente até que seja bem-sucedida. Por fim, uma aplicação deve se certificar de que não está na verdade tornando o buffer de soquete menor quando o configura com um valor "grande" pré-configurado; chamar primeiro `getsockopt` para recuperar o default do sistema e verificar se esse valor é suficientemente grande costuma ser um bom começo.

Opções de soquete `SO_RCVLOWAT` e `SO_SNDLOWAT`

Cada soquete também tem uma marca de menor nível de recebimento e uma de menor nível de envio. Essas marcas são utilizadas pela função `select`, como descrevemos na Seção 6.3. Estas duas opções de soquete, `SO_RCVLOWAT` e `SO_SNDLOWAT`, permitem alterar as duas marcas de nível.

A marca de menor nível de recebimento é a quantidade de dados que deve estar no buffer de recebimento do soquete para que `select` retorne "readable" (legível). Ela assume o default de 1 para soquetes TCP, UDP e SCTP. A marca de menor nível de envio é a quantidade de espaço disponível que deve existir no buffer de envio do soquete para que `select` retorne "writable" (gravável). Essa marca normalmente assume o default de 2.048 para soquetes TCP. Com o UDP, a marca de menor nível também é utilizada, como descrevemos na Seção 6.3, mas como o número de bytes do espaço disponível no buffer de envio para um soquete UDP nunca muda (visto que o UDP não mantém uma cópia dos datagramas enviados pela aplicação), contanto que o tamanho do buffer de envio do soquete UDP seja maior que o limite inferior do soquete, o soquete UDP sempre será gravável. Lembre-se, da Figura 2.16, de que o UDP não tem um buffer de envio; ele tem somente o tamanho de um buffer de envio.

Opções de soquete `SO_RCVTIMEO` e `SO_SNDTIMEO`

Essas duas opções de soquete permitem aplicar um tempo-limite (timeout) no envio e recebimento de soquetes. Observe que o argumento para as duas funções `sockopt` é um ponteiro para uma estrutura `timeval`, a mesma utilizada com `select` (Seção 6.3). Isso permite especificar os tempos-limite em segundos e microssegundos. Desativamos um tempo-limite configurando seu valor como 0 segundo e 0 microssegundo. Esses dois tempos-limite estão desativados por default.

O tempo-limite de recebimento afeta as cinco funções de entrada: `read`, `readv`, `recv`, `recvfrom` e `recvmsg`. O tempo-limite de envio afeta as cinco funções de saída: `write`, `writev`, `send`, `sendto` e `sendmsg`. Discutiremos os tempos-limite de soquete em mais detalhes na Seção 14.2.

> Essas duas opções de soquete e o conceito de tempos-limite inerentes ao envio e recebimento de soquetes foram adicionados com o 4.3BSD Reno.
>
> Nas implementações derivadas do Berkeley, esses dois valores implementam um timer de inatividade e não um timer absoluto na chamada de sistema de leitura ou gravação. As páginas 496 e 516 do TCPv2 discutem isso em mais detalhe.

Opções de soquete SO_REUSEADDR e SO_REUSEPORT

A opção de soquete SO_REUSEADDR serve a quatro propósitos diferentes:

1. SO_REUSEADDR permite a um servidor ouvinte iniciar e vincular (bind) sua porta bem-conhecida, mesmo se houver conexões previamente estabelecidas que utilizam essa porta como sua porta local. Essa condição tipicamente é encontrada como a seguir:
   a. Um servidor ouvinte é iniciado.
   b. Uma solicitação de conexão chega e um processo-filho é gerado para tratar esse cliente.
   c. O servidor ouvinte termina, mas o filho continua a servir o cliente na conexão existente.
   d. O servidor ouvinte é reiniciado.

   Por default, quando o servidor ouvinte é reiniciado em (d) chamando socket, bind e listen, a chamada a bind falha porque o servidor ouvinte está tentando vincular uma porta que faz parte de uma conexão existente (aquela tratada pelo filho gerado anteriormente). Mas, se o servidor ativar a opção de soquete SO_REUSEADDR entre as chamadas a socket e bind, a última função será bem-sucedida. *Todos* os servidores TCP devem especificar essa opção de soquete para permitir que o servidor seja reiniciado nessa situação.

   Esse cenário é uma das perguntas mais freqüentes na USENET.

2. SO_REUSEADDR permite que um novo servidor seja iniciado na mesma porta de um servidor existente que está vinculado ao endereço curinga, contanto que cada instância vincule um endereço IP local diferente. Isso é comum para um site que hospeda múltiplos servidores HTTP utilizando a técnica de alias de IP (Seção A.4). Suponha que o endereço IP primário do host local seja 198.69.10.2, mas ele tem dois aliases: 198.69.10.128 e 198.69.10.129. Três servidores HTTP são iniciados. O primeiro chamaria bind com o curinga como o endereço IP local e porta local 80 (a porta bem-conhecida para HTTP). O segundo chamaria bind com endereço IP local de 198.69.10.128 e porta local 80. Mas essa segunda chamada a bind falha a menos que SO_ REUSEADDR seja ativado antes da chamada. O terceiro servidor chamaria bind em 198.69.10.129 e porta 80. Novamente, SO_REUSEADDR é requerido para que essa chamada final seja bem-sucedida. Supondo que SO_REUSEADDR tenha sido ativado e os três servidores iniciados, as solicitações entrantes de conexão TCP com um endereço IP de destino de 198.69.10.128 e uma porta de destino 80 são entregues ao segundo servidor, solicitações entrantes com um endereço IP de destino de 198.69.10.129 e uma porta de destino 80 são entregues ao terceiro servidor e todas as outras solicitações de conexão TCP com uma porta de destino 80 são entregues ao primeiro servidor. Esse servidor "default" trata solicitações destinadas a 198.69.10.2, além de quaisquer outros aliases de IP que o host possa ter configurado. O curinga significa "tudo que não tem uma correspondência melhor (mais específica)". Observe que esse cenário de permitir múltiplos servidores a um dado serviço é tratado automaticamente se o servidor sempre ativar a opção de soquete SO_REUSEADDR (como recomendamos).

   Com o TCP, nunca somos capazes de iniciar múltiplos servidores que chamam bind no mesmo endereço IP e na mesma porta: uma *vinculação completamente duplicada*. Isto é, não podemos iniciar um servidor que vincula 198.69.10.2 à porta 80 e inicia um outro que também vincula 198.69.10.2 à porta 80, mesmo se ativamos a opção de soquete SO_REUSEADDR para o segundo servidor.

Por razões de segurança, alguns sistemas operacionais impedem qualquer vinculação "mais específica" a uma porta que já está vinculada ao endereço curinga, isto é, a série de vinculações descrita aqui não funcionaria com ou sem SO_REUSEADDR. Nesses sistemas, o servidor que realiza a vinculação curinga deve ser iniciado por último. Isso ocorre para evitar o problema de um servidor trapaceiro vincular-se a um endereço IP e a uma porta que já está sendo servida por um serviço de sistema e interceptar solicitações legítimas. Esse é um problema particular do NFS que, geralmente, não utiliza uma porta privilegiada.

3. SO_REUSEADDR permite que um único processo em múltiplos soquetes seja vinculado à mesma porta, contanto que cada vinculação especifique um endereço IP local diferente. Isso é comum para servidores UDP que precisam conhecer o endereço IP de destino das solicitações de cliente em sistemas que não fornecem a opção de soquete IP_RECVDSTADDR. Normalmente, essa técnica não é utilizada com servidores TCP, uma vez que um servidor TCP sempre pode determinar o endereço IP de destino chamando getsockname depois que a conexão é estabelecida. Entretanto, um servidor TCP que deseja atender conexões somente para alguns endereços pertencentes a um host multihomed deve utilizar essa técnica.

4. SO_REUSEADDR permite *vinculações completamente duplicadas*: uma vinculação de endereço e porta IP, quando esse mesmo endereço e porta IP já estão vinculados a outro soquete, se o protocolo de transporte suportá-lo. Normalmente, esse recurso é suportado somente por soquetes UDP.

   Esse recurso é utilizado com multicasting para permitir à mesma aplicação executar múltiplas vezes no mesmo host. Quando um datagrama UDP é recebido por um desses múltiplos soquetes vinculados, a regra é que, se o destino do datagrama for um endereço de broadcast ou de multicast, uma cópia do datagrama é entregue a cada soquete correspondente. Mas, se o destino do datagrama for um endereço de unicast, o datagrama é entregue somente a um soquete. Se, no caso de um datagrama de unicast, houver múltiplos soquetes que correspondem ao datagrama, a escolha de qual deles recebe o datagrama é dependente da implementação. As páginas 777 a 779 do TCPv2 discutem esse recurso em mais detalhes. Broadcasting e multicasting serão comentados em mais detalhes nos Capítulos 20 e 21.

Os Exercícios 7.5 e 7.6 mostram alguns exemplos dessa opção de soquete.

O 4.4BSD introduziu a opção de soquete SO_REUSEPORT quando o suporte a multicasting foi adicionado. Em vez de sobrecarregar SO_REUSEADDR com a semântica de multicast desejada que permite vinculações completamente duplicadas, essa nova opção de soquete foi introduzida com a seguinte semântica:

1. Essa opção permite vinculações completamente duplicadas, mas somente se cada soquete que está vinculado ao mesmo endereço e porta IP especificar essa opção de soquete.
2. SO_REUSEADDR é considerado equivalente a SO_REUSEPORT se o endereço IP sendo vinculado for um endereço multicast (página 731 do TCPv2).

O problema com essa opção de soquete é que nem todos os sistemas a suportam e, naqueles que não suportam a opção, mas suportam multicasting, SO_REUSEADDR é utilizada em vez de SO_REUSEPORT para permitir vinculações completamente duplicadas quando isso faz sentido (isto é, um servidor UDP que pode ser executado múltiplas vezes no mesmo host ao mesmo tempo e que espera receber datagramas de multicast ou de broadcast).

Podemos resumir nossa discussão sobre essas opções de soquete com as seguintes recomendações:

1. Ative a opção de soquete SO_REUSEADDR antes de chamar bind em todos os servidores TCP.
2. Ao escrever uma aplicação multicast que possa ser executada múltiplas vezes no mesmo host ao mesmo tempo, ative a opção de soquete SO_REUSEADDR e vincule o endereço multicast do grupo como o endereço IP local.

O Capítulo 22 do TCPv2 discute essas duas opções de soquete em mais detalhes.

Há um problema potencial de segurança com SO_REUSEADDR. Se houver um soquete que está vinculado ao, digamos, endereço curinga e à porta 5555, e se especificamos SO_REUSEADDR, podemos vincular essa mesma porta a um endereço IP diferente, por exemplo, o endereço IP primário do host. Quaisquer datagramas futuros que chegam destinados à porta 5555 e ao endereço IP que vinculamos ao nosso soquete são entregues ao nosso soquete, não ao outro soquete vinculado ao endereço curinga. Esses datagramas poderiam ser segmentos TCP SYN, blocos SCTP INIT ou datagramas de UDP. (O Exercício 11.9 mostra esse recurso com UDP.) Para a maioria dos serviços bem-conhecidos (por exemplo, HTTP, FTP e Telnet) isso não é um problema, porque todos esses servidores são vinculados a uma porta reservada. Conseqüentemente, qualquer processo que chega mais tarde e tenta vincular-se a uma instância mais específica dessa porta (isto é, roubar a porta) requer privilégios de superusuário. O NFS, porém, pode ser um problema, uma vez que a sua porta normal (2049) não é reservada.

> Um problema subjacente com a API de soquetes é que a configuração do par de soquetes é feita com duas chamadas de função (bind e connect) em vez de uma. (Torek, 1994) propõe uma única função para resolver esse problema.
>
> ```
> int bind_connect_listen(int sockfd,
>                         const struct sockaddr *laddr, int laddrlen,
>                         const struct sockaddr *faddr, int faddrlen,
>                         int listen) ;
> ```
>
> *laddr* especifica o endereço IP local e a porta local, *faddr* especifica o endereço IP externo e a porta externa e *listen* especifica um cliente (zero) ou um servidor (não-zero; o mesmo que o argumento backlog para listen). Assim, bind seria uma função de biblioteca que chama essa função com *faddr* como um ponteiro nulo e *faddrlen* 0, e connect seria uma função de biblioteca que chama essa função com *laddr* como um ponteiro nulo e *laddrlen* 0. Há algumas aplicações, notavelmente o TFTP, que precisam especificar tanto o par local como o par externo, as quais poderiam chamar bind_connect_listen diretamente. Com essa função, a necessidade de SO_REUSEADDR desaparece, exceto por servidores de multicast UDP que precisam explicitamente permitir vinculações duplicadas por completo do mesmo endereço e porta IP. Um outro benefício dessa nova função é que poderíamos restringir um servidor TCP a servir solicitações de conexão que chegam de um endereço e porta IP específicos, algo que a RFC 793 Postel (1981c) especifica, mas é impossível de implementar com a API de soquetes existente.

Opção de soquete SO_TYPE

Essa opção retorna o tipo de soquete. O valor do tipo inteiro retornado é um valor como SOCK_STREAM ou SOCK_DGRAM. Essa opção é, em geral, utilizada por um processo que herda um soquete quando é iniciado.

Opção de soquete SO_USELOOPBACK

Essa opção é aplicada somente a soquetes no domínio de roteamento (AF_ROUTE). Essa opção assume default de ON para esses soquetes (a única das opções de soquete SO_*xxx* que assume default de ON em vez de OFF). Quando essa opção está ativa, o soquete recebe uma cópia de tudo que é enviado por ele.

> Outra maneira de desativar essas cópias de loopback é chamar shutdown com o segundo argumento igual a SHUT_RD.

7.6 Opções de soquete IPv4

Essas opções de soquete são processadas pelo IPv4 e têm um nível (*level*) de `IPPROTO_IP`. Adiaremos a discussão sobre as opções de soquete de multicasting até a Seção 21.6.

Opção de soquete `IP_HDRINCL`

Se essa opção estiver ativada para um soquete IP bruto (Capítulo 28), deveremos criar nosso próprio cabeçalho IP para todos os datagramas que enviamos pelo soquete bruto. Normalmente, o kernel cria o cabeçalho IP para datagramas enviados por um soquete bruto, mas há algumas aplicações (notavelmente `traceroute`) que criam seus próprios cabeçalhos IP para sobrescrever os valores que o IP colocaria em certos campos de cabeçalho.

Se essa opção estiver ativada, criamos um cabeçalho IP completo, com as seguintes exceções:

- O IP sempre calcula e armazena a soma de verificação (checksum) do cabeçalho IP.
- Se configurarmos o campo de identificação de IP como 0, o kernel irá configurar o campo.
- Se o endereço IP de origem for `INADDR_ANY`, o IP o configura como o endereço IP primário da interface de saída.
- Configurar as opções de IP é dependente da implementação. Algumas implementações recebem quaisquer opções de IP que tenham sido configuradas com a opção de soquete `IP_OPTIONS` e acrescentam essas opções ao cabeçalho que criamos, enquanto outras requerem que o nosso cabeçalho também contenha quaisquer opções de IP desejadas.
- Alguns campos devem estar na ordem de bytes de host e outros na ordem de bytes de rede. Isso é dependente de implementação, o que torna a gravação de pacotes brutos com `IP_HDRINCL` não tão portável quanto gostaríamos.

Mostraremos um exemplo dessa opção na Seção 29.7. As páginas 1056 e 1057 do TCPv2 fornecem detalhes adicionais sobre essa opção de soquete.

Opção de soquete `IP_OPTIONS`

Especificar essa opção permite configurar as opções de IP no cabeçalho IPv4. Isso requer conhecimento íntimo do formato das opções IP no cabeçalho IP. Discutiremos essa opção com relação a rotas de origem do IPv4 na Seção 27.3.

Opção de soquete `IP_RECVDSTADDR`

Essa opção de soquete faz com que o endereço IP de destino de um datagrama UDP recebido retorne como dados auxiliares por `recvmsg`. Mostraremos um exemplo dessa opção na Seção 22.2.

Opção de soquete `IP_RECVIF`

Essa opção de soquete faz com que o índice da interface em que um datagrama UDP é recebido retorne como dados auxiliares por `recvmsg`. Mostraremos um exemplo dessa opção na Seção 22.2.

Opção de soquete `IP_TOS`

Essa opção permite configurar o campo do tipo de serviço (*type-of-service* – TOS) (que contém os campos DSCP e ECN, Figura A.1) no cabeçalho IP para um soquete TCP, UDP ou SCTP. Se chamarmos `getsockopt` para essa opção, o valor atual que seria colocado nos

campos DSCP e ECN no cabeçalho IP (o qual assume default 0) é retornado. Não há nenhuma maneira de buscar o valor a partir de um datagrama IP recebido.

Uma aplicação pode configurar o DSCP como um valor negociado com o provedor de serviço de rede para receber serviços pré-combinados, por exemplo, baixo retardo para telefonia de IP ou throughput mais alto para transferência de dados em grande volume. A arquitetura diffserv, definida na RFC 2474 (Nichols *et al*., 1998), oferece apenas retrocompatibilidade limitada com a definição histórica do campo TOS (da RFC 1349 [Almquist, 1992]). As aplicações que configuram `IP_TOS` para um dos conteúdos de `<netinet/ip.h>` (por exemplo, `IPTOS_LOWDELAY` ou `IPTOS_THROUGHPUT`) devem, em vez disso, utilizar um valor de DSCP especificado pelo usuário. Os únicos valores de TOS que o diffserv retém são os níveis 6 de precedência ("controle inter-redes") e 7 ("controle de rede"); isso significa que aplicações que configuram `IP_TOS` como `IPTOS_PREC_NETCONTROL` ou `IPTOS_PREC_INTERNETCONTROL` *irão* funcionar em uma rede diffserv.

A RFC 3168 (Ramakrishnan, Floyd e Black, 2001) contém a definição do campo ECN. Geralmente, as aplicações devem deixar a configuração do campo ECN para o kernel e especificar valores em zero nos dois bits inferiores dos valores configurados com `IPTOS`.

Opção de soquete `IP_TTL`

Com essa opção, podemos configurar e buscar o TTL default (Figura A.1) que o sistema utilizará para pacotes unicast enviados por um dado soquete. (O TTL de multicast é configurado com a opção de soquete `IP_MULTICAST_TTL`, descrita na Seção 21.6.) Por exemplo, o 4.4BSD utiliza o default de 64 tanto para soquetes TCP como UDP (especificados no registro da IANA "IP Option Numbers" [IANA]) e 255 para soquetes brutos. Como ocorre com o campo TOS, chamar `getsockopt` retorna o valor default do campo que o sistema utilizará nos datagramas de saída – não há como obter o valor a partir de um datagrama recebido. Iremos configurar essa opção de soquete com nosso programa `traceroute` na Figura 28.19.

7.7 Opção de soquete ICMPv6

Essa opção de soquete é processada pelo ICMPv6 e tem um *nível (level)* de `IPPROTO_ICMPV6`.

Opção de soquete `ICMP6_FILTER`

Essa opção permite buscar e configurar uma estrutura `icmp6_filter` que especifica qual dos 256 possíveis tipos de mensagem ICMPv6 serão passados para o processo em um soquete bruto. Discutiremos essa opção na Seção 28.4.

7.8 Opções de soquete IPv6

Essas opções de soquete são processadas pelo IPv6 e têm um *nível (level)* de `IPPROTO_IPV6`. Adiaremos a discussão sobre as opções de soquete de multicasting até a Seção 21.6. Observamos que muitas dessas opções utilizam *dados* auxiliares com a função `recvmsg` e vamos descrever isso na Seção 14.6. Todas as opções de soquete IPv6 estão definidas na RFC 3493 (Gilligan *et al*., 2003) e na RFC 3542 (Stevens *et al*., 2003).

Opção de soquete IPV6_CHECKSUM

Essa opção de soquete especifica o deslocamento em bytes nos dados do usuário em que o campo da soma de verificação (checksum) está localizado. Se esse valor é não-negativo, o kernel irá: (i) calcular e armazenar uma soma de verificação para todos os pacotes de saída e (ii) confirmará a soma de verificação recebida na entrada, descartando os pacotes com uma

soma de verificação inválida. Essa opção afeta todos os soquetes brutos do IPV6, exceto soquetes brutos do ICMPv6. (O kernel sempre calcula e armazena a soma de verificação para soquetes brutos do ICMPv6.) Se um valor de –1 for especificado (o default), o kernel não irá calcular e armazenar a soma de verificação para pacotes de saída nesse soquete bruto e não confirmará a soma de verificação dos pacotes recebidos.

> Todos os protocolos que utilizam o IPv6 devem ter uma soma de verificação no seu próprio cabeçalho de protocolo. Essas somas de verificação incluem um pseudocabeçalho (RFC 2460 [Deering e Hinden, 1998]) que inclui o endereço IPv6 de origem como parte da soma de verificação (que difere de todos os outros protocolos que são normalmente implementados utilizando um soquete bruto com o IPv4). Em vez de forçar a aplicação a utilizar o soquete bruto para realizar a seleção do endereço de origem, o kernel fará isso e então calculará e armazenará a soma de verificação incorporando o pseudocabeçalho IPv6-padrão.

Opção de soquete `IPV6_DONTFRAG`

Configurar essa opção desativa a inserção automática de um cabeçalho de fragmento para soquetes UDP e brutos. Se essa opção estiver configurada, pacotes de saída maiores que o MTU da interface de saída serão descartados. Nenhum erro precisa retornar da chamada de sistema que envia o pacote, uma vez que o pacote poderia exceder o MTU do caminho *en-route*. Em vez disso, a aplicação deve ativar a opção `IPV6_RECVPATHMTU` (Seção 22.9) para aprender sobre alterações do MTU do caminho.

Opção de soquete `IPV6_NEXTHOP`

Essa opção especifica endereço do próximo hop para o datagrama como uma estrutura de endereço de soquete e é uma operação privilegiada. Discutiremos esse recurso em mais detalhes na Seção 22.8.

Opção de soquete `IPV6_PATHMTU`

Essa opção não pode ser configurada, somente recuperada. Quando isso ocorre, o MTU atual, como determinado pela descoberta do MTU do caminho, é retornado (consulte a Seção 22.9).

Opção de soquete `IPV6_RECVDSTOPTS`

Configurar essa opção especifica que quaisquer opções de destino recebidas pelo IPv6 devem ser retornadas como dados auxiliares por `recvmsg`. Essa opção assume por default OFF. Descreveremos as funções utilizadas para construir e processar essas opções na Seção 27.5.

Opção de soquete `IPV6_RECVHOPLIMIT`

Configurar essa opção especifica que o campo de limite de hop recebido deve retornar como dado auxiliar por `recvmsg`. Essa opção assume o default de OFF. Descreveremos essa opção na Seção 22.8.

> Com o IPv4, não há nenhuma maneira de obter o campo TTL recebido.

Opção de soquete `IPV6_RECVHOPOPTS`

Configurar essa opção especifica que quaisquer opções de hop por hop IPv6 recebidas devem ser retornadas como dados auxiliares por `recvmsg`. Essa opção assume o default de OFF. Descreveremos as funções utilizadas para construir e processar essas opções na Seção 27.5.

Opção de soquete IPV6_RECVPATHMTU

Configurar essa opção especifica que o MTU de um caminho deve ser retornado como dado auxiliar por recvmsg (sem nenhum dado acompanhando) quando ele muda. Descreveremos essa opção na Seção 22.9.

Opção de soquete IPV6_RECVPKTINFO

Configurar essa opção especifica que as duas informações a seguir sobre um datagrama IPv6 recebido devem ser retornadas como dados auxiliares por recvmsg: o endereço IPv6 de destino e o índice da interface recebida. Descreveremos essa opção na Seção 22.8.

Opção de soquete IPV6_RECVRTHDR

Ativar essa opção especifica que um cabeçalho de roteamento IPv6 recebido deve ser retornado como dado auxiliar por recvmsg. Essa opção assume OFF por default. Descreveremos as funções utilizadas para construir e processar um cabeçalho de roteamento IPv6 na Seção 27.6.

Opção de soquete IPV6_RECVTCLASS

Ativar essa opção especifica que a classe de tráfego recebida (contendo os campos DSCP e ECN) deve ser retornada como dado auxiliar por recvmsg. Essa opção assume OFF por default. Descreveremos essa opção na Seção 22.8.

Opção de soquete IPV6_UNICAST_HOPS

Essa opção de IPv6 é semelhante à opção de soquete IP_TTL do IPv4. Ativar a opção de soquete especifica o limite de hops default para datagramas de saída enviados no soquete, ao passo que buscá-la retorna o valor para o limite de hops que o kernel utilizará para o soquete. O campo de limite de hops real de um datagrama IPv6 recebido é obtido com a opção de soquete IPV6_RECVHOPLIMIT. Iremos configurar essa opção de soquete com nosso programa traceroute na Figura 28.19.

Opção de soquete IPV6_USE_MIN_MTU

Configurar essa opção como 1 especifica que a descoberta do MTU do caminho não deve ser realizada e que os pacotes são enviados utilizando o MTU mínimo do IPv6 para evitar fragmentação. Configurar essa opção como 0 faz com que a descoberta do MTU do caminho ocorra para todos os destinos. Configurá-la como –1 especifica que a descoberta do MTU do caminho é realizada para destinos de unicast, mas o MTU mínimo é utilizado ao enviar a destinos de multicast. Essa opção assume –1 por default. Descreveremos essa opção na Seção 22.9.

Opção de soquete IPV6_V6ONLY

Ativar essa opção em um soquete AF_INET6 restringe-o à comunicação de IPv6 somente. Essa opção assume OFF por default, apesar de alguns sistemas terem uma opção para ativá-la (ON) por default. Descreveremos a comunicação IPv4 e IPv6 utilizando os soquetes AF_INET6 nas Seções 12.2 e 12.3.

Opções de soquete IPV6_*xxx*

A maioria das opções de IPv6 para modificação de cabeçalho assume um soquete UDP com as informações sendo passadas entre o kernel e a aplicação utilizando dados auxiliares com recvmsg e sendmsg. Um soquete TCP, em vez disso, busca e armazena esses valores utilizando getsockopt e setsockopt. A opção de soquete é a mesma que o tipo dos dados

auxiliares, e o buffer contém as mesmas informações que estariam presentes nos dados auxiliares. Descreveremos isso na Seção 27.7.

7.9 Opções de soquete TCP

Há duas opções de soquete para o TCP. Especificamos o *nível (level)* como `IPPROTO_TCP`.

Opção de soquete `TCP_MAXSEG`

Essa opção de soquete permite buscar ou configurar o MSS para uma conexão TCP. O valor retornado é a quantidade máxima de dados que nosso TCP enviará à outra extremidade; freqüentemente, esse valor é o MSS anunciado pela outra extremidade com seu SYN, a menos que nosso TCP escolha utilizar um valor menor que o MSS anunciado do peer. Se esse valor for buscado antes de o soquete estar conectado, o valor retornado é o valor default que será utilizado se uma opção MSS não for recebida a partir da outra extremidade. Também esteja ciente de que um valor menor que o valor retornado pode na verdade ser utilizado para a conexão se a opção de *timestamp* (registro de data/hora), por exemplo, estiver em utilização, porque essa opção ocupa 12 bytes das opções de TCP em cada segmento.

A quantidade máxima de dados que nosso TCP enviará por segmento também pode mudar durante a vida de uma conexão se o TCP suportar a descoberta do MTU do caminho. Se a rota para o peer mudar, esse valor poderá aumentar ou diminuir.

Observamos na Figura 7.1 que essa opção de soquete também pode ser configurada pela aplicação. Isso não é possível em todos os sistemas; originalmente, era uma opção apenas de leitura. O 4.4BSD limita a aplicação a *diminuir* o valor: não podemos aumentar o valor (página 1023 do TCPv2). Como essa opção controla a quantidade de dados que o TCP envia por segmento, faz sentido proibir que a aplicação aumente o valor. Depois que a conexão é estabelecida, esse valor é a opção de MSS anunciada pelo peer, não podemos excedê-lo. O TCP, porém, sempre pode enviar um valor menor que o MSS anunciado do peer.

Opção de soquete `TCP_NODELAY`

Se ativada, essa opção inibe o *algoritmo de Nagle* do TCP (Seção 19.4 do TCPv1 e páginas 858 e 859 do TCPv2). Por default, esse algoritmo está ativado.

O propósito do algoritmo de Nagle é reduzir o número de pequenos pacotes em uma rede geograficamente distribuída (WAN). O algoritmo estabelece que, se uma dada conexão tiver dados pendentes (isto é, dados que nosso TCP enviou e para os quais atualmente espera um reconhecimento), nenhum pacote pequeno será então enviado na conexão em resposta a uma operação de gravação do usuário até que os dados existentes tenham sido reconhecidos. A definição de pacote "pequeno" é qualquer pacote menor que o MSS. O TCP sempre enviará um pacote de tamanho completo se possível; o propósito do algoritmo de Nagle é impedir que uma conexão tenha múltiplos pacotes pequenos pendentes a qualquer momento.

Os dois geradores comuns de pacotes pequenos são os clientes Rlogin e Telnet, uma vez que ambos normalmente enviam cada pressionamento de tecla como um pacote separado. Em uma rede local rápida, é difícil perceber o algoritmo de Nagle com esses clientes, porque o tempo requerido para que um pequeno pacote seja reconhecido em geral é de alguns milissegundos – bem menor que o tempo entre dois caracteres sucessivos que digitamos. Mas em uma rede geograficamente distribuída (WAN), em que pode demorar um segundo para que um pequeno pacote seja reconhecido, percebemos um retardo no caractere que ecoa, e isso é freqüentemente exagerado pelo algoritmo de Nagle.

Considere o seguinte exemplo: digitamos a string "hello!" de seis caracteres para um cliente Rlogin ou Telnet, com exatamente 250 ms entre cada caractere. O RTT para o servidor é de 600 ms e o servidor imediatamente envia de volta o eco de cada caractere. Supomos que

o ACK do caractere do cliente é enviado de volta para o cliente junto com o eco do caractere e ignoramos os ACKs que o cliente envia ao eco do servidor. (Discutiremos em breve os ACKs retardados.) Supondo que o algoritmo de Nagle esteja desativado, temos os 12 pacotes mostrados na Figura 7.14.

Cada caractere é enviado em um pacote por si mesmo: os segmentos de dados da esquerda para a direita e o ACKs da direita para a esquerda.

Se o algoritmo de Nagle estiver ativado (o default), temos os oito pacotes mostrados na Figura 7.15. O primeiro caractere é enviado como um pacote, mas os dois seguintes não são enviados, uma vez que a conexão tem um pacote pequeno pendente. No tempo 600, quando o ACK do primeiro pacote é recebido, junto com o eco do primeiro caractere, esses dois caracteres são enviados. Até que esse pacote seja reconhecido com um ACK no tempo 1200, nenhum outro pacote pequeno é enviado.

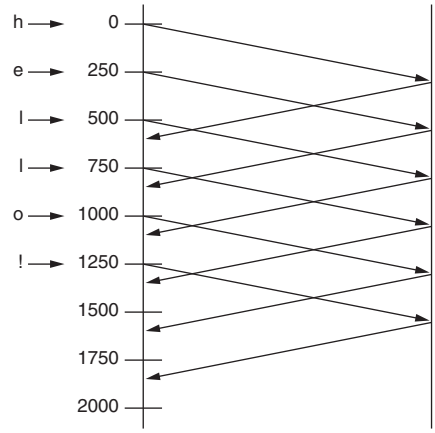


Figura 7.14 Seis caracteres ecoados pelo servidor com o algoritmo de Nagle desativado.

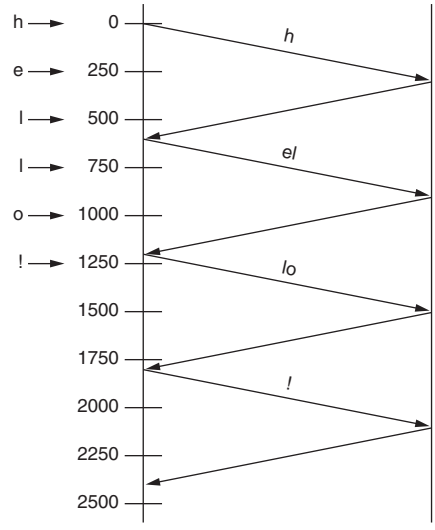


Figura 7.15 Seis caracteres ecoados pelo servidor com o algoritmo de Nagle ativado.

O algoritmo de Nagle freqüentemente interage com um outro algoritmo de TCP: o algoritmo *ACK retardado*. Esse algoritmo faz com que o TCP não envie um ACK logo após receber os dados; em vez disso, o TCP vai esperar um curto período de tempo (em geral, 50-200 ms) e somente então enviará o ACK. A esperança é que, nesse curto período de tempo, haverá dados para enviar de volta ao peer, e o ACK poderá "pegar carona" com eles, economizando um segmento TCP. Normalmente, esse é o caso com clientes Rlogin e Telnet, porque os servidores em geral ecoam cada caractere enviado pelo cliente, portanto, o ACK do caractere do cliente "pega carona" com o eco do servidor desse caractere.

O problema são os outros clientes cujos servidores não geram tráfego na direção inversa em que os ACKs podem pegar uma "carona". Esses clientes podem detectar retardos notáveis porque o TCP cliente não enviará nenhum dado ao servidor até que o timer do ACK adiado do servidor expire. Esses clientes precisam de uma maneira de desativar o algoritmo de Nagle, por isso a opção `TCP_NODELAY`.

Um outro tipo de cliente que interage mal com o algoritmo de Nagle e com ACKs retardados do TCP é aquele que envia uma única solicitação lógica dividida em pequenas partes ao seu servidor. Por exemplo, suponha que um cliente envie uma solicitação de 400 bytes ao seu servidor e isso seja um tipo de solicitação de 4 bytes seguido por 396 bytes de dados da solicitação. Se o cliente realizar um `write` de 4 bytes seguido por um `write` de 396 bytes, o segundo `write` não será enviado pelo TCP cliente até que o TCP servidor reconheça o `write` de 4 bytes. Além disso, como a aplicação servidora não pode operar nos 4 bytes de dados até receber os 396 bytes de dados remanescentes, o TCP servidor retardará o ACK dos 4 bytes de dados (isto é, não haverá nenhum dado do servidor para o cliente que possa carregar o ACK de "carona"). Há três maneiras de corrigir esse tipo de cliente:

1. Utilizar `writev` (Seção 14.4) em vez de duas chamadas a `write`. Uma única chamada a `writev` termina com uma chamada para a saída de TCP, em vez de duas chamadas, resultando em um dos segmentos TCP para nosso exemplo. Essa é a solução preferida.
2. Copiar os 4 bytes de dados e os 396 bytes de dados para um único buffer e chamar `write` uma vez para esse buffer.
3. Configurar a opção do soquete `TCP_NODELAY` e continuar a chamar `write` duas vezes. Essa é a solução menos desejável e é prejudicial à rede, geralmente nem mesmo deve ser considerada.

Os Exercícios 7.8 e 7.9 continuam esse exemplo.

7.10 Opções de soquete SCTP

O número relativamente grande de opções de soquete para SCTP (17 no momento em que este livro era escrito) reflete um controle mais aprimorado que o SCTP fornece ao desenvolvedor de aplicações. Especificamos *nível (level)* como `IPPROTO_SCTP`.

Várias opções utilizadas para obter informações sobre o SCTP requerem que os dados sejam passados para o kernel (por exemplo, ID de associação e/ou endereço de um peer). Embora algumas implementações de `getsockopt` suportem a passagem de dados tanto para dentro como para fora do kernel, nem todas fazem isso. A API de SCTP define uma função `sctp_opt_info` (Seção 9.11) que oculta essa diferença. Nos sistemas em que `getsockopt` suporta isso, ela simplesmente é uma empacotadora de `getsockopt`. Caso contrário, ela realiza a ação exigida, talvez utilizando um `ioctl` personalizado ou uma nova chamada de sistema. Recomendamos sempre utilizar `sctp_opt_info` ao recuperar essas opções para máxima portabilidade. Essas opções são marcadas com o símbolo (†) na Figura 7.2 e incluem `SCTP_ASSOCINFO`, `SCTP_GET_PEER_ADDR_INFO`, `SCTP_PEER_ADDR_PARAMS`, `SCTP_PRIMARY_ADDR`, `SCTP_RTOINFO` e `SCTP_STATUS`.

Opção de soquete SCTP_ADAPTION_LAYER

Durante a inicialização de associação, cada uma das extremidades pode especificar uma indicação de camada de adaptação. Essa indicação é um inteiro sem sinal de 32 bits que pode ser utilizado pelas duas aplicações para coordenar qualquer camada de adaptação da aplicação local. Essa opção permite ao chamador buscar ou configurar a indicação de camada de adaptação que essa extremidade fornecerá aos peers.

Ao buscar esse valor, o chamador irá recuperar apenas o valor que o soquete local fornecerá a todos os peers futuros. Para recuperar a indicação de camada de adaptação do peer, uma aplicação deve inscrever-se em eventos da camada de adaptação.

Opção de soquete SCTP_ASSOCINFO

A opção de soquete SCTP_ASSOCINFO pode ser utilizada para três propósitos: (i) recuperar informações sobre uma associação existente, (ii) alterar os parâmetros de uma associação existente, e/ou (iii) configurar valores default para associações futuras. Ao recuperar informações sobre uma associação existente, a função sctp_opt_info deve ser utilizada em vez de getsockopt. Essa opção aceita como entrada a estrutura sctp_assocparams.

```
struct sctp_assocparams {
   sctp_assoc_t sasoc_assoc_id;
   u_int16_t sasoc_asocmaxrxt;
   u_int16_t sasoc_number_peer_destinations;
   u_int32_t sasoc_peer_rwnd;
   u_int32_t sasoc_local_rwnd;
   u_int32_t sasoc_cookie_life;
};
```

Esses campos têm os seguintes significados:

- sasoc_assoc_id mantém a identificação da associação de interesse. Se esse valor for configurado como 0 ao chamar a função setsockopt, então sasoc_asocmaxrxt e sasoc_cookie_life representarão valores que são definidos como default no soquete. Chamar getsockopt retornará as informações específicas da associação se o ID de associação for fornecido; do contrário, se esse campo for 0, as configurações default da extremidade retornarão.
- sasoc_asocmaxrxt mantém o número máximo de retransmissões que uma associação fará sem um reconhecimento antes de desistir, informar o peer como inutilizável e fechar a associação.
- sasoc_number_peer_destinations mantém o número de endereços de destino do peer. Ela não pode ser configurada, somente recuperada.
- sasoc_peer_rwnd mantém calculada a janela de recebimento atual do peer. Esse valor representa o número total de bytes de dados que ainda pode ser enviado. Esse campo é dinâmico; à medida que a extremidade local envia dados, esse valor diminui. À medida que a aplicação remota lê os dados que foram recebidos, o valor aumenta. Esse valor não pode ser alterado por essa chamada de opção de soquete.
- sasoc_local_rwnd representa a janela de recebimento local que a pilha de SCTP está atualmente informando ao peer. Esse valor é dinâmico e também é influenciado pela opção de soquete SO_SNDBUF. Esse valor não pode ser alterado por essa chamada de opção de soquete.
- sasoc_cookie_life representa o número de milissegundos pelo qual um cookie, fornecido a um peer remoto, é válido. Cada cookie de estado enviado a um peer tem um tempo de vida associado para evitar ataques de replay. O valor default de 60.000 milissegundos pode ser alterado configurando essa opção com um valor sasoc_assoc_id de 0.

Forneceremos um conselho sobre como ajustar o valor de `sasoc_asocmaxrxt` para desempenho na Seção 23.11. O `sasoc_cookie _life` pode ser reduzido para melhor proteção contra ataques de replay de cookie, mas menor tolerância a retardo de rede durante a iniciação da associação. Os outros valores são úteis para depuração.

Opção de soquete `SCTP_AUTOCLOSE`

Essa opção permite buscar ou configurar o tempo de autofechamento de uma extremidade de SCTP. O tempo de autofechamento é o número de segundos durante o qual uma associação SCTP permanecerá aberta quando inativa. O tempo de inatividade é definido pela pilha de SCTP como nenhuma extremidade enviando ou recebendo dados de usuário. O default é que a função de autofechamento esteja desativada.

A opção de autofechamento é concebida para ser utilizada em uma interface SCTP no estilo de um para muitos (Capítulo 9). Se essa opção estiver ativada, o inteiro passado para ela é o número de segundos antes que uma conexão inativa deva ser fechada; um valor de 0 desativa o autofechamento. Apenas associações futuras criadas por essa extremidade serão afetadas por essa opção; associações existentes retêm sua configuração atual.

O autofechamento pode ser utilizado por um servidor para forçar o fechamento de associações inativas sem que o servidor precise manter um estado adicional. Um servidor que utiliza esse recurso precisa avaliar cuidadosamente o maior tempo de inatividade esperado em todas as suas associações. Configurar um valor de autofechamento menor que o necessário resulta no fechamento prematuro das associações.

Opção de soquete `SCTP_DEFAULT_SEND_PARAM`

O SCTP tem muitos parâmetros de envio opcionais que costumam ser passados como dados auxiliares ou utilizados com a chamada de função `sctp_sendmsg` (que é freqüentemente implementada como uma chamada de biblioteca que passa os dados auxiliares para o usuário). Uma aplicação que deseja enviar um grande número de mensagens, todas com os mesmos parâmetros, pode utilizar essa opção para configurar os parâmetros default e assim evitar utilizar dados auxiliares ou a chamada `sctp_sendmsg`. Essa opção possui como entrada a estrutura `sctp_sndrcvinfo`.

```
struct sctp_sndrcvinfo {
   u_int16_t sinfo_stream;
   u_int16_t sinfo_ssn;
   u_int16_t sinfo_flags;
   u_int32_t sinfo_ppid;
   u_int32_t sinfo_context;
   u_int32_t sinfo_timetolive;
   u_int32_t sinfo_tsn;
   u_int32_t sinfo_cumtsn;
   sctp_assoc_t sinfo_assoc_id;
};
```

Esses campos são definidos da seguinte maneira:

- `sinfo_stream` especifica o novo fluxo default ao qual todas as mensagens serão enviadas.
- `sinfo_ssn` é ignorado ao configurar as opções default. Ao receber uma mensagem com a função `recvmsg` ou `sctp_recvmsg`, esse campo manterá o valor que o peer colocou no campo número de seqüência de fluxo (*sequence number* – SSN) no bloco SCTP DATA.
- `sinfo_flags` determina os flags default a serem aplicados a todas as mensagens de envio futuras. Valores de flag admissíveis podem ser encontrados na Figura 7.16.

Constante	Descrição
MSG_ABORT	Invoca o término ABORTIVE da associação
MSG_ADDR_OVER	Especifica que o SCTP deve sobrescrever o endereço primário e, em vez disso, utilizar o endereço fornecido
MSG_EOF	Invoca um término elegante depois do envio dessa mensagem
MSG_PR_BUFFER	Ativa o perfil baseado em buffer do recurso de confiabilidade parcial (se disponível)
MSG_PR_SCTP	Ativa o recurso de confiabilidade parcial (se disponível) nessa mensagem
MSG_UNORDERED	Especifica que essa mensagem utiliza o serviço de mensagem não-ordenado

Figura 7.16 Valores de flag SCTP admissíveis para o campo `sinfo_flags`.

- `sinfo_pid` fornece o valor default a ser utilizado ao configurar o identificador de protocolo de payload do SCTP em todas as transmissões de dados.
- `sinfo_context` especifica o valor default aplicado ao campo `sinfo_context`, que é fornecido como um tag local quando as mensagens que não puderam ser enviadas a um peer são recuperadas.
- `sinfo_timetolive` dita o tempo de vida default que será aplicado a todos os envios de mensagens. O campo de tempo de vida é utilizado pelas pilhas de SCTP para saber quando descartar uma mensagem enviada devido a um retardo excessivo (antes da sua primeira transmissão). Se as duas extremidades suportarem a opção de confiabilidade parcial, o tempo de vida também será utilizado para especificar por quanto tempo uma mensagem será válida depois da sua primeira transmissão.
- `sinfo_tsn` é ignorado ao configurar as opções default. Ao receber uma mensagem com a função `recvmsg` ou com a função `sctp_recvmsg`, esse campo manterá o valor que o peer aplicou ao campo de número de seqüência de transporte (*transport sequence number* – TSN) no bloco SCTP DATA.
- `sinfo_cumtsn` é ignorado ao configurar as opções default. Ao receber uma mensagem com a função `recvmsg` ou `sctp_recvmsg`, esse campo manterá o TSN cumulativo atual que a pilha do SCTP local associou ao seu peer remoto.
- `sinfo_assoc_id` especifica a identificação de associação para a qual o requisitante deseja configurar os parâmetros default. Para soquetes de um para um, esse campo é ignorado.

Observe que todas as configurações default afetarão somente as mensagens enviadas sem uma estrutura `sctp_sndrcvinfo` própria. Qualquer envio que forneça essa estrutura (por exemplo, a função `sctp_sendmsg` ou `sendmsg` com dados auxiliares) sobrescreverá as configurações default. Além de configurar os valores default, essa opção pode ser utilizada para recuperar os parâmetros default atuais utilizando a função `sctp_opt_ info`.

Opção de soquete `SCTP_DISABLE_FRAGMENTS`

Normalmente, o SCTP fragmenta qualquer mensagem de usuário que não cabe em um único pacote SCTP para múltiplos blocos de DATA. Ativar essa opção desativa esse comportamento no emissor. Quando desativado por essa opção, o SCTP retornará o erro `EMSGSIZE` e não enviará a mensagem. O comportamento default é que essa opção esteja desativada; normalmente, o SCTP irá fragmentar as mensagens de usuário.

Essa opção pode ser utilizada por aplicações que desejam controlar o tamanho das mensagens, assegurando que cada mensagem de aplicação de usuário caberá em um único pacote IP. Uma aplicação que ativa essa opção deve estar preparada para tratar o caso de erro (isto é, sua mensagem era muito grande) fornecendo uma fragmentação da camada de aplicação da mensagem ou uma mensagem menor.

Opção de soquete SCTP_EVENTS

Essa opção de soquete permite a um chamador buscar, ativar ou desativar várias notificações de SCTP. Uma notificação de SCTP é uma mensagem que a pilha de SCTP enviará à aplicação. A mensagem é lida como dados normais, com o campo `msg_flags` da função `recvmsg` configurado como `MSG_ NOTIFICATION`. Uma aplicação que não está preparada para utilizar `recvmsg` ou `sctp_recvmsg` não deve ativar eventos. Oito diferentes tipos de eventos podem ser inscritos utilizando essa opção e passando uma estrutura `sctp_event_subscribe`. Qualquer valor em 0 representa uma não-subscrição e um valor em 1 representa uma subscrição.

A estrutura `sctp_event_subscribe` assume a seguinte forma:

```
struct sctp_event_subscribe {
   u_int8_t sctp_data_io_event;
   u_int8_t sctp_association_event;
   u_int8_t sctp_address_event;
   u_int8_t sctp_send_failure_event;
   u_int8_t sctp_peer_error_event;
   u_int8_t sctp_shutdown_event;
   u_int8_t sctp_partial_delivery_event;
   u_int8_t sctp_adaption_layer_event;
};
```

A Figura 7.17 resume os vários eventos. Outros detalhes sobre notificações podem ser encontrados na Seção 9.14.

Constante	Descrição
`sctp_data_io_event`	Ativa/desativa `sctp_sndrcvinfo` para vir com cada `recvmsg`
`sctp_association_event`	Ativa/desativa notificações de associação
`setp_address_event`	Ativa/desativa notificações de endereço
`sctp_send_failure_event`	Ativa/desativa notificações falhas de envio de mensagem
`sctp_peer_error_event`	Ativa/desativa notificações de erro de protocolo do peer
`sctp_shutdown_event`	Ativa/desativa notificações de desligamento
`sctp_partial_delivery_event`	Ativa/desativa notificações API de entrega parcial
`sctp_adaption_layer_event`	Ativa/desativa notificação da camada de adaptação

Figura 7.17 Subscrições de eventos SCTP.

Opção de soquete SCTP_GET_PEER_ADDR_INFO

Essa opção recupera as informações sobre um endereço do peer, incluindo a janela de congestionamento, RTT suavizado e MTU. Essa opção pode ser utilizada somente para recuperar informações sobre um endereço do peer específico. O chamador fornece uma estrutura `sctp_paddrinfo` com o campo `spinfo_address` preenchido com o endereço do peer de interesse e deve utilizar `sctp_opt_info` em vez de `getsockopt` para máxima portabilidade. A estrutura `sctp_paddrinfo` tem o seguinte formato:

```
struct sctp_paddrinfo {
   sctp_assoc_t spinfo_assoc_id;
   struct sockaddr_storage spinfo_address;
   int32_t spinfo_state;
   u_int32_t spinfo_cwnd;
   u_int32_t spinfo_srtt;
   u_int32_t spinfo_rto;
   u_int32_t spinfo_mtu;
};
```

Os dados retornados ao chamador fornecem o seguinte:

- `spinfo_assoc_id` contém as informações de identificação de associação, também fornecidas na notificação "communication up" (`SCTP_COMM_UP`). Esse valor único pode ser utilizado como um método abreviado para representar a associação para quase todas as operações de SCTP.
- `spinfo_address` é configurado pelo chamador para informar ao soquete de SCTP o endereço para o qual retornar as informações. Ao retornar, seu valor deve estar inalterado.
- `spinfo_state` mantém um ou mais dos valores vistos na Figura 7.18.

  Um *endereço não-confirmado* é um endereço que o peer listou como válido, mas a extremidade local do SCTP não foi capaz de confirmar se o peer o mantém. Uma extremidade de SCTP confirma um endereço quando os dados de *heatbeat* ou de usuário, enviados a esse endereço, são reconhecidos. Observe que um endereço não-confirmado também não terá um valor válido de tempo-limite de retransmissão (*retransmission timeout* – RTO). Endereços ativos representam endereços considerados como disponíveis para utilização.
- `spinfo_cwnd` representa a janela atual de congestionamento registrada para o endereço do peer. Uma descrição sobre como o valor de `cwnd` é conseguido pode ser encontrada na página 177 de Stewart e Xie (2001).
- `spinfo_srtt` representa a estimativa atual do RTT suavizado para esse endereço.
- `spinfo_rto` representa o tempo-limite atual de retransmissão em utilização para esse endereço.
- `spinfo_mtu` representa o MTU atual de caminho como encontrado pela descoberta de MTU do caminho.

Uma das utilizações interessantes para essa opção é converter uma estrutura de endereço IP em uma identificação de associação que pode ser utilizada em outras chamadas. Ilustraremos o uso dessa opção de soquete no Capítulo 23. Outra possibilidade é a aplicação monitorar o desempenho de cada endereço de um peer multihomed e atualizar o endereço primário da associação do melhor endereço do peer. Esses valores também são úteis para registro em log e depuração.

Constante	Descrição
`SCTP_ACTIVE`	O endereço está ativo e acessível
`SCTP_INACTIVE`	O endereço atualmente não pode ser acessado
`SCTP_ADDR_UNCONFIRMED`	Nenhum *heartbeat* ou dado confirmou este endereço

Figura 7.18 Estados de endereço do peer de SCTP.

Opção de soquete `SCTP_I_WANT_MAPPED_V4_ADDR`

Esse flag pode ser utilizado para ativar ou desativar endereços mapeados para IPv4 em um soquete do tipo `AF_INET6`. Observe que, quando ativada (que é o comportamento default), todos os endereços IPv4 serão mapeados para um endereço IPv6 antes de serem enviados à aplicação. Se essa opção estiver desativada, o soquete SCTP *não* irá mapear endereços IPv4 e, em vez disso, irá passá-los como uma estrutura `sockaddr_in`.

Opção de soquete SCTP_INITMSG

Essa opção pode ser utilizada para obter ou configurar os parâmetros iniciais default utilizados em um soquete e SCTP ao enviar a mensagem INIT. A opção utiliza a estrutura sctp_initmsg, definida como:

```
struct sctp_initmsg {
   uint16_t sinit_num_ostreams;
   uint16_t sinit_max_instreams;
   uint16_t sinit_max_attempts;
   uint16_t sinit_max_init_timeo;
};
```

Esses campos são definidos da seguinte maneira:

- sinit_num_ostreams representa o número de fluxos SCTP de saída que uma aplicação gostaria de solicitar. Esse valor não é confirmado até que a associação termine o handshake inicial, e pode ser negociado para baixo via limitações da extremidade do peer.
- sinit_max_instreams representa o número máximo de fluxos de entrada que a aplicação está preparada para permitir. Esse valor será sobrescrito pela pilha de SCTP se for maior que o número máximo de fluxos admissíveis que esta suporta.
- sinit_max_attempts expressa quantas vezes a pilha de SCTP deve enviar a mensagem inicial INIT antes de considerar inacessível a extremidade do peer.
- sinit_maxi_init_timeo representa o valor máximo de RTO para o timer de INIT. Durante um recuo binário exponencial (*binary exponential backoff*) do timer inicial, esse valor substitui RTO.max como o teto para retransmissões. Esse valor é expresso em milissegundos.

Observe que, ao configurar esses campos, qualquer valor configurado como 0 será ignorado pelo soquete SCTP. Um usuário de soquete no estilo de um para muitos (descrito na Seção 9.2) também pode passar uma estrutura sctp_initmsg em dados auxiliares durante uma configuração implícita de associação.

Opção de soquete SCTP_MAXBURST

Essa opção de soquete permite que a aplicação busque ou configure o *tamanho máximo de rajada (maximum burst size)* utilizado ao enviar pacotes. Quando uma implementação SCTP envia dados a um peer, não mais que SCTP_MAXBURST pacotes são enviados de uma vez para evitar inundação da rede com pacotes. Uma implementação pode aplicar esse limite de qualquer uma das seguintes maneiras: (i) reduzindo sua janela de congestionamento ao tamanho atual do percurso mais o tamanho máximo de rajada multiplicado pelo MTU do caminho, ou (ii) utilizando esse valor como um microcontrole separado, enviando pacotes no tamanho máximo de rajada em qualquer oportunidade individual de envio.

Opção de soquete SCTP_MAXSEG

Essa opção de soquete permite que a aplicação busque ou configure o *tamanho máximo de fragmento* utilizado durante a fragmentação do SCTP. Essa opção é semelhante à opção de TCP TCP_MAXSEG descrita na Seção 7.9.

Quando um emissor de SCTP recebe uma mensagem de uma aplicação maior que esse valor, ele dividirá a mensagem em múltiplos pedaços para o transporte até a extremidade do peer. O tamanho que o emissor de SCTP normalmente utiliza é o menor MTU entre todos os endereços associados ao peer. Essa opção sobrescreve esse valor para baixo até o valor especificado. Observe que a pilha de SCTP pode fragmentar uma mensagem com um limite menor que o solicitado por essa opção. Essa fragmentação menor ocorrerá se um dos caminhos

para a extremidade do peer tiver um MTU menor que o valor solicitado na opção SCTP_MAXSEG.

Esse valor é uma configuração da extremidade e pode afetar mais de uma associação no estilo de interface de um para muitos.

Opção de soquete SCTP_NODELAY

Se configurada, essa opção desativa o *algoritmo de Nagle* do SCTP. Essa opção está OFF por default (isto é, o algoritmo de Nagle está ON por default). O algoritmo de Nagle do SCTP funciona de maneira idêntica ao TCP, exceto que tenta fundir múltiplos blocos de DATA em oposição a simplesmente fundir bytes em um fluxo. Para uma discussão mais detalhada sobre o algoritmo de Nagle, consulte TCP_MAXSEG.

Opção de soquete SCTP_PEER_ADDR_PARAMS

Essa opção de soquete permite que uma aplicação busque ou configure vários parâmetros em uma associação. O chamador fornece a estrutura sctp_paddrparams, preenchendo a identificação de associação. A estrutura sctp_paddrparams tem o seguinte formato:

```
struct sctp_paddrparams {
   sctp_assoc_t spp_assoc_id;
   struct sockaddr_storage spp_address;
   u_int32_t spp_hbinterval;
   u_int16_t spp_pathmaxrxt;
};
```

Esses campos são definidos da seguinte maneira:

- spp_assoc_id mantém a identificação de associação para as informações sendo solicitadas ou configuradas. Se esse valor for configurado como 0, os valores default da extremidade serão configurados ou recuperados, em vez dos valores específicos da associação.
- spp_address especifica o endereço IP para os quais esses parâmetros são solicitados ou configurados. Se o campo spp_assoc_id estiver configurado como 0, esse campo será ignorado.
- spp_hbinterval é o intervalo entre heartbeats. Um valor em SCTP_NO_HB desativa heartbeats. Um valor em SCTP_ISSUE_HB solicita um heartbeat sob demanda. Qualquer outro valor altera o intervalo de heartbeats para esse valor em milissegundos. Ao configurar os parâmetros default, o valor SCTP_ISSUE_HB não é permitido.
- spp_hbpathmaxrxt mantém o número de retransmissões que serão tentadas nesse destino antes de ser declarado INACTIVE. Quando um endereço é declarado INACTIVE, se for o endereço primário, um endereço alternativo será escolhido como o primário.

Opção de soquete SCTP_PRIMARY_ADDR

Essa opção de soquete busca ou configura o endereço que a extremidade local está utilizando como primário. O endereço primário é utilizado, por default, como o endereço de destino para todas as mensagens enviadas a um peer. Para configurar esse valor, o chamador preenche a identificação de associação e o endereço do peer que deve ser utilizado como o endereço primário. O chamador passa essas informações em uma estrutura sctp_setprim, que é definida como:

```
struct sctp_setprim {
   sctp_assoc_t            ssp_assoc_id;
   struct sockaddr_storage ssp_addr;
};
```

Esses campos são definidos da seguinte maneira:

- `spp_assoc_id` especifica a identificação de associação em que o requisitante deseja configurar ou recuperar o endereço primário atual. Para o estilo de um para um, esse campo é ignorado.
- `sspp_addr` especifica o endereço primário, que deve ser um endereço que pertence ao peer. Se a operação for uma chamada de função `setsockopt`, o valor nesse campo então representará o novo endereço do peer que o requisitante gostaria de transformar no endereço primário de destino.

Observe que recuperar o valor dessa opção em um soquete de um para um com somente um endereço local associado é o mesmo que chamar `getsockname`.

Opção de soquete `SCTP_RTOINFO`

Essa opção de soquete pode ser utilizada para buscar ou configurar várias informações RTO sobre uma associação específica ou sobre os valores default utilizados por essa extremidade. Ao buscar, o chamador deve utilizar `sctp_opt_info` em vez de `getsockopt` para máxima portabilidade. O chamador fornece uma estrutura `sctp_rtoinfo` da seguinte forma:

```
struct sctp_rtoinfo {
   sctp_assoc_t          srto_assoc_id;
   uint32_t              srto_initial;
   uint32_t              srto_max;
   uint32_t              srto_min;
};
```

Esses campos são definidos da seguinte maneira:

- `srto_assoc_id` mantém a associação específica de interesse ou 0. Se esse campo contiver o valor 0, os parâmetros default do sistema serão afetados pela chamada.
- `srto_initial` contém o valor inicial de RTO utilizado para o endereço de um peer. O RTO inicial é utilizado ao enviar um bloco INIT ao peer. Esse valor está em milissegundos com um valor default de 3.000.
- `srto_max` contém o valor máximo de RTO que será utilizado quando uma atualização for feita no timer de retransmissão. Se o valor atualizado for maior que o RTO máximo, o RTO máximo será então utilizado como o RTO, em vez do valor calculado. O valor default para esse campo é 60.000 milissegundos.
- `srto_min` contém o valor RTO mínimo que será utilizado ao iniciar um timer de retransmissão. Sempre que uma atualização for feita no timer de RTO, o valor mínimo de RTO é verificado contra o novo valor. Se o novo valor for menor que o mínimo, o mínimo o substituirá. O valor default para esse campo é 1.000 milissegundos.

Um valor em 0 para `srto_initial`, `srto_max` ou `srto_min` indica que o valor default atualmente configurado não deve ser alterado. Todos os valores de data/hora são expressos em milissegundos. Forneceremos uma orientação sobre a configuração desses timers para melhorar o desempenho na Seção 23.11.

Opção de soquete `SCTP_SET_PEER_PRIMARY_ADDR`

Ativar essa opção faz com que uma mensagem seja enviada solicitando que o peer configure o endereço local especificado como seu endereço primário. O chamador fornece uma estrutura `sctp_setpeerprim` e deve preencher tanto a identificação de associação como um endereço local para solicitar a marca do peer como seu primário. O endereço fornecido deve ser um dos endereços vinculados à extremidade local. A estrutura `sctp_setpeerprim` é definida da seguinte maneira:

```
struct sctp_setpeerprim {
   sctp_assoc_t            sspp_assoc_id;
   struct sockaddr_storage sspp_addr;
};
```

Esses campos são definidos da seguinte maneira:

- `sspp_assoc_id` especifica a identificação de associação em que o requisitante deseja configurar o endereço primário. Para o estilo de um para um, esse campo é ignorado.
- `sspp_addr` mantém o endereço local que o requisitante deseja solicitar ao sistema do peer para configurá-lo como o endereço primário.

Esse recurso é opcional e deve ser suportado para que as duas extremidades operem. Se a extremidade local não suporta o recurso, um erro de `EOPNOTSUPP` será retornado ao chamador. Se a extremidade remota não suportar o recurso, um erro de `EINVAL` será retornado ao chamador. Observe que esse valor pode ser apenas configurado e não pode ser recuperado.

Opção de soquete `SCTP_STATUS`

Essa opção de soquete vai recuperar o estado atual de uma associação SCTP. O chamador deve utilizar `sctp_opt_info` em vez de `getaddrinfo` para máxima portabilidade. O chamador fornece uma estrutura `sctp_status`, preenchendo o campo de identificação de associação, `sstat_assoc_id`. A estrutura retornará preenchida com as informações que pertencem à associação solicitada. A estrutura `sctp_status` tem o seguinte formato:

```
struct sctp_status {
   sctp_assoc_t sstat_assoc_id;
   u_int32_t sstat_state;
   u_int32_t sstat_rwnd;
   u_int16_t sstat_unackdata;
   u_int16_t sstat_penddata;
   u_int16_t sstat_instrms;
   u_int16_t sstat_outstrms;
   u_int32_t sstat_fragmentation_point;
   struct sctp_paddrinfo sstat_primary;
};
```

Esses campos são definidos da seguinte maneira:

- `sstat_assoc_id` mantém a identificação de associação.
- `sstat_state` mantém um dos valores encontrados na Figura 7.19 e indica o estado geral da associação. Uma representação detalhada dos estados pelos quais uma extremidade de SCTP passa durante a configuração ou desativação de associação pode ser encontrada na Figura 2.8.

Constante	Descrição
SCTP_CLOSED	Uma associação fechada
SCTP_COOKIE_WAIT	Uma associação que enviou um INIT
SCTP_COOKIE_ECHOED	Uma associação que ecoou o COOKIE
SCTP_ESTABLISHED	Uma associação estabelecida
SCTP_SHUTDOWN_PENDING	Uma associação pendente que envia a desativação
SCTP_SHUTDOWN_SENT	Uma associação que enviou uma desativação
SCTP_SHUTDOWN_RECEIVED	Uma associação que recebeu uma desativação
SCTP_SHUTDOWN_ACK_SENT	Uma associação que está esperando um SHUTDOWN-COMPLETE

Figura 7.19 Estados de SCTP.

- `sstat_rwnd` mantém a estimativa atual da nossa extremidade para a janela de recebimento do peer.
- `sstat_unackdata` mantém o número de blocos DATA pendentes não-reconhecidos pelo peer.
- `sstat_penddata` mantém o número de blocos DATA não-lidos que a extremidade local está mantendo para a aplicação ler.
- `sstat_instrms` mantém o número de fluxos que o peer está utilizando para enviar dados a essa extremidade.
- `sstat_outstrms` mantém o número de fluxos admissíveis que essa extremidade pode utilizar para enviar dados ao peer.
- `sstat_fragmentation_point` contém o valor atual que a extremidade SCTP local está utilizando como o ponto de fragmentação para mensagens de usuário. Esse valor normalmente é o menor MTU de todos os destinos ou possivelmente o menor valor configurado pela aplicação local com `SCTP_MAXSEG`.
- `sstat_primary` mantém o endereço primário atual. O endereço primário é o endereço default utilizado ao enviar dados à extremidade do peer.

Esses valores são úteis para diagnósticos e para determinar as características da sessão; por exemplo, a função `sctp_get_no_strms` na Seção 10.2 utilizará o membro `sstat_outstrms` para determinar o número de fluxos disponíveis para utilização externa. Um valor de `sstat_rwnd` baixo e/ou um valor de `sstat_unackdata` alto pode ser utilizado para determinar se o buffer de soquete de recebimento do peer está se tornando cheio, o que pode ser utilizado como uma dica para a aplicação desacelerar a transmissão se possível. O `sstat_fragmentation_point` pode ser utilizado por algumas aplicações para reduzir o número de fragmentos que o SCTP tem de criar, enviando mensagens de aplicação menores.

7.11 Função `fcntl`

A função `fcntl` significa "controle de arquivo" e realiza várias operações de controle de descritor. Antes de descrever a função e como ela afeta um soquete, precisamos examinar um cenário maior. A Figura 7.20 resume as diferentes operações realizadas por `fcntl`, `ioctl` e soquetes de roteamento.

Operação	`fcntl`	`ioctl`	Soquete de roteamento	POSIX
Configura soquete para E/S não-bloqueadora	`F_SETFL, O_NONBLOCK`	`FIONBIO`		`fcntl`
Soquete configurado para E/S baseada em sinal	`F_SETFL, O_ASYNC`	`FIOASYNC`		`fcntl`
Configura o proprietário do soquete	`F_SETOWN`	`SIOCSPGRP` ou `FIOSETOWN`		`fcntl`
Obtém o proprietário do soquete	`F_GETOWN`	`SIOCGPGRP` ou `FIOGETOWN`		`fcntl`
Obtém o número de bytes no buffer de recebimento de soquete		`FIONREAD`		
Testa um soquete na marca fora da banda		`SIOCATMARK`		`sockatmark`
Obtém uma lista de interfaces		`SIOCGIFCONF`	`sysctl`	
Operações de interface		`SIOC[GS]IF`*xxx*		
Operações de cache ARP		`SIOC`*x*`ARP`	`RTM_`*xxx*	
Operações de roteamento de tabela		`SIOC`*xxx*`RT`	`RTM_`*xxx*	

Figura 7.20 Resumo de `fcntl`, `ioctl` e das operações de soquete de roteamento.

As primeiras seis operações podem ser aplicadas a soquetes por qualquer processo; as duas segundas (operações de interface) são menos comuns, mas ainda são de uso geral; e as duas últimas (ARP e tabela de roteamento) são emitidas por programas administrativos como `ifconfig` e `route`. Discutiremos as várias operações `ioctl` em detalhes no Capítulo 17 e os soquetes de roteamento no Capítulo 18.

Há múltiplas maneiras de realizar as primeiras quatro operações, mas observamos na coluna final que o POSIX especifica que `fcntl` é a preferida. Também observamos que o POSIX fornece a função `sockatmark` (Seção 24.3) como a maneira preferida de testar a marca fora da banda. As demais operações, com a coluna final em branco, não foram padronizadas pelo POSIX.

> Também observamos que as duas primeiras operações, configurar um soquete para E/S não-bloqueadora e para E/S baseada em sinal, foram configuradas historicamente utilizando os comandos `FNDELAY` e `FASYNC` com `fcntl`. O POSIX define as constantes `O_`*xxx*.

A função `fcntl` fornece os seguintes recursos relacionados com programação de rede:

- E/S não-bloqueadora – podemos configurar o flag de *status* de arquivo `O_NONBLOCK` utilizando o comando `F_SETFL` para configurar um soquete como não-bloqueador. Descreveremos E/S não-bloqueadora no Capítulo 16.
- E/S baseada em sinal – podemos configurar o flag de *status* de arquivo `O_ASYNC` utilizando o comando `F_SETFL`, o que faz com que o sinal `SIGIO` seja gerado quando o *status* de um soquete muda. Discutiremos isso no Capítulo 25.
- O comando `F_SETOWN` permite configurar o proprietário do soquete (o ID do processo ou ID do grupo de processos) para receber os sinais `SIGIO` e `SIGURG`. O `SIGIO` é gerado quando a E/S baseada em sinal é ativada para um soquete (Capítulo 25) e o `SIGURG` é gerado quando novos dados fora da banda chegam a um soquete (Capítulo 24). O comando `F_GETOWN` retorna o proprietário atual do soquete.

> O termo "proprietário do soquete" é definido pelo POSIX. Historicamente, as implementações derivadas do Berkeley chamavam isso de "ID de grupo de processos do soquete" porque a variável que armazena esse ID é o membro `so_pgid` da estrutura `socket` (página 438 do TCPv2).

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* int arg */ );
```

Retorna: depende do *cmd* se OK, –1 em erro

Cada descritor (incluindo soquete) tem um conjunto de flags de arquivo que é obtido com o comando `F_GETFL` e configurado com o comando `F_SETFL`. Os dois flags que afetam um soquete são:

`O_NONBLOCK` — E/S não-bloqueadora
`O_ASYNC` — E/S baseada em sinal

Descreveremos mais tarde esses dois recursos em mais detalhes. Por enquanto, observarmos que o código típico para ativar E/S não-bloqueadora, utilizando `fcntl`, seria:

```
int   flags;

   /* Configura um soquete como não-bloqueador */
if ( (flags = fcntl(fd, F_GETFL, 0)) < 0)
   err_sys("F_GETFL error");
```

```
flags |= O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
    err_sys("F_SETFL error");
```

Cuidado com o código, que você talvez encontre, que simplesmente configure o flag desejado.

```
   /* Maneira errada de configurar um soquete como não-bloqueador */
if (fcntl(fd, F_SETFL, O_NONBLOCK) < 0)
    err_sys("F_SETFL error");
```

Embora isso configure o flag não-bloqueador, ele também limpa todos os outros flags de *status* de arquivo. A única maneira correta de configurar um dos flags de *status* de arquivo é buscar os flags atuais, operar um OU lógico no novo flag e então reconfigurar os flags.

O código a seguir desativa o flag não-bloqueador, supondo que `flags` tenham sido configurados pela chamada a `fcntl` mostrada anteriormente:

```
flags &= ~O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
    err_sys("F_SETFL error");
```

Os dois sinais, `SIGIO` e `SIGURG`, são diferentes dos demais pelo fato de que são gerados para um soquete somente se o soquete foi atribuído a um proprietário com o comando `F_SETOWN`. O valor do tipo inteiro *arg* para o comando `F_SETOWN` pode ser um inteiro positivo, que especifica o ID do processo a receber o sinal, ou um inteiro negativo, cujo valor absoluto é o ID do grupo de processos a receber o sinal. O comando `F_GETOWN` retorna o proprietário de soquete como o valor de retorno da função `fcntl`, o ID do processo (um valor positivo de retorno) ou o ID do grupo de processos (um valor negativo diferente de –1). A diferença entre especificar um processo ou um grupo de processos para receber o sinal é que o primeiro faz com que somente um único processo receba o sinal, enquanto o último faz com que todos os processos no grupo de processos (talvez mais de um) recebam o sinal.

Quando um novo soquete é criado por `socket`, ele não tem nenhum proprietário. Mas, quando um novo soquete é criado a partir de um soquete ouvinte, o proprietário do soquete é herdado do soquete ouvinte pelo soquete conectado (como ocorrem com muitas opções de soquete [páginas 462 e 463 do TCPv2]).

7.12 Resumo

As opções de soquete variam de muito gerais (`SO_ERROR`) a muito específicas (opções de cabeçalho IP). As opções mais comumente utilizadas que poderíamos encontrar são `SO_KEEPALIVE`, `SO_RCVBUF`, `SO_SNDBUF` e `SO_REUSEADDR`. A última sempre deve ser configurada para um servidor TCP antes de ele chamar `bind` (Figura 11.12). A opção `SO_BROADCAST` e as 10 opções de soquete de multicast são somente para aplicações que utilizam broadcast ou multicast, respectivamente.

A opção de soquete `SO_KEEPALIVE` é configurada por muitos servidores TCP e automaticamente termina uma conexão meio aberta. A vantagem dessa opção é que ela é tratada pela camada TCP, sem exigir um timer de inatividade no nível de aplicação; sua desvantagem é que ela não pode informar a diferença entre um host cliente travado e uma perda temporária de conectividade do cliente. O SCTP fornece 17 opções de soquete que são utilizadas pela aplicação para controlar o transporte. `SCTP_NODELAY` e `SCTP_MAXSEG` são semelhantes a `TCP_NODELAY` e `TCP_MAXSEG` e desempenham funções equivalentes. As outras 15 opções fornecem a uma aplicação um controle melhor sobre a pilha de SCTP; discutiremos o uso de muitas dessas opções de soquete no Capítulo 23.

A opção de soquete `SO_LINGER` oferece mais controle sobre quando `close` retorna e também permite forçar o envio de um RST em vez de uma seqüência de término de conexão de quatro pacotes do TCP. Devemos ter cuidado ao enviar RSTs, porque isso impede o estado

TIME_WAIT do TCP. Na maior parte do tempo, essa opção de soquete não fornece as informações de que precisamos; nesses casos, um ACK de nível de aplicação é requerido.

Cada soquete TCP e SCTP tem um buffer de envio e um buffer de recebimento e cada soquete UDP tem um buffer de recebimento. As opções de soquete `SO_SNDBUF` e `SO_RCVBUF` permitem alterar os tamanhos desses buffers. A utilização mais comum dessas opções é para transferência de dados em volume através de pipes longos e gordos (*long fat pipes*): conexões TCP com uma alta largura de banda ou com um retardo longo, freqüentemente utilizando as extensões da RFC 1323. Os soquetes UDP, por outro lado, talvez queiram aumentar o tamanho do buffer de recebimento para permitir ao kernel enfileirar mais datagramas se a aplicação estiver ocupada.

Exercícios

7.1 Escreva um programa que imprima os tamanhos de buffer de envio e recebimento do TCP, do UDP e do SCTP defaults e que seja executado nos sistemas a que você tem acesso.

7.2 Modifique a Figura 1.5 como a seguir: antes de chamar `connect`, chame `getsockopt` para obter o tamanho e o MSS do buffer de recebimento do soquete. Imprima os dois valores. Depois que `connect` retorna com sucesso, busque essas mesmas duas opções de soquete e imprima seus valores. Os valores mudaram? Por quê? Execute o programa conectando-o a um servidor em sua rede local e também o execute conectando-o a um servidor em uma rede geograficamente distribuída (WAN). O MSS muda? Por quê? Você também deve executar o programa em diferentes hosts a que tem acesso.

7.3 Inicie com nosso servidor TCP nas Figuras 5.2 e 5.3 e com nosso cliente TCP nas Figuras 5.4 e 5.5. Modifique a função `main` de cliente para configurar a opção de soquete `SO_LINGER` antes de chamar `exit`, configurando `l_onoff` como 1 e `l_linger` como 0. Inicie o servidor e depois o cliente. Digite uma ou duas linhas no cliente para verificar a operação e então terminar o cliente inserindo seu caractere de EOF. O que acontece? Depois de terminar o cliente, execute `netstat` no host cliente e veja se o soquete passa pelo estado TIME_WAIT.

7.4 Suponha que dois clientes TCP iniciam aproximadamente ao mesmo tempo. Ambos configuram a opção de soquete `SO_REUSEADDR` e então chamam `bind` com o mesmo endereço e porta IP local (digamos 1500). Mas um deles chama `connect` para se conectar a 198.69.10.2 na porta 7000 e o outro chama `connect` para se conectar a 198.69.10.2 (mesmo endereço IP do peer), mas na porta 8000. Descreva a condição de corrida que ocorre.

7.5 Obtenha o código-fonte para os exemplos neste livro (consulte o Prefácio) e compile o programa `sock` (Seção C.3). Primeiro, classifique seu host como (a) sem suporte a multicast, (b) com suporte a multicast, mas sem `SO_REUSEPORT`, ou (c) com suporte a multicast e `SO_REUSEPORT`. Tente iniciar múltiplas instâncias do programa `sock` como um servidor TCP (a opção de linha de comando `-s`) na mesma porta, associando o endereço curinga, um dos endereços de interface do seu host e o endereço de loopback. É necessário especificar a opção `SO_REUSEADDR` (opção de linha de comando `-A`)? Utilize `netstat` para verificar os soquetes ouvintes.

7.6 Continue o exemplo anterior, mas inicie um servidor UDP (opção de linha de comando `-u`) e tente iniciar duas instâncias, ambas vinculando o mesmo endereço e porta IP local. Se sua implementação suportar `SO_REUSEPORT`, tente utilizá-la (opção de linha de comando `-T`).

7.7 Muitas versões do programa `ping` têm um flag `-d` para ativar a opção de soquete `SO_ DEBUG`. O que isso faz?

7.8 Continuando o exemplo no final da nossa discussão sobre a opção de soquete `TCP_NODELAY`, suponha que um cliente realize dois `writes`: o primeiro de 4 bytes e o segundo de 396 bytes. Suponha também que o ACK retardado do servidor seja 100 ms, o RTT entre o cliente e o servidor seja 100 ms e o tempo de processamento do servidor para solicitações do cliente seja 50 ms. Trace uma linha do tempo que mostre a interação do algoritmo de Nagle com os ACKs retardados.

7.9 Refaça o exercício anterior, supondo que a opção de soquete `TCP_NODELAY` esteja configurada.

7.10 Refaça o Exercício 7.8 supondo que o processo chame `writev` uma vez, tanto para o buffer de 4 bytes como para o buffer de 396 bytes.

7.11 Analise a RFC 1122 (Braden, 1989) para determinar o intervalo recomendado para ACKs retardados.

7.12 Onde nosso servidor nas Figuras 5.2 e 5.3 passa a maior parte do tempo? Suponha que o servidor tenha configurado a opção de soquete `SO_KEEPALIVE`, que não há nenhum dado sendo trocado na conexão e que o host cliente trava e não é reinicializado. O que acontece?

7.13 Onde nosso cliente nas Figuras 5.4 e 5.5 passa a maior parte do tempo? Suponha que o cliente configura a opção de soquete `SO_KEEPALIVE`, que não há nenhum dado sendo trocado através da conexão e que o servidor cai e não reinicializa. O que acontece?

7.14 Onde nosso cliente nas Figuras 5.4 e 6.13 passa a maior parte do tempo? Suponha que o cliente configura a opção de soquete `SO_KEEPALIVE`, que não há nenhum dado sendo trocado através da conexão e que o servidor cai e não reinicializa. O que acontece?

7.15 Suponha que um cliente e um servidor configuram a opção de soquete `SO_KEEPALIVE`. A conectividade é mantida entre os dois peers, mas não há nenhum dado de aplicação trocado na conexão. Quando o timer keep-alive expira a cada duas horas, quantos segmentos TCP são trocados na conexão?

7.16 Quase todas as implementações definem a constante `SO_ACCEPTCONN` no cabeçalho `<sys/socket.h>`, mas não descrevemos essa opção. Leia Lanciani (1996) para descobrir por que essa opção existe.

Capítulo 8

Soquetes UDP Elementares

8.1 Visão geral

Há algumas diferenças fundamentais entre aplicações escritas utilizando TCP e aquelas que utilizam UDP. Isso ocorre por causa das diferenças nas duas camadas de transporte: o UDP é um protocolo de datagrama sem conexão, não-confiável, bem diferente do fluxo confiável de bytes orientado a conexão fornecido pelo TCP. Contudo, há ocasiões em que faz sentido utilizar o UDP em vez do TCP; iremos revisar essa escolha de *design* na Seção 22.4. Algumas aplicações populares são construídas utilizando o UDP: por exemplo, DNS, NFS e SNMP.

A Figura 8.1 mostra as chamadas de função para um cliente/servidor de UDP típico. O cliente não estabelece uma conexão com o servidor. Em vez disso, o cliente envia apenas um datagrama ao servidor utilizando a função `sendto` (descrita na próxima seção), que requer o endereço de destino (o servidor) como um parâmetro. De maneira semelhante, o servidor não aceita uma conexão de um cliente. Em vez disso, o servidor simplesmente chama a função `recvfrom`, que espera até que os dados cheguem de algum cliente. `recvfrom` retorna o endereço de protocolo do cliente, junto com o datagrama, de modo que o servidor possa enviar uma resposta ao cliente correto.

A Figura 8.1 mostra uma linha do tempo de um cenário típico que acontece em uma troca entre cliente/servidor UDP. Podemos comparar isso a uma troca típica no TCP, mostrada na Figura 4.1.

Neste capítulo, descreveremos as novas funções utilizadas com os soquetes UDP, `recvfrom` e `sendto` e recriaremos nosso cliente/servidor de eco com o UDP. Também descreveremos o uso da função `connect` com um soquete UDP e o conceito de erros assíncronos.

8.2 Funções `recvfrom` e `sendto`

Essas duas funções são semelhantes às funções `read` e `write` padrão, mas são necessários três argumentos adicionais.

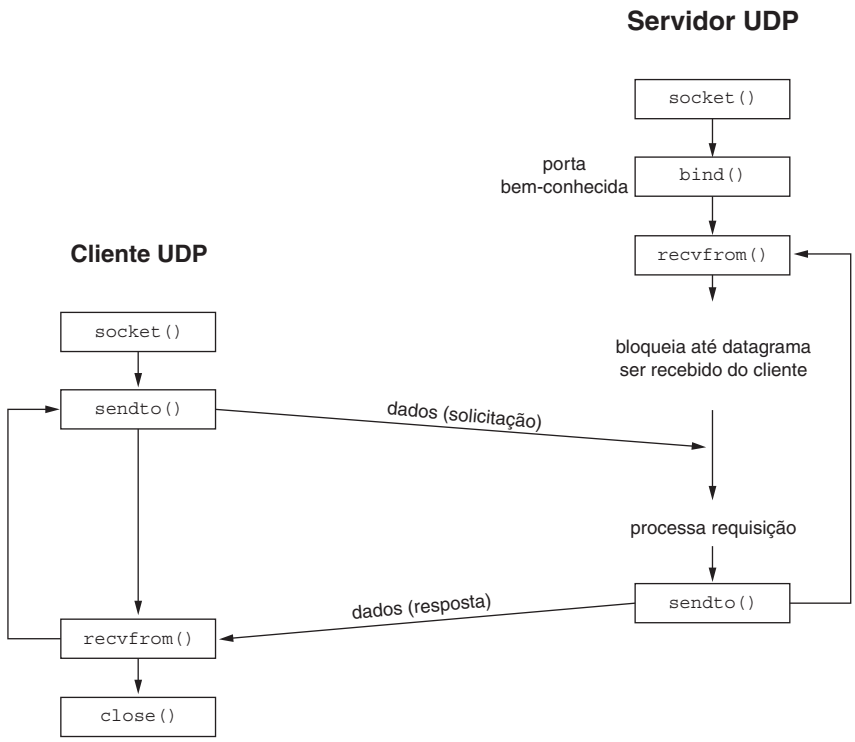


Figura 8.1 Funções socket para cliente/servidor UDP.

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

ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
                 struct sockaddr *from, soeklen_t *addrlen);

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags,
               const struct sockaddr *to, soeklen_t addrlen);
```

As duas retornam: número de bytes lidos ou gravados se OK, –1 no erro

Os três primeiros argumentos, *sockfd*, *buff* e *nbytes,* são idênticos aos três primeiros argumentos para `read` e `write`: descritor, ponteiro para leitura ou gravação do buffer e número de bytes a ler ou gravar.

Descreveremos o argumento *flags* no Capítulo 14 ao discutir as funções `recv`, `send`, `recvmsg` e `sendmsg`, uma vez que não precisamos delas para o nosso exemplo simples do cliente/servidor UDP neste capítulo. Por enquanto, sempre iremos configurar o flag como 0.

O argumento *to* de `sendto` é uma estrutura de endereço de soquete que contém o endereço de protocolo (por exemplo, o endereço e o número de porta IP) para onde os dados deverão ser enviados. O tamanho dessa estrutura de endereço de soquete é especificado por *addrlen*. A função `recvfrom` preenche a estrutura de endereço de soquete apontada por *from* com o endereço de protocolo de quem enviou o datagrama. O número de bytes armazenados nessa estrutura de endereço de soquete também retorna ao chamador no inteiro apontado por

addrlen. Observe que o argumento final para `sendto` é um valor do tipo inteiro, enquanto o argumento final para `recvfrom` é um ponteiro para um valor do tipo inteiro (um argumento valor-resultado).

Os dois argumentos finais para `recvfrom` são semelhantes aos dois argumentos finais para `accept`: o conteúdo da estrutura de endereço de soquete no retorno informa quem enviou o datagrama (no caso de UDP) ou quem iniciou a conexão (no caso de TCP). Os dois argumentos finais para `sendto` são semelhantes aos dois argumentos finais para `connect`: preenchemos a estrutura de endereço de soquete com o endereço de protocolo para onde enviar o datagrama (no caso de UDP) ou de com quem estabelecer uma conexão (no caso de TCP).

As duas funções retornam o comprimento dos dados que foram lidos ou gravados como o valor da função. Na utilização típica de `recvfrom`, com um protocolo de datagrama, o valor de retorno é a quantidade de dados de usuário no datagrama recebido.

Gravar um datagrama de comprimento 0 é aceitável. No caso de UDP, isso resulta em um datagrama IP contendo um cabeçalho IP (normalmente, 20 bytes para IPv4 e 40 bytes para IPv6), um cabeçalho UDP de 8 bytes e nenhum dado. Isso também significa que um valor de retorno de 0 a partir de `recvfrom` é aceitável para um protocolo de datagrama: não significa que o peer fechou a conexão, como faz um valor de retorno de 0 proveniente da leitura de um soquete TCP. Como UDP é sem conexão, não há nada parecido com fechar uma conexão UDP.

Se o argumento *from* de `recvfrom` for um ponteiro nulo, o argumento de comprimento correspondente (*addrlen*) também deve ser um ponteiro nulo e indicar que não estamos interessados em conhecer o endereço de protocolo de quem enviou os dados.

Tanto `recvfrom` como `sendto` podem ser utilizados com TCP, embora normalmente não haja nenhuma razão para fazer isso.

8.3 Servidor de eco UDP: função `main`

Agora recriaremos nosso cliente/servidor de eco simples do Capítulo 5 utilizando UDP. Nosso cliente UDP e programas de servidor seguem o fluxo de chamada de função que diagramamos na Figura 8.1. A Figura 8.2 mostra as funções utilizadas. A Figura 8.3 mostra a função `main` do servidor.

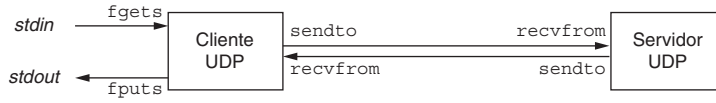


Figura 8.2 Cliente/servidor de eco simples utilizando UDP.

udpcliserv/udpserv01.c

```
1 #include   "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int    sockfd;
6     struct sockaddr_in servaddr, cliaddr;
7     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
8     bzero(&servaddr, sizeof(servaddr));
```

Figura 8.3 Servidor de eco UDP (*continua*).

```
 9      servaddr.sin_family = AF_INET;
10      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
11      servaddr.sin_port = htons(SERV_PORT);

12      Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));

13      dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
14  }
```

udpcliserv/udpserv01.c

Figura 8.3 Servidor de eco UDP (*continuação*).

Criação e vínculo de um soquete UDP a uma porta bem-conhecida do servidor

7-12 Criamos um soquete UDP especificando o segundo argumento para `socket` como `SOCK_DGRAM` (um soquete de datagrama no protocolo IPv4). Como ocorre com o exemplo do servidor TCP, o endereço IPv4 para `bind` é especificado como `INADDR_ANY` e a porta bem-conhecida do servidor é a constante `SERV_PORT` do cabeçalho `unp.h`.

13 A função `dg_echo` é chamada para realizar o processamento de servidor.

8.4 Servidor de eco UDP: função `dg_echo`

A Figura 8.4 mostra a função `dg_echo`.

lib/dg_echo.c

```
 1  #include   "unp.h"

 2  void
 3  dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 4  {
 5      int    n;
 6      socklen_t len;
 7      char   mesg[MAXLINE];

 8      for ( ; ; ) {
 9          len = clilen;
10          n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

11          Sendto(sockfd, mesg, n, 0, pcliaddr, len);
12      }
13  }
```

lib/dg_echo.c

Figura 8.4 Função `dg_echo`: ecoa linhas em um soquete de datagrama.

Leitura do datagrama, eco de volta ao emissor

8-12 Essa função é um loop simples que lê o próximo datagrama recebido na porta do servidor utilizando `recvfrom` e envia-o de volta utilizando `sendto`.

Apesar da simplicidade dessa função, há vários detalhes a serem considerados. Primeiro, essa função nunca termina. Como o UDP é um protocolo sem conexão, não há nada como um EOF como ocorre com o TCP.

Em seguida, essa função fornece um *servidor iterativo*, não um servidor concorrente como ocorre com TCP. Não há nenhuma chamada a `fork`, portanto, um único processo servidor trata um ou todos os clientes. Em geral, a maioria dos servidores TCP é concorrente e a maioria dos servidores UDP é iterativa.

Há um enfileiramento implícito acontecendo na camada UDP nesse soquete. De fato, cada soquete UDP tem um buffer de recebimento e cada datagrama que chega a esse soquete é

colocado nesse buffer de recebimento de soquete. Quando o processo chama `recvfrom`, o próximo datagrama no buffer retorna ao processo em uma ordem primeiro a entrar, primeiro a sair (*first-in, first-out* – FIFO). Dessa maneira, se múltiplos datagramas chegam ao soquete antes de o processo poder ler o que já está enfileirado para o soquete, os datagramas que chegam são simplesmente adicionados ao buffer de recebimento de soquete. Mas esse buffer tem um tamanho limitado. Discutimos esse tamanho e como aumentá-lo com a opção de soquete `SO_RCVBUF` na Seção 7.5.

A Figura 8.5 resume nosso TCP cliente/servidor do Capítulo 5, quando dois clientes estabelecem conexões com o servidor.

Há dois soquetes conectados ao host servidor e cada um deles tem seu próprio buffer de recebimento de soquete.

A Figura 8.6 mostra o cenário quando dois clientes enviam datagramas ao nosso servidor UDP.

Há somente um processo servidor e ele tem um único soquete em que recebe todos os datagramas que chegam e envia todas as respostas. Esse soquete tem um buffer de recebimento em que todos os datagramas que chegam são colocados.

A função `main` na Figura 8.3 é *dependente de protocolo* (ela cria um soquete do protocolo `AF_INET` e aloca e inicializa uma estrutura de endereço de soquete IPv4), porém a função `dg_echo` é *independente de protocolo*. A razão por que `dg_echo` é independente de protocolo é que o chamador (a função `main` no nosso caso) deve alocar uma estrutura de endereço de soquete do tamanho correto, e um ponteiro para essa estrutura e seu tamanho são passados como argumentos para `dg_echo`. A função `dg_echo` nunca examina dentro dessa estrutura dependente de protocolo: ela simplesmente passa um ponteiro da estrutura para `recvfrom` e `sendto`. `recvfrom` preenche essa estrutura com o endereço e o número de porta IP do cliente e, uma vez que o mesmo ponteiro (`pcliaddr`) é então passado para `sendto` como o endereço de destino, essa é a maneira como o datagrama é ecoado de volta ao cliente que o enviou.

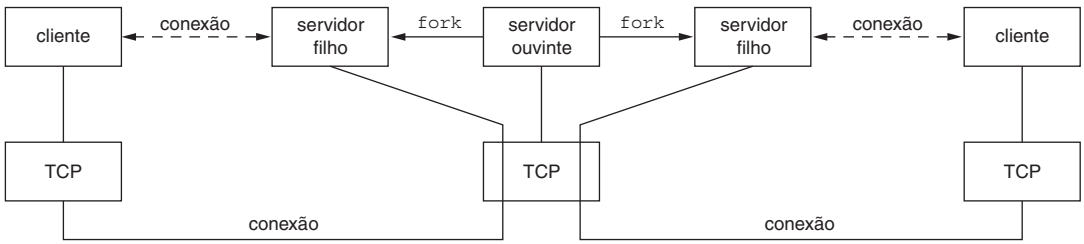


Figura 8.5 Resumo do cliente/servidor TCP com dois clientes.

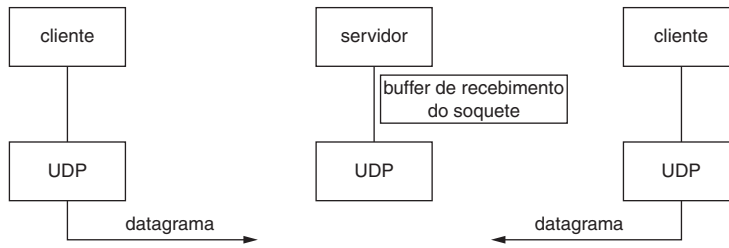


Figura 8.6 O resumo de UDP cliente/servidor com dois clientes.

8.5 Cliente de eco UDP: função `main`

A função `main` do cliente UDP é mostrada na Figura 8.7.

udpcliserv/udpcli01.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    sockfd;
 6     struct sockaddr_in servaddr;
 7     if (argc != 2)
 8         err_quit("usage: udpcli <IPaddress>");
 9     bzero(&servaddr, sizeof(servaddr));
10     servaddr.sin_family = AF_INET;
11     servaddr.sin_port = htons(SERV_PORT);
12     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
13     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
14     dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));
15     exit(0);
16 }
```

udpcliserv/udpcli01.c

Figura 8.7 Cliente de eco UDP.

Preenchimento da estrutura de endereço de soquete com o endereço do servidor

9-12 Uma estrutura de endereço de soquete IPv4 é preenchida com o endereço e o número da porta IP do servidor. Essa estrutura será passada para `dg_cli`, especificando para onde enviar os datagramas.

13-14 Um soquete UDP é criado e a função `dg_cli` é chamada.

8.6 Cliente de eco UDP: função `dg_cli`

A Figura 8.8 mostra a função `dg_cli`, que realiza a maior parte do processamento de cliente.

lib/dg_cli.c

```
 1 #include   "unp.h"
 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int    n;
 6     char   sendline[MAXLINE], recvline[MAXLINE + 1];
 7     while (Fgets(sendline, MAXLINE, fp) != NULL) {
 8         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
 9         n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
10         recvline[n] = 0;          /* null terminate */
11         Fputs(recvline, stdout);
12     }
13 }
```

lib/dg_cli.c

Figura 8.8 Função `dg_cli`: loop de processamento do cliente.

7-12 Há quatro passos no loop de processamento do cliente: ler uma linha da entrada-padrão utilizando `fgets`, enviar a linha ao servidor utilizando `sendto`, ler de volta o eco do servidor utilizando `recvfrom` e imprimir a linha ecoada na saída-padrão utilizando `fputs`.

Nosso cliente não solicitou que o kernel atribuísse uma porta efêmera ao seu soquete. (Com um cliente TCP, dissemos que a chamada `connect` é onde isso acontece.) Com um soquete UDP, na primeira vez que o processo chama `sendto`, se o soquete ainda não tiver uma porta local vinculada, é quando o kernel escolhe uma porta efêmera para o soquete. Como ocorre com o TCP, o cliente pode chamar `bind` explicitamente, mas isso é raro.

Observe que a chamada a `recvfrom` especifica um ponteiro nulo como quinto e sexto argumentos. Isso diz ao kernel que não estamos interessados em conhecer quem enviou a resposta. Há um risco de que um processo, no mesmo ou em um outro host, possa enviar um datagrama ao endereço e à porta IP do cliente e de que o datagrama seja lido pelo cliente, que pensará que é a resposta do servidor. Resolveremos isso na Seção 8.8.

Como ocorre com a função de servidor `dg_echo`, a função de cliente `dg_cli` é independente de protocolo, porém a função `main` de cliente é dependente de protocolo. A função `main` aloca e inicializa uma estrutura de endereço de soquete de algum tipo de protocolo e então passa um ponteiro para essa estrutura, junto com seu tamanho, para `dg_cli`.

8.7 Datagramas perdidos

Nosso exemplo de cliente/servidor UDP não é confiável. Se um datagrama de cliente for perdido (digamos descartado por algum roteador entre o cliente e o servidor), o cliente será bloqueado eternamente na sua chamada a `recvfrom` na função `dg_cli`, esperando uma resposta do servidor que nunca chegará. De maneira semelhante, se o datagrama do cliente chegar ao servidor, mas a resposta do servidor for perdida, o cliente novamente será bloqueado eternamente na sua chamada a `recvfrom`. Uma maneira típica de evitar isso é determinar um tempo-limite na chamada do cliente a `recvfrom`. Discutiremos isso na Seção 14.2.

Simplesmente estabelecer um tempo-limite em `recvfrom` não é uma solução completa. Por exemplo, se o tempo-limite expirou, não poderemos afirmar se foi o nosso datagrama que não chegou ao servidor ou se foi a resposta do servidor que não retornou. Se a solicitação do cliente fosse algo como "transferir uma certa quantidade de dinheiro da conta A para a conta B" (em vez do nosso servidor de eco simples), isso faria uma diferença enorme quanto a se a solicitação ou a resposta foi perdida. Discutiremos como adicionar confiabilidade a um cliente/servidor UDP em mais detalhes na Seção 22.5.

8.8 Verificando a resposta recebida

No final da Seção 8.6, mencionamos que qualquer processo que conhece o número da porta efêmera do cliente poderia enviar datagramas ao nosso cliente que seriam mesclados com as respostas normais de servidor. O que podemos fazer é alterar a chamada a `recvfrom` na Figura 8.8 para retornar o endereço IP e a porta de quem enviou a resposta e ignorar quaisquer datagramas recebidos que não são provenientes do servidor ao qual enviamos o datagrama. Há, porém, algumas armadilhas nisso, como veremos.

Primeiro, alteramos a função `main` de cliente (Figura 8.7) para utilizar o servidor de eco-padrão (Figura 2.18). Simplesmente substituímos a atribuição

```
servaddr.sin_port = htons(SERV_PORT);
```

por

```
servaddr.sin_port = htons(7);
```

Fazemos isso para poder utilizar qualquer host que execute o servidor de eco-padrão com o nosso cliente.

Em seguida, recodificamos a função `dg_cli` a fim de alocar uma outra estrutura de endereço de soquete para armazenar a estrutura retornada por `recvfrom`. Mostramos isso na Figura 8.9.

Alocação de outra estrutura de endereço de soquete

9 Alocamos outra estrutura de endereço de soquete chamando `malloc`. Observe que a função `dg_cli` continua a ser independente de protocolo; como não é importante o tipo de estrutura de endereço de soquete com que estamos lidando, utilizamos somente seu tamanho na chamada a `malloc`.

Comparando o endereço retornado

12-18 Na chamada a `recvfrom`, instruímos o kernel a retornar o endereço do emissor do datagrama. Primeiro, comparamos o comprimento retornado por `recvfrom` no argumento de valor-resultado, para então comparar as estruturas de endereços de soquetes utilizando `memcmp`.

> A Seção 3.2 diz que, mesmo se a estrutura de endereço de soquete contiver um campo de comprimento, nunca precisaremos configurá-lo ou examiná-lo. Entretanto, `memcmp` compara todos os bytes de dados nas duas estruturas de endereços de soquetes e o campo de comprimento é configurado na estrutura de endereço de soquete que o kernel retorna; portanto, nesse caso, devemos configurá-lo ao construir o `sockaddr`. Se não o configurarmos, `memcmp` irá comparar um 0 (uma vez que não o configuramos) com um 16 (assumindo `sockaddr_in`) e não serão iguais.

udpcliserv/dgcliaddr.c

```
 1 #include   "unp.h"
 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int    n;
 6     char   sendline[MAXLINE], recvline[MAXLINE + 1];
 7     socklen_t len;
 8     struct sockaddr *preply_addr;
 9     preply_addr = Malloc(servlen);
10     while (Fgets(sendline, MAXLINE, fp) != NULL) {
11         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
12         len = servlen;
13         n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
14         if (len != servlen || memcmp(pservaddr, preply_addr, len) != 0) {
15             printf("reply from %s (ignored)\n", Sock_ntop(preply_addr, len));
16             continue;
17         }
18         recvline[n] = 0;          /* termina com nulo */
19         Fputs(recvline, stdout);
20     }
21 }
```

udpcliserv/dgcliaddr.c

Figura 8.9 Versão de `dg_cli` que verifica o endereço de soquete retornado.

Essa nova versão do nosso cliente funciona bem se o servidor estiver em um host com somente um único endereço IP. Mas esse programa pode falhar se o servidor for multihomed. Executamos esse programa no nosso host `freebsd4`, que tem duas interfaces e dois endereços IP.

```
macosx % host freebsd4
freebsd4.unpbook.com has address 172.24.37.94
freebsd4.unpbook.com has address 135.197.17.100
macosx % udpcli02 135.197.17.100
hello
reply from 172.24.37.94:7 (ignored)
goodbye
reply from 172.24.37.94:7 (ignored)
```

Especificamos o endereço IP que não compartilha a mesma sub-rede do cliente.

> Normalmente, isso é permitido. A maioria das implementações IP aceita um datagrama IP entrando cujo destino seja qualquer um dos endereços IP do host, independentemente da interface em que o datagrama chega (páginas 217 a 219 do TCPv2). A RFC 1122 (Braden, 1989) denomina isso de *modelo de sistema com extremidade fraca*. Se um sistema implementasse o que essa RFC denomina de *modelo de sistema com extremidade forte*, ele aceitaria um datagrama de entrada somente se esse datagrama chegasse à interface à qual foi endereçado.

O endereço IP retornado por `recvfrom` (o endereço IP de origem do datagrama UDP) não é o endereço IP ao qual enviamos o datagrama. Quando o servidor envia sua resposta, o endereço IP de destino é 172.24.37.78. A função de roteamento dentro do kernel no `freebsd4` escolhe 172.24.37.94 como a interface de saída. Como o servidor não vinculou um endereço IP ao seu soquete (o servidor vinculou o endereço curinga ao seu soquete, algo que podemos verificar executando `netstat` no `freebsd`), o kernel escolhe o endereço de origem para o datagrama IP. Esse datagrama é escolhido como o endereço IP primário da interface de saída (páginas 232 e 233 do TCPv2). Além disso, como ele é o endereço IP primário da interface, se enviarmos nosso datagrama a um endereço IP não-primário da interface (isto é, um alias), isso também fará com que o nosso teste na Figura 8.9 falhe.

Uma solução é o cliente verificar o nome de domínio do host que responde, em vez de seu endereço IP, pesquisando o nome do servidor no DNS (Capítulo 11), dado o endereço IP retornado por `recvfrom`. Uma outra solução é o servidor UDP criar um soquete para cada endereço IP que é configurado no host, chamar `bind` para vincular esse endereço IP ao soquete, utilizar `select` por todos esses soquetes (esperando que qualquer um torne-se legível) e então responder a partir do soquete legível. Como o soquete utilizado para a resposta foi vinculado ao endereço IP que era o endereço-destino da solicitação do cliente (ou o datagrama não teria sido entregue ao soquete), isso garante que o endereço-origem da resposta foi o mesmo que o endereço-destino da solicitação. Mostraremos um exemplo disso na Seção 22.6.

> Em um sistema Solaris multihomed, o endereço IP de origem para a resposta do servidor é o endereço IP de destino da solicitação do cliente. O cenário descrito nesta seção destina-se a implementações derivadas do Berkeley que escolhem o endereço IP de origem com base na interface de saída.

8.9 Servidor não executando

O próximo cenário a examinar é iniciar o cliente sem iniciar o servidor. Se fizermos isso e digitarmos uma única linha para o cliente, nada acontecerá. O cliente é bloqueado eternamente na sua chamada a `recvfrom`, esperando uma resposta do servidor que nunca ocorrerá. Mas esse é um exemplo em que precisamos compreender melhor os protocolos subjacentes para entender o que acontece na nossa aplicação de rede.

Primeiro, iniciamos `tcpdump` no host `macosx` e, então, iniciamos o cliente no mesmo host, especificando `freebsd4` como o host servidor. Em seguida, digitamos uma única linha, mas esta não é ecoada.

```
macosx % udpcli01 172.24.37.94
hello, world                    digitamos essa linha,
                                mas nada é ecoado de volta
```

A Figura 8.10 mostra a saída de `tcpdump`.

```
1  0.0                   arp who-has freebsd4 tell macosx
2  0.003576 ( 0.0036)    arp reply freebsd4 is-at 0:40:5:42:d6:de

3  0.003601 ( 0.0000)    macosx.51139 > freebsd4.9877: udp 13
4  0.009781 ( 0.0062)    freebsd4 > macosx: icmp: freebsd4 udp port 9877
                           unreachable
```

Figura 8.10 Saída de `tcpdump` quando o processo de servidor não foi iniciado no host servidor.

Primeiro, percebemos que uma solicitação e uma resposta ARP são necessárias antes de o host cliente poder enviar o datagrama UDP ao host servidor. (Deixamos essa troca na saída para reiterar o potencial para uma resposta de solicitação ARP antes de um datagrama IP poder ser enviado a outro host ou roteador na rede local.)

Na linha 3, vemos o datagrama de cliente enviado, mas o host servidor responde na linha 4 com um ICMP "port unreachable" ("porta inacessível"). (O comprimento de 13 explica os 12 caracteres e a nova linha.) Esse erro ICMP, porém, não retorna ao processo cliente, por razões que descreveremos em breve. Em vez disso, o cliente é bloqueado eternamente na chamada a `recvfrom` na Figura 8.8. Também percebemos que o ICMPv6 tem um erro de "port unreachable", semelhante ao ICMPv4 (Figuras A.15 e A.16), portanto, os resultados descritos aqui são semelhantes para o IPv6.

Denominamos esse erro ICMP de *erro assíncrono*. O erro foi causado por `sendto`, mas `sendto` retornou com sucesso. Lembre-se, da Seção 2.11, de que um retorno bem-sucedido de uma operação de saída de UDP significa apenas que havia espaço na fila de saída de interface para o datagrama IP resultante. O erro ICMP não é retornado até mais tarde (4 ms mais tarde na Figura 8.10), razão pela qual é chamado de erro assíncrono.

A regra básica é que um erro assíncrono não retorna a um soquete UDP a menos que o soquete esteja conectado. Descreveremos como chamar `connect` para um soquete UDP na Seção 8.11. A razão pela qual essa decisão de *design* foi tomada quando os soquetes foram inicialmente implementados é raramente compreendida. (As implicações sobre a implementação são discutidas nas páginas 748 e 749 do TCPv2.)

Considere um cliente UDP que envia três datagramas em seqüência a três servidores diferentes (isto é, a três endereços IP diferentes) em um único soquete UDP. O cliente então entra em um loop que chama `recvfrom` para ler as respostas. Dois desses datagramas são corretamente entregues (isto é, o servidor estava em execução em dois dos três hosts), mas o terceiro host não estava executando o servidor. Esse terceiro host responde com uma mensagem de porta ICMP inacessível. Essa mensagem de erro ICMP contém o cabeçalho IP e o cabeçalho UDP do datagrama que causou o erro. (Mensagens de erro ICMPv4 e ICMPv6 sempre contêm o cabeçalho IP e todo o cabeçalho UDP ou parte do cabeçalho TCP para permitir que o receptor do erro ICMP determine qual soquete causou o erro. Mostraremos isso nas Figuras 28.21 e 28.22.) O cliente que enviou os três datagramas precisa conhecer o destino do datagrama que causou o erro para distinguir qual dos três datagramas causou o erro. Mas como o kernel pode retornar essas informações ao processo? A única informação que `recvfrom` pode retornar é um valor de `errno`; `recvfrom` não tem como retornar o endereço IP de destino e o número da porta UDP de destino do datagrama no erro. Portanto, tomou-se a decisão de retornar esses erros assíncronos ao processo somente se o processo conectasse o soquete UDP a exatamente um peer.

> O Linux retorna a maioria dos erros ICMP de "destination unreachable" mesmo para soquetes não-conectados, contanto que a opção de soquete `SO_BSDCOMPAT` não esteja ativada. Todos os erros ICMP de "destination unreachable" na Figura A.15 são retornados, exceto os códigos 0, 1, 4, 5, 11 e 12.
>
> Retornaremos ao problema desses erros assíncronos com soquetes UDP na Seção 28.7 e mostraremos uma maneira fácil de obter esses erros em soquetes não-conectados utilizando nosso próprio daemon.

8.10 Resumo de exemplo UDP

A Figura 8.11 mostra como marcar os quatro valores que devem ser especificados ou escolhidos quando o cliente envia um datagrama UDP.

O cliente deve especificar o endereço e o número da porta IP do servidor na chamada a `sendto`. Normalmente, o endereço e a porta IP do cliente são escolhidos automaticamente pelo kernel, embora tenhamos mencionado que o cliente possa chamar `bind` se preferir. No caso de esses dois valores para o cliente serem escolhidos pelo kernel, também mencionamos que a porta efêmera do cliente é escolhida uma vez, na primeira `sendto`, e então nunca muda. Entretanto, o endereço IP do cliente pode mudar para cada datagrama UDP que o cliente envia, supondo que o cliente não chama `bind` para vincular um endereço IP específico ao soquete. A razão é mostrada na Figura 8.11: se o host cliente for multihomed, o cliente poderia alternar entre dois destinos, um saindo do enlace de dados à esquerda e o outro saindo do enlace de dados à direita. Nesse cenário do pior caso, o endereço IP do cliente, como escolhido pelo kernel com base no enlace de dados de saída, iria alterar para cada datagrama.

O que acontece se o cliente chamar `bind` para vincular um endereço IP ao seu soquete, mas o kernel decidir que um datagrama de saída deve ser enviado por algum outro enlace de dados? Nesse caso, o datagrama IP conterá um endereço IP de origem diferente do endereço IP do enlace de dados de saída (consulte o Exercício 8.6).

A Figura 8.12 mostra os mesmos quatro valores, mas da perspectiva do servidor.

Há pelo menos quatro informações que um servidor poderia querer saber a partir de um datagrama IP que chega: o endereço IP de origem, o endereço IP de destino, o número de porta de origem e número de porta de destino. A Figura 8.13 mostra as chamadas de função que retornam essas informações a um servidor TCP e a um servidor UDP.

Um servidor TCP sempre tem acesso fácil a todas essas quatro informações para um soquete conectado e esses quatro valores permanecem constantes pelo tempo de vida de uma conexão. Com um soquete UDP, porém, o endereço IP de destino pode ser somente obtido con-

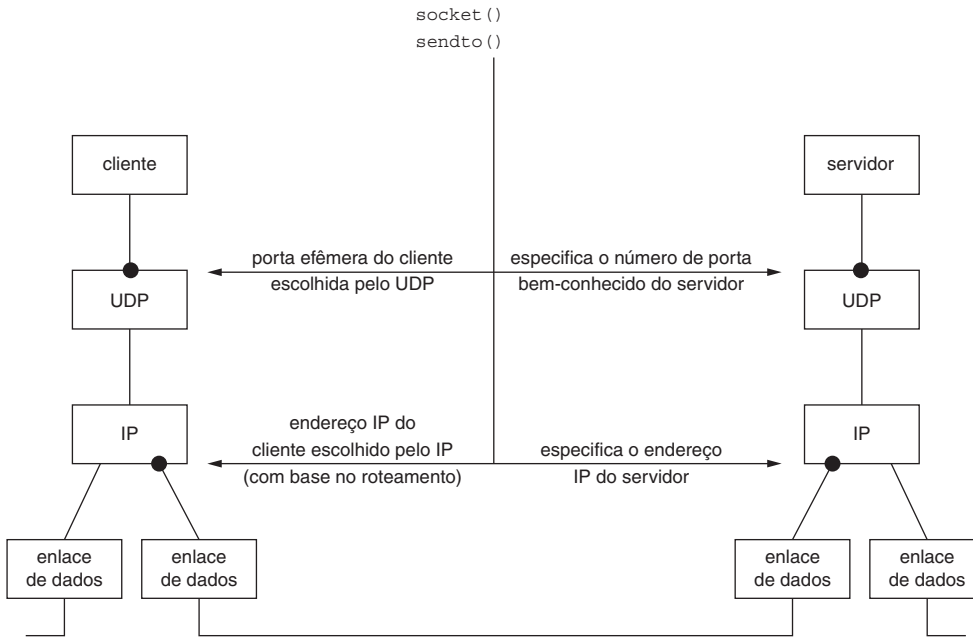


Figura 8.11 Resumo de cliente/servidor UDP da perspectiva do cliente.

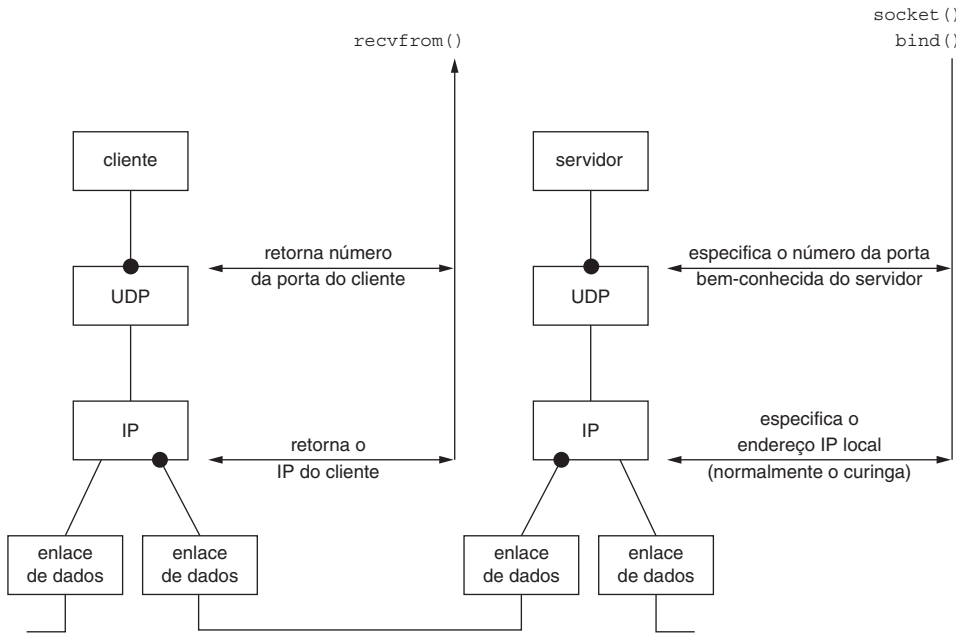


Figura 8.12 Resumo do cliente/servidor UDP da perspectiva do servidor.

Do datagrama IP do cliente	Servidor TCP	Servidor UDP
Endereço IP de origem	`accept`	`recvfrom`
Número da porta de origem	`accept`	`recvfrom`
Endereço IP de destino	`getsockname`	`recvmsg`
Número da porta de destino	`getsockname`	`getsockname`

Figura 8.13 Informações disponíveis ao servidor a partir do datagrama IP que chega.

figurando a opção de soquete `IP_RECVDSTADDR` para o IPv4 ou a opção de soquete `IPV6_PKTINFO` para o IPv6 e então chamando `recvmsg` em vez de `recvfrom`. Como o UDP é sem conexão, o endereço IP de destino pode mudar para cada datagrama enviado ao servidor. Um servidor UDP também pode receber datagramas destinados a um dos endereços de broadcast ou a um endereço de multicast do host, como será discutido nos Capítulos 20 e 21. Mostraremos como determinar o endereço de destino de um datagrama UDP na Seção 22.2, depois de discutir a função `recvmsg`.

8.11 Função `connect` com UDP

Mencionamos no final da Seção 8.9 que um erro assíncrono não retorna em um soquete UDP a menos que este esteja conectado. De fato, podemos chamar `connect` (Seção 4.3) para um soquete UDP. Mas isso não resulta em algo como uma conexão TCP: não há um handshake de três vias. Em vez disso, o kernel apenas verifica quaisquer erros imediatos (por exemplo, um destino obviamente inacessível), registra o endereço IP e o número de porta do peer (a partir da estrutura de endereço de soquete passada para `connect`) e retorna imediatamente ao processo que chama.

> Sobrecarregar a função `connect` com essa capacidade para soquetes UDP é confuso. Se é utilizada a convenção de que `sockname` é o endereço de protocolo local e `peername` é o endereço de protocolo externo, um nome mais apropriado seria `setpeername`. De maneira semelhante, um nome mais apropriado à função `bind` seria `setsockname`.

Com essa capacidade, agora devemos fazer uma distinção entre

- Um soquete UDP *não-conectado*, o default quando criamos um soquete UDP
- Um soquete UDP *conectado*, o resultado de chamar connect em um soquete UDP

Com um soquete UDP conectado, três coisas mudam, comparado ao soquete UDP não-conectado default:

1. Não podemos mais especificar o endereço IP e a porta de destino para uma operação de saída. Isto é, não utilizamos `sendto`, mas, em vez disso, `write` ou `send`. Qualquer coisa gravada em um soquete UDP conectado é automaticamente enviada ao endereço de protocolo (por exemplo, o endereço IP e a porta) especificado por `connect`.

   > Semelhante ao TCP, podemos chamar `sendto` para um soquete UDP conectado, mas não podemos especificar um endereço de destino. O quinto argumento para `sendto` (o ponteiro para a estrutura de endereço de soquete) deve ser um ponteiro nulo e o sexto argumento (o tamanho da estrutura de endereço de soquete) deve ser 0. A especificação POSIX afirma que, se o quinto argumento for um ponteiro nulo, o sexto argumento será ignorado.

2. Não precisamos utilizar `recvfrom` para descobrir o emissor de um datagrama, mas `read`, `recv` ou `recvmsg`. Os únicos datagramas retornados pelo kernel para uma operação de entrada em um soquete UDP conectado são aqueles que chegam do endereço de protocolo especificado em `connect`. Datagramas destinados ao endereço de protocolo local do soquete UDP conectado (por exemplo, o endereço IP e a porta), mas provenientes de um endereço de protocolo diferente daquele ao qual o soquete foi conectado, não são passados para o soquete conectado. Isso limita um soquete UDP conectado a trocar datagramas somente com um único peer.

   > Tecnicamente, um soquete UDP conectado troca datagramas com somente um endereço IP, porque usando `connect` é possível conectar a um endereço de multicast ou de broadcast.

3. Erros assíncronos, para soquetes UDP conectados, retornam ao processo.

   A decorrência disso, como descrevemos anteriormente, é que os soquetes UDP desconectados não recebem erros assíncronos.

A Figura 8.14 resume o primeiro ponto na lista com relação ao 4.4BSD.

> A especificação POSIX declara que uma operação de saída que não especifica um endereço de destino em um soquete UDP desconectado deve retornar `ENOTCONN`, não `EDESTADDRREQ`.

A Figura 8.15 resume os três pontos que fizemos sobre um soquete UDP conectado.

Tipo de soquete	`write` ou `send`	`sendto` que não especifica um destino	`sendto` que especifica um destino
Soquete TCP	OK	OK	`EISCONN`
Soquete UDP, conectado	OK	OK	`EISCONN`
Soquete UDP, não-conectado	`EDESTADDRREQ`	`EDESTADDRREQ`	OK

Figura 8.14 Soquetes TCP e UDP: Um endereço de protocolo de destino pode ser especificado?

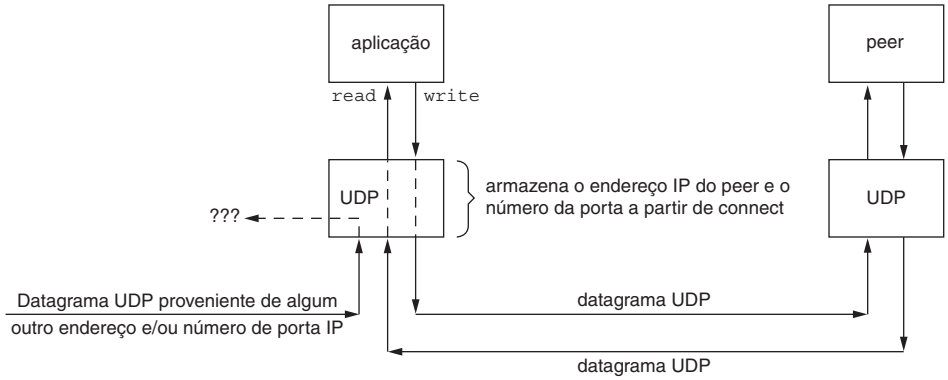


Figura 8.15 Soquete UDP conectado.

A aplicação chama `connect`, especificando o endereço IP e o número da porta do seu peer. Em seguida, ela utiliza `read` e `write` para trocar dados com o peer.

Os datagramas provenientes de qualquer outro endereço IP ou porta (que mostramos como "???" na Figura 8.15) não são passados para o soquete conectado porque o endereço IP de origem ou a porta UDP de origem não correspondem ao endereço de protocolo ao qual o soquete se conecta. Esses datagramas poderiam ser enviados a algum outro soquete UDP no host. Se não houver nenhum outro soquete correspondente ao datagrama entrante, o UDP irá descartá-lo e gerar um erro ICMP "porta inacessível".

Em resumo, podemos dizer que um cliente ou servidor UDP pode chamar `connect` somente se esse processo utilizar o soquete UDP para comunicar-se exatamente com um peer. Normalmente, é um cliente UDP quem chama `connect`, mas há aplicações em que o servidor UDP se comunica com um único cliente por um longo período de tempo (por exemplo, TFTP); nesse caso, tanto o cliente como o servidor podem chamar `connect`.

O DNS fornece um outro exemplo, como mostrado na Figura 8.16.

Um cliente DNS pode ser configurado para utilizar um ou mais servidores, normalmente listando os endereços IP dos servidores no arquivo `/etc/resolv.conf`. Se um único servidor estiver listado (a caixa mais à esquerda na figura), o cliente poderá chamar `connect`, mas se múltiplos servidores estiverem listados (a segunda caixa à direita na figura), o cliente não poderá chamar `connect`. Além disso, um servidor de DNS normalmente trata qualquer solicitação de cliente, portanto, os servidores não podem chamar `connect`.

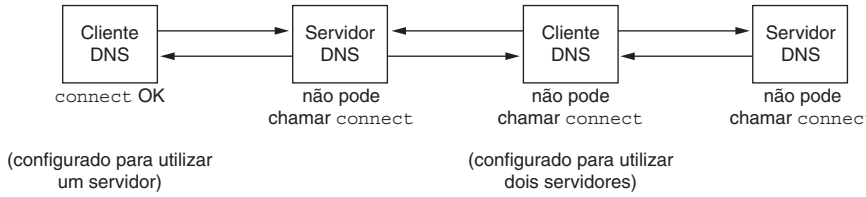


Figura 8.16 O exemplo de clientes e servidores de DNS e a função `connect`.

Chamando `connect` múltiplas vezes para um soquete UDP

Um processo com um soquete UDP conectado pode chamar `connect` novamente nesse soquete por uma entre duas razões:

- Para especificar um novo endereço e porta IP
- Para desconectar o soquete

O primeiro caso, especificar um novo peer para um soquete UDP conectado, difere do uso de `connect` com um soquete TCP: `connect` pode ser chamado somente uma vez para um soquete TCP.

Para desconectar um soquete UDP, chamamos `connect`, mas configuramos o membro da família da estrutura de endereço de soquete (`sin_family` para o IPv4 ou `sin6_family` para o IPv6) como `AF_UNSPEC`. Isso poderia retornar um erro `EAFNOSUPPORT` (página 736 do TCPv2), o que é aceitável. É o processo de chamar `connect` em um soquete UDP já conectado que faz com que o soquete torne-se desconectado (páginas 787 e 788 do TCPv2).

> As variantes do Unix parecem diferir sobre como exatamente desconectar um soquete e é possível encontrar abordagens que funcionam em alguns sistemas e não em outros. Por exemplo, chamar `connect` com `NULL` para endereço funciona somente em alguns sistemas (em outros, apenas se o terceiro argumento, o comprimento, for não zero). A especificação POSIX e as páginas man do BSD não ajudam muito aqui, elas somente mencionam que um *endereço nulo* deve ser utilizado sem mencionar absolutamente nada sobre o retorno de erro (mesmo se bem-sucedido). A solução mais portável é zerar uma estrutura de endereço, configurar a família como `AF_UNSPEC` como mencionado anteriormente e passá-la para `connect`.
>
> Uma outra área de desacordo é a respeito da vinculação local de um soquete durante o processo de desconexão. O AIX mantém tanto o endereço IP como a porta local escolhidos, mesmo em um `bind` implícito. O FreeBSD e o Linux configuram o endereço IP local de volta para tudo zero, mesmo se `bind` tiver sido chamado anteriormente, mas deixam o número da porta intacto. O Solaris configura o endereço IP local de volta para tudo zero se ele tiver sido atribuído por um `bind` implícito; mas, se o programa chamar `bind` explicitamente, o endereço IP permanece inalterado.

Desempenho

Quando uma aplicação chama `sendto` em um soquete UDP não-conectado, kernels derivados do Berkeley conectam o soquete temporariamente, enviam o datagrama e então desconectam o soquete (páginas 762 e 763 do TCPv2). Chamar `sendto` para dois datagramas em um soquete UDP não-conectado envolve os seis passos a seguir do kernel:

- Conectar o soquete
- Gerar a saída do primeiro datagrama
- Desconectar o soquete
- Conectar o soquete
- Gerar a saída do segundo datagrama
- Desconectar o soquete

> Uma outra consideração é sobre o número de pesquisas da tabela de roteamento. A primeira conexão temporária pesquisa o endereço IP de destino na tabela de roteamento e salva (armazena em cache) essas informações. A segunda conexão temporária observa que o endereço de destino é igual ao endereço de destino das informações na tabela de roteamento armazenadas em cache (estamos supondo dois `sendto` para o mesmo destino) e assim não precisamos pesquisar a tabela de roteamento novamente (páginas 737 e 738 do TCPv2).

Quando a aplicação sabe que enviará múltiplos datagramas ao mesmo peer, é mais eficiente conectar o soquete explicitamente. Chamar `connect` e então `write` duas vezes envolve os passos a seguir do kernel:

- Conectar o soquete
- Gerar a saída do primeiro datagrama
- Gerar a saída do segundo datagrama

Nesse caso, o kernel copia somente uma vez a estrutura de endereço de soquete que contém o endereço IP e a porta de destino, *versus* duas vezes quando `sendto` é chamado duas vezes. Partridge e Pink (1993) observam que a conexão temporária de um soquete UDP não-conectado é responsável por quase um terço do custo de cada transmissão UDP.

8.12 Função `dg_cli` (revisitada)

Agora retornamos à função `dg_cli` da Figura 8.8 e a recodificamos e denominamos de `connect`. A Figura 8.17 mostra a nova função.

udpcliserv/dgcliconnect.c

```
 1 #include   "unp.h"

 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int    n;
 6     char   sendline[MAXLINE], recvline[MAXLINE + 1];

 7     Connect(sockfd, (SA *) pservaddr, servlen);

 8     while (Fgets(sendline, MAXLINE, fp) != NULL) {

 9         Write(sockfd, sendline, strlen(sendline));

10         n = Read(sockfd, recvline, MAXLINE);

11         recvline[n] = 0;          /* termina com nulo */
12         Fputs(recvline, stdout);
13     }
14 }
```

udpcliserv/dgcliconnect.c

Figura 8.17 Função `dg_cli` que chama `connect`.

As alterações são a nova chamada a `connect` substituindo as chamadas a `sendto` e `recvfrom` pelas chamadas a `write` e `read`. Essa função ainda é independente de protocolo, uma vez que não examina dentro da estrutura de endereço de soquete que é passada para `connect`. A função `main` do nosso cliente, Figura 8.7, permanece inalterada.

Se executarmos esse programa no host `macosx`, especificando o endereço IP do host `freebsd4` (que não está executando nosso servidor na porta 9877), teremos a seguinte saída:

```
macosx % udpcli04 172.24.37.94
hello, world
read error: Connection refused
```

O primeiro ponto que observamos é que não recebemos o erro quando iniciamos o processo cliente. O erro ocorre somente depois que enviamos o primeiro datagrama ao servidor. É o envio desse datagrama que extrai o erro ICMP do host servidor. Mas, quando um cliente TCP chama `connect`, especificando um host servidor que não está executando o processo servidor, `connect` retorna o erro porque a chamada a `connect` faz com que o handshake de três vias do TCP ocorra e o primeiro pacote desse handshake extrai um RST do TCP de servidor (Seção 4.3).

A Figura 8.18 mostra a saída de `tcpdump`.

```
macosx % tcpdump
 1 0.0                  macosx.51139 > freebsd4.9877: udp 13
 2 0.006180 ( 0.0062)   freebsd4 > macosx: icmp: freebsd4 udp port 9877 unreachable
```

Figura 8.18 A saída de `tcpdump` ao executar a Figura 8.17.

Também veremos na Figura A.15 que esse erro de ICMP é mapeado pelo kernel para o erro `ECONNREFUSED`, que corresponde à saída do string de mensagem pela nossa função `err_sys`: "Connection refused".

> Infelizmente, nem todos os kernels retornam as mensagens ICMP para um soquete UDP conectado, como mostramos nesta seção. Normalmente, kernels derivados do Berkeley retornam o erro, enquanto kernels do System V não retornam. Por exemplo, se executarmos o mesmo cliente em um host do Solaris 2.4 e chamarmos `connect` para fazer a conexão a um host que não está executando nosso servidor, podemos observar com o `tcpdump` e verificar se o erro "porta inacessível" de ICMP é retornado pelo host servidor, porém a chamada do cliente a `read` nunca retorna. Esse bug foi corrigido no Solaris 2.5. O UnixWare não retorna o erro, enquanto o AIX, o Digital Unix, o HP-UX e o Linux, todos, retornam o erro.

8.13 Falta de controle de fluxo com UDP

Agora examinamos o efeito do UDP sem qualquer controle de fluxo. Primeiro, modificamos nossa função `dg_cli` para enviar um número fixo de datagramas. Ela não lê mais a partir da entrada-padrão. A Figura 8.19 mostra a nova versão, a qual grava datagramas UDP de 2.000 e 1.400 bytes no servidor.

Em seguida, modificamos o servidor para receber datagramas e contar o número recebido. Esse servidor não mais ecoa datagramas de volta para o cliente. A Figura 8.20 mostra a nova função `dg_echo`. Quando terminamos o servidor com nossa tecla de interrupção no terminal `(SIGINT)`, ele imprime o número de datagramas recebidos e termina.

udpcliserv/dgcliloop1.c

```
 1 #include   "unp.h"

 2 #define NDG       2000           /* datagramas a enviar */
 3 #define DGLEN     1400           /* comprimento de cada datagrama */

 4 void
 5 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 6 {
 7     int    i;
 8     char   sendline[DGLEN];

 9     for (i = 0; i < NDG; i++) {
10         Sendto(sockfd, sendline, DGLEN, 0, pservaddr, servlen);
11     }
12 }
```

udpcliserv/dgcliloop1.c

Figura 8.19 função `dg_cli` que grava um número fixo de datagramas no servidor.

udpcliserv/dgecholoop1.c

```
 1 #include   "unp.h"

 2 static void recvfrom_int(int);
 3 static int count;
```

Figura 8.20 A função `dg_echo` que conta os datagramas recebidos (*continua*).

```
 4 void
 5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 6 {
 7     socklen_t len;
 8     char    mesg[MAXLINE];

 9     Signal(SIGINT, recvfrom_int);

10     for ( ; ; ) {
11         len = clilen;
12         Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

13         count++;
14     }
15 }

16 static void
17 recvfrom_int(int signo)
18 {
19     printf("\nreceived %d datagrams\n", count);
20     exit(0);
21 }
```

udpcliserv/dgecholoop1.c

Figura 8.20 A função `dg_echo` que conta os datagramas recebidos (*continuação*).

Agora executamos o servidor no host `freebsd`, uma lenta SPARCStation. Executamos o cliente no `aix` do sistema RS/6000, conectado diretamente a uma Ethernet de 100 Mbps. Além disso, executamos `netstat -s` no servidor, antes e depois, uma vez que as estatísticas produzidas informam quantos datagramas foram perdidos. A Figura 8.21 mostra a saída no servidor.

```
freebsd % netstat -s -p udp
udp:
        71208 datagrams received
        0 with incomplete header
        0 with bad data length field
        0 with bad checksum
        0 with no checksum
        832 dropped due to no socket
        16 broadcast/multicast datagrams dropped due to no socket
        1971 dropped due to full socket buffers
        0 not for hashed pcb
        68389 delivered
        137685 datagrams output
freebsd % udpserv06                    inicia nosso servidor
                                       executamos o cliente aqui
^C                                     digitamos nossa chave de interrupção depois
                                          que o cliente conclui
received 30 datagrams
freebsd % netstat -s -p udp
udp:
        73208 datagrams received
        0 with incomplete header
        0 with bad data length field
        0 with bad checksum
        0 with no checksum
        832 dropped due to no socket
        16 broadcast/multicast datagrams dropped due to no socket
        3941 dropped due to full socket buffers
        0 not for hashed pcb
        68419 delivered
        137685 datagrams output
```

Figura 8.21 Saída no host servidor.

O cliente enviou 2.000 datagramas, mas a aplicação servidora recebeu somente 30, para uma taxa de perda de 98%. Não há nenhuma indicação assim seja para a aplicação servidora ou cliente de que esses datagramas foram perdidos. Como dissemos, o UDP não tem nenhum controle de fluxo e é não-confiável. É trivial, como mostramos, para um emissor de UDP exceder (*overrun*) o receptor.

Se examinarmos a saída de `netstat`, o número total de datagramas recebido pelo host servidor (não pela aplicação servidora) é 2.000 (73.208 – 71.208). O contador "descartado devido a buffers de soquete cheios" (*dropped due to full socket buffers*) indica quantos datagramas foram recebidos pelo UDP mas descartados porque a fila de recebimento do soquete receptor estava cheia (página 775 do TCPv2). Esse valor é 1.970 (3.491 – 1.971), o que, quando adicionado à saída de contador pela aplicação (30), é igual aos 2.000 datagramas recebidos pelo host. Infelizmente, o contador `netstat` do número descartado devido a um buffer de soquete cheio afeta todo o sistema. Não há como determinar quais aplicações (por exemplo, quais portas UDP) são afetadas.

O número de datagramas recebidos pelo servidor nesse exemplo não é previsível. Ele depende de vários fatores, como a carga de rede, a carga de processamento no host cliente e a carga de processamento no host servidor.

Se executarmos o mesmo cliente e servidor, mas dessa vez com o cliente em um Sun lento e o servidor em um RS/6000 mais rápido, nenhum datagrama será perdido.

```
aix % udpserv06
^?                                  digitamos nossa tecla de interrupção depois que o cliente conclui
received 2000 datagrams
```

Buffer de recebimento de soquete UDP

O número de datagramas UDP que são enfileirados pelo UDP para um dado soquete é limitado pelo tamanho do buffer de recebimento desse soquete. Podemos alterar esse tamanho com a opção de soquete `SO_RCVBUF`, da maneira como descrevemos na Seção 7.5. O tamanho-padrão do buffer de recebimento de soquete UDP no FreeBSD é 42.080 bytes, o que permite espaço somente para 30 dos nossos datagramas de 1.400 bytes. Se aumentarmos o tamanho do buffer de recebimento de soquete, esperamos que o servidor receba datagramas adicionais. A Figura 8.22 mostra uma modificação na função `dg_echo` da Figura 8.20 que configura o buffer de recebimento de soquete como 240 KB.

udpcliserv/dgecholoop2.c

```
 1 #include   "unp.h"

 2 static void recvfrom_int(int);
 3 static int count;

 4 void
 5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 6 {
 7     int    n;
 8     socklen_t len;
 9     char   mesg[MAXLINE];

10     Signal(SIGINT, recvfrom_int);

11     n = 220 * 1024;
12     Setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &n, sizeof(n));

13     for ( ; ; ) {
14         len = clilen;
15         Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
```

Figura 8.22 A função `dg_echo` que aumenta o tamanho da fila de recebimento do soquete (*continua*).

```
16          count++;
17      }
18  }
19  static void
20  recvfrom_int(int signo)
21  {
22      printf("\nreceived %d datagrams\n", count);
23      exit(0);
24  }
```

udpcliserv/dgecholoop2.c

Figura 8.22 A função `dg_echo` que aumenta o tamanho da fila de recebimento do soquete (*continuação*).

Se executarmos esse servidor no Sun e o cliente no RS/6000, a contagem dos datagramas recebidos será 103. Embora isso seja um pouco melhor que o exemplo anterior com o buffer de recebimento de soquete de recebimento, não é nenhuma panacéia.

> Por que configuramos o buffer de recebimento de soquete como 220 x 1.024 na Figura 8.22? O tamanho máximo de um buffer de recebimento de soquete no FreeBSD 5.1 assume default de 262.144 bytes (256 x 1.024), mas, devido à diretiva de alocação de buffer (descrita no Capítulo 2 do TCPv2), o limite real é 233.016 bytes. Muitos sistemas iniciais baseados no 4.3BSD restringiram o tamanho de um buffer de soquete a aproximadamente 52.000 bytes.

8.14 Determinando a interface de saída com UDP

Um soquete UDP conectado também pode ser utilizado para determinar a interface de saída que será utilizada para um destino particular. Isso ocorre devido a um efeito colateral da função `connect` quando aplicada a um soquete UDP: o kernel escolhe o endereço IP local (assumindo que o processo ainda não chamou `bind` explicitamente para atribuir esse endereço). Esse endereço IP local é escolhido pesquisando o endereço IP de destino na tabela de roteamento e então utilizando o endereço IP primário para a interface resultante.

A Figura 8.23 mostra um programa UDP simples que chama `connect` para conectar-se a um endereço IP especificado e então chama `getsockname`, imprimindo o endereço IP e a porta locais.

udpcliserv/udpcli09.c

```
 1  #include   "unp.h"
 2  int
 3  main(int argc, char **argv)
 4  {
 5      int    sockfd;
 6      socklen_t len;
 7      struct sockaddr_in cliaddr, servaddr;
 8      if (argc != 2)
 9          err_quit("usage: udpcli <IPaddress>");
10      sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
11      bzero(&servaddr, sizeof(servaddr));
12      servaddr.sin_family = AF_INET;
13      servaddr.sin_port = htons(SERV_PORT);
14      Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
15      Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
16      len = sizeof(cliaddr);
```

Figura 8.23 Programa UDP que utiliza `connect` para determinar a interface de saída (*continua*).

```
17      Getsockname(sockfd, (SA *) &cliaddr, &len);
18      printf("local address %s\n", Sock_ntop((SA *) &cliaddr, len));

19      exit(0);
20  }
```

udpcliserv/udpcli09.c

Figura 8.23 Programa UDP que utiliza `connect` para determinar a interface de saída (*continuação*).

Se executarmos o programa no host `freebsd` multihomed, teremos a seguinte saída:

```
freebsd % udpcli09 206.168.112.96
local address 12.106.32.254:52329

freebsd % udpcli09 192.168.42.2
local address 192.168.42.1:52330

freebsd % udpcli09 127.0.0.1
local address 127.0.0.1:52331
```

Na primeira vez que executamos o programa, o argumento de linha de comando é um endereço IP que segue a rota default. O kernel atribui o endereço IP local ao endereço primário da interface à qual a rota default aponta. Na segunda vez, o argumento é o endereço IP de um sistema conectado a uma segunda interface Ethernet, dessa forma o kernel atribui o endereço IP local ao endereço primário dessa segunda interface. Chamar `connect` em um soquete UDP não envia nada a esse host; é uma operação inteiramente local que salva o endereço IP e a porta do peer. Também vemos que chamar `connect` em um soquete UDP não-vinculado ainda atribui uma porta efêmera ao soquete.

> Infelizmente, essa técnica não funciona em todas as implementações, principalmente kernels derivados do SVR4. Por exemplo, essa técnica não funciona no Solaris 2.5, mas funciona no AIX, no HP-UX 11, no MacOS X, no FreeBSD, no Linux e no Solaris 2.6 e superior.

8.15 Servidor de eco de TCP e UDP utilizando `select`

Agora, combinamos nosso servidor TCP concorrente de eco do Capítulo 5 ao nosso servidor UDP iterativo de eco deste capítulo em um único servidor que utiliza `select` para multiplexar um soquete TCP e um UDP. A Figura 8.24 é a primeira metade desse servidor.

Criação do soquete TCP ouvinte

14-22 Um soquete TCP ouvinte é criado e vinculado à porta bem-conhecida do servidor. Ativamos a opção de soquete `SO_REUSEADDR` se houver conexões nessa porta.

Criação do soquete UDP

23-29 Um soquete UDP também é criado e vinculado à mesma porta. Mesmo que a mesma porta seja utilizada pelos soquetes TCP e UDP, não há necessidade de configurar a opção de soquete `SO_REUSEADDR` antes dessa chamada a `bind`, porque as portas TCP são independentes das UDP.

A Figura 8.25 mostra a segunda metade do nosso servidor.

Estabelecimento de um handler de sinal para `SIGCHLD`

30 Um handler de sinal é estabelecido para `SIGCHLD` porque as conexões TCP serão tratadas por um processo-filho. Mostramos esse handler de sinal na Figura 5.11.

Preparando tudo para `select`

31-32 Inicializamos um conjunto de descritores para `select` e calculamos o valor máximo dos dois descritores que iremos esperar.

udpcliserv/udpservselect01.c

```
 1 #include "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    listenfd, connfd, udpfd, nready, maxfdp1;
 6     char   mesg[MAXLINE];
 7     pid_t  childpid;
 8     fd_set rset;
 9     ssize_t n;
10     socklen_t len;
11     const int on = 1;
12     struct sockaddr_in cliaddr, servaddr;
13     void   sig_chld(int);
14         /* para criar soquete TCP ouvinte */
15     listenfd = Socket(AF_INET, SOCK_STREAM, 0);
16     bzero(&servaddr, sizeof(servaddr));
17     servaddr.sin_family = AF_INET;
18     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
19     servaddr.sin_port = htons(SERV_PORT);
20     Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
21     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
22     Listen(listenfd, LISTENQ);
23         /* para criar soquete UDP ouvinte */
24     udpfd = Socket(AF_INET, SOCK_DGRAM, 0);
25     bzero(&servaddr, sizeof(servaddr));
26     servaddr.sin_family = AF_INET;
27     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
28     servaddr.sin_port = htons(SERV_PORT);
29     Bind(udpfd, (SA *) &servaddr, sizeof(servaddr));
```

udpcliserv/udpservselect01.c

Figura 8.24 A primeira metade do servidor de eco que trata TCP e UDP utilizando `select`.

Chamando `select`

34-41 Chamamos `select`, esperando somente pela legibilidade no soquete TCP ouvinte ou no soquete UDP. Como nosso handler `sig_chld` pode interromper nossa chamada a `select`, tratamos um erro de EINTR.

Tratando a nova conexão de cliente

42-51 Chamamos `accept` para aceitar uma nova conexão de cliente quando o soquete TCP ouvinte é legível, utilizamos `fork` para bifurcar um filho e chamamos nossa função `str_echo` no filho. Essa é a mesma seqüência de passos que utilizamos no Capítulo 5.

udpcliserv/udpservselect01.c

```
30     Signal(SIGCHLD, sig_chld);   /* deve chamar waitpid() */
31     FD_ZERO(&rset);
32     maxfdp1 = max(listenfd, udpfd) + 1;
```

Figura 8.25 Segunda metade do servidor de eco que trata TCP e UDP utilizando `select` (*continua*).

```
33      for ( ; ; ) {
34          FD_SET(listenfd, &rset);
35          FD_SET(udpfd, &rset);
36          if ( (nready = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
37              if (errno == EINTR)
38                  continue;          /* de volta para for()*/
39              else
40                  err_sys("select error");
41          }

42          if (FD_ISSET(listenfd, &rset)) {
43              len = sizeof(cliaddr);
44              connfd = Accept(listenfd, (SA *) &cliaddr, &len);

45              if ( (childpid = Fork()) == 0) { /* processo-filho */
46                  Close(listenfd);      /* fecha o soquete ouvinte */
47                  str_echo(connfd);     /* processa a solicitação */
48                  exit(0);
49              }
50              Close(connfd);         /* pai fecha o soquete conectado */
51          }

52          if (FD_ISSET(udpfd, &rset)) {
53              len = sizeof(cliaddr);
54              n = Recvfrom(udpfd, mesg, MAXLINE, 0, (SA *) &cliaddr, &len);

55              Sendto(udpfd, mesg, n, 0, (SA *) &cliaddr, len);
56          }
57      }
58  }
```

udpcliserv/udpservselect01.c

Figura 8.25 Segunda metade do servidor de eco que trata TCP e UDP utilizando `select` (*continuação*).

Tratando a chegada de datagrama

52-57 Se o soquete UDP estiver legível, um datagrama chegou. Lemos esse datagrama com `recvfrom` e o enviamos de volta ao cliente com `sendto`.

8.16 Resumo

Converter nosso cliente/servidor de eco para utilizar UDP em vez de TCP foi simples. Mas está faltando uma grande quantidade de recursos fornecidos pelo TCP: detectar e retransmitir pacotes perdidos, verificar respostas como oriundas do peer correto e outros. Retornaremos a esse tópico na Seção 22.5 e veremos o que é necessário para adicionar alguma confiabilidade a uma aplicação UDP.

Os soquetes UDP podem gerar erros assíncronos, isto é, erros que são informados um pouco depois que um pacote é enviado. Os soquetes TCP sempre informam esses erros à aplicação, mas com o UDP o soquete precisa estar conectado para receber esses erros.

O UDP não tem nenhum controle de fluxo e isso é fácil de demonstrar. Normalmente, isso não é um problema, pois muitas aplicações UDP são construídas utilizando um modelo de solicitação e resposta e não para transferir dados em volume.

Ainda há outros pontos a serem considerados ao escrever aplicações UDP, mas iremos adiar essa discussão até o Capítulo 22, depois de abordar as funções de interface, broadcasting e multicasting.

Exercícios

8.1 Temos duas aplicações, uma utilizando TCP e outra UDP. 4.096 bytes estão no buffer de recebimento para o soquete TCP e dois datagramas de 2.048 bytes estão no buffer de recebimento para o soquete UDP. A aplicação TCP chama `read` com um terceiro argumento de 4.096 e a aplicação UDP chama `recvfrom` com um terceiro argumento de 4.096. Há alguma diferença?

8.2 O que acontece na Figura 8.4 se substituirmos o argumento final para `sendto` (que mostramos como `len`) por `clilen`?

8.3 Compile e execute o servidor UDP das Figuras 8.3 e 8.4 e então o cliente UDP das Figuras 8.7 e 8.8. Verifique se o cliente e o servidor funcionam conjuntamente.

8.4 Execute o programa `ping` em uma janela, especificando a opção `-i 60` (envia um pacote a cada 60 segundos; alguns sistemas utilizam `-I` em vez de `-i`), a opção `-v` (imprime todos os erros ICMP recebidos) e o endereço de loopback (normalmente 127.0.0.1). Utilizaremos esse programa para ver o ICMP para porta inacessível retornado pelo host servidor. Em seguida, execute nosso cliente do exercício anterior em uma outra janela, especificando o endereço IP de algum host que não está executando o servidor. O que acontece?

8.5 Dissemos na Figura 8.5 que cada soquete TCP conectado tem seu próprio buffer de recebimento de soquete. E quanto ao soquete ouvinte, você acha que ele tem seu próprio buffer de recebimento de soquete?

8.6 Utilize o programa `sock` (Seção C.3) e uma ferramenta como `tcpdump` (Seção C.5) para testar o que afirmamos na Seção 8.10: se o cliente chamar `bind` para vincular um endereço IP ao seu soquete, mas enviar um datagrama que sai de alguma outra interface, o datagrama resultante de IP ainda irá conter o endereço IP que foi vinculado ao soquete, mesmo que isso não corresponda à interface de saída.

8.7 Compile os programas da Seção 8.13 e execute o cliente e o servidor em hosts diferentes. Coloque um `printf` no cliente toda vez que um datagrama for gravado no soquete. Isso altera a percentagem de pacotes recebidos? Por quê? Coloque um `printf` no servidor toda vez que um datagrama é lido no soquete. Isso altera a percentagem de pacotes recebidos? Por quê?

8.8 Qual é o maior comprimento que podemos passar a `sendto` em um soquete UDP/IPv4, isto é, qual é o maior volume de dados que pode caber em um datagrama UDP/IPv4? O que muda com o UDP/IPv6?

Modifique a Figura 8.8 para enviar um datagrama UDP de tamanho máximo, leia esse datagrama novamente e imprima o número de bytes retornado por `recvfrom`.

8.9 Modifique a Figura 8.25 para se adaptar à RFC 1122 utilizando `IP_RECVDSTADDR` para o soquete UDP.

Capítulo 9

Soquetes SCTP Elementares

9.1 Visão geral

O SCTP é um protocolo de transporte mais recente, padronizado pelo IETF em 2000 (em comparação com o TCP, padronizado em 1981). Ele foi inicialmente projetado para atender às necessidades do crescente mercado de telefonia IP; em particular, o transporte de sinalização de telefonia pela Internet. Os requisitos para os quais ele foi projetado estão descritos na RFC 2719 (Ong *et al*., 1999). O SCTP é um protocolo confiável orientado a mensagens que fornece múltiplos fluxos entre as extremidades e suporte no nível de transporte a multihoming. Como o SCTP é um protocolo de transporte mais recente, não tem a mesma onipresença do TCP ou do UDP; entretanto, ele fornece alguns novos recursos que podem simplificar certos projetos de aplicação. Discutiremos as razões da utilização do SCTP em vez do TCP na Seção 23.12.

Apesar de haver algumas diferenças fundamentais entre o SCTP e o TCP, a interface de *um para um* do SCTP fornece quase a mesma interface de aplicação do TCP. Isso torna trivial portar aplicações, mas não permite utilizar alguns recursos avançados do SCTP. A interface de *um para muitos* fornece suporte completo a esses recursos, mas talvez exija uma revisão e adaptação significativa das aplicações existentes. A interface de *um para muitos* é recomendada para a maioria das novas aplicações desenvolvidas para o SCTP.

Este capítulo descreve as funções `socket` básicas adicionais que podem ser utilizadas com o SCTP. Primeiro, descreveremos os dois diferentes modelos de interface disponíveis ao desenvolvedor de aplicações. Desenvolveremos uma versão do nosso servidor de eco utilizando o modelo de um para muitos no Capítulo 10. Também descreveremos as novas funções disponíveis e utilizadas exclusivamente com o SCTP. Examinaremos a função `shutdown` e como sua utilização no SCTP difere daquela no TCP. Em seguida, abordaremos brevemente o uso de *notificações* no SCTP. As notificações permitem que uma aplicação seja informada sobre eventos de protocolo significativos além da chegada de dados de usuário. Veremos um exemplo sobre como utilizar as notificações na Seção 23.4.

Como o SCTP é um protocolo mais recente, a interface para todos os seus recursos ainda não é completamente estável. Quando este livro estava sendo escrito, acreditava-se que as interfaces descritas eram estáveis, mas ainda não eram tão onipresentes quanto o restante da API

de soquetes. Os usuários de aplicações projetadas para utilizar exclusivamente o SCTP talvez precisem estar preparados para instalar patches de kernel ou, de outro modo, atualizar seus sistemas operacionais; e as aplicações que precisam ser onipresentes têm de ser capazes de utilizar o TCP se o SCTP não estiver disponível no sistema em que elas são executadas.

9.2 Modelos de interface

Há dois tipos de soquetes SCTP: um soquete de *um para um* e um soquete de *um para muitos*. Um soquete de um para um corresponde exatamente a uma associação SCTP. (Lembre-se, da Seção 2.5, de que uma associação SCTP é uma conexão entre dois sistemas, mas pode envolver mais de dois endereços IP devido ao multihoming.) Esse mapeamento é semelhante ao relacionamento entre um soquete TCP e uma conexão TCP. Com um soquete de um para muitos, várias associações SCTP podem ser ativadas em um dado soquete simultaneamente. Esse mapeamento é semelhante à maneira como um soquete UDP vinculado a uma porta particular pode receber datagramas intercalados a partir de várias extremidades UDP remotas que enviam dados simultaneamente.

Ao decidir sobre o estilo de interface a utilizar, a aplicação precisa levar em consideração vários fatores, incluindo:

- Que tipo de servidor está sendo escrito, *iterativo* ou *concorrente?*
- Quantos descritores de soquete o servidor deseja gerenciar?
- É importante otimizar a configuração de associação para permitir dados no terceiro (e possivelmente quarto) pacote do handshake de quatro vias?
- Quantos estados de conexão a aplicação deseja manter?

> Quando a API de soquetes para SCTP estava em desenvolvimento, uma terminologia diferente foi utilizada para dois estilos de soquetes e, algumas vezes, os leitores podem encontrar esses termos mais antigos na documentação ou no código-fonte. O termo original para o soquete de um para um era soquete "estilo TCP", e o termo original para um soquete de um para muitos era soquete "estilo UDP".
>
> Esses estilos diferentes foram, mais tarde, descartados porque tendiam a causar confusão criando expectativas de que o SCTP iria se comportar mais como o TCP ou UDP, dependendo de qual estilo de soquete era utilizado. De fato, esses termos chamavam a atenção somente para um aspecto das diferenças entre soquetes TCP e UDP (isto é, se um soquete suporta múltiplas associações concorrentes na camada de transporte). A terminologia atual ("um para um" *versus* "um para muitos") se concentra na diferença-chave entre os dois estilos de soquete. Por fim, observe que alguns autores utilizam o termo "muitos para um" em vez de "um para muitos"; esses termos são intercambiáveis.

Estilo um para um

O estilo um para um foi desenvolvido para facilitar portar aplicações TCP existentes para o SCTP. Esse estilo fornece um modelo quase idêntico àquele descrito no Capítulo 4. Há algumas diferenças que devem ser observadas, especialmente ao portar aplicações TCP existentes para o SCTP.

1. Quaisquer opções de soquete devem ser convertidas no equivalente de SCTP. Duas opções comumente encontradas são `TCP_NODELAY` e `TCP_MAXSEG`. Essas opções podem ser facilmente mapeadas para `SCTP_NODELAY` e `SCTP_MAXSEG`.
2. O SCTP preserva o limite de mensagens; portanto, um limite de mensagens na camada de aplicativo não é requerido. Por exemplo, um protocolo de aplicação baseado em TCP poderia utilizar uma chamada de sistema `write()` para gravar um campo *x* de dois bytes contendo o comprimento da mensagem, seguido por outra chamada de sistema `write()` que gravasse *x* bytes de dados. Entretanto, se isso for feito com

o SCTP, o SCTP receptor receberá duas mensagens separadas (isto é, a chamada para `read` irá retornar duas vezes: uma vez com uma mensagem de dois bytes e outra com uma mensagem de *x* bytes).

3. Algumas aplicações TCP utilizam um half-close (meio fechamento) para sinalizar o fim da entrada para a outra extremidade. Para portar essas aplicações para o SCTP, o protocolo da camada de aplicativo precisará ser reescrito de modo que a aplicação sinalize o final da entrada no fluxo de dados da aplicação.
4. A função `send` pode ser utilizada da maneira normal. Para as funções `sendto` e `sendmsg`, quaisquer informações de endereço incluídas são tratadas como sobrescrevendo o endereço primário de destino (consulte a Seção 2.8).

Um usuário típico do estilo um para um irá seguir a linha do tempo mostrada na Figura 9.1. Quando o servidor é iniciado, ele abre e vincula um soquete a um endereço e espera uma conexão de cliente com a chamada de sistema `accept`. Posteriormente, o cliente é iniciado, abre um soquete e inicia uma associação com o servidor. Vamos supor que o cliente envia uma

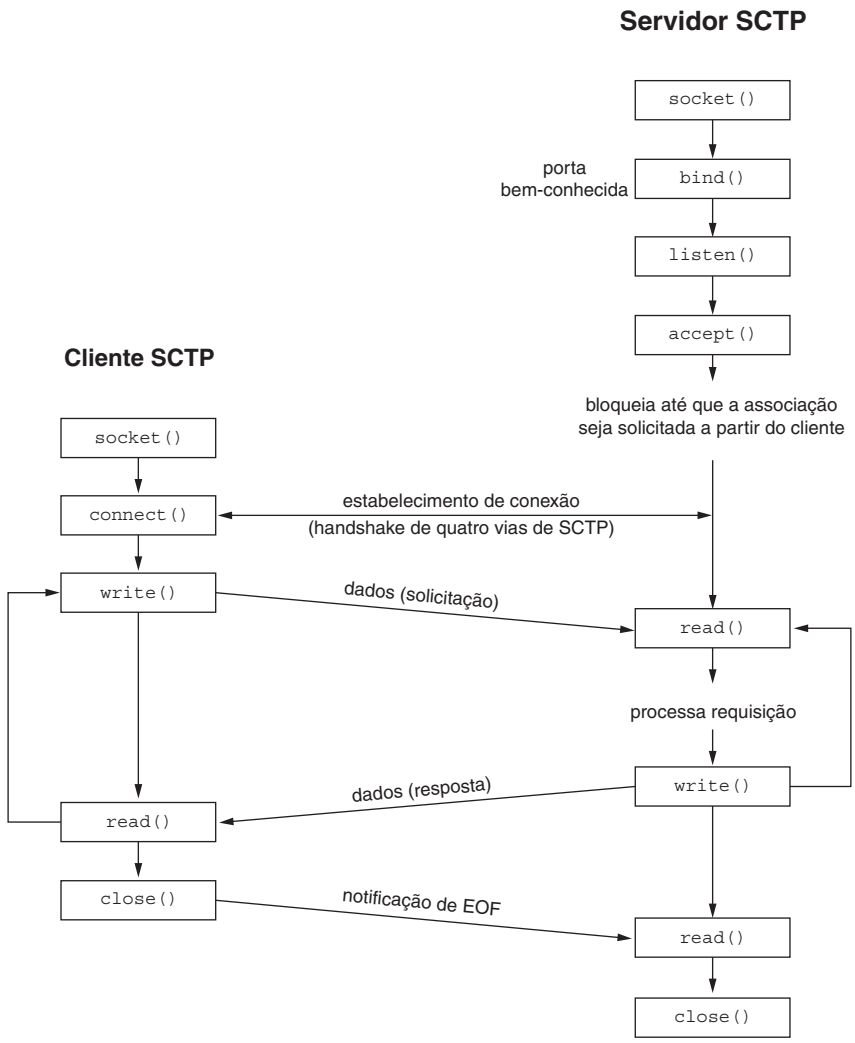


Figura 9.1 Funções de soquete para o estilo um para um do SCTP.

solicitação ao servidor, o servidor processa a solicitação e envia de volta uma resposta ao cliente. Esse ciclo continua até que o cliente inicie uma desativação de associação. Essa ação fecha a associação e então o servidor sai ou espera uma nova associação. Como pode ser visto comparando uma troca TCP típica, uma troca de um soquete de um para um para um SCTP ocorre de maneira semelhante à mostrada na Figura 4.1.

Um soquete SCTP no estilo um para um é um soquete IP (família `AF_INET` ou `AF_INET6`), com o tipo `SOCK_STREAM` e o protocolo `IPPROTO_SCTP`.

Estilo um para muitos

O estilo um para muitos fornece ao autor de uma aplicação a capacidade de escrever um servidor sem gerenciar um grande número de descritores de soquetes. Um único descritor de soquete representará múltiplas associações, quase da mesma maneira como um soquete UDP pode receber mensagens de múltiplos clientes. Um identificador de associação é utilizado para identificar uma única associação no estilo de soquete de um para muitos. Esse identificador de associação é um valor do tipo `sctp_assoc_t`; normalmente um inteiro. Ele é um valor obscuro; uma aplicação não deve utilizar um identificador de associação que não tenha sido previamente fornecido pelo kernel. Os usuários do estilo um para muitos devem ter as seguintes questões em mente:

1. Quando o cliente fecha a associação, o lado do servidor também a fechará automaticamente, removendo assim qualquer estado para ela dentro do kernel.
2. O estilo um para muitos é o único método que pode ser utilizado para transferir dados via *piggyback* no terceiro ou quarto pacote do handshake de quatro vias (veja o Exercício 9.3).
3. Qualquer `sendto`, `sendmsg` ou `sctp_sendmsg` para um endereço ao qual ainda não há uma associação fará com que um open ativo seja tentado, criando assim (se bem-sucedido) uma nova associação com esse endereço. Tal comportamento ocorrerá mesmo que a aplicação que faz o `send` chame a função `listen` para solicitar um open passivo.
4. O usuário deve utilizar as funções `sendto`, `sendmsg` ou `sctp_sendmsg` e não pode utilizar as funções `send` ou `write`. (Se a função `sctp_peeloff` for utilizada para criar um soquete no estilo um para um, `send` ou `write` pode ser utilizado nela.)
5. A qualquer momento em que uma das funções `send` é chamada, o endereço primário de destino escolhido pelo sistema no início da associação (Seção 2.8) será utilizado a menos que o flag `MSG_ADDR_OVER` seja configurado pelo chamador em uma estrutura `sctp_sndrcvinfo` fornecida. Para fornecer essa estrutura, o chamador precisa utilizar a função `sendmsg` com dados auxiliares ou a função `sctp_sendmsg`.
6. Os eventos de associação (uma das várias notificações SCTP discutidas na Seção 9.14) podem ser ativados, de modo que, se uma aplicação não deseja receber esses eventos, ela deverá desativá-los explicitamente utilizando a opção de soquete `SCTP_EVENTS`. Por default, o único evento que é ativado é o `sctp_data_io_event`, que fornece dados auxiliares à chamada `recvmsg` e `sctp_recvmsg`. Essa configuração-padrão é aplicada tanto ao estilo um para um como um para muitos.

   Quando a API de soquetes SCTP foi inicialmente desenvolvida, o estilo de interface um para muitos também foi definido para ter a notificação de associação ativada por default. As versões posteriores do documento sobre a API desativaram desde então todas as notificações, exceto `sctp_data_io_event` tanto para o estilo de interface um para um

como um para muitos. Entretanto, nem todas as implementações talvez apresentem esse comportamento. Sempre é uma boa prática para o autor da aplicação desativar explicitamente (ou ativar) as notificações não-desejadas (ou desejadas). Essa abordagem explícita garante ao desenvolvedor que o resultado do comportamento esperado ocorrerá independentemente de para qual SO o código é portado.

Uma linha do tempo típica para o estilo um para muitos é mostrada na Figura 9.2. Primeiro, o servidor é iniciado, cria um soquete e o vincula a um endereço, chama `listen` para ativar as associações de cliente e chama `sctp_recvmsg`, que bloqueia esperando que a primeira mensagem chegue. Um cliente abre um soquete e chama `sctp_sendto`, que configura implicitamente a associação e transfere via *piggyback* a solicitação de dados ao servidor no terceiro pacote do handshake de quatro vias. O servidor recebe e processa a solicitação e envia de volta uma resposta. O cliente recebe a resposta e fecha o soquete, fechando assim a associação. O servidor retorna ao início do loop para receber a próxima mensagem.

Esse exemplo mostra um servidor iterativo, em que as mensagens (possivelmente intercaladas) provenientes de várias associações (isto é, vários clientes) podem ser processadas por um thread simples de controle. Com o SCTP, um soquete de um para muitos também pode ser utilizado em conjunção com a função `sctp_peeloff` (consulte a Seção 9.12) para permitir que modelos de servidores iterativos e concorrentes sejam combinados como a seguir:

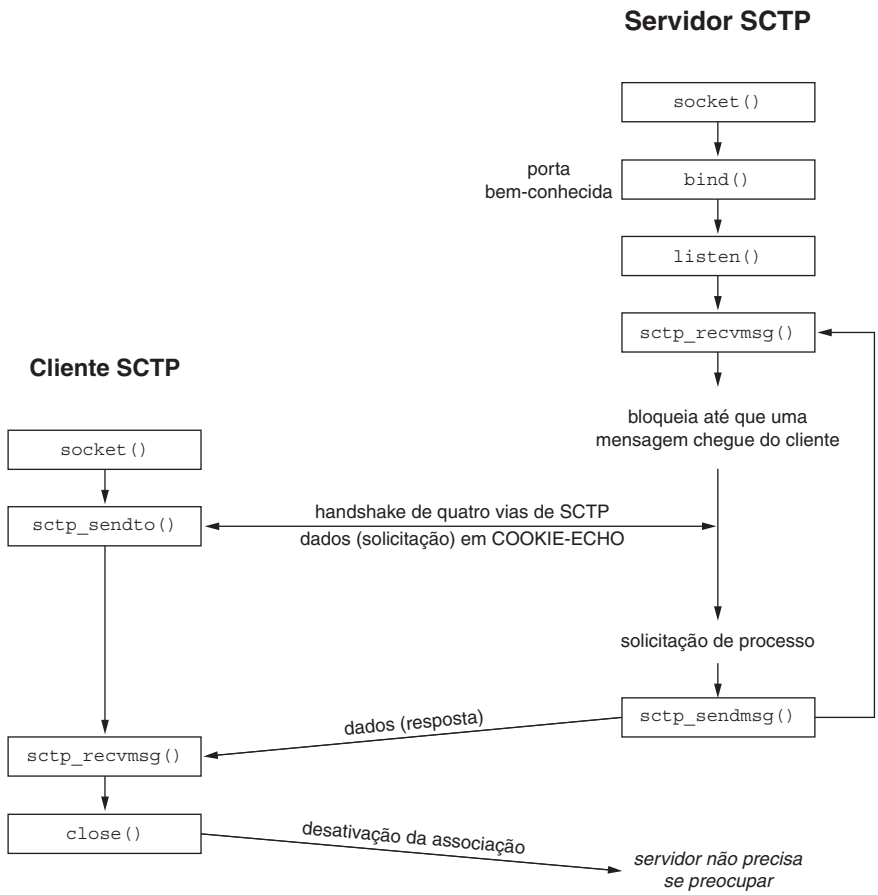


Figura 9.2 Funções socket para o estilo um para muitos SCTP.

1. A função `sctp_peeloff` pode ser utilizada para "extrair" uma associação particular (por exemplo, uma sessão de longa duração) a partir de um soquete de um para muitos para o seu próprio soquete de um para um.
2. O soquete de um para um da associação extraída pode então ser enviado ao próprio thread ou processo bifurcado (como ocorre no modelo concorrente).
3. Enquanto isso, o thread principal continua a tratar mensagens provenientes de quaisquer associações remanescentes, de maneira iterativa, no soquete original.

O soquete SCTP no estilo um para muitos é um soquete IP (família `AF_INET` ou `AF_INET6`) com o tipo `SOCK_SEQPACKET` e o protocolo `IPPROTO_SCTP`.

9.3 Função `sctp_bindx`

Um servidor SCTP talvez deseje vincular um subconjunto de endereços IP associado ao sistema host. Tradicionalmente, um servidor TCP ou UDP pode vincular um ou todos os endereços em um host, mas ele não pode vincular um subconjunto de endereços. A função `sctp_bindx` fornece mais flexibilidade, permitindo a um soquete SCTP vincular um subconjunto particular de endereços.

```
#include <netinet/sctp.h>

int sctp_bindx(int sockfd, const struct sockaddr *addrs, int addrcnt, int flags) ;
```

Retorna: 0 se OK, –1 em erro

O *sockfd* é um descritor de soquete retornado pela função `socket`. O segundo argumento, *addrs*, é um ponteiro para uma lista empacotada de endereços. Cada estrutura de endereço de soquete é colocada no buffer logo após a estrutura de endereço de soquete precedente, sem nenhum preenchimento no meio. Veja na Figura 9.4 um exemplo.

O número de endereços passado para `sctp_bindx` é especificado pelo parâmetro *addrcnt*. O parâmetro *flags* faz com que a chamada `sctp_bindx` realize uma das duas ações mostradas na Figura 9.3.

A chamada a `sctp_bindx` pode ser utilizada em um soquete vinculado ou desvinculado. Para um soquete desvinculado, uma chamada a `sctp_bindx` irá vincular o conjunto de endereços especificado ao descritor de soquete. Se `sctp_bindx` for utilizado em um soquete vinculado, a chamada pode ser utilizada com `SCTP_BINDX_ADD_ADDR` para associar endereços adicionais ao descritor de soquete ou com `SCTP_BINDX_REM_ADDR` para remover uma lista de endereços associados ao descritor de soquete. Se `sctp_bindx` for realizado em um soquete ouvinte, as associações futuras utilizarão a nova configuração de endereço; a alteração não afeta nenhuma associação existente. Os dois flags passados a `sctp_bindx` são mutuamente exclusivos; se os dois forem fornecidos, `sctp_bindx` irá falhar, retornando o código de erro `EINVAL`. O número de porta em todas as estruturas de endereço de soquete deve ser o mesmo e corresponder a qualquer número de porta que já está vinculado; se não corresponder, então `sctp_bindx` irá falhar, retornando o código de erro `EINVAL`.

flags	Descrição
`SCTP_BINDX_ADD_ADDR`	Adiciona o(s) endereço(s) ao soquete
`SCTP_BINDX_REM_ADDR`	Remove o(s) endereço(s) do soquete

Figura 9.3 *Flags* utilizados com a função `sctp_bindx`.

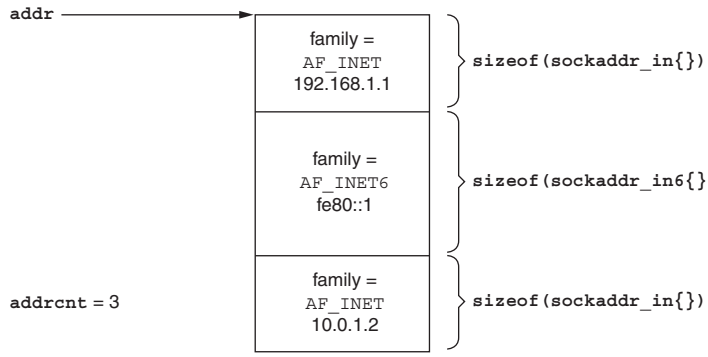


Figura 9.4 Formato de lista de endereços empacotados para chamadas SCTP.

Se uma extremidade suportar o recurso de endereço dinâmico, uma chamada a `sctp_bindx` com o flag `SCTP_BINDX_REM_ADDR` ou `SCTP_BINDX_ADD_ADDR` fará com que a extremidade envie uma mensagem apropriada ao peer para alterar a lista de endereços deste. Uma vez que adicionar e remover endereços de uma associação conectada é uma funcionalidade opcional, implementações que não suportam essa funcionalidade retornarão `EOPNOTSUPP`. Observe que, em uma operação adequada, as duas extremidades de uma associação devem suportar esse recurso. Esse recurso pode ser útil se o sistema suportar a provisão dinâmica de interfaces; por exemplo, se uma nova interface Ethernet for chamada, a aplicação poderá utilizar `SCTP_BINDX_ADD_ADDR` para começar a utilizar a interface adicional em uma conexão existente.

9.4 Função `sctp_connectx`

```
#include <netinet/sctp.h>

int sctp_connectx(int sockfd, const struct sockaddr *addrs, int addrcnt) ;
```

Retorna: 0 se bem-sucedida, –1 em erro

A função `sctp_connectx` é utilizada para conectar-se a um peer multihomed. Especificamos os endereços *addrcnt,* todos pertencentes ao mesmo peer, no parâmetro *addrs*. O parâmetro *addrs* é uma lista de endereços empacotados, como na Figura 9.4. A pilha SCTP utiliza um ou mais dos endereços especificados para estabelecer a associação. Todos os endereços listados em *addrs* são considerados válidos, confirmados.

9.5 Função `sctp_getpaddrs`

A função `getpeername` não foi projetada com o conceito de um protocolo de transporte "ciente de múltiplos endereços" (*multihoming-aware*); ao utilizar SCTP, ela retorna apenas o endereço primário. Se todos os endereços forem exigidos, a função `sctp_getpaddrs` fornecerá um mecanismo para que uma aplicação recupere todos os endereços de um peer.

```
#include <netinet/sctp.h>

int sctp_getpaddrs(int sockfd, sctp_assoc_t id, struct sockaddr **addrs) ;
```

Retorna: o número de endereços do peer armazenado em *addrs*, –1 em erro

O parâmetro *sockfd* é o descritor de soquete retornado pela função `socket`. O *id* é a identificação de associação para um soquete no estilo um para muitos. Se o soquete utilizar o estilo um para um, o campo *id* é ignorado. *addrs é* o endereço de um ponteiro que `sctp_getpaddrs` irá preencher com uma lista de endereços empacotados, alocada localmente. Veja as Figuras 9.4 e 23.12 para os detalhes sobre a estrutura desse valor de retorno. O chamador deve utilizar `sctp_freepaddrs` para liberar os recursos alocados por `sctp_getpaddrs` ao terminar de utilizá-los.

9.6 Função `sctp_freepaddrs`

A função `sctp_freepaddrs` libera os recursos alocados pela função `sctp_getpaddrs`. Ela é chamada desta maneira:

```
#include <netinet/sctp.h>

void sctp_freepaddrs(struct sockaddr *addrs) ;
```

addrs é o ponteiro para o array de endereços retornado por `sctp_getpaddrs`.

9.7 Função `sctp_getladdrs`

A função `sctp_getladdrs` pode ser utilizada para recuperar os endereços locais que fazem parte de uma associação. Freqüentemente, essa função é necessária quando uma extremidade local deseja saber exatamente quais endereços locais estão sendo utilizados (o que pode ser um subconjunto adequado dos endereços do sistema).

```
#include <netinet/sctp.h>

int sctp_getladdrs (int sockfd, sctp_assoc_t id, struct sockaddr **addrs) ;

                                Retorna: o número de endereços local armazenados em addrs, –1 em erro
```

O *sockfd* é o descritor de soquete retornado pela função `socket`. `id` é a identificação de associação para o soquete do estilo um para muitos. Se o soquete utilizar o estilo um para um, o campo *id* é ignorado. O parâmetro *addrs* é um endereço de um ponteiro que `sctp_getladdrs` irá preencher com uma lista de endereços empacotados, alocada localmente. Veja as Figuras 9.4 e 23.12 para os detalhes sobre a estrutura desse valor de retorno. O chamador deve utilizar `sctp_freeladdrs` para liberar os recursos alocados por `sctp_getladdrs` quando terminar.

9.8 Função `sctp_freeladdrs`

A função `sctp_freeladdrs` libera os recursos alocados por `sctp_getladdrs`. Ela é chamada desta maneira:

```
#include <netinet/sctp.h>

void sctp_freeladdrs(struct sockaddr *addrs) ;
```

addrs é o ponteiro para o array de endereços retornado por `sctp_getladdrs`.

9.9 Função `sctp_sendmsg`

Uma aplicação pode controlar vários recursos SCTP utilizando a função `sendmsg` junto com dados auxiliares (descrita no Capítulo 14). Entretanto, como o uso de dados auxiliares talvez não seja conveniente, muitas implementações SCTP fornecem uma chamada de biblioteca auxiliar (possivelmente implementada como uma chamada de sistema) que facilita a utilização dos recursos SCTP avançados pela aplicação. A chamada assume a seguinte forma:

```
ssize_t sctp_sendmsg(int sockfd, const void *msg, size_t msgsz,
                     const struct sockaddr *to, socklen_t tolen,
                     uint32_t ppid,
                     uint32_t flags, uint16_t stream,
                     uint32_t timetolive, uint32_t context) ;
```

Retorna: o número de bytes gravados, –1 em erro

O usuário de `sctp_sendmsg` tem um método de envio significativamente simplificado, mas com um número maior de argumentos. O campo *sockfd* armazena o descritor de soquete retornado de uma chamada de sistema `socket`. O campo *msg* aponta para um buffer com comprimento de *msgsz* bytes a serem enviados à extremidade *to* do peer. O campo *tolen* armazena o comprimento do endereço armazenado em *to*. O campo *ppid* armazena o identificador de protocolo de payload que será passado com o bloco de dados. O campo *flags* será passado para a pilha SCTP para identificar quaisquer opções SCTP; valores válidos para esse campo podem ser encontrados na Figura 7.16.

O chamador da função especifica o número do fluxo SCTP preenchendo o campo *stream*. O chamador pode especificar o tempo de vida da mensagem em milissegundos no campo *lifetime*, em que 0 representa um tempo de vida infinito. Um contexto de usuário, se existir, pode ser especificado em *context*. O contexto de usuário associa uma transmissão malsucedida de uma mensagem, recebida via uma notificação de mensagem, a algum contexto específico da aplicação local. Por exemplo, para enviar uma mensagem ao fluxo de número 1, com os flags de envio configurados como `MSG_PR_SCTP_TTL`, o tempo de vida como 1.000 milissegundos, um identificador de protocolo de payload de 24 e um contexto de 52, um usuário formularia a seguinte chamada:

```
ret = sctp_sendmsg(sockfd,
                   data, datasz, &dest, sizeof(dest),
                   24, MSG_PR_SCTP_TTL, 1, 1000, 52);
```

Essa abordagem é muito mais fácil do que alocar os dados auxiliares necessários e configurar as estruturas apropriadas na estrutura `msghdr`. Observe que, se uma implementação mapear `sctp_sendmsg` para uma chamada de função `sendmsg`, o campo *flags* da chamada `sendmsg` é configurado como 0.

9.10 Função `sctp_recvmsg`

Como ocorre com `sctp_sendmsg`, a função `sctp_recvmsg` fornece uma interface mais amigável ao usuário para os recursos SCTP avançados. O uso dessa função permite a um usuário recuperar não apenas o endereço do peer, mas também o campo `msg_flags` que normalmente acompanharia a chamada de função `recvmsg` (por exemplo, `MSG_NOTIFICATION`, `MSG_EOR`, etc.). Essa função também permite ao usuário recuperar a estrutura `sctp_sndrcvinfo` que acompanha a mensagem que foi lida para o buffer de mensagem. Observe que, se uma aplicação deseja receber as informações de `sctp_sndrcvinfo`, `sctp_data_io_event` deve ser inscrita com a opção de soquete `SCTP_EVENTS` (ON por default). A função `sctp_recvmsg` assume a seguinte forma:

```
ssize_t sctp_recvmsg(int sockfd, void *msg, size_t msgsz,
                     struct sockaddr *from, socklen_t *fromlen,
                     struct sctp_sndrcvinfo *sinfo,
                     int *msg_flags) ;

                                        Retorna: o número de bytes lidos, –1 em erro
```

No retorno a partir dessa chamada, *msg* é preenchido com até *msgsz* bytes de dados. O endereço do remetente da mensagem está contido em *from*, com o tamanho de endereço preenchido no argumento *fromlen*. Quaisquer flags de mensagem estarão contidos no argumento *msg_flags*. Se a notificação `sctp_data_io_event` estiver ativada (o default), a estrutura `sctp_sndrcvinfo` também será preenchida com informações detalhadas sobre a mensagem. Observe que, se uma implementação mapear `sctp_recvmsg` para uma chamada de função `recvmsg`, o campo *flags* da chamada será configurado como 0.

9.11 Função `sctp_opt_info`

A função `sctp_opt_info` é fornecida para implementações que não podem utilizar as funções `getsockopt` para SCTP. Essa incapacidade de utilizar a função `getsockopt` ocorre porque algumas opções de soquete SCTP, por exemplo, `SCTP_STATUS`, precisam de uma variável in-out para passar a identificação de associação. Para sistemas que não podem fornecer uma variável in-out à função `getsockopt`, o usuário precisará utilizar `sctp_opt_info`. Para sistemas como o FreeBSD, que permite variáveis in-out na chamada de opção de soquete, a chamada `sctp_opt_info` é uma chamada de biblioteca que reempacota os argumentos na chamada `getsockopt` apropriada. Para portabilidade, as aplicações devem utilizar `sctp_opt_info` para todas as opções que requerem variáveis in-out (Seção 7.10).

A chamada tem o seguinte formato:

```
int sctp_opt_info(int sockfd, sctp_assoc_t assoc_id, int opt,
                  void *arg, socklen_t *siz) ;

                                        Retorna: 0 se bem-sucedida, –1 em erro
```

sockfd é o descritor de soquete que o usuário gostaria que a opção de soquete afetasse. *assoc_id* é a identificação da associação (se houver alguma) em que o usuário realiza a opção. *opt* é a opção de soquete (como definida na Seção 7.10) para o SCTP. *arg* é o argumento de opção de soquete, e *siz* é um ponteiro para um *socklen_t* que armazena o tamanho do argumento.

9.12 Função `sctp_peeloff`

Como mencionado anteriormente, é possível extrair uma associação contida por um soquete de um para muitos para um soquete individual no estilo um para um. A semântica é quase idêntica à chamada de função `accept` com um argumento adicional. O chamador passa o *sockfd* do soquete de um para muitos e o id de identificação de associação que está sendo extraído. Ao término da chamada, um novo descritor de soquete é retornado. Esse novo descritor será no estilo um para um com a associação solicitada. A função assume a seguinte forma:

```
int sctp_peeloff (int sockfd, sctp_assoc_t id) ;
```

Retorna: um novo descritor de soquete se bem-sucedido, –1 em erro

9.13 Função `shutdown`

A função `shutdown`, discutida na Seção 6.6, pode ser utilizada com uma extremidade SCTP utilizando a interface no estilo um para um. Como o *design* do SCTP não fornece um estado "meio fechado" (*half-closed*), uma extremidade SCTP reage a uma chamada `shutdown` diferentemente de uma extremidade de TCP. Quando uma extremidade SCTP inicia uma seqüência de `shutdown`, as duas extremidades devem completar a transmissão de quaisquer dados atualmente na fila e fechar a associação. A extremidade que iniciou o open ativo talvez deseje invocar `shutdown` em vez de `close`, de modo que a extremidade possa ser utilizada para se conectar a um novo peer. Diferentemente do TCP, não é requerido um `close` seguido pela abertura de um novo soquete. O SCTP permite à extremidade emitir um `shutdown` e, depois que este conclui, a extremidade pode reutilizar o soquete para se conectar a um novo peer. Observe que a nova conexão irá falhar se a extremidade não esperar até que a seqüência de `shutdown` do SCTP esteja concluída. A Figura 9.5 mostra as chamadas de função típicas nesse cenário.

Observe, na Figura 9.5, que mostramos o usuário recebendo os eventos `MSG_NOTIFICATION`. Se o usuário não tivesse se inscrito para receber esses eventos, uma leitura de comprimento 0 teria sido retornada. Os efeitos da função `shutdown` para o TCP foram descritos na Seção 6.6. A função `shutdown` segue a semântica para SCTP a seguir:

`SHUT_RD` A mesma semântica do TCP discutida na Seção 6.6; nenhuma ação de protocolo SCTP ocorre.

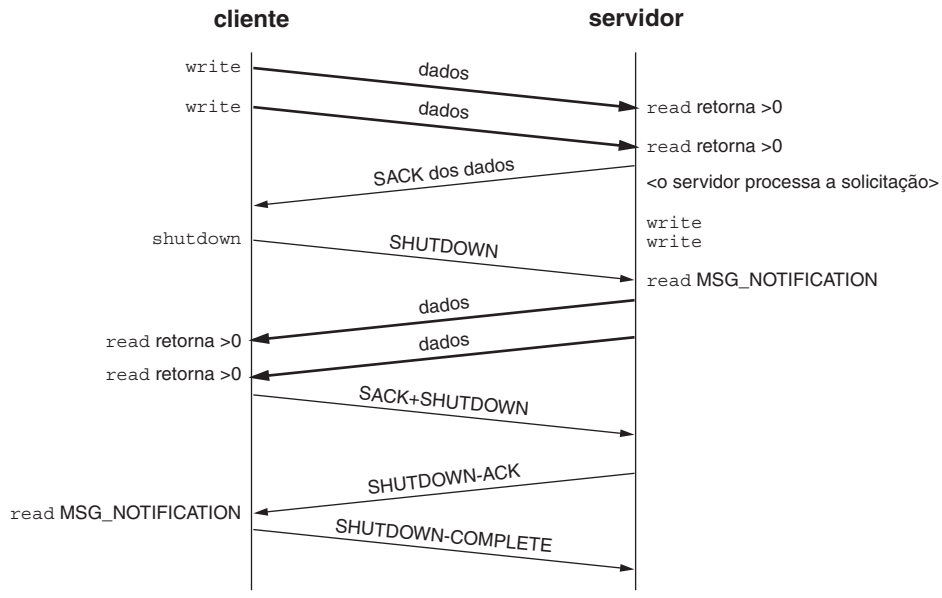


Figura 9.5 Chamando `shutdown` para fechar uma associação SCTP.

SHUT_WR Desativa outras operações de envio e inicia os procedimentos de desativação de SCTP, que terminarão a associação. Observe que essa opção não fornece um estado meio fechado, mas permite à extremidade local ler quaisquer dados enfileirados que o peer possa ter enviado antes de receber a mensagem SCTP_SHUTDOWN.

SHUT_RDWR Desativa todas as operações read e write e inicia o procedimento de desativação de SCTP. Quaisquer dados enfileirados em trânsito para a extremidade local serão reconhecidos e então descartados silenciosamente.

9.14 Notificações

O SCTP torna várias notificações disponíveis ao programador da aplicação. O usuário SCTP pode monitorar o estado da(s) sua(s) associação(s) via essas notificações. As notificações comunicam eventos no nível de transporte, incluindo a alteração do status de rede, inicializações de associação, erros operacionais remotos e mensagens não-entregáveis. Para ambos os estilos, um para um e um para muitos, todos os eventos são desativados por default, com exceção de sctp_data_io_event. Veremos um exemplo da utilização de notificações na Seção 23.7.

Oito eventos podem ser inscritos para utilização da opção de soquete SCTP_EVENTS. Sete desses eventos geram dados adicionais – denominados notificação – que um usuário receberá via o descritor de soquete normal. As notificações são adicionadas ao descritor de soquete em linha com os dados à medida que ocorrem os eventos que os geram. Ao ler um soquete com inscrições de notificação, os dados de usuário e as notificações serão intercalados no buffer de soquete. Para diferenciar entre dados do peer e uma notificação, o usuário utiliza a função recvmsg ou a função sctp_recvmsg. Quando os dados retornados são uma notificação de evento, o campo msg_flags dessas duas funções conterá o flag MSG_NOTIFICATION. Esse flag informa à aplicação que a mensagem recém-lida não são os dados do peer, mas uma notificação proveniente da pilha de SCTP local.

Cada tipo de notificação está na forma tag-comprimento-valor, em que os primeiros oito bytes da mensagem identificam o tipo e o comprimento total da notificação que chegou. Ativar o evento sctp_data_io_event causa a recepção das estruturas sctp_sndrcvinfo em cada leitura de dados do usuário (essa opção é ativada por default para os dois estilos de interface). Normalmente, essas informações são recebidas nos dados auxiliares utilizando a função recvmsg. Uma aplicação também pode utilizar a chamada sctp_recvmsg, que preencherá um ponteiro para a estrutura sctp_sndrcvinfo com essas informações.

Duas notificações contêm um campo de código da causa do erro SCTP. Os valores desse campo estão listados na Seção 3.3.10 da RFC 2960 (Stewart *et al*., 2000) e na seção "CAUSE CODES" de http://www.iana.org/assignments/sctp-parameters.

Notificações têm a seguinte forma:

```
struct sctp_tlv {
    u_int16_t sn_type;
    u_int16_t sn_flags;
    u_int32_t sn_length;
};

/* evento de notificação */
union sctp_notification {
    struct sctp_tlv sn_header;
    struct sctp_assoc_change sn_assoc_change;
    struct sctp_paddr_change sn_paddr_change;
    struct sctp_remote_error sn_remote_error;
    struct sctp_send_failed sn_send_failed;
    struct sctp_shutdown_event sn_shutdown_event;
    struct sctp_adaption_event sn_adaption_event;
    struct sctp_pdapi_event sn_pdapi_event;
};
```

Observe que o campo `sn_header` é utilizado para interpretar o valor de texto, a fim de decodificar a mensagem real sendo enviada. A Figura 9.6 ilustra o valor encontrado no campo `sn_header.sn_type` e o campo correspondente de inscrição utilizado com a opção de soquete `SCTP_EVENTS`.

sn_type	Campo de inscrição
SCTP_ASSOC_CHANGE	sctp_association_event
SCTP_PEER_ADDR_CHANGE	sctp_address_event
SCTP_REMOTE_ERROR	sctp_peer_error_event
SCTP_SEND_FAILED	sctp_send_failure_event
SCTP_SHUTDOWN_EVENT	sctp_shutdown_event
SCTP_ADAPTION_INDICATION	sctp_adaption_layer_event
SCTP_PARTIAL_DELIVERY_EVENT	sctp_partial_delivery_event

Figura 9.6 *sn_type* e o campo de inscrição de evento.

Cada notificação tem sua própria estrutura que fornece outras informações sobre o evento que ocorreu no transporte.

`SCTP_ASSOC_CHANGE`

Essa notificação informa a uma aplicação que ocorreu uma alteração para uma associação; uma nova associação iniciou ou uma existente terminou. As informações fornecidas com esse evento são definidas como segue:

```
struct sctp_assoc_change {
  u_int16_t sac_type;
  u_int16_t sac_flags;
  u_int32_t sac_length;
  u_intl6_t sac_state;
  u_int16_t sac_error;
  u_int16_t sac_outbound_streams;
  u_int16_t sac_inbound_streams;
  sctp_assoc_t sac_assoc_id;
  uint8_t sac info[];
};
```

O *sac_state* descreve o tipo de evento que ocorreu na associação e receberá um dos seguintes valores:

`SCTP_COMM_UP`

Esse estado indica que uma nova associação foi iniciada recentemente. Os campos de fluxo de entrada e de saída indicam o número de fluxos disponíveis em cada direção. A identificação de associação é preenchida com um valor único que pode ser utilizado para comunicar com a pilha SCTP local relacionada com essa associação.

`SCTP_COMM_LOST`

Esse estado indica que a associação especificada pela identificação de associação foi fechada devido a um limiar de inacessibilidade sendo alcançado (isto é, o tempo-limite de uma extremidade SCTP expirou múltiplas vezes e atingiu seu limiar, o que indica que o peer não está mais acessível), ou o peer realizou um fechamento abortivo (normalmente com a opção `SO_LINGER` ou utilizando `sendmsg` com um flag `MSG_ABORT`) da associação. Quaisquer informações específicas do usuário serão encontradas no campo `sac_info` da notificação.

SCTP_RESTART
: Esse estado indica que o peer reiniciou. A causa mais provável dessa notificação é um travamento e reinicialização do peer. A aplicação deve verificar o número de fluxos em cada direção, uma vez que esses valores podem mudar durante uma reinicialização.

SCTP_SHUTDOWN_COMP
: Esse estado indica que uma desativação iniciada pela extremidade local (via uma chamada a `shutdown` ou um `sendmsg` com um flag `MSG_EOF`) foi concluída. Para o estilo um para um, depois de receber essa notificação, o descritor de soquete pode ser utilizado novamente para se conectar a um peer diferente.

SCTP_CANT_STR_ASSOC
: Esse estado indica que um peer não respondeu a uma tentativa de configuração de associação (isto é, a mensagem INIT).

O campo *sac_error* armazena qualquer código de causa de erro do protocolo SCTP que possa ter causado uma alteração de associação. Os campos *sac_outbound_streams* e *sac_inbound_streams* informam à aplicação o número de fluxos em cada direção que foram negociados na associação. *sac_assoc_id* mantém um handle único para uma associação que pode ser utilizado para identificar a associação tanto nas opções de soquete como em notificações futuras. *sac_info* armazena todas outras informações disponíveis ao usuário. Por exemplo, se uma associação tivesse sido abortada pelo peer com um erro definido pelo usuário, esse erro seria encontrado nesse campo.

SCTP_PEER_ADDR_CHANGE
: Essa notificação indica que um dos endereços do peer passou por uma alteração de estado. Essa alteração pode ser uma falha, como o endereço de destino que não responde quando se envia algo a ele, ou uma recuperação, como um destino em um estado de falha que foi recuperado. A estrutura que acompanha uma alteração de endereço é a seguinte:

```
struct sctp_paddr_change {
  u_int16_t spc_type;
  u_int16_t spc_flags;
  u_int32_t spc_length;
  struct sockaddr_storage spc_aaddr;
  u_int32_t spc_state;
  u_int32_t spc_error;
  sctp_assoc_t spc_assoc_id;
};
```

O campo *spc_aaddr* armazena o endereço do peer afetado por esse evento. O campo *spc_state* armazena um dos valores descritos na Figura 9.7.

Quando um endereço é declarado `SCTP_ADDR_UNREACHABLE`, quaisquer dados enviados a ele serão redirecionados a um endereço alternativo. Também observe que alguns estados somente estarão disponíveis em implementações SCTP que suportam a opção de endereço dinâmico (por exemplo, `SCTP_ADDR_ADDED` e `SCTP_ADDR_REMOVED`).

O campo *spc_error* contém todo o código de erro da notificação para fornecer informações adicionais sobre o evento; e *spc_assoc_id* armazena a identificação da associação.

spc_state	Descrição
SCTP_ADDR_ADDED	O endereço agora está adicionado à associação
SCTP_ADDR_AVAILABLE	O endereço agora está acessível
SCTP_ADDR_CONFIRMED	O endereço agora foi confirmado e é válido
SCTP_ADDR_MADE_PRIM	O endereço agora se tornou o destino primário
SCTP_ADDR_REMOVED	O endereço não faz mais parte da associação
SCTP_ADDR_UNREACHABLE	O endereço não pode mais ser alcançado

Figura 9.7 Notificações de estado sobre o endereço do peer SCTP.

SCTP_REMOTE_ERROR

Um peer remoto poderia enviar uma mensagem de erro operacional à extremidade local. Essas mensagens podem indicar várias condições de erro da associação. O bloco de erro inteiro será passado para a aplicação *in wire format* quando essa notificação for ativada. O formato da mensagem será o seguinte:

```
struct sctp_remote_error {
  u_int16_t sre_type;
  u_int16_t sre_flags;
  u_int32_t sre_length;
  u_int16_t sre_error;
  sctp_assoc_t sre_assoc_id;
  u_int8_t sre_data[];
};
```

O *sre_error* armazenará um dos códigos da causa de erro do protocolo SCTP, *sre_assoc_id* conterá a identificação de associação e *sre_data* armazenará o erro completo *in wire format*.

SCTP_SEND_FAILED

Quando uma mensagem não puder ser entregue a um peer, ela é enviada de volta ao usuário por meio dessa notificação. Normalmente, a isso se segue uma notificação de falha de associação. Na maioria dos casos, a única maneira de uma mensagem não ser entregue é se a associação tiver falhado. O único caso em que uma falha de mensagem ocorrerá sem uma falha de associação é quando a extensão de confiabilidade parcial do SCTP estiver sendo utilizada.

Quando uma notificação de erro é enviada, o seguinte formato será lido pela aplicação:

```
struct sctp_send_failed {
  u_int16_t ssf_type;
  u_int16_t ssf_flags;
  u_int32_t ssf_length;
  u_int32_t ssf_error;
  struct sctp_sndrcvinfo ssf_info;
  sctp_assoc_t ssf_assoc_id;
  u_int8_t ssf_data[];
};
```

ssf_flags será configurado como um entre dois valores:

- SCTP_DATA_UNSENT: indica que a mensagem nunca poderá ser transmitida ao peer (por exemplo, o controle de fluxo evitou o envio da mensagem antes de expirar seu tempo de vida), assim o peer nunca a recebeu;

- SCTP_DATA_SENT: indica que os dados foram transmitidos ao peer pelo menos uma vez, mas nunca foram reconhecidos. Nesse caso, o peer *pode* ter recebido a mensagem, mas foi incapaz de reconhecê-la.

Essa distinção pode ser importante para um protocolo de transação, que talvez realize diferentes ações para recuperar-se de uma conexão falha com base no fato de uma dada mensagem ter ou não sido recebida. *ssf_error*, se não for zero, armazena um código de erro específico a essa notificação. O campo *ssf_info* fornece as informações repassadas (se houver alguma) para o kernel quando os dados foram enviados (por exemplo, número de fluxos, contexto, etc.). *ssf_assoc_id* armazena a identificação de associação e *ssf_data* armazena a mensagem não-entregue.

SCTP_SHUTDOWN_EVENT

Essa notificação é passada para uma aplicação quando um peer envia um bloco de SHUTDOWN à extremidade local. Essa notificação informa à aplicação que nenhum novo dado será aceito no soquete. Todos os dados atualmente enfileirados serão transmitidos e, na conclusão dessa transmissão, a associação será desativada. O formato da notificação é o seguinte:

```
struct sctp_shutdown_event {
  uint16_t sse_type;
  uint16_t sse_flags;
  uint32_t sse_length;
   sctp_assoc_t sse_assoc_id;
};
```

sse_assoc_id armazena a identificação de associação da associação que está sendo desativada e não pode mais aceitar dados.

SCTP_ADAPTION_INDICATION

Algumas implementações suportam um parâmetro de indicação de camada de adaptação. Esse parâmetro é trocado no INIT e no INIT-ACK para informar a cada peer que tipo de adaptação de aplicação está sendo realizado. A notificação terá a seguinte forma:

```
struct sctp_adaption_event {
  u_int16_t    sai_type;
  u_int16_t    sai_flags;
  u_int32_t    sai_length;
  u_int32_t    sai_adaption_ind;
  sctp_assoc_t sai_assoc_id;
};
```

O *sai_assoc_id* identifica a associação dessa notificação de camada de adaptação. *sai_adaption_ind* é um inteiro de 32 bits que o peer comunica ao host local na mensagem INIT ou INIT-ACK. A camada de adaptação é configurada com a opção de soquete SCTP_ADAPTION_LAYER (Seção 7.10). A opção INIT/INIT-ACK da camada de adaptação é descrita em Stewart *et al*. (2003b) e uma amostra de uso da opção para acesso remoto direto à memória / inserção de dados direta é descrita em Stewart *et al*. (2003a).

SCTP_PARTIAL_DELIVERY_EVENT

A interface da aplicação de entrega parcial é utilizada para enviar grandes mensagens ao usuário pelo buffer de soquete. Considere um usuário que grava uma única mensagem de 4 MB. Uma mensagem desse tamanho iria sobrecarregar ou exaurir os recursos do sistema. Uma implementação SCTP não conseguiria tratar essa mensagem a menos que tivesse um mecanismo para começar a entrega antes que toda a mensagem chegasse. Quando uma implementação realiza essa forma de entrega, é denominada *"API de entrega parcial"*. A API de entrega parcial é invocada pela implementação SCTP, que envia dados com o campo `msg_flags` permanecendo limpo até que a última parte da mensagem esteja pronta para ser entregue. A última parte da mensagem terá o `msg_flags` configurado como `MSG_EOR`. Observe que, se uma aplicação for receber grandes mensagens, deve utilizar `recvmsg` ou `sctp_recvmsg` de modo que o campo `msg_flags` possa ser examinado para essa condição.

Em alguns casos, a API de entrega parcial precisará transmitir um *status* à aplicação. Por exemplo, se a API de entrega parcial precisa ser abortada, a notificação `SCTP_PARTIAL_DELIVERY_EVENT` deverá ser enviada à aplicação receptora. Essa notificação tem o seguinte formato:

```
struct sctp_pdapi_event {
   uint16_t pdapi_type;
   uint16_t pdapi_flags;
   uint32_t pdapi_length;
   uint32_t pdapi_indication;
   sctp_assoc_t pdapi_assoc_id;
};
```

O campo *pdapi_assoc_id* identifica a associação em que ocorreu o evento da API de entrega parcial. O *pdapi_indication* armazena o evento que ocorreu. Atualmente, o único valor válido encontrado nesse campo é `SCTP_PARTIAL_DELIVERY_ABORTED`, que indica que a entrega parcial atualmente ativa foi abortada.

9.15 Resumo

O SCTP fornece ao autor da aplicação dois estilos diferentes de interface: o estilo um para um, compatível principalmente com aplicações TCP existentes para facilitar a migração para o SCTP; e o estilo um para muitos, que permite acesso a todos os recursos do SCTP. A função `sctp_peeloff` fornece um método de extrair uma associação de um estilo para o outro. O SCTP também fornece numerosas notificações de eventos de transporte nos quais uma aplicação talvez deseje inscrever-se. Esses eventos podem ajudar uma aplicação a gerenciar melhor as associações que mantém.

Como o SCTP é multihomed, nem todas as funções `socket`-padrão, introduzidas no Capítulo 4, são adequadas. Funções como `sctp_bindx`, `sctp_connectx`, `sctp_getladdrs` e `sctp_getpaddrs` fornecem métodos para examinar e controlar melhor os múltiplos endereços que podem compor uma associação SCTP. Funções utilitárias, como `sctp_sendmsg` e `sctp_recvmsg`, podem simplificar o uso desses recursos avançados. Iremos explorar vários conceitos introduzidos neste capítulo em mais detalhes por meio de exemplos nos Capítulos 10 e 23.

Exercícios

9.1 Em que situação um programador de aplicação provavelmente utilizaria a função *sctp_peeloff*?

9.2 Dissemos "o lado do servidor também será automaticamente fechado" na nossa discussão sobre o estilo um para muitos; por que isso é verdadeiro?

9.3 Por que o estilo um para muitos deve ser utilizado para fazer com que os dados sejam transferidos via *piggyback* no terceiro pacote do handshake de quatro vias? (*Dica*: Você deve ser capaz de enviar dados no momento da configuração da associação.)

9.4 Em que cenário você encontraria dados transferidos via *piggyback* tanto no terceiro como no quarto pacote do handshake de quatro vias?

9.5 A Seção 9.7 indica que o conjunto de endereços locais pode ser um subconjunto adequado dos endereços vinculados. Em que circunstância isso ocorreria?

Capítulo **10**

Exemplo de Cliente/Servidor SCTP

10.1 Visão geral

Agora, utilizaremos algumas funções básicas dos Capítulos 4 e 9 para escrever um exemplo completo de cliente/servidor SCTP de um para muitos. Nosso exemplo simples é semelhante ao servidor de eco apresentado no Capítulo 5 e segue os seguintes passos:

1. Um cliente lê uma linha de texto da entrada-padrão e envia essa linha ao servidor. A linha segue a forma **`[#] text`**, em que o número entre colchetes é o número de fluxo de SCTP pelo qual a mensagem de texto deve ser enviada.
2. O servidor recebe a mensagem de texto da rede, soma um ao número de fluxo no qual a mensagem chegou e envia-a de volta ao cliente nesse novo número de fluxo.
3. O cliente lê a linha ecoada e a imprime na saída-padrão, exibindo o número de fluxo, o número de seqüência de fluxo e a string de texto.

A Figura 10.1 representa esse cliente/servidor simples junto com as funções utilizadas para entrada e saída.

Mostramos duas setas entre o cliente e o servidor para descrever dois fluxos unidirecionais sendo utilizados, apesar da associação geral ser full-duplex. As funções `fgets` e `puts` são provenientes da biblioteca E/S-padrão. Não utilizamos as funções `writen` e `readline` definidas na Seção 3.9 visto que são desnecessárias. Em vez disso, utilizaremos as funções `sctp_sendmsg` e `sctp_recvmsg` definidas nas Seções 9.9 e 9.10, respectivamente.

Para esse exemplo, utilizamos o servidor no estilo um para muitos. Fazemos essa escolha por uma razão importante. Os exemplos no Capítulo 5 podem ser modificados e executados

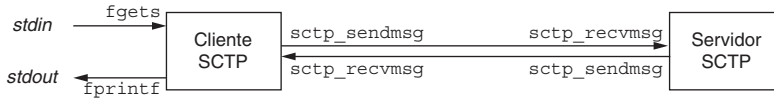


Figura 10.1 Cliente e servidor de eco SCTP simples com fluxo.

pelo SCTP com uma pequena alteração: modificar a chamada de função `socket` para especificar `IPPROTO_SCTP` em vez de `IPPROTO_TCP` como o terceiro argumento. Entretanto, simplesmente fazer essa alteração não tiraria proveito de quaisquer recursos adicionais fornecidos pelo SCTP, exceto multihoming. A utilização do estilo um para muitos permite colocar em prática todos os recursos do SCTP.

10.2 Servidor de eco com fluxo no estilo SCTP de um para muitos: função `main`

Nossos cliente e servidor SCTP seguem o fluxo de funções diagramado na Figura 9.2. Mostramos um programa de servidor iterativo na Figura 10.2.

Configuração da opção de incremento de fluxo

13-14 Por default, nosso servidor responde utilizando o próximo fluxo mais alto do aquele em que a mensagem foi recebida. Se um inteiro for passado como argumento na linha de comando, o servidor interpretará o argumento como o valor de `stream_increment`, isto é, ele decidirá se deve incrementar ou não o número de fluxo das mensagens entrantes. Utilizaremos essa opção na nossa discussão sobre bloqueio de início de linha na Seção 10.5.

Criação de um soquete SCTP

15 Um soquete SCTP no estilo um para muitos é criado.

Vinculando um endereço

16-20 Uma estrutura de endereço de soquete de Internet é preenchida com o endereço curinga (`INADDR_ANY`) e a porta bem-conhecida do servidor, `SERV_PORT`. Vincular o endereço curinga informa ao sistema que esse ponto final de SCTP utilizará todos os endereços locais disponíveis em qualquer associação configurada. Para hosts com multihome, essa vinculação significa que uma extremidade remota será capaz de criar associações e de enviar pacotes a qualquer endereço roteável do host local. Nossa escolha do número de porta SCTP está baseada na Figura 2.10. Observe que o servidor faz as mesmas considerações feitas anteriormente no nosso exemplo prévio da Seção 5.2.

Configuração para notificações de interesse

21-23 O servidor altera a sua inscrição de notificação para o soquete SCTP de um para muitos. O servidor inscreve-se somente para o `sctp_data_io_event`, que lhe permitirá ver a estrutura `sctp_sndrcvinfo`. A partir dessa estrutura, o servidor pode determinar o número do fluxo em que a mensagem chegou.

Ativando associações entrantes

24 O servidor permite associações entrantes com a chamada a `listen`. Em seguida, o controle entra no loop de processamento principal.

sctp/sctpserv01.c

```
1 #include   "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int    sock_fd,msg_flags;
6     char   readbuf[BUFFSIZE];
7     struct sockaddr_in servaddr, cliaddr;
8     struct sctp_sndrcvinfo sri;
```

Figura 10.2 Servidor de eco SCTP com fluxo (*continua*).

```
 9      struct sctp_event_subscribe evnts;
10      int     stream_increment = 1;
11      socklen_t len;
12      size_t rd_sz;

13      if (argc == 2)
14          stream_increment = atoi(argv[1]);
15      sock_fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
16      bzero(&servaddr, sizeof(servaddr));
17      servaddr.sin_family = AF_INET;
18      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
19      servaddr.sin_port = htons(SERV_PORT);

20      Bind(sock_fd, (SA *) &servaddr, sizeof(servaddr));

21      bzero(&evnts, sizeof(evnts));
22      evnts.sctp_data_io_event = 1;
23      Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_EVENTS, &evnts, sizeof(evnts));

24      Listen(sock_fd, LISTENQ);
25      for ( ; ; ) {
26          len = sizeof(struct sockaddr_in);
27          rd_sz = Sctp_recvmsg(sock_fd, readbuf, sizeof(readbuf),
28                               (SA *) &cliaddr, &len, &sri, &msg_flags);
29          if(stream_increment) {
30             sri.sinfo_stream++;
31             if(sri.sinfo_stream >=
32                sctp_get_no_strms(sock_fd,(SA *) &cliaddr, len))
33                sri.sinfo_stream = 0;
34          }
35          Sctp_sendmsg(sock_fd, readbuf, rd_sz,
36                       (SA *) &cliaddr, len,
37                       sri.sinfo_ppid,
38                       sri.sinfo_flags, sri.sinfo_stream, 0, 0);
39      }
40  }
```

sctp/sctpserv01.c

Figura 10.2 Servidor de eco SCTP com fluxo (*continuação*).

Esperando a mensagem

26-28 O servidor inicializa o tamanho da estrutura de endereço de soquete do cliente, e então bloqueia enquanto espera uma mensagem proveniente de qualquer peer remoto.

Incrementando o número do fluxo se desejado

29-34 Quando uma mensagem chega, o servidor verifica o flag `stream_increment` para ver se deve incrementar o número do fluxo. Se o flag estiver configurado (nenhum argumento foi passado para a linha de comando), o servidor incrementará o número do fluxo da mensagem. Se esse número aumentar até ou além do fluxo máximo, o qual é obtido chamando nossa função interna `sctp_get_no_strms`, o servidor irá redefinir o fluxo para 0. A função `sctp_get_no_strms` não é mostrada. Ela utiliza a opção `SCTP_STATUS` de soquete SCTP discutida na Seção 7.10 para encontrar o número de fluxos negociado.

Enviando de volta a resposta

35-38 O servidor envia de volta a mensagem utilizando o ID do protocolo de payload, flags e o número possivelmente modificado do fluxo da estrutura `sri`.

Observe que esse servidor não quer uma notificação de associação, assim ele desativa todos os eventos que passariam mensagens até o buffer do soquete. O servidor conta com as in-

formações na estrutura `sctp_sndrcvinfo` e no endereço retornado encontrado em *cliaddr* para localizar a associação do peer e retornar o eco.

Esse programa executa initerruptamente até que o usuário o feche com um sinal externo.

10.3 Cliente de eco com fluxo no estilo SCTP de um para muitos: função `main`

A Figura 10.3 mostra nossa função `main` do cliente SCTP.

Validação dos argumentos e criação de um soquete

9-15 O cliente valida os argumentos que lhe são passados. Primeiro, ele verifica se o chamador forneceu um host para o qual enviar mensagens. Em seguida, se a opção "echo to all" está ativada (veremos essa opção utilizada na Seção 10.5). Por fim, o cliente cria um soquete SCTP no estilo um para muitos.

Configuração do endereço do servidor

16-20 O cliente traduz o endereço do servidor, passado na linha de comando, utilizando a função `inet_pton`. Ele combina esse endereço com o número da porta bem-conhecida do servidor e utiliza o endereço resultante como o destino para as solicitações.

Configuração para notificações de interesse

21-23 O cliente configura explicitamente a assinatura de notificação fornecida pelo nosso soquete SCTP de um para muitos. Novamente, ele não quer eventos `MSG_NOTIFICATION`. Portanto, o cliente desativa esses eventos (como ocorreu no servidor) e somente permite a recepção da estrutura `sctp_sndrcvinfo`.

Chamando a função de processamento de eco

24-28 Se o flag `echo_to_all` não estiver configurado, o cliente chama a função `sctpstr_cli`, discutida na Seção 10.4. Se o flag `echo_to_all` estiver configurado, o cliente chama a função `sctpstr_cli_echoall`. Discutiremos essa função na Seção 10.5, quando explorarmos os usos para os fluxos de SCTP.

sctp/sctpclient01.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int      sock_fd;
 6     struct sockaddr_in servaddr;
 7     struct sctp_event_subscribe evnts;
 8     int      echo_to_all = 0;
 9     if (argc < 2)
10         err_quit("Missing host argument - use '%s host [echo]'\n", argv[0]);
11     if (argc > 2) {
12         printf("Echoing messages to all streams\n");
13         echo_to_all = 1;
14     }
15     sock_fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
16     bzero(&servaddr, sizeof(servaddr));
17     servaddr.sin_family = AF_INET;
18     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
19     servaddr.sin_port = htons(SERV_PORT);
20     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
```

Figura 10.3 Função main do cliente de eco SCTP com fluxo (*continua*).

```
21      bzero(&evnts, sizeof(evnts));
22      evnts.sctp_data_io_event = 1;
23      Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_EVENTS, &evnts, sizeof(evnts));
24      if (echo_to_all == 0)
25          sctpstr_cli(stdin, sock_fd, (SA *) &servaddr, sizeof(servaddr));
26      else
27          sctpstr_cli_echoall(stdin, sock_fd, (SA *) &servaddr,
28                              sizeof(servaddr));
29      Close(sock_fd);
30      return (0);
31  }
```

sctp/sctpclient01.c

Figura 10.3 Função main do cliente de eco SCTP com fluxo (*continuação*).

Conclusão

29-31 Ao retornar do processamento, o cliente fecha o soquete SCTP, o que desativa todas as associações SCTP que utilizam o soquete. O cliente então retorna de main com um código de retorno de 0, indicando que o programa foi executado com sucesso.

10.4 Cliente de eco com fluxo SCTP: função `str_cli`

A Figura 10.4 mostra nossa função de processamento do cliente SCTP-padrão.

sctp/sctp_strcli.c

```
 1 #include    "unp.h"

 2 void
 3 sctpstr_cli(FILE *fp, int sock_fd, struct sockaddr *to, socklen_t tolen)
 4 {
 5     struct  sockaddr_in peeraddr;
 6     struct  sctp_sndrcvinfo sri;
 7     char    sendline[MAXLINE], recvline[MAXLINE];
 8     socklen_t len;
 9     int     out_sz,rd_sz;
10     int     msg_flags;

11     bzero(&sri,sizeof(sri));
12     while (fgets(sendline, MAXLINE, fp) != NULL) {
13         if  (sendline[0] != '[') {
14             printf("Error, line must be of the form '[streamnum]text'\n");
15             continue;
16         }
17         sri.sinfo_stream = strtol(&sendline[1],NULL,0);
18         out_sz = strlen(sendline);
19         Sctp_sendmsg(sock_fd, sendline, out_sz,
20                      to, tolen, 0, 0, sri.sinfo_stream, 0, 0);

21         len = sizeof(peeraddr);
22         rd_sz = Sctp_recvmsg(sock_fd, recvline, sizeof(recvline),
23                              (SA *) &peeraddr, &len, &sri, &msg_flags);
24         printf("From str:%d seq:%d (assoc:0x%x):",
25                sri.sinfo_stream, sri.sinfo_ssn, (u_int) sri.sinfo_assoc_id);
26         printf("%.*s", rd_sz, recvline);
27     }
28 }
```

sctp/sctp_strcli.c

Figura 10.4 Função `sctp_strcli` do SCTP.

Inicialização da estrutura `sri` e entrada no loop

11-12 O cliente inicia limpando a estrutura `sctp_sndrcvinfo`, `sri`. Ele, então, entra em um loop que lê a partir do `fp` passado pelo nosso chamador com uma chamada bloqueadora a `fgets`. O programa principal passa `stdin` a essa função, assim a entrada do usuário é lida e processada no loop até que o caractere EOF de término (Control-D) seja digitado pelo usuário. Essa ação do usuário finaliza a função e causa um retorno ao chamador.

Validação da entrada

13-16 O cliente examina a entrada do usuário para certificar-se de que ela é da forma **`[#]text`**. Se o formato for inválido, o cliente imprime uma mensagem de erro e entra novamente na chamada bloqueadora à função `fgets`.

Tradução do número de fluxo

17 O cliente traduz o fluxo solicitado pelo usuário encontrado na entrada no campo `sinfo_stream` na estrutura `sri`.

Envio da mensagem

18-20 Depois de inicializar os comprimentos apropriados de endereço e tamanho dos dados reais do usuário, o cliente envia a mensagem utilizando a função `sctp_sendmsg`.

Bloqueio durante a espera da mensagem

21-23 O cliente agora bloqueia e espera a mensagem ecoada a partir do servidor.

Exibição da mensagem e loop retornado

24-26 O cliente exibe a mensagem retornada ecoada a ele exibindo o número do fluxo, o número da seqüência do fluxo, bem como a mensagem de texto. Depois de exibir a mensagem, ele volta ao loop para obter uma outra solicitação do usuário.

Executando o código

Um usuário inicia o servidor de eco SCTP sem argumentos na máquina FreeBSD. O cliente é iniciado somente com o endereço do nosso servidor.

```
freebsd4% sctpclient01 10.1.1.5
[0]Hello                                                  Envia uma mensagem no fluxo 0
From str:1 seq:0 (assoc:0xc99e15a0):[0]Hello              Servidor ecoa no fluxo 1
[4]Message two                                            Envia uma mensagem no fluxo 4
From str:5 seq:0 (assoc:0xc99e15a0):[4]Message two        Servidor ecoa no fluxo 5
[4]Message three                                          Envia uma segunda mensagem
                                                            no fluxo 4
From str:5 seq:1 (assoc:0xc99e15a0):[4]Message three      Servidor ecoa no fluxo 5
^D                                                        Control-D é o nosso caractere
                                                            de EOF
freebsd4%
```

Observe que o cliente envia a mensagem nos fluxos 0 e 4 enquanto nosso servidor envia as mensagens de volta nos fluxos 1 e 5. Esse comportamento é esperado no nosso servidor sem argumentos. Também observe que o número de seqüência do fluxo incrementou na segunda mensagem recebida no fluxo 5, como esperado.

10.5 Explorando o bloqueio de início de linha

Nosso servidor simples fornece um método para enviar mensagens de texto a qualquer número de fluxos. Um *fluxo (stream)* em SCTP não é um fluxo de bytes (como em TCP), mas uma

seqüência de mensagens ordenadas dentro da associação. Esses fluxos subordenados são utilizados para evitar o bloqueio de início de linha encontrado no TCP.

O bloqueio de início de linha ocorre quando um segmento TCP é perdido e um segmento TCP subseqüente chega fora de ordem. Esse segmento subseqüente é mantido até que o primeiro segmento TCP seja retransmitido e chegue ao receptor. O retardo da entrega do segmento subseqüente garante que a aplicação receptora veja todos os dados na ordem em que a aplicação emissora os enviou. Esse retardo para alcançar um ordenamento completo é bem útil, mas tem uma desvantagem. Suponha que mensagens semanticamente independentes sejam enviadas por uma única conexão TCP. Por exemplo, um servidor talvez envie três diferentes imagens para um navegador Web exibir. Para fazer com que essas imagens apareçam na tela do usuário em paralelo, um servidor envia uma parte da primeira imagem, em seguida uma parte da segunda imagem e, por fim, uma parte da terceira imagem. O servidor repete esse processo até que todas as três imagens são transmitidas com sucesso ao navegador. Mas o que acontece se um pacote TCP que mantém uma parte da primeira imagem for perdido? O cliente armazenará todos os dados até que essa parte ausente é retransmitida e chega com sucesso, retardando todos os dados da segunda e da terceira imagens bem como os da primeira imagem. A Figura 10.5 ilustra esse problema.

> Embora essa não seja a maneira como o HTTP funcione, várias extensões, como SCP (Spero, 1996) e SMUX (Gettys e Nielsen, 1998), foram propostas para permitir esse tipo de funcionalidade paralela por cima do TCP. Esses protocolos de multiplexação foram propostos para evitar o comportamento prejudicial de múltiplas conexões TCP paralelas que não compartilham um estado (Touch, 1997). Embora criar uma conexão TCP por imagem (como normalmente fazem os clientes HTTP) evite o problema do bloqueio de início de linha, cada conexão tem de descobrir o RTT e a largura de banda disponível de maneira independente; uma perda em uma das conexões (um sinal de congestionamento no caminho) não necessariamente diminui a velocidade das outras conexões. Isso leva a uma menor utilização agregada de redes congestionadas.

Esse bloqueio não é o que a aplicação gostaria que ocorresse. Idealmente, apenas as últimas partes da primeira imagem sofreriam um retardo enquanto a segunda e a terceira partes das imagens que chegassem em ordem seriam entregues imediatamente ao usuário.

O bloqueio de início de linha pode ser minimizado pelo recurso de fluxo múltiplo (*multistream*) do SCTP. Na Figura 10.6, vemos as mesmas três imagens sendo enviadas. Desta vez, o servidor utiliza fluxos de modo que o bloqueio de início de linha ocorra somente onde é de-

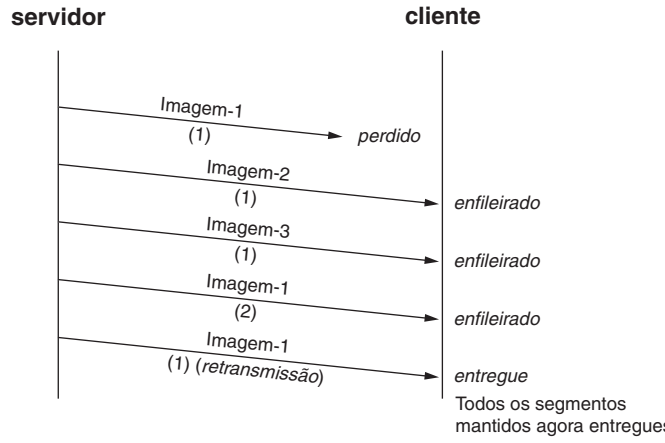


Figura 10.5 Enviando três imagens por uma conexão TCP.

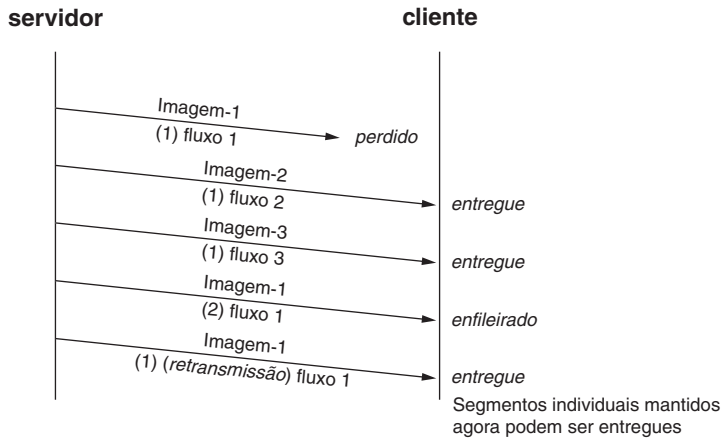


Figura 10.6 Enviando três imagens por três fluxos de SCTP.

sejado, permitindo a entrega da segunda e da terceira imagens, mas mantendo a primeira imagem parcialmente recebida até que a entrega em ordem seja possível.

Agora, completamos nosso código para o cliente, incluindo a função ausente `sctpstr_cli_echoall` (Figura 10.7), que utilizaremos para demonstrar como o SCTP minimiza o bloqueio de início de linha. Essa função é semelhante à nossa função `sctpstr_cli` anterior, exceto que o cliente não mais espera um número de fluxo entre colchetes precedendo cada mensagem. Em vez disso, a função envia a mensagem do usuário a todos os fluxos `SERV_MAX_SCTP_STRM`. Depois de enviar as mensagens, o cliente espera todas as respostas provenientes do servidor. Ao executar o código, também passamos um argumento adicional para o servidor de modo que ele responda no mesmo fluxo em que uma mensagem foi recebida. Dessa maneira, o usuário pode monitorar melhor as respostas enviadas e a ordem de chegada.

Inicialização das estruturas de dados e espera pela entrada

13-15 Como anteriormente, o cliente inicializa a estrutura `sri` utilizada para configurar o fluxo de envio e recebimento. Além disso, ele zera o buffer de dados em que coletará a entrada do usuário. Em seguida, entra no loop principal, mais uma vez bloqueando a entrada do usuário.

Pré-processamento da mensagem

16-20 O cliente configura o tamanho da mensagem e então exclui o caractere de nova linha que está no final do buffer (se houver algum).

Envio da mensagem para cada fluxo

21-26 O cliente envia a mensagem utilizando a função `sctp_sendmsg`, enviando o buffer inteiro cujo tamanho é `SCTP_MAXLINE` bytes. Antes de enviar a mensagem, ele acrescenta a string "**.msg.**" e o número de fluxo, de modo que possamos observar a ordem de chegada das mensagens. Dessa maneira, podemos comparar a ordem de chegada com a ordem em que o cliente enviou as mensagens reais. Também observe que o cliente envia as mensagens a um número especificado de fluxos sem considerar quantos fluxos na verdade foram especificados. É possível que um ou mais dos envios possa falhar se o peer negociar o número de fluxos para baixo.

> Esse código tem o potencial de falhar se as janelas de envio ou recebimento forem muito pequenas. Se a janela de recebimento do peer for muito pequena, é possível que o cliente seja bloqueado. Como o cliente não lê nenhuma informação até que todo o seu envio esteja com-

pleto, o servidor também poderia potencialmente ser bloqueado enquanto espera que o cliente termine de ler as respostas que o servidor já enviou. O resultado desse cenário seria um impasse entre os dois pontos finais. Esse código não é para ser escalável, mas, em vez disso, serve para ilustrar os fluxos e o bloqueio de início de linha de uma maneira simples e direta.

sctp/sctp_strcliecho.c

```
 1 #include     "unp.h"
 2 #define SCTP_MAXLINE 800
 3 void
 4 sctpstr_cli_echoall(FILE *fp, int sock_fd, struct sockaddr *to,
 5                     socklen_t tolen)
 6 {
 7     struct sockaddr_in peeraddr;
 8     struct sctp_sndrcvinfo sri;
 9     char   sendline[SCTP_MAXLINE], recvline[SCTP_MAXLINE];
10     socklen_t len;
11     int     rd_sz,i,strsz;
12     int     msg_flags;
13     bzero(sendline,sizeof(sendline));
14     bzero(&sri,sizeof(sri));
15     while (fgets(sendline, SCTP_MAXLINE - 9, fp) != NULL) {
16         strsz = strlen(sendline);
17         if  (sendline[strsz - 1] == '\n') {
18             sendline[strsz - 1] = '\0';
19             strsz--;
20         }
21         for(i = 0; i < SERV_MAX_SCTP_STRM; i++) {
22             snprintf(sendline + strsz, sizeof(sendline) - strsz,
23                      ".msg.%d", i);
24             Sctp_sendmsg(sock_fd, sendline, sizeof(sendline),
25                          to, tolen, 0, 0, i, 0, 0);
26         }
27         for (i=0; i < SERV_MAX_SCTP_STRM; i++) {
28             len = sizeof(peeraddr);
29             rd_sz = Sctp_recvmsg(sock_fd, recvline, sizeof(recvline),
30                                  (SA *) &peeraddr, &len, &sri, &msg_flags);
31             printf("From str:%d seq:%d (assoc:0x%x):",
32                    sri.sinfo_stream, sri.sinfo_ssn,
33                    (u_int) sri.sinfo_assoc_id);
34             printf("%.*s\n", rd_sz, recvline);
35         }
36     }
37 }
```

sctp/sctp_strcliecho.c

Figura 10.7 sctp_strcliecho.

Leitura e exibição das mensagens ecoadas de volta

27-35 Agora bloqueamos, lendo todas as mensagens de resposta provenientes do nosso servidor e exibindo cada uma como fizemos anteriormente. Depois que a última mensagem é lida, o cliente entra em um loop procurando outras entradas de usuário.

Execução do código

Executamos o cliente e o servidor em duas máquinas FreeBSD separadas por um roteador configurável, como ilustrado na Figura 10.8. O roteador pode ser configurado para inserir tanto retardo como perda. Primeiro, executamos o programa sem perdas inseridas pelo roteador.

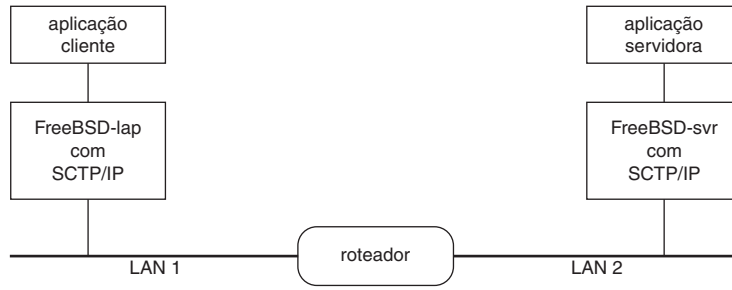


Figura 10.8 Laboratório de cliente/servidor SCTP.

Iniciamos o servidor com um argumento adicional em "0", forçando-o a não incrementar o número de fluxo nas suas respostas.

Em seguida, iniciamos o cliente, passando-lhe o endereço do servidor de eco e um argumento adicional, de modo que ele enviará uma mensagem a cada fluxo.

```
freebsd4% sctpclient01 10.1.4.1 echo
Echoing messages to all streams
Hello
From str:0 seq:0 (assoc:0xc99e15a0):Hello.msg.0
From str:1 seq:0 (assoc:0xc99e15a0):Hello.msg.1
From str:2 seq:0 (assoc:0xc99e15a0):Hello.msg.2
From str:3 seq:0 (assoc:0xc99e15a0):Hello.msg.3
From str:4 seq:0 (assoc:0xc99e15a0):Hello.msg.4
From str:5 seq:0 (assoc:0xc99e15a0):Hello.msg.5
From str:6 seq:0 (assoc:0xc99e15a0):Hello.msg.6
From str:7 seq:0 (assoc:0xc99e15a0):Hello.msg.7
From str:8 seq:0 (assoc:0xc99e15a0):Hello.msg.8
From str:9 seq:0 (assoc:0xc99e15a0):Hello.msg.9
^D
freebsd4%
```

Sem perdas, o cliente vê as respostas chegarem de volta na ordem em que ele as enviou. Agora, alteramos os parâmetros do nosso roteador para perder 10% de todos os pacotes que percorrem ambas as direções e reiniciamos nosso cliente.

```
freebsd4% sctpclient01 10.1.4.1 echo
Echoing messages to all streams
Hello
From str:0 seq:0 (assoc:0xc99e15a0):Hello.msg.0
From str:2 seq:0 (assoc:0xc99e15a0):Hello.msg.2
From str:3 seq:0 (assoc:0xc99e15a0):Hello.msg.3
From str:5 seq:0 (assoc:0xc99e15a0):Hello.msg.5
From str:1 seq:0 (assoc:0xc99e15a0):Hello.msg.1
From str:8 seq:0 (assoc:0xc99e15a0):Hello.msg.8
From str:4 seq:0 (assoc:0xc99e15a0):Hello.msg.4
From str:7 seq:0 (assoc:0xc99e15a0):Hello.msg.7
From str:9 seq:0 (assoc:0xc99e15a0):Hello.msg.9
From str:6 seq:0 (assoc:0xc99e15a0):Hello.msg.6
^D
freebsd4%
```

Podemos verificar que as mensagens dentro de um fluxo são adequadamente mantidas seguras para o reordenamento fazendo com que o cliente envie duas mensagens para cada fluxo. Também modificamos o cliente para adicionar um sufixo ao número da sua mensagem para ajudar na identificação de cada duplicata de mensagem. As modificações no servidor são mostradas na Figura 10.9.

sctp/sctp_strcliecho2.c

```
21          for (i = 0; i < SERV_MAX_SCTP_STRM; i++) {
22              snprintf(sendline + strsz, sizeof(sendline) - strsz,
23                      ".msg.%d 1", i);
24              Sctp_sendmsg(sock_fd, sendline, sizeof(sendline),
25                           to, tolen, 0, 0, i, 0, 0);
26              snprintf(sendline + strsz, sizeof(sendline) - strsz,
27                      ".msg.%d 2", i);
28              Sctp_sendmsg(sock_fd, sendline, sizeof(sendline),
29                           to, tolen, 0, 0, i, 0, 0);
30          }
31          for (i = 0; i < SERV_MAX_SCTP_STRM * 2; i++) {
32              len = sizeof(peeraddr);
```

sctp/sctp_strcliecho2.c

Figura 10.9 Modificações em sctp_strcliecho.

Adicionando e enviando um número extra de mensagens

22-25 O cliente adiciona um número de mensagem extra, "1", pra ajudar a monitorar qual mensagem é enviada. Em seguida, ele envia a mensagem utilizando a função `sctp_sendmsg`.

Alterando o número de mensagem e enviando novamente

26-29 O cliente agora altera o número de "1" para "2" e envia essa mensagem atualizada ao mesmo fluxo.

Leitura e exibição das mensagens

31 Aqui, o código requer apenas uma pequena alteração: dobramos o número de mensagens que o cliente espera receber de volta do servidor de eco.

Executando o código modificado

Iniciamos nossos servidor e cliente modificados, como anteriormente, e obtemos a seguinte saída do cliente:

```
freebsd4% sctpclient01 10.1.4.1 echo
Echoing messages to all streams
Hello
From str:0 seq:0 (assoc:0xc99e15a0):He11o.msg.0 1
From str:0 seq:1 (assoc:0xc99e15a0):Hello.msg.0 2
From str:l seq:0 (assoc:0xc99e15a0):Hello.msg.l 1
From str:4 seq:0 (assoc:0xc99e15a0):Hello.msg.4 1
From str:5 seq:0 (assoc:0xc99e15a0):Hello.msg.5 1
From str:7 seq:0 (assoc:0xc99e15a0):Hello.msg.7 1
From str:8 seq:0 (assoc:0xc99e15a0):Hello.msg.8 1
From str:9 seq:0 (assoc:0xc99e15a0):Hello.msg.9 1
From str:3 seq:0 (assoc:0xc99e15a0):Hello.msg.3 1
From str:3 seq:1 (assoc:0xc99e15a0):Hello.msg.3 2
From str:l seq:l (assoc:0xc99e15a0):Hello.msg.l 2
From str:5 seq:1 (assoc:0xc99e15a0):He11o.msg.5 2
From str:2 seq:0 (assoc:0xc99e15a0):Hello.msg.2 1
From str:6 seq:0 (assoc:0xc99e15a0):Hello.msg.6 1
From str:6 seq:l (assoc:0xc99e15a0):Hello.msg.6 2
From str:2 seq:1 (assoc:0xc99e15a0):Hello.msg.2 2
From str:7 seq:l (assoc:0xc99e15a0):Hello.msg.7 2
From str:8 seq:l (assoc:0xc99e15a0):Hello.msg.8 2
From str:9 seq:l (assoc:0xc99e15a0):Hello.msg.9 2
From str:4 seq:l (assoc:0xc99e15a0):Hello.msg.4 2
^D
freebsd4%
```

Como podemos ver na saída, mensagens são perdidas e, mesmo assim, somente aquelas em um fluxo particular sofrem retardos. Os dados dos outros fluxos não sofrem retardos. Os fluxos de SCTP podem ser um mecanismo poderoso para "escapar" do bloqueio de início de linha e, no entanto, preservar a ordem dentro de um conjunto de mensagens relacionadas.

10.6 Controlando o número de fluxos

Vimos como os fluxos de SCTP podem ser utilizados, mas como podemos controlar o número de fluxos que uma extremidade solicita na inicialização de associação? Nossos exemplos anteriores utilizaram o padrão de sistema quanto ao número de fluxos externos. Na implementação FreeBSD KAME do SCTP, esse padrão é configurado como 10 fluxos. E se nossas aplicação e servidor quisessem utilizar mais de 10 fluxos? Na Figura 10.10, mostramos uma modificação que permite a um servidor aumentar o número de fluxos que uma extremidade solicita na inicialização da associação. Observe que essa alteração deve ser feita no soquete antes de uma associação ser criada.

sctp/sctpserv02.c

```
14      if (argc == 2)
15          stream_increment = atoi(argv[1]);
16      sock_fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
17      bzero(&initm,sizeof(initm));
18      initm.sinit_num_ostreams = SERV_MORE_STRMS_SCTP;
19      Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_INITMSG, &initm, sizeof(initm));
```

sctp/sctpserv02.c

Figura 10.10 Solicitando mais fluxos no nosso servidor.

Configuração inicial

14-16 Como ocorreu anteriormente, o servidor configura os flags com base em argumentos adicionais e abre o soquete.

Modificação das solicitações de fluxos

17-19 Essas linhas contêm o novo código que adicionamos ao nosso servidor. Primeiro o servidor zera a estrutura `sctp_initmsg`. Essa alteração garante que a chamada `setsockopt` não mudará não-intencionalmente outros valores quaisquer. O servidor então configura o campo `sinit_max_ostreams` como o número de fluxos que ele gostaria de solicitar. Em seguida, configura a opção de soquete com os parâmetros iniciais de mensagem.

Uma alternativa à configuração de uma opção de soquete seria utilizar a função `sendmsg` e fornecer dados auxiliares para solicitar diferentes parâmetros de fluxo do padrão. Esse tipo de dado auxiliar é apenas eficaz na interface de soquete no estilo um para muitos.

10.7 Controlando a terminação

Nos nossos exemplos, dependemos do cliente que fecha o soquete para desativar a associação. Mas a aplicação cliente talvez nem sempre deseje fechar o soquete. Nesse sentido, nosso servidor talvez não queira manter a associação aberta depois de enviar a mensagem de resposta. Nesses casos, precisamos examinar dois mecanismos alternativos para desativar uma associação. Para a interface no estilo um para muitos, há dois possíveis métodos disponíveis à aplicação: um é elegante, enquanto o outro é problemático.

Se um servidor quiser desativar uma associação depois de enviar uma mensagem, aplicamos o flag `MSG_EOF` à mensagem de resposta no campo `sinfo_flags` da estrutura `sctp_sndrcvinfo`. Esse flag força a desativação de uma associação depois que a mensa-

gem sendo enviada é reconhecida. A outra alternativa é aplicar o flag `MSG_ABORT` ao campo `sinfo_flags`. Esse flag forçará um término imediato da associação com um bloco de ABORT. Um bloco de ABORT é semelhante a um segmento `TCP RST`, que termina qualquer associação sem nenhum retardo. Observe que quaisquer dados ainda não transferidos serão descartados. Entretanto, fechar uma sessão SCTP com um bloco de ABORT não tem nenhum efeito colateral, como impedir o estado TIME_WAIT do TCP; o bloco de ABORT causa um close abortivo "elegante". A Figura 10.11 mostra as modificações necessárias para que nosso servidor de eco inicie uma desativação elegante quando a mensagem de resposta é enviada ao peer. A Figura 10.12 mostra um cliente modificado que envia um bloco de ABORT antes de fechar o soquete.

sctp/sctpserv03.c

```
25    for ( ; ; ) {
26        len = sizeof(struct sockaddr_in);
27        rd_sz = Sctp_recvmsg(sock_fd, readbuf, sizeof(readbuf),
28                             (SA *) &cliaddr, &len, &sri, &msg_flags);
29        if(stream_increment) {
30           sri.sinfo_stream++;
31           if(sri.sinfo_stream >=
32               sctp_get_no_strms(sock_fd,(SA *) &cliaddr, len))
33               sri.sinfo_stream = 0;
34        }
35        Sctp_sendmsg(sock_fd, readbuf, rd_sz,
36                     (SA *) &cliaddr, len,
37                     sri.sinfo_ppid,
38                     (sri.sinfo_flags | MSG_EOF), sri.sinfo_stream, 0, 0);
39    }
```

sctp/sctpserv03.c

Figura 10.11 O servidor termina uma associação na resposta.

Enviando de volta uma resposta, mas desativando a associação

38 Podemos ver que a alteração nessa linha é simplesmente fazer uma operação OU no flag `MSG_EOF` para a função `sctp_sendmsg`. Esse valor de flag faz com que nosso servidor desative a associação depois que a mensagem de resposta é reconhecida com sucesso.

sctp/sctpclient02.c

```
25    if (echo_to_all == 0)
26        sctpstr_cli(stdin,sock_fd,(SA *) &servaddr, sizeof(servaddr));
27    else
28        sctpstr_cli_echoall(stdin, sock_fd, (SA *) &servaddr,
29                            sizeof(servaddr));
30    strcpy(byemsg, "goodbye");
31    Sctp_sendmsg(sock_fd, byemsg, strlen(byemsg),
32                 (SA *) &servaddr, sizeof(servaddr), 0, MSG_ABORT, 0, 0, 0);
33    Close(sock_fd);
```

sctp/sctpclient02.c

Figura 10.12 O cliente aborta a associação antes de fechar.

Abortando a associação antes de fechar

30-32 Nessas linhas, o cliente prepara uma mensagem que é incluída com o abortamento como causa de erro de usuário. O cliente então chama a função `sctp_sendmsg` com o flag MSG_ABORT. Esse flag envia um bloco de ABORT, que imediatamente termina a associação. O bloco de ABORT inclui a causa de erro iniciada pelo usuário com a mensagem ("goodbye") no campo do motivo da camada superior.

Fechando o descritor de soquete

33 Mesmo que a associação tenha sido abortada, ainda precisamos fechar o descritor de soquete para liberar os recursos do sistema associados com ela.

10.8 Resumo

Vimos um cliente e um servidor SCTP simples abrangendo aproximadamente 150 linhas de código. Tanto o cliente como o servidor utilizaram a interface SCTP no estilo um para muitos. O servidor foi construído em um estilo iterativo, comum ao utilizar a interface no estilo um para muitos, recebendo e respondendo cada mensagem no mesmo fluxo em que a mensagem foi enviada ou em um fluxo mais alto. Em seguida, examinamos o problema do bloqueio de início de linha. Modificamos nosso cliente para enfatizar o problema e mostrar como os fluxos de SCTP podem ser utilizados para evitá-lo. Examinamos como o número de fluxos pode ser manipulado utilizando uma das muitas opções de soquete disponíveis para controlar o comportamento do SCTP. Por fim, modificamos novamente nossos servidor e cliente de modo que eles possam tanto abortar uma associação incluindo um código do motivo na camada superior do usuário como, no caso do nosso servidor, desativar a associação elegantemente depois de enviar uma mensagem.

Examinaremos o SCTP em mais detalhes no Capítulo 23.

Exercícios

10.1 No código do nosso cliente, mostrado na Figura 10.4, o que aconteceria se o SCTP retornasse um erro? Como você corrigiria esse problema?

10.2 O que acontecerá se nosso servidor terminar antes de responder? Há alguma maneira de o cliente tornar-se ciente disso?

10.3 Na Figura 10.7, linha 22, configuramos `out_sz` como 800 bytes. Por que fizemos isso? Há uma maneira melhor de descobrir um tamanho melhor otimizado para configurar isso?

10.4 Quais efeitos o algoritmo de Nagle (consulte a Seção 7.10) terá no nosso cliente mostrado na Figura 10.7? Desativar o algoritmo de Nagle seria mais apropriado para esse programa? Construa o código de cliente e de servidor, modifique então os dois para desativar o algoritmo de Nagle.

10.5 Na Seção 10.6, afirmamos que uma aplicação deve alterar o número de fluxos antes de configurar uma associação. O que acontece se a aplicação alterar o número de fluxos depois de uma associação?

10.6 Ao modificar o número de fluxos, afirmamos que um soquete no estilo um para muitos é o único estilo que pode utilizar dados auxiliares para solicitar mais fluxos. Por que isso é verdadeiro? (*Dica:* Os dados auxiliares devem ser enviados com uma mensagem.)

10.7 Por que um servidor não pode escapar das conseqüências do monitoramento de associações que estão abertas? Há algum perigo em não monitorar as associações?

10.8 Na Seção 10.7, modificamos o servidor para terminar a associação depois de responder a cada mensagem. Isso causará algum problema? Essa é uma boa decisão de *design*?

Capítulo 11

Conversões de Nomes e Endereços

11.1 Visão geral

Até agora, todos os exemplos neste livro utilizaram endereços numéricos para os hosts (por exemplo, 206.6.226.33) e valores numéricos para endereços de portas numéricas para identificar os servidores (por exemplo, a porta 13 para o servidor de data/hora padrão e a porta 9877 para nosso servidor de eco). Devemos, porém, utilizar nomes em vez de números por várias razões: nomes são mais fáceis de lembrar; o endereço numérico pode mudar, mas o nome pode permanecer o mesmo; e com a mudança para o IPv6, os endereços numéricos tornaram-se bem mais longos, tornando a inserção manual de um endereço mais propensa a erros. Este capítulo descreverá as funções que fazem a conversão entre nomes e valores numéricos: `gethostbyname` e `gethostbyaddr` para converter entre endereços IPv4 e hostnames; e `getservbyname` e `getservbyport` para converter entre nomes de serviços e números de portas. Este capítulo também descreverá duas funções independentes de protocolo: `getaddrinfo` e `getnameinfo`, que convertem entre hostnames e endereços IP *e* entre nomes de serviços e números de portas.

11.2 Domain Name System (DNS)

O DNS é utilizado principalmente para mapear entre hostnames e endereços IP. Um hostname pode ser um *nome simples*, como `solaris` ou `freebsd`, ou um *nome completamente qualificado de domínio* (*fully qualified domain name* – FQDN), como `solaris.unpbook.com`.

> Tecnicamente, um FQDN também é denominado *nome absoluto* e deve terminar com um ponto, mas os usuários costumam omitir esse ponto de término. O ponto final informa ao resolvedor que esse nome é completamente qualificado e ele não precisa pesquisar a lista dos possíveis domínios.

Nesta seção, discutiremos somente os princípios básicos do DNS que precisamos para programação de rede. Os leitores interessados em detalhes adicionais devem consultar o Capítulo 14 do TCPv1 e Albitz e Liu (2001). As adições requeridas ao IPv6 estão na RFC 1886 (Thomson e Huitema, 1995) e na RFC 3152 (Bush, 2001).

Registros de recursos

Entradas no DNS são conhecidas como *registros de recursos* (*resource records* – RRs). Há apenas alguns tipos de RRs em que estamos interessados.

A Um registro A mapeia um hostname para um endereço IPv4 de 32 bits. Por exemplo, eis os quatro registros de DNS para o host `freebsd` no domínio `unpbook.com`, o primeiro dos quais é um A:

```
freebsd   IN    A       12.106.32.254
          IN    AAAA    3ffe:b80:1f8d:1:a00:20ff:fea7:686b
          IN    MX      5  freebsd.unpbook.com.
          IN    MX      10 mailhost.unpbook.com.
```

AAAA Um registro AAAA, denominado registro "quad A", mapeia um hostname para um endereço IPv6 de 128 bits. O termo "quad A" foi escolhido porque um endereço de 128 bits é quatro vezes maior que um de 32 bits.

PTR Registros PTR (denominados "registros de ponteiro") mapeiam endereços IP para hostnames. Para um endereço IPv4, os 4 bytes do endereço de 32 bits são invertidos, cada byte é convertido no valor ASCII decimal (0-255) e `in-addr.arpa` é então acrescentado. A string resultante é utilizada na consulta do PTR.

Para um endereço IPv6, os 4 bytes de 32 nibbles do endereço de 128 bits são invertidos, cada nibble é convertido no valor ASCII hexadecimal correspondente (`0-9a-f`) e `ip6.arpa` é acrescentado.

Por exemplo, os dois registros de PTR para nosso host `freebsd` seriam `254.32.106.12.in-addr.arpa` e `b.6.8.6.7.a.e.f.f.f.0.2.0.0.a.0.1.0.0.0.d.8.f.1.0.8.b.0.e.f.f.3.ip6.arpa`.

> Os padrões iniciais especificavam que endereços IPv6 fossem pesquisados no domínio `ip6.int`. Esses padrões foram alterados para `ip6.arpa` para compatibilidade com o IPv4. Haverá um período de transição no qual as duas zonas serão preenchidas.

MX Um registro MX especifica um host que atua como um "exchanger de mensagens servidor de correio eletrônico" para o host especificado. No exemplo do host `freebsd` anterior, dois registros MX são fornecidos: o primeiro tem um valor de preferência de 5 e o segundo um valor de preferência de 10. Se houver múltiplos registros MX, eles serão utilizados na ordem de preferência, iniciando com o menor valor.

> Não utilizamos registros MX neste livro, mas os mencionamos porque são largamente utilizados no mundo real.

CNAME CNAME significa "nome canônico". Uma utilização comum é para atribuir registros CNAME a serviços comuns, como `ftp` e `www`. Se as pessoas utilizarem esses nomes de serviços em vez dos hostnames reais, o processo será transparente quando um serviço é movido para um outro host. Por exemplo, os seguintes poderiam ser CNAMEs para nosso host `linux`:

```
ftp IN CNAME linux.unpbook.com.
www IN CNAME linux.unpbook.com.
```

A fase de distribuição do IPv6 ainda está no início para sabermos quais convenções os administradores utilizarão para hosts que suportam tanto o IPv4 como o IPv6. No nosso exemplo anterior nesta seção, especificamos tanto um registro A como um registro AAAA para nosso host `freebsd`. Uma possibilidade é posicionar o registro A e o registro AAAA sob o nome normal do host (como mostrado anteriormente) e criar outro RR cujo nome termine em `-4` contendo o registro A, um outro RR cujo nome termine em `-6` contendo o registro AAAA e um outro RR cujo nome termine em `-6ll` contendo um registro AAAA com o endereço do

enlace local do host (o que, às vezes, é útil para propósitos de depuração). Todos os registros para outros hosts são, então:

```
aix         IN    A        192.168.42.2
            IN    AAAA     3ffe:b80:1f8d:2:204:acff:fe17:bf38
            IN    MX       5 aix.unpbook.com.
            IN    MX       10 mailhost.unpbook.com.
aix-4       IN    A        192.168.42.2
aix-6       IN    AAAA     3ffe:b80:1f8d:2:204:acff:fe17:bf38
aix-6ll     IN    AAAA              fe80::204:acff:fe17:bf38
```

Isso fornece controle adicional sobre o protocolo escolhido por algumas aplicações, como veremos no próximo capítulo.

Resolvedores (*resolvers*) e servidores de nomes

As empresas executam um ou mais *servidores de nomes*, freqüentemente um programa conhecido como BIND (Berkeley Internet Name Domain). Aplicações como os clientes e os servidores, que estamos escrevendo neste capítulo, contatam um servidor de DNS chamando funções em uma biblioteca conhecidas como *resolvedoras (resolver)*. As funções *resolvedoras* comuns são `gethostbyname` e `gethostbyaddr`, descritas neste capítulo. A primeira mapeia um hostname para seus endereços IPv4 e a última faz o mapeamento inverso.

A Figura 11.1 mostra um arranjo típico de aplicações, funções resolvedoras e servidores de nomes. Agora escrevemos o código da aplicação. Em alguns sistemas, o código da função resolvedora está contido em uma biblioteca do sistema e é linkeditada quando a aplicação é construída. Em outros, há um daemon resolvedor centralizado que todas as aplicações compartilham, e o código da biblioteca do sistema realiza as RPCs para ele. Em qualquer caso, o código da aplicação chama o código resolvedor utilizando chamadas de função normais, em geral chamando as funções `gethostbyname` e `gethostbyaddr`.

O código resolvedor lê os arquivos de configuração dependentes do sistema para determinar a localização dos servidores de nomes da organização. (Utilizamos "servidores de nomes" no plural porque a maioria das organizações executa múltiplos servidores de nomes, ainda que mostremos somente um servidor local na figura. Múltiplos servidores de nomes são

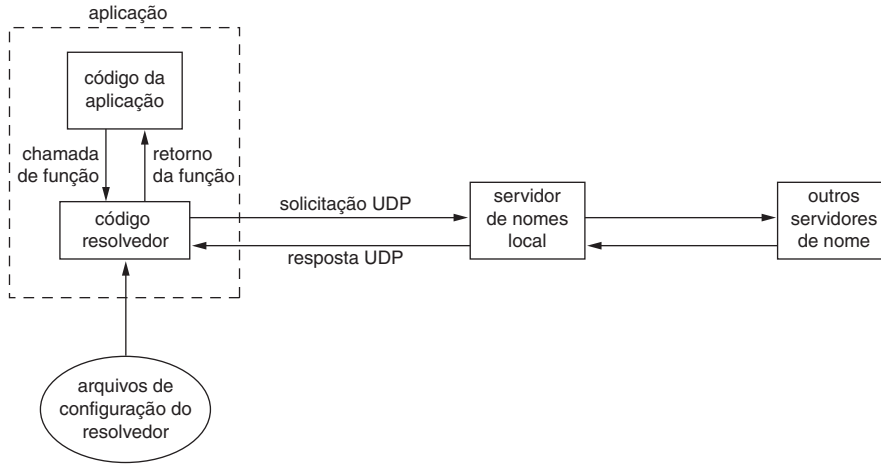


Figura 11.1 Arranjo típico de clientes, resolvedores e servidores de nomes.

absolutamente exigidos para confiabilidade e redundância.) O arquivo `/etc/resolv.conf` normalmente contém os endereços IP dos servidores de nomes locais.

> Talvez seja adequado utilizar os nomes dos servidores de nomes no arquivo `/etc/resolv.conf`, uma vez que nomes são mais fáceis de lembrar e configurar, mas isso introduz um problema do tipo "quem veio antes: o ovo ou a galinha" sobre onde ir para a conversão de nome para endereço quanto ao servidor que fará a conversão entre nome e endereço!

O resolvedor envia a consulta ao servidor de nomes local utilizando o UDP. Se o servidor de nomes local não souber a resposta, normalmente ele consultará outros servidores de nomes na Internet, também utilizando o UDP. Se as respostas forem muito grandes para se encaixarem em um pacote UDP, o resolvedor alternará automaticamente para o TCP.

Alternativas de DNS

É possível obter informações sobre o nome e o endereço sem utilizar o DNS. Alternativas comuns são arquivos de host estáticos (normalmente o arquivo `/etc/hosts`, como descrito na Figura 11.21), o Network Information System (NIS) ou o Lightweight Directory Access Protocol (LDAP). Infelizmente, a maneira como um administrador configura um host para utilizar os diferentes tipos de serviços de nome é dependente da implementação. O Solaris 2.x, o HP-UX 10 e superiores posteriores e o FreeBSD 5.x e superiores posteriores utilizam o arquivo `/etc/nsswitch.conf` e o AIX utiliza o arquivo `/etc/netsvc.conf`. O BIND 9.2.2 fornece sua própria versão, identificada como Information Retrieval Service (IRS), que utiliza o arquivo `/etc/irs.conf`. Se um servidor de nomes for utilizado para pesquisas de hostname, todos esses sistemas utilizarão o arquivo `/etc/resolv.conf` para especificar os endereços IP dos servidores de nomes. Felizmente, essas diferenças não costumam ser mostradas para o programador da aplicação, portanto, simplesmente chamamos as funções resolvedoras como `gethostbyname` e `gethostbyaddr`.

11.3 Função `gethostbyname`

Os computadores host normalmente são conhecidos por nomes legíveis por humanos. Todos os exemplos que mostramos até agora utilizaram intencionalmente endereços IP em vez de nomes, assim sabemos exatamente o que entra nas estruturas de endereço de soquete para funções como `connect` e `sendto` e o que retorna das funções como `accept` e `recvfrom`. Mas a maioria das aplicações deve lidar com nomes, não com endereços. Isso é especialmente verdade à medida que nos movemos para o IPv6, uma vez que os endereços IPv6 (strings hexadecimais) são muito mais longos que os números decimais com pontos decimais do IPv4. (Os exemplos do registro AAAA e do registro PTR, `ip6.arpa`, na seção anterior, devem tornar isso óbvio.)

A função mais básica que pesquisa um hostname é `gethostbyname`. Se bem-sucedida, essa função retorna um ponteiro para uma estrutura `hostent` que contém todos os endereços IPv4 para o host. Entretanto, ela é limitada pelo fato de que pode retornar apenas endereços IPv4. Consulte na Seção 11.6 uma função que trata tanto endereços IPv4 como IPv6. A especificação POSIX adverte que `gethostbyname` pode ser removida em uma versão futura da especificação.

> Na verdade, é improvável que implementações de `gethostbyname` desaparecerão até que a Internet inteira utilize o IPv6, o que ocorrerá em um futuro distante. Entretanto, remover a função da especificação POSIX é uma maneira de afirmar que ela não deve ser utilizada em novos códigos. Encorajamos o uso de `getaddrinfo` (Seção 11.6) nos novos programas.

```
#include <netdb.h>

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

Retorna: ponteiro não-nulo se OK, NULL em erro com `h_errno` configurado

O ponteiro não-nulo retornado por essa função aponta para esta estrutura `hostent`:

```
struct hostent {
    char   *h_name;       /* nome oficial (canônico) do host */
    char  **h_aliases;    /* ponteiro para array de ponteiros para nomes
                             alternativos (alias) */
    int     h_addrtype;   /* host address: AF_INET */
    int     h_length;     /* comprimento do endereço: 4 */
    char  **h_addr_list;  /* ptr para array de ptrs com endereços IPv4 */
};
```

Em termos do DNS, `gethostbyname` realiza uma consulta em um registro A. Essa função pode retornar somente endereços IPv4.

A Figura 11.2 mostra o arranjo e as informações da estrutura `hostent` e as informações às quais ela aponta supondo que o hostname pesquisado tem dois nomes de alias e três endereços IPv4. Entre esses campos, o hostname oficial e todos os aliases são strings C terminadas por um caractere nulo.

O `h_name` retornado é denominado nome *canônico* do host. Por exemplo, com os registros CNAME mostrados na seção anterior, o nome canônico do host `ftp.unpbook.com` seria `linux.unpbook.com`. Além disso, se chamarmos `gethostbyname` a partir do host `aix` com um hostname não-qualificado, digamos `solaris`, o FQDN (`solaris.unpbook.com`) retorna como o nome canônico.

> Algumas versões de `gethostbyname` permitem que o argumento *hostname* seja uma string de decimais com pontos. Isto é, uma chamada na forma
>
> ```
> hptr = gethostbyname("192.168.42.2");
> ```
>
> funcionará. Esse código foi adicionado porque o cliente Rlogin aceita somente um hostname, chamando `gethostbyname`, e não aceitará uma string de decimais com pontos (Vixie, 1996). A especificação POSIX permite, mas não exige, esse comportamento, portanto, uma aplicação portável não pode depender dele.

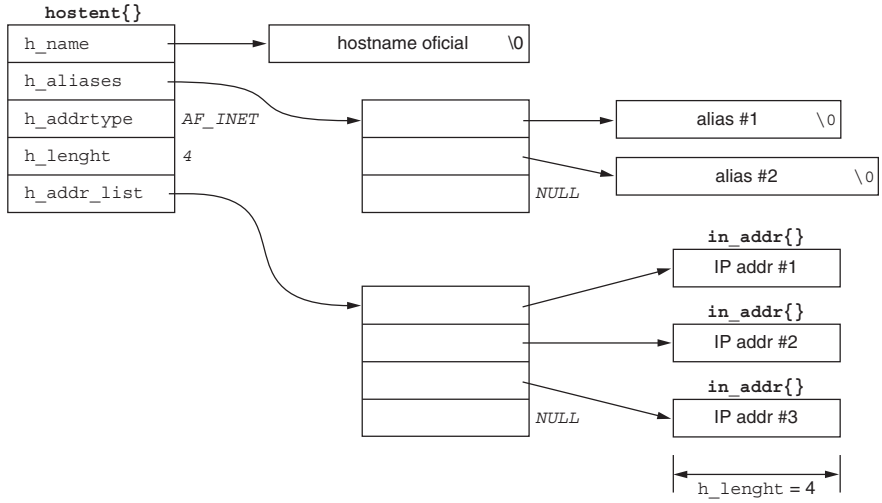


Figura 11.2 A estrutura `hostent` e as informações que contém.

`gethostbyname` difere das outras funções de soquete que descrevemos pelo fato de que ela não configura `errno` quando um erro ocorre. Em vez disso, ela configura o inteiro global, `h_errno`, como uma das seguintes constantes definidas incluindo `<netdb.h>`:

- `HOST_NOT_FOUND`
- `TRY_AGAIN`
- `NO_RECOVERY`
- `NO_DATA` (idêntico a `NO_ADDRESS`)

O erro `NO_DATA` significa que o nome especificado é válido, mas não tem um registro A. Um exemplo disso é um hostname com somente um registro MX. Resolvedores mais modernos fornecem a função `hstrerror`, que recebe um valor `h_errno` como seu único argumento e que retorna um ponteiro `const char *` para uma descrição do erro. Mostraremos alguns exemplos das strings retornadas por essa função no próximo exemplo.

Exemplo

A Figura 11.3 mostra um programa simples que chama `gethostbyname` para qualquer número de argumentos da linha de comando e imprime todas as informações retornadas.

names/hostent.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     char   *ptr, **pptr;
 6     char   str[INET_ADDRSTRLEN];
 7     struct hostent *hptr;
 8     while (--argc > 0) {
 9         ptr = *++argv;
10         if ( (hptr = gethostbyname(ptr)) == NULL) {
11            err_msg("gethostbyname error for host: %s: %s",
12                    ptr, hstrerror(h_errno));
13            continue;
14         }
15         printf("official hostname: %s\n", hptr->h_name);
16         for (pptr = hptr->h_aliases; *pptr != NULL; pptr++)
17            printf("\talias: %s\n", *pptr);
18         switch (hptr->h_addrtype) {
19         case AF_INET:
20            pptr = hptr->h_addr_list;
21            for ( ; *pptr != NULL; pptr++)
22                printf("\taddress: %s\n",
23                       Inet_ntop(hptr->h_addrtype, *pptr, str, sizeof(str)));
24            break;
25         default:
26            err_ret("unknown address type");
27            break;
28         }
29     }
30     exit(0);
31 }
```

names/hostent.c

Figura 11.3 Chama `gethostbyname` e imprime as informações retornadas.

8–14 `gethostbyname` é chamada para cada argumento da linha de comando.

15–17 O hostname oficial é gerado seguido por uma lista de nomes de alias.

18–24 `pptr` aponta para o array de ponteiros para os endereços individuais. Para cada endereço, chamamos `inet_ntop` e imprimimos a string retornada.

Primeiro, executamos o programa com o nome do nosso host `aix`, que tem somente um endereço IPv4.

```
freebsd % hostent aix
official hostname: aix.unpbook.com
        address: 192.168.42.2
```

Observe que o hostname oficial é o FQDN. Também observe que, mesmo que esse host tenha um endereço IPv6, somente o endereço IPv4 retorna.

Em seguida, há um servidor Web com múltiplos endereços IPv4.

```
freebsd % hostent cnn.com
official hostname: cnn.com
        address: 64.236.16.20
        address: 64.236.16.52
        address: 64.236.16.84
        address: 64.236.16.116
        address: 64.236.24.4
        address: 64.236.24.12
        address: 64.236.24.20
        address: 64.236.24.28
```

Em seguida, há um nome que mostramos na Seção 11.2 com um registro CNAME.

```
freebsd % hostent www
official hostname: linux.unpbook.com
        alias: www.unpbook.com
        address: 206.168.112.219
```

Como esperado, o hostname oficial difere do nosso argumento de linha de comando.

Para ver as strings de erro retornadas pela função `hstrerror`, primeiro especificamos um hostname inexistente e então um nome com apenas um registro MX.

```
freebsd % hostent nosuchname.invalid
gethostbyname error for host: nosuchname.invalid: Unknown host

freebsd % hostent uunet.uu.net
gethostbyname error for host: uunet.uu.net: No address associated with name
```

11.4 Função `gethostbyaddr`

A função `gethostbyaddr` aceita um endereço IPv4 binário e tenta encontrar o hostname correspondente a esse endereço. Isso é o contrário de `gethostbyname`.

```
#include <netdb.h>

struct hostent *gethostbyaddr(const char *addr, socklen_t len, int fa-
mily);
```

Retorna: ponteiro não-nulo se OK, `NULL` em erro com `h_errno` configurado

Essa função retorna um ponteiro para a mesma estrutura `hostent` que descrevemos com `gethostbyname`. O campo de interesse nessa estrutura normalmente é `h_name`, o hostname canônico.

O argumento *addr* não é um `char*`, mas na verdade um ponteiro para uma estrutura `in_addr` que contém o endereço IPv4. *len* é o tamanho dessa estrutura: 4 para um endereço IPv4. O argumento *family* é `AF_INET`.

Em termos do DNS, `gethostbyaddr` consulta um servidor de nomes sobre um registro PTR no domínio `in-addr.arpa`.

11.5 Funções `getservbyname` e `getservbyport`

Como ocorre com os hosts, os serviços também são freqüentemente conhecidos pelos nomes. Se nos referirmos a um serviço pelo nome no nosso código, em vez do número de porta, e se o mapeamento do nome para o número da porta estiver contido em um arquivo (normalmente, `/etc/services`), então, se o número da porta mudar, tudo o que precisamos modificar é uma linha no arquivo `/etc/services` em vez de recompilar as aplicações. A próxima função, `getservbyname`, pesquisa um serviço pelo nome.

> A lista canônica de números de porta atribuídos aos serviços é mantida pela IANA em `http://www.iana.org/assignments/port-numbers` (Seção 2.9). Um dado arquivo `/etc/services` possivelmente contém um subconjunto das atribuições da IANA.

```
#include <netdb.h>

struct servent *getservbyname(const char *servname, const char *protoname);
```

Retorna: ponteiro não-nulo se OK, NULL em erro

Essa função retorna um ponteiro para a estrutura a seguir:

```
struct servent {
   char   *s_name;       /* nome oficial do serviço */
   char  **s_aliases;    /* lista de aliases */
   int     s_port;       /* número da porta, ordem de byte de rede */
   char   *s_proto;      /* protocolo a utilizar */
};
```

O nome de serviço, *servname,* deve ser especificado. Se um protocolo também for especificado (*protoname* é um ponteiro não-nulo), então a entrada também deverá ter um protocolo correspondente. Alguns serviços de Internet são fornecidos utilizando o TCP ou o UDP (por exemplo, o DNS e todos os serviços na Figura 2.18), enquanto outros suportam somente um único protocolo (por exemplo, FTP requer TCP). Se *protoname* não estiver especificado e o serviço suportar múltiplos protocolos, ele é dependente de implementação, como qual número de porta é retornado. Normalmente isso não tem importância, porque os serviços que suportam múltiplos protocolos costumam utilizar o mesmo número de porta TCP e UDP, mas isso não é garantido.

O principal campo de interesse na estrutura `servent` é o número de porta. Como o número de porta retorna na ordem de bytes de rede, não devemos chamar `htons` ao armazená-lo em uma estrutura de endereço de soquete.

Chamadas típicas a essa função poderiam ser estas:

```
struct servent *sptr;

sptr = getservbyname("domain", "udp");      /* DNS utilizando UDP */
sptr = getservbyname("ftp", "tcp");         /* FTP utilizando TCP */
sptr = getservbyname("ftp", NULL);          /* FTP utilizando TCP */
sptr = getservbyname("ftp", "udp");         /* esta chamada falhará */
```

Como FTP suporta somente TCP, a segunda e a terceira chamadas são as mesmas e a quarta chamada falhará. Linhas típicas do arquivo `/etc/services` são:

```
freebsd % grep -e ^ftp -e ^domain /etc/services
ftp-data         20/tcp    #File Transfer [Default Data]
ftp              21/tcp    #File Transfer [Control]
domain           53/tcp    #Domain Name Server
domain           53/udp    #Domain Name Server
ftp-agent       574/tcp    #FTP Software Agent System
ftp-agent       574/udp    #FTP Software Agent System
ftps-data       989/tcp                # ftp protocol, data, over TLS/SSL
ftps            990/tcp                # ftp protocol, control, over TLS/SSL
```

A próxima função, `getservbyport`, pesquisa um serviço pelo número de porta e um protocolo opcional.

```
#include <netdb.h>

struct servent *getservbyport(int port, const char *protoname);
```

Retorna: ponteiro não-nulo se OK, NULL em erro

O valor de *port* deve ser o da ordem de bytes de rede. Chamadas típicas a essa função poderiam ser estas:

```
struct servent *sptr;

sptr = getservbyport(htons(53), "udp");     /* DNS utilizando UDP */
sptr = getservbyport(htons(21), "tcp");     /* FTP utilizando TCP */
sptr = getservbyport(htons(21), NULL);      /* FTP utilizando TCP */
sptr = getservbyport(htons(21), "udp");     /* esta chamada falhará */
```

A última chamada falha porque não há nenhum serviço que utiliza a porta 21 com o UDP.

Esteja ciente de que alguns números de porta são utilizados com o TCP para um serviço, mas o mesmo número de porta é utilizado com o UDP para um serviço totalmente diferente. Por exemplo, o seguinte:

```
freebsd % grep 514 /etc/services
shell           514/tcp     cmd        #como exec, mas automático
syslog          514/udp
```

mostra que a porta 514 é utilizada pelo comando `rsh` com o TCP, mas com o daemon `syslog` com o UDP. As portas 512 a 514 têm essa propriedade.

Exemplo: Utilizando `gethostbyname` e `getservbyname`

Agora, podemos modificar nosso cliente TCP de data/hora na Figura 1.5 para utilizar `gethostbyname` e `getservbyname` e receber dois argumentos de linha de comando: um hostname e um nome de serviço. A Figura 11.4 mostra nosso programa. Esse programa também mostra o comportamento desejado da tentativa de conectar todos os endereços IP a um servidor multihomed, até que uma tenha sido bem-sucedida ou todos os endereços tenham sido tentados.

names/daytimetcpcli1.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    sockfd, n;
 6     char   recvline[MAXLINE + 1];
 7     struct sockaddr_in servaddr;
 8     struct in_addr **pptr;
 9     struct in_addr *inetaddrp[2];
10     struct in_addr inetaddr;
11     struct hostent *hp;
12     struct servent *sp;
13     if (argc != 3)
14         err_quit("usage: daytimetcpcli1 <hostname> <service>");
15     if ( (hp = gethostbyname(argv[1])) == NULL) {
16         if (inet_aton(argv[1], &inetaddr) == 0) {
17             err_quit("hostname error for %s: %s", argv[1],
18                      hstrerror(h_errno));
19         } else {
20             inetaddrp[0] = &inetaddr;
21             inetaddrp[1] = NULL;
22             pptr = inetaddrp;
23         }
24     } else {
25         pptr = (struct in_addr **) hp->h_addr_list;
26     }
27     if ( (sp = getservbyname(argv[2], "tcp")) == NULL)
28         err_quit("getservbyname error for %s", argv[2]);
29     for ( ; *pptr != NULL; pptr++) {
30         sockfd = Socket(AF_INET, SOCK_STREAM, 0);
31         bzero(&servaddr, sizeof(servaddr));
32         servaddr.sin_family = AF_INET;
33         servaddr.sin_port = sp->s_port;
34         memcpy(&servaddr.sin_addr, *pptr, sizeof(struct in_addr));
35         printf("trying %s\n", Sock_ntop((SA *) &servaddr, sizeof(servaddr)));
36         if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) == 0)
37             break;              /* sucesso */
38         err_ret("connect error");
39         close(sockfd);
40     }
41     if (*pptr == NULL)
42         err_quit("unable to connect");
43     while ( (n = Read(sockfd, recvline, MAXLINE)) > 0) {
44         recvline[n] = 0;        /* termina em nulo */
45         Fputs(recvline, stdout);
46     }
47     exit(0);
48 }
```

names/daytimetcpcli1.c

Figura 11.4 Nosso cliente de data/hora que utiliza `gethostbyname` e `getservbyname`.

Chamada a `gethostbyname` e `getservbyname`

13–28 O primeiro argumento da linha de comando é um hostname, que passamos como um argumento para `gethostbyname`; e o segundo é um nome de serviço, que passamos como um

argumento para `getservbyname`. Nosso código assume o TCP, é isso o que utilizamos como o segundo argumento para `getservbyname`. Se `gethostbyname` não conseguir pesquisar o nome, tentaremos utilizar a função `inet_aton` (Seção 3.6) para ver se o argumento era um endereço no formato ASCII. Se era, construímos uma lista de um único elemento que consiste no endereço correspondente.

Tentando cada endereço de servidor

29–35 Agora, codificamos as chamadas para `socket` e `connect` em um loop que é executado para cada endereço de servidor até que uma `connect` seja bem-sucedida ou a lista de endereços IP tiver sido exaurida. Depois de chamar `socket`, preenchemos uma estrutura de endereço de soquete de Internet com o endereço IP e a porta do servidor. Embora possamos mover para fora do loop a chamada a `bzero` e as duas atribuições subseqüentes fora do loop, visando eficiência, o código é mais fácil de ler da maneira como mostrado. Estabelecer a conexão com o servidor é raramente um gargalo de desempenho para um cliente de rede.

Chamada a `connect`

36–39 `connect` é chamada e, se for bem-sucedida, `break` termina o loop. Se o estabelecimento de conexão falhar, imprimimos um erro e fechamos o soquete. Lembre-se de que um descritor que não consegue estabelecer uma chamada a `connect` deverá ser fechado e não será mais utilizável.

Verificação de falhas

41–42 Se o loop terminar porque nenhuma chamada a `connect` foi bem-sucedida, o programa termina.

Leitura da resposta do servidor

43–47 Caso contrário, lemos a resposta do servidor, terminando quando este fecha a conexão.

Se executarmos esse programa especificando um dos nossos hosts que está executando o servidor de data/hora, obteremos a saída esperada.

```
freebsd % daytimetcpcli1 aix daytime
trying 192.168.42.2:13
Sun Jul 27 22:44:19 2003
```

O mais interessante é executar o programa para um sistema com multihome que não está executando o servidor de data/hora.

```
freebsd % daytimetcpcli1 gateway.tuc.noao.edu daytime
trying 140.252.108.1:13
connect error: Operation timed out
trying 140.252.1.4:13
connect error: Operation timed out
trying 140.252.104.1:13
connect error: Connection refused
unable to connect
```

11.6 Função `getaddrinfo`

As funções `gethostbyname` e `gethostbyaddr` suportam apenas o IPv4. A API para converter endereços IPv6 passou por várias iterações, como descrito na Seção 11.20; o resultado final é a função `getaddrinfo`. A função `getaddrinfo` trata tanto da conversão entre nome e endereço como entre serviço e porta e retorna as estruturas `sockaddr` em vez de uma lista de endereços. As estruturas `sockaddr` podem então ser utilizadas pelas funções de soquete diretamente. Dessa maneira, a função `getaddrinfo` oculta todas as dependências de protocolo na função de biblioteca, o local em que deveriam estar. A aplicação lida apenas

com as estruturas de endereço de soquete que são preenchidas por `getaddrinfo`. Essa função é definida na especificação POSIX.

> A definição POSIX dessa função vem de uma proposta anterior por Keith Sklower para uma função identificada como `getconninfo`. Essa função foi o resultado de discussões com Eric Allman, William Durst, Michael Karels e Steven Wise e de uma implementação inicial escrita por Eric Allman. A observação de que especificar um hostname e um nome de serviço bastaria para conectar um serviço independentemente dos detalhes de protocolo foi feita por Marshall Rose como uma proposta ao X/Open.

```
#include <netdb.h>

int getaddrinfo(const char *hostname, const char *service,
                const struct addrinfo *hints, struct addrinfo **result);
```

Retorna: 0 se OK, não-zero em erro (veja a Figura 11.7)

Essa função retorna, por meio do ponteiro *result,* um ponteiro para uma lista vinculada de estruturas `addrinfo`, definidas incluindo `<netdb.h>`.

```
struct addrinfo {
   int       ai_flags;        /* AI_PASSIVE, AI_CANONNAME */
   int       ai_family;       /* AF_xxx */
   int       ai_socktype;     /* SOCK_xxx */
   int       ai_protocol;     /* 0 ou IPPROTO_xxx para IPv4 e IPv6 */
   socklen_t ai_addrlen;      /* comprimento de ai_addr */
   char     *ai_canonname;    /* ptr para nome canônico do host */
   struct sockaddr *ai_addr;  /* ptr para estrutura de endereço de
                                 soquete */
   struct addrinfo *ai_next;  /* ptr para próxima estrutura em lista
                                 vinculada */
};
```

O *hostname* é um nome de host ou uma string de endereço (em decimais com pontos para o IPv4 ou uma string hexadecimal para o IPv6). O *service* é um nome de serviço ou uma string decimal do número de porta. (Consulte também o Exercício 11.4, em que queremos permitir uma string de endereço para o host ou uma string de número de porta para o serviço.)

hints é um ponteiro nulo ou um ponteiro para uma estrutura `addrinfo` que o chamador preenche com *dicas* sobre as informações que ele quer que retornem. Por exemplo, se o serviço especificado for oferecido tanto para o TCP como para o UDP (por exemplo, o serviço `domain`, que se refere a um servidor de DNS), o chamador pode configurar o membro `ai_socktype` da estrutura *hints* como `SOCK_DGRAM`. As únicas informações retornadas serão para os soquetes de datagrama.

Os membros da estrutura *hints* que podem ser configurados pelo chamador são:

- `ai_flags` (zero ou mais valores `AI_`*XXX unidos por* "*OU*")
- `ai_family` (um valor `AF_` *xxx*)
- `ai_socktype` (um valor `SOCK_` *xxx*)
- `ai_protocol`

Os possíveis valores para o membro `ai_flags` e seus significados são:

`AI_PASSIVE`	O chamador utilizará o soquete para um open passivo.
`AI_CANONNAME`	Informa à função para retornar o nome canônico do host.
`AI_NUMERICHOST`	Evita qualquer tipo de mapeamento entre nome e endereço; o argumento *hostname* deve ser uma string de endereço.

`AI_NUMERICSERV` Impede qualquer tipo de mapeamento entre nome e serviço; o argumento *service* deve ser uma string decimal do número de porta.

`AI_V4MAPPED` Se especificado junto com um `ai_family` de `AF_INET6`, retorna os endereços IPv6 mapeados de IPv4 que correspondem aos registros A se não houver nenhum registro AAAA disponível.

`AI_ALL` Se especificado junto com `AI_V4MAPPED`, retorna os endereços IPv6 mapeados de IPv4 além de quaisquer registros AAAA que pertencem ao nome.

`AI_ADDRCONFIG` Somente pesquisa endereços em uma determinada versão IP se houver uma ou mais interfaces que não sejam de loopback configuradas com um endereço IP dessa versão.

Se o argumento *hints* (*dicas*) for um ponteiro nulo, a função assume um valor de 0 para `ai_flags`, `ai_socktype` e `ai_protocol` e um valor de `AF_UNSPEC` para `ai_family`.

Se a função for bem-sucedida (0), a variável apontada pelo argumento *result* é preenchida com um ponteiro para uma lista vinculada de estruturas `addrinfo`, vinculadas por meio do ponteiro `ai_next`. Há duas maneiras como múltiplas estruturas podem retornar:

1. Se houver múltiplos endereços associados ao *hostname*, uma estrutura é retornada para cada endereço que seja utilizável com a família de endereços solicitada (dica `ai_family`, se especificada).
2. Se o serviço for oferecido a múltiplos tipos de soquete, uma estrutura poderá retornar para cada tipo de soquete, dependendo da dica `ai_socktype`. (Observe que a maioria das implementações `getaddrinfo` considera uma string de número de porta a ser implementada somente pelo tipo de soquete solicitado em `ai_socktype`; se `ai_socktype` não estiver especificado, em vez disso, retorna um erro.)

Por exemplo, se nenhuma dica for fornecida e se o serviço `domain` for pesquisado em um host com dois endereços IP, quatro estruturas `addrinfo` retornam:

- Uma para o primeiro endereço IP e um soquete do tipo `SOCK_STREAM`
- Uma para o primeiro endereço IP e um soquete do tipo `SOCK_DGRAM`
- Uma para o segundo endereço IP e um soquete do tipo `SOCK_STREAM`
- Uma para o segundo endereço IP e um soquete do tipo `SOCK_DGRAM`

Mostramos esse exemplo na Figura 11.5. Não há uma ordem garantida de estruturas quando múltiplos itens retornam; isto é, não podemos supor que os serviços TCP retornarão antes dos serviços UDP.

> Embora não haja garantias, uma implementação deverá retornar os endereços IP na mesma ordem em que eles retornam pelo DNS. Alguns resolvedores permitem ao administrador especificar uma ordem de classificação de endereços no arquivo `/etc/resolv.conf`. O IPv6 especifica regras de seleção de endereços (RFC 3484 [Draves, 2003]), que poderiam afetar a ordem dos endereços que retornam por `getaddrinfo`.

As informações que retornam nas estruturas `addrinfo` estão prontas para uma chamada a `socket` e então uma chamada a `connect` ou `sendto` (para um cliente) ou `bind` (para um servidor). Os argumentos para `socket` são os membros `ai_family`, `ai_socktype` e `ai_protocol`. O segundo e o terceiro argumentos para `connect` ou `bind` são `ai_addr` (um ponteiro para uma estrutura de endereço de soquete do tipo apropriado, preenchida por `getaddrinfo`) e `ai_addrlen` (o comprimento dessa estrutura de endereço de soquete).

Se o flag `AI_CANONNAME` estiver configurado na estrutura *hints*, o membro `ai_canonname` da primeira estrutura retornada aponta para o nome canônico do host. Em termos

do DNS, isso normalmente é o FQDN. Programas como `telnet` comumente utilizam esse flag para imprimir o hostname canônico do sistema ao qual estão se conectando, de modo que, se o usuário fornecer um atalho ou um alias, ele saberá o que foi pesquisado.

A Figura 11.5 mostra as informações que retornam se executarmos o seguinte:

```
struct addrinfo hints, *res;

bzero(&hints, sizeof(hints));
hints.ai_flags = AI_CANONNAME;
hints.ai_family = AF_INET;

getaddrinfo("freebsd4", "domain", &hints, &res);
```

Nessa figura, tudo, exceto a variável `res`, é dinamicamente alocado na memória (por exemplo, a partir de `malloc`). Assumimos que o nome canônico do host `freebsd4` é `freebsd4.unpbook.com` e que esse host tem dois endereços IPv4 no DNS.

A porta 53 é para o serviço `domain`. Esse número de porta estará na ordem de bytes de rede nas estruturas de endereços de soquetes. Também mostramos os valores `ai_protocol` retornados como IPPROTO_TCP ou IPPROTO_UDP. Também seria aceitável que `getaddrinfo` retorne um `ai_protocol` em 0 para as duas estruturas `SOCK_STREAM` se isso for suficiente para especificar o TCP (por exemplo, não é suficiente se o sistema implementar o SCTP) e um `ai_protocol` em 0 para as duas estruturas `SOCK_DGRAM` se o sistema não implementar nenhum outro protocolo `SOCK_DGRAM` para o IP (no momento em que este livro estava sendo escrito, nenhum ainda havia sido padronizado, mas dois estavam em desenvolvimento no IETF). É mais seguro que `getaddrinfo` sempre retorne o protocolo específico.

A Figura 11.6 resume o número de estruturas `addrinfo` retornadas para cada endereço que está sendo retornado, com base no nome do serviço especificado (que pode ser um número decimal de porta) e em qualquer dica `ai_socktype`.

Múltiplas estruturas `addrinfo` retornam a cada endereço IP somente quando nenhuma dica `ai_socktype` é fornecida e o nome do serviço é suportado por múltiplos protocolos de transporte (como indicado no arquivo `/etc/services`).

Se enumerássemos todas as 64 entradas possíveis para `getaddrinfo` (há seis variáveis de entrada), muitas seriam inválidas e outras fariam pouco sentido. Em vez disso, examinaremos os casos comuns.

- Especificação do nome de host (*hostname*) e do serviço (*service*). Isso é normal para um cliente TCP ou UDP. No retorno, um cliente TCP faz um loop por todos os endereços IP que retornam, chamando `socket` e `connect` para cada endereço, até que a conexão tenha sido bem-sucedida ou todos os endereços tenham sido tentados. Mostraremos um exemplo disso com a nossa função `tcp_connect` na Figura 11.10.

  Para um cliente UDP, a estrutura de endereço de soquete preenchida por `getaddrinfo` seria utilizada em uma chamada a `sendto` ou `connect`. Se o cliente puder informar que o primeiro endereço aparentemente não funciona (recebendo um erro em um soquete de UDP conectado ou experimentando um tempo-limite em um soquete não-conectado), endereços adicionais poderão ser tentados.

  Se o cliente souber que trata somente um tipo de soquete (por exemplo, clientes Telnet e FTP tratam somente TCP; clientes TFTP tratam somente UDP), então o membro `ai_socktype` da estrutura *hints* deve ser especificado como `SOCK_STREAM` ou `SOCK_DGRAM`.

- Um servidor típico especifica o *service*, mas não o *hostname,* e especifica o flag `AI_PASSIVE` na estrutura *hints*. As estruturas de endereços de soquetes retornadas devem conter um endereço IP de `INADDR_ANY` (para IPv4) ou `IN6ADDR_ANY_INIT` (para IPv6). Um servidor TCP chama então `socket`, `bind` e `listen`. Se o servidor quiser chamar `malloc` para alocar uma outra estrutura de endereço de soquete a fim de obter o endereço do cliente a partir de `accept`, o valor `ai_addrlen` retornado especificará esse tamanho.

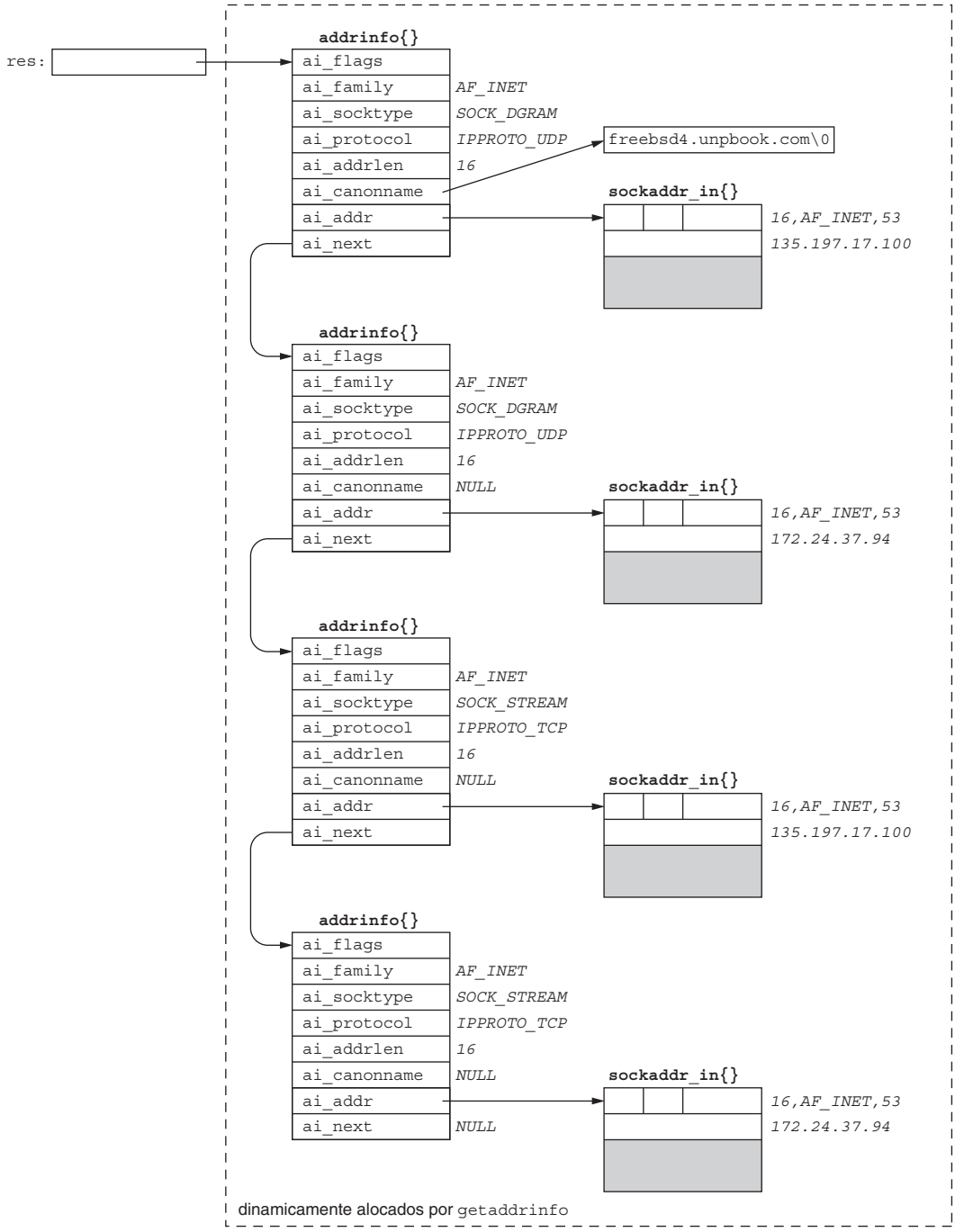


Figura 11.5 Exemplo das informações retornadas por `getaddrinfo`.

Um servidor UDP chamaria `socket`, `bind` e então `recvfrom`. Se o servidor quiser chamar `malloc` para alocar uma outra estrutura de endereço de soquete a fim de obter o endereço do cliente a partir de `recvfrom`, o valor `ai_addrlen` retornado especificará esse tamanho.

O serviço é um nome, e é fornecido por:							
dica `ai_socktype`	TCP somente	UDP somente	SCTP somente	TCP e UDP	TCP e SCTP	TCP, UDP e SCTP	O serviço é um número de porta
0	1	1	1	2	2	3	erro
`SOCK_STREAM`	1	erro	1	1	2	2	2
`SOCK_DGRAM`	erro	1	erro	1	erro	1	1
`SOCK_SEQPACKET`	erro	erro	1	erro	1	1	1

Figura 11.6 O número de estruturas `addrinfo` retornadas por endereço IP.

Como ocorre com o código de cliente típico, se o servidor souber que ele trata apenas um tipo de soquete, o membro `ai_socktype` da estrutura *hints* deve ser configurado como `SOCK_STREAM` ou `SOCK_DGRAM`. Isso evita que múltiplas estruturas possam retornar, possivelmente com o valor `ai_socktype` errado.

- Os servidores TCP que mostramos até agora criam um soquete ouvinte e os servidores UDP criam um soquete de datagrama. Isso é o que pressupomos no item anterior. Um *design* de servidor alternativo é o servidor tratar múltiplos soquetes utilizando `select` ou `poll`. Nesse cenário, o servidor passaria pela lista inteira de estruturas retornadas por `getaddrinfo`, criaria um soquete por estrutura e utilizaria `select` ou `poll`.

  O problema dessa técnica é que uma das razões de `getaddrinfo` retornar múltiplas estruturas é quando um serviço puder ser tratado por IPv4 e IPv6 (Figura 11.8). Mas esses dois protocolos não são completamente independentes, como veremos na Seção 12.2. Isto é, se criarmos um soquete IPv6 ouvinte para uma dada porta, não haverá necessidade de criar também um soquete IPv4 ouvinte para essa mesma porta, porque as conexões provenientes de clientes IPv4 são automaticamente tratadas pela pilha de protocolos e pelo soquete IPv6 ouvinte, supondo que a opção de soquete `IPV6_V6ONLY` não esteja configurada.

Apesar de `getaddrinfo` ser "melhor" que as funções `gethostbyname` e `getservbyname` (ela torna mais fácil escrever um código independente de protocolo; uma função trata ambos, o hostname e o serviço; todas as informações que retornam são dinamicamente alocadas, não estaticamente alocadas), ainda não é tão fácil utilizá-lo quanto poderia ser. O problema é que devemos alocar uma estrutura *hints*, inicializá-la como 0, preencher os campos desejados, chamar `getaddrinfo` e então percorrer uma lista vinculada tentando cada um. Nas seções a seguir, forneceremos algumas interfaces mais simples para clientes e servidores TCP e UDP típicos que escreveremos no restante deste capítulo.

`getaddrinfo` resolve o problema da conversão entre hostnames e nomes de serviço nas estruturas de endereços de soquetes. Na Seção 11.17, descreveremos a função inversa, `getnameinfo`, que converte estruturas de endereços de soquetes em hostnames e nomes de serviço.

11.7 Função `gai_strerror`

Os valores de retorno de erro não-zero provenientes de `getaddrinfo` têm os nomes e significados mostrados na Figura 11.7. A função `gai_strerror` recebe um desses valores como um argumento e retorna um ponteiro para a string de erro correspondente.

```
#include <netdb.h>

const char *gai_strerror(int error);

                                    Retorna: ponteiro para string que descreve a mensagem de erro
```

Constante	Descrição
EAI_AGAIN	Falha temporária na resolução de nome
EAI_BADFLAGS	Valor inválido para ai_flags
EAI_FAIL	Falha irrecuperável na resolução de nome
EAI_FAMILY	ai_family não-suportado
EAI_MEMORY	Falha de alocação de memória
EAI_NONAME	*hostname* ou *serviço* não fornecido, ou não conhecido
EAI_OVERFLOW	Buffer de argumento de usuário esgotado (somente *getnameinfo*())
EAI_SERVICE	*service* não suportado para ai_socktype
EAI_SOCKTYPE	ai_socktype não-suportado
EAI_SYSTEM	Erro de sistema retornado em errno

Figura 11.7 Constantes de erro não-zero retornadas a partir de getaddrinfo.

11.8 Função freeaddrinfo

Todo o armazenamento retornado pelas estruturas getaddrinfo, addrinfo, ai_addr e a string ai_canonname é obtido dinamicamente (por exemplo, a partir de malloc). Esse armazenamento é retornado chamando freeaddrinfo.

```
#include <netdb.h>

void freeaddrinfo(struct addrinfo *ai);
```

ai deve apontar para a primeira estrutura addrinfo retornada por getaddrinfo. Todas as estruturas na lista vinculada são liberadas, junto com qualquer armazenamento dinâmico apontado para elas (por exemplo, as estruturas de endereços de soquetes e hostnames canônicos).

Suponha que chamamos getaddrinfo, para percorrer a lista vinculada de estruturas addrinfo e encontrar a estrutura desejada. Se, em seguida, tentarmos salvar uma cópia das informações copiando somente a estrutura addrinfo e chamando freeaddrinfo, teremos um bug escondido. A razão é que a própria estrutura addrinfo aponta para a memória alocada dinamicamente (para a estrutura de endereço de soquete e possivelmente para o nome canônico) e a memória apontada pela nossa estrutura salva é retornada para o sistema quando freeaddrinfo é chamado e pode ser utilizada para algo mais.

> A ação de copiar apenas a estrutura addrinfo e não as estruturas para as quais ela aponta é chamada de *cópia superficial*. A ação de copiar a estrutura addrinfo e todas as estruturas para as quais ela aponta é chamada de *cópia profunda*.

11.9 Função getaddrinfo: IPv6

A especificação POSIX define a função getaddrinfo e as informações que retornam tanto ao IPv4 como ao IPv6. Observamos os seguintes pontos antes de resumir esses valores de retorno na Figura 11.8.

- getaddrinfo lida com duas entradas diferentes: o tipo da estrutura de endereço de soquete que o chamador quer de volta e o tipo de registros que devem ser pesquisados no DNS ou em outro banco de dados.
- A família de endereços na estrutura *hints* fornecida pelo chamador especifica o tipo de estrutura de endereço de soquete que o chamador espera que retorne. Se o chamador especificar AF_INET, a função não deve retornar nenhuma estrutura sockaddr_in6; se o

chamador especificar AF_INET6, a função não deve retornar nenhuma estrutura sockaddr_in.

- O POSIX diz que especificar AF_UNSPEC retornará endereços que podem ser utilizados com *qualquer* família de protocolos que possa ser utilizada com o hostname e o nome de serviço. Isso implica que, se um host tiver tanto registros AAAA como registros A, os registros AAAA retornarão como estruturas sockaddr_in6 e os registros A retornarão como estruturas sockaddr_in. Não faz sentido também retornar os registros A como endereços IPv6 mapeados de IPv4 nas estruturas sockaddr_in6 porque nenhuma informação adicional está sendo retornada: esses endereços já são retornados nas estruturas sockaddr_in.
- Essa afirmação na especificação POSIX também implica que, se o flag AI_PASSIVE for especificado sem um hostname, o endereço curinga IPv6 (IN6ADDR_ANY_INIT ou 0::0) então deverá retornar como uma estrutura sockaddr_in6, juntamente com o endereço curinga IPv4 (INADDR_ANY ou 0.0.0.0), que retorna como uma estrutura sockaddr_in. Também faz sentido primeiro retornar o endereço curinga IPv6 porque, como veremos na Seção 12.2, um soquete de servidor IPv6 pode tratar tanto clientes IPv6 como IPv4 em um host de pilha dual.
- A família de endereços especificada no membro ai_family da estrutura *hints*, juntamente com flags como AI_V4MAPPED e AI_ALL especificados no membro ai_flags, determina os tipos de registros que são pesquisados no DNS (A e/ou AAAA) e os tipos de endereços que retornam (IPv4, IPv6 e/ou IPv6 mapeado de IPv4). Resumimos isso na Figura 11.8.
- O hostname também pode ser uma string hexadecimal do IPv6 ou uma string decimal com pontos do IPv4. A validade dessa string depende da família de endereços especificada pelo chamador. Uma string hexadecimal do IPv6 não é aceitável se AF_INET for especificado; e uma string decimal com pontos do IPv4 não é aceitável se AF_INET6 for especificado. Mas, se AF_UNSPEC for especificado, qualquer uma é aceitável e o tipo apropriado da estrutura de endereço de soquete é retornado.

  > Poderíamos argumentar que, se AF_INET6 for especificado, então uma string decimal com pontos deve ser retornada como um endereço IPv6 mapeado de IPv4 em uma estrutura sockaddr_in6. Mas uma outra maneira de obter esse resultado é prefixar a string decimal com pontos com 0::ffff:.

A Figura 11.8 resume como esperamos que getaddrinfo trate endereços IPv4 e IPv6. A coluna "Resultado" é o que queremos que retorne ao chamador, dadas as variáveis nas primeiras três colunas. A coluna "Ação" é como obtemos esse resultado.

Observe que a Figura 11.8 especifica somente a maneia como getaddrinfo trata o IPv4 e o IPv6; isto é, o número de endereços retornado ao chamador. O número real da estrutura addrinfo retornada ao chamador também depende do tipo de soquete especificado e do nome do serviço, como resumido anteriormente na Figura 11.6.

11.10 Função getaddrinfo: Exemplos

Agora, mostraremos alguns exemplos de getaddrinfo utilizando um programa de teste que permite inserir todos os parâmetros: hostname, nome de serviço, família de endereço, tipo de soquete e os flags AI_CANONNAME e AI_PASSIVE. (Não mostramos esse programa de teste, uma vez que contém aproximadamente 350 linhas de código desinteressantes. Ele é fornecido com o código-fonte para este livro, como descrito no Prefácio.) O programa de teste gera saída com informações sobre o número variável de estruturas addrinfo que retornam, mostrando os argumentos para uma chamada a socket e o endereço em cada estrutura de endereço de soquete.

Hostname especificado pelo chamador	Família de endereços especificada pelo chamador	Hostname string contém	Resultado	Ação
string de hostname não-nula; ativo ou passivo	AF_UNSPEC	hostname	Todos os registros AAAA retornados como sockaddr_in6{}s *e* todos os registros A retornados como sockaddr_in{}s	Pesquisa de registro AAAA *e* pesquisa de registro A
		string hexadecimal	Um sockaddr_in6{}	inet_pton(AF_INET6)
		notação decimal separada por pontos	Um sockaddr_in{}	inet_pton(AF_INET)
	AF_INET6	hostname	Todos os registros AAAA retornados como sockaddr_in6{}s	Pesquisa de registro AAAA
			Se ai_flags contiver AI_V4MAPPED, todos os registros AAAA retornados como sockaddr_in6{}s *caso contrário*, todos os registros A são retornados como IPv6 mapeados para de IPv4 sockaddr_in6{}s	Pesquisa de registro AAAA se nenhum resultado então pesquisa de registro A
			Se ai_flags contiver AI_V4MAPPED e AI_ALL, todos os registros AAAA retornados como sockaddr_in6{}s *e* todos os registros A retornados como IPv6 mapeados de IPv4 sockaddr_in6{}s	Pesquisa de registro AAAA *e* pesquisa de registro A
		string hexadecimal	Um sockaddr_in6{}	inet_pton(AF_INET6)
		notação decimal separada por pontos	Pesquisado como hostname	
	AF_INET	hostname	Todos os registros A retornados como sockaddr_in{}s	Pesquisa de registro A
		string hexadecimal	Pesquisado como hostname	
		notação decimal separada por pontos	Um sockaddr_in{}	inet_pton(AF_INET)
string de hostname nula; passivo	AF_UNSPEC	0::0 implícito 0.0.0.0 implícito	Um sockaddr_in6{} e um sockaddr_in{}	inet_pton(AF_INET6) inet_pton(AF_INET)
	AF_INET6	0::0 implícito	Um sockaddr_in6{}	inet_pton(AF_INET6)
	AF_INET	0.0.0.0 implícito	Um sockaddr_in{}	inet_pton(AF_INET)
string de hostname nula; ativo	AF_UNSPEC	0::1 implícito 127.0.0.1 implícito	Um sockaddr_in6{} e um sockaddr_in{}	inet_pton(AF_INET6) inet_pton(AF_INET)
	AF_INET6	0::1 implícito	Um sockaddr_in6{}	inet_pton(AF_INET6)
	AF_INET	127.0.0.1 implícito	Um sockaddr_in{}	inet_pton(AF_INET)

Figura 11.8 O resumo de `getaddrinfo`, suas ações e resultados.

Primeiro, mostramos o mesmo exemplo da Figura 11.5.

```
freebsd % testga -f inet -c -h freebsd4 -s domain

socket(AF_INET, SOCK_DGRAM, 17), ai_canonname = freebsd4.unpbook.com
        address: 135.197.17.100:53

socket(AF_INET, SOCK_DGRAM, 17)
        address: 172.24.37.94:53

socket(AF_INET, SOCK_STREAM, 6), ai_canonname = freebsd4.unpbook.com
        address: 135.197.17.100:53

socket(AF_INET, SOCK_STREAM, 6)
        address: 172.24.37.94:53
```

A opção `-f inet` especifica a família de endereços, `-c` instrui a retornar o nome canônico, `-h bsdi` especifica o hostname e `-s domain` especifica o nome de serviço.

O cenário do cliente comum é especificar a família de endereços, o tipo de soquete (a opção `-t`), o hostname e o nome de serviço. O exemplo a seguir mostra isso para um host multihomed com três endereços IPv4:

```
freebsd % testga -f inet -t stream -h gateway.tuc.noao.edu -s daytime

socket(AF_INET, SOCK_STREAM, 6)
        address: 140.252.108.1:13

socket(AF_INET, SOCK_STREAM, 6)
        address: 140.252.1.4:13

socket(AF_INET, SOCK_STREAM, 6)
        address: 140.252.104.1:13
```

Em seguida, especificamos nosso host `aix`, que tem tanto um registro AAAA como um registro A. Não especificamos a família de endereços, mas fornecemos um nome de serviço de `ftp`, fornecido apenas pelo TCP.

```
freebsd % testga -h aix -s ftp -t stream

socket(AF_INET6, SOCK_STREAM, 6)
        address: [3ffe:b80:1f8d:2:204:acff:fe17:bf38]:21

socket(AF_INET, SOCK_STREAM, 6)
        address: 192.168.42.2:21
```

Como não especificamos a família de endereços e como executamos esse exemplo em um host que suporta tanto o IPv4 como o IPv6, duas estruturas retornam: uma para o IPv4 e outra para o IPv6.

Em seguida, especificamos o flag `AI_PASSIVE` (a opção `-p`); não especificamos uma família de endereços nem um hostname (implicando o endereço curinga). Também especificamos um número de porta de 8888 e um soquete de fluxo.

```
freebsd % testga -p -s 8888 -t stream

socket(AF_INET6, SOCK_STREAM, 6)
        address: [::]:8888

socket(AF_INET, SOCK_STREAM, 6)
        address: 0.0.0.0:8888
```

Duas estruturas retornam. Como executamos isso em um host que suporta o IPv6 e o IPv4 sem especificar uma família de endereços, `getaddrinfo` retorna o endereço curinga IPv6 e o endereço curinga IPv4. A estrutura IPv6 retorna antes da estrutura IPv4, como veremos no Capítulo 12 que um cliente ou servidor IPv6 em um host de pilha dual pode se comunicar tanto com peers IPv6 ou IPv4.

11.11 Função `host_serv`

Nossa primeira interface a `getaddrinfo` não exige que o chamador aloque e preencha uma estrutura *hints*. Em vez disso, os dois campos de interesse, a família de endereços e o tipo de soquete, são argumentos para nossa função `host_serv`.

```
#include "unp.h"

struct addrinfo *host_serv(const char *hostname, const char *service,
                           int family, int socktype);
```

Retorna: ponteiro para a estrutura addrinfo se OK, `NULL` em erro

A Figura 11.9 mostra o código-fonte dessa função.

lib/host_serv.c

```
 1 #include   "unp.h"

 2 struct addrinfo *
 3 host_serv(const char *host, const char *serv, int family, int socktype)
 4 {
 5     int    n;
 6     struct addrinfo hints, *res;

 7     bzero(&hints, sizeof(struct addrinfo));
 8     hints.ai_flags = AI_CANONNAME;  /* sempre retorna nome canônico */
 9     hints.ai_family = family;   /* AF_UNSPEC, AF_INET, AF_INET6 etc. */
10     hints.ai_socktype = socktype;    /* 0, SOCK_STREAM, SOCK_DGRAM, etc. */

11     if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
12         return(NULL);

13     return(res); /* retorna ponteiro para o primeiro na lista vinculada */
14 }
```

lib/host_serv.c

Figura 11.9 Função `host_serv`.

7–13 A função inicializa uma estrutura *hints*, chama `getaddrinfo` e retorna um ponteiro nulo se ocorrer um erro.

Chamaremos essa função na Figura 16.17 ao utilizar `getaddrinfo` para obter as informações sobre o host e o serviço, porém nós mesmos queremos estabelecer a conexão.

11.12 Função `tcp_connect`

Agora, escreveremos duas funções que utilizam `getaddrinfo` para tratar a maioria dos cenários para os clientes e servidores TCP que escrevemos. A primeira função, `tcp_connect`, realiza os passos normais de cliente: cria um soquete de TCP e o conecta a um servidor.

```
#include "unp.h"

int tcp_connect(const char *hostname, const char *service);
```

Retorna: descritor de soquete conectado se OK, nenhum retorno em erro

A Figura 11.10 mostra o código-fonte.

lib/tcp_connect.c

```
 1 #include   "unp.h"

 2 int
 3 tcp_connect(const char *host, const char *serv)
 4 {
 5     int    sockfd, n;
 6     struct addrinfo hints, *res, *ressave;

 7     bzero(&hints, sizeof(struct addrinfo));
 8     hints.ai_family = AF_UNSPEC;
 9     hints.ai_socktype = SOCK_STREAM;

10     if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
11         err_quit("tcp_connect error for %s, %s: %s",
12                  host, serv, gai_strerror(n));
```

Figura 11.10 Função `tcp_connect`: executa os passos normais de cliente (*continua*).

```
13      ressave = res;

14      do {
15          sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
16          if (sockfd < 0)
17              continue;       /* ignora este */

18          if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
19              break;          /* sucesso */

20          Close(sockfd);      /* ignora este */
21      } while ( (res = res->ai_next) != NULL);

22      if (res == NULL)        /* errno configurado a partir de connect() final */
23          err_sys("tcp_connect error for %s, %s", host, serv);

24      freeaddrinfo(ressave);

25      return(sockfd);
26  }
```

lib/tcp_connect.c

Figura 11.10 Função `tcp_connect`: SegueExecuta os passos normais de cliente (*continuação*).

Chamada a `getaddrinfo`

7–13 `getaddrinfo` é chamado uma vez e especificamos a família de endereços como `AF_UNSPEC` e o tipo de soquete como `SOCK_STREAM`.

Tentar cada estrutura `addrinfo` até ser bem-sucedida ou final da lista

14–25 Cada endereço IP retornado é então tentado. `socket` e `connect` são chamadas. Não é um erro fatal `socket` falhar, como poderia acontecer se um endereço IPv6 fosse retornado mas o kernel do host não suportasse o IPv6. Se `connect` for bem-sucedida, um `break` encerra o loop. Caso contrário, quando todos os endereços foram tentados, o loop também termina. `freeaddrinfo` retorna toda a memória dinâmica.

Essa função (e nossas outras funções que fornecem uma interface para `getaddrinfo` mais simples nas seções a seguir) termina se `getaddrinfo` falhar ou nenhuma chamada a `connect` for bem-sucedida. O único retorno é no caso de sucesso. Seria difícil retornar um código de erro (uma das constantes `EAI_`*xxx*) sem adicionar um outro argumento. Isso significa que nossa função empacotadora é trivial.

```
int
Tcp_connect(const char *host, const char *serv)
{
   return(tcp_connect(host, serv));
}
```

Contudo, ainda chamamos nossa função empacotadora em vez de `tcp_connect`, para manter a consistência com o restante do texto.

> O problema com o valor de retorno é que os descritores são não-negativos, mas não sabemos se os valores `EAI_`*xxx* são positivos ou negativos. Se esses valores fossem positivos, poderíamos retornar seu negativo se `getaddrinfo` falhar, mas também temos de retornar algum outro valor negativo para indicar que todas as estruturas foram tentadas sem sucesso.

Exemplo: Cliente de data/hora

A Figura 11.11 mostra nosso cliente de data/hora da Figura 1.5 recodificado para utilizar `tcp_connect`.

names/daytimetcpcli.c

```
 1 #include   "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    sockfd, n;
 6     char   recvline[MAXLINE + 1];
 7     socklen_t len;
 8     struct sockaddr_storage ss;

 9     if (argc != 3)
10         err_quit
11             ("usage: daytimetcpcli <hostname/IPaddress> <service/port#>");

12     sockfd = Tcp_connect(argv[1], argv[2]);

13     len = sizeof(ss);
14     Getpeername(sockfd, (SA *) &ss, &len);
15     printf("connected to %s\n", Sock_ntop_host((SA *) &ss, len));

16     while ( (n = Read(sockfd, recvline, MAXLINE)) > 0) {
17         recvline[n] = 0;          /* termina com nulo */
18         Fputs(recvline, stdout);
19     }
20     exit(0);
21 }
```

names/daytimetcpcli.c

Figura 11.11 Cliente de data/hora recodificado para utilizar `tcp_connect`.

Argumentos de linha de comando

9–11 Agora, requeremos um segundo argumento de linha de comando para especificar o nome de serviço ou o número de porta, que permite a nosso programa conectar-se a outras portas.

Conexão a um servidor

12 Todo o código de soquete para esse cliente agora é realizado por `tcp_connect`.

Impressão do endereço do servidor

13–15 Chamamos `getpeername` para buscar e imprimir o endereço de protocolo do servidor. Fazemos isso para verificar o protocolo sendo utilizado nos exemplos que mostraremos.

Observe que `tcp_connect` não retorna o tamanho da estrutura de endereço de soquete que foi utilizado para `connect`. Poderíamos ter adicionado um argumento de ponteiro para retornar esse valor, mas um dos objetivos do *design* dessa função era reduzir o número de argumentos se comparado com `getaddrinfo`. O que fazemos, em vez disso, é utilizar uma estrutura de endereço de soquete `sockaddr_storage` suficientemente grande para manter e atender as restrições de alinhamento de qualquer tipo de endereço de soquete que o sistema suporta.

Essa versão do nosso cliente funciona com o IPv4 e com o IPv6, enquanto a versão na Figura 1.5 funcionou somente com o IPv4 e a versão na Figura 1.6 somente com o IPv6. Você também deve comparar nossa nova versão com a Figura E.12, que codificamos para utilizar `gethostbyname` e `getservbyname` para suportar tanto o IPv4 como o IPv6.

Primeiro, especificamos o nome de um host que suporte somente o IPv4.

```
freebsd % daytimetcpcli linux daytime
connected to 206.168.112.96
Sun Jul 27 23:06:24 2003
```

Em seguida, especificamos o nome de um host que suporte tanto o IPv4 como o IPv6.

```
freebsd % daytimetcpcli aix daytime
connected to 3ffe:b80:1f8d:2:204:acff:fe17:bf38
Sun Jul 27 23:17:13 2003
```

O endereço IPv6 é utilizado porque o host tem um registro AAAA e um registro A e, como observado na Figura 11.8, uma vez que `tcp_connect` configura a família de endereços como `AF_UNSPEC`, os registros AAAA são primeiramente pesquisados, e somente se essa pesquisa falhar ocorrerá uma pesquisa em um registro A.

No próximo exemplo, forçamos o uso do endereço IPv4 especificando o hostname com o nosso sufixo `-4` que, como observamos na Seção 11.2, é a nossa convenção para o hostname apenas com registros A.

```
freebsd % daytimetcpcli aix-4 daytime
connected to 192.168.42.2
Sun Jul 27 23:17:48 2003
```

11.13 Função `tcp_listen`

Nossa próxima função, `tcp_listen`, executa os passos normais do servidor TCP: cria um soquete de TCP, chama `bind` para vincular a uma porta bem-conhecida do servidor e permite que solicitações de conexões entrantes sejam aceitas. A Figura 11.12 mostra o código-fonte.

```
#include "unp.h"

int tcp_listen(const char *hostname, const char *service, socklen_t *addrlenp);
```

Retorna: descritor de soquete conectado se OK, nenhum retorno em erro

Chamada a `getaddrinfo`

8–15 Inicializamos uma estrutura `addrinfo` com nossas dicas: `AI_PASSIVE`, visto que essa função é para um servidor, `AF_UNSPEC` para a família de endereços e `SOCK_STREAM`. Lembre-se, da Figura 11.8, de que, se um hostname não for especificado (o que é comum para um servidor que quer vincular o endereço curinga), as dicas `AI_PASSIVE` e `AF_UNSPEC` farão com que duas estruturas de endereço de soquete retornem: a primeira para o IPv6 e a seguinte para o IPv4 (supondo um host de pilha dual).

Criação de um soquete e vínculo de um endereço

16–25 As funções `socket` e `bind` são chamadas. Se uma dessas chamadas falhar, simplesmente ignoramos essa estrutura `addrinfo` e prosseguimos com a próxima. Como afirmado na Seção 7.5, sempre configuramos a opção de soquete `SO_REUSEADDR` para um servidor TCP.

Verificação de falhas

26–27 Se todas as chamadas a `socket` e `bind` falharem, imprimiremos um erro e terminaremos. Como ocorre com a nossa função `tcp_connect` na seção anterior, não tentamos retornar um erro a partir dessa função.

28 O soquete transforma-se em um soquete ouvinte por `listen`.

Retorno do tamanho da estrutura de endereço de soquete

29–32 Se o argumento *addrlenp* for não-nulo, retornaremos o tamanho dos endereços de protocolo por meio desse ponteiro. Isso permite ao chamador alocar memória para uma estrutura de endereço de soquete a fim de obter o endereço de protocolo do cliente a partir de `accept`. (Veja também o Exercício 11.7.)

Exemplo: Servidor de data/hora

A Figura 11.13 mostra nosso servidor de data/hora da Figura 4.11, recodificado para utilizar `tcp_listen`.

Exigindo nome de serviço ou número de porta como o argumento de linha de comando

11–12 Exigimos um argumento de linha de comando para especificar o nome do serviço ou número da porta. Essa exigência torna mais fácil testar nosso servidor, uma vez que vincular a porta 13 ao servidor de data/hora exige privilégios de superusuário.

Criação do soquete ouvinte

13 `tcp_listen` cria o soquete ouvinte. Passamos um ponteiro NULL como o terceiro argumento porque o tamanho da estrutura de endereço que a família de endereços utiliza não é importante; utilizaremos `sockaddr_storage`.

lib/tcp_listen.c

```
 1 #include   "unp.h"

 2 int
 3 tcp_listen(const char *host, const char *serv, socklen_t *addrlenp)
 4 {
 5     int    listenfd, n;
 6     const int on = 1;
 7     struct addrinfo hints, *res, *ressave;

 8     bzero(&hints, sizeof(struct addrinfo));
 9     hints.ai_flags = AI_PASSIVE;
10     hints.ai_family = AF_UNSPEC;
11     hints.ai_socktype = SOCK_STREAM;

12     if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
13         err_quit("tcp_listen error for %s, %s: %s",
14                  host, serv, gai_strerror(n));
15     ressave = res;

16     do {
17         listenfd =
18             socket(res->ai_family, res->ai_socktype, res->ai_protocol);
19         if (listenfd < 0)
20             continue;             /* erro, tenta o próximo */

21         Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
22         if (bind(listenfd, res->ai_addr, res->ai_addrlen) == 0)
23             break;                /* sucesso */

24         Close(listenfd);  /* erro de vinculação, fecha e tenta o próximo */
25     } while ( (res = res->ai_next) != NULL);

26     if (res == NULL)    /* errno proveniente de socket() final ou bind() */
27         err_sys("tcp_listen error for %s, %s", host, serv);

28     Listen(listenfd, LISTENQ);

29     if (addrlenp)
30         *addrlenp = res->ai_addrlen;      /* retorna tamanho de endereço do
                                                * protocolo */

31     freeaddrinfo(ressave);

32     return(listenfd);
33 }
```

lib/tcp_listen.c

Figura 11.12 Função `tcp_listen`: segue e executa os passos normais de servidor.

Loop do servidor

14–22 `accept` espera cada conexão de cliente. Imprimimos o endereço de cliente chamando `sock_ntop`. No caso do IPv4 ou do IPv6, essa função imprime o endereço IP e o número da porta. Poderíamos utilizar a função `getnameinfo` (Seção 11.17) para tentar obter o hostname do cliente, mas isso envolve uma consulta de PTR no DNS, que pode levar algum tempo, especialmente se a consulta de PTR falhar. A Seção 14.8 do TCPv3 indica que, em um servidor Web ocupado, quase 25% de todos os clientes que se conectam a esse servidor não tinham registros PTR no DNS. Como não queremos que um servidor (especialmente um servidor iterativo) espere segundos por uma consulta PTR, simplesmente imprimimos o endereço e a porta IP.

names/daytimetcpsrv1.c

```
 1 #include   "unp.h"
 2 #include   <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     listenfd, connfd;
 7     socklen_t len;
 8     char    buff[MAXLINE];
 9     time_t ticks;
10     struct sockaddr_storage cliaddr;

11     if (argc != 2)
12         err_quit("usage: daytimetcpsrv1 <service or port#>");

13     listenfd = Tcp_listen(NULL, argv[1], NULL);

14     for ( ; ; ) {
15         len = sizeof(cliaddr);
16         connfd = Accept(listenfd, (SA *) &cliaddr, &len);
17         printf("connection from %s\n", Sock_ntop((SA *) &cliaddr, len));

18         ticks = time(NULL);
19         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
20         Write(connfd, buff, strlen(buff));

21         Close(connfd);
22     }
23 }
```

names/daytimetcpsrv1.c

Figura 11.13 Mostra nosso servidor de data/hora recodificado para utilizar `tcp_listen` (veja também a Figura 11.14).

Exemplo: Servidor de data/hora com especificação de protocolo

Há um pequeno problema com a Figura 11.13: o primeiro argumento para `tcp_listen` é um ponteiro nulo, que, combinado com a família de endereços de `AF_UNSPEC` que `tcp_listen` especifica, poderia fazer com que `getaddrinfo` retornasse uma estrutura de endereço de soquete com uma família de endereços diferente da desejada. Por exemplo, a primeira estrutura do endereço de soquete retornado será para o IPv6 em um host de pilha dual (Figura 11.8), mas poderíamos querer que nosso servidor tratasse somente o IPv4.

Os clientes não têm esse problema visto que o cliente sempre deve especificar um endereço IP ou um hostname. Aplicações clientes normalmente permitem ao usuário inserir isso como um argumento de linha de comando. Isso oferece a oportunidade de especificar um hostname associado a um tipo de endereço IP particular (lembre-se dos nossos hostnames `-4` e `-6` na Seção 11.2) ou de especificar uma string decimal com pontos do IPv4 (que force o IPv4) ou uma string hexadecimal do IPv6 (que force o IPv6).

Mas há uma técnica simples para servidores que permite forçar um dado protocolo em um servidor, seja IPv4 ou IPv6: permitir ao usuário inserir um endereço IP ou um hostname como um argumento de linha de comando para o programa e passar isso para `getaddrinfo`. No caso de um endereço IP, uma string de pontos decimais com pontos do IPv4 difere de uma string hexadecimal do IPv6. As chamadas a `inet_pton` a seguir falham ou são bem-sucedidas, como indicado:

```
inet_pton(AF_INET,   "0.0.0.0",  &foo);      /* tem sucesso */
inet_pton(AF_INET,   "0::0",     &foo);      /* falha */
inet_pton(AF_INET6,  "0.0.0.0",  &foo);      /* falha */
inet_pton(AF_INET6,  "0::0",     &foo);      /* tem sucesso */
```

Portanto, se alterarmos nossos servidores para aceitar um argumento opcional e se inserirmos

```
% server
```

ele assume o padrão do IPv6 em um host de pilha dual, mas inserir

```
% server 0.0.0.0
```

explicitamente especifica o IPv4 e

```
% server 0::0
```

explicitamente especifica o IPv6.

A Figura 11.14 mostra essa versão final do nosso servidor de data/hora.

Tratando argumentos de linha de comando

11–16 A única alteração a partir da Figura 11.13 é o tratamento dos argumentos de linha de comando, permitindo ao usuário especificar um hostname ou um endereço IP que o servidor vincula, além de um nome de serviço ou porta.

Primeiro, iniciamos esse servidor com um soquete IPv4 e então nos conectamos ao servidor a partir de clientes em dois outros hosts na sub-rede local.

```
freebsd % daytimetcpsrv2 0.0.0.0 9999
connection from 192.168.42.2:32961
connection from 192.168.42.3:1389
```

Agora iniciamos o servidor com um soquete IPv6.

```
freebsd % daytimetcpsrv2 0::0 9999
connection from [3ffe:b80:1f8d:2:204:acff:fe17:bf38]:32964
connection from [3ffe:b80:1f8d:2:230:65ff:fe15:caa7]:49601
connection from [::ffff:192.168.42.2]:32967
connection from [::ffff:192.168.42.3]:49602
```

A primeira conexão é a partir do host `aix` utilizando o IPv6 e a segunda é a partir do host `macosx` utilizando também o IPv6. As duas próximas conexões são a partir dos hosts `aix` e `macosx`, mas utilizando o IPv4, não o IPv6. Podemos afirmar isso porque os endereços do cliente retornados por `accept` são endereços IPv6 mapeados de IPv4.

O que acabamos de mostrar é que um servidor IPv6 em execução em um host de pilha dual pode tratar clientes IPv4 ou IPv6. Os endereços do cliente IPv4 são passados para o servidor IPv6 como endereços IPv6 mapeados de IPv4, como discutiremos na Seção 12.2.

names/daytimetcpsrv2.c

```
1 #include   "unp.h"
2 #include   <time.h>
```

Figura 11.14 Servidor de data/hora independente de protocolo que utiliza `tcp_listen` (*continua*).

```
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int    listenfd, connfd;
 7     socklen_t len;
 8     char   buff[MAXLINE];
 9     time_t ticks;
10     struct sockaddr_storage cliaddr;

11     if (argc == 2)
12         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
13     else if (argc == 3)
14         listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
15     else
16         err_quit("usage: daytimetcpsrv2 [ <host> ] <service or port>");

17     for ( ; ; ) {
18         len = sizeof(cliaddr);
19         connfd = Accept(listenfd, (SA *) &cliaddr, &len);
20         printf("connection from %s\n", Sock_ntop((SA *) &cliaddr, len));

21         ticks = time(NULL);
22         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
23         Write(connfd, buff, strlen(buff));

24         Close(connfd);
25     }
26 }
```

names/daytimetcpsrv2.c

Figura 11.14 Servidor de data/hora independente de protocolo que utiliza `tcp_listen` (*continuação*).

11.14 Função `udp_client`

Nossas funções que fornecem uma interface mais simples a `getaddrinfo` mudam com o UDP porque fornecemos uma função de cliente que cria um soquete UDP não-conectado e uma outra na seção seguinte que cria um soquete UDP conectado.

```
#include "unp.h"

int udp_client(const char *hostname, const char *service,
               struct sockaddr **saptr, socklen_t *lenp);
```

Retorna: descritor de soquete não-conectado se OK, nenhum retorno em erro

Essa função cria um soquete não-conectado de UDP, retornando três itens. Primeiro, o valor de retorno é o descritor de soquete. Segundo, *saptr* é o endereço de um ponteiro (declarado pelo chamador) para uma estrutura de endereço de soquete (alocada dinamicamente por `udp_client`) e, nessa estrutura, a função armazena o endereço IP e a porta de destino para chamadas futuras a `sendto`. O tamanho da estrutura de endereço de soquete é retornada na variável apontada por *lenp*. Esse argumento final não pode ser um ponteiro nulo (uma vez que permitimos o argumento final para `tcp_listen`) porque o comprimento da estrutura de endereço de soquete é exigido em quaisquer chamadas a `sendto` e `recvfrom`.

A Figura 11.15 mostra o código-fonte dessa função.

lib/udp_client.c

```
 1 #include   "unp.h"
 2 int
 3 udp_client(const char *host, const char *serv, SA **saptr, socklen_t *lenp)
 4 {
 5     int    sockfd, n;
 6     struct addrinfo hints, *res, *ressave;
 7     bzero(&hints, sizeof(struct addrinfo));
 8     hints.ai_family = AF_UNSPEC;
 9     hints.ai_socktype = SOCK_DGRAM;
10     if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
11         err_quit("udp_client error for %s, %s: %s",
12                  host, serv, gai_strerror(n));
13     ressave = res;
14     do {
15         sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
16         if (sockfd >= 0)
17             break;                /* sucesso */
18     } while ( (res = res->ai_next) != NULL);
19     if (res == NULL) /* errno configurado a partir de socket() final */
20         err_sys("udp_client error for %s, %s", host, serv);
21     *saptr = Malloc(res->ai_addrlen);
22     memcpy(*saptr, res->ai_addr, res->ai_addrlen);
23     *lenp = res->ai_addrlen;
24     freeaddrinfo(ressave);
25     return(sockfd);
26 }
```

lib/udp_client.c

Figura 11.15 Função `udp_client`: cria um soquete UDP não-conectado.

`getaddrinfo` converte os argumentos *hostname* e *service*. Um soquete de datagrama é criado. A memória é alocada para uma das estruturas de endereço de soquete e a estrutura de endereço de soquete correspondente ao soquete que foi criado é copiada para a memória.

Exemplo: Cliente de data/hora de protocolo independente

Agora recodificamos nosso cliente de data/hora da Figura 11.11 para utilizar UDP e nossa função `udp_client`. A Figura 11.16 mostra o código-fonte independente de protocolo.

names/daytimeudpcli1.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    sockfd, n;
 6     char   recvline[MAXLINE + 1];
 7     socklen_t salen;
 8     struct sockaddr *sa;
 9     if (argc != 3)
10         err_quit
11             ("usage: daytimeudpcli1 <hostname/IPaddress> <service/port#>");
```

Figura 11.16 O cliente UDP de data/hora utilizando nossa função `udp_client` (*continua*).

```
12      sockfd = Udp_client(argv[1], argv[2], (void **) &sa, &salen);
13      printf("sending to %s\n", Sock_ntop_host(sa, salen));
14      Sendto(sockfd, "", 1, 0, sa, salen); /* envia um datagrama de 1 byte */
15      n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
16      recvline[n] = '\0';          /* termina com nulo */
17      Fputs(recvline, stdout);
18      exit(0);
19  }
```

names/daytimeudpcli1.c

Figura 11.16 O cliente UDP de data/hora utilizando nossa função `udp_client` (*continuação*).

12–17 Chamamos nossa função `udp_client` e então imprimimos o endereço IP e a porta do servidor ao qual enviaremos o datagrama de UDP. Enviamos um datagrama de um byte e então lemos e imprimimos a resposta.

> Precisamos enviar somente um datagrama UDP de zero byte, uma vez que o que desencadeia a resposta do servidor de data/hora é simplesmente a chegada de um datagrama, independentemente do comprimento e do conteúdo. Mas muitas implementações SVR4 não permitem um datagrama UDP de comprimento zero.

Executamos nosso cliente especificando um hostname que tem um registro AAAA e um registro A. Uma vez que a estrutura com o registro AAAA é retornada primeiramente por `getaddrinfo`, um soquete de IPv6 é criado.

```
freebsd % daytimeudpcli1 aix daytime
sending to 3ffe:b80:1f8d:2:204:acff:fe17:bf38
Sun Jul 27 23:21:12 2003
```

Em seguida, especificamos o endereço decimal com pontos do mesmo host, resultando em um soquete IPv4.

```
freebsd % daytimeudpcli1 192.168.42.2 daytime
sending to 192.168.42.2
Sun Jul 27 23:21:40 2003
```

11.15 Função `udp_connect`

Nossa função `udp_connect` cria um soquete UDP conectado.

```
#include "unp.h"

int udp_connect(const char *hostname, const char *service);
```

Retorna: descritor de soquete conectado se OK, nenhum retorno em erro

Com um soquete UDP conectado, os dois argumentos finais exigidos por `udp_client` não são mais necessários. O chamador pode chamar `write` em vez de `sendto`, assim nossa função não precisa retornar uma estrutura de endereço de soquete e seu comprimento.

A Figura 11.17 mostra o código-fonte.

lib/udp_connect.c

```
 1 #include   "unp.h"
 2 int
 3 udp_connect(const char *host, const char *serv)
 4 {
 5     int    sockfd, n;
 6     struct addrinfo hints, *res, *ressave;
 7     bzero(&hints, sizeof(struct addrinfo));
 8     hints.ai_family = AF_UNSPEC;
 9     hints.ai_socktype = SOCK_DGRAM;
10     if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
11         err_quit("udp_connect error for %s, %s: %s",
12                  host, serv, gai_strerror(n));
13     ressave = res;
14     do {
15         sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
16         if (sockfd < 0)
17             continue;             /* ignora este */
18         if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
19             break;                /* sucesso */
20         Close(sockfd);            /* ignora este */
21     } while ( (res = res->ai_next) != NULL);
22     if (res == NULL)    /* errno configurado a partir de connect() final */
23         err_sys("udp_connect error for %s, %s", host, serv);
24     freeaddrinfo(ressave);
25     return(sockfd);
26 }
```

lib/udp_connect.c

Figura 11.17 Função `udp_connect`: cria um soquete UDP conectado.

Essa função é quase idêntica a `tcp_connect`. Uma das diferenças, porém, é que a chamada a `connect` com um soquete UDP não envia nada ao peer. Se algo estiver errado (o peer está inacessível ou não há nenhum servidor na porta especificada), o chamador não descobrirá isso até enviar um datagrama ao peer.

11.16 Função `udp_server`

Nossa função UDP final que fornece uma interface mais simples a `getaddrinfo` é `udp_server`.

```
#include "unp.h"

int udp_server(const char *hostname, const char *service, socklen_t *lenptr);

                    Retorna: descritor de soquete não-conectado se OK, nenhum retorno em erro
```

Os argumentos são os mesmos de `tcp_listen`: um *hostname* opcional, um *service* requerido (de modo que o número da porta possa ser vinculado) e um ponteiro opcional para uma variável em que o tamanho da estrutura de endereço de soquete retorna.

A Figura 11.18 mostra o código-fonte.

lib/udp_server.c

```
 1 #include   "unp.h"
 2 int
 3 udp_server(const char *host, const char *serv, socklen_t *addrlenp)
 4 {
 5     int     sockfd, n;
 6     struct addrinfo hints, *res, *ressave;
 7     bzero(&hints, sizeof(struct addrinfo));
 8     hints.ai_flags = AI_PASSIVE;
 9     hints.ai_family = AF_UNSPEC;
10     hints.ai_socktype = SOCK_DGRAM;
11     if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
12         err_quit("udp_server error for %s, %s: %s",
13                  host, serv, gai_strerror(n));
14     ressave = res;
15     do {
16         sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
17         if (sockfd < 0)
18             continue;         /* error - tenta o próximo */
19         if (bind(sockfd, res->ai_addr, res->ai_addrlen) == 0)
20             break;            /* sucesso */
21         Close(sockfd);   /* erro de vinculação - fecha e tenta o próximo */
22     } while ( (res = res->ai_next) != NULL);
23     if (res == NULL)    /* errno proveniente de socket() final ou bind() */
24         err_sys("udp_server error for %s, %s", host, serv);
25     if (addrlenp)
26         *addrlenp = res->ai_addrlen;      /* retorna tamanho de endereço do
                                                * protocolo */
27     freeaddrinfo(ressave);
28     return(sockfd);
29 }
```

lib/udp_server.c

Figura 11.18 Função `udp_server`: cria um soquete não-conectado para um servidor UDP.

Essa função é quase idêntica a `tcp_listen`, mas sem a chamada a `listen`. Configuramos a família de endereços como `AF_UNSPEC`, mas o chamador pode utilizar a mesma técnica que descrevemos na Figura 11.14 para forçar um protocolo particular (IPv4 ou IPv6).

Não configuramos a opção de soquete `SO_REUSEADDR` para o soquete de UDP porque ela pode permitir que múltiplos soquetes vinculem-se à mesma porta UDP em hosts que suportam multicasting, como descrevemos na Seção 7.5. Como não há nada parecido com o estado TIME_WAIT do TCP para um soquete de UDP, não há necessidade de configurar essa opção de soquete quando o servidor é iniciado.

Exemplo: Servidor de data/hora independente de protocolo

A Figura 11.19 mostra nosso servidor de data/hora, modificado a partir da Figura 11.14 para utilizar o UDP.

names/daytimeudpsrv2.c

```
 1 #include   "unp.h"
 2 #include   <time.h>
 3 int
```

Figura 11.19 Servidor UDP de data/hora independente de protocolo (*continua*).

```
 4 main(int argc, char **argv)
 5 {
 6     int    sockfd;
 7     ssize_t n;
 8     char   buff[MAXLINE];
 9     time_t ticks;
10     socklen_t len;
11     struct sockaddr_storage cliaddr;

12     if (argc == 2)
13         sockfd = Udp_server(NULL, argv[1], NULL);
14     else if (argc == 3)
15         sockfd = Udp_server(argv[1], argv[2], NULL);
16     else
17         err_quit("usage: daytimeudpsrv [ <host> ] <service or port>");

18     for ( ; ; ) {
19         len = sizeof(cliaddr);
20         n = Recvfrom(sockfd, buff, MAXLINE, 0, (SA *) &cliaddr, &len);
21         printf("datagram from %s\n", Sock_ntop((SA *) &cliaddr, len));

22         ticks = time(NULL);
23         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
24         Sendto(sockfd, buff, strlen(buff), 0, (SA *) &cliaddr, len);
25     }
26 }
```

names/daytimeudpsrv2.c

Figura 11.19 Servidor UDP de data/hora independente de protocolo (*continuação*).

11.17 Função `getnameinfo`

Essa função é o complemento de `getaddrinfo`: ela recebe um endereço de soquete e retorna uma string de caracteres que descreve o host e uma outra string de caracteres que descreve o serviço. Essa função fornece tais informações de uma maneira independente de protocolo; isto é, o chamador não se preocupa com o tipo de endereço do protocolo que está contido na estrutura de endereço de soquete, uma vez que esse detalhe é tratado pela função.

```
#include <netdb.h>

int getnameinfo(const struct sockaddr *sockaddr, socklen_t addrlen,
                char *host, socklen_t hostlen,
                char *serv, socklen_t servlen, int flags);
```

Retorna: 0 se OK, não-zero em erro (veja a Figura 11.7)

sockaddr aponta para a estrutura de endereço de soquete que contém o endereço de protocolo a ser convertido em uma string legível por humanos e *addrlen* é o comprimento dessa estrutura. Essa estrutura e seu comprimento normalmente são retornados por `accept`, `recvfrom`, `getsockname` ou `getpeername`.

O chamador aloca espaço para as duas strings legíveis por humanos: *host* e *hostlen* especificam a string do host e *serv* e *servlen* especificam a string do serviço. Se o chamador não quiser que a string do host retorne, um *hostlen* de 0 é especificado. De maneira semelhante, um *servlen* de 0 não especifica a não retornar as informações sobre o serviço.

A diferença entre `sock_ntop` e `getnameinfo` é que a primeira não envolve o DNS e apenas retorna uma versão imprimível do endereço IP e número da porta. Normalmente, a última tenta obter um nome tanto para o host como para o serviço.

A Figura 11.20 mostra os seis *flags* que podem ser especificados para alterar a operação de `getnameinfo`.

`NI_DGRAM` deve ser especificado quando o chamador sabe que está lidando com um soquete de datagrama. A razão é que dados apenas o endereço IP e o número da porta na estrutura de endereço de soquete, `getnameinfo` não pode determinar o protocolo (TCP ou UDP). Há alguns números de porta que são utilizados para um serviço com o TCP e um serviço completamente diferente com o UDP. Um exemplo é a porta 514, que é o serviço `rsh` com TCP, mas o serviço `syslog` com UDP.

`NI_NAMEREQD` faz com que um erro retorne se o hostname não puder ser resolvido utilizando o DNS. Isso pode ser utilizado pelos servidores que exigem que o endereço IP do cliente seja mapeado para um hostname. Assim, esses servidores aceitam esse hostname retornado, chamam `getaddrinfo` e então verificam se um dos endereços retornados é o endereço na estrutura de endereço de soquete.

`NI_NOFQDN` faz com que o hostname retornado seja truncado no primeiro ponto. Por exemplo, se o endereço IP na estrutura de endereço de soquete fosse 192.168.42.2, `gethostbyaddr` retornaria um nome de `aix.unpbook.com`. Mas, se esse flag fosse especificado para `getnameinfo`, ele retornaria o hostname apenas como `aix`.

`NI_NUMERICHOST` informa `getnameinfo` a não chamar o DNS (que pode levar tempo). Em vez disso, a representação numérica do endereço IP é retornada como uma string, provavelmente chamando `inet_ntop`. De maneira semelhante, `NI_NUMERICSERV` especifica que o número decimal da porta deve ser retornado como uma string em vez de pesquisar o nome de serviço; e `NI_NUMERICSCOPE` especifica que a forma numérica do identificador de escopo deve ser retornada em vez do seu nome. Normalmente, os servidores devem especificar `NI_NUMERICSERV` porque os números de porta do cliente em geral não têm nenhum nome de serviço associado – eles são portas efêmeras.

O OU lógico de múltiplos flags pode ser especificado se estes fizerem sentido juntos (por exemplo, `NI_DGRAM` e `NI_NUMERICHOST`).

Constante	Descrição
`NI_DGRAM`	Serviço de datagrama
`NI_NAMEREQD`	Retorna um erro se o nome não puder ser resolvido no endereço
`NI_NOFQDN`	Retorna somente parte do hostname de FQDN
`NI_NUMERICHOST`	Retorna uma string numérica ao hostname
`NI_NUMERICSCOPE`	Retorna uma string numérica ao identificador de escopo
`NI_NUMERICSERV`	Retorna uma string numérica ao nome do serviço

Figura 11.20 *flags* para `getnameinfo`.

11.18 Funções reentrantes

A função `gethostbyname` na Seção 11.3 apresenta um problema interessante que nós ainda não examinamos no texto: ela não é *reentrante*. Encontraremos esse problema em geral ao lidar com threads no Capítulo 26, mas é interessante examiná-lo agora (sem ter de lidar com o conceito de threads) e ver como corrigi-lo.

Primeiro, vamos examinar como a função trabalha. Se examinarmos seu código-fonte (o que é fácil, uma vez que para toda a distribuição do BIND ele está publicamente disponível), veremos que um dos arquivos contém tanto `gethostbyname` como `gethostbyaddr` e o arquivo tem o seguinte esboço geral:

```
static struct hostent host;        /* resultado armazenado aqui */

struct hostent *
gethostbyname(const char *hostname)
{
   return(gethostbyname2(hostname, family));
}

struct hostent *
gethostbyname2(const char *hostname, int family)
{
   /* chama funções de DNS para consulta de A ou AAAA */

   /* preenche a estrutura do host */

   return(&host);
}

struct hostent *
gethostbyaddr(const char *addr, socklen_t len, int family)
{
   /* chama funções de DNS para consulta de PTR no domínio in-addr.arpa */

   /* preenche a estrutura do host */

   return(&host);

}
```

Destacamos o especificador da classe de armazenamento `static` da estrutura resultante porque esse é o problema básico. O fato de que essas três funções compartilham uma única variável `host` apresenta ainda um outro problema que discutiremos no Exercício 11.1. (`gethostbyname2` foi introduzido com o suporte IPv6 no BIND 4.9.4. Desde então, tornou-se obsoleto; consulte a Seção 11.20 para mais detalhes. Ignoraremos o fato de que `gethostbyname2` está envolvido quando chamamos `gethostbyname`, visto que isso não afeta essa discussão.)

O problema de reentrância pode ocorrer em um processo normal do Unix que chama `gethostbyname` ou `gethostbyaddr` tanto a partir do fluxo principal de controle como a partir de um handler de sinal. Quando o handler de sinal é chamado (digamos que seja um sinal `SIGALRM` que é gerado uma vez por segundo), o fluxo principal de controle do processo é temporariamente interrompido e o sinal que trata da função é chamado. Considere o seguinte:

```
main()
{
   struct hostent *hptr;

   ...
   signal(SIGALRM, sig_alrm);

   ...
   hptr = gethostbyname( ... );
   ...
}

void
sig_alrm(int signo)
{
   struct hostent *hptr;

   ...
   hptr = gethostbyname( ... );
   ...
}
```

Se o fluxo principal de controle estiver no meio de `gethostbyname` quando é temporariamente interrompido (digamos que a função preencheu a variável `host` e está em vias de

retornar) e se o handler de sinal então chamar `gethostbyname`, como há somente uma cópia da variável `host` no processo, ela é reutilizada. Isso sobrescreve os valores que foram calculados para a chamada a partir do fluxo principal de controle com os valores calculados para a chamada a partir do handler de sinal.

Se examinarmos as funções de conversão de nomes e endereços apresentadas neste capítulo, juntamente com as funções `inet_`*XXX* do Capítulo 4, observaremos o seguinte:

- Historicamente, `gethostbyname`, `gethostbyaddr`, `getservbyname` e `getservbyport` não são reentrantes porque todas retornam um ponteiro para uma estrutura estática. Algumas implementações que suportam threads (Solaris 2.x) fornecem versões reentrantes dessas quatro funções com nomes que terminam com o sufixo `_r`, que descreveremos na próxima seção.

  Alternativamente, algumas implementações que suportam threads (HP-UX 10.30 e superior) fornecem versões reentrantes dessas funções que utilizam dados específicos do thread (Seção 26.5).
- `inet_pton` e `inet_ntop` são sempre reentrantes.
- Historicamente, `inet_ntoa` não é reentrante, mas algumas implementações que suportam threads fornecem uma versão reentrante que utiliza dados específicos do thread.
- `getaddrinfo` é reentrante somente se ela mesmo chamar funções reentrantes; isto é, se chamar versões reentrantes de `gethostbyname` para o hostname e `getservbyname` para o nome de serviço. Uma das razões pelas quais toda a memória para os resultados é alocada dinamicamente é permitir que ela seja reentrante.
- `getnameinfo` é reentrante somente se ela mesma chamar funções reentrantes; isto é, se chamar versões reentrantes de `gethostbyaddr` para obter o hostname e `getservbyport` para obter o nome de serviço. Observe que as duas strings resultantes (para o hostname e para o nome de serviço) são alocadas pelo chamador para permitir essa reentrância.

Um problema semelhante ocorre com a variável `errno`. Historicamente, há uma única cópia dessa variável do tipo inteiro por processo. Se um processo fizer uma chamada de sistema que retorne um erro, um código de erro do tipo inteiro é armazenado nessa variável. Por exemplo, quando a função identificada como `close` na biblioteca C-padrão é chamada, ela poderia executar algo como o pseudocódigo a seguir:

- Coloque o argumento para a chamada de sistema (um descritor do tipo inteiro) em um registrador
- Coloque um valor em um outro registrador indicando que a chamada de sistema `close` está sendo feita
- Invoque a chamada de sistema (comute para o kernel com uma instrução especial)
- Teste o valor de um registrador para ver se ocorreu um erro
- Se não ocorreu erro, `return(0)`
- Armazene o valor de algum outro registrador em `errno`
- `return(-1)`

Primeiro, observe que, se um erro não ocorrer, o valor de `errno` não é alterado. Essa é a razão pela qual não podemos examinar o valor de `errno` a menos que saibamos que ocorreu um erro (normalmente indicado pela função que retorna –1).

Suponha que um programa teste o valor de retorno da função `close` e então imprima o valor de `errno` se um erro tiver ocorrido, como no seguinte exemplo:

```
if (close(fd) < 0) {
    fprintf(stderr, "close error, errno = %d\n", errno)
    exit(1);
}
```

Há um pequeno espaço de tempo entre o armazenamento do código de erro em `errno`, quando a chamada de sistema retorna, e a impressão desse valor pelo programa, durante o qual um outro thread de execução dentro desse processo (isto é, um handler de sinal) pode alterar o valor de `errno`. Se, por exemplo, quando o handler de sinal é chamado, o fluxo principal de controle estiver entre `close` e `fprintf` e o handler de sinal chamar alguma outra chamada de sistema que retorna um erro (digamos, `write`), então o valor de `errno` armazenado a partir da chamada de sistema `write` sobrescreverá o valor armazenado pela chamada de sistema `close`.

Ao examinar esses dois problemas com relação aos handlers de sinal, uma solução ao problema com `gethostbyname` (que retorna um ponteiro para uma variável estática) é *não* chamar funções não-reentrantes a partir de um handler de sinal. O problema com `errno` (uma única variável global que pode ser alterada pelo handler de sinal) pode ser evitado codificando o handler de sinal para salvar e restaurar o valor de `errno` nele como a seguir:

```
void
sig_alrm(int signo)
{
   int errno_save;

   errno_save = errno;       /* salve seu valor na entrada */
   if (write( ... ) != nbytes)
       fprintf(stderr, "write error, errno = %d\n", errno);
   errno = errno_save;       /* restaura seu valor no retorno */
}
```

Nesse exemplo de código, também chamamos `fprintf`, uma função de E/S-padrão, a partir do handler de sinal. Esse ainda é um outro problema de reentrância porque muitas versões da biblioteca E/S-padrão são não-reentrantes: funções de E/S-padrão não devem ser chamadas a partir de handlers de sinal.

Revisitaremos esse problema de reentrância no Capítulo 26 e veremos como threads tratam o problema da variável `errno`. A seção a seguir descreve algumas versões reentrantes das funções de hostname.

11.19 Funções `gethostbyname_r` e `gethostbyaddr_r`

Há duas maneiras de tornar uma função não-reentrante, como, por exemplo, `gethostbyname` reentrante.

1. Em vez de preencher e retornar uma estrutura estática, o chamador aloca a estrutura e a função reentrante preenche a estrutura do chamador. Essa é a técnica utilizada para ir da `gethostbyname` não-reentrante à `gethostbyname_r` reentrante. Mas essa solução fica mais complicada porque o chamador não apenas deve fornecer a estrutura `hostent` a ser preenchida, mas essa estrutura também aponta para outras informações: o nome canônico, o array de ponteiros de alias, as strings de alias, o array de ponteiros de endereço e os endereços (por exemplo, a Figura 11.2). O chamador deve fornecer um grande buffer que é utilizado para essas informações adicionais e a estrutura `hostent` que é preenchida contém então vários ponteiros nesse outro buffer. Isso adiciona pelo menos três argumentos à função: um ponteiro para a estrutura `hostent` preencher, um ponteiro para o buffer a ser utilizado para todas as outras informações e o tamanho desse buffer. Um quarto argumento adicional também é exigido: um ponteiro para um inteiro em que um código de erro pode ser armazena-

do, uma vez que o inteiro global `h_errno` não mais pode ser utilizado. (O inteiro global `h_errno` apresenta o mesmo problema de reentrância que descrevemos com `errno`.)

Essa técnica também é utilizada por `getnameinfo` e `inet_ntop`.

2. A função reentrante chama `malloc` e aloca dinamicamente a memória. Essa é a técnica utilizada por `getaddrinfo`. O problema com essa abordagem é que a aplicação que chama essa função também deve chamar `freeaddrinfo` para liberar a memória dinâmica. Se essa função livre não for chamada, ocorre *um vazamento de memória*: toda vez que o processo chama a função que aloca a memória, a utilização de memória por ele aumenta. Se o processo for executado por muito tempo (uma característica comum dos servidores de rede), o uso de memória cresce cada vez mais com o passar do tempo.

Discutiremos agora as funções reentrantes do Solaris 2.x para conversão entre nome e endereço e entre endereço e nome.

```
#include <netdb.h>

struct hostent *gethostbyname_r(const char *hostname,
                                struct hostent *result,
                                char *buf, int buflen, int *h_errnop);

struct hostent *gethostbyaddr_r(const char *addr, int len, int type,
                                struct hostent *result,
                                char *buf, int buflen, int *h_errnop);
```

As duas retornam: ponteiro não-nulo se OK, NULL em erro

Quatro argumentos adicionais são exigidos para cada função. *result* é uma estrutura `hostent` alocada pelo chamador. Ela é preenchida pela função. Se bem-sucedido, esse ponteiro também é o valor de retorno da função.

buf é um buffer alocado pelo chamador e *buflen* é seu tamanho. Esse buffer conterá o hostname canônico, os ponteiros e as strings de alias, os ponteiros de endereço e os endereços reais. Todos os ponteiros na estrutura apontados por *result* apontam para esse buffer. Qual o tamanho que esse buffer dever ter? Infelizmente, tudo o que a maioria das páginas man informa é algo vago como "o buffer deve ser suficientemente grande para armazenar todos os dados associados com a entrada de host". Implementações atuais de `gethostbyname` podem retornar até 35 ponteiros de alias e 35 ponteiros de endereço e internamente utilizam um buffer de 8192 bytes para armazenar nomes de alias e endereços. Portanto, um tamanho de buffer de 8192 bytes deve ser adequado.

Se ocorrer um erro, o código de erro é retornado pelo ponteiro *h_errnop*, não pelo global `h_errno`.

> Infelizmente, esse problema de reentrância é ainda pior do que parece. Primeiro, não há nenhum padrão com relação à reentrância e a `gethostbyname` e `gethostbyaddr`. A especificação POSIX diz que `gethostbyname` e `gethostbyaddr` não precisam ser reentrantes. O Unix 98 informa apenas que essas duas funções não precisam ser seguras a threads.
>
> Segundo, não há nenhum padrão para as funções `_r`. O que mostramos nessa seção (para propósitos de exemplo) são duas funções `_r` fornecidas pelo Solaris 2.x. O Linux fornece funções `_r` semelhantes, exceto que, em vez de retornar o `hostent` como o valor de retorno da função, o `hostent` é retornado utilizando um parâmetro valor-resultado co-

mo o penúltimo argumento da função. Ele retorna o sucesso da pesquisa como o valor de retorno da função bem como no argumento `h_errno`. O Digital Unix 4.0 e o HP-UX 10.30 têm versões dessas funções com argumentos diferentes. Os dois primeiros argumentos para `gethostbyname_r` são os mesmos da versão do Solaris, mas os três argumentos restantes para a versão do Solaris são combinados em uma nova estrutura `hostent_data` (que deve ser alocada pelo chamador) e um ponteiro para essa estrutura é o terceiro e último argumento. As funções normais `gethostbyname` e `gethostbyaddr` no Digital Unix 4.0 e no HP-UX 10.30 são reentrantes utilizando dados específicos do thread (Seção 26.5). Uma história interessante sobre o desenvolvimento das funções `_r` do Solaris 2.x é encontrada em Maslen (1997).

Por fim, embora uma versão reentrante de `gethostbyname` possa fornecer segurança a partir de diferentes threads que a chamam ao mesmo tempo, isso não diz nada sobre a reentrância das funções subjacentes do resolvedor.

11.20 Funções de pesquisa de endereço IPv6 obsoletas

Enquanto o IPv6 estava sendo desenvolvido, a API para solicitar a pesquisa de um endereço IPv6 passou por várias iterações. A API resultante não era suficientemente flexível e complicada, assim tornou-se obsoleta na RFC 2553 (Gilligan *et al*., 1999). A RFC 2553 introduziu suas próprias novas funções que, por fim, foram simplesmente substituídas por `getaddrinfo` e `getnameinfo` na RFC 3493 (Gilligan *et al*., 2003). Esta seção descreverá brevemente algumas das APIs antigas para ajudar na conversão de programas que utilizam a API antiga.

A constante `RES_USE_INET6`

Como `gethostbyname` não tem um argumento para especificar qual família de endereços é de interesse (como a entrada de estrutura `hints.ai_family` da `getaddrinfo`), a primeira revisão da API utilizava a constante `RES_USE_INET6`, que teve de ser adicionada aos flags do resolvedor utilizando uma interface interna e privada. Essa API não era muito portável uma vez que os sistemas que utilizavam uma interface resolvedora interna diferente tinham de simular a interface resolvedora do BIND para fornecê-la.

Ativar `RES_USE_INET6` fazia com que `gethostbyname` pesquisasse primeiro os registros AAAA e somente pesquisava registros A se um nome não tivesse nenhum registro AAAA. Como a estrutura `hostent` tem apenas um campo de comprimento de endereço, `gethostbyname` poderia retornar apenas endereços IPv6 ou IPv4, mas não ambos.

Ativar `RES_USE_INET6` também fazia com que `gethostbyname2` retornasse endereços IPv4 como endereços IPv6 mapeados de IPv4. Descreveremos `gethostbyname2` a seguir.

A função `gethostbyname2`

A função `gethostbyname2` adiciona como argumento uma família de endereços a `gethostbyname`.

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

struct hostent *gethostbyname2(const char *name, int af);
```

Retorna: ponteiro não-nulo se OK, `NULL` em erro com `h_errno` configurado

Quando o argumento `af` é `AF_INET`, `gethostbyname2` comporta-se da mesma maneira como `gethostbyname`, pesquisando e retornando endereços IPv4. Quando o argu-

mento `af` é `AF_INET6`, `gethostbyname2` pesquisa e retorna somente registros AAAA para endereços IPv6.

A função `getipnodebyname`

A RFC 2553 (Gilligan *et al*., 1999) tornou `RES_USE_INET6` e `gethostbyname2` obsoletos por causa da natureza global do flag `RES_USE_INET6` e do desejo de fornecer mais controle sobre as informações retornadas. Essa RFC introduziu a função `getipnodebyname` para solucionar alguns desses problemas.

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

struct hostent *getipnodebyname(const char *name, int af,
                                int flags, int *error_num);
```

Retorna: ponteiro não-nulo se OK, `NULL` em erro com `error_num` configurado

Essa função retorna um ponteiro para a mesma estrutura `hostent` que descrevemos com `gethostbyname`. Os argumentos `af` e `flags` mapeiam diretamente para os argumentos `hints.ai_family` e `hints.ai_flags` de `getaddrinfo`. Para segurança do thread, o valor de retorno é alocado dinamicamente, portanto, ele deve ser liberado com a função `freehostent`.

```
#include <netdb.h>

void freehostent(struct hostent *ptr);
```

A função `getipnodebyname` e suas funções `getipnodebyaddr` correspondentes tornaram-se obsoletas pela RFC 3493 (Gilligan *et al*., 2003), a favor de `getaddrinfo` e `getnameinfo`.

11.21 Outras informações sobre rede

Nosso foco neste capítulo foram hostnames, endereços IP, nomes de serviço e seus números de porta. Mas, examinando um cenário maior, há quatro tipos de informações (relacionadas à rede) que uma aplicação poderia querer pesquisar: hosts, redes, protocolos e serviços. A maioria das pesquisas é para hosts (`gethostbyname` e `gethostbyaddr`), com um número menor para serviços (`getservbyname` e `getservbyaddr`) e um número ainda menor para redes e protocolos.

Todos os quatro tipos de informações podem ser armazenados em um arquivo e três funções são definidas para cada um deles:

1. Uma função `get`*XXX*`ent` que lê a próxima entrada no arquivo, abrindo-o se necessário.
2. Uma função `set`*XXX*`ent` que abre (se ainda não estiver aberto) e reposiciona o ponteiro no início do arquivo.
3. Uma função `end`*XXX*`ent` que fecha o arquivo.

Cada um dos quatro tipos de informações define sua própria estrutura e as seguintes definições são fornecidas incluindo o cabeçalho `<netdb.h>`: as estruturas `hostent`, `netent`, `protoent` e `servent`.

Além das três funções `get`, `set` e `end`, que permitem um processamento seqüencial do arquivo, cada um dos quatro tipos de informações fornece algumas funções de *pesquisa com chaves* (“*keyed lookup* functions”). Essas funções passam pelo arquivo seqüencialmente (chamando a função `get` *XXX* `ent` para ler cada linha), mas, em vez de retornar cada linha ao chamador, procuram uma entrada que corresponda a um argumento. Essas funções de pesquisa com chaves têm nomes na forma `get`*XXX*`by`*YYY*. Por exemplo, as duas funções de pesquisa com chaves para as informações sobre o host são `gethostbyname` (procura uma entrada que corresponda a um hostname) e `gethostbyaddr` (procura uma entrada que corresponda a um endereço IP). A Figura 11.21 resume essas informações.

Informações	Arquivo de dados	Estrutura	Funções de pesquisa com chaves
Hosts	`/etc/hosts`	`hostent`	`gethostbyaddr`, `gethostbyname`
Redes	`/etc/networks`	`netent`	`getnetbyaddr`, `getnetbyname`
Protocolos	`/etc/protocols`	`protoent`	`getprotobyname`, `getprotobynumber`
Serviços	`/etc/services`	`servent`	`getservbyname`, `getservbyport`

Figura 11.21 Quatro tipos de informações relacionadas à rede.

Como isso se aplica quando o DNS está sendo utilizado? Primeiro, somente as informações sobre o host e a rede estão disponíveis por meio do DNS. Informações sobre protocolos e serviços sempre são lidas a partir do arquivo correspondente. Mencionamos anteriormente neste capítulo (Figura 11.1) que diferentes implementações empregam diferentes maneiras de o administrador especificar se deve utilizar o DNS ou um arquivo para informações sobre o host e a rede.

Segundo, se o DNS estiver sendo utilizado para informações sobre o host e a rede, somente as funções de pesquisa com chaves fazem sentido. Você não pode, por exemplo, utilizar `gethostent` e esperar seqüenciar por todas as entradas no DNS! Se `gethostent` for chamada, ela lê somente o arquivo `/etc/hosts` e evita o DNS.

> Embora as informações de rede possam ser disponibilizadas pelo DNS, poucas pessoas configuram isso. Albitz e Liu (2001) descrevem esse recurso. Em geral, porém, os administradores constroem e mantêm um arquivo `/etc/networks` que é utilizado no lugar do DNS. O programa `netstat` com a opção `-i` utiliza esse arquivo, se presente, e imprime o nome para cada rede. Contudo, o endereçamento sem classe (Seção A.4) torna essas funções relativamente inúteis e elas simplesmente não suportam o IPv6; portanto, aplicações novas devem evitar utilizar nomes de rede.

11.22 Resumo

O conjunto de funções que uma aplicação chama para converter um hostname em um endereço IP e vice-versa é chamado de funções *resolvedoras* (*resolver*, em inglês). As duas funções `gethostbyname` e `gethostbyaddr` são os pontos de entrada históricos. Com a mudança para o IPv6 e modelos de programação com thread, as funções `getaddrinfo` e `getnameinfo` são mais úteis, com a capacidade de resolver endereços IPv6 e suas convenções de chamada seguras a threads.

A função comumente utilizada que lida com nomes de serviço e números de porta é `getservbyname`, que recebe um nome de serviço e retorna uma estrutura que contém o número de porta. Esse mapeamento normalmente está contido em um arquivo de texto. Há funções adicionais para mapear nomes de protocolos para números de protocolo e nomes de rede para números de rede, mas estas raramente são utilizadas.

Uma outra alternativa que não mencionamos é chamar as funções resolvedoras diretamente, em vez de utilizar `gethostbyname` e `gethostbyaddr`. Um programa que invo-

ca o DNS dessa maneira é `sendmail`, que procura um registro MX, algo que as funções `gethostby`*XXX* não podem fazer. As funções resolvedoras têm nomes que iniciam com `res_`; a função `res_init` é um exemplo. Uma descrição dessas funções e um exemplo de programa que as chama podem ser encontrados no Capítulo 15 de Albitz e Liu (2001) e digitar `man resolver` deve exibir as páginas man para essas funções.

Exercícios

11.1 Modifique o programa na Figura 11.3 para chamar `gethostbyaddr` para cada endereço retornado e então imprimir o `h_name` retornado. Primeiro execute o programa que especifica um hostname com somente um endereço IP e então execute o programa que especifica um hostname com mais de um endereço IP. O que acontece?

11.2 Corrija o problema mostrado no exercício anterior.

11.3 Execute a Figura 11.4 especificando um nome de serviço de `chargen`.

11.4 Execute a Figura 11.4 especificando um endereço IP de pontos decimais como o hostname. Suas funções resolvedoras permitem isso? Modifique a Figura 11.4 para permitir um endereço IP de pontos decimais como o hostname e uma string de números decimais de porta como o nome de serviço. Ao testar o endereço IP para uma string de pontos decimais com pontos ou para um hostname, em qual ordem esses dois testes devem ser realizados?

11.5 Modifique a Figura 11.4 para funcionar com o IPv4 ou o IPv6.

11.6 Modifique a Figura 8.9 para consultar o DNS e comparar o endereço IP retornado com todos os endereços IP do host de destino. Isto é, chame `gethostbyaddr` utilizando o endereço IP retornado por `recvfrom`, seguido por `gethostbyname` para localizar todos os endereços IP do host.

11.7 Na Figura 11.12, o chamador deve passar um ponteiro para um inteiro a fim de obter o tamanho do endereço do protocolo. Se o chamador não fizer isso (isto é, passar um ponteiro nulo como o argumento final), como ainda poderá obter o tamanho real dos endereços do protocolo?

11.8 Modifique a Figura 11.14 para chamar `getnameinfo` em vez de `sock_ntop`. Quais flags você deve passar para `getnameinfo`?

11.9 Na Seção 7.5, discutimos o roubo de porta com a opção de soquete `SO_REUSEADDR`. Para ver como isso funciona, construa o servidor UDP de data/hora independente de protocolo na Figura 11.19. Inicie uma das instâncias do servidor em uma janela, vinculando o endereço curinga e alguma porta da sua escolha. Inicie um cliente em uma outra janela e verifique se esse servidor está tratando o cliente (observe `printf` no servidor). Em seguida, inicie uma outra instância do servidor em uma outra janela, dessa vez vinculando um dos endereços unicast do host e a mesma porta do primeiro servidor. Qual problema você encontra imediatamente? Corrija esse problema e reinicie esse segundo servidor. Inicie um cliente, envie um datagrama e verifique se o segundo servidor roubou a porta do primeiro servidor. Se possível, inicie o segundo servidor novamente a partir de uma diferente conta de login no primeiro servidor para ver se o roubo ainda é bem-sucedido. Alguns fornecedores não permitirão o segundo bind a menos que o ID do usuário seja o mesmo do processo que já tem a porta vinculada.

11.10 No final da Seção 2.12, mostramos dois exemplos de `telnet`: para o servidor de data/hora e para o servidor de eco. Sabendo que um cliente passa pelos passos `gethostbyname` e `connect`, quais linhas geradas pelo cliente indicam quais passos?

11.11 `getnameinfo` pode levar muito tempo (até 80 segundos) para retornar um erro se um endereço IP de um hostname não puder ser encontrado. Escreva uma nova função identificada como `getnameinfo_timeo` que recebe um argumento do tipo inteiro adicional especificando o número máximo de segundos de espera para uma resposta. Se o timer expirar e o flag `NI_NAMEREQD` não estiver especificado, simplesmente chame `inet_ntop` e retorne uma string de endereço.

PARTE III

Soquetes Avançados

Capítulo 12

Interoperabilidade entre IPv4 e IPv6

12.1 Visão geral

Nos próximos anos, provavelmente haverá uma transição gradual na Internet do IPv4 para o IPv6. Durante essa fase de transição, é importante que aplicações IPv4 existentes continuem a funcionar com aplicações IPv6 mais recentes. Por exemplo, um distribuidor não pode fornecer um cliente `telnet` que funcione apenas com servidores `telnet` IPv6, mas deve fornecer um cliente que funcione com servidores IPv4 e outro que funcione com servidores IPv6. Melhor ainda seria um cliente `telnet` IPv6 que possa funcionar com servidores IPv4 e IPv6, juntamente com um servidor `telnet` que possa funcionar tanto com clientes IPv4 como IPv6. Veremos como isso é feito neste capítulo.

Vamos supor por todo este capítulo que os hosts estão executando *pilhas duais*, isto é, uma pilha de protocolos IPv4 e outra de protocolos IPv6. Nosso exemplo na Figura 2.1 é um host de pilha dual. Os hosts e roteadores, provavelmente, irão executar dessa maneira por muitos anos durante a transição para o IPv6. Em algum momento, muitos sistemas serão capazes de desativar a pilha IPv4, mas somente o tempo dirá quando (e se) isso ocorrerá.

Neste capítulo, discutiremos como as aplicações IPv4 e IPv6 podem se comunicar entre si. Há quatro combinações de clientes e servidores que utilizam o IPv4 ou o IPv6, as quais serão mostradas na Figura 12.1.

Não discutiremos em detalhes os dois cenários em que o cliente e o servidor utilizam o mesmo protocolo. Os casos interessantes são quando o cliente e servidor utilizam protocolos diferentes.

	Servidor IPv4	**Servidor IPv6**
Cliente IPv4	Quase todos os clientes e servidores existentes	Discutido na Seção 12.2
Cliente IPv6	Discutido na Seção 12.3	Modificações simples para a maioria dos clientes e servidores existentes (por exemplo, Figuras 1.5 e 1.6)

Figura 12.1 Combinações de clientes e servidores utilizando o IPv4 ou o IPv6.

12.2 Cliente IPv4, servidor IPv6

Uma propriedade geral de um host de pilha dual é que os servidores IPv6 podem tratar tanto clientes IPv4 como IPv6. Isso é feito utilizando endereços IPv6 mapeados de IPv4 (Figura A.10). A Figura 12.2 mostra um exemplo disso.

Temos um cliente IPv4 e um cliente IPv6 à esquerda. O servidor à direita é escrito utilizando o IPv6 que está em execução em um host de pilha dual. O servidor criou um soquete TCP ouvinte com o IPv6 que está vinculado ao endereço curinga IPv6 e à porta 9999 TCP.

Vamos supor que os clientes e o servidor estão na mesma ethernet. Eles também poderiam estar conectados por roteadores, contanto que todos os roteadores suportem o IPv4 e o IPv6, mas isso não acrescenta nada a esta discussão. A Seção B.3 discutirá um caso diferente em que os clientes e servidores IPv6 são conectados somente por roteadores IPv4.

Agora, vamos supor que ambos os clientes enviam segmentos SYN para estabelecer uma conexão com o servidor. O host cliente IPv4 enviará o SYN em um datagrama IPv4 e o host cliente IPv6 enviará o SYN em um datagrama IPv6. O segmento TCP do cliente IPv4 aparece na rede como um cabeçalho Ethernet seguido por um cabeçalho IPv4, um cabeçalho TCP e dados TCP. O cabeçalho Ethernet contém um campo do tipo `0x0800`, que identifica o quadro como um quadro IPv4. O cabeçalho TCP contém a porta de destino 9999. (O Apêndice A abrange em mais detalhes o formato e o conteúdo desse cabeçalho.) O endereço IP de destino no cabeçalho IPv4, que não mostramos, seria 206.62.226.42.

O segmento TCP do cliente IPv6 aparece na rede como um cabeçalho Ethernet seguido por um cabeçalho IPv6, um cabeçalho TCP e os dados TCP. O cabeçalho Ethernet contém um campo do tipo `0x86dd`, que identifica o quadro como um quadro IPv6. O cabeçalho TCP tem o mesmo formato do cabeçalho TCP no pacote IPv4 e contém a porta de destino de 9999. O endereço IP de destino no cabeçalho IPv6, que não mostramos, seria `5f1b:df00:ce3e:e200:20:800:2b37:6426`.

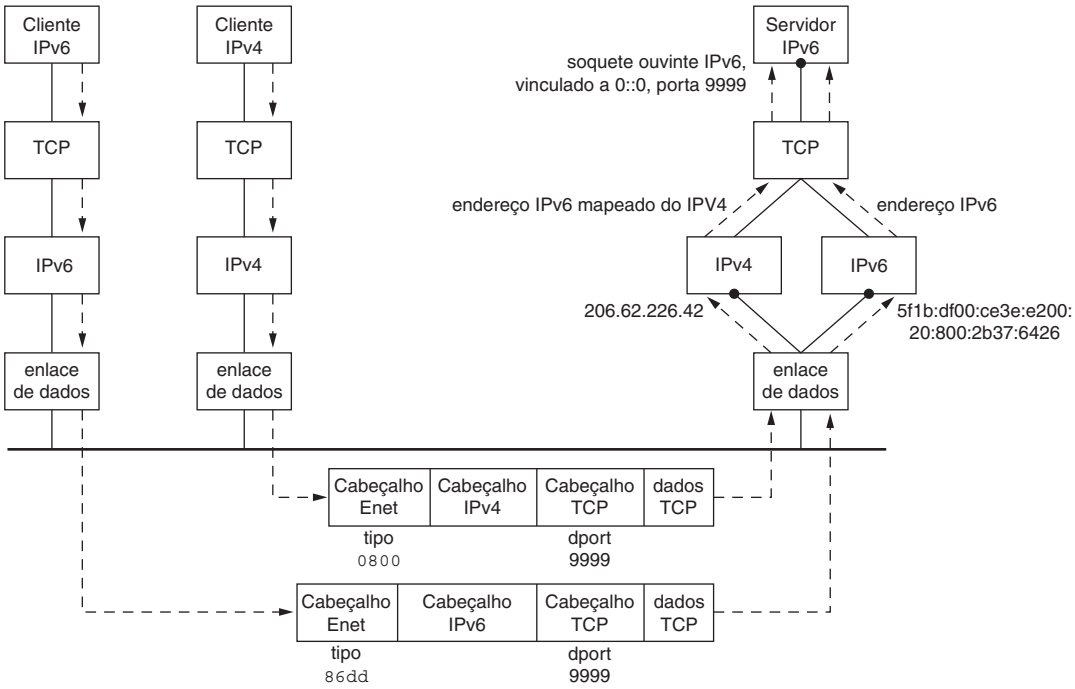


Figura 12.2 O servidor IPv6 em um host de pilha dual atendendo clientes IPv4 e IPv6.

O enlace de dados receptor examina o campo do tipo Ethernet e passa cada quadro para o módulo IP apropriado. O módulo IPv4, provavelmente em conjunção com o módulo TCP, detecta que o soquete de destino é um soquete IPv6 e o endereço IPv4 de origem no cabeçalho IPv4 é convertido no endereço IPv6 equivalente mapeado do IPv4. Esse endereço mapeado retorna ao soquete IPv6 como o endereço IPv6 do cliente quando `accept` retorna ao servidor com a conexão do cliente IPv4. Todos os datagramas restantes para essa conexão são datagramas IPv4.

Quando `accept` retorna ao servidor com a conexão do cliente IPv6, o endereço IPv6 do cliente não muda, qualquer que seja o endereço de origem que aparece no cabeçalho IPv6. Todos os datagramas restantes para essa conexão são datagramas IPv6.

Podemos resumir os passos que permitem a um cliente TCP IPv4 comunicar-se com um servidor IPv6 como a seguir:

1. O servidor IPv6 inicia, cria um soquete IPv6 ouvinte e supomos que ele chama `bind` para vincular o endereço curinga ao soquete.
2. O cliente IPv4 chama `gethostbyname` e localiza um registro A para o servidor. O host servidor terá tanto um registro A como um registro AAAA, visto que ele suporta os dois protocolos, mas o cliente IPv4 solicita somente um registro A.
3. O cliente chama `connect` e o host cliente envia um SYN do IPv4 ao servidor.
4. O host servidor recebe o SYN do IPv4 direcionado ao soquete IPv6 ouvinte, configura um flag indicando que essa conexão utiliza endereços IPv6 mapeados do IPv4 e responde com um SYN/ACK do IPv4 quando a conexão é estabelecida; o endereço retornado ao servidor por `accept` é o endereço IPv6 mapeado do IPv4.
5. Quando o host do servidor envia ao endereço IPv6 mapeado do IPv4, sua pilha de IP gera datagramas IPv4 para o endereço IPv4. Portanto, toda a comunicação entre o cliente e o servidor acontece utilizando datagramas IPv4.
6. A menos que o servidor verifique explicitamente se esse endereço IPv6 é um endereço IPv6 mapeado de IPv4 (utilizando a macro `IN6_IS_ADDR_V4MAPPED` descrita na Seção 12.4), ele nunca saberá que está se comunicando com um cliente IPv4. A pilha de protocolos dual trata esse detalhe. De maneira semelhante, o cliente IPv4 não tem nenhuma idéia de que está se comunicando com um servidor IPv6.

Uma suposição subjacente nesse cenário é que o host servidor na pilha dual tem tanto um endereço IPv4 como um endereço IPv6. Isso funcionará até que todos os endereços IPv4 tenham sido tomados.

O cenário é semelhante para um servidor UDP com o IPv6, mas o formato do endereço pode mudar para cada datagrama. Por exemplo, se o servidor IPv6 receber um datagrama de um cliente IPv4, o endereço retornado por `recvfrom` será o endereço IPv6 mapeado do IPv4. O servidor responde à solicitação desse cliente chamando `sendto` com o endereço IPv6 mapeado do IPv4 como destino. Esse formato de endereço instrui o kernel a enviar um datagrama IPv4 ao cliente. Mas o próximo datagrama recebido para o servidor poderia ser um datagrama IPv6, e assim `recvfrom` retornará o endereço IPv6. Se o servidor responder, o kernel irá gerar um datagrama IPv6.

A Figura 12.3 resume como é processado um datagrama IPv4 ou IPv6 recebido, dependendo do tipo do soquete receptor, para TCP e UDP, supondo um host de pilha dual.

- Se um soquete IPv4 receber um datagrama IPv4, nada especial ocorre. Essas são as duas setas rotuladas "IPv4" na figura: uma para o TCP e outra para UDP. Datagramas IPv4 são trocados entre o cliente e o servidor.
- Se um soquete IPv6 receber um datagrama IPv6, nada especial ocorre. Essas são as duas setas rotuladas "IPv6" na figura: uma para o TCP e outra para UDP. Datagramas IPv6 são trocados entre o cliente e o servidor.

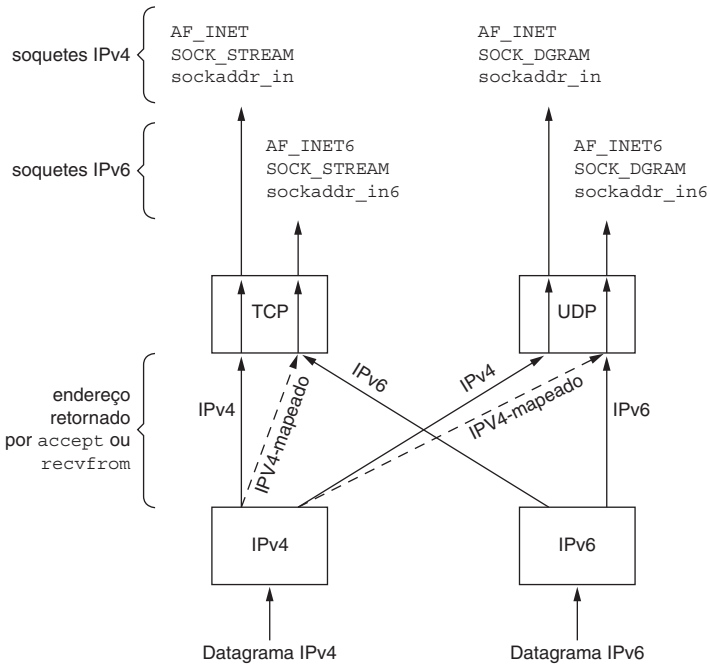


Figura 12.3 Processamento dos datagramas IPv4 ou IPv6 recebidos, dependendo do tipo do soquete de recebimento.

- Quando um soquete IPv6 recebe um datagrama IPv4, o kernel retorna o endereço IPv6 mapeado do IPv4 correspondente como o endereço retornado por `accept` (TCP) ou `recvfrom` (UDP). Essas são as duas setas tracejadas na figura. Esse mapeamento é possível porque um endereço IPv4 sempre pode ser representado como um endereço IPv6. Datagramas IPv4 são trocados entre o cliente e o servidor.
- O inverso do parágrafo anterior é falso. Em geral, um endereço IPv6 não pode ser representado como um endereço IPv4; portanto, não há nenhuma seta a partir da caixa de protocolo IPv6 para os dois soquetes IPv4.

A maioria dos hosts de pilha dual deve utilizar as seguintes regras ao lidar com soquetes ouvintes:

1. Um soquete IPv4 ouvinte pode aceitar conexões entrantes provenientes apenas de clientes IPv4.
2. Se um servidor tiver um soquete IPv6 ouvinte vinculado ao endereço curinga e a opção de soquete `IPV6_V6ONLY` (Seção 7.8) não estiver configurada, esse soquete poderá aceitar conexões entrantes provenientes de clientes IPv4 ou IPv6. Para uma conexão proveniente de um cliente IPv4, o endereço local do servidor para a conexão será o endereço IPv6 mapeado do IPv4 correspondente.
3. Se um servidor tiver um soquete IPv6 ouvinte vinculado a um endereço IPv6 outro que um endereço IPv6 mapeado de IPv4 ou vinculado ao endereço curinga, porém configurou a opção de soquete `IPV6_V6ONLY` (Seção 7.8), esse soquete poderá aceitar conexões entrantes provenientes apenas de clientes IPv6.

12.3 Cliente IPv6, servidor IPv4

Agora faremos uma troca dos protocolos utilizados pelo cliente e pelo servidor no exemplo da seção anterior. Primeiro, leve em consideração um cliente TCP IPv6 em execução em um host de pilha dual.

1. Um servidor IPv4 inicia apenas em um host IPv4 e cria um soquete IPv4 ouvinte.
2. O cliente IPv6 inicia e chama `getaddrinfo` solicitando apenas endereços IPv6 (ele solicita a família de endereços `AF_INET6` e configura o flag `AI_V4MAPPED` na sua estrutura *hints*). Como o host servidor com apenas o IPv4 tem somente registros A, vemos na Figura 11.8 que um endereço IPv6 mapeado de IPv4 é retornado ao cliente.
3. O cliente IPv6 chama `connect` com o endereço IPv6 mapeado de IPv4 na estrutura de endereço de soquete IPv6. O kernel detecta o endereço mapeado e automaticamente envia um SYN de IPv4 para o servidor.
4. O servidor responde com um SYN/ACK de IPv4 e a conexão é estabelecida utilizando datagramas IPv4.

Podemos resumir esse cenário na Figura 12.4.

- Se um cliente TCP IPv4 chamar `connect` para especificar um endereço IPv4 ou se um cliente UDP com IPv4 chamar `sendto` para especificar um endereço IPv4, nada especial ocorre. Essas são as duas setas rotuladas "IPv4" na figura.
- Se um cliente TCP com IPv6 chamar `connect` para especificar um endereço IPv6 ou se um cliente UDP com IPv6 chamar `sendto` para especificar um endereço IPv6, nada especial ocorre. Essas são as duas setas rotuladas "IPv6" na figura.

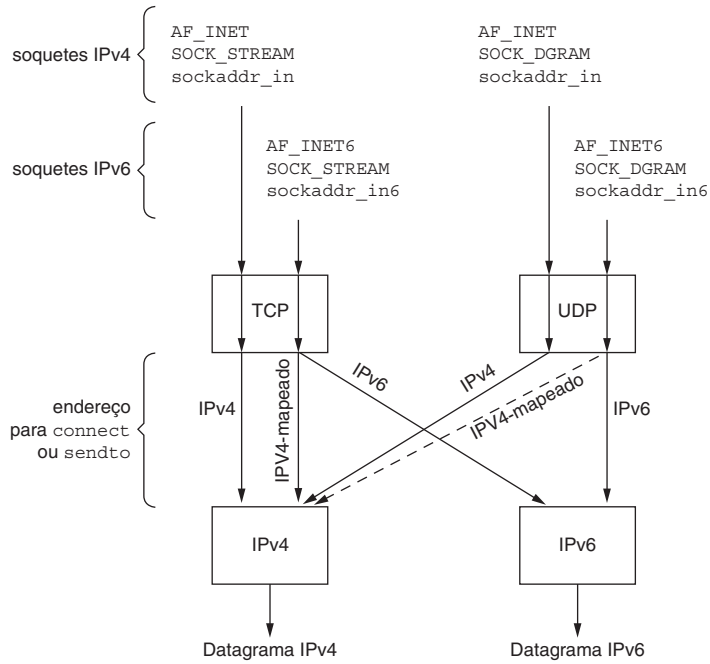


Figura 12.4 Processamento de solicitações do cliente, dependendo do tipo de endereço e do tipo de soquete.

- Se um cliente TCP com IPv6 especificar um endereço IPv6 mapeado de IPv4 para `connect` ou se um cliente UDP com IPv6 especificar um endereço IPv6 mapeado de IPv4 para `sendto`, o kernel irá detectar o endereço mapeado e fará com que um datagrama IPv4 seja enviado no lugar de um datagrama IPv6. Essas são as duas setas tracejadas na figura.
- Um cliente IPv4 não pode especificar um endereço IPv6 para `connect` ou `sendto` porque um endereço IPv6 de 16 bytes não cabe nos 4 bytes da estrutura `in_addr` dentro da estrutura `sockaddr_in` do IPv4. Portanto, não há nenhuma seta saindo dos soquetes IPv4 para a caixa de protocolo IPv6 na figura.

Na seção anterior (um datagrama IPv4 recebido por um soquete de servidor IPv6), a conversão do endereço recebido para o endereço IPv6 mapeado do IPv4 é feita pelo kernel e é retornada transparentemente para a aplicação por `accept` ou `recvfrom`. Nesta seção (um datagrama IPv4 que precisa ser enviado a um soquete IPv6), a conversão do endereço IPv4 no endereço IPv6 mapeado do IPv4 é feita pelo resolvedor de acordo com as regras na Figura 11.8 e o endereço mapeado é então passado transparentemente pela aplicação para `connect` ou `sendto`.

Resumo da interoperabilidade

A Figura 12.5 resume esta seção e a seção anterior, mais as combinações de clientes e servidores.

Cada coluna contém "IPv4" ou "IPv6" se a combinação for perfeita, indicando qual protocolo é utilizado, ou "(não)" se a combinação for inválida. A terceira coluna na linha final está marcada com um asterisco porque a interoperabilidade depende do endereço escolhido pelo cliente. Escolher o registro AAAA e enviar um datagrama IPv6 não funcionará. Mas escolher o registro A, que retorna ao cliente como um endereço IPv6 mapeado de IPv4, faz com que um datagrama IPv4 seja enviado e funcionará. Fazendo loop por todos os endereços que `getaddrinfo` retorna, como mostrado na Figura 11.4, podemos assegurar que tentaremos (talvez depois de alguns tempos-limite) o endereço IPv6 mapeado de IPv4.

Embora pareça que cinco entradas na tabela não irão interoperar, no mundo real, em um futuro não muito distante, as implementações IPv6 estarão em hosts de pilha dual e não serão implementações somente com o IPv6. Se, portanto, removermos a segunda linha e a segunda coluna, todas as entradas "(não)" desaparecem e o único problema será a entrada com o asterisco.

	Servidor IPv4 Host somente com IPv4 (somente A)	Servidor IPv6 Host somente com IPv6 (somente AAAA)	Servidor IPv4 Host de pilha dual (A e AAAA)	Servidor IPv6 Host de pilha dual (A e AAAA)
Cliente IPv4, host apenas com IPv4	IPv4	(não)	IPv4	IPv4
Cliente IPv6, host apenas com IPv6	(não)	IPv6	(não)	IPv6
Cliente IPv4, host de pilha dual	IPv4	(não)	IPv4	IPv4
Cliente IPv6, host de pilha dual	IPv4	IPv6	(não*)	IPv6

Figura 12.5 Resumo da interoperabilidade entre clientes e servidores IPv4 e IPv6.

12.4 Macros para teste de endereço IPv6

Há uma pequena classe de aplicações IPv6 que devem saber se elas estão conversando com um peer IPv4. Essas aplicações precisam saber se o endereço do peer é um endereço IPv6 mapeado de IPv4. As 12 macros a seguir são definidas para testar um endereço IPv6 para certas propriedades.

```
#include <netinet/in.h>

int IN6_IS_ADDR_UNSPECIFIED(const struct in6_addr *aptr);
int IN6_IS_ADDR_LOOPBACK(const struct in6_addr * aptr) ;
int IN6_IS_ADDR_MULTICAST(const struct in6_addr * aptr);
int IN6_IS_ADDR_LINKLOCAL(const struct in6_addr * aptr);
int IN6_IS_ADDR_SITELOCAL(const struct in6_addr * aptr);
int IN6_IS_ADDR_V4MAPPED(const struct in6_addr * aptr);
int IN6_IS_ADDR_V4COMPAT(const struct in6_addr * aptr);

int IN6_IS_ADDR_MC_NODELOCAL(const struct in6_addr * aptr);
int IN6_IS_ADDR_MC_LINKLOCAL(const struct in6_addr * aptr);
int IN6_IS_ADDR_MC_SITELOCAL(const struct in6_addr * aptr);
int IN6_IS_ADDR_MC_ORGLOCAL(const struct in6_addr * aptr);
int IN6_IS_ADDR_MC_GLOBAL(const struct in6_addr * aptr);
```

Todas retornam: não-zero se o endereço IPv6 for do tipo especificado, zero do contrário

As sete primeiras macros testam o tipo básico de endereço IPv6. Mostramos esses vários tipos de endereços na Seção A.5. As cinco macros finais testam o escopo de um endereço multicast com IPv6 (Seção 21.2).

> Os endereços compatíveis com o IPv4 são utilizados por um mecanismo de transição que perdeu a preferência. Na verdade, você provavelmente não verá esse tipo de endereço nem precisará testá-lo.

Um cliente IPv6 poderia chamar a macro `IN6_IS_ADDR_V4MAPPED` para testar o endereço IPv6 retornado pelo resolvedor. Um servidor IPv6 poderia chamar essa macro para testar o endereço IPv6 retornado por `accept` ou `recvfrom`.

Como um exemplo de uma aplicação que precisa dessa macro, considere o FTP e seu comando `PORT`. Se iniciarmos um cliente FTP, efetuarmos login em um servidor FTP e emitirmos um comando FTP `dir`, o cliente FTP enviará um comando `PORT` ao servidor FTP pela conexão de controle. Isso informa ao servidor o endereço e porta IP do cliente, com os quais o servidor então cria uma conexão de dados. (O Capítulo 27 do TCPv1 contém todos os detalhes sobre o protocolo de aplicação FTP.) Mas, um cliente FTP com o IPv6 deve saber se o servidor é um servidor IPv4 ou um servidor IPv6, porque o primeiro requer um comando na forma `PORT a1,a2,a3,a4,p1,p2` na qual os primeiros quatro números (cada um entre 0 e 255) formam o endereço IPv4 de 4 bytes e os dois últimos formam o número de porta de 2 bytes. Um servidor IPv6, porém, requer um comando `EPRT` (RFC 2428 [Allman, Ostermann e Metz, 1998]), contendo uma família de endereços, o endereço do formato de texto e a porta do formato de texto. O Exercício 12.1 fornecerá um exemplo do comportamento do protocolo FTP com IPv6 e IPv4.

12.5 Portabilidade do código-fonte

A maioria das aplicações de rede existentes é escrita assumindo o IPv4. As estruturas `sockaddr_in` são alocadas e preenchidas e as chamadas a `socket` especificam `AF_INET` como o primeiro argumento. Vimos, na conversão da Figura 1.5 para a 1.6, que essas aplicações IPv4 poderiam ser convertidas para utilizar o IPv6 sem muito esforço. Muitas alterações que mostramos poderiam ocorrer automaticamente utilizando alguns scripts de edição. Programas que são mais dependentes do IPv4, que utilizam recursos como multicasting, opções IP ou soquetes brutos, exigirão mais esforços para realizar a conversão.

Se convertermos uma aplicação para utilizar o IPv6 e a distribuirmos em código-fonte, teremos de nos preocupar com a possibilidade de o sistema do destinatário suportar ou não o

IPv6. A maneira típica de tratar isso é com `#ifdefs` por todo o código, utilizando o IPv6 quando possível (como vimos neste capítulo, um cliente IPv6 pode se comunicar com servidores IPv4 e vice-versa). O problema dessa abordagem é que o código torna-se muito rapidamente propenso a falhas com `#ifdefs` e é mais difícil de seguir e manter.

Uma melhor abordagem é considerar a possibilidade de mudar para o IPv6 para tornar o programa independente de protocolo. O primeiro passo é remover as chamadas a `gethostbyname` e `gethostbyaddr` e utilizar as funções `getaddrinfo` e `getnameinfo` que descrevemos no capítulo anterior. Essa abordagem permite lidar com estruturas de endereço de soquete como objetos opacos, referenciados por um ponteiro e um tamanho, que é exatamente o que as funções de soquete básicas fazem: `bind`, `connect`, `recvfrom` e assim por diante. Nossas funções `sock_`*`XXX`* na Seção 3.8 podem ajudar a manipular essas estruturas, independentemente dos protocolos IPv4 ou IPv6. Obviamente, essas funções contêm `#ifdefs` para tratar o IPv4 e o IPv6, mas ocultar toda essa dependência de protocolo em algumas funções de biblioteca torna nosso código mais simples. Desenvolveremos um conjunto de funções `mcast_`*`XXX`* na Seção 21.7 que podem tornar aplicações multicast independentes de IPv4 ou de IPv6.

Um outro ponto a considerar é o que acontecerá se compilarmos nosso código-fonte em um sistema que suporta tanto o IPv4 como o IPv6, distribuirmos o código executável ou arquivos-objeto (mas não o código-fonte) e uma pessoa executar nossa aplicação em um sistema que não suporta o IPv6? Há uma probabilidade de que o servidor de nome local suporte registros AAAA e retorne registros AAAA e registros A para um peer com o qual nossa aplicação tenta se conectar. Se nossa aplicação, que é compatível com o IPv6, chamar `socket` para criar um soquete IPv6, ela falhará se o host não suportar o IPv6. Tratamos isso nas funções auxiliares descritas no capítulo anterior ignorando o erro a partir de `socket` e tentando o endereço seguinte na lista retornada pelo servidor de nome. Supondo que o peer tem um registro A e que o servidor de nome retorne o registro A além de quaisquer registros AAAA, a criação de um soquete IPv4 será bem-sucedida. Esse é o tipo de funcionalidade que faz parte de uma função de biblioteca e não do código-fonte de toda aplicação.

Para habilitar a passagem de descritores de soquete para programas com somente IPv4 ou IPv6, a RFC 2133 (Gilligan *et al*., 1997) introduziu a opção de soquete `IPV6_ADDRFORM` que poderia retornar ou, potencialmente, alterar a família de endereços associada a um soquete. Entretanto, a semântica nunca foi completamente descrita e era útil apenas em casos muito específicos; portanto, ela foi removida na revisão seguinte da API.

12.6 Resumo

Um servidor IPv6 em um host de pilha dual pode atender tanto clientes IPv4 como clientes IPv6. Um cliente IPv4 continua a enviar datagramas IPv4 ao servidor, mas a pilha de protocolos do servidor converte o endereço do cliente em um endereço IPv6 mapeado de IPv4, visto que o servidor IPv6 está lidando com estruturas do endereço de soquete IPv6.

De maneira semelhante, um cliente IPv6 em um host de pilha dual pode se comunicar com um servidor IPv4. O resolvedor do cliente retornará endereços IPv6 mapeados de IPv4 para todos os registros A do servidor e chamar `connect` para um desses endereços resulta na pilha dual enviando um segmento SYN IPv4. Somente alguns clientes e servidores especiais precisam conhecer o protocolo utilizado pelo peer (por exemplo, FTP) e a macro `IN6_IS_ADDR_V4MAPPED` pode ser utilizada para verificar se o peer está utilizando o IPv4.

Exercícios

12.1 Inicie um cliente FTP IPv6 em um host de pilha dual executando o IPv4 e o IPv6. Conecte-se a um servidor FTP IPv4, certifique-se de que o cliente está no modo "ativo" (talvez emitindo o comando `passive` para desativar o modo "passivo"), emita o comando `debug` e então o comando `dir`. Em seguida, realize as mesmas operações, mas para um servidor IPv6, e compare os comandos `PORT` emitidos como resultado dos comandos `dir`.

12.2 Escreva um programa que requeira um argumento da linha de comando que seja um endereço IPv4 com pontos decimais. Crie um soquete de TCP IPv4 e chame `bind` para vincular esse endereço ao soquete juntamente com alguma porta (por exemplo, 9999). Chame `listen` e então `pause`. Escreva um programa semelhante que receba uma string hexadecimal IPv6 como o argumento da linha de comando e crie um soquete de TCP IPv6 ouvinte. Inicie o programa IPv4, especificando o endereço curinga como o argumento. Em seguida, vá a uma outra janela e inicie o programa IPv6, especificando o endereço curinga IPv6 como o argumento. Você pode iniciar o programa IPv6 uma vez que o programa IPv4 já vinculou essa porta? A opção de soquete `SO_REUSEADDR` faz alguma diferença? E se você primeiro iniciar o programa IPv6 e então tentar iniciar o programa IPv4?

Capítulo 13

Processos Daemon e o Superservidor `inetd`

13.1 Visão Geral

Um *daemon* é um processo executado no segundo plano e que não está associado a um terminal controlador. Em geral, os sistemas Unix têm muitos processos que são daemons (entre 20 e 50) em execução no segundo plano e que realizam diferentes tarefas administrativas.

A falta de um terminal controlador é, em geral, um efeito colateral do fato de ele ser iniciado por um script de inicialização de sistema (por exemplo, em tempo de inicialização). Mas, se um daemon for iniciado por um usuário que digita em um prompt de shell, é importante que o daemon se desassocie do terminal controlador para evitar qualquer interação indesejável com o controle de tarefas, gerenciamento de sessão de terminal ou, simplesmente, para evitar uma saída inesperada para o terminal do daemon em execução no segundo plano.

Há diversas maneiras de iniciar um daemon:

1. Durante a inicialização de sistema, muitos daemons são iniciados pelos scripts de inicialização de sistema. Freqüentemente, esses scripts estão no diretório `/etc` ou em um diretório cujo nome inicia por `/etc/rc`, mas sua localização e conteúdo são dependentes de implementação. Os daemons iniciados por esses scripts começam com privilégios de superusuário.

   Alguns servidores de rede costumam ser iniciados a partir desses scripts: o superservidor `inetd` (descrito mais adiante neste capítulo), um servidor Web e um servidor de correio (freqüentemente `sendmail`). O daemon `syslogd`, que descreveremos na Seção 13.2, é normalmente iniciado por um desses scripts.

2. Muitos servidores de rede são iniciados pelo superservidor `inetd`. O próprio `inetd` é iniciado a partir de um dos scripts no Passo 1. O `inetd` recebe solicitações de rede (Telnet, FTP, etc.) e, quando uma solicitação chega, invoca o servidor real (servidor Telnet, servidor FTP, etc.).

3. A execução de programas numa base regular é realizada pelo daemon `cron`; e os programas que ele invoca são executados como daemons. O próprio daemon `cron` é iniciado no Passo 1 durante a inicialização de sistema.

4. A execução de um programa em um tempo no futuro é especificada pelo comando `at`. Normalmente, o daemon `cron` inicia esses programas quando chega sua vez, portanto, esses programas são executados como daemons.
5. Os daemons podem ser iniciados a partir de terminais de usuário, em primeiro ou segundo plano. Isso costuma ser feito ao se testar um daemon ou reiniciar um que foi terminado por alguma razão.

Como um daemon não tem um terminal controlador, precisa, de alguma maneira, gerar saída para mensagens quando algo acontece; mensagens informacionais normais ou mensagens emergenciais que precisam ser tratadas por um administrador. A função `syslog` é a maneira-padrão de gerar saída e enviar essas mensagens ao daemon `syslogd`.

13.2 Daemon `syslogd`

Os sistemas Unix normalmente iniciam um daemon chamado `syslogd` a partir de um dos scripts de inicialização de sistema e o executam enquanto o sistema está ativo. As implementações derivadas do Berkeley do `syslogd` realizam as seguintes ações na inicialização:

1. O arquivo de configuração, normalmente `/etc/syslog.conf`, é lido especificando o que fazer com cada tipo de mensagem de registro em log que o daemon pode receber. Essas mensagens podem ser anexadas a um arquivo (um caso especial disso é o arquivo `/dev/console`, que grava a mensagem no console), gravadas para um usuário específico (se esse usuário estiver conectado) ou encaminhadas ao daemon `syslogd` em um outro host.
2. Um soquete de domínio do Unix é criado e vinculado ao nome de caminho `/var/run/log` (em alguns sistemas, `/dev/log`).
3. Um soquete UDP é criado e vinculado à porta 514 (o serviço `syslog`).
4. O nome de caminho `/dev/klog` é aberto. Quaisquer mensagens de erro de dentro do kernel aparecem como uma entrada nesse dispositivo.

O daemon `syslogd` executa em um loop infinito que chama `select`, esperando que qualquer um dos seus três descritores (nos Passos 2, 3 e 4) torne-se legível; lê a mensagem de log e faz o que o arquivo de configuração diz para fazer com essa mensagem. Se o daemon receber o sinal `SIGHUP`, ele relê o arquivo de configuração.

Poderíamos enviar mensagens de log ao daemon `syslogd` a partir dos nossos daemons criando um soquete de datagrama de domínio do Unix e enviando nossas mensagens ao nome de caminho ao qual o daemon foi vinculado, mas uma interface mais fácil é a função `syslog` que descreveremos na próxima seção. Alternativamente, poderíamos criar um soquete de UDP e enviar nossas mensagens de log ao endereço de loopback e porta 514.

> Implementações mais recentes desativam a criação do soquete UDP, a menos que especificado pelo administrador, visto que permitir a qualquer pessoa enviar datagramas UDP a essa porta abre o sistema a ataques de recusa de serviço, em que alguém poderia encher o sistema de arquivos (por exemplo, com arquivos de log) ou fazer com que mensagens de log sejam descartadas (por exemplo, inundando o buffer de recebimento de soquete de `syslog`).
>
> Há diferenças entre as várias implementações do `syslogd`. Por exemplo, soquetes de domínio Unix são utilizados por implementações derivadas do Berkeley, mas implementações System V utilizam um driver de log STREAMS. Diferentes implementações derivadas do Berkeley utilizam diferentes nomes de caminho para o soquete de domínio Unix. Podemos ignorar todos esses detalhes se utilizarmos a função `syslog`.

13.3 Função `syslog`

Como um daemon não tem um terminal controlador, ele não pode simplesmente chamar `fprintf` para imprimir `stderr`. A técnica comum para registrar mensagens em log a partir de um daemon é chamar a função `syslog`.

```
#include <syslog.h>

void syslog(int priority, const char *message, ... );
```

Embora essa função tenha sido originalmente desenvolvida para sistemas BSD, hoje ela é disponibilizada por praticamente todos os fornecedores Unix. A descrição de `syslog` na especificação POSIX é coerente com o que descrevemos aqui. A RFC 3164 fornece a documentação do protocolo syslog do BSD. O argumento *priority* (prioridade) é uma combinação de um *nível* (*level*) e um *recurso* (*facility*), que mostramos nas Figuras 13.1 e 13.2. Detalhes adicionais sobre *priority* podem ser encontrados na RFC 3164. *message* é como uma string de formato para `printf`, com a adição de uma especificação `%m`, que é substituída pela mensagem de erro correspondente ao valor atual de `errno`. Uma nova linha pode aparecer no final de *message*, mas não é obrigatória.

As mensagens de log têm um *nível* entre 0 e 7, que mostramos na Figura 13.1. Esses níveis são valores ordenados. Se nenhum *nível* for especificado pelo remetente, `LOG_NOTICE` é o padrão.

As mensagens de log também contêm um *recurso* (*facility*) para identificar o tipo de processo que envia a mensagem. Mostramos os diferentes valores na Figura 13.2. Se nenhum *recurso* for especificado, `LOG_USER` é o padrão.

Por exemplo, a seguinte chamada poderia ser emitida por um daemon quando uma chamada à função `rename` falha inesperadamente:

```
syslog(LOG_INFO|LOG_LOCAL2, "rename(%s, %s): %m", file1, file2);
```

O propósito de *recurso* e *nível* é permitir que todas as mensagens a partir de um dado recurso sejam tratadas da mesma maneira no arquivo `/etc/syslog.conf` ou permitir que todas as mensagens de um dado nível sejam tratadas da mesma maneira. Por exemplo, o arquivo de configuração poderia conter as linhas

```
kern.*          /dev/console
local7.debug    /var/log/cisco.log
```

para especificar que todas as mensagens de kernel sejam registradas em log no console e todas as mensagens `debug` provenientes do recurso `local7` sejam anexadas ao arquivo `/var/log/cisco.log`.

nível	Valor	Descrição
LOG_EMERG	0	Sistema não é utilizável (prioridade mais alta)
LOG_ALERT	1	Uma ação deve ser tomada imediatamente
LOG_CRIT	2	Condições críticas
LOG_ERR	3	Condições de erro
LOG_WARNING	4	Condições de alerta
LOG_NOTICE	5	Condição normal, exceto significativa (padrão)
LOG_INFO	6	Informacional
LOG_DEBUG	7	Mensagens de nível de depuração (menor prioridade)

Figura 13.1 *nível* de mensagens de log.

recurso	Descrição
LOG_AUTH	Mensagens de segurança/autorização
LOG_AUTHPRIV	Mensagens de segurança/autorização (privadas)
LOG_CRON	daemon cron
LOG_DAEMON	Daemons de sistema
LOG_FTP	Daemon FTP
LOG_KERN	Mensagens de kernel
LOG_LOCAL0	Utilização local
LOG_LOCAL1	Utilização local
LOG_LOCAL2	Utilização local
LOG_LOCAL3	Utilização local
LOG_LOCAL4	Utilização local
LOG_LOCAL5	Utilização local
LOG_LOCAL6	Utilização local
LOG_LOCAL7	Utilização local
LOG_LPR	Sistema de impressora de linhas
LOG_MAIL	Sistema de envio de mensagens
LOG_NEWS	Sistema de notícias de rede
LOG_SYSLOG	Mensagens geradas internamente por syslogd
LOG_USER	Mensagens aleatórias de nível do usuário (padrão)
LOG_UUCP	Sistema UUCP

Figura 13.2 *recurso* de mensagens de log.

Quando a aplicação chama `syslog` pela primeira vez, ela cria um soquete de datagrama de domínio Unix e então chama `connect` com o nome de caminho bem-conhecido do soquete criado pelo daemon `syslogd` (por exemplo, `/var/run/log`). Esse soquete permanece aberto até que o processo termine. Alternativamente, o processo pode chamar `openlog` e `closelog`.

```
#include <syslog.h>

void openlog(const char *ident, int options, int facility);

void closelog(void);
```

`openlog` pode ser chamado antes da primeira chamada a `syslog`; e `closelog` pode ser chamado quando a aplicação termina de enviar mensagens de log.

ident é uma string que será prefixada a cada mensagem de log por `syslog`. Freqüentemente, isso é o nome do programa.

O argumento *options* é formado como o OU lógico de uma ou mais constantes na Figura 13.3.

Normalmente, o soquete de domínio do Unix não é criado quando `openlog` é chamado. Em vez disso, ele é aberto durante a primeira chamada a `syslog`. A opção `LOG_NDELAY` faz com que o soquete seja criado quando `openlog` é chamado.

opções	Descrição
LOG_CONS	Registre em log no console se não puder enviar ao daemon syslogd
LOG_NDELAY	Não retarde a abertura, crie o soquete agora
LOG_PERROR	Registre em log o erro-padrão bem como envie essa mensagem ao daemon syslogd
LOG_PID	Registre em log o ID de processo de cada mensagem

Figura 13.3 *opções* para `openlog`.

O argumento *facility* para `openlog` especifica um recurso-padrão para quaisquer chamadas subseqüentes a `syslog` que não especificam um recurso. Alguns daemons chamam `openlog` e especificam o recurso (que normalmente não muda para um dado daemon). Eles então especificam somente o *nível* em cada chamada a `syslog` (uma vez que *nível* pode mudar dependendo do erro).

As mensagens de log também podem ser geradas pelo comando `logger`. Isso pode ser utilizado de dentro de scripts de shell, por exemplo, para enviar mensagens a `syslogd`.

13.4 Função `daemon_init`

A Figura 13.4 mostra uma função identificada como `daemon_init` que podemos chamar (normalmente, a partir de um servidor) para transformar o processo em daemon. Essa função deve ser adequada para utilização em todas as variantes do Unix, mas algumas oferecem uma função da biblioteca C denominada `daemon` que fornece recursos semelhantes. O BSD oferece a função `daemon`, da mesma maneira como o Linux.

daemon_init.c

```
 1 #include   "unp.h"
 2 #include   <syslog.h>

 3 #define MAXFD     64

 4 extern int daemon_proc;          /* definido no error.c */

 5 int
 6 daemon_init(const char *pname, int facility)
 7 {
 8     int    i;
 9     pid_t  pid;

10     if ( (pid = Fork()) < 0)
11         return (-1);
12     else if (pid)
13         _exit(0);                 /* pai termina */

14     /* filho 1 continua... */

15     if (setsid() < 0)             /* torna-se líder da sessão */
16         return (-1);

17     Signal(SIGHUP, SIG_IGN);
18     if ( (pid = Fork()) < 0)
19         return (-1);
20     else if (pid)
21         _exit(0);                 /* filho 1 termina */

22     /* filho 2 continua... */

23     daemon_proc = 1;              /* para funções err_XXX()*/

24     chdir("/");                   /* altera o diretório de trabalho */

25     /* fecha descritores de arquivo */
26     for (i = 0; i < MAXFD; i++)
27         close(i);

28     /* redireciona stdin, stdout e stderr para / dev/null */
29         open("/dev/null", O_RDONLY);
30     open("/dev/null", O_RDWR);
31     open("/dev/null", O_RDWR);

32     openlog(pname, LOG_PID, facility);

33     return (0);                   /* bem-sucedida */
34 }
```

daemon_init.c

Figura 13.4 função `daemon_init`: transforma o processo em daemon.

fork

10–13 Primeiro, chamamos `fork` e, então, o pai termina e o filho continua. Se o processo foi iniciado como um comando de shell no primeiro plano, quando o pai termina, o shell pensa que o comando foi concluído. Isso executa automaticamente o processo-filho no segundo plano. Além disso, o filho herda o ID do grupo de processos a partir do pai, mas obtém seu próprio ID de processo. Isso garante que o filho não seja um líder do grupo de processos, que é requerido para a próxima chamada a `setsid`.

setsid

15–16 `setsid` é uma função POSIX que cria uma nova sessão. (O Capítulo 9 do APUE discute os relacionamentos e sessões de processos em detalhes.) O processo torna-se o líder de sessão da nova sessão, torna-se o líder do grupo de processos de um novo grupo de processos e não tem nenhum terminal controlador.

Ignorando SIGHUP e fork novamente

17–21 Ignoramos `SIGHUP` e chamamos `fork` novamente. Quando essa função retorna, na verdade o pai é o primeiro filho e termina, deixando o segundo filho em execução. O propósito desse segundo `fork` é garantir que o daemon não possa adquirir automaticamente um terminal controlador se ele abrir um dispositivo terminal no futuro. Quando um líder de sessão sem um terminal controlador abre um dispositivo terminal (que atualmente não está em outro terminal controlador de uma sessão), o terminal torna-se o terminal controlador do líder de sessão. Mas, chamando `fork` pela segunda vez, garantimos que o segundo filho não mais seja líder de sessão, assim ele não pode adquirir um terminal controlador. Devemos ignorar `SIGHUP` porque, quando o líder de sessão termina (o primeiro filho), todos os processos na sessão (nosso segundo filho) recebem o sinal `SIGHUP`.

Configuração de flag para funções de erros

23 Configuramos a variável global `daemon_proc` como não-zero. Essa variável externa é definida pelas nossas funções `err_`*XXX* (Seção D.3) e, quando seu valor é não-zero, ela nos informa a chamar `syslog` em vez de um `fprintf` para erro-padrão. Isso poupa-nos de examinar todo o nosso código e chamar uma das nossas funções de erro se o servidor não estiver sendo executado como um daemon (isto é, quando estamos testando o servidor), mas chama `syslog` se estiver sendo executado como um daemon.

Alteração do diretório funcional

24 Alteramos o diretório de trabalho para o diretório root, embora alguns daemons talvez tenham alguma razão para mudar para outro diretório. Por exemplo, um daemon de impressora poderia alterar para o diretório do spool de impressora, onde realiza todo o trabalho. Se o daemon gerar um arquivo `core`, esse arquivo é gerado no diretório de trabalho atual.

Uma outra razão de alterar o diretório de trabalho é que o daemon poderia ter sido iniciado em qualquer sistema de arquivos e, se permanecer aí, esse sistema de arquivos não poderá ser desmontado (pelo menos não sem utilizar algumas medidas enérgicas potencialmente destrutivas).

Fechamento de quaisquer descritores abertos

25–27 Fechamos quaisquer descritores abertos herdados do processo que executou o daemon (normalmente um shell). O problema é determinar o descritor mais alto em utilização: não há nenhuma função do Unix que forneça esse valor. Há maneiras de determinar o número máximo de descritores que o processo pode abrir, mas mesmo isso torna-se complicado (consulte a pá-

gina 43 do APUE) porque o limite pode ser infinito. Nossa solução é fechar os primeiros 64 descritores, mesmo que a maioria deles provavelmente não esteja aberta.

> O Solaris fornece uma função chamada `closefrom` para utilização pelos daemons para resolver esse problema.

Redirecionamento `stdin`, `stdout` e `stderr` para `/dev/null`

29–31 Abrimos `/dev/null` para entrada-padrão, saída-padrão e erro-padrão. Isso garante que esses descritores comuns estejam abertos e que uma leitura a partir de qualquer um deles retorne 0 (EOF), e o kernel simplesmente descarta qualquer coisa gravada neles. A razão para abrir esses descritores é que qualquer função de biblioteca chamada pelo daemon, que assuma que pode ler da entrada-padrão ou gravar na saída-padrão ou no erro-padrão, não falhará. Essa falha é potencialmente perigosa. Se o daemon abrir um soquete para um cliente, esse descritor de soquete termina como `stdout` ou `stderr` e alguma chamada errônea para algo como `perror` envia então dados inesperados a um cliente.

Utilização de `syslogd` para erros

32 `openlog` é chamado. O primeiro argumento é proveniente do chamador e, normalmente, é o nome do programa (por exemplo, `argv[0]`). Especificamos que o ID do processo deve ser adicionado a cada mensagem de log. *facility* também é especificado pelo chamador, como um dos valores de recurso da Figura 13.2, ou 0, se o default de `LOG_USER` for aceitável.

Observamos que, como um daemon executa sem um terminal controlador, ele nunca deve receber o sinal `SIGHUP` do kernel. Portanto, vários daemons utilizam esse sinal como uma notificação do administrador de que o arquivo de configuração do daemon foi alterado e o daemon deve reler o arquivo. Dois outros sinais que um daemon nunca deve receber são `SIGINT` e `SIGWINCH`, de modo que os daemons possam utilizá-los de maneira segura como uma outra forma de os administradores indicarem alguma alteração à qual o daemon deve reagir.

Exemplo: Servidor de data/hora como um daemon

A Figura 13.5 é uma modificação do nosso servidor de data/hora independente de protocolo da Figura 11.14 que chama nossa função `daemon_init` para executar como daemons.

Há somente duas alterações: chamamos a função `daemon_init` logo que o programa inicia e chamamos a função `err_msg`, em vez de `printf`, para imprimir o endereço e a porta IP do cliente. Na verdade, se quisermos que nossos programas sejam capazes de executar como um daemon, deveremos evitar chamar as funções `printf` e `fprintf` e, em vez delas, utilizar a função `err_msg`.

Observe como verificamos `argc` e emitimos a mensagem de uso apropriada *antes* de chamar `daemon_init`. Isso permite ao usuário que inicia o daemon obter feedback imediato se o comando tiver um número incorreto de argumentos. Depois de chamar `daemon_init`, todas as mensagens de erro subseqüentes vão para syslog.

Se executarmos esse programa no nosso host `linux` e então verificarmos o arquivo `/var/log/messages` (em que enviamos todas as mensagens `LOG_USER`) depois de conectarmos na mesma máquina (por exemplo, host local), teremos

```
Jun 10 09:54:37 linux daytimetcpsrv2[24288]:
connection from 127.0.0.1.55862
```

(Quebramos a linha longa.) A data, a hora e o hostname são prefixados automaticamente pelo daemon `syslogd`.

inetd/daytimetcpsrv2.c

```
 1 #include   "unp.h"
 2 #include   <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int    listenfd, connfd;
 7     socklen_t addrlen, len;
 8     struct sockaddr *cliaddr;
 9     char   buff[MAXLINE];
10     time_t ticks;

11     if (argc < 2 || argc > 3)
12         err_quit("usage: daytimetcpsrv2 [ <host> ] <service or port>");

13     daemon_init(argv[0], 0);

14     if (argc == 2)
15         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
16     else
17         listenfd = Tcp_listen(argv[1], argv[2], &addrlen);

18     cliaddr = Malloc(addrlen);

19     for ( ; ; ) {
20         len = addrlen;
21         connfd = Accept(listenfd, cliaddr, &len);
22         err_msg("connection from %s", Sock_ntop(cliaddr, len));

23         ticks = time(NULL);
24         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
25         Write(connfd, buff, strlen(buff));

26         Close(connfd);
27     }
28 }
```

inetd/daytimetcpsrv2.c

Figura 13.5 Servidor de data/hora independente de protocolo que executa como um daemon.

13.5 Daemon `inetd`

Em um sistema Unix típico, há muitos servidores simplesmente esperando que uma solicitação de cliente chegue. Exemplos são: FTP, Telnet, Rlogin, TFTP e assim por diante. Com sistemas anteriores ao 4.3BSD, cada um desses serviços tinha um processo associado. Esse processo era iniciado em tempo de inicialização a partir do arquivo `/etc/rc` e cada processo fazia tarefas de inicialização quase idênticas: crie um soquete, chame `bind` para vinculá-lo ao soquete de uma porta bem-conhecida do servidor, espere uma conexão (se TCP) ou um datagrama (se UDP) e então chame `fork`. O processo-filho atendeu o cliente e o pai esperou a próxima solicitação do cliente. Há dois problemas com esse modelo:

1. Todos esses daemons continham um código de inicialização quase idêntico, primeiro com relação à criação do soquete e, também, com relação a transformar-se em um processo de daemon (semelhante à nossa função `daemon_init`).
2. Cada daemon ocupava um slot na tabela de processos, mas estava adormecido a maior parte do tempo.

A distribuição 4.3BSD simplificou isso fornecendo um *superservidor* de Internet: o daemon `inetd`. Esse daemon pode ser utilizado pelos servidores que utilizam TCP ou UDP. Ele não trata outros protocolos, como soquetes de domínio do Unix. Esse daemon corrige os dois problemas recém-mencionados:

1. Ele simplifica escrever os processos de daemon uma vez que a maioria dos detalhes de inicialização é tratada por `inetd`. Isso evita a necessidade de cada servidor chamar nossa função `daemon_init`.
2. Ele permite que um único processo (`inetd`) espere múltiplos serviços para solicitações entrantes de clientes, em vez de um processo para cada serviço. Isso reduz o número total de processos no sistema.

O processo `inetd` se auto-estabelece como um daemon utilizando as técnicas que descrevemos com a nossa função `daemon_init`. Em seguida, ele lê e processa o arquivo de configuração, em geral `/etc/inetd.conf`. Esse arquivo especifica os serviços que o superservidor deve tratar e o que fazer quando uma solicitação de serviço chega. Cada linha contém os campos mostrados na Figura 13.6.

Campo	Descrição
service-name	Deve estar em `/etc/services`
socket-type	`stream` (TCP) ou `dgram` (UDP)
protocol	Dever estar em `/etc/protocols`: `tcp` ou `udp`
wait-flag	Em geral `nowait` para TCP ou `wait` para UDP
login-name	Proveniente de `/etc/passwd`: em geral `root`
server-program	O nome do caminho completo para `exec`
server-program-arguments	Argumentos para `exec`

Figura 13.6 Campos no arquivo `inetd.conf`.

Algumas linhas de amostra são:

```
ftp     stream  tcp  nowait  root     /usr/bin/ftpd      ftpd -l
telnet  stream  tcp  nowait  root     /usr/bin/telnetd   telnetd
login   stream  tcp  nowait  root     /usr/bin/rlogind   rlogind -s
tftp    dgram   udp  wait    nobody   /usr/bin/tftpd     tftpd -s /tftpboot
```

O nome real do servidor sempre é passado como o primeiro argumento para um programa quando ele é executado.

> Essa figura e as linhas de amostra são apenas exemplos. A maioria dos fornecedores adicionou seus próprios recursos a `inetd`. Exemplos são a capacidade de tratar servidores RPC, além de TCP e UDP, e a capacidade de tratar protocolos outros que o TCP e o UDP. Além disso, o nome de caminho para `exec` e os argumentos de linha de comando para o servidor obviamente dependem da implementação.
>
> O campo *wait-flag* pode ser um pouco confuso. Em geral, ele especifica se o daemon iniciado por `inetd` pretende controlar o soquete ouvinte associado ao serviço. Os serviços UDP não têm soquetes ouvintes e aceitadores separados e praticamente sempre são configurados como `wait`. Os serviços TCP podem ser tratados de uma ou outra maneira, a critério da pessoa que escreve o daemon, mas `nowait` é o mais comum.
>
> A interação do IPv6 com `/etc/inetd.conf` depende do fornecedor e é exigida atenção especial aos detalhes para conseguir o que você quer. Alguns utilizam um *protocol* `tcp6` ou `udp6` para indicar que um soquete IPv6 deve ser criado para um serviço. Alguns valores permitidos de *protocol* `tcp46` ou `udp46` indicam que o daemon quer soquetes que permitem tanto conexões IPv6 como IPv4. Esses nomes de protocolo especiais em geral não aparecem no arquivo `/etc/protocols`.

Uma descrição do que o daemon `inetd` faz é mostrada na Figura 13.7.

1. Na inicialização, ele lê o arquivo `/etc/inetd.conf` e cria um soquete do tipo apropriado (de fluxo ou de datagrama) para todos os serviços especificados no arquivo. O número máximo de servidores que `inetd` pode tratar depende do número má-

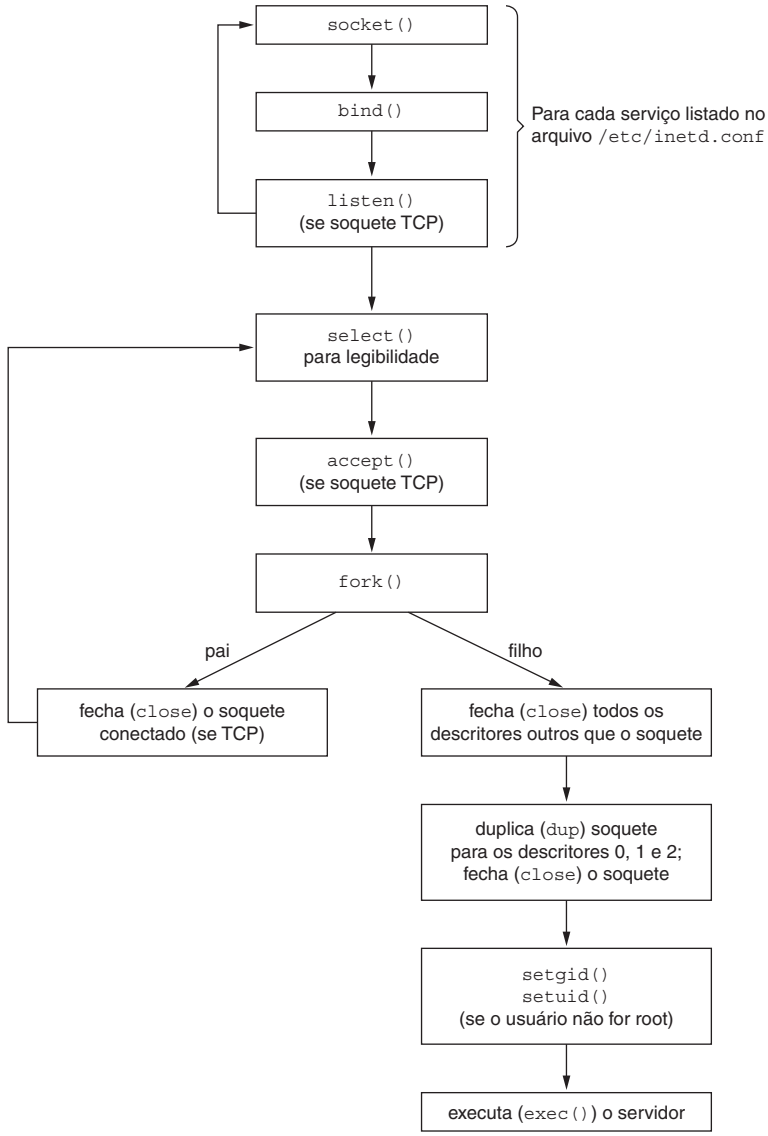


Figura 13.7 Passos seguidos por `inetd`.

ximo de descritores que `inetd` pode criar. Cada novo soquete é adicionado a um conjunto de descritores que será utilizado em uma chamada a `select`.

2. `bind` é chamado para o soquete, especificando a porta para o servidor e o endereço IP curinga. Esse número de porta TCP ou UDP é obtido chamando `getservbyname` com os campos *service-name* e *protocol* do arquivo de configuração como argumentos.
3. Para soquetes TCP, `listen` é chamado de modo que solicitações de conexões entrantes sejam aceitas. Esse passo não ocorre para soquetes de datagrama.
4. Depois que todos os soquetes são criados, `select` é chamado para esperar que qualquer um deles torne-se legível. Lembre-se, a partir do que foi discutido na Seção 6.3,

de que um soquete TCP ouvinte torna-se legível quando uma nova conexão está pronta para chamar `accept` para ser aceita, um soquete UDP torna-se legível quando um datagrama chega. `inetd` passa a maior parte do tempo bloqueado nessa chamada a `select` esperando que um soquete esteja legível.

5. Quando `select` retorna que um soquete está legível, se o soquete for TCP e o flag `nowait` é fornecido, `accept` é chamado para aceitar a nova conexão.

6. O daemon `inetd` chama `fork` e o processo-filho trata a solicitação de serviço. Isso é semelhante a um servidor concorrente-padrão (Seção 4.8).

   O filho fecha todos os descritores, exceto o de soquete que ele está tratando: o novo soquete conectado que retorna por `accept` para um servidor TCP ou o soquete UDP original. O filho chama `dup2` três vezes, duplicando o soquete nos descritores 0, 1 e 2 (entrada-padrão, saída-padrão e erro-padrão). O descritor de soquete original é então fechado. Fazendo isso, os únicos descritores que estão abertos no filho são 0, 1 e 2. Se o filho ler da entrada-padrão, ele estará lendo do soquete e qualquer coisa que ele grava na saída-padrão ou no erro-padrão é gravado no soquete. O filho chama `getpwnam` para obter a entrada do arquivo de senha para o campo *login-name* especificado no arquivo de configuração. Se esse campo não for `root`, o filho torna-se então o usuário especificado executando as chamadas de função `setgid` e `setuid`. (Visto que o processo `inetd` está executando com um ID de usuário 0, o processo-filho herda esse ID de usuário por meio de `fork` e é capaz de tornar-se qualquer usuário que ele escolher.)

   O processo-filho agora faz um `exec` para executar o *server-program* apropriado para tratar a solicitação, passando os argumentos especificados no arquivo de configuração.

7. Se o soquete for um soquete de fluxo, o processo-pai deve fechar o soquete conectado (como ocorre com o nosso servidor concorrente-padrão). O pai chama `select` novamente, esperando que o próximo soquete torne-se legível.

Se examinarmos em mais detalhes o tratamento do descritor que está acontecendo, a Figura 13.8 mostra os descritores no `inetd` quando uma nova solicitação de conexão chega de um cliente FTP.

A solicitação de conexão é direcionada à porta TCP 21, mas um novo soquete conectado é criado por `accept`.

A Figura 13.9 mostra os descritores no filho, após a chamada a `fork`, depois que o filho fechou todos os descritores, exceto o soquete conectado.

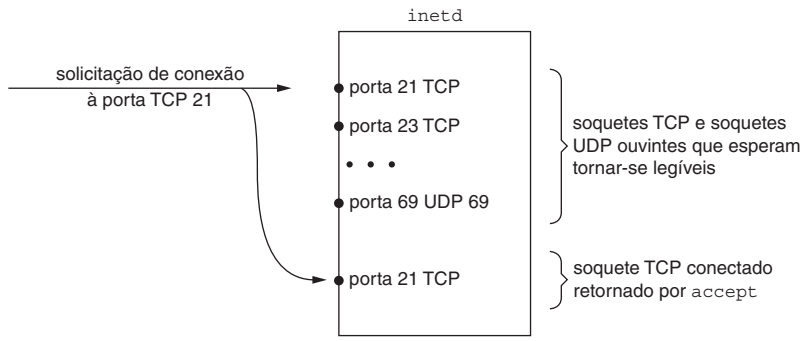


Figura 13.8 Descritores `inetd` quando a solicitação de conexão chega à porta TCP 21.

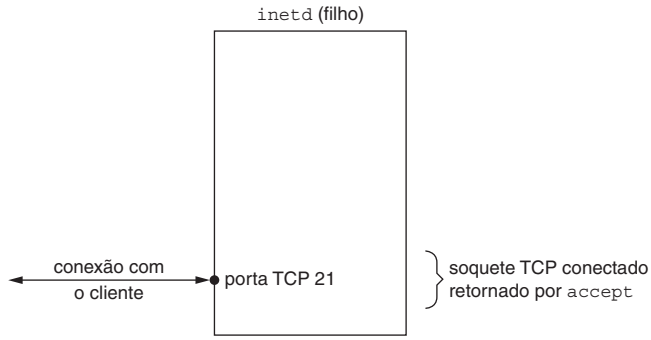


Figura 13.9 Descritores `inetd` no filho.

O próximo passo é o filho duplicar o soquete conectado para os descritores 0, 1 e 2 e então fechar o soquete conectado. Isso fornece os descritores mostrados na Figura 13.10.

O filho chama então `exec`. Lembre-se, da Seção 4.7, de que todos os descritores normalmente permanecem abertos por todo um `exec`, de modo que o servidor real que é `exec`utado utiliza qualquer um dos descritores, 0, 1 ou 2, para se comunicar com o cliente. Estes devem ser os únicos descritores abertos no servidor.

O cenário que descrevemos trata o caso em que o arquivo de configuração especifica `nowait` para o servidor. Isso é típico para todos os serviços TCP e significa que `inetd` não precisa esperar que seu filho termine antes de aceitar uma outra conexão para esse serviço. Se uma outra solicitação de conexão chegar no mesmo serviço, ela é retornada ao processo-pai logo que ele chamar `select` novamente. Os passos 4, 5 e 6 listados anteriormente são executados novamente e um outro processo-filho trata essa nova solicitação.

Especificar o flag `wait` para um serviço de datagrama altera os passos seguidos pelo processo-pai. Esse flag informa que `inetd` deve esperar que seu filho termine antes de selecionar esse soquete novamente. As seguintes alterações ocorrem:

1. Quando `fork` retorna no pai, o pai salva o ID de processo do filho. Isso permite ao pai conhecer quando esse processo-filho específico termina, examinando o valor retornado por `waitpid`.
2. O pai desativa o soquete para futuras chamadas a `select` utilizando a macro `FD_CLR` para desativar o bit no seu conjunto de descritores. Isso significa que o processo-filho assume o controle do soquete até que ele termine.

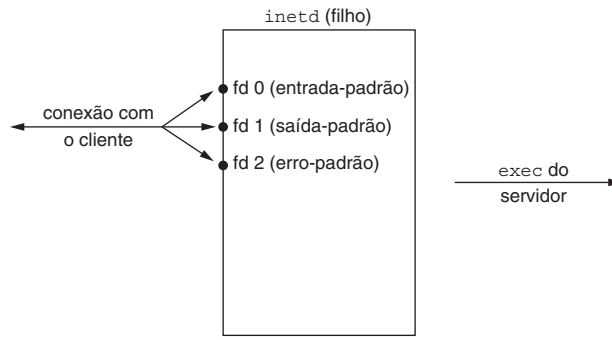


Figura 13.10 Descritores `inetd` depois de `dup2`.

3. Quando o filho termina, o pai é notificado por um sinal `SIGCHLD` e o handler de sinal do pai obtém o ID de processo do filho que termina. Ele reativa `select` para o soquete correspondente ativando o bit no seu conjunto de descritores para esse soquete.

A razão pela qual um servidor de datagrama assume o controle do soquete até que ele termine, evitando que `inetd` chame `select` nesse soquete para legibilidade (esperando um outro datagrama de cliente), é porque há somente um soquete para um servidor de datagrama, diferente de um servidor TCP que tem um soquete ouvinte e um soquete conectado por cliente. Se `inetd` não desativasse a legibilidade no soquete de datagrama e se o pai (`inetd`) fosse executado antes do filho, o datagrama do cliente ainda estaria no buffer de recebimento de soquete, fazendo com que `select` retornasse legível novamente e `inetd` chamasse `fork` para bifurcar um outro (desnecessário) filho. `inetd` deve ignorar o soquete de datagrama até que saiba que o filho leu o datagrama na fila de recebimento de soquete. A maneira como `inetd` sabe quando esse filho concluiu com o soquete é recebendo `SIGCHLD`, indicando que o filho terminou. Mostraremos um exemplo disso na Seção 22.7.

Os cinco serviços-padrão de Internet que descrevemos na Figura 2.18 são tratados internamente por `inetd` (veja o Exercício 13.2).

Como `inetd` é o processo que chama `accept` para um servidor TCP, o servidor real que é invocado por `inetd` normalmente chama `getpeername` para obter o endereço IP e o número da porta do cliente. Lembre-se da Figura 4.18, em que mostramos que, depois de um `fork` e um `exec` (que é o que `inetd` faz), a única maneira de o servidor real obter a identificação do cliente é chamando `getpeername`.

Normalmente, `inetd` não é utilizado para servidores de alto volume, notavelmente servidores Web e de correio. Por exemplo, `sendmail` é comumente executado como um servidor concorrente-padrão, como descrevemos na Seção 4.8. Desse modo, o custo do controle do processo para cada conexão cliente é apenas uma chamada a `fork`, enquanto o custo para um servidor TCP invocado por `inetd` é uma chamada a `fork` e outra a `exec`. Os servidores Web utilizam diversas técnicas para minimizar os overheads do controle de processo para cada conexão cliente, como discutiremos no Capítulo 30.

> Agora é comum encontrar um daemon estendido de serviços de Internet, chamado `xinetd`, no Linux e em outros sistemas. `xinetd` fornece a mesma função básica de `inetd`, mas também inclui uma longa lista de outros recursos interessantes. Esses recursos incluem opções para registro em log, aceitar ou rejeitar conexões com base no endereço do cliente, configurar serviços do tipo "um por arquivo", em vez de uma única configuração monolítica e muitas outras. Isso não é descrito em mais detalhes aqui visto que a idéia básica por trás do *superservidor* desses recursos é a mesma.

13.6 Função `daemon_inetd`

A Figura 13.11 mostra uma função identificada como `daemon_inetd` que pode ser chamada a partir de um servidor que sabemos que é invocado por `inetd`.

daemon_inetd.c

```
1 #include   "unp.h"
2 #include   <syslog.h>

3 extern int daemon_proc;          /* definido no error.c */

4 void
5 daemon_inetd(const char *pname, int facility)
6 {
7     daemon_proc = 1;             /* para nossas funções err_XXX ()*/
8     openlog(pname, LOG_PID, facility);
9 }
```

daemon_inetd.c

Figura 13.11 Função `daemon_inetd`: processo transformado em daemon executado pelo `inetd`.

Essa função é trivial se comparada com `daemon_init`, porque todos os passos da ativação do daemon são realizados pelo `inetd` quando ele inicia. Tudo o que fazemos é configurar o flag `daemon_proc` para nossas funções de erro (Figura D.3) e chamar `openlog` com os mesmos argumentos da chamada na Figura 13.4.

Exemplo: Servidor de data/hora como um daemon invocado por `inetd`

A Figura 13.12 é uma modificação do nosso servidor de data/hora da Figura 13.5 que pode ser invocado por `inetd`.

inetd/daytimetcpsrv3.c

```
 1 #include   "unp.h"
 2 #include   <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     socklen_t len;
 7     struct sockaddr *cliaddr;
 8     char   buff[MAXLINE];
 9     time_t ticks;

10     daemon_inetd(argv[0], 0);

11     cliaddr = Malloc(sizeof(struct sockaddr_storage));
12     len = sizeof(struct sockaddr_storage);
13     Getpeername(0, cliaddr, &len);
14     err_msg("connection from %s", Sock_ntop(cliaddr, len));

15     ticks = time(NULL);
16     snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
17     Write(0, buff, strlen(buff));

18     Close(0);                    /* fecha conexão TCP */
19     exit(0);
20 }
```

inetd/daytimetcpsrv3.c

Figura 13.12 Servidor de data/hora independente de protocolo que pode ser invocado por `inetd`.

Há duas alterações importantes nesse programa. Primeira, todo o código de criação do soquete já foi concluído: as chamadas a `tcp_listen` e a `accept`. Esses passos são tomados pelo `inetd` e referenciamos a conexão TCP utilizando o descritor 0 (entrada-padrão). Segunda, o loop `for` infinito desapareceu porque somos invocados uma vez por conexão cliente. Depois de atender esse cliente, terminamos.

Chamada a `getpeername`

11–14 Como não chamamos `tcp_listen`, não conhecemos o tamanho da estrutura do endereço de soquete que ele retorna, e como não chamamos `accept`, não conhecemos o endereço de protocolo do cliente. Portanto, alocamos um buffer para a estrutura do endereço de soquete utilizando `sizeof(struct sockaddr_storage)` e chamamos `getpeername` com o descritor 0 como o primeiro argumento.

Para executar esse exemplo no nosso sistema Solaris, primeiro atribuímos um nome ao serviço e à porta, adicionando a seguinte linha a `/etc/services`:

```
mydaytime 9999/tcp
```

Em seguida, adicionamos a seguinte linha a /etc/inetd.conf:

```
mydaytime stream tcp nowait andy
      /home/andy/daytimetcpsrv3 daytimetcpsrv3
```

(Quebramos a linha longa.) Colocamos o executável na localização especificada e enviamos o sinal SIGHUP ao inetd, informando-lhe a reler seu arquivo de configuração. O próximo passo é executar netstat para verificar se um soquete ouvinte foi criado na porta 9999 TCP.

```
solaris % netstat -na | grep 9999
      *.9999               *.*                0       0 49152      0 LISTEN
```

Então invocamos o servidor a partir de um outro host.

```
linux % telnet solaris 9999
Trying 192.168.1.20...
Connected to solaris.
Escape character is '^]'.
Tue Jun 10 11:04:02 2003
Connection closed by foreign host.
```

O arquivo /var/adm/messages (onde direcionamos e registramos em log as mensagens do recurso LOG_USER no nosso arquivo /etc/syslog.conf) contém a seguinte entrada:

```
Jun 10 11:04:02 solaris daytimetcpsrv3[28724]: connection from
                                                192.168.1.10.58145
```

13.7 Resumo

Os daemons são processos que executam no segundo plano independentemente do controle de todos os terminais. Muitos servidores de rede executam como daemons. Toda saída de um daemon é normalmente enviada ao daemon syslogd, chamando a função syslog. O administrador tem, portanto, controle total sobre o que acontece na mensagem, com base no daemon que a enviou e na importância dela.

Para iniciar um programa arbitrário e fazer com que ele execute como um daemon exige alguns passos: chamar fork para executar no segundo plano, chamar setsid para criar uma nova sessão POSIX e torná-lo o líder de sessão, chamar fork novamente para evitar obter um novo terminal controlador, alterar o diretório de trabalho e a máscara de modo de criação de arquivo e fechar todos os arquivos desnecessários. Nossa função daemon_init trata todos esses detalhes.

Muitos servidores Unix são iniciados pelo daemon inetd. Ele trata todos os passos requeridos para a ativação do daemon e, quando o servidor real é iniciado, o soquete é aberto na entrada-padrão, na saída-padrão e no erro-padrão. Isso permite omitir chamadas a socket, bind, listen e accept, uma vez que todos esses passos são tratados pelo inetd.

Exercícios

13.1 O que acontecerá à Figura 13.5 se movermos a chamada a daemon_init antes de os argumentos de linha de comando serem verificados, de modo que a chamada a err_quit ocorra depois dela?

13.2 Para os cinco serviços tratados internamente por inetd (Figura 2.18), considere a versão TCP e a versão UDP de cada um deles. Quais dos 10 servidores você acha que são implementados com uma chamada a fork e quais não requerem uma chamada a fork?

13.3 O que acontece se criarmos um soquete UDP, vincularmos a porta 7 ao soquete (o servidor de `echo`-padrão da Figura 2.18) e enviarmos um datagrama UDP a um servidor `chargen`?

13.4 A página man do Solaris 2.x para `inetd` descreve um flag `-t` que faz com que `inetd` chame `syslog` (com um recurso de `LOG_DAEMON` e um nível de `LOG_NOTICE`) para registrar em log o endereço e a porta IP do cliente para qualquer serviço TCP que o `inetd` trata. Como o `inetd` obtém essas informações?

Essa página man também informa que `inetd` não pode fazer isso para um serviço UDP. Por quê?

Há uma maneira de contornar essa limitação dos serviços UDP?

Capítulo 14

Funções de E/S Avançadas

14.1 Visão Geral

Este capítulo abrange diversas funções e técnicas que agrupamos na categoria "E/S avançada". Primeiro, é a configuração de um tempo-limite em uma operação de E/S, o que pode ser feito de três maneiras diferentes. Em seguida, há mais outras três variações nas funções `read` e `write`: `recv` e `send`, que permitem um quarto argumento que contém flags do processo para o kernel, `readv` e `writev`, que permitem especificar um vetor de buffers para entrada ou saída, `recvmsg` e `sendmsg`, que combina todos os recursos das outras funções de E/S juntamente com a nova capacidade de receber e enviar dados auxiliares.

Também consideramos como determinar a quantidade de dados no buffer de recebimento de soquete, como utilizar a biblioteca de E/S padrão do C com soquetes e discutimos algumas maneiras avançadas de esperar por eventos.

14.2 Tempos-limite de soquete

Há três maneiras de determinar um tempo-limite em uma operação de E/S com um soquete:

1. Chamar `alarm`, que gera o sinal `SIGALRM` quando o tempo especificado expirou. Isso envolve o tratamento de sinal, que pode diferir entre implementações e interferir em outras chamadas existentes a `alarm` no processo.
2. Bloquear esperando por uma E/S em `select`, que tem um tempo-limite predefinido, em vez de bloquear em uma chamada a `read` ou `write`.
3. Utilizar as opções de soquete `SO_RCVTIMEO` e `SO_SNDTIMEO` mais recentes. O problema dessa abordagem é que nem todas as implementações suportam essas duas opções de soquete.

Todas as três técnicas funcionam com operações de entrada e de saída (por exemplo, `read`, `write` e outras variações como `recvfrom` e `sendto`), mas também gostaríamos de uma técnica que pudéssemos utilizar com `connect`, visto que uma conexão TCP pode levar muito tempo para expirar (em geral, 75 segundos). `select` pode ser utilizada para determinar um tempo-limite em `connect` somente quando o soquete está em um modo

não-bloqueador (que mostramos na Seção 16.3), e as duas opções de soquete não funcionam com `connect`. Também observamos que as duas primeiras técnicas funcionam com qualquer descritor, enquanto a terceira funciona somente com os descritores de soquete.

Agora, mostraremos exemplos dessas três técnicas.

`connect` com um tempo-limite utilizando `SIGALRM`

A Figura 14.1 mostra nossa função `connect_timeo` que chama `connect` com um limite superior especificado pelo chamador. Os primeiros três argumentos são os três requeridos por `connect` e o quarto é o número de segundos a esperar.

lib/connect_timeo.c

```
 1 #include   "unp.h"
 2 static void connect_alarm(int);
 3 int
 4 connect_timeo(int sockfd, const SA *saptr, socklen_t salen, int nsec)
 5 {
 6     Sigfunc *sigfunc;
 7     int    n;
 8     sigfunc = Signal(SIGALRM, connect_alarm);
 9     if(alarm(nsec) != 0)
10         err_msg("connect_timeo: alarm was already set");
11     if ( (n = connect(sockfd, saptr, salen)) < 0) {
12         close(sockfd);
13         if (errno == EINTR)
14             errno = ETIMEDOUT;
15     }
16     alarm(0);                    /* desativa o alarme */
17     Signal(SIGALRM, sigfunc);    /* restaura o handler de sinal anterior */
18     return (n);
19 }
20 static void
21 connect_alarm(int signo)
22 {
23     return;                      /* apenas interrompe connect() */
24 }
```

lib/connect_timeo.c

Figura 14.1 `connect` com um tempo-limite.

Estabelecimento do handler de sinal

8 Um handler de sinal é estabelecido para `SIGALRM`. O handler de sinal atual (se houver algum) é salvo, assim podemos restaurá-lo no final da função.

Configuração do alarme

9–10 O despertador para o processo é configurado como o número de segundos especificado pelo chamador. O valor de retorno de `alarm` é o número de segundos que atualmente resta no despertador para o processo (se algum valor já houver sido configurado pelo processo). No primeiro caso, imprimimos uma mensagem de alerta visto que estamos apagando esse alarme anteriormente configurado (veja o Exercício 14.2).

Chamada a `connect`

11–15 `connect` é chamada e, se a função for interrompida (`EINTR`), em vez disso, configuraremos o valor de `errno` como `ETIMEDOUT`. O soquete é fechado para impedir que o handshake de três vias prossiga.

Desativação do alarme e restauração de qualquer handler de sinal anterior

16–18 O alarme é desativado configurando-o como 0 e o handler de sinal anterior (se houver algum) é restaurado.

Tratando SIGALRM

20–24 O handler de sinal simplesmente retorna, assumindo que esse retorno irá interromper o `connect` pendente, fazendo com que `connect` retorne um erro de `EINTR`. Lembre-se de que a nossa função `signal` (Figura 5.6) não configura o flag `SA_RESTART` se o sinal sendo capturado for `SIGALRM`.

Uma observação a ser feita com relação a esse exemplo é que sempre podemos reduzir o período do tempo-limite de um `connect` utilizando essa técnica, mas não podemos estender o tempo-limite existente do kernel. Isto é, em um kernel derivado do Berkeley o tempo-limite para um `connect` é normalmente 75 segundos. Podemos especificar um valor menor para nossa função, digamos 10, mas, se especificarmos um valor maior, digamos 80, o próprio `connect` continuará a expirar depois de 75 segundos.

Uma outra observação com relação a esse exemplo é que utilizamos a interruptibilidade da chamada de sistema (`connect`) para retornar antes de o tempo-limite do kernel expirar. Isso é adequado quando realizamos a chamada de sistema e podemos tratar o retorno do erro `EINTR`. Mas, na Seção 29.7, encontraremos uma função de biblioteca que realiza a chamada de sistema e que a reemite quando `EINTR` retorna. Ainda podemos utilizar `SIGALRM` nesse cenário, mas veremos na Figura 29.10 que também temos de utilizar `sigsetjmp` e `siglongjmp` para resolver o problema de a biblioteca ignorar `EINTR`.

Embora esse exemplo seja relativamente simples, os sinais são bem difíceis de utilizar corretamente com programas de múltiplos threads (consulte o Capítulo 26). Portanto, a técnica mostrada aqui é recomendável apenas para programas de um único thread.

`recvfrom` com um tempo-limite que utiliza `SIGALRM`

A Figura 14.2 é uma recriação da nossa função `dg_cli` da Figura 8.8, mas com uma chamada a `alarm` para interromper a `recvfrom` se uma resposta não for recebida dentro de cinco segundos.

advio/dgclitimeo3.c

```
 1 #include   "unp.h"
 2 static void sig_alrm(int);
 3 void
 4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 5 {
 6     int    n;
 7     char   sendline[MAXLINE], recvline[MAXLINE + 1];
 8     Signal(SIGALRM, sig_alrm);
 9     while (Fgets(sendline, MAXLINE, fp) != NULL) {
10         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
11         alarm(5);
12         if ( (n = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL)) < 0) {
13             if (errno == EINTR)
14                 fprintf(stderr, "socket timeout\n");
15             else
16                 err_sys("recvfrom error");
17         } else {
```

Figura 14.2 Função `dg_cli` com `alarm` para expirar `recvfrom` (*continua*).

```
18            alarm(0);
19            recvline[n] = 0;      /* termina com nulo */
20            Fputs(recvline, stdout);
21        }
22    }
23 }
24 static void
25 sig_alrm(int signo)
26 {
27    return;                      /* apenas interrompe o recvfrom() */
28 }
```

advio/dgclitimeo3.c

Figura 14.2 Função `dg_cli` com `alarm` para expirar `recvfrom` (*continuação*).

Tratando o tempo-limite de `recvfrom`

8–22 Estabelecemos um handler de sinal para `SIGALRM` e então chamamos `alarm` para um tempo-limite de cinco segundos antes de cada chamada a `recvfrom`. Se `recvfrom` for interrompido pelo nosso handler de sinal, imprimimos uma mensagem e prosseguimos. Se uma linha for lida a partir do servidor, desativamos o `alarm` pendente e imprimimos a resposta.

handler de sinal `SIGALRM`

24–28 Nosso handler de sinal apenas retorna, para interromper a `recvfrom` bloqueada.

Esse exemplo funciona corretamente porque estamos lendo somente uma resposta toda vez que estabelecemos um `alarm`. Na Seção 20.4, utilizaremos a mesma técnica, mas como estamos lendo múltiplas respostas para um dado `alarm`, há uma condição de corrida que devemos tratar.

`recvfrom` com um tempo-limite utilizando `select`

Demonstramos a segunda técnica para configurar um tempo-limite (utilizando `select`) na Figura 14.3. Ela mostra nossa função identificada como `readable_timeo` que espera até um número especificado de segundos para que um descritor torne-se legível.

lib/readable_timeo.c

```
 1 #include   "unp.h"
 2 int
 3 readable_timeo(int fd, int sec)
 4 {
 5     fd_set rset;
 6     struct timeval tv;
 7     FD_ZERO(&rset);
 8     FD_SET(fd, &rset);
 9     tv.tv_sec = sec;
10     tv.tv_usec = 0;
11     return (select(fd + 1, &rset, NULL, NULL, &tv));
12         /* > 0 se descritor for legível */
13 }
```

lib/readable_timeo.c

Figura 14.3 Função `readable_timeo`: espera que um descritor torne-se legível.

Preparando argumentos para `select`

7–10 O bit correspondente ao descritor é ativado no conjunto de descritores de leitura. Uma estrutura `timeval` é configurada como o número de segundos que o chamador quer esperar.

Bloqueio em `select`

11–12 `select` espera que o descritor torne-se legível ou que o tempo-limite expire. O valor de retorno dessa função é o valor de retorno de `select`: –1 em um erro, 0 se um tempo-limite ocorrer, ou um valor positivo que especifica o número de descritores prontos.

Essa função não realiza a operação de leitura; ela apenas espera que o descritor esteja pronto para ler. Portanto, essa função pode ser utilizada com qualquer tipo de soquete, TCP ou UDP.

É trivial criar uma função semelhante identificada como `writable_timeo` que espera que um descritor torne-se gravável.

Utilizamos essa função na Figura 14.4, que é uma recriação da nossa função `dg_cli` na Figura 8.8. Essa nova versão chama `recvfrom` somente quando nossa função `readable_timeo` retorna um valor positivo.

Não chamamos `recvfrom` até que a função `readable_timeo` informe que o descritor está legível. Isso garante que `recvfrom` não bloqueará.

advio/dgclitimeo1.c

```
 1 #include   "unp.h"

 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int    n;
 6     char   sendline[MAXLINE], recvline[MAXLINE + 1];

 7     while (Fgets(sendline, MAXLINE, fp) != NULL) {

 8         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

 9         if(Readable_timeo(sockfd, 5) == 0) {
10            fprintf(stderr, "socket timeout\n");
11         } else {
12            n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
13            recvline[n] = 0;      /* termina com nulo */
14            Fputs(recvline, stdout);
15         }
16     }
17 }
```

advio/dgclitimeo1.c

Figura 14.4 A função `dg_cli` que chama `readable_timeo` para configurar um tempo-limite.

`recvfrom` com um tempo-limite utilizando a opção de soquete `SO_RCVTIMEO`

Nosso exemplo final ilustra a opção de soquete `SO_RCVTIMEO`. Configuramos essa opção uma vez para um descritor, especificando o valor do tempo-limite e esse tempo-limite é então aplicado a todas as operações de leitura nesse descritor. O melhor aspecto quanto a esse método é que configuramos a opção somente uma vez, comparado com os dois métodos

anteriores que exigiram que algo fosse feito antes de cada operação em que queríamos determinar um limite de tempo. Mas essa opção de soquete se aplica somente a operações de leitura e a opção semelhante `SO_SNDTIMEO` se aplica somente a operações de gravação; nenhuma das opções de soquete pode ser utilizada para configurar um tempo-limite para um `connect`.

A Figura 14.5 mostra uma outra versão da nossa função `dg_cli` que utiliza a opção de soquete `SO_RCVTIMEO`.

Configuração da opção de soquete

8–10 O quarto argumento para `setsockopt` é um ponteiro para uma estrutura `timeval` preenchida com o tempo-limite desejado.

Teste do tempo-limite

15–17 Se a operação de E/S expirar, a função (nesse caso, `recvfrom`) retornará `EWOULDBLOCK`.

advio/dgclitimeo2.c

```
 1 #include   "unp.h"
 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int    n;
 6     char   sendline[MAXLINE], recvline[MAXLINE + 1];
 7     struct timeval tv;
 8     tv.tv_sec = 5;
 9     tv.tv_usec = 0;
10     Setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
11     while (Fgets(sendline, MAXLINE, fp) != NULL) {
12         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
13         n = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
14         if (n < 0) {
15             if (errno == EWOULDBLOCK) {
16                 fprintf(stderr, "socket timeout\n");
17                 continue;
18             } else
19                 err_sys("recvfrom error");
20         }
21         recvline[n] = 0;          /* termina com nulo */
22         Fputs(recvline, stdout);
23     }
24 }
```

advio/dgclitimeo2.c

Figura 14.5 A função `dg_cli` que utiliza a opção de soquete `SO_RCVTIMEO` para configurar um tempo-limite.

14.3 Funções `recv` e `send`

Essas duas funções são semelhantes às funções `read` e `write` padrão, mas um argumento adicional é exigido.

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

ssize_t recv(int sockfd, void *buff, size_t nbytes, int flags);

ssize_t send(int sockfd, const void *buff, size_t nbytes, int flags);
```

As duas retornam: número de bytes lidos ou gravados se OK, –1 no erro

Os três primeiros argumentos para `recv` e `send` são os mesmos três primeiros argumentos para `read` e `write`. O argumento *flags* é 0 ou é formado aplicando um OU lógico a uma ou mais das constantes mostradas na Figura 14.6.

`MSG_DONTROUTE` Esse flag informa ao kernel que o destino está em uma rede localmente anexada e a não realizar uma pesquisa na tabela de roteamento. Fornecemos informações adicionais sobre esse recurso com a opção de soquete `SO_DONTROUTE` (Seção 7.5). Esse recurso pode ser ativado para uma única operação de saída com o flag `MSG_DONTROUTE` ou ativado para todas as operações de saída de um dado soquete utilizando a opção de soquete.

`MSG_DONTWAIT` Esse flag especifica uma única operação de E/S como não-bloqueadora, sem ter de ativar o flag não-bloqueador para o soquete, realiza a operação de E/S e então desativa o flag não-bloqueador. Descreveremos a E/S não-bloqueadora no Capítulo 16, juntamente com a ativação e desativação do flag não-bloqueador para todas as operações de E/S em um soquete.

Esse flag é mais recente do que os outros e talvez não seja suportado por todos os sistemas.

`MSG_OOB` Com `send`, esse flag especifica que dados fora da banda estão sendo enviados. Com TCP, somente um byte deve ser enviado como dados fora da banda, como descreveremos no Capítulo 24. Com `recv`, esse flag especifica que os dados fora da banda devem ser lidos em vez de dados normais.

`MSG_PEEK` Esse flag permite examinar os dados disponíveis para leitura, sem fazer com que o sistema descarte os dados depois que `recv` ou `recvfrom` retorna. Discutiremos outros detalhes disso na Seção 14.7.

`MSG_WAITALL` Esse flag foi introduzido com o 4.3BSD Reno. Ele informa o kernel a não retornar de uma operação `read` até que o número solicitado de bytes tenha sido lido. Se o sistema suportar esse flag, podemos então omitir a função `readn` (Figura 3.15) e substituí-la pela seguinte macro:

```
#define readn(fd, ptr, n) recv(fd, ptr, n, MSG_WAITALL)
```

Mesmo se especificarmos `MSG_WAITALL`, a função ainda pode retornar um número de bytes menor que o solicitado se (i) um sinal for capturado, (ii) a conexão for terminada ou (iii) um erro estiver pendente para o soquete.

flags	Descrição	recv	send
MSG_DONTROUTE	Pule a pesquisa na tabela de roteamento		•
MSG_DONTWAIT	Somente essa operação é não-bloqueadora	•	•
MSG_OOB	Envie ou receba dados fora da banda	•	•
MSG_PEEK	Examine uma mensagem entrante	•	
MSG_WAITALL	Espere todos os dados	•	

Figura 14.6 *flags* para funções de E/S.

Há flags adicionais utilizados por outros protocolos, mas não com o TCP/IP. Por exemplo, a camada de transporte OSI é baseada em registro (não em um fluxo de bytes como o TCP) e suporta o flag `MSG_EOR` em operações de saída para especificar o final de um registro lógico.

Há um problema fundamental de *design* com o argumento *flags*: ele é passado por valor; não é um argumento de valor-resultado. Portanto, esse argumento pode ser utilizado somente para passar flags do processo para o kernel. O kernel não pode passar flags de volta para o processo. Esse não é um problema com o TCP/IP, porque é raro haver necessidade de passar flags de volta para o processo a partir do kernel. Mas quando os protocolos OSI foram adicionados ao 4.3BSD Reno, surgiu a necessidade de retornar `MSG_EOR` ao processo com uma operação de entrada. Portanto, a decisão foi tomada com o 4.3BSD Reno para deixar como estão os argumentos para as funções de entrada comumente utilizadas (`recv` e `recvfrom`) e para alterar a estrutura `msghdr` utilizada com `recvmsg` e `sendmsg`. Veremos na Seção 14.5 que um membro inteiro `msg_flags` foi adicionado a essa estrutura e, como esta é passada por referência, o kernel pode modificar esses flags no retorno. Isso também significa que, se um processo precisar que os flags sejam atualizados pelo kernel, deverá chamar `recvmsg` em vez de `recv` ou `recvfrom`.

14.4 Funções `readv` e `writev`

Essas duas funções são semelhantes a `read` e `write`, mas `readv` e `writev` permitem ler ou gravar a partir de um ou mais buffers com uma única chamada de função. Essas operações são chamadas de *leitura dispersa (scatter read)* (uma vez que os dados de entrada estão dispersos em múltiplos buffers de aplicação) e *gravação agrupada (gather write)* (uma vez que múltiplos buffers estão agrupados em uma única operação de saída).

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

ssize_t readv(int filedes, const struct iovec *iov, int iovcnt);

ssize_t writev(int filedes, const struct iovec *iov, int iovcnt);
```

As duas retornam: o número de bytes lidos ou gravados, –1 em erro

O segundo argumento para as duas funções é um ponteiro para um array das estruturas `iovec`, que é definido incluindo o cabeçalho `<sys/uio.h>`.

```
struct iovec {
   void *iov_base;    /* endereço inicial do buffer */
   size_t iov_len;    /* tamanho do buffer */
};
```

> Os tipos de dados mostrados para os membros da estrutura `iovec` são aqueles especificados pelo POSIX. Você pode encontrar implementações que definem `iov_base` como um `char *` e `iov_len` como um `int`.

Há algum limite quanto ao número de elementos no array das estruturas `iovec` que uma implementação permite. O Linux, por exemplo, permite até 1.024, enquanto o HP-UX tem um limite de 2.100. O POSIX exige que a constante `IOV_MAX` seja definida incluindo o cabeçalho `<sys/uio.h>` e que seu valor seja pelo menos 16.

As funções `readv` e `writev` podem ser utilizadas com qualquer descritor, não apenas com soquetes. Além disso, `writev` é uma operação atômica. Para um protocolo baseado em registro, como o UDP, uma chamada a `writev` gera um único datagrama UDP.

Mencionamos uma das utilizações de `writev` com a opção de soquete `TCP_NODELAY` na Seção 7.9. Dissemos que um `write` de 4 bytes seguido por um de 396 bytes poderia invocar o algoritmo de Nagle e uma solução adequada é chamar `writev` para os dois buffers.

14.5 Funções recvmsg e sendmsg

Essas duas funções são as mais gerais entre todas as funções de E/S. De fato, poderíamos substituir todas as chamadas a read, readv, recv e recvfrom por chamadas a recvmsg. De maneira semelhante, todas as chamadas a várias funções de saída poderiam ser substituídas por chamadas a sendmsg.

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

ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);

ssize_t sendmsg(int sockfd, struct msghdr *msg, int flags);
```

As duas retornam: número de bytes lidos ou gravados se OK, –1 no erro

As duas funções empacotam a maioria dos argumentos em uma estrutura msghdr.

```
struct msghdr {
    void         *msg_name;       /* endereço do protocolo */
    socklen_t     msg_namelen;    /* tamanho do endereço de protocolo */
    struct iovec *msg_iov;        /* array de dispersão/agrupamento */
    int           msg_iovlen;     /* nº de elementos em msg_iov */
    void         *msg_control;    /* dados auxiliares (estrutura cmsghdr) */
    socklen_t     msg_controllen; /* comprimento dos dados auxiliares */
    int           msg_flags;      /* flags retornados por recvmsg() */
};
```

A estrutura msghdr que mostramos é a especificada no POSIX. Alguns sistemas ainda utilizam uma estrutura msghdr mais antiga que se originou com o 4.2BSD. Essa estrutura mais antiga não tem o membro msg_flags e os membros msg_control e msg_controllen são identificados como msg_accrights e msg_accrightslen. A forma mais recente da estrutura msghdr geralmente está disponível utilizando flags de compilação condicional. A única forma de dados auxiliares suportada pela estrutura mais antiga é a passagem de descritores de arquivo (denominados direitos de acesso).

Os membros msg_name e msg_namelen são utilizados quando o soquete não está conectado (por exemplo, um soquete UDP não-conectado). Eles são semelhantes ao quinto e sexto argumentos para recvfrom e sendto: msg_name aponta para uma estrutura do endereço de soquete em que o chamador armazena o endereço de protocolo do destino para sendmsg ou em que recvmsg armazena o endereço de protocolo do emissor. Se um endereço de protocolo não precisa ser especificado (por exemplo, um soquete TCP ou UDP conectado), msg_name deverá ser configurado como um ponteiro nulo. msg_namelen é um valor para sendmsg, mas um valor-resultado para recvmsg.

Os membros msg_iov e msg_iovlen especificam o array de buffers de entrada ou de saída (o array de estruturas iovec), semelhantes ao segundo e terceiro argumentos para readv ou writev. Os membros msg_control e msg_controllen especificam a localização e o tamanho dos dados auxiliares opcionais. msg_controllen é um argumento de valor-resultado para recvmsg. Descreveremos os dados auxiliares na Seção 14.6.

Com recvmsg e sendmsg, precisamos distinguir entre duas variáveis de flag: o argumento *flags*, que é passado por valor, e o membro msg_flags da estrutura msghdr, que é passado por referência (visto que o endereço da estrutura é passado para a função).

- O membro msg_flags é utilizado somente por recvmsg. Quando recvmsg é chamado, o argumento *flags* é copiado para o membro msg_flags (página 502 do TCPv2) e esse valor é utilizado pelo kernel para orientar seu processamento de recepção. Esse valor é então atualizado com base no resultado de recvmsg.

- O membro `msg_flags` é ignorado por `sendmsg` porque essa função utiliza o argumento *flags* para orientar seu processamento de saída. Isso significa que, se quisermos configurar o flag `MSG_DONTWAIT` em uma chamada a `sendmsg`, iremos configurar o argumento *flags* como esse valor; configurar o membro `msg_flags` como esse valor não tem nenhum efeito.

A Figura 14.7 resume os flags que são examinados pelo kernel tanto para as funções de entrada como de saída, bem como o `msg_flags` que poderia ser retornado por `recvmsg`. Não há nenhuma coluna para `sendmsg` `msg_flags` porque, como mencionamos, ele não é utilizado.

Os primeiros quatro flags são apenas examinados e nunca são retornados; os dois seguintes são examinados e retornados; e os últimos quatro são apenas retornados. Os comentários a seguir se aplicam aos seis flags retornados por `recvmsg`:

`MSG_BCAST` Esse flag é relativamente recente, suportado ao menos pelo BSD, e é retornado se o datagrama tiver sido recebido como um broadcast de camada de enlace ou com um endereço IP de destino que seja um endereço de broadcast. Esse flag é uma maneira melhor de determinar se um datagrama UDP foi enviado a um endereço de broadcast, se comparado como a opção de soquete `IP_RECVDSTADDR`.

`MSG_MCAST` Esse flag também é uma adição relativamente recente, suportado ao menos pelo BSD, e é retornado se o datagrama tiver sido recebido como um multicast da camada de enlace.

`MSG_TRUNC` Esse flag é retornado se o datagrama foi truncado; em outras palavras, o kernel tem mais dados a retornar do que o processo alocou espaço para receber (a soma de todos os membros `iov_len`). Discutiremos isso em mais detalhes na Seção 22.3.

`MSG_CTRUNC` Esse flag é retornado se os dados auxiliares foram truncados; em outras palavras, o kernel tem mais dados auxiliares a retornar do que o processo alocou espaço para receber (`msg_controllen`).

`MSG_EOR` Esse flag é limpo se os dados retornados não terminarem um registro lógico; o flag é ativado se os dados retornados terminarem um registro lógico. O TCP *não* utiliza esse flag, uma vez que é um protocolo de fluxo de bytes.

Flag	Examinado por: `send` *flags* `sendto` *flags* `sendmsg` *flags*	Examinado por: `recv` *flags* `recvfrom` *flags* `recvmsg` *flags*	Retornado por: `recvmsg msg_flags`
MSG_DONTROUTE	•		
MSG_DONTWAIT	•	•	
MSG_PEEK		•	
MSG_WAITALL		•	
MSG_EOR	•		•
MSG_OOB	•	•	•
MSG_BCAST			•
MSG_MCAST			•
MSG_TRUNC			•
MSG_CTRUNC			•
MSG_NOTIFICATION			•

Figura 14.7 Resumo dos flags de entrada e de saída por várias funções de E/S.

`MSG_OOB` Esse flag *nunca* é retornado para dados fora da banda do TCP. Ele é retornado por outros conjuntos de protocolos (por exemplo, os protocolos OSI).

`MSG_NOTIFICATON` Esse flag é retornado aos receptores SCTP para indicar que a leitura da mensagem é uma notificação de evento, não uma mensagem de dados.

Implementações podem retornar alguns dos *flags* de entrada no membro `msg_flags`, portanto, devemos examinar somente os valores de flag em que estamos interessados (por exemplo, os últimos seis na Figura 14.7).

A Figura 14.8 mostra uma estrutura `msghdr` e as várias informações para as quais ela aponta. Supomos nessa figura que o processo está em vias de chamar `recvmsg` para um soquete UDP.

Dezesseis bytes são alocados ao endereço de protocolo e 20 bytes são alocados aos dados auxiliares. Um array de três estruturas `iovec` é inicializado: a primeira especifica um buffer de 100 bytes, a segunda um buffer de 60 bytes e a terceira um buffer de 80 bytes. Também supomos que a opção de soquete `IP_RECVDSTADDR` foi configurada para o soquete, para receber o endereço IP de destino proveniente do datagrama UDP.

Em seguida, supomos que um datagrama UDP de 170 bytes chega de 192.6.38.100, porta 2000, destinado ao nosso soquete UDP com um endereço IP de destino de 206.168.112.96. A Figura 14.9 mostra todas as informações na estrutura `msghdr` quando `recvmsg` retorna.

Os campos sombreados são modificados por `recvmsg`. Os itens a seguir foram alterados da Figura 14.8 para a 14.9:

- O buffer apontado por `msg_name` foi preenchido como uma estrutura de endereço de soquete de Internet, contendo o endereço IP de origem e a porta UDP de origem provenientes do datagrama recebido.
- `msg_namelen`, um argumento de valor-resultado, é atualizado de acordo com o volume de dados armazenado em `msg_name`. Nada muda desde que seu valor antes da chamada era 16 e seu valor quando `recvmsg` retorna também é 16.
- Os primeiros 100 bytes de dados são armazenados no primeiro buffer; os próximos 60 bytes são armazenados no segundo buffer; e os 10 bytes finais são armazenados no tercei-

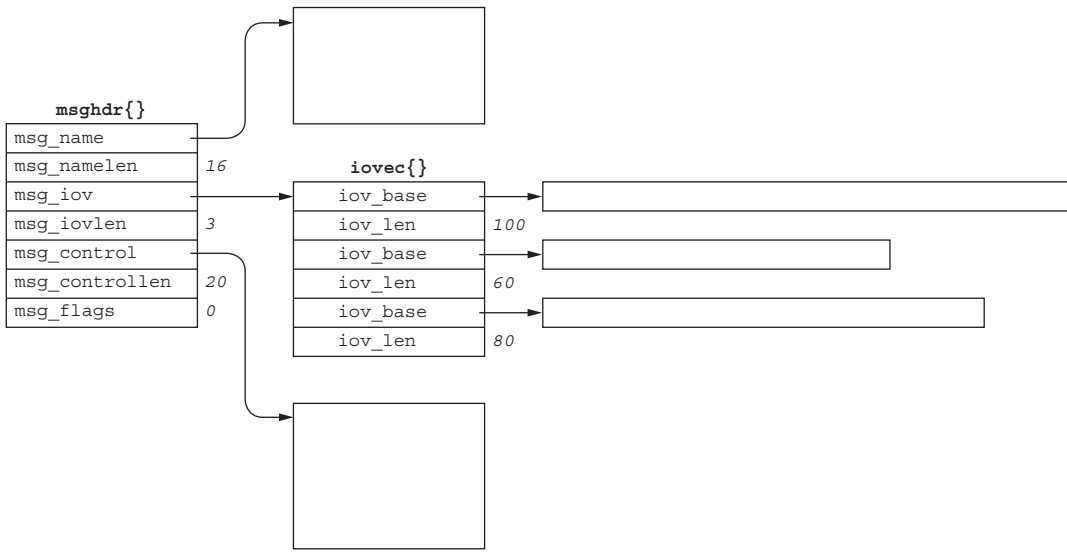


Figura 14.8 Estruturas de dados quando `recvmsg` é chamado para um soquete UDP.

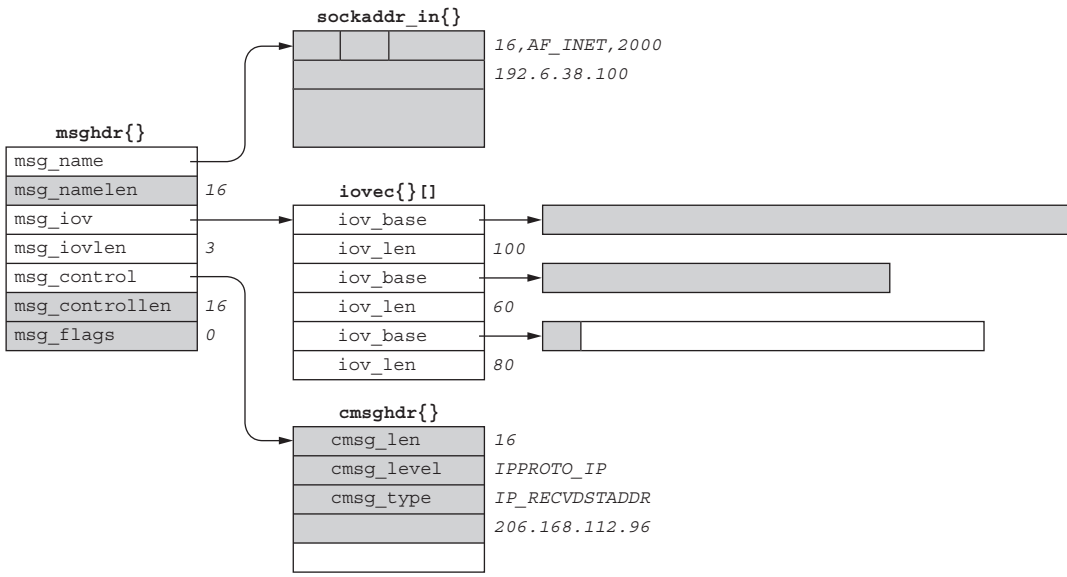


Figura 14.9 Atualização da Figura 14.8 quando `recvmsg` retorna.

ro buffer. Os últimos 70 bytes do buffer final não são modificados. O valor de retorno da função `recvmsg` é o tamanho do datagrama, 170.

- O buffer apontado por `msg_control` é preenchido como uma estrutura `cmsghdr`. (Discutiremos os dados auxiliares em mais detalhes na Seção 14.6 e também essa opção particular de soquete na Seção 22.2.) O `cmsg_len` é 16; o `cmsg_level` é `IPPROTO_IP`; o `cmsg_type` é `IP_RECVDSTADDR`; os próximos 4 bytes contêm o endereço IP de destino do datagrama UDP recebido. Os 4 bytes finais do buffer de 20 bytes que fornecemos para armazenar os dados auxiliares não são modificados.
- O membro `msg_controllen` é atualizado de acordo com o volume real de dados auxiliares que foi armazenado. Ele também é um argumento de valor-resultado e seu resultado no retorno é 16.
- O membro `msg_flags` é atualizado por `recvmsg`, mas não há nenhum flag para retornar ao processo.

A Figura 14.10 resume as diferenças entre os cinco grupos de funções de E/S que descrevemos.

Função	Qualquer descritor	Somente descritor de soquete	Buffer de leitura/ gravação único	Leitura/ gravação dispersa/ agrupada	Flags opcionais	Endereço do peer opcional	Informações de controle opcionais
`read, write`	•		•				
`readv, writev`	•			•			
`recv, send`		•	•		•		
`recvfrom, sendto`		•	•		•	•	
`recvmsg, sendmsg`		•		•	•	•	•

Figura 14.10 Comparação entre os cinco grupos de funções E/S.

14.6 Dados auxiliares

Os dados auxiliares podem ser enviados e recebidos utilizando os membros `msg_control` e `msg_controllen` da estrutura `msghdr` com as funções `sendmsg` e `recvmsg`. Outro termo para os dados auxiliares é *informações de controle*. Nesta seção, descreveremos o conceito e mostraremos a estrutura e as macros utilizadas para construir e processar dados auxiliares, mas deixaremos os exemplos de código para os capítulos posteriores que descrevem as utilizações reais dos dados auxiliares.

A Figura 14.11 é um resumo das várias utilizações dos dados auxiliares que abordamos neste texto.

O conjunto de protocolos OSI também utiliza dados auxiliares para vários propósitos que não discutimos no texto.

Os dados auxiliares consistem em um ou mais *objetos de dados auxiliares*, cada qual começando com uma estrutura `cmsghdr`, definida incluindo `<sys/socket.h>`.

```
struct cmsghdr {
 socklen_t   cmsg_len;     /* comprimento em bytes, incluindo essa estrutura */
 int         cmsg_level;   /* protocolo originante */
 int         cmsg_type;    /* tipo específico do protocolo */
   /* seguido por unsigned char cmsg_data[] */
};
```

Já vimos essa estrutura na Figura 14.9, quando foi utilizada com a opção de soquete `IP_RECVDSTADDR` para retornar o endereço IP de destino de um datagrama UDP recebido. Os dados auxiliares apontados por `msg_control` devem ser adequadamente alinhados para uma estrutura `cmsghdr`. Mostraremos uma das maneiras de fazer isso na Figura 15.11.

A Figura 14.12 mostra um exemplo de dois objetos de dados auxiliares no buffer de controle.

`msg_control` aponta para o primeiro objeto de dados auxiliares e o comprimento total dos dados auxiliares é especificado por `msg_controllen`. Cada objeto é precedido por uma estrutura `cmsghdr` que descreve o objeto. Pode haver um preenchimento (*padding*) entre o membro `cmsg_type` e os dados reais e, também, um preenchimento no final dos dados, antes do próximo objeto de dados auxiliares. As cinco macros `CMSG_` *xxx* que descrevemos brevemente explicam esse possível preenchimento.

> Nem todas as implementações suportam múltiplos objetos de dados auxiliares no buffer de controle.

Protocolo	cmsg_level	cmsg_type	Descrição
IPv4	IPPROTO_IP	IP_RECVDSTADDR	Recebe o endereço de destino com datagrama UDP
		IP_RECVIF	Recebe o índice de interface com datagrama UDP
IPv6	IPPROTO_IPV6	IPV6_DSTOPTS	Especifica opções de destino
		IPV6_HOPLIMIT	Especifica o limite de hop
		IPV6_HOPOPTS	Especifica as opções hop por hop
		IPV6_NEXTHOP	Especifica o endereço do próximo hop
		IPV6_PKTINFO	Especifica as informações de pacote
		IPV6_RTHDR	Especifica o cabeçalho de roteamento
		IPV6_TCLASS	Especifica a classe de tráfego
Domínio Unix	SOL_SOCKET	SCM_RIGHTS	Descritores de envio/recebimento
		SCM_CREDS	Credenciais do usuário de envio/recebimento

Figura 14.11 Resumo das utilizações de dados auxiliares.

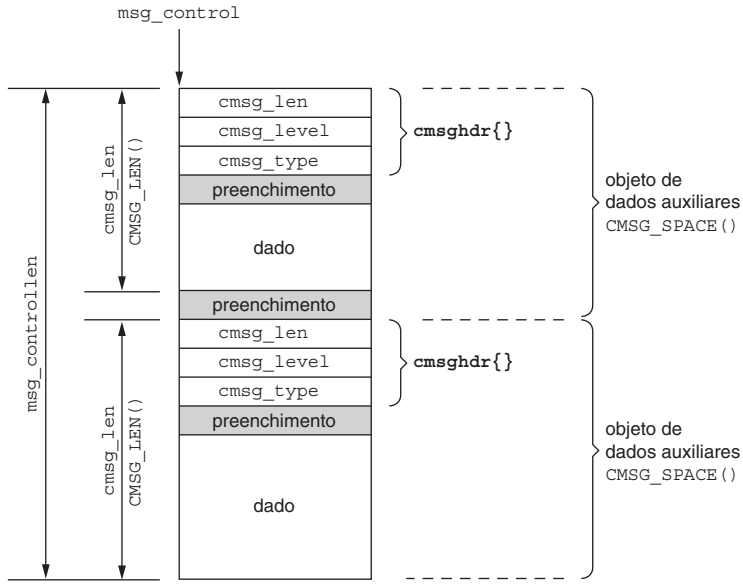


Figura 14.12 Dados auxiliares contendo dois objetos de dados auxiliares.

A Figura 14.13 mostra o formato da estrutura `cmsghdr` quando utilizada com um soquete de domínio Unix para passagem de descritor (Seção 15.7) ou de credencial (Seção 15.8).

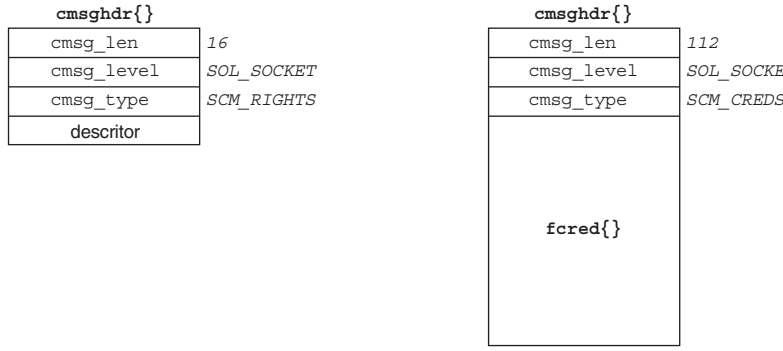


Figura 14.13 A estrutura `cmsghdr` quando utilizada com soquetes de domínio Unix.

Nessa figura, supomos que cada um dos três membros da estrutura `cmsghdr` ocupa 4 bytes e não há nenhum preenchimento entre a estrutura `cmsghdr` e os dados reais. Quando os descritores são passados, o conteúdo do array `cmsg_data` é o valor real do descritor. Nessa figura, mostramos somente um dos descritores sendo passado, mas, em geral, mais de um pode ser passado (nesse caso, o valor de `cmsg_len` será 12 mais 4 vezes o número de descritores, assumindo que cada descritor ocupa 4 bytes).

Como os dados auxiliares retornados por `recvmsg` podem conter um número qualquer de objetos de dados auxiliares, e para ocultar o possível preenchimento por parte da aplicação, as cinco macros a seguir são definidas, incluindo o cabeçalho `<sys/socket.h>`, para simplificar o processamento dos dados auxiliares:

```
#include <sys/socket.h>
#include <sys/param.h>  /* para a macro ALIGN em muitas implementações */

struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *mhdrptr);
```

Retorna: ponteiro para a primeira estrutura de cmsghdr ou NULL se nenhum dado auxiliar

```
struct cmsghdr *CMSG_NXTHDR(struct msghdr *mhdrptr, struct cmsghdr *cmsgptr);
```

Retorna: ponteiro para a próxima estrutura de cmsghdr ou NULL se nenhum outro objeto de dados auxiliares

```
unsigned char *CMSG_DATA(struct cmsghdr *cmsgptr);
```

Retorna: ponteiro para o primeiro byte de dados associado com estrutura cmsghdr

```
unsigned int CMSG_LEN(unsigned int length);
```

Retorna: valor a ser armazenado em cmsg_len dada a quantidade de dados

```
unsigned int CMSG_SPACE(unsigned int length);
```

Retorna: tamanho total de um objeto de dados auxiliares dada a quantidade de dados

O POSIX define as três primeiras macros; a RFC 3542 (de Stevens *et al*., 2003), as últimas duas.

Essas macros seriam utilizadas no pseudocódigo a seguir:

```
struct msghdr  msg;
struct cmsghdr *cmsgptr;

/* preenche a estrutura msg */

/* chama recvmsg() */

for (cmsgptr = CMSG_FIRSTHDR(&msg); cmsgptr != NULL;
     cmsgptr = CMSG_NXTHDR(&msg, cmsgptr)) {
   if (cmsgptr->cmsg_level == ... &&
       cmsgptr->cmsg_type == ... ) {
       u_char *ptr;

       ptr = CMSG_DATA(cmsgptr);
       /* processa os dados apontados por ptr */
   }
}
```

CMSG_FIRSTHDR retorna um ponteiro para o primeiro objeto de dados auxiliares ou para um ponteiro nulo se não houver nenhum dado auxiliar na estrutura msghdr (msg_control é um ponteiro nulo ou cmsg_len é menor que o tamanho de uma estrutura cmsghdr). CMSG_NXTHDR retorna um ponteiro nulo quando não há nenhum outro objeto de dados auxiliares no buffer de controle.

Muitas implementações existentes de CMSG_FIRSTHDR nunca examinam msg_controllen e apenas retornam o valor de cmsg_control. Na Figura 22.2, testaremos o valor de msg_controllen antes de chamar essa macro.

A diferença entre CMSG_LEN e CMSG_SPACE é que o primeiro não considera nenhum preenchimento depois da parte dos dados no objeto de dados auxiliares e é, portanto, o valor a armazenar em cmsg_len, enquanto o último considera o preenchimento no final e é, portanto, o valor a utilizar se o espaço for dinamicamente alocado ao objeto de dados auxiliares.

14.7 Quantos dados estão enfileirados?

Há momentos em que queremos ver quantos dados estão enfileirados para serem lidos em um soquete, sem os ler. Há três técnicas disponíveis para isso:

1. Se o objetivo não for bloquear no kernel porque temos algo mais a fazer quando nada está pronto para ser lido, a E/S não-bloqueadora poderá ser utilizada. Descreveremos isso no Capítulo 16.
2. Se quisermos examinar os dados e ainda deixá-los na fila de recebimento para que alguma outra parte do nosso processo leia, podemos utilizar o flag `MSG_PEEK` (Figura 14.6). Se quisermos fazer isso, mas não temos certeza de que algo está pronto para ser lido, podemos utilizar esse flag com um soquete não-bloqueador ou então combiná-lo com o flag `MSG_DONTWAIT`.

   Esteja ciente de que a quantidade de dados na fila de recebimento pode mudar entre duas chamadas sucessivas a `recv` em um soquete de fluxo. Por exemplo, suponha que chamamos `recv` para um soquete TCP especificando o comprimento de um buffer de 1.024 juntamente com o flag `MSG_PEEK` e que o valor de retorno seja 100. Se chamarmos `recv` novamente, é possível que mais de 100 bytes retornem (supondo que especificamos um comprimento de buffer maior que 100), porque mais dados podem ser recebidos pelo TCP entre nossas duas chamadas.

   No caso de um soquete UDP com um datagrama na fila de recebimento, se chamarmos `recvfrom` especificando `MSG_PEEK`, seguido por uma outra chamada sem especificar `MSG_PEEK`, os valores de retorno das duas chamadas (o tamanho do datagrama, seu conteúdo e o endereço do emissor) serão idênticos, mesmo se outros datagramas forem adicionados ao buffer de recebimento do soquete entre elas. (Estamos supondo, naturalmente, que outro processo qualquer não está compartilhando o mesmo descritor e lendo a partir desse soquete ao mesmo tempo.)
3. Algumas implementações suportam o comando `FIONREAD` de `ioctl`. O terceiro argumento para `ioctl` é um ponteiro para um inteiro; e o valor retornado nesse inteiro é o número atual de bytes na fila de recebimento do soquete (página 553 do TCPv2). Esse valor é o número total de bytes enfileirados que, para um soquete UDP, inclui todos os datagramas enfileirados. Também esteja ciente de que a contagem retornada para um soquete UDP por implementações derivadas do Berkeley inclui o espaço requerido para a estrutura de endereço de soquete que contém o endereço e a porta IP do emissor para cada datagrama (16 bytes para IPv4; 24 bytes para IPv6).

14.8 Soquetes e E/S-padrão

Em todos os nossos exemplos até agora, utilizamos o que às vezes é chamado de *E/S do Unix*, as funções `read` e `write` e suas variantes (`recv`, `send`, etc.). Essas funções funcionam com *descritores* e normalmente são implementadas como chamadas de sistema dentro do kernel do Unix.

Outro método de realizar a E/S é a *biblioteca de E/S-padrão.* Essa biblioteca é especificada pelo padrão ANSI C e é concebida para ser portável em sistemas não-Unix que suportam o ANSI C. A biblioteca de E/S-padrão trata alguns detalhes com os quais devemos nos preocupar ao utilizar as funções de E/S do Unix, como armazenar automaticamente em buffer os fluxos de entrada e saída. Infelizmente, seu tratamento do armazenamento em buffer de um fluxo pode apresentar um novo conjunto de problemas com os quais devemos nos preocupar. O Capítulo 5 do APUE discute a biblioteca de E/S-padrão em detalhes, e Plauger (1992) apresenta e discute uma implementação completa da biblioteca de E/S-padrão.

> O termo *fluxo* (*stream*) é utilizado com a biblioteca de E/S-padrão, como em "abrimos um fluxo de entrada" ou "esvaziamos o fluxo de saída". Não confunda esse termo com o subsistema STREAMS, que discutiremos no Capítulo 31.

A biblioteca de E/S-padrão pode ser utilizada com soquetes, mas há alguns itens a considerar:

- Um fluxo de E/S-padrão pode ser criado a partir de qualquer descritor chamando a função `fdopen`. De maneira semelhante, dado um fluxo de E/S-padrão, podemos obter o descritor correspondente chamando `fileno`. Nosso primeiro encontro com `fileno` foi na Figura 6.9, quando queríamos chamar `select` em um fluxo de E/S-padrão. `select` funciona somente com descritores, assim tivemos de obter o descritor para o fluxo de E/S-padrão.
- Soquetes TCP e UDP são full duplex. Fluxos de E/S-padrão também podem ser full duplex: simplesmente abrimos o fluxo com um tipo de `r+`, o que significa read-write (leitura e gravação). Mas, nesse fluxo, uma função de saída não pode ser seguida por uma função de entrada sem uma chamada interposta a `fflush`, `fseek`, `fsetpos` ou `rewind`. De maneira semelhante, uma função de entrada não pode ser seguida por uma função de saída sem uma chamada interposta a `fseek`, `fsetpos` ou `rewind`, a menos que a função de entrada encontre um EOF. O problema com essas três últimas funções é que todas chamam `lseek`, que falha em soquete.
- A maneira mais fácil de tratar esse problema de leitura e gravação é abrir dois fluxos de E/S-padrão para um dado soquete: um para leitura e outro para gravação.

Exemplo: Função `str_echo` utilizando E/S-padrão

Agora, mostraremos uma versão alternativa do nosso servidor de eco TCP (Figura 5.3), que utiliza a E/S-padrão em vez de `read` e `writen`. A Figura 14.14 é uma versão da nossa função `str_echo` que utiliza a E/S-padrão. (Essa versão tem um problema que descreveremos em breve.)

advio/str_echo_stdio02.c

```
 1 #include "unp.h"
 2 void
 3 str_echo(int sockfd)
 4 {
 5     char   line[MAXLINE];
 6     FILE   *fpin, *fpout;
 7     fpin = Fdopen(sockfd, "r");
 8     fpout = Fdopen(sockfd, "w");
 9     while (Fgets(line, MAXLINE, fpin) != NULL)
10         Fputs(line, fpout);
11 }
```

advio/str_echo_stdio02.c

Figura 14.14 Função `str_echo` recodificada para utilizar a E/S-padrão.

Conversão do descritor em fluxo de entrada e de saída

7–10 Dois fluxos de E/S-padrão são criados por `fdopen`: um para entrada e outro para saída. As chamadas a `read` e `writen` são substituídas por chamadas a `fgets` e `fputs`.

Se executarmos nosso servidor com essa versão de `str_echo` e então executarmos nosso cliente, veremos o seguinte:

```
hpux % tcpcli02 206.168.112.96
hello, world                    digitamos essa linha, mas nada é ecoado
and hi                          e essa, ainda nenhum eco
hello??                         e essa, ainda nenhum eco
^D                              e nosso caractere de EOF
hello, world                    e então as três linhas ecoadas são enviadas para a saída
and hi
hello??
```

Aqui, há um problema no armazenamento em buffer porque nada é ecoado pelo servidor até inserirmos nosso caractere de EOF. Ocorrem os seguintes passos:

- Digitamos a primeira linha de entrada e ela é enviada ao servidor.
- O servidor lê a linha com `fgets` e ecoa com `fputs`.
- O fluxo de E/S-padrão do servidor é *completamente armazenado em buffer* pela biblioteca de E/S-padrão. Isso significa que a biblioteca copia a linha ecoada para seu buffer de E/S-padrão para esse fluxo, mas não grava o buffer no descritor, porque o buffer não está cheio.
- Digitamos a segunda linha de entrada e ela é enviada ao servidor.
- O servidor lê a linha com `fgets` e ecoa com `fputs`.
- Novamente, a biblioteca de E/S-padrão do servidor somente copia a linha para seu buffer, mas não grava o buffer porque ainda não está cheio.
- O mesmo cenário acontece com a terceira linha de entrada que inserimos.
- Digitamos nosso caractere de EOF e nossa função `str_cli` (Figura 6.13) chama `shutdown`, enviando um FIN ao servidor.
- O TCP do servidor recebe o FIN, que `fgets` lê, fazendo com que `fgets` retorne um ponteiro nulo.
- A função `str_echo` retorna a função `main` do servidor (Figura 5.12) e o filho termina chamando `exit`.
- A função `exit` da biblioteca C chama a função de limpeza de E/S-padrão (páginas 162 a 164 do APUE). O buffer de saída que foi parcialmente preenchido pelas nossas chamadas a `fputs` é agora enviado para a saída.
- O processo servidor-filho termina, fazendo com que seu soquete conectado seja fechado, enviando um FIN ao cliente, completando a seqüência de término de quatro pacotes TCP.
- As três linhas ecoadas são recebidas pela nossa função `str_cli` e enviadas para a saída.
- `str_cli` recebe então um EOF no seu soquete e o cliente termina.

O problema aqui é o armazenamento em buffer realizado automaticamente pela biblioteca de E/S-padrão no servidor. Há três tipos de armazenamentos em buffer realizados pela biblioteca de E/S-padrão:

1. *Buffer completo (fully buffered)* significa que a E/S acontece somente quando o buffer está cheio, o processo chama explicitamente `fflush` ou termina chamando `exit`. Um tamanho comum para o buffer de E/S-padrão é 8.192 bytes.
2. *Buffer de linha (line buffered)* significa que a E/S acontece quando uma nova linha é encontrada, quando o processo chama `fflush` ou termina chamando `exit`.
3. *Sem buffer (unbuffered)* significa que a E/S acontece toda vez que uma função de saída de E/S-padrão é chamada.

A maioria das implementações Unix da biblioteca de E/S-padrão utiliza as seguintes regras:

- Erros-padrão são sempre sem buffer.

- A entrada e a saída-padrão são do tipo buffer completo, a menos que referenciem um dispositivo terminal em que sejam do tipo buffer de linha.
- Todos os outros fluxos são do tipo buffer completo, a menos que referenciem um dispositivo terminal em que sejam do tipo buffer de linha.

Como um soquete não é um dispositivo terminal, o problema visto com a nossa função `str_echo` na Figura 14.14 é que o fluxo de saída (`fpout`) é do tipo buffer completo. Uma maneira de contornar isso é forçar o fluxo de saída a ser do tipo buffer de linha chamando `setvbuf`. Uma outra maneira é forçar cada linha ecoada a ser enviada para a saída chamando `fflush` depois de cada chamada a `fputs`. Mas, na prática, qualquer uma dessas soluções ainda está propensa a erros e pode interagir mal com o algoritmo de Nagle descrito na Seção 7.9. Na maioria dos casos, a melhor solução é simplesmente não utilizar a biblioteca de E/S-padrão para soquetes e operar em buffers em vez de em linhas, como descrito na Seção 3.9. Utilizar a E/S-padrão para soquetes pode fazer sentido quando a conveniência dos fluxos de E/S-padrão excede as preocupações quanto a bugs devido ao armazenamento em buffer, mas tais casos são raros.

> Esteja ciente de que algumas implementações da biblioteca de E/S-padrão ainda têm problema com descritores maiores que 255. Isso pode ser um problema com servidores de rede que tratam uma grande quantidade de descritores. Verifique a definição da estrutura `FILE` no seu cabeçalho `<stdio.h>` para ver que tipo de variável armazena o descritor.

14.9 Polling avançado

Anteriormente neste capítulo, discutimos várias maneiras de configurar um limite de tempo em uma operação de soquete. Agora, vários sistemas operacionais oferecem uma outra alternativa e fornecem os recursos de `select` e `poll` também descritos no Capítulo 6. Como nenhum desses métodos ainda foi adotado pelo POSIX e cada implementação parece ser levemente diferente, o código que utiliza esses mecanismos deve ser considerado não-portável. Descreveremos dois mecanismos aqui; outros mecanismos disponíveis são semelhantes.

Interface `/dev/poll`

O Solaris fornece um arquivo especial chamado `/dev/poll`, que permite uma maneira mais escalonável de fazer *polling* (ou consulta seqüencial) de um grande número de descritores de arquivo. O problema com `select` e `poll` é que os descritores de arquivo de interesse devem ser passados para cada chamada. O dispositivo de polling mantém um estado entre chamadas de modo que um programa possa configurar a lista de descritores a consultar e então fazer um loop, esperando por eventos, sem configurar a lista novamente a cada passagem pelo loop.

Depois de abrir `/dev/poll`, o programa de polling deve inicializar um array de estruturas `pollfd` (a mesma estrutura utilizada pela função `poll`, mas o campo `revents` não é utilizado nesse caso). O array é então passado para o kernel chamando `write` para gravar a estrutura diretamente no dispositivo `/dev/poll`. O programa então utiliza uma chamada `ioctl`, `DP_POLL`, para bloquear e esperar os eventos. A estrutura a seguir é passada para a chamada `ioctl`:

```
struct dvpoll {
   struct pollfd* dp_fds;
   int            dp_nfds;
   int            dp_timeout;
}
```

O campo `dp_fds` aponta para um buffer que é utilizado para armazenar um array de estruturas `pollfd` retornadas da chamada `ioctl`. O campo `dp_nfds` especifica o tamanho do buffer. A chamada `ioctl` é bloqueada até que haja eventos interessantes em qualquer um dos descritores de arquivo de polling ou até que `dp_timeout` milissegundos tenham se passado. Utilizar o valor zero para `dp_timeout` fará com que a chamada `ioctl` retorne ime-

diatamente, o que fornece uma maneira não-bloqueadora de utilizar essa interface. Passar `-1` para o tempo-limite indica que não se deseja um tempo-limite.

Modificamos nossa função `str_cli`, que utilizou `select` na Figura 6.13, para utilizar `/dev/poll` na Figura 14.15.

advio/str_cli_poll03.c

```
 1 #include   "unp.h"
 2 #include   <sys/devpoll.h>
 3 void
 4 str_cli(FILE *fp, int sockfd)
 5 {
 6     int    stdineof;
 7     char   buf[MAXLINE];
 8     int    n;
 9     int    wfd;
10     struct pollfd pollfd[2];
11     struct dvpoll dopoll;
12     int    i;
13     int    result;
14     wfd = Open("/dev/poll", O_RDWR, 0);
15     pollfd[0].fd = fileno(fp);
16     pollfd[0].events = POLLIN;
17     pollfd[0].revents = 0;
18     pollfd[1].fd = sockfd;
19     pollfd[1].events = POLLIN;
20     pollfd[1].revents = 0;
21     Write(wfd, pollfd, sizeof(struct pollfd) * 2);
22     stdineof = 0;
23     for ( ; ; ) {
24         /* bloqueia até que /dev/poll informe que algo está pronto */
25         dopoll.dp_timeout = -1;
26         dopoll.dp_nfds = 2;
27         dopoll.dp_fds = pollfd;
28         result = Ioctl(wfd, DP_POLL, &dopoll);
29         /* faz um loop pelos descritores de arquivo prontos */
30         for (i = 0; i < result; i++) {
31             if (dopoll.dp_fds[i].fd == sockfd) {
32                 /* o soquete é legível */
33                 if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
34                     if (stdineof == 1)
35                         return; /* término normal */
36                     else
37                         err_quit("str_cli: server terminated prematurely");
38                 }
39                 Write(fileno(stdout),buf, n);
40             } else {
41                 /* a entrada é legível */
42                 if ( (n = Read(fileno(fp), buf, MAXLINE)) == 0) {
43                     stdineof = 1;
44                     Shutdown(sockfd, SHUT_WR); /* envia FIN */
45                     continue;
46                 }
47                 Writen(sockfd,buf, n);
48             }
49         }
50     }
51 }
```

advio/str_cli_poll03.c

Figura 14.15 Função `str_cli` utilizando `/dev/poll`.

Listando os descritores para `/dev/poll`

14–21 Depois de preencher um array de estruturas `pollfd`, nós as passamos para `/dev/poll`. Nosso exemplo somente requer dois descritores de arquivo, assim utilizamos um array estático de estruturas. Na prática, os programas que utilizam `/dev/poll` precisam monitorar centenas ou mesmo milhares de descritores de arquivo; dessa forma, o array possivelmente seria alocado dinamicamente.

Esperando o trabalho

24–28 Em vez de chamar `select`, esse programa bloqueia, esperando o trabalho, na chamada `ioctl`. O retorno é o número de descritores de arquivo que está pronto.

Fazendo um loop pelos descritores

30–49 O código no nosso exemplo é simplificado porque sabemos que os descritores de arquivo prontos serão `sockfd`, o descritor de arquivo de entrada, ou ambos. Em um programa de larga escala, esse loop seria mais complexo, talvez até mesmo despachando o trabalho para threads.

A interface `kqueue`

O FreeBSD introduziu a interface `kqueue` em sua versão 4.1. Essa interface permite a um processo registrar um "filtro de evento" que descreve os eventos `kqueue` em que está interessado. Os eventos incluem E/S de arquivo e tempos-limite como `select`, mas também adiciona E/S assíncrona, notificação de modificação de arquivo (por exemplo, uma notificação quando um arquivo é removido ou modificado), monitoramento de processo (por exemplo, uma notificação quando um dado processo termina ou chama `fork`) e tratamento de sinal. A interface `kqueue` inclui as duas funções e a macro a seguir:

```
#include <sys/types.h>
#include <sys/event.h>
#include <sys/time.h>

int kqueue(void);
int kevent(int kq, const struct kevent *changelist, int nchanges,
           struct kevent *eventlist, int nevents,
           const struct timespec *timeout);
void EV_SET(struct kevent *kev, uintptr_t ident, short filter,
            u_short flags, u_int fflags, intptr_t data, void *udata);
```

A função `kqueue` retorna um novo descritor `kqueue`, que pode ser utilizado em futuras chamadas a `kevent`. A função `kevent` é utilizada para registrar eventos de interesse e determinar se algum evento ocorreu. Os parâmetros *changelist* e *nchanges* descrevem as alterações a serem feitas nos eventos de interesse, ou são `NULL` e `0`, respectivamente, se nenhuma alteração deve ser feita. Se *nchanges* não for zero, cada alteração no filtro de evento solicitada no array *changelist* é realizada. Quaisquer filtros cujas condições foram desencadeadas, incluindo aquelas recém-adicionadas ao *changelist*, são retornados pelo parâmetro *eventlist*, que aponta para um array de *nevents* estruturas `struct kevent`. A função `kevent` retorna o número de eventos que são retornados ou zero se excedeu um limite de tempo. O argumento *timeout* armazena o tempo-limite, tratado da mesma maneira como `select`: `NULL` para bloquear, um `timespec` não-zero para especificar um tempo-limite explícito e um `timespec` zero para realizar uma verificação não-bloqueadora nos eventos. Observe que o parâmetro *timeout* é um `struct timespec`, o que é diferente do `struct timeval` de `select` pelo fato de utilizar uma precisão de nanossegundo em vez microssegundo.

A estrutura `kevent` é definida incluindo o cabeçalho `<sys/event.h>`.

```
struct kevent {
    uintptr_t   ident;      /* identificador (por exemplo, descritor de arquivo) */
    short       filter;     /* tipo de filtro (por exemplo, EVFILT_READ) */
    u_short     flags;      /* flags de ação (por exemplo, EV_ADD) */
    u_int       fflags;     /* flags específicos de filtros */
    intptr_t    data;       /* dados específicos de filtros */
    void        *udata;     /* dados de usuário opacos */
};
```

As ações para alterar um filtro e os valores de retorno de flag são mostrados na Figura 14.16. Os tipos de filtro são mostrados na Figura 14.17.

Modificamos nossa função `str_cli`, que utilizava `select` na Figura 6.13, para `kqueue` na Figura 14.18.

Determinando se o ponteiro de arquivo aponta para um arquivo

10–11 O comportamento de `kqueue` no EOF é diferente dependendo se o descritor de arquivo está associado a arquivo, pipe ou terminal, assim utilizamos a chamada `fstat` para determinar se ele é um arquivo. Utilizaremos essa determinação mais tarde.

Configurando as estruturas `kevent` para `kqueue`

12–13 Utilizamos a macro `EV_SET` para configurar duas estruturas `kevent`; ambas especificam um filtro de leitura (`EVFILT_READ`) e uma solicitação para adicionar esse evento ao filtro (`EV_ADD`).

Criando `kqueue` e adicionando filtros

14–16 Chamamos `kqueue` para obter um descritor `kqueue` e configurar o tempo-limite como zero para permitir uma chamada não-bloqueadora a `kevent`, e chamamos `kevent` com nosso array de `kevents` como a solicitação de alteração.

flags	Descrição	alteração	retorno
`EV_ADD`	Adiciona um novo evento; automaticamente ativado, a menos que `EV_DISABLE` esteja especificado	•	
`EV_CLEAR`	Inicializa o estado de evento depois de o usuário recuperá-lo	•	
`EV_DELETE`	Exclui evento do filtro	•	
`EV_DISABLE`	Desativa o evento, mas não o remove do filtro	•	
`EV_ENABLE`	Reativa o evento anteriormente desativado	•	
`EV_ONESHOT`	Exclui o evento depois de ele ser desencadeado uma vez	•	
`EV_EOF`	Uma condição de EOF ocorreu		•
`EV_ERROR`	Um erro ocorreu; `errno` está no elemento `data`		•

Figura 14.16 *flags* para `k` operações `event`.

filter	Descrição
`EVFILT_AIO`	Eventos de E/S assíncronos (Seção 6.2)
`EVFILT_PROC`	Eventos do processo `exit`, `fork` ou `exec`
`EVFILT_READ`	Descritor é legível, semelhante a `select`
`EVFILT_SIGNAL`	Recepção de sinal
`EVFILT_TIMER`	Timers periódicos ou
`EVFILT_VNODE`	Eventos de modificação e exclusão de arquivo
`EVFILT_WRITE`	Descritor está gravável, semelhante a `select`

Figura 14.17 *filtros* para operações `kevent`.

Ficando em loop eternamente, bloqueando em `kevent`

17–18 Ficamos em loop eternamente, bloqueando em `kevent`. Passamos uma lista de alteração `NULL`, pois estamos interessados apenas nos eventos que já registramos e em um tempo-limite `NULL` para bloquear para sempre.

Fazendo um o loop pelos eventos retornados

19 Verificamos cada evento que retornou e o processamos individualmente.

advio/str_cli_kqueue04.c

```
 1 #include   "unp.h"
 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     int    kq, i, n, nev, stdineof = 0, isfile;
 6     char   buf[MAXLINE];
 7     struct kevent kev[2];
 8     struct timespec ts;
 9     struct stat st;
10     isfile = ((fstat(fileno(fp), &st) == 0) &&
11                (st.st_mode & S_IFMT) == S_IFREG);
12     EV_SET(&kev[0], fileno(fp), EVFILT_READ, EV_ADD, 0, 0, NULL);
13     EV_SET(&kev[1], sockfd, EVFILT_READ, EV_ADD, 0, 0, NULL);
14     kq = Kqueue();
15     ts.tv_sec = ts.tv_nsec = 0;
16     Kevent(kq, kev, 2, NULL, 0, &ts);
17     for ( ; ; ) {
18         nev = Kevent(kq, NULL, 0, kev, 2, NULL);
19         for (i = 0; i < nev; i++) {
20             if (kev[i].ident == sockfd) { /* o soquete é legível */
21                 if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
22                     if (stdineof == 1)
23                         return; /* término normal */
24                     else
25                         err_quit("str_cli: server terminated prematurely");
26                 }
27                 Write(fileno(stdout), buf, n);
28             }
29             if (kev[i].ident == fileno(fp)) { /* a entrada é legível */
30                 n = Read(fileno(fp), buf, MAXLINE);
31                 if (n > 0)
32                     Writen(sockfd, buf, n);
33                 if (n == 0 || (isfile && n == kev[i].data)) {
34                     stdineof = 1;
35                     Shutdown(sockfd, SHUT_WR); /* envia FIN */
36                     kev[i].flags = EV_DELETE;
37                     Kevent(kq, &kev[i], 1, NULL, 0, &ts); /* remove kevent */
38                     continue;
39                 }
40             }
41         }
42     }
43 }
```

advio/str_cli_kqueue04.c

Figura 14.18 Função `str_cli` utilizando `kqueue`.

O soquete é legível

20–28 Esse código é exatamente o mesmo da Figura 6.13.

A entrada é legível

29–40 Esse código é semelhante à Figura 6.13, mas é estruturado de maneira um pouco diferente para tratar como `kqueue` informa um EOF. Em pipes e terminais, `kqueue` retorna uma indicação legível de que um EOF está pendente, assim como `select`. Mas, em arquivos, `kqueue` simplesmente retorna o número de bytes que permanecem no arquivo no membro `data` de `struct kevent` e assume que a aplicação saberá quando alcança o fim. Portanto, reestruturamos o loop para gravar os dados na rede se um número de bytes diferente de zero foi lido. Em seguida, verificamos nossa condição de EOF modificada: se tivermos lido zero bytes ou se for um arquivo e tivermos lido o número de bytes deixados nele. A outra modificação em relação à Figura 6.13 é que, em vez de utilizar `FD_CLR` para remover o descritor de entrada do conjunto de arquivos, configuramos os flags como `EV_DELETE` e chamamos `kevent` para remover esse evento do filtro no kernel.

Sugestões

Devemos ter cuidado com essas interfaces recentemente desenvolvidas ao ler a documentação específica a versões do SO. Essas interfaces são freqüentemente alteradas de maneiras sutis entre versões enquanto os fornecedores trabalham nos detalhes de como elas devem funcionar.

Embora escrever um código não-portável seja, em geral, algo a ser evitado, é bem comum utilizar qualquer meio possível para otimizar uma aplicação de rede utilizada maciçamente para o servidor específico em que ela executa.

14.10 Resumo

Há três maneiras principais de configurar um limite de tempo em uma operação de soquete:

- Utilizar a função `alarm` e o sinal `SIGALRM`
- Utilizar o limite de tempo que é fornecido por `select`
- Utilizar as opções de soquete `SO_RCVTIMEO` e `SO_SNDTIMEO` mais recentes

A primeira é fácil, mas envolve tratamento de sinal e, como veremos na Seção 20.5, pode levar a condições de corrida. Utilizar `select` significa que bloqueamos nessa função com o tempo-limite fornecido em vez de bloquear em uma chamada a `read`, `write` ou `connect`. A terceira alternativa, utilizar as novas opções de soquete, também é fácil, mas não é fornecida por todas as implementações.

`recvmsg` e `sendmsg` são as mais gerais entre os cinco grupos de funções de E/S fornecidos. Elas combinam a capacidade de especificar um flag `MSG_`*xxx* (a partir de `recv` e `send`), acrescida do emprego da capacidade de retornar ou especificar o endereço de protocolo do peer (a partir de `recvfrom` e `sendto`), com a capacidade de utilizar múltiplos buffers (a partir de `readv` e `writev`), juntamente com os dois novos recursos: retornar flags à aplicação e receber ou enviar dados auxiliares.

Descrevemos dez formas diferentes de dados auxiliares neste texto, seis das quais são novas no IPv6. Dados auxiliares consistem em um ou mais objetos de dados auxiliares, cada qual precedido por uma estrutura `cmsghdr` para especificar seu comprimento, nível de protocolo e tipo de dados. Cinco funções que iniciam com `CMSG_` são utilizadas para construir e analisar sintaticamente os dados auxiliares.

Os soquetes podem ser utilizados com a biblioteca de E/S-padrão do C, mas fazer isso acrescenta um outro nível de armazenamento em buffer a aquele já sendo realizado pelo TCP.

De fato, a falta de entendimento sobre o armazenamento em buffer realizado pela biblioteca de E/S-padrão é o problema mais comum com essa biblioteca. Como um soquete não é um dispositivo terminal, a solução comum a esse problema potencial é configurar o fluxo de E/S-padrão como não-armazenado no buffer ou então evitar completamente a E/S-padrão nos soquetes.

Muitos distribuidores fornecem maneiras avançadas de fazer polling (ou consulta seqüencial) para vários eventos sem o overhead requerido por `select` e `poll`. Embora escrever um código não-portável deva ser evitado sempre que possível, às vezes os benefícios da melhoria de desempenho excedem o risco da não-portabilidade.

Exercícios

14.1 O que acontece na Figura 14.1 quando redefinimos o handler de sinal e o processo não estabeleceu um handler para `SIGALRM`?

14.2 Na Figura 14.1, imprimimos um aviso se o processo já tiver um timer `alarm` configurado. Modifique a função para redefinir esse `alarm` para o processo depois de `connecte` antes que a função retorne.

14.3 Modifique a Figura 11.11 como a seguir: antes de chamar `read`, chame `recv` especificando `MSG_PEEK`. Quando isso retornar, chame `ioctl` com um comando de `FIONREAD` e imprima o número de bytes enfileirados no buffer de recebimento do soquete. Em seguida, chame `read` para na verdade ler os dados.

14.4 O que acontece com os dados em um buffer de E/S-padrão para o qual ainda não foi gerada a saída se o processo ultrapassar o final da função `main` em vez de chamar `exit`?

14.5 Aplique cada uma das duas alterações descritas seguindo a Figura 14.14 e verifique se ambas corrigem o problema do armazenamento em buffer.

Capítulo 15

Protocolos de Domínio Unix

15.1 Visão Geral

Os protocolos de domínio Unix não são um conjunto de protocolos real, mas uma maneira de realizar a comunicação cliente/servidor em um único host com a mesma API utilizada para clientes e servidores em diferentes hosts. Os protocolos de domínio Unix são uma alternativa aos métodos de comunicação entre processos (*interprocess communication* – IPC) descritos no Volume 2 desta série, quando o cliente e o servidor estão no mesmo host. Os detalhes sobre a implementação real dos soquetes de domínio Unix em um kernel derivado do Berkeley são fornecidos na Parte 3 do TCPv3.

Dois tipos de soquetes são fornecidos no domínio Unix: de fluxo (semelhantes aos do TCP) e de datagrama (semelhantes aos do UDP). Mesmo que um soquete bruto também seja fornecido, sua semântica nunca foi documentada, não é utilizado por nenhum programa que estes autores tenham conhecimento e não é definido pelo POSIX.

Os soquetes de domínio Unix são utilizados por três razões:

1. Em implementações derivadas do Berkeley, os soquetes de domínio Unix são freqüentemente duas vezes mais rápidos que um soquete TCP quando ambos os peers estão no mesmo host (páginas 223 e 224 do TCPv3). Uma aplicação sabe tirar proveito disso: o X Window System. Quando um cliente X11 inicia e abre uma conexão com o servidor X11, o cliente verifica o valor da variável de ambiente `DISPLAY`, que especifica o hostname, a janela e a tela do servidor. Se o servidor estiver no mesmo host do cliente, o cliente abre uma conexão de fluxo de domínio Unix com o servidor; caso contrário, abre uma conexão TCP com o servidor.
2. Os soquetes de domínio Unix são utilizados ao passar descritores entre processos no mesmo host. Forneceremos um exemplo completo disso na Seção 15.7.
3. Implementações mais recentes dos soquetes de domínio Unix fornecem as credenciais do cliente (ID de usuário e IDs de grupo) para o servidor, que pode fornecer uma verificação de segurança adicional. Descreveremos isso na Seção 15.8.

Os endereços de protocolo utilizados para identificar clientes e servidores no domínio Unix são os nomes de caminho dentro do sistema de arquivos normal. Lembre-se de que o

IPv4 utiliza uma combinação de endereços de 32 bits e números de porta de 16 bits para seus endereços de protocolo e o IPv6 utiliza uma combinação de endereços de 128 bits e números de porta de 16 bits para seus endereços de protocolo. Esses nomes de caminho não são arquivos Unix normais: não podemos ler nem gravar nesses arquivos exceto a partir de um programa que associou o nome de caminho a um soquete de domínio Unix.

15.2 Estrutura de endereços de soquete de domínio Unix

A Figura 15.1 mostra a estrutura de endereço de soquete de domínio Unix, definida incluindo o cabeçalho `<sys/un.h>`.

```
struct sockaddr_un {
   sa_family_t  sun_family;        /* AF_LOCAL */
   char         sun_path[104];     /* nome de caminho terminado por
                                      caractere nulo */
};
```

Figura 15.1 A estrutura de endereço de soquete de domínio Unix: `sockaddr_un`.

> A especificação POSIX não define o comprimento do array `sun_path` e, especificamente, adverte que as aplicações não devem assumir um comprimento particular. Utilize o operador `sizeof` para encontrar o comprimento em tempo de execução e para verificar se um nome de caminho se ajusta no array. Provavelmente, o comprimento está entre 92 e 108, em vez de um valor maior suficientemente grande para armazenar qualquer nome de caminho. A razão desses limites é um artefato da implementação que data do 4.2BSD exigindo que essa estrutura se ajuste em um mbuf (um buffer de memória do kernel) de 128 bytes.

O nome de caminho armazenado no array `sun_path` deve ser terminado por caractere nulo. A macro `SUN_LEN` é fornecida e aceita um ponteiro para uma estrutura `sockaddr_un` e retorna o comprimento da estrutura, incluindo o número de bytes não-nulos no nome de caminho. O endereço não-especificado é indicado por uma string nula como o nome de caminho, isto é, uma estrutura com `sun_path[0]` igual a 0. Essa estrutura é equivalente, no domínio Unix, à constante `INADDR_ANY` do IPv4 e à constante `IN6ADDR_ANY_INIT` do IPv6.

> POSIX renomeia os protocolos de domínio Unix como "IPC local" para remover a dependência no sistema operacional Unix. A constante histórica `AF_UNIX` torna-se `AF_LOCAL`. Contudo, ainda utilizamos o termo "domínio Unix", uma vez que isso se tornou *de facto* seu nome, independentemente do SO subjacente. Além disso, mesmo com o POSIX tentando tornar esses protocolos independentes de SO, a estrutura do endereço de soquete ainda mantém o sufixo `_un`!

Exemplo: `bind` do soquete de domínio Unix

O programa na Figura 15.2 cria um soquete de domínio Unix, chama `bind` para vincular um nome de caminho e então chama `getsockname` e imprime o nome do caminho vinculado.

unixdomain/unixbind.c

```
1 #include   "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5      int      sockfd;
6      socklen_t len;
7      struct sockaddr_un addr1, addr2;
```

Figura 15.2 `bind` de um nome de caminho para um soquete de domínio Unix (*continua*).

```
 8     if(argc != 2)
 9         err_quit("usage: unixbind <pathname>");

10     sockfd = Socket(AF_LOCAL, SOCK_STREAM, 0);

11     unlink(argv[1]);      /* OK se isso falhar */

12     bzero(&addr1, sizeof(addr1));
13     addr1.sun_family = AF_LOCAL;
14     strncpy(addr1.sun_path, argv[1], sizeof(addr1.sun_path) - 1);
15     Bind(sockfd, (SA *) &addr1, SUN_LEN(&addr1));

16     len = sizeof(addr2);
17     Getsockname(sockfd, (SA *) &addr2, &len);
18     printf("bound name = %s, returned len = %d\n", addr2.sun_path, len);

19     exit(0);
20 }
```

unixdomain/unixbind.c

Figura 15.2 `bind` de um nome de caminho para um soquete de domínio Unix (*continuação*).

Remoção do nome de caminho primeiro

11 O nome de caminho em que utilizamos `bind` para vinculá-lo ao soquete é o argumento de linha de comando. Mas o `bind` irá falhar se o nome de caminho já existir no sistema de arquivos. Portanto, chamamos `unlink` para excluir o nome de caminho, caso ele já exista. Se não existir, `unlink` retorna um erro, o que ignoramos.

`bind` e então `getsockname`

12–18 Copiamos o argumento de linha de comando utilizando `strncpy`, para evitar que a estrutura estoure se o nome de caminho for muito longo. Uma vez que inicializamos a estrutura como zero e então subtraímos um do tamanho do array `sun_path`, sabemos que o nome de caminho é terminado por caractere nulo. `bind` é chamado e utilizamos a macro `SUN_LEN` para calcular o argumento de comprimento da função. Em seguida, chamamos `getsockname` para buscar o nome recém-vinculado e imprimimos o resultado.

Se executarmos esse programa no Solaris, obteremos os seguintes resultados:

```
solaris % umask                                    primeiro, imprime nosso valor de umask
022                                                shells imprimem esse valor em octal
solaris % unixbind /tmp/moose
bound name = /tmp/moose, returned len = 13
solaris % unixbind /tmp/moose                      execute-o novamente
bound name = /tmp/moose, returned len = 13
solaris % ls -l /tmp/moose
srwxr-xr-x  1 andy    staff            0 Aug 10 13:13 /tmp/moose
solaris % ls -lF /tmp/moose
srwxr-xr-x  1 andy    staff            0 Aug 10 13:13 /tmp/moose=
```

Primeiro, imprimimos nosso valor de `umask` porque o POSIX especifica que as permissões de acesso a arquivos do nome de caminho resultante devem ser modificadas por esse valor. Nosso valor 22 desativa os bits de gravação do grupo e de outros bits de gravação. Em seguida, executamos o programa e verificamos se o comprimento retornado por `getsockname` é 13: 2 bytes para o membro `sun_family` e 11 bytes para o nome do caminho real (excluindo o byte nulo de término). Esse é um exemplo de um argumento de valor-resultado cujo resultado, quando a função retorna, é um valor diferente daquele quando a função foi chamada. Podemos dar saída para o nome de caminho utilizando o formato `%s` de `printf` porque o nome de caminho é terminado por um caractere nulo no membro `sun_path`. Depois, executamos o programa novamente, para verificar se chamar `unlink` remove o nome de caminho.

Executamos `ls -l` para ver as permissões e o tipo de arquivo. No Solaris (e na maioria das variantes do Unix), o tipo de arquivo é um soquete, o que é impresso como `s`. Também prestamos atenção se os bits de permissão foram modificados apropriadamente pelo valor de `umask`. Por fim, executamos `ls` novamente, com a opção `-F`, que faz com que o Solaris acrescente um sinal de igual ao nome de caminho.

> Historicamente, o valor de `umask` não se aplicava à criação de soquetes de domínio Unix, mas, com o passar dos anos, a maioria dos fornecedores do Unix corrigiu isso de modo que as permissões se ajustem às expectativas. Ainda há sistemas em que os bits de permissão de arquivo podem mostrar todas ou nenhuma permissão (independentemente da configuração de `umask`). Além disso, alguns sistemas mostram o arquivo como um FIFO, que é impresso como `p` e nem todos os sistemas mostram o sinal de igual com `ls -F`. O comportamento que mostramos anteriormente é o mais comum.

15.3 Função `socketpair`

A função `socketpair` cria dois soquetes que são então conectados. Essa função se aplica somente a soquetes de domínio Unix.

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

int socketpair(int family, int type, int protocol, int sockfd[2]);
```

Retorna: não-zero se OK, –1 se erro

family deve ser `AF_LOCAL` e *protocol* dever ser 0. *type*, porém, pode ser `SOCK_STREAM` ou `SOCK_DGRAM`. Os dois descritores de soquete criados são retornados como *sockfd[0]* e *sockfd[1]*.

> Essa função é semelhante à função `pipe` do Unix: dois descritores são retornados e cada um é conectado ao outro. De fato, as implementações derivadas do Berkeley empregam `pipe` realizando as mesmas operações internas a `socketpair` (páginas 253 e 254 do TCPv3).

Os dois soquetes criados são não-identificados; isto é, não há nenhum `bind` implícito envolvido.

O resultado de `socketpair` com um *type* de `SOCK_STREAM` é chamado *pipe de fluxo*. Ele é semelhante a um pipe regular do Unix (criado pela função `pipe`), mas um pipe de fluxo é *full-duplex*; isto é, os dois descritores podem ser lidos e gravados. Mostramos um exemplo de pipe de fluxo criado por `socketpair` na Figura 15.7.

> O POSIX não exige pipes full-duplex. No SVR4, `pipe` retorna dois descritores full-duplex, enquanto kernels derivados do Berkeley tradicionalmente retornam dois descritores half-duplex (Figura 17.31 do TCPv3).

15.4 Funções de soquete

Há várias diferenças e restrições nas funções de soquete ao utilizar soquetes de domínio Unix. Listamos os requisitos POSIX se aplicáveis e observamos que nem todas as implementações estão atualmente nesse nível.

1. As permissões default de acesso a arquivo para um nome de caminho criado por `bind` devem ser 0777 (leitura, gravação e execução pelo usuário, grupo e outros), modificadas pelo valor de `umask` atual.

2. O nome de caminho associado a um soquete de domínio Unix deve ser absoluto, não relativo. A razão de evitar este último é que sua solução depende do diretório atual de trabalho do chamador. Isto é, se o servidor vincular um nome de caminho relativo, o cliente deve estar no mesmo diretório que o servidor (ou deve conhecer esse diretório) para que a chamada do cliente a `connect` ou `sendto` seja bem-sucedida.

   O POSIX informa que vincular um nome de caminho relativo a um soquete de domínio Unix fornece resultados imprevisíveis.

3. O nome do caminho especificado em uma chamada a `connect` deve estar atualmente vinculado a um soquete aberto de domínio Unix do mesmo tipo (fluxo ou datagrama). Erros ocorrem se: (i) o nome de caminho existe, mas não é um soquete; (ii) o nome de caminho existe e é um soquete, mas nenhum descritor de soquete aberto está associado ao nome de caminho; ou (iii) o nome de caminho existe e é um soquete aberto, mas é do tipo errado (isto é, um soquete de fluxo de domínio Unix não pode ser conectado a um nome de caminho associado a um soquete de datagrama de domínio Unix e vice-versa).

4. O teste de permissão associado a `connect` para um soquete de domínio Unix é o mesmo se `open` tiver sido chamado somente para acesso de gravação no nome de caminho.

5. Os soquetes de fluxo de domínio Unix são semelhantes aos soquetes TCP: eles fornecem uma interface de fluxo de bytes para o processo sem limites de registro.

6. Se uma chamada a `connect` para um soquete de fluxo de domínio Unix descobrir que a fila do soquete ouvinte está cheia (Seção 4.5), `ECONNREFUSED` é retornado imediatamente. Isso difere do TCP: o ouvinte do TCP ignora um SYN que chega se a fila do soquete estiver cheia e o conector TCP faz mais uma tentativa enviando o SYN várias vezes.

7. Os soquetes de datagrama de domínio Unix são semelhantes aos soquetes UDP: eles fornecem um serviço de datagrama não-confiável que preserva os limites de registro.

8. Diferentemente dos soquetes UDP, enviar um datagrama em um soquete não-vinculado de datagrama de domínio Unix não vincula um nome de caminho ao soquete. (Lembre-se de que enviar um datagrama UDP a um soquete UDP não-vinculado faz com que uma porta efêmera seja vinculada ao soquete.) Isso significa que o receptor do datagrama não será capaz de enviar uma resposta a menos que o emissor tenha vinculado um nome de caminho ao seu soquete. De maneira semelhante, diferentemente do TCP e do UDP, chamar `connect` em um soquete de datagrama de domínio Unix não vincula um nome de caminho ao soquete.

15.5 Cliente/servidor de fluxo de domínio Unix

Agora, recodificamos nosso cliente/servidor TCP de eco do Capítulo 5 para utilizar soquetes de domínio Unix. A Figura 15.3 mostra o servidor, que é uma modificação da Figura 5.12, para utilizar o protocolo de fluxo de domínio Unix em vez do TCP.

8 O tipo de dados das duas estruturas de endereço de soquete agora é `sockaddr_un`.

10 O primeiro argumento para `socket` é `AF_LOCAL`, para criar um soquete de fluxo de domínio Unix.

11–15 A constante `UNIXSTR_PATH` é definida em `unp.h` como `/tmp/unix.str`. Primeiro, chamamos `unlink` para desvincular o nome de caminho, se ele existir a partir de uma execução anterior do servidor, e então inicializamos a estrutura de endereço de soquete antes de chamar `bind`. Um erro de `unlink` é aceitável.

Observe que essa chamada a `bind` difere da chamada na Figura 15.2. Aqui, especificamos o tamanho da estrutura de endereço de soquete (o terceiro argumento) como o tamanho total da estrutura `sockaddr_un`, não apenas o número de bytes ocupado pelo nome de caminho. Ambos os comprimentos são válidos, uma vez que o nome de caminho precisa terminar por caractere nulo.

O restante da função é o mesmo da Figura 5.12. A mesma função `str_echo` é utilizada (Figura 5.3).

A Figura 15.4 é o cliente de eco do protocolo de fluxo de domínio Unix. Ela é uma modificação da Figura 5.4.

6 A estrutura de endereço de soquete que deve conter o endereço do servidor agora é uma estrutura `sockaddr_un`.

7 O primeiro argumento para `socket` é `AF_LOCAL`.

8–10 O código para preencher a estrutura de endereço de soquete é idêntico ao código mostrado para o servidor: inicialize a estrutura em 0, configure a família como `AF_LOCAL` e copie o nome de caminho para o membro `sun_path`.

12 A função `str_cli` é a mesma que a anterior (a Figura 6.13 foi a última versão que desenvolvemos).

unixdomain/unixstrserv01.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5      int      listenfd, connfd;
 6      pid_t childpid;
 7      socklen_t clilen;
 8      struct sockaddr_un cliaddr, servaddr;
 9      void  sig_chld(int);
10      listenfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
11      unlink(UNIXSTR_PATH);
12      bzero(&servaddr, sizeof(servaddr));
13      servaddr.sun_family = AF_LOCAL;
14      strcpy(servaddr.sun_path, UNIXSTR_PATH);
15      Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
16      Listen(listenfd, LISTENQ);
17      Signal(SIGCHLD, sig_chld);
18      for ( ; ; ) {
19        clilen = sizeof(cliaddr);
20        if ( (connfd = accept(listenfd, (SA *) &cliaddr, &clilen)) < 0) {
21            if (errno == EINTR)
22                continue;          /* de volta para for()*/
23            else
24                err_sys("accept error");
25        }
26        if ( (childpid = Fork()) == 0) { /* processo-filho */
27            Close(listenfd);       /* fecha o soquete ouvinte */
28            str_echo(connfd);      /* solicitação de processo */
29            exit(0);
30        }
31        Close(connfd);             /* o pai fecha o soquete conectado */
32      }
33 }
```

unixdomain/unixstrserv01.c

Figura 15.3 O servidor de eco do protocolo de fluxo de domínio Unix.

15.6 Cliente/servidor de datagrama de domínio Unix

Agora, recodificamos nosso cliente/servidor UDP das Seções 8.3 e 8.5 para utilizar soquetes de datagrama de domínio Unix. A Figura 15.5 mostra o servidor, uma modificação da Figura 8.3.

unixdomain/unixstrcli01.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5      int       sockfd;
 6      struct sockaddr_un servaddr;
 7      sockfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
 8      bzero(&servaddr, sizeof(servaddr));
 9      servaddr.sun_family = AF_LOCAL;
10      strcpy(servaddr.sun_path, UNIXSTR_PATH);
11      Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
12      str_cli(stdin, sockfd);     /* faz tudo */
13      exit(0);
14 }
```

unixdomain/unixstrcli01.c

Figura 15.4 O cliente de eco do protocolo de fluxo de domínio Unix.

unixdomain/unixdgserv01.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5      int       sockfd;
 6      struct sockaddr_un servaddr, cliaddr;
 7      sockfd = Socket(AF_LOCAL, SOCK_DGRAM, 0);
 8      unlink(UNIXDG_PATH);
 9      bzero(&servaddr, sizeof(servaddr));
10      servaddr.sun_family = AF_LOCAL;
11      strcpy(servaddr.sun_path, UNIXDG_PATH);
12      Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));
13      dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
14 }
```

unixdomain/unixdgserv01.c

Figura 15.5 O servidor de eco do protocolo de datagrama de domínio Unix.

6 O tipo de dados das duas estruturas de endereço de soquete agora é `sockaddr_un`.

7 O primeiro argumento para `socket` é `AF_LOCAL`, para criar um soquete de datagrama de domínio Unix.

8–12 A constante `UNIXDG_PATH` é definida em `unp.h` como sendo `/tmp/unix.dg`. Primeiro, chamamos `unlink` para desvincular o nome de caminho, no caso de haver uma vinculação de uma execução anterior do servidor, e então inicializamos a estrutura de endereço de soquete antes de chamar `bind`. Um erro de `unlink` é aceitável.

13 A mesma função `dg_echo` é utilizada (Figura 8.4).

A Figura 15.6 mostra o cliente de eco com o protocolo de datagrama no domínio Unix. Ela é uma modificação da Figura 8.7.

unixdomain/unixdgcli01.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5      int       sockfd;
 6      struct sockaddr_un cliaddr, servaddr;
 7      sockfd = Socket(AF_LOCAL, SOCK_DGRAM, 0);
 8      bzero(&cliaddr, sizeof(cliaddr)); /* vincula um endereço para nós */
 9      cliaddr.sun_family = AF_LOCAL;
10      strcpy(cliaddr.sun_path, tmpnam(NULL));
11      Bind(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
12      bzero(&servaddr, sizeof(servaddr)); /* preenche o endereço do servidor */
13      servaddr.sun_family = AF_LOCAL;
14      strcpy(servaddr.sun_path, UNIXDG_PATH);
15      dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));
16      exit(0);
17 }
```

unixdomain/unixdgcli01.c

Figura 15.6 O cliente de eco com o protocolo de datagrama no domínio Unix.

6 A estrutura de endereço de soquete que deve conter o endereço do servidor agora é uma estrutura `sockaddr_un`. Também alocamos uma dessas estruturas que deve conter o endereço do cliente, que discutiremos em breve.

7 O primeiro argumento para `socket` é `AF_LOCAL`.

8–11 Diferentemente do nosso cliente UDP, ao utilizar o protocolo de datagrama de domínio Unix, devemos chamar `bind` explicitamente para vincular um nome de caminho ao nosso soquete, de modo que o servidor tenha um nome de caminho ao qual possa enviar sua resposta. Chamamos `tmpnam` para atribuir um nome de caminho único e então `bind` para vinculá-lo ao nosso soquete. Lembre-se, da Seção 15.4, de que enviar um datagrama a um soquete de datagrama desvinculado de domínio Unix não vincula implicitamente um nome de caminho ao soquete. Portanto, se omitirmos esse passo, a chamada do servidor a `recvfrom` na função `dg_echo` retorna um nome de caminho nulo, que então causa um erro quando o servidor chama `sendto`.

12–14 O código para preencher a estrutura de endereço de soquete com o nome do caminho bem-conhecido do servidor é idêntico ao código mostrado anteriormente para o servidor.

15 A função `dg_cli` é a mesma que a mostrada na Figura 8.8.

15.7 Passando descritores

Quando pensamos em passar um descritor aberto de um processo para outro, normalmente pensamos em um destes pontos:

- Um filho que compartilha todos os descritores abertos com o pai depois de uma chamada a fork
- Todos os descritores normalmente permanecem abertos quando `exec` é chamado

No primeiro exemplo, o processo abre um descritor, chama `fork` e então o pai fecha o descritor, deixando que o filho trate o descritor. Isso passa um descritor aberto do pai para o

filho. Mas também gostaríamos que o filho possa abrir um descritor e passá-lo de volta para o pai.

Os sistemas Unix atuais fornecem uma maneira de passar qualquer descritor aberto de um processo para qualquer outro processo. Isto é, não há necessidade de os processos serem relacionados, como um pai e seu filho. A técnica requer primeiro o estabelecimento de um soquete de domínio Unix entre os dois processos e então a utilização de `sendmsg` para enviar uma mensagem especial pelo soquete de domínio Unix. Essa mensagem é tratada de maneira especial pelo kernel, passando o descritor aberto do emissor para o receptor.

> A "magia negra" realizada pelo kernel do 4.4BSD na passagem de um descritor aberto por um soquete de domínio Unix é descrita em detalhes no Capítulo 18 do TCPv3.
>
> O SVR4 utiliza uma técnica diferente dentro do kernel para passar um descritor aberto: os comandos `I_SENDFD` e `I_RECVFD` `ioctl`, descritos na Seção 15.5.1 do APUE. Mas o processo ainda pode acessar esse recurso do kernel utilizando um soquete de domínio Unix. Neste texto, descrevemos o uso dos soquetes de domínio Unix para passar descritores abertos, uma vez que essa é a técnica de programação mais portável: ela funciona tanto nos kernels derivados do Berkeley como nos do SVR4, ao passo que utilizar `I_SENDFD` e `I_RECVFD` `ioctl` funciona somente no SVR4.
>
> A técnica do 4.4BSD permite que múltiplos descritores sejam passados com um único `sendmsg`, enquanto a técnica do SVR4 passa somente um único descritor por vez. Todos os nossos exemplos passam um descritor por vez.

Os passos envolvidos na passagem de um descritor entre dois processos são estes:

1. Criação um soquete de domínio Unix, de um soquete de fluxo ou de um soquete de datagrama.

   Se o objetivo for bifurcar um filho com `fork` e fazer com que ele abra e passe o descritor de volta para o pai, o pai poderá chamar `socketpair` para criar um pipe de fluxo que pode ser utilizado para trocar o descritor.

   Se os processos não estiverem relacionados, o servidor deve criar um soquete de fluxo de domínio Unix e chamar `bind` para vincular um nome de caminho a ele, permitindo que o cliente chame `connect` para se conectar a esse soquete. O cliente pode então enviar uma solicitação ao servidor para abrir algum descritor e o servidor pode passar de volta o descritor pelo soquete de domínio Unix. Alternativamente, um soquete de datagrama de domínio Unix também pode ser utilizado entre o cliente e o servidor, mas há pouca vantagem em fazer isso, além da possibilidade de um datagrama ser descartado. Utilizaremos um soquete de fluxo entre o cliente e o servidor em um exemplo apresentado mais adiante nesta seção.
2. Um processo abre um descritor chamando uma das funções do Unix que retorna um descritor: por exemplo, `open`, `pipe`, `mkfifo`, `socket` ou `accept`. *Qualquer* tipo de descritor pode ser passado de um processo para outro, razão pela qual chamamos essa técnica de "passagem de descritor" e não "passagem de descritor de arquivo".
3. O processo emissor constrói uma estrutura `msghdr` (Seção 14.5) que contém o descritor a ser passado. O POSIX especifica que o descritor é enviado como dados auxiliares (o membro `msg_control` da estrutura `msghdr`, Seção 14.6), porém implementações mais antigas utilizam o membro `msg_accrights`. O processo emissor chama `sendmsg` para enviar o descritor pelo soquete de domínio Unix no Passo 1. Nesse ponto, dizemos que o descritor está "em vôo" ("*in flight*"). Mesmo se o processo emissor fechar o descritor depois de chamar `sendmsg` – porém antes de o processo receptor chamar `recvmsg` (no próximo passo) –, o descritor permanece aberto para o processo receptor. Enviar um descritor incrementa a contagem de referência do descritor por um.

4. O processo receptor chama `recvmsg` para receber o descritor no soquete de domínio Unix no Passo 1. É normal que o número do descritor no processo receptor difira do número do descritor no processo de envio. Passar um descritor não é passar um número de descritor, mas envolve a criação de um novo descritor no processo receptor que faça referência dentro do kernel à mesma entrada de tabela de arquivo do descritor que foi enviado pelo processo emissor.

O cliente e o servidor devem ter algum protocolo de aplicação, de modo que o receptor do descritor saiba quando esperá-lo. Se o receptor chamar `recvmsg` sem alocar espaço para receber o descritor, e um descritor foi passado e está pronto para ser lido, o descritor que estava sendo passado é fechado (página 518 do TCPv2). Além disso, o flag `MSG_PEEK` deve ser evitado com `recvmsg` se um descritor for esperado, uma vez que o resultado é imprevisível.

Exemplo da passagem de descritor

Agora, forneceremos um exemplo da passagem de descritor. Escreveremos um programa identificado como `mycat` que recebe um nome de caminho como um argumento de linha de comando, abre o arquivo e copia-o para a saída-padrão. Mas, em vez de chamar a função `open` normal do Unix, chamamos nossa própria função identificada como `my_open`. Essa função cria um pipe de fluxo e chama `fork` e `exec` para iniciar outro programa que abre o arquivo desejado. Esse programa deve então passar o descritor aberto de volta para o pai pelo pipe de fluxo.

A Figura 15.7 mostra o primeiro passo: nosso programa `mycat` depois de criar um pipe de fluxo chamando `socketpair`. Designamos os dois descritores retornados por `socketpair` como `[0]` e `[1]`.

O processo então chama `fork` e o filho chama `exec` para executar o programa `openfile`. O pai fecha o descritor `[1]` e o filho fecha o descritor `[0]`. (Não há nenhuma diferença em qualquer uma das extremidades do pipe de fluxo; o filho poderia fechar `[1]` e o pai poderia fechar `[0]`.) Isso fornece o arranjo mostrado na Figura 15.8.

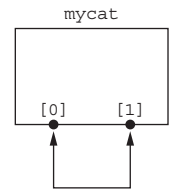


Figura 15.7 Programa `mycat` depois de criar o pipe de fluxo utilizando `socketpair`.

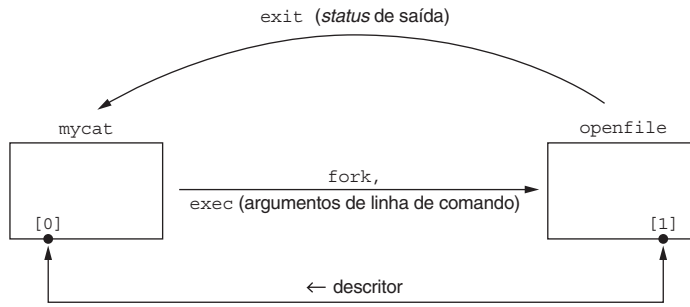


Figura 15.8 Programa `mycat` depois de chamar o programa `openfile`.

O pai deve passar três informações para o programa `openfile`: (i) o nome de caminho do arquivo a ser aberto, (ii) o modo de abertura (como apenas leitura, como leitura e gravação ou como apenas gravação) e (iii) o número do descritor correspondente à sua extremidade do pipe de fluxo (o que nós mostramos como `[1]`). Optamos por passar esses três itens como argumentos de linha de comando na chamada a `exec`. Um método alternativo é enviar esses três itens como dados pelo pipe de fluxo. O programa `openfile` reenvia o descritor aberto pelo pipe de fluxo e termina. O *status* da saída do programa diz ao pai se o arquivo poderia ser aberto e, se não, que tipo de erro ocorreu.

A vantagem de executar um outro programa para abrir o arquivo é que o programa poderia ser um binário do tipo "set-user-ID" que é executado com privilégios de root, permitindo que ele abra os arquivos que nós normalmente não temos permissão de abrir. Esse programa poderia estender o conceito sobre permissões do Unix normais (usuário, grupo e outros) a qualquer forma de verificação de acesso que ele deseja.

Iniciamos com o programa `mycat`, mostrado na Figura 15.9.

unixdomain/mycat.c

```
 1 #include   "unp.h"
 2 int    my_open(const char *, int);
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int    fd, n;
 7     char   buff[BUFFSIZE];
 8     if(argc != 2)
 9         err_quit("usage: mycat <pathname>");
10     if ( (fd = my_open(argv[1], O_RDONLY)) < 0)
11         err_sys("cannot open %s", argv[1]);
12     while ( (n = Read(fd, buff, BUFFSIZE)) > 0)
13         Write(STDOUT_FILENO, buff, n);
14     exit(0);
15 }
```

unixdomain/mycat.c

Figura 15.9 O programa `mycat`: copia um arquivo para a saída-padrão.

Se substituirmos a chamada a `my_open` por uma chamada a `open`, esse programa simples copia somente um arquivo para a saída-padrão.

A função `my_open`, mostrada na Figura 15.10, é concebida para se parecer como a função `open` normal do Unix para o seu chamador. Ela recebe dois argumentos, um nome de caminho e um modo de abertura (como `O_RDONLY` significando apenas leitura), abre o arquivo e retorna um descritor.

Criação do pipe de fluxo

8 `socketpair` cria um pipe de fluxo. Dois descritores são retornados: `sockfd[0]` e `sockfd[1]`. Esse é o estado que mostramos na Figura 15.7.

`fork` e `exec`

9–16 `fork` é chamado e o filho então fecha uma das extremidades do pipe de fluxo. O número do descritor na outra extremidade do pipe de fluxo é formatado no array `argsockfd` e o modo de abertura é formatado no array `argmode`. Chamamos `snprintf` porque os argumentos para `exec` devem ser strings de caracteres. O programa `openfile` é executado. A função

`execl` não deve retornar a menos que encontre um erro. Se bem-sucedida, a função `main` do programa `openfile` começa a executar.

O pai espera o filho

17–22 O pai fecha a outra extremidade do pipe de fluxo e chama `waitpid` para esperar que o filho termine. O *status* de término do filho é retornado na variável `status` e nós primeiro verificamos se o programa terminou normalmente (isto é, não foi terminado por um sinal). A macro `WEXITSTATUS` então converte o *status* de término no *status* de saída, cujo valor estará entre 0 e 255. Veremos em breve que, se o programa `openfile` encontrar um erro ao abrir o arquivo solicitado, ele termina com o correspondente valor de `errno` como seu *status* de saída.

unixdomain/myopen.c

```
 1 #include   "unp.h"

 2 int
 3 my_open(const char *pathname, int mode)
 4 {
 5     int      fd, sockfd[2], status;
 6     pid_t    childpid;
 7     char     c, argsockfd[10], argmode[10];

 8     Socketpair(AF_LOCAL, SOCK_STREAM, 0, sockfd);

 9     if ( (childpid = Fork()) == 0) { /* processo-filho */
10         Close(sockfd[0]);
11         snprintf(argsockfd, sizeof(argsockfd), "%d", sockfd[1]);
12         snprintf(argmode, sizeof(argmode), "%d", mode);
13         execl("./openfile", "openfile", argsockfd, pathname, argmode,
14               (char*) NULL);
15         err_sys("execl error");
16     }

17     /* processo pai - espera que o filho termine */
18     Close(sockfd[1]);     /* fecha a extremidade que não utilizamos */

19     Waitpid(childpid, &status, 0);
20     if (WIFEXITED(status) == 0)
21         err_quit("child did not terminate");
22     if ( (status = WEXITSTATUS(status)) == 0)
23         Read_fd(sockfd[0], &c, 1, &fd);
24     else {
25         errno = status;    /* configura o valor de errno a partir do
                                 status do filho */
26         fd = -1;
27     }

28     Close(sockfd[0]);
29     return (fd);
30 }
```

unixdomain/myopen.c

Figura 15.10 Função `my_open`: abre um arquivo e retorna um descritor.

Recebendo o descritor

23 Nossa função `read_fd`, mostrada a seguir, recebe o descritor no pipe de fluxo. Além do descritor, lemos um byte de dados, mas não fazemos nada com ele.

> Ao enviar e receber um descritor por um pipe de fluxo, sempre enviamos pelo menos um byte de dados, mesmo se o receptor não fizer nada com os dados. Caso contrário, o receptor não poderá informar se um valor de retorno igual a 0 proveniente de `read_fd` significa "nenhum dado (mas possivelmente um descritor)" ou "fim de arquivo" (end-of-file).

A Figura 15.11 mostra a função `read_fd` que chama `recvmsg` para receber dados e um descritor em um soquete de domínio Unix. Os três primeiros argumentos para essa função são os mesmos da função `read`, com um quarto argumento como um ponteiro para um inteiro que irá conter o descritor recebido no retorno.

9–26 Essa função deve lidar com duas versões de `recvmsg`: as com o membro `msg_control` e aquelas com o membro `msg_accrights`. Nosso cabeçalho `config.h` (Figura D.2) define a constante `HAVE_MSGHDR_MSG_CONTROL` se a versão `msg_control` for suportada.

Certificando-se de que `msg_control` está adequadamente alinhado

10–13 O buffer `msg_control` deve estar adequadamente alinhado para uma estrutura `cmsghdr`. Simplesmente alocar um array `char` é inadequado. Aqui declaramos uma `union` de uma estrutura `cmsghdr` com o array de caracteres, o que garante que o array seja adequadamente alinhado. Uma outra técnica é chamar `malloc`, mas isso iria requerer liberar a memória antes que a função retorne.

27–45 `recvmsg` é chamado. Se dados auxiliares retornarem, o formato é o mostrado na Figura 14.13. Verificamos se o comprimento, o nível e o tipo estão corretos, buscamos então o descritor recém-criado e o retornamos por meio do ponteiro `recvfd` do chamador. `CMSG_DATA` retorna o ponteiro para o membro `cmsg_data` do objeto de dados auxiliar como um ponteiro `unsigned char`. Aplicamos uma coerção nisso para um ponteiro `int` e buscamos o descritor de inteiro que está sendo apontado.

lib/read_fd.c

```
 1 #include   "unp.h"

 2 ssize_t
 3 read_fd(int fd, void *ptr, size_t nbytes, int *recvfd)
 4 {
 5     struct msghdr msg;
 6     struct iovec iov[1];
 7     ssize_t n;

 8 #ifdef HAVE_MSGHDR_MSG_CONTROL
 9     union {
10         struct cmsghdr cm;
11         char   control[CMSG_SPACE(sizeof(int))];
12     } control_un;
13     struct cmsghdr *cmptr;

14     msg.msg_control = control_un.control;
15     msg.msg_controllen = sizeof(control_un.control);
16 #else
17     int    newfd;

18     msg.msg_accrights = (caddr_t) & newfd;
19     msg.msg_accrightslen = sizeof(int);
20 #endif

21     msg.msg_name = NULL;
22     msg.msg_namelen = 0;

23     iov[0].iov_base = ptr;
24     iov[0].iov_len = nbytes;
25     msg.msg_iov = iov;
26     msg.msg_iovlen = 1;

27     if ( (n = recvmsg(fd, &msg, 0)) <= 0)
28         return (n);

29 #ifdef HAVE_MSGHDR_MSG_CONTROL
30     if ( (cmptr = CMSG_FIRSTHDR(&msg)) != NULL &&
```

Figura 15.11 Função `read_fd`: recebe dados e um descritor (*continua*).

```
31          cmptr->cmsg_len == CMSG_LEN(sizeof(int))) {
32          if (cmptr->cmsg_level != SOL_SOCKET)
33              err_quit("control level != SOL_SOCKET");
34          if (cmptr->cmsg_type != SCM_RIGHTS)
35              err_quit("control type != SCM_RIGHTS");
36          *recvfd = *((int *) CMSG_DATA(cmptr));
37      } else
38          *recvfd = -1;       /* descritor não foi passado */
39  #else
40      if (msg.msg_accrightslen == sizeof(int))
41          *recvfd = newfd;
42      else
43          *recvfd = -1;       /* descritor não foi passado */
44  #endif
45      return (n);
46  }
```

lib/read_fd.c

Figura 15.11 Função `read_fd`: recebe dados e um descritor (*continuação*).

Se o membro `msg_accrights` mais antigo for suportado, o comprimento deverá ser o tamanho de um inteiro e o descritor recém-criado retorna por meio do ponteiro `recvfd` do chamador.

A Figura 15.12 mostra o programa `openfile`. Ele recebe os três argumentos de linha de comando que devem ser passados e chama a função `open` normal.

unixdomain/openfile.c

```
 1  #include   "unp.h"
 2  int
 3  main(int argc, char **argv)
 4  {
 5      int     fd;
 6      if(argc != 4)
 7          err_quit("openfile <sockfd#> <filename> <mode>");
 8      if ( (fd = open(argv[2], atoi(argv[3]))) < 0)
 9          exit((errno > 0) ? errno : 255);
10      if (write_fd(atoi(argv[1]), "", 1, fd) < 0)
11          exit((errno > 0) ? errno : 255);
12      exit(0);
13  }
```

unixdomain/openfile.c

Figura 15.12 Função `openfile`: abre um arquivo e passa de volta o descritor.

Argumentos de linha de comando

7–12 Como dois desses três argumentos de linha de comando foram formatados para strings de caracteres por `my_open`, ambos são convertidos de volta em inteiros utilizando `atoi`.

Abrir o arquivo chamando `open`

9–10 O arquivo é aberto chamando `open`. Se um erro for encontrado, o valor de `errno` correspondente ao erro `open` é retornado como o *status* de saída do processo.

Passando de volta o descritor

11–12 O descritor é passado de volta por `write_fd`, que mostraremos a seguir. Esse processo então termina. Mas lembre-se de que anteriormente dissemos que era aceitável que o processo de envio fechasse o descritor que foi passado (o que acontece quando chamamos `exit`), porque o kernel sabe que o descritor está em retrocesso e o mantém aberto no processo receptor.

> O *status* de saída deve estar entre 0 e 255. O maior valor de `errno` é em torno de 150. Uma técnica alternativa que não requer que os valores de `errno` sejam menores que 256 seria passar de volta uma indicação de erro como dados normais na chamada a `sendmsg`.

A Figura 15.13 mostra a função final, `write_fd`, que chama `sendmsg` para enviar um descritor (e dados opcionais, que não utilizamos) por um soquete de domínio Unix.

lib/write_fd.c

```
 1 #include   "unp.h"

 2 ssize_t
 3 write_fd(int fd, void *ptr, size_t nbytes, int sendfd)
 4 {
 5     struct msghdr msg;
 6     struct iovec iov[1];

 7 #ifdef HAVE_MSGHDR_MSG_CONTROL
 8     union {
 9         struct cmsghdr cm;
10         char  control[CMSG_SPACE(sizeof(int))];
11     } control_un;
12     struct cmsghdr *cmptr;

13     msg.msg_control = control_un.control;
14     msg.msg_controllen = sizeof(control_un.control);

15     cmptr = CMSG_FIRSTHDR(&msg);
16     cmptr->cmsg_len = CMSG_LEN(sizeof(int));
17     cmptr->cmsg_level = SOL_SOCKET;
18     cmptr->cmsg_type = SCM_RIGHTS;
19     *((int *) CMSG_DATA(cmptr)) = sendfd;
20 #else
21     msg.msg_accrights = (caddr_t) & sendfd;
22     msg.msg_accrightslen = sizeof(int);
23 #endif

24     msg.msg_name = NULL;
25     msg.msg_namelen = 0;

26     iov[0].iov_base = ptr;
27     iov[0].iov_len = nbytes;
28     msg.msg_iov = iov;
29     msg.msg_iovlen = 1;

30     return (sendmsg(fd, &msg, 0));
31 }
```

lib/write_fd.c

Figura 15.13 Função `write_fd`: passa um descritor chamando `sendmsg`.

Como ocorre com `read_fd`, essa função deve lidar com dados auxiliares ou direitos de acesso mais antigos. Em qualquer caso, a estrutura `msghdr` é inicializada e então `sendmsg` é chamado.

Mostraremos um exemplo de passagem de descritor na Seção 28.7 que envolve processos não-relacionados. Além disso, mostraremos um exemplo na Seção 30.9 que envolve processos relacionados. Utilizaremos as funções `read_fd` e `write_fd` que acabamos de descrever.

15.8 Recebendo credenciais do emissor

Na Figura 14.13, mostramos um outro tipo de dados que podem ser passados ao longo de um soquete de domínio Unix como dados auxiliares: credenciais de usuário. A forma exata de como as credenciais são empacotadas e enviadas como dados auxiliares tende a ser específica de SO. Aqui, descrevemos o FreeBSD e as outras variantes do Unix são semelhantes (normalmente, o desafio é determinar qual estrutura utilizar nas credenciais). Descrevemos esse recurso, apesar de ele não ser uniforme em todos os sistemas, porque é uma adição importante, ainda que simples, para os protocolos de domínio Unix.

Quando um cliente e um servidor comunicam-se utilizando esses protocolos, o servidor freqüentemente precisa saber com exatidão quem é o cliente, para validar o fato de que o cliente tem permissão para requerer o serviço sendo solicitado.

O FreeBSD passa credenciais em uma estrutura `cmsgcred`, que é definida incluindo o cabeçalho `<sys/socket.h>`.

```
struct cmsgcred {
        pid_t   cmcred_pid;          /* PID do processo de envio */
        uid_t   cmcred_uid;          /* UID real do processo de envio */
        uid_t   cmcred_euid;         /* UID efetivo do processo de envio */
        gid_t   cmcred_gid;          /* GID real do processo de envio */
        short   cmcred_ngroups;      /* número de grupos */
        gid_t   cmcred_groups[CMGROUP_MAX];     /* grupos */
};
```

Normalmente, `CMGROUP_MAX` é 16. `cmcred_ngroups` sempre é pelo menos 1, com o primeiro elemento do array como o ID de grupo efetivo.

Essas informações sempre estão disponíveis em um soquete de domínio Unix, embora freqüentemente haja arranjos especiais que o emissor deve fazer para incluir as informações ao enviar e também arranjos especiais (por exemplo, opções de soquete) que o receptor deve fazer para obter as credenciais. No nosso sistema FreeBSD, o receptor não tem de fazer nada especial além de chamar `recvmsg` com um buffer auxiliar suficientemente grande para armazenar as credenciais, como mostrado na Figura 15.14. O emissor, entretanto, deve incluir uma estrutura `cmsgcred` ao enviar dados utilizando `sendmsg`. É importante observar que, embora o FreeBSD requeira que o emissor inclua a estrutura, o conteúdo é preenchido pelo kernel e não pode ser forjado pelo emissor. Isso torna a passagem de credenciais em um soquete de domínio Unix uma maneira confiável de verificar a identidade do cliente.

Exemplo

Como um exemplo da passagem de credencial, modificamos nosso servidor de fluxo de domínio Unix para solicitar as credenciais do cliente. A Figura 15.14 mostra uma nova função, `read_cred`, que é semelhante a `read`, mas também retorna uma estrutura `cmsgcred` que contém as credenciais do emissor.

3–4 Os três primeiros argumentos são idênticos para `read`, com o quarto argumento sendo um ponteiro para uma estrutura `cmsgcred` que será preenchida.

22–31 Se as credenciais retornarem, o comprimento, o nível e o tipo dos dados auxiliares são verificados e a estrutura resultante é copiada de volta para o chamador. Se nenhuma credencial retornar, configuramos a estrutura como 0. Uma vez que o número de grupos (`cmcred_ngroups`)

sempre será 1 ou mais, o valor de 0 indica ao chamador que nenhuma credencial foi retornada pelo kernel.

A função `main` para nosso servidor de eco, Figura 15.3, permanece inalterada. A Figura 15.15 mostra a nova versão da função `str_echo`, modificada da Figura 5.3. Essa função é chamada pelo filho depois que o pai aceitou uma nova conexão cliente e chamou `fork`.

11–23 Se as credenciais retornarem, elas são impressas.

24–25 O restante do loop permanece inalterado. Esse código lê os buffers a partir do cliente e grava-os de volta no cliente.

Nosso cliente na Figura 15.4 é alterado minimamente para passar uma estrutura `cmsgcred` vazia que será preenchida quando ele chama `sendmsg`.

unixdomain/readcred.c

```
 1 #include  "unp.h"
 2 #define CONTROL_LEN (sizeof(struct cmsghdr) + sizeof(struct cmsgcred))
 3 ssize_t
 4 read_cred(int fd, void *ptr, size_t nbytes, struct cmsgcred *cmsgcredptr)
 5 {
 6       struct msghdr msg;
 7       struct iovec iov[1];
 8       char control[CONTROL_LEN];
 9       int     n;
10       msg.msg_name = NULL;
11       msg.msg_namelen = 0;
12       iov[0].iov_base = ptr;
13       iov[0].iov_len = nbytes;
14       msg.msg_iov = iov;
15       msg.msg_iovlen = 1;
16       msg.msg_control = control;
17       msg.msg_controllen = sizeof(control);
18       msg.msg_flags = 0;
19       if ( (n = recvmsg(fd, &msg, 0)) < 0)
20         return (n);
21       cmsgcredptr->cmcred_ngroups = 0;    /* indica que nenhuma credencial
                                                retornou */
22       if (cmsgcredptr && msg.msg_controllen > 0) {
23         struct cmsghdr *cmptr = (struct cmsghdr *) control;
24       if (cmptr->cmsg_len < CONTROL_LEN)
25         err_quit("control length = %d", cmptr->cmsg_len);
26       if (cmptr->cmsg_level != SOL_SOCKET)
27         err_quit("control level != SOL_SOCKET");
28       if (cmptr->cmsg_type != SCM_CREDS)
29         err_quit("control type != SCM_CREDS");
30       memcpy(cmsgcredptr, CMSG_DATA(cmptr), sizeof(struct cmsgcred));
31       }
32       return (n);
33 }
```

unixdomain/readcred.c

Figura 15.14 Função `read_cred`: lê e retorna as credenciais do emissor.

unixdomain/strecho.c

```
 1 #include   "unp.h"
 2 ssize_t read_cred(int, void *, size_t, struct cmsgcred *);
 3 void
 4 str_echo(int sockfd)
 5 {
 6     ssize_t n;
 7     int    i;
 8     char   buf[MAXLINE];
 9     struct cmsgcred cred;
10   again:
11     while ( (n = read_cred(sockfd, buf, MAXLINE, &cred)) > 0) {
12        if (cred.cmcred_ngroups == 0) {
13            printf("(no credentials returned)\n");
14        } else {
15            printf("PID of sender = %d\n", cred.cmcred_pid);
16            printf("real user ID = %d\n", cred.cmcred_uid);
17            printf("real group ID = %d\n", cred.cmcred_gid);
18            printf("effective user ID = %d\n", cred.cmcred_euid);
19            printf("%d groups:", cred.cmcred_ngroups - 1);
20            for (i = 1; i < cred.cmcred_ngroups; i++)
21                printf(" %d", cred.cmcred_groups[i]);
22            printf("\n");
23        }
24        Writen(sockfd, buf, n);
25     }
26     if (n < 0 && errno == EINTR)
27        goto again;
28     else if (n < 0)
29        err_sys("str_echo: read error");
30 }
```

unixdomain/strecho.c

Figura 15.15 Função `str_echo`: solicita as credenciais do cliente.

Antes de executar o cliente, podemos ver nossas credenciais atuais utilizando o comando `id`.

```
freebsd % id
uid=1007(andy) gid=1007(andy) groups=1007(andy), 0(wheel)
```

Iniciar o servidor e então executar o cliente uma vez em uma outra janela produz a seguinte saída do servidor:

```
freebsd % unixstrserv02
PID of sender = 26881
real user ID = 1007
real group ID = 1007
ID efetivo de usuário = 1007
2 groups: 1007 0
```

Essas informações são produzidas somente depois que o cliente enviou os dados ao servidor. Vemos que as informações correspondem com o que vimos no comando `id`.

15.9 Resumo

Os soquetes de domínio Unix são uma alternativa ao IPC quando o cliente e o servidor estão no mesmo host. A vantagem da utilização dos soquetes de domínio Unix em relação a algumas formas de IPC é que a API é quase idêntica a um cliente/servidor em rede. A vantagem da utilização dos soquetes de domínio Unix em relação ao TCP, quando o cliente e o servidor estão no mesmo host, é o aumento de desempenho para os soquetes de domínio Unix em relação a muitas implementações TCP.

Modificamos nossos clientes e servidores de eco TCP e UDP para utilizar os protocolos do domínio Unix e a única diferença importante foi ter de chamar `bind` para vincular um nome de caminho ao soquete do cliente UDP, de modo que o servidor UDP tivesse algum lugar para enviar as respostas.

A passagem de descritor é uma técnica poderosa entre clientes e servidores no mesmo host e acontece em um soquete de domínio Unix. Mostramos um exemplo na Seção 15.7 que passou um descritor a partir de um filho de volta para o pai. Na Seção 28.7, mostraremos um exemplo em que o cliente e o servidor não estão relacionados e, na Seção 30.9, um outro exemplo que passa um descritor a partir de um pai para um filho.

Exercícios

15.1 O que acontece se um servidor de domínio Unix chamar `unlink` depois de chamar `bind`?

15.2 O que acontece se um servidor de domínio Unix não chamar `unlink` para desvincular seu nome de caminho bem-conhecido quando ele termina e se um cliente chama `connect` para tentar conectar-se ao servidor depois que o servidor termina?

15.3 Inicie com a Figura 11.11 e a modifique para chamar `sleep(5)` depois que o endereço de protocolo do peer for impresso e também para imprimir o número de bytes retornado por `read` toda vez que `read` retorna um valor positivo.

Inicie com a Figura 11.14 e a modifique para chamar `write` em cada byte do resultado que é enviado ao cliente. (Discutimos modificações semelhantes na solução do Exercício 1.5.) Execute o cliente e o servidor no mesmo host utilizando o TCP. Quantos bytes são lidos utilizando `read` pelo cliente?

Execute o cliente e o servidor no mesmo host utilizando um soquete de domínio Unix. Algo muda?

Agora chame `send` em vez de `write` no servidor e especifique o flag `MSG_EOR`. (Você precisa de uma implementação derivada do Berkeley para concluir esse exercício.) Execute o cliente e o servidor no mesmo host utilizando um soquete de domínio Unix. Algo muda?

15.4 Escreva um programa para determinar os valores mostrados na Figura 4.10. Uma abordagem é criar um pipe de fluxo e então bifurcar com `fork` em um pai e um filho. O pai entra em um loop `for`, incrementando o backlog de 0 a 14. Toda vez através do loop, o pai primeiro grava o valor do backlog no pipe de fluxo. O filho lê esse valor, cria um soquete ouvinte vinculado ao endereço de loopback e configura o backlog como esse valor. Em seguida, o filho grava no pipe de fluxo, apenas para dizer ao pai que ele está pronto. O pai então tenta o maior número de conexões possível, detectando quando ele atingiu o limite de backlog porque o `connect` bloqueia. O pai pode utilizar um `alarm` configurado como dois segundos para detectar o `connect` bloqueador. O filho nunca chama `accept` para deixar que o kernel enfileire as conexões provenientes do pai. Quando o `alarm` do pai expira, ele sabe pelo contador de loop qual `connect` atingiu o limite de backlog. O pai então fecha seus soquetes e grava no filho o próximo novo valor de backlog no pipe de fluxo. Quando o filho lê esse próximo valor, ele fecha seu soquete ouvinte e cria um novo, iniciando o procedimento novamente.

15.5 Verifique se omitir a chamada a `bind` na Figura 15.6 causa um erro no servidor.

Capítulo 16

E/S Não-Bloqueadora

16.1 Visão geral

Por default, soquetes são bloqueadores. Isso significa que, quando emitimos uma chamada de soquete que não possa ser completada imediatamente, nosso processo é colocado para dormir, esperando que a condição torne-se verdadeira. Podemos dividir as chamadas de soquete que podem bloquear em quatro categorias:

1. Operações de entrada – Essas incluem as funções `read`, `readv`, `recv`, `recvfrom` e `recvmsg`. Se chamarmos uma dessas funções de entrada em um soquete TCP bloqueador (o default) e se não houver nenhum dado disponível no buffer de recebimento do soquete, somos colocados para dormir até que algum dado chegue. Como o TCP é um fluxo de bytes, seremos acordados quando "algum" dado chegar: "algum" poderia ser um único byte de dados ou um segmento TCP completo de dados. Se quisermos esperar até que algum volume fixo de dados torne-se disponível, poderemos chamar nossa própria função `readn` (Figura 3.15) ou especificar o flag `MSG_WAITALL` (Figura 14.6).

   Como o UDP é um protocolo de datagrama, se o buffer de recebimento do soquete estiver vazio em um soquete UDP bloqueador, somos colocados para dormir até que um datagrama UDP chegue.

   Com um soquete não-bloqueador, se a operação de entrada não puder ser satisfeita (pelo menos um byte de dados para um soquete TCP ou um datagrama completo para um soquete UDP), teremos um retorno imediato com um erro de `EWOULDBLOCK`.

2. Operações de saída – Essas incluem as funções `write`, `writev`, `send`, `sendto` e `sendmsg`. Para um soquete TCP, afirmamos na Seção 2.11 que o kernel copia os dados do buffer da aplicação para o buffer de envio do soquete. Se não houver espaço no buffer de envio do soquete para um soquete bloqueador, o processo é colocado para dormir até que haja espaço.

   Com um soquete TCP não-bloqueador, se não houver absolutamente nenhum espaço no buffer de envio do soquete, retornaremos imediatamente com um erro de

EWOULDBLOCK. Se houve algum espaço no buffer de envio do soquete, o valor de retorno será o número de bytes que o kernel foi capaz de copiar para o buffer. (Isso é chamado *contagem curta.*)

Também dissemos na Seção 2.11 que não há nenhum buffer de envio de soquete UDP real. O kernel apenas copia os dados da aplicação e move-os para baixo da pilha, prefixando os cabeçalhos UDP e IP. Portanto, uma operação de saída em um soquete UDP bloqueador (o padrão) não será bloqueada pela mesma razão como em um soquete TCP, mas é possível que operações de saída sejam bloqueadas em alguns sistemas devido ao armazenamento em buffer e ao controle de fluxo que acontece dentro do código de rede no kernel.

3. Aceitar conexões entrantes – essa é a função `accept`. Se `accept` for chamada para um soquete bloqueador e uma nova conexão não estiver disponível, o processo será colocado para dormir.

   Se `accept` for chamado para um soquete não-bloqueador e uma nova conexão não estiver disponível, o erro EWOULDBLOCK é retornado.

4. Iniciar conexões de saída – Essa é a função `connect` para o TCP. (Lembre-se de que `connect` pode ser utilizada com o UDP, mas ela não faz com que uma conexão "real" seja estabelecida; apenas faz com que o kernel armazene o endereço IP e o número da porta do peer.) Mostramos na Seção 2.6 que o estabelecimento de uma conexão TCP envolve um handshake de três vias e que a função `connect` não retorna até que o cliente receba o ACK do seu SYN. Isso significa que estabelecer uma conexão TCP com `connect` sempre bloqueia o processo chamador ao menos pelo tempo correspondente ao RTT do servidor.

   Se `connect` for chamada em um soquete TCP não-bloqueador e a conexão não puder ser estabelecida imediatamente, o estabelecimento da conexão é iniciado (por exemplo, o primeiro pacote do handshake de três vias do TCP é enviado), mas o erro EINPROGRESS é retornado. Observe que esse erro difere do erro retornado nos três cenários anteriores. Também observe que algumas conexões podem ser estabelecidas de imediato, normalmente quando o servidor está no mesmo host do cliente. Assim, mesmo com uma `connect` não-bloqueadora, devemos estar preparados para que `connect` retorne com sucesso. Mostraremos um exemplo de uma `connect` não-bloqueadora na Seção 16.3.

   > Tradicionalmente, o System V retornava o erro EAGAIN a uma operação de E/S não-bloqueadora que não pudesse ser satisfeita, enquanto as implementações derivadas do Berkeley retornavam o erro EWOULDBLOCK. Devido a esse histórico, a especificação POSIX diz que, nesse caso, qualquer um desses erros pode retornar. Felizmente, a maioria dos sistemas atuais define esses dois códigos de erro da mesma maneira (verifique o cabeçalho `<sys/errno.h>` no seu sistema), portanto, não importa qual utilizamos. Neste texto, utilizamos EWOULDBLOCK.

A Seção 6.2 resumiu os diferentes modelos disponíveis para E/S e comparou a E/S não-bloqueadora com outros modelos. Neste capítulo, forneceremos exemplos dos quatro tipos de operações e desenvolveremos um novo tipo de cliente, semelhante a um cliente Web, que inicia múltiplas conexões TCP ao mesmo tempo utilizando uma `connect` não-bloqueadora.

16.2 Leituras e gravações não-bloqueadoras: função `str_cli` (revisitada)

Retornamos mais uma vez a nossa função `str_cli`, que discutimos nas Seções 5.5 e 6.4. A última versão, com `select`, ainda utiliza uma E/S bloqueadora. Por exemplo, se uma linha estiver disponível na entrada-padrão, nós a lemos com `read` e então a enviamos ao servidor

com `writen`. Mas a chamada a `writen` pode bloquear se o buffer de envio do soquete estiver cheio. Enquanto estamos bloqueados na chamada a `writen`, os dados podem ser disponibilizados para leitura a partir do buffer de recebimento do soquete. De maneira semelhante, se uma linha de entrada estiver disponível no soquete, podemos bloquear na chamada subseqüente a `write`, se a saída-padrão for mais lenta do que a rede. Nosso objetivo nesta seção é desenvolver uma versão dessa função que utiliza uma E/S não-bloqueadora. Isso evita bloquear enquanto poderíamos estar fazendo algo produtivo.

Infelizmente, a adição de uma E/S não-bloqueadora complica o gerenciamento de buffer da função de maneira notável, assim apresentaremos essa função em partes. Como discutimos no Capítulo 6, utilizar uma E/S-padrão com soquetes pode ser difícil e esse é exatamente o caso com uma E/S não-bloqueadora. Portanto, continuaremos a evitar uma E/S-padrão nesse exemplo.

Mantemos dois buffers: `to` contém os dados indo da entrada-padrão para o servidor e `fr` contém os dados que chegam do servidor indo para a saída-padrão. A Figura 16.1 mostra o arranjo do buffer `to` e os ponteiros para o buffer.

O ponteiro `toiptr` aponta para o próximo byte do qual os dados podem ser lidos a partir da entrada-padrão. `tooptr` aponta para o próximo byte que deve ser gravado no soquete. Há `toiptr` menos `tooptr` bytes a serem gravados no soquete. O número de bytes que pode ser lido da entrada padrão é `&to[MAXLINE]` menos `toiptr`. Assim que `tooptr` alcança `toiptr`, os dois ponteiros são reiniciados para o começo do buffer.

A Figura 16.2 mostra o arranjo correspondente do buffer `fr`.

A Figura 16.3 mostra a primeira parte da função.

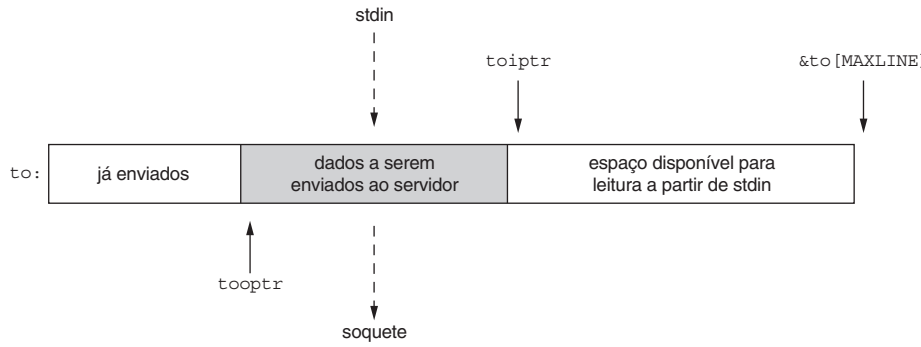


Figura 16.1 O buffer que contém os dados da entrada-padrão indo para o soquete.

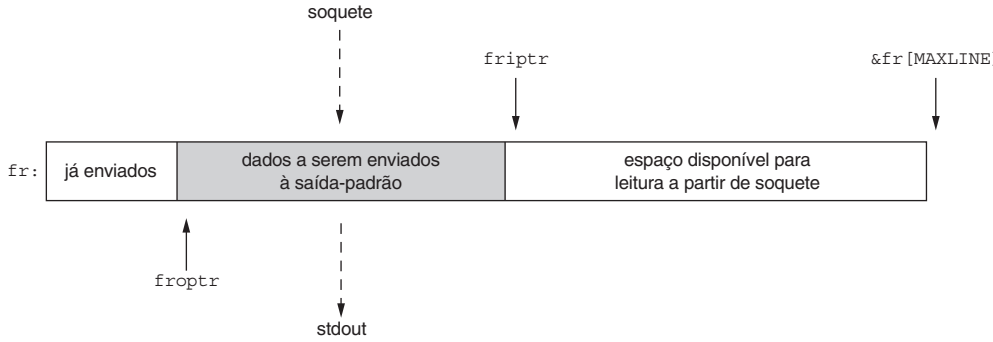


Figura 16.2 O buffer que contém os dados do soquete indo para a saída-padrão.

nonblock/strclinonb.c

```
 1 #include   "unp.h"

 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     int    maxfdp1, val, stdineof;
 6     ssize_t n, nwritten;
 7     fd_set rset, wset;
 8     char   to[MAXLINE], fr[MAXLINE];
 9     char   *toiptr, *tooptr, *friptr, *froptr;

10     val = Fcntl(sockfd, F_GETFL, 0);
11     Fcntl(sockfd, F_SETFL, val | O_NONBLOCK);

12     val = Fcntl(STDIN_FILENO, F_GETFL, 0);
13     Fcntl(STDIN_FILENO, F_SETFL, val | O_NONBLOCK);

14     val = Fcntl(STDOUT_FILENO, F_GETFL, 0);
15     Fcntl(STDOUT_FILENO, F_SETFL, val | O_NONBLOCK);

16     toiptr = tooptr = to;        /* inicializa os ponteiros de buffer */
17     friptr = froptr = fr;
18     stdineof = 0;

19     maxfdp1 = max(max(STDIN_FILENO, STDOUT_FILENO), sockfd) + 1;
20     for ( ; ; ) {
21        FD_ZERO(&rset);
22        FD_ZERO(&wset);
23        if (stdineof == 0 && toiptr < &to[MAXLINE])
24            FD_SET(STDIN_FILENO, &rset); /* lê de stdin */
25        if (friptr < &fr[MAXLINE])
26            FD_SET(sockfd, &rset); /* lê do soquete */
27        if (tooptr != toiptr)
28            FD_SET(sockfd, &wset); /* dados a serem gravados no soquete */
29        if (froptr != friptr)
30            FD_SET(STDOUT_FILENO, &wset); /* dados a serem gravados em stdout */

31        Select(maxfdp1, &rset, &wset, NULL, NULL);
```

nonblock/strclinonb.c

Figura 16.3 Função `str_cli`, primeira parte: inicializa e chama `select`.

Configuração dos descritores como não-bloqueadores

10–15 Todos os três descritores são configurados como não-bloqueadores utilizando `fcntl`: o soquete para e do servidor, a entrada-padrão e a saída-padrão.

Inicialização dos ponteiros de buffer

16–19 Os ponteiros nos dois buffers são inicializados e o descritor máximo mais um é calculado, o qual será utilizado como o primeiro argumento para `select`.

Loop principal: preparando para chamar `select`

20 Como ocorreu com a versão anterior dessa função, Figura 6.13, o loop principal é uma chamada a `select` seguida por testes individuais das várias condições em que estamos interessados.

Especificação dos descritores em que estamos interessados

21–30 Os dois conjuntos de descritores são configurados como 0 e até 2 bits e são então ativados em cada conjunto. Se ainda não tivermos lido um EOF na entrada-padrão e se houver espaço para ao menos um byte de dados no buffer `to`, o bit correspondente à entrada-padrão é ativado

no conjunto de leitura. Se houver espaço para ao menos um byte de dados no buffer `fr`, o bit correspondente ao soquete é ativado no conjunto de leitura. Se houver dados a gravar para o soquete no buffer `to`, o bit correspondente ao soquete é ativado no conjunto de gravação. Por fim, se houver dados no buffer `fr` a enviar à saída-padrão, o bit correspondente à saída-padrão é ativado no conjunto de gravação.

Chamada a `select`

31 `select` é chamada, esperando que uma das quatro possíveis condições seja verdadeira. Não especificamos um tempo-limite para essa função.

A próxima parte da função é mostrada na Figura 16.4. Esse código contém os primeiros dois testes (entre quatro) que ocorrem depois que `select` retorna.

Leitura (`read`) a partir da entrada-padrão

32–33 Se a entrada-padrão estiver legível, chamamos `read`. O terceiro argumento é a quantidade de espaço disponível no buffer `to`.

Tratando erro não-bloqueador

34–35 Se um erro ocorrer e for `EWOULDBLOCK`, nada acontece. Normalmente essa condição "não deve acontecer", isto é, `select` informa que o descritor está legível e `read` retorna `EWOULDBLOCK`, mas mesmo assim o tratamos.

`read` retorna o EOF

36–40 Se `read` retornar 0, a entrada-padrão está concluída. Nosso flag `stdineof` é ligado. Se não houver outros dados no buffer `to` a enviar (`tooptr` igual a `toiptr`), `shutdown` envia um FIN ao servidor. Se ainda houver dados no buffer `to` a enviar, o FIN não poderá ser enviado até que o buffer seja gravado no soquete.

> Geramos a saída de uma linha para o erro-padrão com uma observação sobre o EOF; juntamente com a data/hora atual, mostraremos como utilizar essa saída depois de descrever essa função. Chamadas semelhantes a `fprintf` são encontradas por toda essa função.

nonblock/strclinonb.c

```
32          if (FD_ISSET(STDIN_FILENO, &rset)) {
33              if ( (n = read(STDIN_FILENO, toiptr, &to[MAXLINE] - toiptr)) < 0) {
34                  if (errno != EWOULDBLOCK)
35                      err_sys("read error on stdin");
36              } else if (n == 0) {
37                  fprintf(stderr, "%s: EOF on stdin\n", gf_time());
38                  stdineof = 1;       /* stdin concluído */
39                  if (tooptr == toiptr)
40                      Shutdown(sockfd, SHUT_WR); /* envia FIN */
41              } else {
42                  fprintf(stderr, "%s: read %d bytes from stdin\n", gf_time(),
43                          n);
44                  toiptr += n; /* nº que acabou de ser lido */
45                  FD_SET(sockfd, &wset); /* tenta e grava no soquete abaixo */
46              }
47          }
48          if (FD_ISSET(sockfd, &rset)) {
49              if ( (n = read(sockfd, friptr, &fr[MAXLINE] - friptr)) < 0) {
50                  if (errno != EWOULDBLOCK)
51                      err_sys("read error on socket");
```

Figura 16.4 Função `str_cli`, segunda parte: lê da entrada ou soquete-padrão (*continua*).

```
52            } else if (n == 0) {
53                fprintf(stderr, "%s: EOF on socket\n", gf_time());
54                if(stdineof)
55                    return;     /* término normal */
56                else
57                    err_quit("str_cli: server terminated prematurely");
58            } else {
59                fprintf(stderr, "%s: read %d bytes from socket\n",
60                        gf_time(), n);
61                friptr += n; /* n° que acabou de ser lido */
62                FD_SET(STDOUT_FILENO, &wset); /* tenta e grava abaixo */
63            }
64        }
```

nonblock/strclinonb.c

Figura 16.4 Função `str_cli`, segunda parte: lê da entrada ou soquete-padrão (*continuação*).

`read` retorna os dados

41–45 Quando `read` retorna os dados, incrementamos `toiptr` de acordo com isso. Também ativamos o bit correspondente ao soquete no conjunto de gravação, para fazer com que o teste desse bit seja posteriormente verdadeiro no loop, fazendo, assim, com que um `write` seja tentado no soquete.

> Essa é uma das decisões difíceis de *design* ao escrever o código. Aqui, há algumas alternativas. Em vez de configurar o bit no conjunto de gravação, poderíamos não fazer nada, nesse caso, `select` testará a gravabilidade do soquete na próxima vez que é chamado. Mas isso requer um outro loop e uma outra chamada a `select` quando já sabemos que há dados a gravar no soquete. Outra escolha é duplicar o código que grava no soquete aqui, mas isso parece inútil e uma fonte potencial de erros (se houver um bug no fragmento desse código duplicado e o corrigirmos em um dos locais, mas não em outro). Por último, podemos criar uma função que grave no soquete e chamá-la em vez de duplicar o código, mas essa função precisa compartilhar três variáveis locais com `str_cli`, o que necessitaria tornar essas variáveis globais. A escolha feita é o ponto de vista destes autores sobre qual é a melhor alternativa.

Leitura (`read`) a partir do soquete

48–64 Essas linhas do código são semelhantes à instrução `if` que acabamos de descrever quando a entrada-padrão está legível. Se `read` retornar `EWOULDBLOCK`, nada acontece. Se encontrarmos um EOF do servidor, não há problemas se já tivermos encontrado um EOF na entrada-padrão, senão isso não é o esperado. Se `read` retornar alguns dados, `friptr` é incrementado e o bit da saída-padrão é ativado no conjunto de descritores de leitura, para tentar gravar os dados na próxima parte da função.

A Figura 16.5 mostra a parte final dessa função.

nonblock/strclinonb.c

```
65        if (FD_ISSET(STDOUT_FILENO, &wset) && ((n = friptr - froptr) > 0)) {
66            if ( (nwritten = write(STDOUT_FILENO, froptr, n)) < 0) {
67                if (errno != EWOULDBLOCK)
68                    err_sys("write error to stdout");
69            } else {
70                fprintf(stderr, "%s: wrote %d bytes to stdout\n",
71                        gf_time(), nwritten);
72                froptr += nwritten; /* n° que acabou de ser gravado */
73                if (froptr == friptr)
74                    froptr = friptr = fr;  /* volta ao começo do buffer */
```

Figura 16.5 Função `str_cli`, terceira parte: grava na saída-padrão ou soquete (*continua*).

```
75              }
76          }
77          if (FD_ISSET(sockfd, &wset) && ((n = toiptr - tooptr) > 0)) {
78              if ( (nwritten = write(sockfd, tooptr, n)) < 0) {
79                  if (errno != EWOULDBLOCK)
80                      err_sys("write error to socket");
81              } else {
82                  fprintf(stderr, "%s: wrote %d bytes to socket\n",
83                          gf_time(), nwritten);
84                  tooptr += nwritten; /* n° que acabou de ser gravado */
85                  if (tooptr == toiptr) {
86                      toiptr = tooptr = to;  /* volta ao começo do buffer */
87                      if(stdineof)
88                          Shutdown(sockfd, SHUT_WR);  /* envia FIN */
89                  }
90              }
91          }
92      }
93  }
```

nonblock/strclinonb.c

Figura 16.5 Função `str_cli`, terceira parte: grava na saída-padrão ou soquete (*continuação*).

Gravação (`write`) na saída-padrão

65–68 Se a saída-padrão for gravável e o número de bytes a gravar for maior que 0, `write` é chamado. Se `EWOULDBLOCK` retornar, nada acontece. Observe que essa condição é completamente possível porque o código no final da parte anterior dessa função ativa o bit de saída-padrão no conjunto de gravação, sem saber se `write` será ou não bem-sucedido.

`write` OK

69–75 Se `write` for bem-sucedido, `froptr` é incrementado pelo número de bytes gravado. Se o ponteiro de saída tiver alcançado o ponteiro de entrada, os dois ponteiros são reinicializados para apontar para o começo do buffer.

Gravação (`write`) no soquete

77–91 Essa seção do código é semelhante ao código que acabamos de descrever para gravar na saída-padrão. A diferença é que, quando o ponteiro de saída alcança o ponteiro de entrada, não apenas os dois ponteiros são reinicializados no começo do buffer, mas, se encontrarmos um EOF na entrada-padrão, também um FIN pode ser enviado ao servidor.

Agora, examinaremos a operação dessa função e a sobreposição da E/S não-bloqueadora. A Figura 16.6 mostra nossa função `gf_time`, chamada a partir da nossa função `str_cli`.

lib/gf_time.c

```
 1 #include   "unp.h"
 2 #include   <time.h>
 3 char *
 4 gf_time(void)
 5 {
 6     struct timeval tv;
 7     static char str[30];
 8     char   *ptr;
 9     if(gettimeofday(&tv, NULL) < 0)
10        err_sys("gettimeofday error");
```

Figura 16.6 Função `gf_time`: retorna um ponteiro para a string de data/hora (*continua*).

```
11      ptr = ctime(&tv.tv_sec);
12      strcpy(str, &ptr[11]);
13      /* Fri Sep 13 00:00:00 1986\n\0 */
14      /* 012345678901234567890123456 5 */
15      snprintf(str + 8, sizeof(str) - 8, ".%06ld", tv.tv_usec);
16      return (str);
17  }
```

lib/gf_time.c

Figura 16.6 Função `gf_time`: retorna um ponteiro para a string de data/hora (*continuação*).

Essa função retorna uma string que contém a data/hora atual, incluindo microssegundos, no seguinte formato:

```
12:34:56.123456
```

Esse formato é intencionalmente o mesmo da saída de registros de data/hora (*timestamps*) de `tcpdump`. Também observe que todas as chamadas a `fprintf` na nossa função `str_cli` são gravadas no erro-padrão, permitindo separar a saída-padrão (as linhas ecoadas pelo servidor) da nossa saída de diagnóstico. Podemos então executar nosso cliente e `tcpdump`, tomar essa saída de diagnóstico junto com a saída de `tcpdump` e classificar as duas saídas conjuntamente, ordenadas por data/hora. Isso permite verificar o que acontece no nosso programa e correlacioná-lo à ação TCP correspondente.

Por exemplo, primeiro executamos `tcpdump` no nosso host `solaris`, capturando somente os segmentos TCP para ou da porta 7 (o servidor de eco), salvando a saída no arquivo de nome `tcpd`.

```
solaris % tcpdump -w tcpd tcp and port 7
```

Em seguida, executamos nosso cliente TCP nesse host, especificando o servidor no host `linux`.

```
solaris % tcpcli02 192.168.1.10 < 2000.lines > out 2> diag
```

A entrada-padrão é o arquivo `2000.lines`, o mesmo utilizado na Figura 6.13. A saída-padrão é enviada ao arquivo `out` e o erro-padrão ao arquivo `diag`. Na conclusão do código, executamos

```
solaris % diff 2000.lines out
```

para verificar se as linhas ecoadas são idênticas às linhas de entrada. Por fim, terminamos `tcpdump` com nossa chave de interrupção e então imprimimos os registros de `tcpdump`, classificando-os com a saída de diagnóstico do cliente. A Figura 16.7 mostra a primeira parte desse resultado.

```
solaris % tcpdump -r tcpd -N | sort diag -
10:18:34.486392 solaris.33621 > linux.echo: S 1802738644:1802738644(0)
                                            win 8760 <mss 1460>
10:18:34.488278 linux.echo > solaris.33621: S 3212986316:3212986316(0)
                                            ack 1802738645 win 8760 <mss 1460>
10:18:34.488490 solaris.33621 > linux.echo: . ack 1 win 8760
10:18:34.491482: read 4096 bytes from stdin
10:18:34.518663 solaris.33621 > linux.echo: P 1:1461(1460) ack 1 win 8760
10:18:34.519016: wrote 4096 bytes to socket
10:18:34.528529 linux.echo > solaris.33621: P 1:1461(1460) ack 1461 win 8760
10:18:34.528785 solaris.33621 > linux.echo: . 1461:2921(1460) ack 1461 win 8760
10:18:34.528900 solaris.33621 > linux.echo: P 2921:4097(1176) ack 1461 win 8760
10:18:34.528958 solaris.33621 > linux.echo: . ack 1461 win 8760
10:18:34.536193 linux.echo > solaris.33621: . 1461:2921(1460) ack 4097 win 8760
```

Figura 16.7 Saída classificada a partir de `tcpdump` e saída de diagnóstico (*continua*).

```
10:18:34.536697 linux.echo > solaris.33621: P 2921:3509(588) ack 4097 win 8760
10:18:34.544636: read 4096 bytes from stdin
10:18:34.568505: read 3508 bytes from socket
10:18:34.580373 solaris.33621 > linux.echo: . ack 3509 win 8760
10:18:34.582244 linux.echo > solaris.33621: P 3509:4097(588) ack 4097 win 8760
10:18:34.593354: wrote 3508 bytes to stdout
10:18:34.617272 solaris.33621 > linux.echo: P 4097:5557(1460) ack 4097 win 8760
10:18:34.617610 solaris.33621 > linux.echo: P 5557:7017(1460) ack 4097 win 8760
10:18:34.617908 solaris.33621 > linux.echo: P 7017:8193(1176) ack 4097 win 8760
10:18:34.618062: wrote 4096 bytes to socket
10:18:34.623310 linux.echo > solaris.33621: . ack 8193 win 8760
10:18:34.626129 linux.echo > solaris.33621: . 4097:5557(1460) ack 8193 win 8760
10:18:34.626339 solaris.33621 > linux.echo: . ack 5557 win 8760
10:18:34.626611 linux.echo > solaris.33621: P 5557:6145(588) ack 8193 win 8760
10:18:34.628396 linux.echo > solaris.33621: . 6145:7605(1460) ack 8193 win 8760
10:18:34.643524: read 4096 bytes from stdin
10:18:34.667305: read 2636 bytes from socket
10:18:34.670324 solaris.33621 > linux.echo: . ack 7605 win 8760
10:18:34.672221 linux.echo > solaris.33621: P 7605:8193(588) ack 8193 win 8760
10:18:34.691039: wrote 2636 bytes to stdout
```

Figura 16.7 Saída classificada a partir de `tcpdump` e saída de diagnóstico (*continuação*).

Quebramos as linhas longas que contêm os SYNs e também removemos as notações "don't fragment (não fragmentar)" `(DF)` dos segmentos do Solaris, indicando que o bit do DF está configurado (descoberta do MTU do caminho).

Utilizando essa saída, podemos desenhar uma linha do tempo daquilo que está acontecendo. Mostramos isso na Figura 16.8, com a data/hora aumentando para baixo na página.

Nessa figura, não mostramos os segmentos ACK. Também perceba que, quando o programa gera "wrote *N* bytes to stdout" (gravados *N* bytes em stdout), a função `write` havia retornado, possivelmente fazendo com que o TCP enviasse um ou mais segmentos de dados.

O que podemos ver nessa linha do tempo é a dinâmica de uma troca cliente/servidor. Utilizar uma E/S não-bloqueadora deixa o programa tirar proveito dessa dinâmica, lendo ou gravando quando a operação pode acontecer. Deixamos o kernel informar quando uma operação de E/S pode ocorrer utilizando a função `select`.

Podemos medir o tempo da nossa versão não-bloqueadora utilizando o mesmo arquivo de 2.000 linhas e o mesmo servidor (um RTT de 175 ms no cliente), como na Seção 6.7. O tempo de clock agora é 6,9 segundos, se comparado com os 12,3 segundos da versão na Seção 6.7. Portanto, uma E/S não-bloqueadora reduz o tempo total desse exemplo que envia um arquivo ao servidor.

Uma versão mais simples de `str_cli`

A versão não-bloqueadora de `str_cli` que acabamos de mostrar é não-trivial: aproximadamente 135 linhas de código, comparado com 40 linhas da versão que utiliza `select` com uma E/S bloqueadora na Figura 6.13 e 20 linhas da nossa versão "pare e espere" original (Figura 5.5). Sabemos que dobrar o tamanho do código de 20 para 40 linhas valeu o esforço, porque a velocidade aumentou em quase 30% em um modo de lote e utilizar `select` com descritores bloqueadores não foi nem um pouco complicado. Mas, valeria o esforço de codificar uma aplicação utilizando uma E/S não-bloqueadora, dada a complexidade do código resultante? A resposta é não.

Sempre que temos necessidade de utilizar uma E/S não-bloqueadora, normalmente será mais simples dividir a aplicação em processos (utilizando `fork`) ou threads (Capítulo 26).

A Figura 16.10 é, todavia, uma outra versão da nossa função `str_cli`, com a função se autodividindo em dois processos utilizando `fork`.

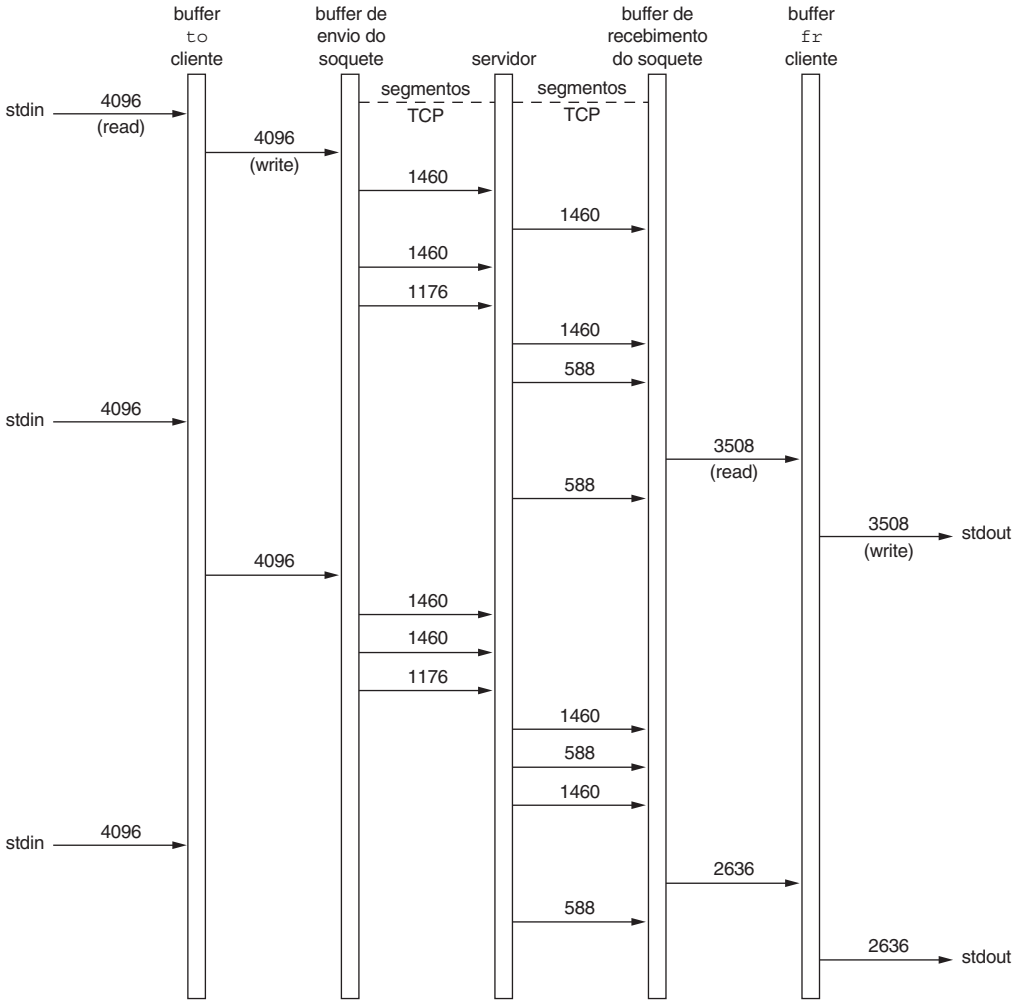


Figura 16.8 Linha de tempo do exemplo não-bloqueador.

A função chama `fork` imediatamente para dividir-se em um pai e um filho. O filho copia as linhas do servidor para a saída-padrão e o pai copia as linhas da entrada-padrão para o servidor, como mostrado na Figura 16.9.

Observamos explicitamente que a conexão TCP é full-duplex e que o pai e o filho estão compartilhando o mesmo descritor de soquete: o pai grava no soquete e o filho lê do soquete.

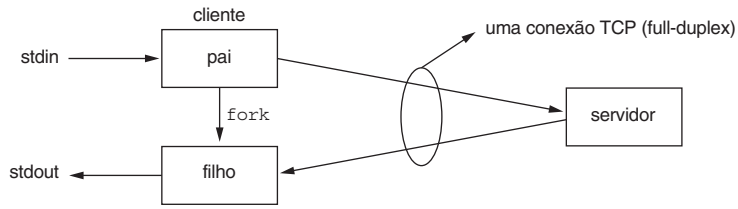


Figura 16.9 Função `str_cli` que utiliza dois processos.

Há somente um soquete, um buffer de recebimento do soquete e um buffer de envio do soquete, mas esse soquete é referenciado por dois descritores: um no pai e outro no filho.

Precisamos novamente nos preocupar com a seqüência de término. O término normal ocorre quando é encontrado o EOF na entrada-padrão. O pai lê esse EOF e chama `shutdown` para enviar um FIN. (O pai não pode chamar `close`. Veja o Exercício 16.1.) Mas, quando isso acontece, o filho precisa continuar copiando do servidor para a saída-padrão, até que ele leia um EOF no soquete.

Também é possível que o processo servidor termine prematuramente (Seção 5.12); se isso ocorrer, o filho lerá um EOF no soquete. Se isso acontecer, o filho deve informar o pai a parar de copiar da entrada-padrão para o soquete (veja o Exercício 16.2). Na Figura 16.10, o filho envia o sinal `SIGTERM` ao pai, se este ainda estiver em execução (veja o Exercício 16.3). Uma outra maneira de tratar isso seria o filho terminar e fazer com que o pai capture `SIGCHLD`, se o pai ainda estiver em execução.

nonblock/strclifork.c

```
 1 #include   "unp.h"
 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     pid_t  pid;
 6     char   sendline[MAXLINE], recvline[MAXLINE];
 7     if ( (pid = Fork()) == 0) { /* filho: servidor -> stdout */
 8         while (Readline(sockfd, recvline, MAXLINE) > 0)
 9             Fputs(recvline, stdout);
10         kill(getppid(), SIGTERM);  /* se o pai ainda estiver em execução */
11         exit(0);
12     }
13     /* pai: stdin -> servidor */
14     while (Fgets(sendline, MAXLINE, fp) != NULL)
15         Writen(sockfd, sendline, strlen(sendline));
16     Shutdown(sockfd, SHUT_WR); /* EOF em stdin, envia um FIN */
17     pause();
18     return;
19 }
```

nonblock/strclifork.c

Figura 16.10 A versão da função `str_cli` que utiliza `fork`.

O pai chama `pause` ao terminar de copiar, o que o coloca para dormir até que um sinal seja capturado. Mesmo que o pai não capture nenhum sinal, isso o coloca para dormir até que ele recebe o sinal `SIGTERM` do filho. A ação-padrão desse sinal é terminar o processo, o que é bom para esse exemplo. A razão por que fazemos o pai esperar o filho é para medir um tempo de clock preciso para essa versão de `str_cli`. Normalmente, o filho conclui depois do pai, mas, como medimos o tempo de clock utilizando o comando `time` do shell, a medição acaba quando o pai termina.

Observe a simplicidade dessa versão se comparada à versão E/S não-bloqueadora mostrada anteriormente nesta seção. Nossa versão não-bloqueadora gerenciou quatro fluxos diferentes de E/S ao mesmo tempo e, como todos eram não-bloqueadores, temos de nos preocupar com leituras e gravações parciais em todos os quatro fluxos. Mas, na versão `fork`, cada processo trata somente dois fluxos de E/S, copiando de um para o outro. Não há necessidade de uma E/S não-bloqueadora porque, se não houver nenhum dado a ler no fluxo de entrada, não haverá nada a gravar no fluxo de saída correspondente.

Temporização de `str_cli`

Agora, mostramos quatro versões diferentes da função `str_cli`. Resumimos o tempo de clock requerido para essas versões, juntamente com uma versão que utiliza threads (Figura 26.2), ao copiar 2.000 linhas de um cliente Solaris para um servidor com um RTT de 175 ms:

- 354,0 seg, pare e espere (Figura 5.5)
- 12,3 seg, `select` e E/S bloqueadora (Figura 6.13)
- 6,9 seg, E/S não-bloqueadora (Figura 16.3)
- 8,7 seg, `fork` (Figura 16.10)
- 8,5 seg, versão com thread (Figura 26.2)

Nossa versão E/S não-bloqueadora é quase duas vezes mais rápida que a nossa versão que utiliza uma E/S bloqueadora com `select`. Nossa versão simples que utiliza `fork` é mais lenta que a nossa versão E/S não-bloqueadora. Contudo, dada a complexidade do código de uma E/S não-bloqueadora *versus* o código de `fork`, recomendamos a abordagem simples.

16.3 `connect` não-bloqueadora

Quando um soquete TCP é configurado como não-bloqueador e então `connect` é chamada, `connect` retorna imediatamente com um erro de `EINPROGRESS`, mas o handshake de três vias do TCP continua. Em seguida, verificamos uma conclusão bem ou mal-sucedida do estabelecimento da conexão utilizando `select`. Há três utilizações para uma `connect` não-bloqueadora:

1. Podemos sobrepor outro processamento com o handshake de três vias. Uma `connect` leva um RTT para completar (Seção 2.6) e isso pode ser alguns milissegundos em uma rede local, centenas de milissegundos ou alguns segundos em uma rede geograficamente distribuída (WAN). Poderá haver outro processamento que desejamos realizar desta vez.
2. Podemos estabelecer múltiplas conexões ao mesmo tempo utilizando essa técnica. Essa técnica tornou-se popular com os navegadores Web e mostraremos um exemplo dela na Seção 16.5.
3. Uma vez que esperamos que a conexão seja estabelecida utilizando `select`, podemos especificar um limite de tempo para `select`, permitindo reduzir o tempo-limite para `connect`. Muitas implementações têm um tempo-limite para `connect` entre 75 segundos e vários minutos. Há momentos em que uma aplicação quer um tempo-limite mais curto e utilizar uma `connect` não-bloqueadora é uma das maneiras de realizar isso. A Seção 14.2 discute outras maneiras de determinar tempos-limite nas operações de soquete.

Por mais simples que a `connect` não-bloqueadora pareça ser, há outros detalhes que devemos tratar:

- Ainda que o soquete seja não-bloqueador, se o servidor ao qual estamos nos conectando estiver no mesmo host, a conexão costuma ser estabelecida imediatamente quando chamamos `connect`. Devemos tratar esse cenário.
- Implementações derivadas do Berkeley (e do POSIX) têm as duas regras a seguir com relação a `select` e conexões com `connect` não-bloqueadoras:

  1. Quando a conexão completa com sucesso, o descritor torna-se gravável (página 531 do TCPv2).

2. Quando o estabelecimento da conexão encontra um erro, o descritor torna-se tanto legível como gravável (página 530 do TCPv2).

> Essas duas regras com relação a `select` são opostas às nossas regras na Seção 6.3 sobre as condições que tornam um descritor pronto. Um soquete TCP é gravável se houver espaço disponível no buffer de envio (o que sempre será o caso para um soquete conector, visto que ainda não gravamos qualquer coisa no soquete) *e* o soquete estiver conectado (o que ocorre somente quando o handshake de três vias é concluído). Um erro pendente faz com que um soquete torne-se tanto legível como gravável.

Há muitos problemas quanto à portabilidade com conexões a `connect` não-bloqueadoras que mencionaremos nos exemplos a seguir.

16.4 `connect` não-bloqueadora: cliente de data/hora

A Figura 16.11 mostra nossa função `connect_nonb` que realiza uma `connect` não-bloqueadora. Substituímos a chamada a `connect` na Figura 1.5 por

```
if (connect_nonb(sockfd, (SA *) &servaddr, sizeof(servaddr), 0) < 0)
   err_sys("connect error");
```

Os três primeiros argumentos são os normais para `connect` e o quarto é o número de segundos a esperar que a conexão complete. Um valor de 0 implica em nenhum tempo-limite em `select`; conseqüentemente, o kernel utilizará um tempo-limite normal para o estabelecimento da conexão TCP.

Configuração de um soquete não-bloqueador

9–10 Chamamos `fcntl` para configurar o soquete como não-bloqueador.

11–14 Iniciamos a `connect` não-bloqueadora. O erro que esperamos é `EINPROGRESS`, indicando que a conexão iniciou, mas ainda não está completada (página 466 do TCPv2). Qualquer outro erro é retornado ao chamador.

Sobrepondo o processamento ao estabelecimento da conexão

15 Nesse ponto, podemos fazer qualquer coisa que queremos enquanto esperamos que a conexão complete.

Verificação da conclusão imediata

16–17 Se a `connect` não-bloqueadora retornar 0, a conexão estará concluída. Como dissemos, isso pode ocorrer quando o servidor está no mesmo host do cliente.

Chamada a `select`

18–24 Chamamos `select` e esperamos que o soquete esteja pronto para leitura ou gravação. Zeramos `rset`, ativamos o bit correspondente a `sockfd` nesse conjunto de descritor e então copiamos `rset` para `wset`. Essa atribuição provavelmente é de estrutura, visto que os conjuntos de descritores costumam ser representados como estruturas. Também inicializamos a estrutura `timeval` e então chamamos `select`. Se o chamador especificar um quarto argumento em 0 (utiliza o tempo-limite padrão), deveremos especificar um ponteiro nulo como o argumento final para `select` e não uma estrutura `timeval` com um valor 0 (o que significa absolutamente nenhuma espera).

Tratando os tempos-limite

25–28 Se `select` retornar 0, o timer expirou e retornamos `ETIMEDOUT` ao chamador. Também fechamos o soquete, impedindo que o handshake de três vias prossiga adiante.

lib/connect_nonb.c

```
 1 #include   "unp.h"

 2 int
 3 connect_nonb(int sockfd, const SA *saptr, socklen_t salen, int nsec)
 4 {
 5     int    flags, n, error;
 6     socklen_t len;
 7     fd_set rset, wset;
 8     struct timeval tval;

 9     flags = Fcntl(sockfd, F_GETFL, 0);
10     Fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);

11     error = 0;
12     if ( (n = connect(sockfd, saptr, salen)) < 0)
13         if (errno != EINPROGRESS)
14             return (-1);

15     /* Faz o que quisermos enquanto connect está acontecendo. */

16     if (n == 0)
17         goto done;               /* connect concluído imediatamente */

18     FD_ZERO(&rset);
19     FD_SET(sockfd, &rset);
20     wset = rset;
21     tval.tv_sec = nsec;
22     tval.tv_usec = 0;

23     if ( (n = Select(sockfd + 1, &rset, &wset, NULL,
24                      nsec ? &tval : NULL)) == 0) {
25         close(sockfd);           /* tempo-limite */
26         errno = ETIMEDOUT;
27         return (-1);
28     }

29     if (FD_ISSET(sockfd, &rset) || FD_ISSET(sockfd, &wset)) {
30         len = sizeof(error);
31         if (getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &error, &len) < 0)
32             return (-1);         /* erro pendente do Solaris */
33     } else
34         err_quit("select error: sockfd not set");

35   done:
36     Fcntl(sockfd, F_SETFL, flags); /* restaura os flags de status do arquivo */

37     if (error) {
38         close(sockfd);           /* para qualquer eventualidade */
39         errno = error;
40         return (-1);
41     }
42     return (0);
43 }
```

lib/connect_nonb.c

Figura 16.11 Emissão de uma chamada a `connect` não-bloqueadora.

Verificação da legibilidade ou gravabilidade

29–34 Se o descritor for legível ou gravável, chamamos `getsockopt` para buscar o erro pendente do soquete (`SO_ERROR`). Se a conexão tiver sido completada com sucesso, esse valor será 0. Se a conexão encontrou um erro, esse valor será o valor de `errno` correspondente ao erro da conexão (por exemplo, `ECONNREFUSED`, `ETIMEDOUT`, etc.). Também encontramos nosso primeiro problema de portabilidade. Se um erro ocorreu, as implementações derivadas do

Berkeley de `getsockopt` retornam 0 com o erro pendente retornado na nossa variável `error`. Mas o Solaris faz com que o próprio `getsockopt` retorne –1 com `errno` configurada como o erro pendente. Nosso código trata os dois cenários.

Desativação do não-bloqueador e retorno

36–42 Restauramos os flags de *status* do arquivo e retornamos. Se nossa variável `error` for não-zero em `getsockopt`, esse valor será armazenado em `errno` e a função retorna –1.

Como dissemos anteriormente, há problemas de portabilidade com várias implementações de soquete e conexões com connect não-bloqueadoras. Primeiro, é possível que uma conexão complete e que dados cheguem de um peer antes de `select` ser chamada. Nesse caso, o soquete estará legível e gravável se bem-sucedido, da mesma forma como se a conexão tivesse falhado. Nosso código na Figura 16.11 trata esse cenário chamando `getsockopt` e verificando o erro pendente no soquete.

Em seguida, é determinar se a conexão foi ou não completada com sucesso, se não pudermos assumir que a gravabilidade é a única maneira de retornar com sucesso. Várias soluções foram postadas no Usenet. Essas soluções substituiriam nossa chamada a `getsockopt` na Figura 16.11.

1. Chame `getpeername` em vez de `getsockopt`. Se isso falhar com `ENOTCONN`, a conexão falhou e deveremos chamar `getsockopt` com `SO_ERROR` para buscar o erro pendente no soquete.
2. Chame `read` com um comprimento de 0. Se `read` falhar, a chamada a `connect` falhou e o `errno` da chamada a `read` indica a razão da falha da conexão. Quando uma conexão é bem-sucedida, `read` deve retornar 0.
3. Chame `connect` novamente. Essa chamada deve falhar e, se o erro for `EISCONN`, o soquete já está conectado e a primeira conexão foi bem-sucedida.

Infelizmente, conexões com `connect` não-bloqueadoras são uma das áreas mais não-portáveis da programação de rede. Esteja preparado para problemas de portabilidade, especialmente com implementações mais antigas. Uma técnica mais simples é criar um thread (Capítulo 26) para tratar uma conexão.

`connect` interrompida

O que acontece se nossa chamada a `connect` em um soquete bloqueador normal for interrompida, digamos, por um sinal capturado, antes de o handshake de três vias do TCP concluir? Supondo que `connect` não seja automaticamente reiniciada, ela retorna `EINTR`. Mas não podemos chamar `connect` novamente e esperar que a conexão complete. Fazer isso retornará `EADDRINUSE`.

O que devemos fazer nesse cenário é chamar `select`, assim como fizemos nesta seção para uma chamada a `connect` não-bloqueadora. `select` retorna quando a conexão é concluída com sucesso (tornando o soquete gravável) ou quando a conexão falha (tornando o soquete legível e gravável).

16.5 `connect` não-bloqueadora: cliente Web

Um exemplo prático de conexões com `connect` não-bloqueadoras começou com o cliente Web Netscape (Seção 13.4 do TCPv3). O cliente estabelece uma conexão HTTP com um servidor Web e busca uma home page. Freqüentemente, essa página terá várias referências a outras páginas da Web. Em vez de buscar essas outras páginas serialmente, uma por vez, o cliente pode buscar mais de uma ao mesmo tempo utilizando conexões com `connect` não-bloqueadoras. A Figura 16.12 mostra um exemplo do estabelecimento de múltiplas conexões em

paralelo. O cenário à esquerda mostra todas as três conexões realizadas serialmente. Assumimos que a primeira conexão demora 10 unidades de tempo; a segunda, 15; e a terceira, 4; totalizando 29 unidades de tempo.

No cenário intermediário, realizamos duas conexões em paralelo. No tempo 0, as duas primeiras conexões são iniciadas e, quando a primeira delas termina, iniciamos a terceira. O tempo total é quase reduzido à metade, de 29 a 15, mas perceba que esse é o caso ideal. Se as conexões paralelas estiverem compartilhando um link comum (digamos que o cliente está por trás de um modem discado de conexão à Internet), cada uma poderá competir com as demais pelos recursos limitados e todos os tempos das conexões individuais poderiam demorar mais. Por exemplo, o tempo de 10 poderia ser 15, o tempo de 15 poderia ser 20 e o tempo de 4 poderia ser 6. Contudo, o tempo total seria 21, ainda menor que o cenário serial.

No terceiro cenário, realizamos três conexões em paralelo e novamente assumimos que não há nenhuma interferência entre as três conexões (o caso ideal). Mas o tempo total permanece o mesmo (15 unidades), como no segundo cenário, dado os tempos de exemplo que escolhemos.

Ao lidar com clientes Web, a primeira conexão se realiza por si própria, seguida por múltiplas conexões para as referências encontradas nos dados dessa primeira conexão. Mostramos isso na Figura 16.13.

Para otimizar ainda mais essa seqüência, o cliente pode começar a analisar os dados que retornam à primeira conexão antes de ela completar e iniciar conexões adicionais logo que ele souber que são necessárias conexões adicionais.

Como estamos fazendo múltiplas conexões com `connect` não-bloqueadoras ao mesmo tempo, não podemos utilizar nossa função `connect_nonb` da Figura 16.11 porque ela não retorna até que a conexão seja estabelecida. Em vez disso, nós mesmos precisamos monitorar múltiplas conexões.

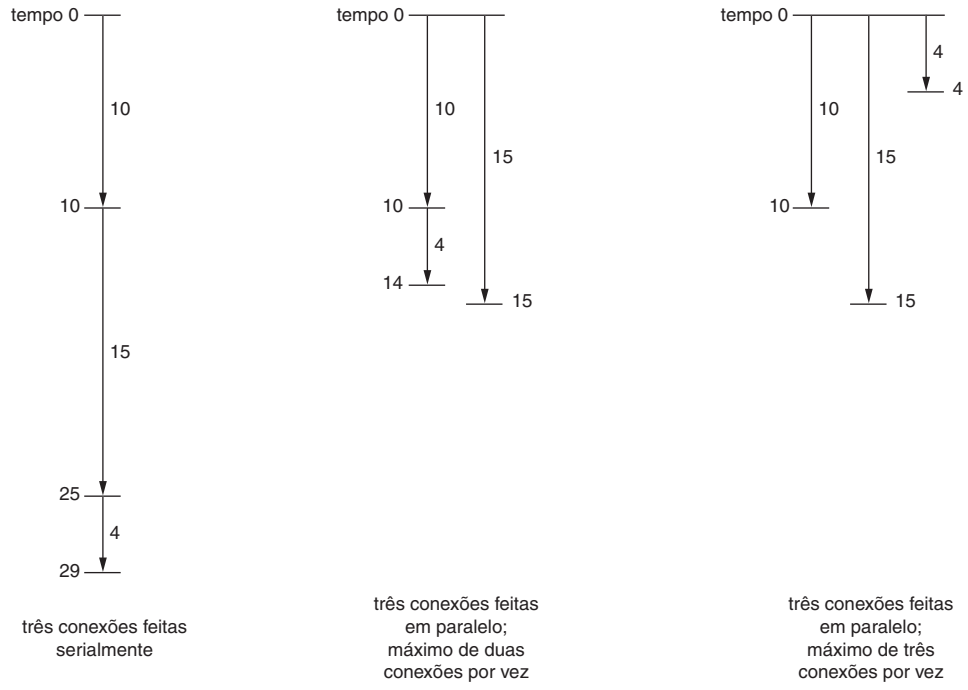


Figura 16.12 Estabelecendo múltiplas conexões em paralelo.

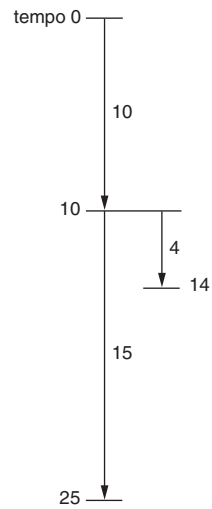


Figura 16.13 Completa a primeira conexão, então as múltiplas conexões em paralelo.

Nosso programa irá ler até 20 arquivos de um servidor Web. Especificamos como argumentos da linha de comando o número máximo de conexões paralelas, o hostname do servidor e cada um dos nomes de arquivo a serem buscados no servidor. Uma execução típica do nosso programa é

```
solaris % web 3 www.foobar.com / image1.gif image2.gif \
image3.gif image4.gif image5.gif \
image6.gif image7.gif
```

Os argumentos da linha de comando especificam três conexões simultâneas: o hostname do servidor, o nome de arquivo da home page (/, página root do servidor) e sete arquivos para então ler (que nesse exemplo são imagens GIF). Esses sete arquivos normalmente seriam referenciados na home page e um cliente Web iria ler a home page e analisar sintaticamente a HTML para obter esses nomes de arquivo. Não queremos complicar esse exemplo com análise sintática de HTML, portanto, simplesmente especificamos os nomes dos arquivos na linha de comando.

Este é um exemplo maior, assim o mostraremos em partes. A Figura 16.14 é o nosso cabeçalho `web.h` que cada arquivo inclui.

nonblock/web.h

```
 1 #include   "unp.h"

 2 #define MAXFILES  20
 3 #define SERV     "80"          /* número da porta ou nome de serviço */

 4 struct file {
 5     char   *f_name;            /* nome de arquivo */
 6     char   *f_host;            /* hostname ou endereço IPv4/IPv6 */
 7     int    f_fd;               /* descritor */
 8     int    f_flags;            /* F_xxx abaixo */
 9 } file[MAXFILES];

10 #define F_CONNECTING     1     /* connect() em progresso */
11 #define F_READING        2     /* connect() completa; agora lendo */
12 #define F_DONE           4     /* tudo concluído */
```

Figura 16.14 Cabeçalho `web.h` (*continua*).

```
13 #define GET_CMD      "GET %s HTTP/1.0\r\n\r\n"

14           /* globais */
15 int    nconn, nfiles, nlefttoconn, nlefttoread, maxfd;
16 fd_set rset, wset;

17           /* protótipos de função */
18 void   home_page(const char *, const char *);
19 void   start_connect(struct file *);
20 void   write_get_cmd(struct file *);
```

nonblock/web.h

Figura 16.14 Cabeçalho `web.h` (*continuação*).

Definição da estrutura `file`

2–13 O programa lê até `MAXFILES` arquivos a partir do servidor Web. Mantemos uma estrutura `file` com as informações sobre cada arquivo: o nome do arquivo (copiado do argumento da linha de comando), o hostname ou endereço IP do servidor para ler o arquivo, o descritor de soquete utilizado pelo arquivo e um conjunto de flags para especificar o que estamos fazendo com esse arquivo (conectar, ler ou concluir).

Definição dos protótipos de função e variáveis globais

14–20 Definimos as variáveis globais e os protótipos de função para as funções que descreveremos em breve.

A Figura 16.15 mostra a primeira parte do programa `main`.

nonblock/web.c

```
 1 #include   "web.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    i, fd, n, maxnconn, flags, error;
 6     char   buf[MAXLINE];
 7     fd_set rs, ws;

 8     if(argc < 5)
 9        err_quit("usage: web <#conns> <hostname> <homepage> <file1> ...");
10     maxnconn = atoi(argv[1]);

11     nfiles = min(argc - 4, MAXFILES);
12     for (i = 0; i < nfiles; i++) {
13         file[i].f_name = argv[i + 4];
14         file[i].f_host = argv[2];
15         file[i].f_flags = 0;
16     }
17     printf("nfiles = %d\n", nfiles);

18     home_page(argv[2], argv[3]);

19     FD_ZERO(&rset);
20     FD_ZERO(&wset);
21     maxfd = -1;
22     nlefttoread = nlefttoconn = nfiles;
23     nconn = 0;
```

nonblock/web.c

Figura 16.15 Primeira parte de conexões (`connect`) simultâneas: globais e início de `main`.

Processamento dos argumentos da linha de comando

11–17 As estruturas `file` são preenchidas com as informações relevantes provenientes dos argumentos da linha de comando.

Leitura da home page

18 A função `home_page`, que mostraremos em seguida, cria uma conexão TCP, envia um comando ao servidor e então lê a home page. Essa é a primeira conexão, realizada por si própria, antes de iniciarmos o estabelecimento de múltiplas conexões em paralelo.

Inicialização das globais

19–23 Dois conjuntos de descritores, um para ler e outro para gravar, são inicializados. `maxfd` é o descritor máximo para `select` (que inicializamos em –1, uma vez que os descritores são não-negativos), `nlefttoread` é o número de arquivos remanescente a ser lido (quando isso alcança 0, concluímos), `nlefttoconn` é o número de arquivos que ainda precisa de uma conexão TCP e `nconn` é o número de conexões atualmente abertas (que nunca pode exceder ao primeiro argumento da linha de comando).

A Figura 16.16 mostra a função `home_page` que é chamada uma vez quando a função `main` inicia.

nonblock/home_page.c

```
 1 #include   "web.h"

 2 void
 3 home_page(const char *host, const char *fname)
 4 {
 5     int    fd, n;
 6     char   line[MAXLINE];

 7     fd = Tcp_connect(host, SERV);        /* connect() bloqueadora */

 8     n = snprintf(line, sizeof(line), GET_CMD, fname);
 9     Writen(fd, line, n);

10     for ( ; ; ) {
11         if ( (n = Read(fd, line, MAXLINE)) == 0)
12             break;         /* servidor fechou a conexão */

13         printf("read %d bytes of home page\n", n);
14         /* faz qualquer coisa com os dados */
15     }
16     printf("end-of-file on home page\n");
17     Close(fd);
18 }
```

nonblock/home_page.c

Figura 16.16 Função `home_page`.

Estabelecimento de uma conexão com o servidor

7 Nosso `tcp_connect` estabelece uma conexão com o servidor.

Envio de um comando HTTP ao servidor, leitura da resposta

8–17 Um comando HTTP, `GET`, é emitido para a home page (freqüentemente identificado como `/`). A resposta é lida (não fazemos nada com a resposta) e a conexão é fechada.

A próxima função, `start_connect`, mostrada na Figura 16.17, inicia uma conexão com `connect` não-bloqueadora.

nonblock/start_connect.c

```
 1 #include   "web.h"

 2 void
 3 start_connect(struct file *fptr)
 4 {
 5     int    fd, flags, n;
 6     struct addrinfo *ai;

 7     ai = Host_serv(fptr->f_host, SERV, 0, SOCK_STREAM);

 8     fd = Socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
 9     fptr->f_fd = fd;
10     printf("start_connect for %s, fd %d\n", fptr->f_name, fd);

11         /* Configura o soquete não-bloqueador */
12     flags = Fcntl(fd, F_GETFL, 0);
13     Fcntl(fd, F_SETFL, flags | O_NONBLOCK);

14         /* Inicia uma connect não-bloqueadora para o servidor. */
15     if ( (n = connect(fd, ai->ai_addr, ai->ai_addrlen)) < 0) {
16         if (errno != EINPROGRESS)
17             err_sys("nonblocking connect error");
18         fptr->f_flags = F_CONNECTING;
19         FD_SET(fd, &rset);        /* select para leitura e gravação */
20         FD_SET(fd, &wset);
21         if (fd > maxfd)
22             maxfd = fd;

23     } else if (n >= 0)        /* connect já foi completada */
24         write_get_cmd(fptr);  /* chama write() para escrever o comando GET */
25 }
```

nonblock/start_connect.c

Figura 16.17 Iniciando uma `connect` não-bloqueadora.

Criação do soquete, configuração como não-bloqueador

7–13 Chamamos nossa função `host_serv` (Figura 11.9) para pesquisar e converter o hostname e o nome de serviço, retornando um ponteiro a um array das estruturas `addrinfo`. Utilizamos somente a primeira estrutura. Um soquete TCP é criado e configurado como não-bloqueador.

Início de uma `connect` não-bloqueadora

14–22 A chamada a `connect` não-bloqueadora é iniciada e o flag do arquivo é configurado como `F_CONNECTING`. O descritor de soquete é ativado tanto no conjunto de leitura como no de gravação, uma vez que `select` irá esperar uma condição como uma indicação de que a conexão terminou. Também atualizamos `maxfd`, se necessário.

Tratando a conclusão da conexão

23–24 Se `connect` retornar com sucesso, a conexão já foi completada e a função `write_get_cmd` (mostrada a seguir) envia um comando ao servidor.

Configuramos o soquete como não-bloqueador para `connect`, mas nunca o redefinimos como seu modo default bloqueador. Isso é bom porque gravamos somente uma pequena quantidade de dados no soquete (o comando `GET` na próxima função) e assumimos que esse comando é bem menor que o buffer de envio do soquete. Mesmo se `write` retornar uma contagem curta por causa do flag não-bloqueador, nossa função `writen` trata isso. Deixar o soquete como não-bloqueador não tem nenhum efeito nas chamadas a `read` subseqüentes porque sempre chamamos `select` para esperar que o soquete torne-se legível.

A Figura 16.18 mostra a função `write_get_cmd`, que envia um comando `GET` de HTTP ao servidor.

nonblock/write_get_cmd.c

```
 1 #include   "web.h"

 2 void
 3 write_get_cmd(struct file *fptr)
 4 {
 5     int    n;
 6     char   line[MAXLINE];

 7     n = snprintf(line, sizeof(line), GET_CMD, fptr->f_name);
 8     Writen(fptr->f_fd, line, n);
 9     printf("wrote %d bytes for %s\n", n, fptr->f_name);

10     fptr->f_flags = F_READING;    /* limpa F_CONNECTING */

11     FD_SET(fptr->f_fd, &rset);    /* irá ler a resposta do servidor */
12     if (fptr->f_fd > maxfd)
13         maxfd = fptr->f_fd;
14 }
```

nonblock/write_get_cmd.c

Figura 16.18 Envio de um comando `GET` de HTTP ao servidor.

Construção e envio do comando

7–9 O comando é construído e gravado no soquete.

Configuração dos flags

10–13 O flag `F_READING` do arquivo é ligado, o que também limpa o flag `F_CONNECTING` (se ligado). Isso indica ao loop principal que esse descritor está pronto para entrada. O descritor também é ativado no conjunto de leitura e `maxfd` é atualizado, se necessário.

Agora, retornamos a nossa função `main` na Figura 16.19, escolhendo o que deixamos de fora na Figura 16.15. Esse é o loop principal do programa: contanto que haja outros arquivos a processar (`nlefttoread` é maior que 0), inicie uma outra conexão se possível e então utilize `select` em todos os descritores ativos, tratando tanto as conclusões de conexões não-bloqueadoras quanto a chegada dos dados.

Início de uma outra conexão, se possível

24–35 Se não estivermos no limite especificado das conexões simultâneas e houver conexões adicionais a estabelecer, encontre um arquivo que ainda não processamos (indicado por um `f_flags` em 0) e chame `start_connect` para iniciar a conexão. O número de conexões ativas é incrementado (`nconn`) e o número de conexões a ser ainda estabelecido é decrementado (`nlefttoconn`).

`select`: esperando que algo aconteça

36–37 `select` espera a legibilidade ou gravabilidade. Os descritores com uma `connect` não-bloqueadora em progresso serão ativados nos dois conjuntos, enquanto os descritores com uma conexão completada que estão esperando os dados do servidor serão ativados somente no conjunto de leitura.

Tratando todos os descritores prontos

39–55 Agora, processamos cada elemento no array das estruturas `file` para determinar quais descritores precisam ser processados. Se o flag `F_CONNECTING` está ligado e o descritor está

ativo no conjunto de leitura ou de gravação, a `connect` não-bloqueadora é concluída. Como descrevemos na Figura 16.11, chamamos `getsockopt` para buscar o erro pendente no soquete. Se esse valor for 0, a conexão é completada com sucesso. Nesse caso, desativamos o descritor no conjunto de gravação e chamamos `write_get_cmd` para enviar a solicitação HTTP ao servidor.

Verificando se o descritor tem dados

56–67 Se o flag `F_READING` estiver ligado e o descritor pronto para ler, chamamos `read`. Se a conexão foi fechada pela outra extremidade, fechamos o soquete, ligamos o flag `F_DONE`, desativamos o descritor no conjunto de leitura e decrementamos o número de conexões ativas e o número total de conexões a ser processado.

Há duas otimizações que não realizamos nesse exemplo (para não complicá-lo ainda mais). Primeiro, poderíamos terminar o loop `for` na Figura 16.19 quando concluímos o processamento do número de descritores que `select` informou como prontos. Em seguida, poderíamos diminuir o valor de `maxfd` quando possível, para fazer com que `select` não precise examinar os bits do descritor que não mais estão ligados. Como o número de descritores que esse código lida em um momento qualquer provavelmente é menor que 10 e não milhares, há dúvidas de que qualquer uma dessas otimizações vale as complicações adicionais.

nonblock/web.c

```
24    while (nlefttoread > 0) {
25        while (nconn < maxnconn && nlefttoconn > 0) {
26                /* localiza um arquivo para ler */
27            for (i = 0; i < nfiles; i++)
28                if (file[i].f_flags == 0)
29                    break;
30            if (i == nfiles)
31                err_quit("nlefttoconn = %d but nothing found", nlefttoconn);
32            start_connect(&file[i]);
33            nconn++;
34            nlefttoconn--;
35        }

36        rs = rset;
37        ws = wset;
38        n = Select(maxfd + 1, &rs, &ws, NULL, NULL);

39        for (i = 0; i < nfiles; i++) {
40            flags = file[i].f_flags;
41            if (flags == 0 || flags & F_DONE)
42                continue;
43            fd = file[i].f_fd;
44            if (flags & F_CONNECTING &&
45                (FD_ISSET(fd, &rs) || FD_ISSET(fd, &ws))) {
46                n = sizeof(error);
47                if (getsockopt(fd, SOL_SOCKET, SO_ERROR, &error, &n) < 0 ||
48                    error != 0) {
49                    err_ret("nonblocking connect failed for %s",
50                        file[i].f_name);
51                }
52                    /* conexão estabelecida */
53                printf("connection established for %s\n", file[i].f_name);
54                FD_CLR(fd, &wset); /* nenhum outro teste de gravabilidade */
55                write_get_cmd(&file[i]); /* chama write() para escrever o
                   comando GET*/
```

Figura 16.19 Loop principal da função `main` (*continua*).

```
56              } else if (flags & F_READING && FD_ISSET(fd, &rs)) {
57                  if ( (n = Read(fd, buf, sizeof(buf))) == 0) {
58                      printf("end-of-file on %s\n", file[i].f_name);
59                      Close(fd);
60                      file[i].f_flags = F_DONE;     /* limpa F_READING */
61                      FD_CLR(fd, &rset);
62                      nconn--;
63                      nlefttoread--;
64                  } else {
65                      printf("read %d bytes from %s\n", n, file[i].f_name);
66                  }
67              }
68          }
69      }
70      exit(0);
71  }
```

nonblock/web.c

Figura 16.19 Loop principal da função `main` (*continuação*).

Desempenho das conexões simultâneas

Qual é o ganho de desempenho ao estabelecer múltiplas conexões ao mesmo tempo? A Figura 16.20 mostra o tempo de clock requerido para buscar uma home page do servidor Web, seguido por nove arquivos de imagem desse servidor. O RTT para o servidor é cerca de 150 ms. O tamanho da home page era de 4.017 bytes e o tamanho médio dos nove arquivos de imagem era de 1.621 bytes. O tamanho do segmento TCP era de 512 bytes. Também incluímos nessa figura, para comparação, os valores para uma versão desse programa que desenvolveremos na Seção 26.9 utilizando threads.

A maior parte da melhoria é obtida com três conexões simultâneas (o tempo de clock é reduzido à metade) e o aumento de desempenho é bem menor com quatro ou mais conexões simultâneas.

> Fornecemos esse exemplo, que utiliza `connect` simultâneas, porque é um bom exemplo da utilização de uma E/S não-bloqueadora, além disso, seu impacto sobre o desempenho pode ser medido. Ele também é um recurso utilizado por uma aplicação Web popular, o navegador Netscape. Há armadilhas nessa técnica se houver algum congestionamento na rede. O Capítulo 21 do TCPv1 descreve os algoritmos de início lento e que visam a evitar congestionamento do TCP em detalhe. Quando múltiplas conexões são estabelecidas de um cliente para um servidor, não há nenhuma comunicação entre elas na camada TCP. Isto é, se uma conexão sofrer uma perda de pacote, as outras conexões ao mesmo servidor

nº de conexões simultâneas	Tempo de clock (segundos), não-bloqueadora	Tempo de clock (segundos), threads
1	6,0	6,3
2	4,1	4,2
3	3,0	3,1
4	2,8	3,0
5	2,5	2,7
6	2,4	2,5
7	2,3	2,3
8	2,2	2,3
9	2,0	2,2

Figura 16.20 Tempo de clock para vários números de conexões simultâneas.

não serão notificadas e é altamente provável que logo sofrerão perda de pacotes a menos que diminuam a velocidade. Essas conexões adicionais enviam mais pacotes em uma rede já congestionada. Essa técnica também aumenta a carga em um dado momento no servidor.

16.6 `accept` não-bloqueadora

Afirmamos no Capítulo 6 que um soquete ouvinte retorna como legível por `select` quando uma conexão completada está pronta para ser aceita por `accept`. Portanto, se utilizarmos `select` para esperar conexões entrantes, não precisaremos configurar o soquete ouvinte como não-bloqueador porque se `select` informar que a conexão está pronta, `accept` não deve bloquear.

Infelizmente, aqui há um problema de temporização que pode nos confundir (Gierth, 1996). Para ver esse problema, modificamos nosso cliente de eco TCP (Figura 5.4) para estabelecer a conexão e então enviar um RST ao servidor. A Figura 16.21 mostra essa nova versão.

nonblock/tcpcli03.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    sockfd;
 6     struct linger ling;
 7     struct sockaddr_in servaddr;
 8     if (argc != 2)
 9         err_quit("usage: tcpcli <IPaddress>");
10     sockfd = Socket(AF_INET, SOCK_STREAM, 0);
11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_port = htons(SERV_PORT);
14     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
15     Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
16     ling.l_onoff = 1;       /* faz com que o RST seja enviado no close() */
17     ling.l_linger = 0;
18     Setsockopt(sockfd, SOL_SOCKET, SO_LINGER, &ling, sizeof(ling));
19     Close(sockfd);
20     exit(0);
21 }
```

nonblock/tcpcli03.c

Figura 16.21 Cliente de eco TCP que cria a conexão e envia um RST.

Configuração da opção de soquete `SO_LINGER`

16–19 Depois que a conexão é estabelecida, configuramos a opção de soquete `SO_LINGER`, configurando o flag `l_onoff` como 1 e o tempo de `l_linger` como 0. Como afirmado na Seção 7.5, isso faz com que um RST seja enviado em um soquete TCP quando a conexão é fechada. Chamamos então `close` para fechar o soquete.

Em seguida, modificamos nosso servidor TCP nas Figuras 6.21 e 6.22 para pausar depois que `select` retorna que o soquete ouvinte está legível, mas antes de chamar `accept`. No código a seguir no início da Figura 6.22, as duas linhas precedidas por um sinal de adição são novas:

```
        if (FD_ISSET(listenfd, &rset)) {     /* nova conexão de cliente */
+           printf("listening socket readable\n");
+           sleep(5);
            clilen = sizeof(cliaddr);
            connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
```

O que estamos simulando aqui é um servidor ocupado que não pode chamar `accept` tão logo `select` retorna que o soquete ouvinte está legível. Normalmente, essa lentidão por parte do servidor não é um problema (de fato, essa é a razão por que uma fila de conexões completadas é mantida), mas, se combinada com o RST do cliente, depois que a conexão é estabelecida, podemos ter um problema.

Na Seção 5.11, observamos que, quando o cliente aborta a conexão antes de o servidor chamar `accept`, as implementações derivadas do Berkeley não retornam a conexão abortada ao servidor, enquanto outras implementações devem retornar `ECONNABORTED`, mas, em vez disso, freqüentemente retornam `EPROTO`. Considere o seguinte exemplo de uma implementação derivada do Berkeley:

- O cliente estabelece a conexão e então a aborta como na Figura 16.21.
- `select` retorna legível ao processo do servidor, mas o servidor leva algum tempo para chamar `accept`.
- Entre o retorno de `select` do servidor e sua chamada a `accept`, o RST é recebido do cliente.
- A conexão concluída é removida da fila e assumimos que não há nenhuma outra conexão completada.
- O servidor chama `accept`, mas como há conexões não-completadas, ele as bloqueia.

O servidor permanecerá bloqueado na chamada a `accept` até que algum outro cliente estabeleça uma conexão. Mas, nesse ínterim, assumindo um servidor como a Figura 6.22, o servidor é bloqueado na chamada a `accept` e não tratará nenhum outro descritor pronto.

> Esse problema é um pouco semelhante ao ataque de recusa de serviço descrito na Seção 6.8, mas com esse novo bug o servidor "escapa" do `accept` bloqueado logo que um outro cliente estabelece uma conexão.

A correção desse problema é a seguinte:

1. Sempre configure um soquete ouvinte como não-bloqueador ao utilizar `select` para indicar quando uma conexão está pronta para ser aceita por `accept`.
2. Ignore os erros a seguir na chamada subseqüente a `accept`: `EWOULDBLOCK` (para implementações derivadas do Berkeley, quando o cliente aborta a conexão), `ECONNABORTED` (para implementações POSIX, quando o cliente aborta a conexão), `EPROTO` (para implementações SVR4, quando o cliente aborta a conexão) e `EINTR` (se sinais estão sendo capturados).

16.7 Resumo

Nosso exemplo de chamadas a read e write não-bloqueadoras na Seção 16.2 selecionou nosso cliente de eco `str_cli` e modificou-o para utilizar uma E/S não-bloqueadora na conexão TCP ao servidor. `select` é normalmente utilizada com E/S não-bloqueadora para determinar quando um descritor está legível ou gravável. Essa versão do nosso cliente é a mais rápida que mostramos, embora as modificações no código sejam não-triviais. Em seguida, mostramos que é mais simples dividir o cliente em duas partes utilizando `fork`; empregaremos a mesma técnica utilizando threads na Figura 26.2.

Conexões com `connect` não-bloqueadoras permitem fazer outro processamento enquanto o handshake de três vias do TCP acontece, em vez de sermos bloqueados na chamada a `connect`. Infelizmente, essas chamadas também não são portáveis, com diferentes implementações tendo diferentes maneiras de indicar que a conexão completou com sucesso ou encontrou um erro. Utilizamos chamadas a `connect` não-bloqueadoras para desenvolver um novo cliente, semelhante a um cliente Web que abre múltiplas conexões TCP ao mesmo tempo para reduzir o tempo de clock requerido para buscar vários arquivos em um servidor. Iniciar múltiplas conexões como essa pode reduzir o tempo de clock, mas também é "não-amigável a redes" quanto a evitar congestionamento do TCP.

Exercícios

16.1 Na nossa discussão na Figura 16.10, mencionamos que o pai deve chamar `shutdown`, não `close`. Por quê?

16.2 O que acontece na Figura 16.10 se o processo do servidor terminar prematuramente, acrescido do fato de o filho receber o EOF e terminar, porém sem notificar o pai?

16.3 O que acontece na Figura 16.10 se o pai morrer inesperadamente antes do filho e se este então ler um EOF no soquete?

16.4 O que acontece na Figura 16.11 se removermos as duas linhas

```
if (n == 0)
    goto done;        /* connect concluído imediatamente */
```

16.5 Na Seção 16.3, dissemos que é possível que os dados cheguem a um soquete antes de connect retornar. Como isso pode acontecer?

Capítulo 17

Operações ioctl

17.1 Visão geral

Tradicionalmente, a função `ioctl` tem sido a interface de sistema utilizada para tudo que não se encaixa em alguma categoria bem definida. O POSIX está abolindo `ioctl` para certas funcionalidades por meio da criação de funções empacotadoras específicas para substituir funções `ioctl` cuja funcionalidade esteja sendo padronizada por ele. Por exemplo, a interface de terminal do Unix era tradicionalmente acessada utilizando `ioctl`, mas o POSIX criou 12 novas funções para terminais: `tcgetattr` para obter os atributos de terminal, `tcflush` para esvaziar a entrada ou saída pendente, e assim por diante. Em uma tendência semelhante, o POSIX substituiu uma das funções `ioctl` de rede: a nova função `sockatmark` (Seção 24.3) substitui a `ioctl SIOCATMARK`. Contudo, várias funções `ioctl` permanecem para recursos dependentes da implementação relacionados à programação de rede: por exemplo, obter informações sobre a interface e acessar a tabela de roteamento e cache ARP.

Este capítulo proporcionará uma visão geral das solicitações a `ioctl` relacionadas à programação de rede, mas muitas dessas solicitações são dependentes de implementação. Além disso, algumas implementações, incluindo sistemas derivados do 4.4BSD e Solaris 2.6 e superiores, utilizam soquetes no domínio `AF_ROUTE` (soquetes de roteamento) para realizar boa parte dessas operações. Discutiremos os soquetes de roteamento no Capítulo 18.

Uma utilização comum da `ioctl` por programas de rede (em geral servidores) é obter as informações sobre todas as interfaces do host quando o programa inicia: endereços de interface, se a interface suporta broadcasting ou multicasting, e assim por diante. Neste capítulo, desenvolveremos nossa própria função para retornar essas informações e fornecer uma implementação que utiliza `ioctl`, bem como examinaremos uma outra implementação que utiliza os soquetes de roteamento no Capítulo 18.

17.2 Função `ioctl`

Essa função afeta um arquivo aberto referenciado pelo argumento *fd*.

```
#include <unistd.h>

int ioctl(int fd, int request, ... /* void *arg */ );
```

Retorna: 0 se OK, –1 em erro

O terceiro argumento sempre é um ponteiro, mas o tipo de ponteiro depende da *solicitação*.

> O 4.4BSD define o segundo argumento como um `unsigned long` em vez de um `int`, mas isso não é um problema uma vez que arquivos de cabeçalho definem as constantes utilizadas para esse argumento. Contanto que o protótipo esteja no escopo (isto é, o programa que utiliza `ioctl` incluiu `<unistd.h>`), o tipo correto para o sistema será utilizado.
>
> Algumas implementações especificam o terceiro argumento como um ponteiro `void *` em vez da notação de reticências do ANSI C.
>
> Não há nenhum padrão a ser incluído no cabeçalho para definir o protótipo da função para `ioctl`, visto que não é padronizada pelo POSIX. Muitos sistemas a definem no `<unistd.h>`, como mostramos, mas os sistemas BSD tradicionais a definem no `<sys/ioctl.h>`.

Podemos dividir as *solicitações (requests)* relacionadas à rede em seis categorias:

- Operações de soquete
- Operações de arquivo
- Operações de interface
- Operações de cache ARP
- Operações de tabela de roteamento
- Sistema STREAMS (Capítulo 31)

Lembre-se, da Figura 7.20, de que não apenas algumas das operações de `ioctl` se sobrepõem a algumas operações de `fcntl` (por exemplo, configurar um soquete como não-bloqueador), como também há operações que podem ser especificadas por mais de uma maneira utilizando `ioctl` (por exemplo, configurar a posse do grupo do processo de um soquete).

A Figura 17.1 lista as *solicitações* (*request*s), junto com o tipo de dados para o qual o endereço *arg* deve apontar. As seções a seguir descrevem essas solicitações em mais detalhes.

17.3 Operações de soquete

Três solicitações `ioctl` são explicitamente utilizadas em soquetes (páginas 551 a 553 do TCPv2). Todas as três requerem que o terceiro argumento para `ioctl` seja um ponteiro para um inteiro.

`SIOCATMARK` Retorna um valor de não-zero por meio do inteiro apontado pelo terceiro argumento se o ponteiro de leitura do soquete estiver atualmente na marca fora da banda ou um valor zero se o ponteiro de leitura não estiver na marca fora da banda. Descreveremos os dados fora da banda em mais detalhes no Capítulo 24. O POSIX substitui essa solicitação pela função `sockatmark`; mostraremos uma implementação dessa nova função utilizando `ioctl` na Seção 24.3.

Categoria	*request*	Descrição	Tipo de dados
Soquete	SIOCATMARK	Na marca fora da banda?	int
	SIOCSPGRP	Configura o ID do processo ou o ID do grupo do processo do soquete	int
	SIOCGPGRP	Obtém o ID do processo ou ID do grupo do processo do soquete	int
Arquivo	FIONBIO	Configura/limpa o flag não-bloqueador	int
	FIOASYNC	Configura/limpa o flag de E/S assíncrona	int
	FIONREAD	Obtém o nº de bytes no buffer de recebimento	int
	FIOSETOWN	Configura o ID do processo ou o ID do grupo do processo do arquivo	int
	FIOGETOWN	Obtém o ID do processo ou o ID do grupo do processo do arquivo	int
Interface	SIOCGIFCONF	Obtém a lista de todas as interfaces	struct ifconf
	SIOCSIFADDR	Configura o endereço de interface	struct ifreq
	SIOCGIFADDR	Obtém o endereço de interface	struct ifreq
	SIOCSIFFLAGS	Configura os flags de interface	struct ifreq
	SIOCGIFFLAGS	Obtém os flags de interface	struct ifreq
	SIOCSIFDSTADDR	Configura o endereço ponto a ponto	struct ifreq
	SIOCGIFDSTADDR	Obtém o endereço ponto a ponto	struct ifreq
	SIOCGIFBRDADDR	Obtém o endereço de broadcast	struct ifreq
	SIOCSIFBRDADDR	Configura o endereço de broadcast	struct ifreq
	SIOCGIFNETMASK	Obtém a máscara de sub-rede	struct ifreq
	SIOCSIFNETMASK	Configura a máscara de sub-rede	struct ifreq
	SIOCGIFMETRIC	Obtém a métrica da interface	struct ifreq
	SIOCSIFMETRIC	Configura a métrica da interface	struct ifreq
	SIOCGIFMTU	Obtém o MTU da interface	struct ifreq
ARP	SIOC*xxx*	(muitos outros; dependente de implementação)	
	SIOCSARP	Cria/modifica a entrada ARP	struct arpreq
	SIOCGARP	Obtém a entrada ARP	struct arpreq
	SIOCDARP	Exclui a entrada ARP	struct arpreq
Roteamento	SIOCADDRT	Adiciona rota	struct rtentry
	SIOCDELRT	Exclui rota	struct rtentry
STREAMS	I_ *xxx*	(Consulte a Seção 31.5)	

Figura 17.1 O resumo das solicitações `ioctl` de rede.

`SIOCGPGRP` Retorna por meio do inteiro apontado pelo terceiro argumento de um ID de processo ou de um ID de grupo de processos configurado para receber o sinal `SIGIO` ou `SIGURG` para esse soquete. Essa solicitação é idêntica a um `fcntl` de `F_GETOWN` e observamos na Figura 7.20 que POSIX padroniza a `fcntl`.

`SIOCSPGRP` Configura um ID de processo ou um ID do grupo de processos para receber o sinal `SIGIO` ou `SIGURG` para esse soquete do inteiro apontado pelo terceiro argumento. Essa solicitação é idêntica a uma `fcntl` de `F_SETOWN` e observamos na Figura 7.20 que o POSIX padroniza a `fcntl`.

17.4 Operações de arquivo

O próximo grupo de solicitações inicia com `FIO` e pode ser aplicado a certos tipos de arquivos, além de soquetes. Discutiremos somente as solicitações que se aplicam a soquetes (página 553 do TCPv2). Todas as cinco solicitações a seguir requerem que o terceiro argumento para `ioctl` aponte para um inteiro:

`FIONBIO` O flag não-bloqueador para o soquete é limpo ou ativado, dependendo se o terceiro argumento para `ioctl` apontar para um valor zero ou não-zero, respectivamente. Essa solicitação tem o mesmo efeito do flag de *status* de arquivo `O_NONBLOCK`, que pode ser ligado e limpo com o comando `F_SETFL` para a função `fcntl`.

FIOASYNC O flag que administra a recepção de sinais de E/S assíncrona (SIGIO) para o soquete é limpo ou ativado, dependendo se o terceiro argumento para ioctl apontar para um valor zero ou não-zero, respectivamente. Esse flag tem o mesmo efeito do flag de *status* de arquivo O_ASYNC, que pode ser ligado e limpo com o comando F_SETFL para a função fcntl.

FIONREAD Retorna no inteiro apontado pelo terceiro argumento para ioctl o número de bytes atualmente no buffer de recebimento do soquete. Esse recurso também funciona para arquivos, pipes e terminais. Discutimos essa solicitação em mais detalhes na Seção 14.7.

FIOSETOWN O equivalente a SIOCSPGRP para um soquete.

FIOGETOWN O equivalente a SIOCGPGRP para um soquete.

17.5 Configuração de interface

Um dos primeiros passos seguidos por muitos programas que lidam com as interfaces de rede em um sistema é obter do kernel todas as interfaces configuradas no sistema. Isso é feito com a solicitação SIOCGIFCONF, que utiliza a estrutura ifconf, que, por sua vez, utiliza a estrutura ifreq, ambas mostradas na Figura 17.2.

Antes de chamar ioctl, alocamos um buffer e uma estrutura ifconf e então a inicializamos. Mostramos uma descrição disso na Figura 17.3, supondo que o tamanho do nosso buffer seja 1.024 bytes. O terceiro argumento para ioctl é um ponteiro para nossa estrutura ifconf.

Se supormos que o kernel retorna duas estruturas ifreq, poderíamos ter o arranjo mostrado na Figura 17.4 quando a ioctl retorna. As regiões sombreadas foram modificadas por ioctl. O buffer foi preenchido com as duas estruturas e o membro ifc_len da estrutura ifconf foi atualizado para refletir o volume de informações armazenadas no buffer. Supomos nessa figura que cada estrutura ifreq ocupa 32 bytes.

Um ponteiro para uma estrutura ifreq também é utilizado como um argumento a solicitações ioctl remanescentes de interface mostradas na Figura 17.1, que descreveremos na Seção 17.7. Observe que cada estrutura ifreq contém um union e há vários #defines para ocultar o fato de que esses campos são membros de um union. Todas as referências a membros individuais são feitas utilizando nomes definidos. Esteja ciente de que alguns sistemas adicionaram vários membros dependentes de implementação a ifr_ifru union.

<net/if.h>

```
struct ifconf {
    int ifc_len;                    /* tamanho do buffer, valor-resultado */
    union {
       caddr_t ifcu_buf;            /* entrada do usuário -> kernel */
       struct  ifreq *ifcu_req;     /* retorno do kernel -> usário */
    } ifc_ifcu;
};
#define ifc_buf ifc_ifcu.ifcu_buf   /* endereço do buffer */
#define ifc_req ifc_ifcu.ifcu_req   /* array de estruturas retornadas */

#define IFNAMSIZ 16

struct ifreq {
   char   ifr_name[IFNAMSIZ];       /* nome da interface, p.ex., "le0" */
   union {
       struct    sockaddr ifru_addr;
       struct    sockaddr ifru_dstaddr;
```

Figura 17.2 Estruturas ifconf e ifreq utilizadas com várias solicitações ioctl de interface (*continua*).

```
        struct    sockaddr ifru_broadaddr;
        short     ifru_flags;
        int       ifru_metric;
        caddr_t   ifru_data;
    } ifr_ifru;
};
#define   ifr_addr       ifr_ifru.ifru_addr       /* endereço */
#define   ifr_dstaddr    ifr_ifru.ifru_dstaddr    /* outra extremidade do
                                                     enlace ponto a ponto */
#define   ifr_broadaddr  ifr_ifru.ifru_broadaddr  /* endereço de broadcast */
#define   ifr_flags      ifr_ifru.ifru_flags      /* flags */
#define   ifr_metric     ifr_ifru.ifru_metric     /* métrica */
#define   ifr_data       ifr_ifru.ifru_data       /* para uso da interface */
```

<net/if.h>

Figura 17.2 Estruturas `ifconf` e `ifreq` utilizadas com várias solicitações `ioctl` de interface (*continuação*).

17.6 Função `get_ifi_info`

Como muitos programas precisam conhecer todas as interfaces em um sistema, desenvolveremos nossa própria função chamada `get_ifi_info` que retorna uma lista vinculada de estruturas, uma para cada interface atualmente "ativa". Nesta seção, implementaremos essa função utilizando `ioctl SIOCGIFCONF` e, no Capítulo 18, desenvolveremos uma versão utilizando soquetes de roteamento.

O FreeBSD fornece uma função chamada `getifaddrs` com uma funcionalidade semelhante.

Pesquisar toda a árvore-fonte do FreeBSD 4.8 mostra que 12 programas emitem a `ioctl SIOCGIFCONF` para determinar as interfaces presentes.

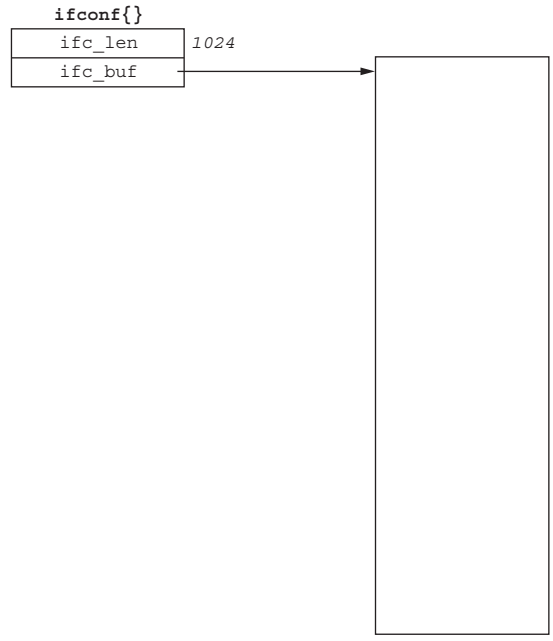


Figura 17.3 Inicialização da estrutura `ifconf` antes de `SIOCGIFCONF`.

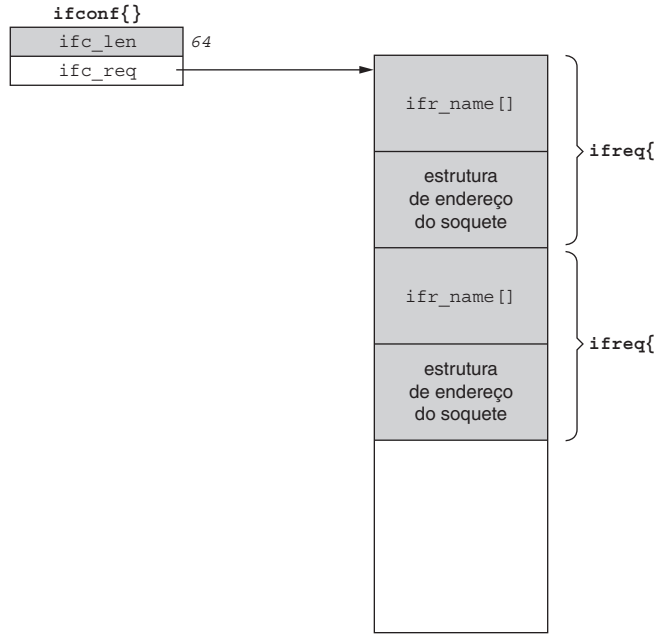


Figura 17.4 Valores retornados por `SIOCGIFCONF`.

Primeiro, definimos a estrutura `ifi_info` em um novo cabeçalho chamado `unpifi.h`, mostrado na Figura 17.5.

lib/unpifi.h

```
 1 /* Nosso próprio cabeçalho para os programas que precisam de informações
      sobre a configuração de interface.
 2    Inclua esse arquivo, em vez de "unp.h". */

 3 #ifndef __unp_ifi_h
 4 #define __unp_ifi_h

 5 #include   "unp.h"
 6 #include   <net/if.h>

 7 #define IFI_NAME      16          /* mesmo de IFNAMSIZ no <net/if. h> */
 8 #define IFI_HADDR      8          /* p/ permitir EUI-64 de 64 bits no futuro */

 9 struct ifi_info {
10     char   ifi_name[IFI_NAME];    /* nome da interface, terminado por
                                        caractere nulo */
11     short  ifi_index;             /* índice da interface */
12     short  ifi_mtu;               /* MTU da interface */
13     u_char ifi_haddr[IFI_HADDR]; /* endereço de hardware*/
14     u_shortifi_hlen;              /* n° de bytes no end. de hardware: 0, 6, 8*/
15     short  ifi_flags;             /* constante IFF_xxx de <net/if.h> */
16     short  ifi_myflags;           /* nossos próprios flags IFI_xxx */
17     struct sockaddr *ifi_addr;   /* endereço primário */
18     struct sockaddr *ifi_brdaddr;        /* endereço de broadcast */
19     struct sockaddr *ifi_dstaddr;               /* endereço de destino */
20     struct ifi_info *ifi_next;   /* próxima dessas estruturas */
21 };

22 #define IFI_ALIAS     1           /* ifi_addr é um alias */
```

Figura 17.5 Cabeçalho `unpifi.h` (*continua*).

```
23                      /* protótipos de função */
24 struct ifi_info *get_ifi_info(int, int);
25 struct ifi_info *Get_ifi_info(int, int);
26 void   free_ifi_info(struct ifi_info *);
27 #endif /* __unp_ifi_h */
```

lib/unpifi.h

Figura 17.5 Cabeçalho `unpifi.h` (*continuação*).

9-21 Uma lista vinculada dessas estruturas é retornada pela nossa função, cada membro `ifi_next` da estrutura apontando para o próximo membro. Retornamos nessa estrutura somente as informações nas quais uma aplicação típica provavelmente está interessada: o nome de interface, o índice de interface, o MTU, o endereço de hardware (por exemplo, um endereço Ethernet), os flags de interface (para deixar a aplicação determinar se a interface suporta broadcasting ou multicasting, ou se é uma interface ponto a ponto), o endereço de interface, o endereço de broadcast e o endereço de destino para um link ponto a ponto. Toda a memória utilizada para armazenar as estruturas `ifi_info`, juntamente com as estruturas de endereço do soquete contidas nelas, é obtida dinamicamente. Portanto, também fornecemos uma função `free_ifi_info` para liberar toda essa memória.

Antes de mostrar a implementação da nossa função `get_ifi_info`, mostraremos um programa simples que chama essa função e então gera a saída de todas as informações. Esse programa é uma versão em miniatura do programa `ifconfig` e é mostrado na Figura 17.6.

ioctl/prifinfo.c

```
 1 #include   "unpifi.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     struct ifi_info *ifi, *ifihead;
 6     struct sockaddr *sa;
 7     u_char *ptr;
 8     int    i, family, doaliases;
 9     if(argc != 3)
10         err_quit("usage: prifinfo <inet4|inet6> <doaliases>");
11     if (strcmp(argv[1], "inet4") == 0)
12         family = AF_INET;
13     else if (strcmp(argv[1], "inet6") == 0)
14         family = AF_INET6;
15     else
16         err_quit("invalid <address-family>");
17     doaliases = atoi(argv[2]);
18     for (ifihead = ifi = Get_ifi_info(family, doaliases);
19          ifi != NULL; ifi = ifi->ifi_next) {
20         printf("%s: ", ifi->ifi_name);
21         if (ifi->ifi_index != 0)
22             printf("(%d)", ifi->ifi_index);
23         printf("<");
24         if (ifi->ifi_flags & IFF_UP)            printf("UP ");
25         if (ifi->ifi_flags & IFF_BROADCAST)     printf("BCAST ");
26         if (ifi->ifi_flags & IFF_MULTICAST)     printf("MCAST ");
27         if (ifi->ifi_flags & IFF_LOOPBACK)      printf("LOOP ");
28         if (ifi->ifi_flags & IFF_POINTOPOINT)   printf("P2P ");
29         printf(">\n");
30         if ( (i = ifi->ifi_hlen) > 0) {
31             ptr = ifi->ifi_haddr;
```

Figura 17.6 Programa `prifinfo` que chama nossa função `get_ifi_info` (*continua*).

```
32              do {
33                  printf("%s%x", (i == ifi->ifi_hlen) ? " " : ":", *ptr++);
34              } while (--i > 0);
35              printf("\n");
36          }
37          if (ifi->ifi_mtu != 0)
38              printf(" MTU: %d\n", ifi->ifi_mtu);
39          if ( (sa = ifi->ifi_addr) != NULL)
40              printf(" IP addr: %s\n", Sock_ntop_host(sa, sizeof(*sa)));
41          if ( (sa = ifi->ifi_brdaddr) != NULL)
42              printf(" broadcast addr: %s\n",
43                     Sock_ntop_host(sa, sizeof(*sa)));
44          if ( (sa = ifi->ifi_dstaddr) != NULL)
45              printf(" destination addr: %s\n",
46                     Sock_ntop_host(sa, sizeof(*sa)));
47      }
48      free_ifi_info(ifihead);
49      exit(0);
50  }
```

ioctl/prifinfo.c

Figura 17.6 Programa `prifinfo` que chama nossa função `get_ifi_info` (*continuação*).

18–47 O programa é um loop `for` que chama `get_ifi_info` uma vez e então examina todas as estruturas `ifi_info` retornadas.

20–36 O nome da interface, o índice e os flags são impressos. Se o comprimento do endereço de hardware for maior que 0, ele é impresso como números hexadecimais. (Nossa função `get_ifi_info` retorna um `ifi_hlen` de 0 se não estiver disponível.)

37–46 O MTU e três endereços IP são impressos, se retornados.

Se executarmos esse programa no nosso host `macosx` (Figura 1.16), teremos a seguinte saída:

```
macosx % prifinfo inet4 0
lo0: <UP MCAST LOOP >
  MTU: 16384
  IP addr: 127.0.0.1
en1: <UP BCAST MCAST >
  MTU: 1500
  IP addr: 172.24.37.78
  broadcast addr: 172.24.37.95
```

O primeiro argumento da linha de comando de `inet4` especifica os endereços IPv4 e o segundo argumento em 0 especifica que nenhum alias de endereço dever ser retornado (descreveremos aliases de endereços IP na Seção A.4). Observe que, no MacOS X, o endereço de hardware da interface Ethernet não está disponível utilizando esse método.

Se adicionarmos três endereços de alias à interface Ethernet (`en1`) com IDs de host de 79, 80 e 81, e se alterarmos o segundo argumento da linha de comando para 1, teremos o seguinte:

```
macosx % prifinfo inet4 1
lo0: <UP MCAST LOOP >
  MTU: 16384
  IP addr: 127.0.0.1
en1: <UP BCAST MCAST >
  MTU: 1500
  IP addr: 172.24.37.78                  endereço IP primário
  broadcast addr: 172.24.37.95
en1: <UP BCAST MCAST >
  MTU: 1500
  IP addr: 172.24.37.79                  primeiro alias
```

```
    broadcast addr: 172.24.37.95
en1: <UP BCAST MCAST >
    MTU: 1500
    IP addr: 172.24.37.80                    segundo alias
    broadcast addr: 172.24.37.95
en1: <UP BCAST MCAST >
    MTU: 1500
    IP addr: 172.24.37.81                    terceiro alias
    broadcast addr: 172.24.37.95
```

Se executarmos o mesmo programa no FreeBSD utilizando a implementação de `get_ifi_info` da Figura 18.16 (que pode obter facilmente o endereço de hardware), teremos o seguinte:

```
freebsd4 % prifinfo inet4 1
de0: <UP BCAST MCAST >
    0:80:c8:2b:d9:28
    IP addr: 135.197.17.100
    broadcast addr: 135.197.17.255
de1: <UP BCAST MCAST >
    0:40:5:42:d6:de
    IP addr: 172.24.37.94                    endereço primário
    broadcast addr: 172.24.37.95
de1: <UP BCAST MCAST >
    0:40:5:42:d6:de
    IP addr: 172.24.37.93                    alias
    broadcast addr: 172.24.37.93
lo0: <UP MCAST LOOP >
    IP addr: 127.0.0.1
```

Nesse exemplo, orientamos o programa a imprimir os aliases e vemos que um deles é definido para a segunda interface Ethernet (`de1`) com um ID de host de 93.

Agora mostramos nossa implementação de `get_ifi_info` que utiliza o `ioctl` para `SIOCGIFCONF`. A Figura 17.7 mostra a primeira parte da função, que obtém a configuração de interface do kernel.

lib/get_ifi_info.c

```
 1 #include   "unpifi.h"

 2 struct ifi_info *
 3 get_ifi_info(int family, int doaliases)
 4 {
 5     struct ifi_info *ifi, *ifihead, **ifipnext;
 6     int    sockfd, len, lastlen, flags, myflags, idx = 0, hlen = 0;
 7     char   *ptr, *buf, lastname[IFNAMSIZ], *cptr, *haddr, *sdlname;
 8     struct ifconf ifc;
 9     struct ifreq *ifr, ifrcopy;
10     struct sockaddr_in *sinptr;
11     struct sockaddr_in6 *sin6ptr;

12     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

13     lastlen = 0;
14     len = 100 * sizeof(struct ifreq);    /* suposição do tamanho do buffer
                                               inicial */
15     for ( ; ; ) {
16         buf = Malloc(len);
17         ifc.ifc_len = len;
18         ifc.ifc_buf = buf;
19         if (ioctl(sockfd, SIOCGIFCONF, &ifc) < 0) {
20             if (errno != EINVAL || lastlen != 0)
21                 err_sys("ioctl error");
```

Figura 17.7 Emissão de uma solicitação `SIOCGIFCONF` para obter a configuração da interface (*continua*).

```
22          } else {
23              if (ifc.ifc_len == lastlen)
24                  break;      /* bem-sucedido, len não mudou */
25              lastlen = ifc.ifc_len;
26          }
27          len += 10 * sizeof(struct ifreq); /* incrementa */
28          free(buf);
29      }
30      ifihead = NULL;
31      ifipnext = &ifihead;
32      lastname[0] = 0;
33      sdlname = NULL;
```

lib/get_ifi_info.c

Figura 17.7 Emissão de uma solicitação `SIOCGIFCONF` para obter a configuração da interface (*continuação*).

Criação de um soquete Internet

11 Criamos um soquete UDP que será utilizado com `ioctl`. Um soquete TCP ou UDP pode ser utilizado (página 163 do TCPv2).

Emissão de uma solicitação `SIOCGIFCONF` em um loop

12–28 Um problema fundamental com a solicitação `SIOCGIFCONF` é que algumas implementações não retornam um erro se o buffer não for suficientemente grande para armazenar o resultado. Em vez disso, o resultado é truncado e sucesso é retornado (um valor de retorno de 0 a partir de `ioctl`). Isso significa que a única maneira de saber que nosso buffer é suficientemente grande é emitir a solicitação, salvar o comprimento de retorno, emitir a solicitação novamente com um buffer maior e comparar o comprimento com o valor salvo. Somente se os dois comprimentos forem iguais é que nosso buffer é suficientemente grande.

> Implementações derivadas do Berkeley não retornam um erro se o buffer for muito pequeno (páginas 118 e 119 do TCPv2); o resultado é simplesmente truncado para se ajustar ao buffer disponível. O Solaris 2.5, por outro lado, retorna `EINVAL` se o comprimento retornado for maior ou igual ao comprimento do buffer. Mas não podemos supor sucesso se o comprimento retornado for menor que o tamanho do buffer porque as implementações derivadas do Berkeley podem ter um retorno menor que o tamanho do buffer se uma outra estrutura não se ajustar.
>
> Algumas implementações fornecem uma solicitação `SIOCGIFNUM` que retorna o número de interfaces. Essa solicitação permite que a aplicação então aloque um buffer de tamanho suficiente antes de emitir a solicitação `SIOCGIFCONF`, mas essa nova solicitação não está disseminada.
>
> Alocar um tamanho fixo de buffer ao resultado da solicitação `SIOCGIFCONF` tornou-se um problema devido ao crescimento da Web, porque grandes servidores Web estão alocando muitos endereços de alias a uma única interface. O Solaris 2.5, por exemplo, tinha um limite de 256 aliases por interface, mas esse limite aumentou para 8.192 com o 2.6. Sites com numerosos aliases descobriram que os programas com buffers de tamanho fixo para informações sobre a interface começavam a falhar. Mesmo que o Solaris retorne um erro se um buffer for muito pequeno, esses programas alocam seus buffers de tamanho fixo, emitem a `ioctl`, mas então morrem se um erro for retornado.

12–15 Alocamos dinamicamente um buffer, começando com espaço para 100 estruturas `ifreq`. Também monitoramos o comprimento retornado pela última solicitação `SIOCGIFCONF` em `lastlen` e inicializamos isso em 0.

19–20 Se um erro de `EINVAL` for retornado por `ioctl` e se ainda não tivermos um retorno bem-sucedido (isto é, `lastlen` ainda é 0), não alocamos ainda um buffer suficientemente grande e continuamos pelo loop.

22–23 Se o retorno de `ioctl` for bem-sucedido e se o comprimento retornado for igual a `lastlen`, o comprimento não mudou (nosso buffer é suficientemente grande), e utilizamos um `break` para sair do loop, visto que temos todas as informações.

26–27 A cada passagem pelo loop, aumentamos o tamanho do buffer para armazenar 10 outras estruturas `ifreq`.

Inicialização dos ponteiros na lista vinculada

29–31 Como iremos retornar um ponteiro para o início de uma lista vinculada de estruturas `ifi_info`, utilizamos as variáveis `ifihead` e `ifipnext` para armazenar os ponteiros na lista à medida que a criamos.

A próxima etapa da nossa função `get_ifi_info`, o começo do loop principal, é mostrada na Figura 17.8.

lib/get_ifi_info.c

```
34      for (ptr = buf; ptr < buf + ifc.ifc_len;) {
35          ifr = (struct ifreq *) ptr;

36  #ifdef HAVE_SOCKADDR_SA_LEN
37          len = max(sizeof(struct sockaddr), ifr->ifr_addr.sa_len);
38  #else
39          switch (ifr->ifr_addr.sa_family) {
40  #ifdef IPV6
41          case AF_INET6:
42              len = sizeof(struct sockaddr_in6);
43              break;
44  #endif
45          case AF_INET:
46          default:
47              len = sizeof(struct sockaddr);
48              break;
49          }
50  #endif /* HAVE_SOCKADDR_SA_LEN */
51          ptr += sizeof(ifr->ifr_name) + len; /* para o próximo no buffer */

52  #ifdef HAVE_SOCKADDR_DL_STRUCT
53          /* assume que AF_LINK precede AF_INET ou AF_INET6 */
54          if (ifr->ifr_addr.sa_family == AF_LINK) {
55              struct sockaddr_dl *sdl = (struct sockaddr_dl *) &ifr->ifr_addr;
56              sdlname = ifr->ifr_name;
57              idx = sdl->sdl_index;
58              haddr = sdl->sdl_data + sdl->sdl_nlen;
59              hlen = sdl->sdl_alen;
60          }
61  #endif

62          if (ifr->ifr_addr.sa_family != family)
63              continue; /* ignora se não for a família desejada de endereços */

64          myflags = 0;
65          if ( (cptr = strchr(ifr->ifr_name, ':')) != NULL)
66              *cptr = 0;              /* substitui dois-pontos por null */
67          if (strncmp(lastname, ifr->ifr_name, IFNAMSIZ) == 0) {
68              if (doaliases == 0)
69                  continue;           /* já processou essa interface */
70              myflags = IFI_ALIAS;
71          }
72          memcpy(lastname, ifr->ifr_name, IFNAMSIZ);

73          ifrcopy = *ifr;
```

Figura 17.8 Processamento da configuração da interface (*continua*).

```
74          Ioctl(sockfd, SIOCGIFFLAGS, &ifrcopy);
75          flags = ifrcopy.ifr_flags;
76          if ((flags & IFF_UP) == 0)
77              continue;         /* ignora se a interface não estiver ativa */
```
lib/get_ifi_info.c

Figura 17.8 Processamento da configuração da interface (*continuação*).

Passando para a próxima estrutura de endereço do soquete

35–51 À medida que fazemos o loop por todas as estruturas `ifreq`, `ifr` aponta para cada estrutura e então incrementamos `ptr` para apontar para a próxima. Mas devemos lidar com os sistemas mais recentes que fornecem um campo de comprimento a estruturas de endereço de soquete e com sistemas mais antigos que não fornecem esse comprimento. Embora a Figura 17.2 declare a estrutura de endereço de soquete contida dentro da estrutura `ifreq` como uma estrutura de endereço de soquete genérica, nos sistemas mais recentes, isso pode ser qualquer tipo de estrutura de endereço de soquete. De fato, no 4.4BSD, uma estrutura de endereço de soquete de enlace de dados também é retornada para cada interface (página 118 do TCPv2). Portanto, se o membro de comprimento for suportado, devemos utilizar seu valor para atualizar nosso ponteiro para a próxima estrutura de endereço de soquete. Caso contrário, utilizamos um comprimento com base na família de endereços, usando o tamanho da estrutura do endereço de soquete genérica (16 bytes) como o padrão.

> Acrescentamos um `case` ao IPv6, para sistemas mais recentes, para qualquer eventualidade. O problema é que o `union` na estrutura `ifreq` define os endereços retornados como estruturas `sockaddr` genéricas de 16 bytes, adequadas para estruturas `sockaddr_in` de 16 bytes do IPv4, mas muito pequenas para estruturas `sockaddr_in6` de 28 bytes do IPv6. Isso não é um problema nos sistemas que têm o campo `sa_len` na `sockaddr`, uma vez que eles podem indicar facilmente estruturas `sockaddr` de tamanho variável.

Tratando `AF_LINK`

52–60 Se o sistema retornar as estruturas `sockaddr` do `AF_LINK` no `SIOCGIFCONF`, copie as informações sobre o índice da interface e o endereço de hardware da `sockaddr` do `AF_LINK`.

62–63 Ignoramos quaisquer endereços das famílias, exceto daquelas desejadas pelo chamador.

Tratando aliases

64–72 Devemos detectar quaisquer aliases que possam existir para a interface, isto é, os endereços adicionais atribuídos à interface. Observe nos nossos exemplos que seguem a Figura 17.6 que no Solaris um alias para nome da interface contém dois-pontos, enquanto no 4.4BSD o nome da interface não muda para um alias. Para tratar os dois casos, salvamos o último nome da interface em `lastname` e somente comparamos até um dois-pontos, se houver algum. Se não houver um dois-pontos, ainda ignoramos essa interface se o nome for equivalente à última interface que processamos.

Busca por flags de interface

73–77 Emitimos uma solicitação `ioctl` de `SIOCGIFFLAGS` (Seção 17.5) para buscar os flags de interface. O terceiro argumento para `ioctl` é um ponteiro para uma estrutura `ifreq` que deve conter o nome da interface na qual quereremos o flag. Criamos uma cópia da estrutura `ifreq` antes de emitir a solicitação `ioctl`, porque, se não fizermos isso, essa solicitação sobrescreveria o endereço IP da interface uma vez que as duas são membros do mesmo `union` na Figura 17.2. Se a interface não estiver ativa, nós a ignoramos.

A Figura 17.9 contém a terceira parte da nossa função.

lib/get_ifi_info.c

```
78          ifi = Calloc(1, sizeof(struct ifi_info));
79          *ifipnext = ifi;        /* prev aponta para esse novo */
80          ifipnext = &ifi->ifi_next; /* ponteiro para a próxima entra aqui */
81          ifi->ifi_flags = flags; /* valores IFF_xxx */
82          ifi->ifi_myflags = myflags; /* valores IFI_xxx */
83  #if defined(SIOCGIFMTU) && defined(HAVE_STRUCT_IFREQ_IFR_MTU)
84          Ioctl(sockfd, SIOCGIFMTU, &ifrcopy);
85          ifi->ifi_mtu = ifrcopy.ifr_mtu;
86  #else
87          ifi->ifi_mtu = 0;
88  #endif
89          memcpy(ifi->ifi_name, ifr->ifr_name, IFI_NAME);
90          ifi->ifi_name[IFI_NAME - 1] = '\0';
91          /* Se o sockaddr_dl for de uma interface diferente, ignore-o */
92          if (sdlname == NULL || strcmp(sdlname, ifr->ifr_name) != 0)
93              idx = hlen = 0;
94          ifi->ifi_index = idx;
95          ifi->ifi_hlen = hlen;
96          if (ifi->ifi_hlen > IFI_HADDR)
97              ifi->ifi_hlen = IFI_HADDR;
98          if (hlen)
99              memcpy(ifi->ifi_haddr, haddr, ifi->ifi_hlen);
```

lib/get_ifi_info.c

Figura 17.9 Alocação e inicialização da estrutura `ifi_info`.

Alocação e inicialização da estrutura `ifi_info`

78–99 Nesse ponto, sabemos que iremos retornar essa interface ao chamador. Alocamos memória a nossa estrutura `ifi_info` e a adicionamos ao final da lista vinculada que estamos construindo. Copiamos os flags, o MTU e o nome da interface para a estrutura. Então nos certificamos de que o nome da interface termina com um caractere nulo; e como `calloc` inicializa a região alocada para todos os bits zeros, sabemos que `ifi_hlen` é inicializado como 0 e que `ifi_next` é inicializado como um ponteiro nulo. Copiamos o índice da interface salvo e o comprimento de hardware; se o comprimento for não-zero, também copiamos o endereço de hardware salvo.

A Figura 17.10 contém a última parte da nossa função.

102–104 Copiamos o endereço IP que retornou da nossa solicitação `SIOCGIFCONF` original na estrutura que estamos construindo.

106–119 Se a interface suportar broadcasting, buscamos o endereço de broadcast com uma `ioctl` de `SIOCGIFBRDADDR`. Alocamos memória à estrutura de endereço do soquete que contém esse endereço e o adicionamos à estrutura `ifi_info` que estamos construindo. De maneira semelhante, se a interface for ponto a ponto, o `SIOCGIFDSTADDR` retorna o endereço IP da outra extremidade do link.

123–133 Esse é o caso do IPv6; é exatamente o mesmo do IPv4, exceto que não há nenhuma chamada a `SIOCGIFBRDADDR` porque o IPv6 não suporta broadcasting.

A Figura 17.11 mostra a função `free_ifi_info`, que recebe um ponteiro retornado por `get_ifi_info` e libera toda a memória dinâmica.

lib/get_ifi_info.c

```
100         switch (ifr->ifr_addr.sa_family) {
101         case AF_INET:
102             sinptr = (struct sockaddr_in *) &ifr->ifr_addr;
103             ifi->ifi_addr = Calloc(1, sizeof(struct sockaddr_in));
104             memcpy(ifi->ifi_addr, sinptr, sizeof(struct sockaddr_in));
```

Figura 17.10 Busca e retorno dos endereços de interface (*continua*).

```
105 #ifdef SIOCGIFBRDADDR
106             if (flags & IFF_BROADCAST) {
107                 Ioctl(sockfd, SIOCGIFBRDADDR, &ifrcopy);
108                 sinptr = (struct sockaddr_in *) &ifrcopy.ifr_broadaddr;
109                 ifi->ifi_brdaddr = Calloc(1, sizeof(struct sockaddr_in));
110                 memcpy(ifi->ifi_brdaddr, sinptr, sizeof(struct sockaddr_in));
111             }
112 #endif

113 #ifdef SIOCGIFDSTADDR
114             if (flags & IFF_POINTOPOINT) {
115                 Ioctl(sockfd, SIOCGIFDSTADDR, &ifrcopy);
116                 sinptr = (struct sockaddr_in *) &ifrcopy.ifr_dstaddr;
117                 ifi->ifi_dstaddr = Calloc(1, sizeof(struct sockaddr_in));
118                 memcpy(ifi->ifi_dstaddr, sinptr, sizeof(struct sockaddr_in));
119             }
120 #endif
121             break;

122         case AF_INET6:
123             sin6ptr = (struct sockaddr_in6 *) &ifr->ifr_addr;
124             ifi->ifi_addr = Calloc(1, sizeof(struct sockaddr_in6));
125             memcpy(ifi->ifi_addr, sin6ptr, sizeof(struct sockaddr_in6));

126 #ifdef SIOCGIFDSTADDR
127             if (flags & IFF_POINTOPOINT) {
128                 Ioctl(sockfd, SIOCGIFDSTADDR, &ifrcopy);
129                 sin6ptr = (struct sockaddr_in6 *) &ifrcopy.ifr_dstaddr;
130                 ifi->ifi_dstaddr = Calloc(1, sizeof(struct sockaddr_in6));
131                 memcpy(ifi->ifi_dstaddr, sin6ptr,
132                        sizeof(struct sockaddr_in6));
133             }
134 #endif
135             break;

136         default:
137             break;
138         }
139     }
140     free(buf);
141     return (ifihead);   /* ponteiro para a 1ª estrutura na lista vinculada */
142 }
```

lib/get_ifi_info.c

Figura 17.10 Busca e retorno dos endereços de interface (*continuação*).

lib/get_ifi_info.c

```
143 void
144 free_ifi_info(struct ifi_info *ifihead)
145 {
146   struct ifi_info *ifi, *ifinext;

147   for (ifi = ifihead; ifi != NULL; ifi = ifinext) {
148       if (ifi->ifi_addr != NULL)
149           free(ifi->ifi_addr);
150       if (ifi->ifi_brdaddr != NULL)
151           free(ifi->ifi_brdaddr);
152       if (ifi->ifi_dstaddr != NULL)
153           free(ifi->ifi_dstaddr);
154       ifinext = ifi->ifi_next;    /* não pode buscar ifi_next depois de free() */
155       free(ifi);              /* a própria ifi_info{} */
156   }
157 }
```

lib/get_ifi_info.c

Figura 17.11 Função `free_ifi_info`: libera a memória dinâmica alocada por `get_ifi_info`.

17.7 Operações de interface

Como mostramos na seção anterior, a solicitação SIOCGIFCONF retorna o nome e uma estrutura de endereço de soquete para cada interface configurada. Há várias outras solicitações que podemos então emitir para configurar ou obter todas as demais características da interface. A versão *get* dessas solicitações (SIOCG*xxx*) freqüentemente é emitida pelo programa netstat e a versão *set* (SIOCS*xxx*), pelo programa ifconfig. Qualquer usuário pode obter as informações sobre a interface, enquanto configurá-las requer privilégios de superusuário.

Essas solicitações recebem ou retornam uma estrutura ifreq cujo endereço é especificado como o terceiro argumento para ioctl. A interface sempre é identificada pelo seu nome: le0, lo0, ppp0, etc., no membro ifr_name.

Muitas dessas solicitações utilizam uma estrutura de endereço de soquete para especificar ou retornar um endereço IP ou uma máscara de endereço com a aplicação. Para o IPv4, o endereço ou a máscara está contido no membro sin_addr de uma estrutura de endereço de soquete da Internet; para o IPv6, o endereço ou máscara está no membro sin6_addr de uma estrutura de endereço de soquete IPv6.

SIOCGIFADDR
: Retorna o endereço unicast no membro ifr_addr.

SIOCSIFADDR
: Configura o endereço da interface do membro ifr_addr. A função de inicialização da interface também é chamada.

SIOCGIFFLAGS
: Retorna os flags da interface no membro ifr_flags. Os nomes dos vários flags são IFF_*xxx* e são definidos incluindo o cabeçalho <net/if.h>. Os flags indicam, por exemplo, se a interface está ativa (IFF_UP), se é uma interface ponto a ponto (IFF_POINTOPOINT), se suporta broadcasting (IFF_BROADCAST), e assim por diante.

SIOCSIFFLAGS
: Configura os flags da interface no membro ifr_flags.

SIOCGIFDSTADDR
: Retorna o endereço ponto a ponto no membro ifr_dstaddr.

SIOCSIFDSTADDR
: Configura o endereço ponto a ponto no membro ifr_dstaddr.

SIOCGIFBRDADDR
: Retorna o endereço de broadcast no membro ifr_broadaddr. A aplicação deve primeiro buscar os flags da interface e então emitir a solicitação correta: SIOCGIFBRDADDR para uma interface de broadcast ou SIOCGIFDSTADDR para uma interface ponto a ponto.

SIOCSIFBRDADDR
: Configura o endereço de broadcast no membro ifr_broadaddr.

SIOCGIFNETMASK
: Retorna a máscara de sub-rede no membro ifr_addr.

SIOCSIFNETMASK
: Configura a máscara de sub-rede no membro ifr_addr.

SIOCGIFMETRIC
: Retorna a métrica da interface no membro `ifr_metric`. A métrica da interface é mantida pelo kernel para cada interface, mas é utilizada pelo `routed` do daemon de roteamento. A métrica da interface é adicionada à contagem de hop (para tornar uma interface menos favorável).

SIOCSIFMETRIC
: Configura a métrica de roteamento da interface no membro `ifr_metric`.

Nesta seção, descrevemos as solicitações de interface genérica. Muitas implementações também têm solicitações adicionais.

17.8 Operações de cache ARP

Em alguns sistemas, a cache ARP também é manipulada pela função `ioctl`. Os sistemas que utilizam soquetes de roteamento (Capítulo 18) normalmente os preferem à `ioctl` para acessar a cache ARP. Essas solicitações utilizam uma estrutura `arpreq`, mostrada na Figura 17.12 e definida incluindo o cabeçalho `<net/if_arp.h>`.

<net/if_arp.h>

```
struct arpreq {
    struct  sockaddr   arp_pa;       /* endereço do protocolo */
    struct  sockaddr   arp_ha;       /* endereço de hardware */
    int                arp_flags;    /* flags */
};

#define  ATF_INUSE     0x01  /* entrada em utilização */
#define  ATF_COM       0x02  /* entrada completada (endereço de hardware válido) */
#define  ATF_PERM      0x04  /* entrada permanente */
#define  ATF_PUBL      0x08  /* entrada publicada (responde para outro host) */
```

<net/if_arp.h>

Figura 17.12 Estrutura `arpreq` utilizada com solicitações `ioctl` para cache ARP.

O terceiro argumento para `ioctl` deve apontar para uma dessas estruturas. As três solicitações (*request*s) a seguir são suportadas:

SIOCSARP
: Adicione uma nova entrada à cache ARP ou modifique uma entrada existente. `arp_pa` é uma estrutura de endereço de soquete da Internet que contém o endereço IP e `arp_ha` é uma estrutura de endereço de soquete genérica com `sa_family` configurado como `AF_UNSPEC` e `sa_data` contendo o endereço de hardware (por exemplo, o endereço ethernet de 6 bytes). Os dois flags, `ATF_PERM` e `ATF_PUBL`, podem ser especificados pela aplicação. Os outros dois flags, `ATF_INUSE` e `ATF_COM`, são configurados pelo kernel.

SIOCDARP
: Exclui uma entrada do cache ARP. O chamador especifica o endereço Internet na entrada a ser excluída.

SIOCGARP
: Obtém uma entrada do cache ARP. O chamador especifica o endereço Internet e o endereço Ethernet correspondente é retornado juntamente com os flags.

Somente o superusuário pode adicionar ou excluir uma entrada. Essas três solicitações normalmente são emitidas pelo programa `arp`.

> Essas solicitações `ioctl` relacionadas ao ARP não são suportadas em alguns sistemas mais recentes, que utilizam soquetes de roteamento para essas operações de ARP.

Observe que não há nenhuma maneira de `ioctl` listar todas as entradas na cache ARP. Em muitos sistemas, o comando `arp`, quando invocado com o flag `-a` (lista todas as entradas na cache ARP), lê a memória do kernel (`/dev/kmem`) para obter o conteúdo atual da cache ARP. Veremos uma maneira mais fácil (e melhor) de fazer isso utilizando `sysctl`, que funciona apenas em alguns sistemas (Seção 18.4).

Exemplo: Imprimindo endereços de hardware do host

Agora, utilizamos nossa função `get_ifi_info` para retornar todos os endereços IP de um host, seguidos por uma solicitação `ioctl` de `SIOCGARP` a cada endereço IP para obter e imprimir os endereços de hardware. Mostramos nosso programa na Figura 17.13.

Obtendo uma lista de endereços e fazendo um loop por cada um

12 Chamamos `get_ifi_info` para obter os endereços IP do host e então fazer um loop por cada um deles.

Imprimindo o endereço IP

13 Imprimimos o endereço IP utilizando `sock_ntop`. Solicitamos que `get_ifi_info` retorne somente endereços IPv4, uma vez que ARP não é utilizado com o IPv6.

Emitindo `ioctl` e verificando erros

14–19 Preenchemos a estrutura `arp_pa` como uma estrutura de endereço de soquete IPv4 contendo o endereço IPv4. `ioctl` é chamado e, se retornar um erro (por exemplo, porque o endereço fornecido não está em uma interface que suporta ARP), imprimimos o erro e fazemos um loop para o próximo endereço.

Imprimindo o endereço de hardware

20–22 O endereço de hardware retornado da `ioctl` é impresso.

ioctl/prmac.c

```
 1 #include   "unpifi.h"
 2 #include   <net/if_arp.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int    sockfd;
 7     struct ifi_info *ifi;
 8     unsigned char *ptr;
 9     struct arpreq arpreq;
10     struct sockaddr_in *sin;

11     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
12     for (ifi = get_ifi_info(AF_INET, 0); ifi != NULL; ifi = ifi->ifi_next) {
13         printf("%s: ", Sock_ntop(ifi->ifi_addr, sizeof(struct sockaddr_in)));

14         sin = (struct sockaddr_in *) &arpreq.arp_pa;
15         memcpy(sin, ifi->ifi_addr, sizeof(struct sockaddr_in));

16         if (ioctl(sockfd, SIOCGARP, &arpreq) < 0) {
17             err_ret("ioctl SIOCGARP");
```

Figura 17.13 Impressão dos endereços de hardware de um host (*continua*).

```
18              continue;
19          }

20          ptr = &arpreq.arp_ha.sa_data[0];
21          printf("%x:%x:%x:%x:%x:%x\n", *ptr, *(ptr + 1),
22                 *(ptr + 2), *(ptr + 3), *(ptr + 4), *(ptr + 5));
23      }
24      exit(0);
25  }
```

ioctl/prmac.c

Figura 17.13 Impressão dos endereços de hardware de um host (*continuação*).

Executar esse programa no nosso host `hpux` fornece:

```
hpux % prmac
192.6.38.100: 0:60:b0:c2:68:9b
192.168.1.1: 0:60:b0:b2:28:2b
127.0.0.1: ioctl SIOCGARP: Invalid argument
```

17.9 Operações da tabela de roteamento

Em alguns sistemas, são fornecidas duas solicitações `ioctl` para operar na tabela de roteamento. Essas duas solicitações requerem que o terceiro argumento para `ioctl` seja um ponteiro para uma estrutura `rtentry`, definida incluindo o cabeçalho `<net/route.h>`. Essas solicitações são normalmente emitidas pelo programa `route`. Somente o superusuário pode emitir essas solicitações. Nos sistemas com soquetes de roteamento (Capítulo 18), essas solicitações utilizam soquetes de roteamento em vez de `ioctl`.

`SIOCADDRT`	Adicione uma entrada à tabela de roteamento.
`SIOCDELRT`	Exclua uma entrada da tabela de roteamento.

Não há nenhuma maneira com `ioctl` de listar todas as entradas na tabela de roteamento. Essa operação é normalmente realizada pelo programa `netstat` quando invocado com o flag `-r`. Esse programa obtém a tabela de roteamento lendo a memória do kernel (`/dev/kmem`). Como ocorre ao listar a cache ARP, veremos uma maneira mais fácil (e melhor) de fazer isso utilizando `sysctl` na Seção 18.4.

17.10 Resumo

Os comandos `ioctl` utilizados nos programas de rede podem ser divididos em seis categorias:

- Operações de soquete (estamos na marca fora da banda?)
- Operações de arquivo (configuram ou limpam o flag não-bloqueador)
- Operações de interface (retornam a lista de interfaces, obtêm endereço de broadcast)
- Operações da tabela ARP (criar, modificar, obter, excluir)
- Operações da tabela de roteamento (adicionam ou excluem)
- Sistema STREAMS (Capítulo 31)

Utilizaremos as operações de soquete e de arquivo. Obter a lista de interfaces é uma operação tão comum que desenvolvemos nossa própria função para fazer isso. Utilizaremos essa função várias vezes no restante do texto. Somente alguns programas especializados utilizam as operações `ioctl` com a cache ARP e tabela de roteamento.

Exercícios

17.1 Na Seção 17.7, afirmamos que o endereço de broadcast retornado pela solicitação SIOCGIFBRDADDR é retornado no membro `ifr_broadaddr`. Mas, na página 173 do TCPv2, observe que ele é retornado no membro `ifr_dstaddr`. Isso faz diferença?

17.2 Modifique o programa `get_ifi_info` para emitir sua primeira solicitação SIOCGIFCONF a uma estrutura `ifreq` e então incremente o comprimento toda vez que passar pelo loop de acordo com o tamanho de uma dessas estruturas. Em seguida, acrescente algumas instruções ao loop para imprimir o tamanho do buffer toda vez que a solicitação é emitida, se `ioctl` retornar ou não um erro, e, se bem-sucedido, imprimir o comprimento do buffer retornado. Execute o programa `prifinfo` e verifique como seu sistema trata essa solicitação quando o tamanho do buffer é muito pequeno. Também imprima a família de endereços de quaisquer estruturas retornadas cuja família de endereços não seja o valor desejado para ver quais as outras estruturas que são retornadas pelo seu sistema.

17.3 Modifique a função `get_ifi_info` para retornar as informações sobre um endereço de alias se o endereço adicional estiver em uma sub-rede diferente do endereço anterior dessa interface. Isto é, nossa versão na Seção 17.6 ignorou os endereços 206.62.226.44 a 206.62.226.46 de aliases, o que é aceitável, uma vez que eles estão na mesma sub-rede do endereço primário da interface, 206.62.226.33. Mas se, nesse exemplo, um alias está em uma sub-rede diferente, suponha que 192.3.4.5. retorna uma estrutura `ifi_info` com as informações sobre o endereço adicional.

17.4 Se seu sistema suportar o `ioctl` para uma solicitação SIOCGIFNUM, modifique a Figura 17.7 para emitir essa solicitação e use o valor de retorno como a suposição inicial do tamanho do buffer.

Capítulo 18

Soquetes de Roteamento

18.1 Visão geral

Tradicionalmente, a tabela de roteamento do Unix dentro do kernel é acessada com os comandos `ioctl`. Na Seção 17.9, descrevemos os dois comandos fornecidos, `SIOCADDRT` e `SIOCDELRT`, para adicionar ou excluir uma rota. Também mencionamos que não há nenhum comando para fazer o dump de toda a tabela de roteamento e, em vez disso, programas como o `netstat` lêem a memória do kernel para obter o conteúdo da tabela de roteamento. Um outro aspecto dessa miscelânea é o fato de que os daemons de roteamento como `gated` precisam monitorar as mensagens de ICMP redirecionadas que são recebidas pelo kernel e freqüentemente fazem isso criando um soquete ICMP bruto (Capítulo 28) e ouvindo nesse soquete todas as mensagens ICMP recebidas.

O 4.3BSD Reno limpa a interface do subsistema de roteamento do kernel criando o domínio `AF_ROUTE`. O único tipo de soquete suportado no domínio de rota é um soquete bruto. Três tipos de operações são suportados em um soquete de roteamento:

1. Um processo pode enviar uma mensagem ao kernel gravando em um soquete de roteamento. Por exemplo, essa é a maneira como as rotas são adicionadas e excluídas.
2. Um processo pode ler uma mensagem a partir do kernel em um soquete de roteamento. Essa é a maneira como o kernel notifica um processo de que uma mensagem ICMP redirecionada foi recebida e processada ou a maneira como ele solicita uma conversão de rota a partir de um processo de roteamento externo.

   Algumas operações envolvem os dois procedimentos. Por exemplo, o processo envia uma mensagem ao kernel em um soquete de roteamento solicitando todas as informações sobre uma dada rota e então lê de volta a resposta do kernel no soquete de roteamento.
3. Um processo pode utilizar a função `sysctl` (Seção 18.4) para fazer dump de toda a tabela de roteamento ou listar todas as interfaces configuradas.

As duas primeiras operações exigem privilégios de superusuário na maioria dos sistemas, enquanto a última pode ser realizada por qualquer processo.

Algumas distribuições mais recentes flexibilizaram o requisito de superusuário para abrir um soquete de roteamento, em vez disso, apenas restringem mensagens do roteamento de soquete que mudam a tabela. Isso permite a qualquer processo utilizar, por exemplo, `RTM_GET` para pesquisar uma rota sem ser o superusuário.

Tecnicamente, a terceira operação não é realizada utilizando um soquete de roteamento, mas invoca a função `sysctl` genérica. Veremos que um dos parâmetros de entrada é a família de endereços (`AF_ROUTE` para as operações que descrevemos neste capítulo) e as informações retornadas estão no mesmo formato das informações retornadas pelo kernel em um soquete de roteamento. De fato, o processamento de `sysctl` para a família `AF_ROUTE` faz parte do código do soquete de roteamento em um kernel 4.4BSD (páginas 632 a 643 do TCPv2).

O utilitário `sysctl` apareceu no 4.4BSD. Infelizmente, nem todas as implementações que suportam soquetes de roteamento fornecem o `sysctl`. Por exemplo, o AIX 5.1 e o Solaris 9 suportam soquetes de roteamento, mas nenhum suporta o `sysctl`.

18.2 Estrutura de endereços de soquete de enlace de dados

Encontraremos estruturas de endereço de soquete de enlace de dados como valores de retorno contidos em algumas mensagens retornadas em um soquete de roteamento. A Figura 18.1 mostra a definição da estrutura, que inclui `<net/if_dl.h>`.

```
struct sockaddr_dl {
  uint8_t      sdl_len;
  sa_family_t  sdl_family;   /* AF_LINK */
  uint16_t     sdl_index;    /* índice atribuído pelo sistema, se > 0 */
  uint8_t      sdl_type;     /* IFT_ETHER etc. de <net/if_types.h> */
  uint8_t      sdl_nlen;     /* comprimento do nome, iniciando em sdl_data[0] */
  uint8_t      sdl_alen;     /* comprimento do endereço da camada de enlace */
  uint8_t      sdl_slen;     /* comprimento do seletor da camada de enlace */
  char         sdl_data[12]; /* área de trabalho mínima, pode ser maior; contém
                                nome i/f e o endereço da camada de enlace*/
};
```

Figura 18.1 Estrutura de endereço de soquete do enlace de dados.

Cada interface tem um índice positivo único, o qual veremos retornado pelas funções `if_nametoindex` e `if_nameindex` mais adiante neste capítulo, juntamente com as opções de soquete de multicasting do IPv6 no Capítulo 21 e algumas opções avançadas do soquete IPv4 e IPv6 no Capítulo 27.

O membro `sdl_data` contém tanto o nome como o endereço da camada de link (por exemplo, o endereço MAC de 48 bits para uma interface Ethernet). O nome inicia com `sdl_data[0]` e não é terminado por caractere nulo. O endereço da camada de enlace inicia com `sdl_nlen` bytes depois do nome. Esse cabeçalho define a macro seguinte a retornar o ponteiro para o endereço da camada de enlace:

```
#define LLADDR(s) ((caddr_t)((s)->sdl_data + (s)->sdl_nlen))
```

Essas estruturas de endereço de soquete têm comprimento variável (página 89 do TCPv2). Se o endereço e o nome da camada de enlace excederem a 12 bytes, a estrutura será maior que 20 bytes. Normalmente, o tamanho é arredondado para o próximo múltiplo de 4 bytes nos sistemas de 32 bits. Também veremos na Figura 22.3 que, quando uma dessas estruturas é retornada pela opção de soquete `IP_RECVIF`, todos os três comprimentos são 0 e não há absolutamente nenhum membro `sdl_data`.

18.3 Leitura e gravação

Depois que um processo cria um soquete de roteamento, ele pode enviar os comandos ao kernel gravando no soquete e ler as informações do kernel lendo a partir do soquete. Há 12 diferentes comandos de roteamento, cinco dos quais podem ser emitidos pelo processo. Esses comandos são definidos incluindo o cabeçalho `<net/route.h>` e são mostrados na Figura 18.2.

Tipo de mensagem	Para o kernel?	Do kernel?	Descrição	Tipo de estrutura
RTM_ADD	•	•	Adiciona rota	rt_msghdr
RTM_CHANGE	•	•	Altera gateway, métrica ou flags	rt_msghdr
RTM_DELADDR		•	Endereço sendo removido da interface	ifa_msghdr
RTM_DELETE	•	•	Exclui rota	rt_msghdr
RTM_DELMADDR		•	Endereço multicast sendo removido da interface	ifma_msghdr
RTM_GET	•	•	Notifica a métrica e outras informações sobre a rota	rt_msghdr
RTM_IFANNOUNCE		•	Interface sendo adicionada ou removida do sistema	if_announcemsghdr
RTM_IFINFO		•	Interface subindo, descendo, etc.	if_msghdr
RTM_LOCK	•	•	Bloqueia dada métrica	rt_msghdr
RTM_LOSING		•	O kernel suspeita que a rota está falhando	rt_msghdr
RTM_MISS		•	A pesquisa falhou nesse endereço	rt_msghdr
RTM_NEWADDR		•	Endereço sendo adicionado à interface	ifa_msghdr
RTM_NEWMADDR		•	Endereço multicast sendo incorporado à interface	ifma_msghdr
RTM_REDIRECT		•	Kernel instrui a utilizar uma rota diferente	rt_msghdr
RTM_RESOLVE		•	Solicitação para resolver o endereço da camada de enlace de destino	rt_msghdr

Figura 18.2 Tipos de mensagens trocadas por meio de um soquete de roteamento.

Cinco diferentes estruturas são trocadas por meio de um soquete de roteamento, como mostrado na coluna final desta figura: `rt_msghdr`, `if_announcemsghdr`, `if_msghdr`, `ifa_msghdr` e `ifma_msghdr`, que mostramos na Figura 18.3.

```
struct rt_msghdr {    /* de <net/route.h> */
    u_short   rtm_msglen;    /* para pular mensagens não-entendidas */
    u_char    rtm_version;   /* futura compatibilidade binária */
    u_char    rtm_type;      /* tipo de mensagem */
    u_short   rtm_index;     /* índice para ifp associado */
    int       rtm_flags;     /* flags, incl. kern & mensagem, p. ex., DONE */
    int       rtm_addrs;     /* máscara de bits identificando sockaddrs na msg */
    pid_t     rtm_pid;       /* identifica o emissor */
    int       rtm_seq;       /* para o emissor identificar a ação */
    int       rtm_errno;     /* por que falhou */
    int       rtm_use;       /* de rtentry */
    u_long    rtm_inits;     /* qual métrica estamos inicializando */
    struct    rt_metrics rtm_rmx;    /* a própria métrica */
};
struct if_msghdr {    /* de <net/if.h> */
    u_short    ifm_msglen;   /* para pular mensagens não-entendidas */
    u_char     ifm_version;  /* futura compatibilidade binária */
    u_char     ifm_type;     /* tipo de mensagem */
    int        ifm_addrs;    /* como rtm_addrs */
    int        ifm_flags;    /* valor de if_flags */
    u_short    ifm_index;    /* índice para ifp associado */
    struct     if_data ifm_data; /* estatística e outros dados sobre if */
};
```

Figura 18.3 As cinco estruturas retornadas com mensagens de roteamento (*continua*).

```
struct ifa_msghdr {   /* de <net/if.h> */
    u_short    ifam_msglen;  /* para pular mensagens não-entendidas */
    u_char     ifam_version; /* futura compatibilidade binária */
    u_char     ifam_type;    /* tipo de mensagem */
    int        ifam_addrs;   /* como rtm_addrs */
    int        ifam_flags;   /* valor de ifa_flags */
    u_short    ifam_index;   /* índice para ifp associado */
    int        ifam_metric;  /* valor de ifa_metric */
};
struct ifma_msghdr {  /* de <net/if.h> */
    u_short    ifmam_msglen; /* para pular mensagens não-entendidas */
    u_char     ifmam_version;/* futura compatibilidade binária */
    u_char     ifmam_type;   /* tipo de mensagem */
    int        ifmam_addrs;  /* como rtm_addrs */
    int        ifmam_flags;  /* valor de ifa_flags */
    u_short    ifmam_index;  /* índice para ifp associado */
};
struct if_announcemsghdr {   /* de <net/if.h> */
    u_short    ifan_msglen;  /* para pular mensagens não-entendidas */
    u_char     ifan_version; /* futura compatibilidade binária */
    u_char     ifan_type;    /* tipo de mensagem */
    u_short    ifan_index;   /* índice para ifp associado */
    char       ifan_name[IFNAMSIZ]; /* se nome, por exemplo "en0" */
    u_short    ifan_what;    /* que tipo de anúncio */
};
```

Figura 18.3 As cinco estruturas retornadas com mensagens de roteamento (*continuação*).

Os três primeiros membros de cada estrutura são os mesmos: comprimento, versão e tipo de mensagem. O tipo é uma das constantes da primeira coluna na Figura 18.2. O membro de comprimento permite a uma aplicação pular tipos de mensagem que ele não entende.

Os membros `rtm_addrs`, `ifm_addrs`, `ifam_addrs` e `ifmam_addrs` são máscaras de bits que especificam quais das oito possíveis estruturas de endereço do soquete seguem a mensagem. A Figura 18.4 mostra os valores e as constantes para cada máscara de bits, definidos incluindo o cabeçalho `<net/route.h>`.

Se múltiplas estruturas de endereço de soquete estiverem presentes, elas sempre estarão na ordem mostrada na tabela.

Exemplo: Buscando e imprimindo uma entrada da tabela de roteamento

Agora, mostramos um exemplo que utiliza soquetes de roteamento. Nosso programa aceita um argumento da linha de comando que consiste em um endereço IPv4 com pontos decimais

Máscara de bits		Índice do array		
Constante	**Valor**	**Constante**	**Valor**	**Estrutura de endereço de soquete contendo**
RTA_DST	0x01	*RTAX_DST*	0	Endereço de destino
RTA_GATEWAY	0x02	*RTAX_GATEWAY*	1	Endereço de gateway
RTA_NETMASK	0x04	*RTAX_NETMASK*	2	Máscara de rede
RTA_GENMASK	0x08	*RTAX_GENMASK*	3	Máscara de clonagem
RTA_IFP	0x10	*RTAX_IFP*	4	Nome da interface
RTA_IFA	0x20	*RTAX_IFA*	5	Endereço da interface
RTA_AUTHOR	0x40	*RTAX_AUTHOR*	6	Autor do redirecionamento
RTA_BRD	0x80	*RTAX_BRD*	7	Endereço de broadcast ou ponto a ponto de destino
		RTAX_MAX	8	Nº máximo de elementos

Figura 18.4 Constante utilizada para referenciar estruturas de endereço de soquete nas mensagens de roteamento.

e envia uma mensagem RTM_GET ao kernel para esse endereço. O kernel pesquisa o endereço na sua tabela de roteamento IPv4 e retorna uma mensagem RTM_GET com as informações sobre a entrada da tabela de roteamento. Por exemplo, se executarmos

```
freebsd % getrt 206.168.112.219
dest: 0.0.0.0
gateway: 12.106.32.1
netmask: 0.0.0.0
```

no nosso host freebsd, veremos que esse endereço de destino utiliza a rota-padrão (armazenada na tabela de roteamento com um endereço IP de destino de 0.0.0.0 e uma máscara de 0.0.0.0). O roteador do próximo hop é o gateway desse sistema para a Internet. Se executarmos

```
freebsd % getrt 192.168.42.0
dest: 192.168.42.0
gateway: AF_LINK, index=2
netmask: 255.255.255.0
```

especificando a Ethernet secundária como o destino, o destino será a própria rede. O gateway agora é a interface de saída, retornada como uma estrutura sockaddr_dl com um índice de interface de 2.

Antes de mostrar o código-fonte, mostramos o que gravamos no soquete de roteamento na Figura 18.5 juntamente com o que é retornado pelo kernel.

Construímos um buffer que contém uma estrutura rt_msghdr, seguido por uma estrutura de endereço de soquete que contém o endereço de destino para o kernel pesquisar. O rtm_type é RTM_GET e o rtm_addrs é RTA_DST (lembre-se da Figura 18.4), indicando que a única estrutura de endereço de soquete depois de rt_msghdr é uma que contém o endereço de destino. Esse comando pode ser utilizado com qualquer família de protocolos

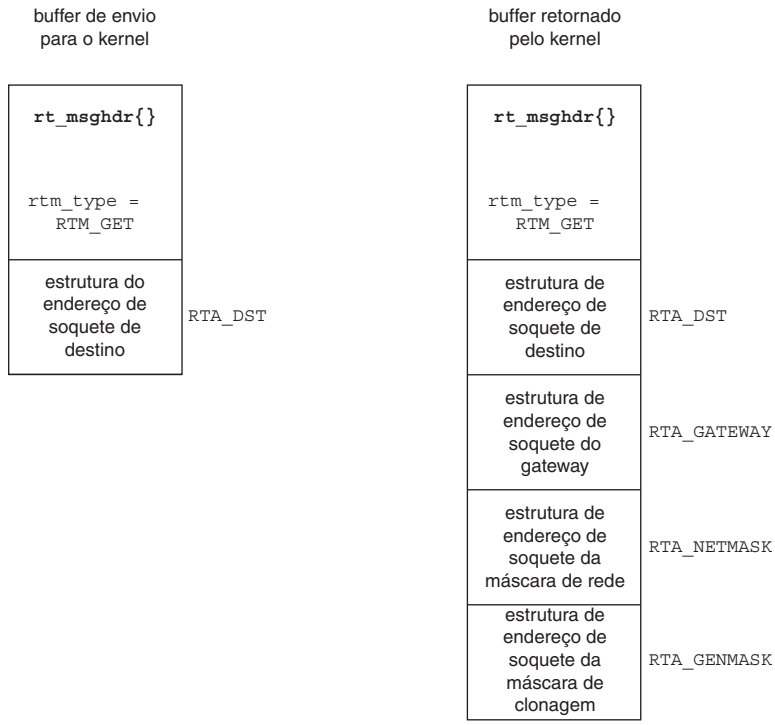


Figura 18.5 Dados trocados com o kernel no soquete de roteamento para o comando RTM_GET.

(que forneça uma tabela de roteamento), porque a família do endereço a pesquisar está contida na estrutura de endereço de soquete.

Depois de enviar a mensagem ao kernel, utilizamos `read` para ler a resposta a essa mensagem que tem o formato mostrado à direita da Figura 18.5: uma estrutura `rt_msghdr` seguida por até quatro estruturas de endereço de soquete. Qual das quatro estruturas de endereço de soquete é retornada depende da entrada da tabela de roteamento; somos informados sobre qual das quatro pelo valor no membro `rtm_addrs` da estrutura `rt_msghdr` retornada. A família de cada estrutura de endereço de soquete está contida no membro `sa_family` e, como vimos nos nossos exemplos anteriormente, na primeira vez o gateway retornado era uma estrutura de endereço de soquete IPv4 e, na segunda, era uma estrutura de endereço de soquete de enlace de dados.

A Figura 18.6 mostra a primeira parte do nosso programa.

route/getrt.c

```
 1 #include   "unproute.h"

 2 #define BUFLEN (sizeof(struct rt_msghdr) + 512)
 3                   /*sizeof(struct sockaddr_in6) * 8 = 192 */
 4 #define SEQ   9999

 5 int
 6 main(int argc, char **argv)
 7 {
 8     int    sockfd;
 9     char   *buf;
10     pid_t  pid;
11     ssize_t n;
12     struct rt_msghdr *rtm;
13     struct sockaddr *sa, *rti_info[RTAX_MAX];
14     struct sockaddr_in *sin;

15     if (argc != 2)
16         err_quit("usage: getrt <IPaddress>");

17     sockfd = Socket(AF_ROUTE, SOCK_RAW, 0); /* precisa privilégios de
                                                  superusuário */

18     buf = Calloc(1, BUFLEN);     /* e inicializado para 0 */

19     rtm = (struct rt_msghdr *) buf;
20     rtm->rtm_msglen = sizeof(struct rt_msghdr) + sizeof(struct sockaddr_in);
21     rtm->rtm_version = RTM_VERSION;
22     rtm->rtm_type = RTM_GET;
23     rtm->rtm_addrs = RTA_DST;
24     rtm->rtm_pid = pid = getpid();
25     rtm->rtm_seq = SEQ;

26     sin = (struct sockaddr_in *) (rtm + 1);
27     sin->sin_len = sizeof(struct sockaddr_in);
28     sin->sin_family = AF_INET;
29     Inet_pton(AF_INET, argv[1], &sin->sin_addr);

30     Write(sockfd, rtm, rtm->rtm_msglen);

31     do {
32         n = Read(sockfd, rtm, BUFLEN);
33     } while (rtm->rtm_type != RTM_GET || rtm->rtm_seq != SEQ ||
34              rtm->rtm_pid!= pid);
```

route/getrt.c

Figura 18.6 Primeira metade do programa para emitir o comando `RTM_GET` no soquete de roteamento.

1–3 Nosso cabeçalho `unproute.h` inclui alguns arquivos necessários e então o nosso arquivo `unp.h`. A constante `BUFLEN` é o tamanho do buffer que alocamos para armazenar nossa mensagem para o kernel, juntamente com a resposta deste. Precisamos de espaço para uma

estrutura `rt_msghdr` e possivelmente para oito estruturas de endereço de soquete (o número máximo é retornado em um soquete de roteamento). Como uma estrutura de endereço de soquete IPv6 tem um tamanho de 28 bytes, o valor de 512 é mais do que adequado.

Criação de um soquete de roteamento

17 Criamos um soquete bruto no domínio `AF_ROUTE` e, como discutimos anteriormente, isso talvez requeira privilégios de superusuário. Um buffer é alocado e inicializado em 0.

Preenchimento da estrutura `rt_msghdr`

18–25 Preenchemos a estrutura com nossa solicitação. Armazenamos nosso ID de processo e um número de seqüência da nossa escolha na estrutura. Vamos comparar esses valores nas respostas que lemos, procurando a resposta correta.

Preenchimento da estrutura de endereço de soquete de Internet com o destino

26–29 Seguindo-se à estrutura `rt_msghdr`, construímos uma estrutura `sockaddr_in` que contém o endereço IPv4 de destino para o kernel para pesquisar na sua tabela de roteamento. Tudo o que configuramos são o comprimento do endereço, a família do endereço e o endereço.

Utilização de `write` para gravar a mensagem no kernel e de `read` para responder

30–34 Utilizamos `write` para gravar a mensagem no kernel e `read` para ler de volta a resposta. Como outros processos podem ter os soquetes de roteamento abertos, e visto que o kernel passa uma cópia de todas as mensagens de roteamento para todos os soquetes de roteamento, devemos verificar o tipo de mensagem, o número de seqüência e o ID do processo para verificar se a mensagem recebida é a que estamos esperando.

A segunda metade desse programa é mostrada na Figura 18.7. Essa metade processa a resposta.

route/getrt.c

```
35      rtm = (struct rt_msghdr *) buf;
36      sa = (struct sockaddr *) (rtm + 1);
37      get_rtaddrs(rtm->rtm_addrs, sa, rti_info);
38      if ( (sa = rti_info[RTAX_DST]) != NULL)
39          printf("dest: %s\n", Sock_ntop_host(sa, sa->sa_len));
40      if ( (sa = rti_info[RTAX_GATEWAY]) != NULL)
41          printf("gateway: %s\n", Sock_ntop_host(sa, sa->sa_len));
42      if ( (sa = rti_info[RTAX_NETMASK]) != NULL)
43          printf("netmask: %s\n", Sock_masktop(sa, sa->sa_len));
44      if ( (sa = rti_info[RTAX_GENMASK]) != NULL)
45          printf("genmask: %s\n", Sock_masktop(sa, sa->sa_len));
46      exit(0);
47  }
```

route/getrt.c

Figura 18.7 Segunda metade do programa para emitir um comando `RTM_GET` no soquete de roteamento.

35–36 `rtm` aponta para a estrutura `rt_msghdr` e `sa` aponta para a primeira estrutura de endereço de soquete que se segue.

37 `rtm_addrs` é uma máscara de bits indicando qual das oito possíveis estruturas de endereço de soquete segue a estrutura `rt_msghdr`. Nossa função `get_rtaddrs` (que mostraremos em seguida) recebe essa máscara mais o ponteiro para a primeira estrutura de endereço de soquete (`sa`) e preenche o array `rti_info` com os ponteiros para as estruturas de endereço de soquete correspondentes. Supondo que todas as quatro estruturas de endereço de soquete

mostradas na Figura 18.5 são retornados pelo kernel, o array `rti_info` resultante será como mostrado na Figura 18.8.

Nosso programa então examina o array `rti_info`, fazendo o que for necessário com todos os ponteiros não-nulos no array.

38–45 Cada um dos quatro possíveis endereços são impressos, se presentes. Chamamos nossa função `sock_ntop_host` para imprimir o endereço de destino e o endereço do gateway, mas chamamos `sock_masktop` para imprimir as duas máscaras. Mostraremos essa nova função em seguida.

A Figura 18.9 mostra nossa função `get_rtaddrs` que chamamos na Figura 18.7.

Fazendo um loop por oito possíveis ponteiros

17–23 `RTAX_MAX` é 8 na Figura 18.4, o número máximo de estruturas de endereço de soquete retornado em uma mensagem de roteamento do kernel. O loop nessa função examina cada uma das oito constantes da máscara de bits `RTA_`*xxx* na Figura 18.4 que podem ser configuradas nos membros `rtm_addrs`, `ifm_addrs` ou `ifam_addrs` das estruturas na Figura 18.3. Se o bit estiver configurado, o elemento correspondente no array `rti_info` é configurado como o ponteiro para a estrutura de endereço de soquete; caso contrário, o elemento no array é configurado como um ponteiro nulo.

Passando para a próxima estrutura de endereço do soquete

2–12 As estruturas de endereço de soquete são de comprimento variável, mas esse código assume que cada uma tem um campo `sa_len` para especificar o comprimento. Há duas complicações que devem ser tratadas. Primeira: as duas máscaras, a máscara de rede e a máscara de clonagem, podem ser retornadas em uma estrutura de endereço de soquete com um `sa_len` de 0, mas isso na verdade ocupa o tamanho de um `unsigned long`. (O Capítulo 19 do

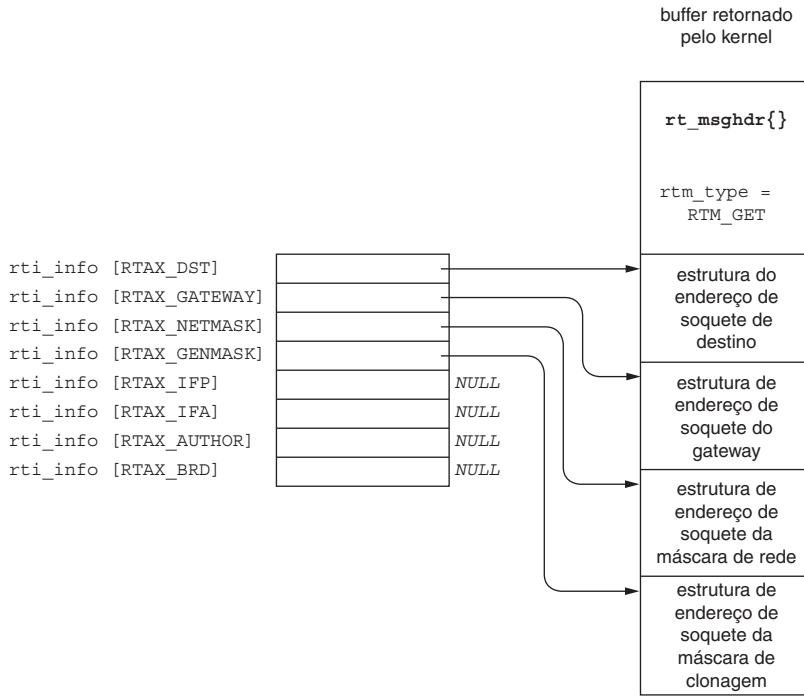


Figura 18.8 Estrutura `rti_info` preenchida pela nossa função `get_rtaddrs`.

libroute/get_rtaddrs.c

```
 1 #include   "unproute.h"
 2 /*
 3  * Arredonda 'a' para o próximo múltiplo do "tamanho", que deve ser uma
       potência de 2
 4  */
 5 #define ROUNDUP(a, size) (((a) & ((size)-1)) ? (1 + ((a) | ((size)-1))) : (a))
 6 /*
 7  * Passa para a próxima estrutura de endereço de soquete;
 8  * Se sa_len for 0, assume que ela é sizeof(u_long).
 9  */
10 #define NEXT_SA(ap) ap = (SA *) \
11     ((caddr_t) ap + (ap->sa_len ? ROUNDUP(ap->sa_len, sizeof (u_long)) : \
12                                       sizeof(u_long)))
13 void
14 get_rtaddrs(int addrs, SA *sa, SA **rti_info)
15 {
16     int    i;
17     for (i = 0; i < RTAX_MAX; i++) {
18         if (addrs & (1 << i)) {
19             rti_info[i] = sa;
20             NEXT_SA(sa);
21         } else
22             rti_info[i] = NULL;
23     }
24 }
```

libroute/get_rtaddrs.c

Figura 18.9 Construa um array de ponteiros para estruturas de endereço de soquete na mensagem de roteamento.

TCPv2 discute o recurso de clonagem da tabela de roteamento do 4.4BSD.) Esse valor representa uma máscara com todos os bits zeros, que imprimimos como 0.0.0.0 para a máscara de rede da rota default no nosso exemplo anterior. Segunda: cada estrutura de endereço de soquete pode ser preenchida no final, de modo que a próxima inicie em um limite específico, que nesse caso é o tamanho de um `unsigned long` (por exemplo, um limite de 4 bytes para uma arquitetura de 32 bits). Embora as estruturas `sockaddr_in` ocupem 16 bytes, o que não requer nenhum preenchimento, freqüentemente as máscaras têm preenchimento no final.

A última função que devemos mostrar para o nosso programa de exemplo é `sock_masktop` na Figura 18.10, que retorna a string de apresentação para um dos dois valores da máscara que podem ser retornados. As máscaras são armazenadas nas estruturas de endereço de soquete. O membro `sa_family` não é definido, mas as estruturas de endereço de soquete da máscara contêm um `sa_len` de 0, 5, 6, 7 ou 8 para máscaras IPv4 de 32 bits. Quando o comprimento é maior que 0, a máscara real inicia no mesmo deslocamento a partir do início que o endereço IPv4 em uma estrutura `sockaddr_in`: 4 bytes a partir do início da estrutura (como mostrado na Figura 18.21, página 577 do TCPv2), que é o membro `sa_data[2]` da estrutura de endereço de soquete genérica.

libroute/sock_masktop.c

```
1 #include   "unproute.h"
2 const char *
3 sock_masktop(SA *sa, socklen_t salen)
4 {
```

Figura 18.10 Conversão de um valor de máscara no seu formato de apresentação (*continua*).

```
 5     static char str[INET6_ADDRSTRLEN];
 6     unsigned char *ptr = &sa->sa_data[2];

 7     if (sa->sa_len == 0)
 8         return ("0.0.0.0");
 9     else if (sa->sa_len == 5)
10         snprintf(str, sizeof(str), "%d.0.0.0", *ptr);
11     else if (sa->sa_len == 6)
12         snprintf(str, sizeof(str), "%d.%d.0.0", *ptr, *(ptr + 1));
13     else if (sa->sa_len == 7)
14         snprintf(str, sizeof(str), "%d.%d.%d.0", *ptr, *(ptr + 1),
15                  *(ptr + 2));
16     else if (sa->sa_len == 8)
17         snprintf(str, sizeof(str), "%d.%d.%d.%d",
18                  *ptr, *(ptr + 1), *(ptr + 2), *(ptr + 3));
19     else
20         snprintf(str, sizeof(str), "(unknown mask, len = %d, family = %d)",
21                  sa->sa_len, sa->sa_family);
22     return (str);
23 }
```

libroute/sock_masktop.c

Figura 18.10 Conversão de um valor de máscara no seu formato de apresentação (*continuação*).

7–21 Se o comprimento for 0, a máscara implícita será 0.0.0.0. Se o comprimento for 5, somente o primeiro byte da máscara de 32 bits será armazenado, com um valor implícito de 0 para os 3 bytes restantes. Quando o comprimento é 8, todos os 4 bytes da máscara são armazenados.

Nesse exemplo, queremos ler a resposta do kernel porque ela contém as informações que estamos procurando. Mas, em geral, o valor de retorno de nosso `write` para o soquete de roteamento informa se o comando foi bem-sucedido ou não. Se isso corresponde a todas as informações que precisamos, poderemos chamar `shutdown` com um segundo argumento de `SHUT_RD` imediatamente depois de abrir o soquete para evitar que uma resposta seja enviada. Por exemplo, se estivermos excluindo uma rota, um retorno de 0 a partir de `write` significa sucesso, enquanto um retorno de erro de `ESRCH` significa que a rota não pode ser encontrada (página 608 do TCPv2). De maneira semelhante, um retorno de erro de `EEXIST` a partir de `write` ao adicionar uma rota significa que a entrada já existe. No nosso exemplo na Figura 18.6, se a entrada da tabela de roteamento não existir (digamos que o nosso host não tem uma rota-padrão), então `write` retorna um erro de `ESRCH`.

18.4 Operações `sysctl`

Nosso principal interesse quanto aos soquetes de roteamento é o uso da função `sysctl` para examinar tanto a tabela de roteamento como a lista de interfaces. Apesar de a criação de um soquete de roteamento (um soquete bruto no domínio `AF_ROUTE`) exigir privilégios de superusuário, qualquer processo pode examinar a tabela de roteamento e a lista de interfaces utilizando `sysctl`.

```
#include <sys/param.h>
#include <sys/sysctl.h>

int sysctl(int *name, u_int namelen, void *oldp, size_t *oldlenp,
           void *newp, size_t newlen);
```

Retorna: 0 se OK, –1 em erro

Essa função utiliza nomes semelhantes aos nomes MIB (*management information base*) do SNMP. O Capítulo 25 do TCPv1 discute o SNMP e seu MIB em detalhes. Esses nomes são hierárquicos.

O argumento *name* é um array de inteiros que especificam o nome e *namelen* especifica o número de elementos no array. O primeiro elemento no array especifica o subsistema do kernel para o qual a solicitação é direcionada. O segundo elemento especifica alguma parte desse subsistema, e assim por diante. A Figura 18.11 mostra o arranjo hierárquico, com algumas das constantes utilizadas nos três primeiros níveis.

Para buscar um valor, *oldp* aponta para um buffer em que o kernel armazena o valor. *oldlenp* é um argumento de valor-resultado: quando a função é chamada, o valor apontado por *oldlenp* especifica o tamanho desse buffer e, no retorno, o valor contém a quantidade de dados armazenados no buffer pelo kernel. Se o buffer não for suficientemente grande, `ENOMEM` é retornado. Como um caso especial, *oldp* pode ser um ponteiro nulo e *oldlenp* um ponteiro não-nulo, o kernel pode determinar o volume de dados que a chamada retornou e retorna esse tamanho por meio do *oldlenp*.

Para configurar um novo valor, *newp* aponta para um buffer do tamanho de *newlen*. Se um novo valor não for especificado, *newp* deverá ser um ponteiro nulo e *newlen* 0.

A página man do `sysctl` detalha todas as várias informações sobre o sistema que podem ser obtidas com essa função: informações sobre os sistemas de arquivos, memória virtual, limites do kernel, hardware e assim por diante. Nosso interesse é no subsistema de rede, especificado pelo primeiro elemento do array *name* sendo configurado como `CTL_NET`. (A constante `CTL_`*xxx* é definida incluindo o cabeçalho `<sys/sysctl.h>`.) O segundo elemento pode então ser como segue:

- `AF_INET` – Obtém ou configura variáveis que afetam os protocolos de Internet. O próximo nível especifica o protocolo utilizando a constante `IPPROTO_`*xxx*. O FreeBSD 5.0 fornece cerca de 75 variáveis nesse nível, controlando recursos como, por exemplo, se o kernel deve gerar um redirecionamento de ICMP, se o TCP deve utilizar as opções da RFC 1323, se as somas de verificação UDP devem ser enviadas, e assim por diante. Mostraremos um exemplo dessa utilização de `sysctl` no final desta seção.
- `AF_LINK` – Obtém ou configura as informações sobre a camada de enlace como, por exemplo, o número de interfaces PPP.
- `AF_ROUTE` – Retorna as informações sobre uma tabela de roteamento ou lista de interface. Descreveremos essas informações em seguida.
- `AF_UNSPEC` – Obtém ou configura algumas variáveis da camada de soquete como, por exemplo, o tamanho máximo de um buffer de envio ou recebimento de soquete.

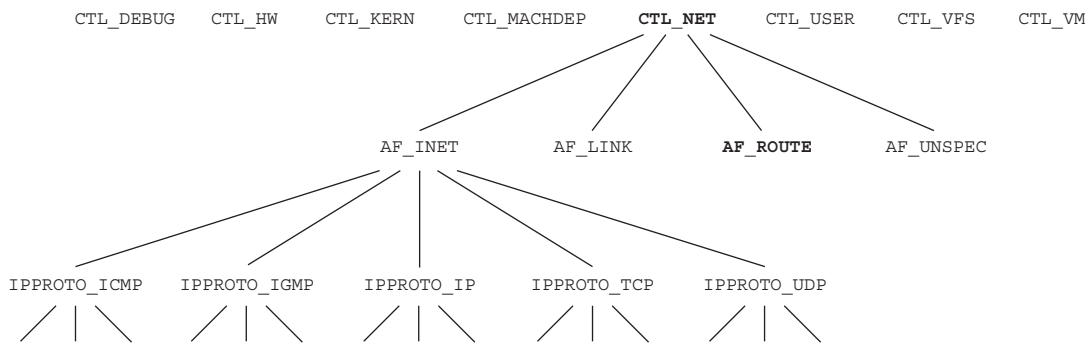


Figura 18.11 Arranjo hierárquico dos nomes de `sysctl`.

Quando o segundo elemento do array *name* é AF_ROUTE, o terceiro elemento (um número de protocolo) sempre é 0 (porque não há nenhum protocolo dentro da família AF_ROUTE, como há na família AF_INET, por exemplo), o quarto elemento é uma família de endereços e o quinto e sexto níveis especificam o que fazer. Resumimos isso na Figura 18.12.

Três operações são suportadas, especificadas por *name*[4]. (A constante NET_RT_*xxx* é definida incluindo o cabeçalho <sys/socket.h>.) As informações retornadas por essas quatro operações são retornadas por meio do ponteiro *oldp* na chamada a sysctl. Esse buffer contém um número variável de mensagens RTM_*xxx* (Figura 18.2).

1. NET_RT_DUMP retorna a tabela de roteamento à família de endereços especificada por *name*[3]. Se essa família de endereços for 0, as tabelas de roteamento para todas as famílias de endereços são retornadas.

   A tabela de roteamento é retornada como um número variável de mensagens RTM_GET, com cada mensagem seguida por até quatro estruturas de endereço de soquete: destino, gateway, máscara de rede e máscara de clonagem da entrada da tabela de roteamento. Mostramos uma dessas mensagens à direita na Figura 18.5 e o nosso código, que analisou uma dessas mensagens, na Figura 18.7. Tudo o que muda com essa operação sysctl é que ou uma ou mais dessas mensagens são retornadas pelo kernel.

2. NET_RT_FLAGS retorna a tabela de roteamento à família de endereços especificada por *name*[3], mas somente as entradas da tabela de roteamento com um valor de flag RTF_*xxx* que contém o flag especificado por *name*[5]. Todas as entradas da cache ARP na tabela de roteamento têm o bit de flag RTF_LLINFO configurado.

   As informações são retornadas no mesmo formato do item anterior.

3. NET_RT_IFLIST retorna as informações sobre todas as interfaces configuradas. Se *name*[5] for não-zero, ele será um número de índice da interface e somente as informações sobre essa interface são retornadas. (Discutiremos índices de interface em mais detalhes na Seção 18.6.) Todos os endereços atribuídos a cada interface também são retornados e, se *name*[3] for não-zero, somente os endereços para essa família de endereços são retornados.

   Para cada interface, uma mensagem RTM_IFINFO é retornada, seguida por uma mensagem RTM_NEWADDR para cada endereço atribuído à interface. A mensagem RTM_IFINFO é seguida por uma estrutura de endereço de soquete de enlace de dados e cada mensagem RTM_NEWADDR é seguida por até três estruturas de endereço de soquete: o endereço de interface, a máscara de rede e o endereço de broadcast. Essas duas mensagens são mostradas na Figura 18.14.

name[]	Retorna a tabela de roteamento do IPv4	Retorna a cache ARP do IPv4	Retorna a tabela de roteamento do IPv6	Retorna a lista de interfaces
0	CTL_NET	CTL_NET	CTL_NET	CTL_NET
1	AF_ROUTE	AF_ROUTE	AF_ROUTE	AF_ROUTE
2	0	0	0	0
3	AF_INET	AF_INET	AF_INET6	0
4	NET_RT_DUMP	NET_RT_FLAGS	NET_RT_DUMP	NET_RT_IFLIST
5	0	RTF_LLINFO	0	0

Figura 18.12 Informações sobre o sysctl retornadas para domínio de rota.

Exemplo: Determine se as somas de verificação do UDP estão ativadas

Agora, fornecemos um exemplo simples do `sysctl` com os protocolos de Internet para verificar se somas de verificação (checksums) do UDP estão ativadas. Algumas aplicações UDP (por exemplo, BIND) verificam se as somas de verificação do UDP estão ativadas quando iniciam e, se não estiverem, tentam ativá-las. Naturalmente, são necessários privilégios de superusuário para ativar um recurso como esse, mas tudo o que fazemos agora é verificar se o recurso está ou não ativado. A Figura 18.13 é o nosso programa.

route/checkudpsum.c

```
 1 #include   "unproute.h"
 2 #include   <netinet/udp.h>
 3 #include   <netinet/ip_var.h>
 4 #include   <netinet/udp_var.h> /* para constante UDPCTL_xxx */

 5 int
 6 main(int argc, char **argv)
 7 {
 8     int    mib[4], val;
 9     size_t len;

10     mib[0] = CTL_NET;
11     mib[1] = AF_INET;
12     mib[2] = IPPROTO_UDP;
13     mib[3] = UDPCTL_CHECKSUM;

14     len = sizeof(val);
15     Sysctl(mib, 4, &val, &len, NULL, 0);
16     printf("udp checksum flag: %d\n", val);

17     exit(0);
18 }
```

route/checkudpsum.c

Figura 18.13 Verificando se as somas de verificação do UDP estão ativadas.

Inclusão de cabeçalhos de sistema

2–4 Devemos incluir o cabeçalho `<netinet/udp_var.h>` para obter a definição das constantes `sysctl` do UDP. Os dois outros cabeçalhos são requeridos para esse cabeçalho.

Chamada a `sysctl`

10–16 Alocamos um array de inteiros com quatro elementos e armazenamos a constante que corresponde à hierarquia mostrada na Figura 18.11. Como estamos buscando apenas uma variável e não configurando um novo valor, especificamos um ponteiro nulo para o argumento *newp* para `sysctl` e um valor de 0 para o argumento *newlen*. *oldp* aponta para um inteiro variável nosso em que o resultado é armazenado e *oldlenp* aponta para uma variável valor-resultado para o tamanho desse inteiro. O flag que imprimimos será 0 (desativado) ou 1 (ativado).

18.5 Função `get_ifi_info` (revisitada)

Agora, voltamos ao exemplo na Seção 17.6: retornando todas as interfaces que estão ativas como uma lista vinculada de estruturas `ifi_info` (Figura 17.5). O programa `prifinfo` permanece o mesmo (Figura 17.6), mas agora mostramos uma versão da função `get_ifi_info` que utiliza `sysctl` em vez da `SIOCGIFCONF ioctl` utilizada na Figura 17.7.

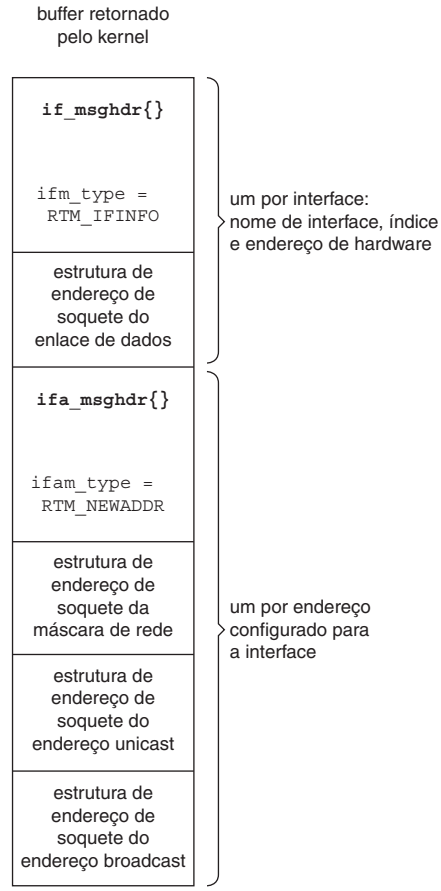


Figura 18.14 Informações retornadas para o comando `sysctl`, `CTL_NET`, `NET_RT_IFLIST`.

Primeiro, mostramos a função `net_rt_iflist` na Figura 18.15. Essa função chama `sysctl` com o comando `NET_RT_IFLIST` para retornar a lista de interface de uma família de endereços especificada.

libroute/net_rt_iflist.c

```
 1 #include   "unproute.h"

 2 char *
 3 net_rt_iflist(int family, int flags, size_t *lenp)
 4 {
 5     int    mib[6];
 6     char   *buf;

 7     mib[0] = CTL_NET;
 8     mib[1] = AF_ROUTE;
 9     mib[2] = 0;
10     mib[3] = family;              /* somente endereços dessa família */
11     mib[4] = NET_RT_IFLIST;
12     mib[5] = flags;               /* índice de interface ou 0 */
13     if (sysctl(mib, 6, NULL, lenp, NULL, 0) < 0)
14         return (NULL);
```

Figura 18.15 Chamada a `sysctl` para retornar a lista de interface (*continua*).

```
15      if ( (buf = malloc(*lenp)) == NULL)
16          return (NULL);
17      if (sysctl(mib, 6, buf, lenp, NULL, 0) < 0) {
18          free(buf);
19          return (NULL);
20      }
21      return (buf);
22  }
```

libroute/net_rt_iflist.c

Figura 18.15 Chamada a `sysctl` para retornar a lista de interface (*continuação*).

7–14 O array `mib` é inicializado como mostrado na Figura 18.12 para retornar a lista de interface e todos os endereços configurados da família especificada. `sysctl` é então chamada duas vezes. Na primeira chamada, o terceiro argumento é nulo, o que retorna o tamanho do buffer requerido para armazenar todas as informações sobre a interface na variável apontada por `lenp`.

15–21 Um espaço é então alocado ao buffer e `sysctl` é chamada novamente, agora com um terceiro argumento não-nulo. Desta vez, a variável apontada por `lenp` retornará com o volume de informações armazenado no buffer e será alocada pelo chamador. Um ponteiro para o buffer também é retornado ao chamador.

> Como o tamanho da tabela de roteamento ou o número de interfaces pode mudar entre as duas chamadas a `sysctl`, o valor retornado pela primeira chamada contém um fator de incerteza de 10% (páginas 639 e 640 do TCPv2).

A Figura 18.16 mostra a primeira metade da função `get_ifi_info`.

route/get_ifi_info.c

```
 3  struct ifi_info *
 4  get_ifi_info(int family, int doaliases)
 5  {
 6      int     flags;
 7      char    *buf, *next, *lim;
 8      size_t len;
 9      struct if_msghdr *ifm;
10      struct ifa_msghdr *ifam;
11      struct sockaddr *sa, *rti_info[RTAX_MAX];
12      struct sockaddr_dl *sdl;
13      struct ifi_info *ifi, *ifisave, *ifihead, **ifipnext;
14      buf = Net_rt_iflist(family, 0, &len);
15      ifihead = NULL;
16      ifipnext = &ifihead;
17      lim = buf + len;
18      for (next = buf; next < lim; next += ifm->ifm_msglen) {
19           ifm = (struct if_msghdr *) next;
20           if (ifm->ifm_type == RTM_IFINFO) {
21             if (((flags = ifm->ifm_flags) & IFF_UP) == 0)
22                 continue;      /* ignora se a interface não estiver ativa */
23             sa = (struct sockaddr *) (ifm + 1);
24             get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
25             if ( (sa = rti_info[RTAX_IFP]) != NULL) {
26                 ifi = Calloc(1, sizeof(struct ifi_info));
27                 *ifipnext = ifi; /* prev aponta para esse novo */
28                 ifipnext = &ifi->ifi_next; /* ponteiro para a próxima entra
                                                  aqui */
```

Figura 18.16 Função `get_ifi_info`, primeira metade (*continua*).

```
29                ifi->ifi_flags = flags;
30                if (sa->sa_family == AF_LINK) {
31                    sdl = (struct sockaddr_dl *) sa;
32                    ifi->ifi_index = sdl->sdl_index;
33                    if (sdl->sdl_nlen > 0)
34                        snprintf(ifi->ifi_name, IFI_NAME, "%*s",
35                                 sdl->sdl_nlen, &sdl->sdl_data[0]);
36                    else
37                        snprintf(ifi->ifi_name, IFI_NAME, "index %d",
38                                 sdl->sdl_index);
39                    if ( (ifi->ifi_hlen = sdl->sdl_alen) > 0)
40                        memcpy(ifi->ifi_haddr, LLADDR(sdl),
41                               min(IFI_HADDR, sdl->sdl_alen));
42                }
43            }
```

route/get_ifi_info.c

Figura 18.16 Função `get_ifi_info`, primeira metade (*continuação*).

6–14 Declaramos as variáveis locais e então chamamos nossa função `net_rt_iflist`.

17–19 O loop `for` examina cada mensagem de roteamento no buffer preenchido por `sysctl`. Assumimos que a mensagem é uma estrutura `if_msghdr` e examinamos o campo `ifm_type`. (Lembre-se de que os três primeiros membros de todas as três estruturas são idênticos, assim não importa qual delas utilizamos para examinar o membro *type*.)

Verificando se a interface está ativa

20–22 Uma estrutura `RTM_IFINFO` é retornada para cada interface. Se a interface não estiver ativa, ela é ignorada.

Determinando quais estruturas de endereço de soquete estão presentes

23–24 `sa` aponta para a primeira estrutura de endereço de soquete seguindo a estrutura `if_msghdr`. Nossa função `get_rtaddrs` inicializa o array `rti_info`, dependendo de quais estruturas de endereço de soquete estão presentes.

Tratando o nome de interface

25–43 Se a estrutura de endereço de soquete com o nome da interface estiver presente, uma estrutura `ifi_info` é alocada e os flags da interface são armazenados. A família esperada dessa estrutura de endereço de soquete é `AF_LINK`, que indica uma estrutura de endereço de soquete do enlace de dados. Armazenamos o índice de interface no membro `ifi_index`. Se o membro `sdl_nlen` for não-zero, o nome de interface é então copiado para a estrutura `ifi_info`. Caso contrário, uma string contendo o índice de interface é armazenada como o nome. Se o membro `sdl_alen` for não-zero, o endereço de hardware (por exemplo, o endereço Ethernet) é então copiado para a estrutura `ifi_info` e seu comprimento também é retornado como `ifi_hlen`.

A Figura 18.17 mostra a segunda metade da nossa função `get_ifi_info`, que retorna os endereços IP para a interface.

Retornando endereços IP

44–65 Uma mensagem `RTM_NEWADDR` é retornada por `sysctl` para cada endereço associado com a interface: o endereço primário e todos os aliases. Se já tivermos preenchido o endereço IP para essa interface, estamos então lidando com um alias. Nesse caso, se o chamador quiser o endereço do alias, deveremos alocar memória para uma outra estrutura `ifi_info`, copiar os campos que foram preenchidos e então preencher os endereços que retornaram.

Retornando os endereços de destino e de broadcast

66–75 Se a interface suportar broadcasting, o endereço de broadcast é retornado; se a interface for ponto a ponto, o endereço de destino é retornado.

route/get_ifi_info.c

```
44          } else if (ifm->ifm_type == RTM_NEWADDR) {
45              if (ifi->ifi_addr) {  /* já tem um endereço IP para i/f */
46                  if (doaliases == 0)
47                      continue;

48                  /* temos um novo endereço IP para a interface existente */
49                  ifisave = ifi;
50                  ifi = Calloc(1, sizeof(struct ifi_info));
51                  *ifipnext = ifi;  /* anterior aponta para esse novo */
52                  ifipnext = &ifi->ifi_next; /* ponteiro para próxima entra aqui */
53                  ifi->ifi_flags = ifisave->ifi_flags;
54                  ifi->ifi_index = ifisave->ifi_index;
55                  ifi->ifi_hlen = ifisave->ifi_hlen;
56                  memcpy(ifi->ifi_name, ifisave->ifi_name, IFI_NAME);
57                  memcpy(ifi->ifi_haddr, ifisave->ifi_haddr, IFI_HADDR);
58              }

59              ifam = (struct ifa_msghdr *) next;
60              sa = (struct sockaddr *) (ifam + 1);
61              get_rtaddrs(ifam->ifam_addrs, sa, rti_info);

62              if ( (sa = rti_info[RTAX_IFA]) != NULL) {
63                  ifi->ifi_addr = Calloc(1, sa->sa_len);
64                  memcpy(ifi->ifi_addr,sa, sa->sa_len);
65              }

66              if ((flags & IFF_BROADCAST) && (sa = rti_info[RTAX_BRD]) != NULL) {
67                  ifi->ifi_brdaddr = Calloc(1, sa->sa_len);
68                  memcpy(ifi->ifi_brdaddr,sa, sa->sa_len);
69              }

70              if ((flags & IFF_POINTOPOINT) &&
71                  (sa = rti_info[RTAX_BRD]) != NULL) {
72                  ifi->ifi_dstaddr = Calloc(1, sa->sa_len);
73                  memcpy(ifi->ifi_dstaddr,sa, sa->sa_len);
74              }

75          } else
76              err_quit("unexpected message type %d", ifm->ifm_type);
77      }
78      /* "ifihead" aponta para a primeira estrutura na lista vinculada */
79      return (ifihead);  /* aponta para a primeira estrutura na lista vinculada */
80  }
```

route/get_ifi_info.c

Figura 18.17 Função `get_ifi_info`, segunda metade.

18.6 Funções de nome e índice de interface

A RFC 3493 (Gilligan *et al*., 2003) define quatro funções que lidam com nomes e índices de interface. Essas quatro funções são utilizadas em vários lugares em que é necessário descrever uma interface. Elas foram introduzidas para utilização com a API do IPv6, como descreveremos nos Capítulos 21 e 27, mas também encontramos índices de interface na API do IPv4 (por exemplo, na chamada `IP_RECVIF` ou nos endereços de soquetes `AF_LINK` vistos no soquete de roteamento). O conceito básico é que cada interface tem um nome único e um índice positivo único (0 nunca é utilizado como um índice).

```
#include <net/if.h>

unsigned int if_nametoindex(const char *ifname);
```

Retorna: índice positivo da interface se OK, 0 no erro

```
char *if_indextoname(unsigned int ifindex, char *ifname);
```

Retorna: ponteiro para o nome da interface se OK, NULL no erro

```
struct if_nameindex *if_nameindex(void);
```

Retorna: ponteiro não-nulo se OK, NULL em erro

```
void if_freenameindex(struct if_nameindex *ptr);
```

`if_nametoindex` retorna o índice da interface cujo nome é *ifname*. `if_indextoname` retorna um ponteiro para o nome da interface dado seu *ifindex*. O argumento *ifname* aponta para um buffer do tamanho `IFNAMSIZ` (definido incluindo o cabeçalho `<net/if.h>`; também mostrado na Figura 17.2) que o chamador deve alocar para armazenar o resultado e esse ponteiro também é o valor de retorno no sucesso.

`if_nameindex` retorna um ponteiro para um array de estruturas `if_nameindex` como a seguir:

```
struct if_nameindex {
   unsigned int   if_index;     /* 1, 2, ... */
   char           *if_name;     /* nome terminado por nulo: "le0", ... */
};
```

A entrada final nesse array contém uma estrutura com um `if_index` de 0 e um `if_name` que é um ponteiro nulo. A memória para esse array, juntamente com os nomes apontados pelos membros do array, é obtida dinamicamente e é retornada chamando `if_freenameindex`.

Agora, fornecemos uma implementação dessas quatro funções utilizando soquetes de roteamento.

Função `if_nametoindex`

A Figura 18.18 mostra a função `if_nametoindex`.

libroute/if_nametoindex.c

```
 1 #include   "unpifi.h"
 2 #include   "unproute.h"

 3 unsigned int
 4 if_nametoindex(const char *name)
 5 {
 6     unsigned int idx, namelen;
 7     char   *buf, *next, *lim;
 8     size_t len;
 9     struct if_msghdr *ifm;
10     struct sockaddr *sa, *rti_info[RTAX_MAX];
11     struct sockaddr_dl *sdl;

12     if ( (buf = net_rt_iflist(0, 0, &len)) == NULL)
13         return (0);

14     namelen = strlen(name);
15     lim = buf + len;
```

Figura 18.18 Retornando um índice de interface dado seu nome (*continua*).

```
16      for (next = buf; next < lim; next += ifm->ifm_msglen) {
17          ifm = (struct if_msghdr *) next;
18          if (ifm->ifm_type == RTM_IFINFO) {
19              sa = (struct sockaddr *) (ifm + 1);
20              get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
21              if ( (sa = rti_info[RTAX_IFP]) != NULL) {
22                  if (sa->sa_family == AF_LINK) {
23                      sdl = (struct sockaddr_dl *) sa;
24                      if (sdl->sdl_nlen == namelen
25                          && strncmp(&sdl->sdl_data[0], name,
26                                     sdl->sdl_nlen) == 0) {
27                          idx = sdl->sdl_index; /* salva antes de chamar free()*/
28                          free(buf);
29                          return (idx);
30                      }
31                  }
32              }
33          }
34      }
35      free(buf);
36      return (0);             /* nenhuma correspondência com o nome */
37  }
```

libroute/if_nametoindex.c

Figura 18.18 Retorna um índice de interface dado seu nome (*continuação*).

Obtendo a lista de interface

12–13 Nossa função `net_rt_iflist` retorna a lista de interface.

Processando somente mensagens `RTM_IFINFO`

17–30 Processamos as mensagens no buffer (Figura 18.14), examinando somente as mensagens `RTM_IFINFO`. Quando localizamos uma, chamamos nossa função `get_rtaddrs` para configurar os ponteiros como estruturas de endereço de soquete e, se uma estrutura do nome de interface estiver presente (o elemento `RTAX_IFP` do array `rti_info`), o nome da interface é comparado com o argumento.

Função `if_indextoname`

A próxima função, `if_indextoname`, é mostrada na Figura 18.19.

libroute/if_indextoname.c

```
 1 #include   "unpifi.h"
 2 #include   "unproute.h"

 3 char *
 4 if_indextoname(unsigned int idx, char *name)
 5 {
 6     char   *buf, *next, *lim;
 7     size_t len;
 8     struct if_msghdr *ifm;
 9     struct sockaddr *sa, *rti_info[RTAX_MAX];
10     struct sockaddr_dl *sdl;

11     if ( (buf = net_rt_iflist(0, idx, &len)) == NULL)
12         return (NULL);

13     lim = buf + len;
14     for (next = buf; next < lim; next += ifm->ifm_msglen) {
```

Figura 18.19 Retornando um nome de interface dado seu índice (*continua*).

```
15          ifm = (struct if_msghdr *) next;
16          if (ifm->ifm_type == RTM_IFINFO) {
17              sa = (struct sockaddr *) (ifm + 1);
18              get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
19              if ( (sa = rti_info[RTAX_IFP]) != NULL) {
20                  if (sa->sa_family == AF_LINK) {
21                      sdl = (struct sockaddr_dl *) sa;
22                      if (sdl->sdl_index == idx) {
23                          int    slen = min(IFNAMSIZ - 1, sdl->sdl_nlen);
24                          strncpy(name, sdl->sdl_data, slen);
25                          name[slen] = 0; /* termina com nulo */
26                          free(buf);
27                          return (name);
28                      }
29                  }
30              }
31          }
32      }
33      free(buf);
34      return (NULL);          /* nenhuma correspondência para o índice */
35  }
```

libroute/if_indextoname.c

Figura 18.19 Retornando um nome de interface dado seu índice (*continuação*).

Essa função é quase idêntica à anterior, mas, em vez de procurar um nome de interface, comparamos o índice de interface contra o argumento do chamador. Além disso, o segundo argumento para a nossa função `net_rt_iflist` é o índice desejado, assim, o resultado deve conter as informações somente sobre a interface desejada. Quando uma correspondência é encontrada, o nome da interface é retornado e também é terminado por caractere nulo.

Função `if_nameindex`

A próxima função, `if_nameindex`, retorna um array das estruturas `if_nameindex` contendo todos os nomes e índices da interface. Ela é mostrada na Figura 18.20.

libroute/if_nameindex.c

```
 1 #include   "unpifi.h"
 2 #include   "unproute.h"
 3 struct if_nameindex *
 4 if_nameindex(void)
 5 {
 6     char   *buf, *next, *lim;
 7     size_t len;
 8     struct if_msghdr *ifm;
 9     struct sockaddr *sa, *rti_info[RTAX_MAX];
10     struct sockaddr_dl *sdl;
11     struct if_nameindex *result, *ifptr;
12     char   *namptr;
13     if ( (buf = net_rt_iflist(0, 0, &len)) == NULL)
14         return (NULL);
15     if ( (result = malloc(len)) == NULL) /* superestimado */
16         return (NULL);
17     ifptr = result;
18     namptr = (char *) result + len; /* nomes iniciam no final do buffer */
19     lim = buf + len;
20     for (next = buf; next < lim; next += ifm->ifm_msglen) {
```

Figura 18.20 Retornando todos os nomes e índices de interface (*continua*).

```
21          ifm = (struct if_msghdr *) next;
22          if (ifm->ifm_type == RTM_IFINFO) {
23              sa = (struct sockaddr *) (ifm + 1);
24              get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
25              if ( (sa = rti_info[RTAX_IFP]) != NULL) {
26                  if (sa->sa_family == AF_LINK) {
27                      sdl = (struct sockaddr_dl *) sa;
28                      namptr -= sdl->sdl_nlen + 1;
29                      strncpy(namptr, &sdl->sdl_data[0], sdl->sdl_nlen);
30                      namptr[sdl->sdl_nlen] = 0; /* termina com nulo */
31                      ifptr->if_name = namptr;
32                      ifptr->if_index = sdl->sdl_index;
33                      ifptr++;
34                  }
35              }
36          }
37      }
38      ifptr->if_name = NULL;    /* marca o final das estruturas de array */
39      ifptr->if_index = 0;
40      free(buf);
41      return (result); /* chamador deve liberar isso com free() ao concluir */
42  }
```

libroute/if_nameindex.c

Figura 18.20 Retornando todos os nomes e índices de interface (*continuação*).

Obtenção da lista de interface, alocação do espaço ao resultado

13–18 Chamamos nossa função `net_rt_iflist` para retornar a lista de interface. Também utilizamos o tamanho retornado como o tamanho do buffer que alocamos para conter o array das estruturas `if_nameindex` que retornamos. Isso é uma superestimativa, mas é mais simples do que fazer duas passagens pela lista de interfaces: uma para contar o número de interfaces e o tamanho total dos nomes e outra para preencher as informações. Criamos o array `if_nameindex` no começo desse buffer e armazenamos os nomes de interface começando no final do buffer.

Processando somente mensagens `RTM_IFINFO`

22–36 Processamos todas as mensagens procurando mensagens `RTM_IFINFO` e as estruturas de endereço de soquete do enlace de dados que se seguem. O nome e o índice de interface são armazenados no array que estamos construindo.

Término do array

38–39 A entrada final no array tem um `if_name` nulo e um índice de 0.

Função `if_freenameindex`

A função final, mostrada na Figura 18.21, libera a memória que foi alocada ao array das estruturas `if_nameindex` e os nomes contidos nelas.

libroute/if_nameindex.c

```
43  void
44  if_freenameindex(struct if_nameindex *ptr)
45  {
46      free(ptr);
47  }
```

libroute/if_nameindex.c

Figura 18.21 Liberando a memória alocada por `if_nameindex`.

Essa função é trivial porque armazenamos tanto o array das estruturas como os nomes no mesmo buffer. Se tivéssemos chamado `malloc` para cada nome, para liberar a memória, teríamos de examinar o array inteiro, utilizar `free` para liberar a memória para cada nome e então liberar o array.

18.7 Resumo

A última estrutura de endereço de soquete que encontramos neste texto é `sockaddr_dl`, a estrutura de endereço de soquete de enlace de dados de comprimento variável. Kernels derivados do Berkeley associam essas estruturas a interfaces, retornando o nome e o índice de interface e o endereço de hardware em uma dessas estruturas.

Cinco tipos de mensagens podem ser gravados em um soquete de roteamento por um processo e 15 diferentes mensagens podem ser retornadas pelo kernel assincronamente em um soquete de roteamento. Mostramos um exemplo em que o processo solicita ao kernel as informações sobre uma entrada da tabela de roteamento e o kernel responde com todos os detalhes. Essas respostas do kernel contêm até oito estruturas de endereço de soquete, e temos de analisar sintaticamente essa mensagem para obter todas as informações.

A função `sysctl` é uma maneira geral de buscar e armazenar parâmetros de SO. As informações em que estamos interessados com `sysctl` são:

- Fazer dump da lista de interface
- Fazer dump da tabela de roteamento
- Fazer dump da cache ARP

As alterações requeridas pelo IPv6 para a API de soquetes incluem quatro funções para mapear entre nomes de interface e seus índices. A cada interface é atribuído um índice positivo único. Implementações derivadas do Berkeley já associam um índice a cada interface, portanto, podemos implementar facilmente essas funções utilizando `sysctl`.

Exercícios

18.1 O que você esperaria que o campo `sdl_len` de uma estrutura de endereço de soquete de enlace de dados contivesse para um dispositivo denominado `eth10` cujo endereço da camada de enlace é um IEEE EUI-64 de 64 bits?

18.2 Na Figura 18.6, desative a opção de soquete `SO_USELOOPBACK` antes de chamar `write`. O que acontece?

Capítulo 19

Soquetes de Gerenciamento de Chaves

19.1 Visão geral

Com a introdução da arquitetura de segurança para IP (IPSec, descrito na RFC 2401 [Kent e Atkinson, 1998a]), foi necessário um mecanismo-padrão para gerenciar chaves secretas de cifragem e autenticação. A RFC 2367 (McDonald, Metz e Phan, 1998) introduziu uma API genérica de gerenciamento de chaves que pode ser utilizada no IPSec e em outros serviços de segurança de rede. Semelhante aos soquetes de roteamento (Capítulo 18), essa API cria uma nova família de protocolos, o domínio `PF_KEY`. Como ocorre com os soquetes de roteamento, o único tipo de soquete suportado no domínio da chave é um soquete bruto.

> Como descrito na Seção 4.2, na maioria dos sistemas, `AF_KEY` seria definido com o mesmo valor de `PF_KEY`. Entretanto, a RFC 2367 é bem específica quanto ao fato de `PF_KEY` ser a constante que deve ser utilizada com os soquetes de gerenciamento de chaves.

Abrir um soquete de gerenciamento de chaves bruto exige privilégios. Nos sistemas em que os privilégios estão segmentados, deve haver um privilégio individual para abrir soquetes de gerenciamento de chaves. Nos sistemas UNIX regulares, abrir um soquete de gerenciamento de chaves é limitado ao superusuário.

O IPSec fornece serviços de segurança para pacotes com base nas *associações de segurança* (*security associations* – SAs). Uma SA descreve uma combinação de endereços de origem e de destino (e, opcionalmente, protocolo e portas de transporte), mecanismos (por exemplo, autenticação) e material de chaveamento. Mais de uma SA (por exemplo, autenticação e cifragem) pode ser aplicada a um único fluxo de tráfego. O conjunto das associações de segurança armazenado para utilização em um sistema é chamado de banco de dados de associação de segurança (*security association database* – SADB).

O SADB em um sistema pode ser utilizado por outros protocolos além do IPSec; por exemplo, OSPFv2, RIPv2, RSVP e Mobile-IP também podem ter entradas no SADB. Por essa razão, soquetes `PF_KEY` não são específicos ao IPSec.

O IPSec também requer um banco de dados da *política de segurança* (*security policy database* – SPDB). O banco de dados da política de segurança descreve os requisitos para tráfe-

go; por exemplo, o tráfego entre os hosts A e B deve ser autenticado utilizando o IPSec AH, e quaisquer outros que não o sejam devem ser descartados. O SADB descreve como realizar os procedimentos requeridos de segurança, como, por exemplo, se o tráfego entre os hosts A e B utiliza o IPSec AH, então o SADB contém o algoritmo e a chave a serem utilizados. Infelizmente, não há nenhum mecanismo-padrão para manter o SPDB. O `PF_KEY` permite a manutenção do SADB, mas não a do SPDB. A implementação IPSec do KAME utiliza as extensões `PF_KEY` para manutenção do SPDB, mas não há nenhum padrão para isso.

Três tipos de operações são suportados nos soquetes de gerenciamento de chaves:

1. Um processo pode enviar uma mensagem ao kernel, e a todos os outros processos com soquetes de gerenciamento de chaves abertos, gravando em um soquete de gerenciamento de chaves. Essa é a maneira como as entradas no SADB são adicionadas e excluídas e como os processos que realizam sua própria segurança, como o OSPFv2, solicitam uma chave a partir de um daemon de gerenciamento de chaves.
2. Um processo pode ler uma mensagem do kernel (ou de outro processo) em um soquete de gerenciamento de chaves. O kernel pode utilizar esse recurso para solicitar que um daemon de gerenciamento de chaves instale uma associação de segurança para uma nova sessão TCP que a política exige que seja protegida.
3. Um processo pode enviar uma mensagem de solicitação de dump ao kernel e ele responderá com um dump do SADB atual. Esse é um recurso de depuração que talvez não esteja disponível em todos os sistemas.

19.2 Leitura e gravação

Todas as mensagens em um soquete de gerenciamento de chaves têm o mesmo cabeçalho básico, mostrado na Figura 19.1. Cada mensagem pode ser seguida por várias extensões, dependendo de quais informações adicionais estão disponíveis ou são requeridas. Todas as estruturas apropriadas são definidas incluindo `<net/pfkeyv2.h>`. Cada mensagem e extensão é alinhada a 64 bits e o seu comprimento é um múltiplo de 64 bits. Todos os campos de comprimento estão em unidades de 64 bits, isto é, um comprimento de 1 significa 8 bytes. Qualquer extensão que não requer um volume de dados suficiente para ser um múltiplo de 64 bits de comprimento é preenchida com o próximo múltiplo de 64 bits. O valor desse preenchimento não é definido.

O valor `sadb_msg_type` determina qual dos 10 comandos de gerenciamento de chaves é invocado. Esses tipos de mensagem estão listados na Figura 19.2. Cada cabeçalho `sadb_msg` será seguido por zero ou mais extensões. A maioria dos tipos de mensagem possui extensões requeridas e opcionais; descreveremos esses tipos à medida que descrevemos cada tipo de mensagem. Os 16 tipos de extensões, juntamente com o nome da estrutura que define cada extensão, estão listados na Figura 19.4.

```
struct sadb_msg {
  u_int8_t sadb_msg_version;  /* PF_KEY_V2 */
  u_int8_t sadb_msg_type;     /* veja Figura 19.2 */
  u_int8_t sadb_msg_errno;    /* indicação de erro */
  u_int8_t sadb_msg_satype;   /* veja Figura 19.3 */
  u_int16_t sadb_msg_len;     /* comprimento do cabeçalho + extensões / 8 */
  u_int16_t sadb_msg_reserved;  /* zero na transmissão, ignorado na recepção */
  u_int32_t sadb_msg_seq;     /* número de seqüência */
  u_int32_t sadb_msg_pid;     /* ID de processo da origem ou destino */
};
```

Figura 19.1 Cabeçalho da mensagem de gerenciamento de chaves.

Tipo de mensagem	Para o kernel?	Do kernel?	Descrição
SADB_ACQUIRE	•	•	Solicitação da criação de uma entrada no SADB
SADB_ADD	•	•	Adição de uma entrada completa no banco de dados de segurança
SADB_DELETE	•	•	Exclusão de uma entrada
SADB_DUMP	•	•	Dump do SADB (depuração)
SADB_EXPIRE		•	Notificação da expiração de uma entrada
SADB_FLUSH	•	•	Esvaziamento de todo o banco de dados
SADB_GET	•	•	Obtenção de uma entrada
SADB_GETSPI	•	•	Alocação de um SPI para criar uma entrada no SADB
SADB_REGISTER	•		Registro como um respondente a SADB_ACQUIRE
SADB_UPDATE	•	•	Atualização de uma entrada parcial no SADB

Figura 19.2 Tipos de mensagens trocadas em um soquete PF_KEY.

Tipo de associação de segurança	Descrição
SADB_SATYPE_AH	Cabeçalho de autenticação IPSec
SADB_SATYPE_ESP	Payload de segurança para encapsulamento do IPSec
SADB_SATYPE_MIP	Autenticação IP móvel
SADB_SATYPE_OSPFV2	Autenticação OSPFv2
SADB_SATYPE_RIPV2	Autenticação RIPv2
SADB_SATYPE_RSVP	Autenticação RSVP
SADB_SATYPE_UNSPECIFIED	Não especificado; válido somente em solicitações

Figura 19.3 Tipos de SAs.

Tipo de cabeçalho da extensão	Descrição	Estrutura
SADB_EXT_ADDRESS_DST	Endereço de destino da SA	sadb_address
SADB_EXT_ADDRESS_PROXY	Endereço do proxy da SA	sadb_address
SADB_EXT_ADDRESS_SRC	Endereço da origem da SA	sadb_address
SADB_EXT_IDENTITY_DST	Identidade de destino	sadb_ident
SADB_EXT_IDENTITY_SRC	Identidade de origem	sadb_ident
SADB_EXT_KEY_AUTH	Autenticação de chave	sadb_key
SADB_EXT_KEY_ENCRYPT	Chave de cifragem	sadb_key
SADB_EXT_LIFETIME_CURRENT	Tempo de vida atual da SA	sadb_lifetime
SADB_EXT_LIFETIME_HARD	Limite rígido (hard) de tempo de vida da SA	sadb_lifetime
SADB_EXT_LIFETIME_SOFT	Limite flexível (soft) de tempo de vida da SA	sadb_lifetime
SADB_EXT_PROPOSAL	Situação proposta	sadb_prop
SADB_EXT_SA	SA	sadb_sa
SADB_EXT_SENSITIVITY	Grau de sensibilidade da SA	sadb_sens
SADB_EXT_SPIRANGE	Intervalo aceitável de valores SPI	sadb_spirange
SADB_EXT_SUPPORTED_AUTH	Algoritmos de autenticação suportados	sadb_supported
SADB_EXT_SUPPORTED_ENCRYPT	Algoritmos de cifragem suportados	sadb_supported

Figura 19.4 Tipos de extensão PF_KEY.

Agora, mostramos vários exemplos, mensagens e extensões que fazem parte de várias operações comuns nos soquetes de gerenciamento de chaves.

19.3 Dump do Security Association Database (SADB)

Para fazer dump do SADB atual, um processo utiliza a mensagem SADB_DUMP. Essa é a mensagem mais simples a enviar, pois não requer nenhuma extensão, simplesmente o cabeçalho sadb_msg de 16 bytes. Depois que o processo envia a mensagem SADB_DUMP ao kernel em um soquete de gerenciamento de chaves, o kernel responde com uma série de mensagens SADB_DUMP de volta ao mesmo soquete, cada uma com uma entrada do SADB. O final da lista é indicado por uma mensagem com um valor de 0 para o campo sadb_msg_seq.

O tipo da SA pode ser limitado configurando o campo sadb_msg_satype na solicitação como um dos valores na Figura 19.3. Se esse campo for configurado como SADB_SATYPE_UNSPEC, todas as SAs no banco de dados são retornadas. Caso contrário, somente as SAs do tipo especificado são retornadas. Nem todos os tipos de associações de segurança são suportados por todas as implementações. A implementação KAME suporta apenas SAs do IPSec (SADB_SATYPE_AH e SADB_SATYPE_ESP), assim, uma tentativa de fazer o dump das entradas ADB_SATYPE_RIPV2 no SADB resultará em uma resposta de erro com errno EINVAL. Ao solicitar um tipo específico cuja tabela está vazia, o errno ENOENT é retornado.

Nosso programa para fazer o dump do SADB segue na Figura 19.5.

key/dump.c

```
 1 void
 2 sadb_dump(int type)
 3 {
 4     int     s;
 5     char    buf[4096];
 6     struct sadb_msg msg;
 7     int     goteof;

 8     s = Socket(PF_KEY, SOCK_RAW, PF_KEY_V2);

 9     /*Constrói e grava a solicitação SADB_DUMP */
10     bzero(&msg, sizeof(msg));
11     msg.sadb_msg_version = PF_KEY_V2;
12     msg.sadb_msg_type = SADB_DUMP;
13     msg.sadb_msg_satype = type;
14     msg.sadb_msg_len = sizeof(msg) / 8;
15     msg.sadb_msg_pid = getpid();
16     printf("Sending dump message:\n");
17     print_sadb_msg(&msg, sizeof(msg));
18     Write(s, &msg, sizeof(msg));

19     printf("\nMessages returned:\n");
20     /* Lê e imprime respostas SADB_DUMP até terminar */
21     goteof = 0;
22     while (goteof == 0) {
23         int     msglen;
24         struct sadb_msg *msgp;

25     msglen = Read(s, &buf, sizeof(buf));
26     msgp = (struct sadb_msg *) &buf;
27     print_sadb_msg(msgp, msglen);
28     if (msgp->sadb_msg_seq == 0)
29         goteof = 1;
30     }
31     close(s);
32 }

33 int
34 main(int argc, char **argv)
```

Figura 19.5 O programa para emitir o comando SADB_DUMP no soquete de gerenciamento de chaves (*continua*).

```
35 {
36     int     satype = SADB_SATYPE_UNSPEC;
37     int     c;

38     opterr = 0;           /* não queremos getopt() gravando em stderr */
39     while ( (c = getopt(argc, argv, "t:")) != -1) {
40         switch (c) {
41         case 't':
42             if ( (satype = getsatypebyname(optarg)) == -1)
43                 err_quit("invalid -t option %s", optarg);
44             break;

45         default:
46             err_quit("unrecognized option: %c", c);
47         }
48     }

49     sadb_dump(satype);
50 }
```

key/dump.c

Figura 19.5 O programa para emitir o comando `SADB_DUMP` no soquete de gerenciamento de chaves (*continuação*).

Esse é nosso primeiro encontro com a função `getopt` do POSIX. O terceiro argumento é uma string de caracteres que especifica os caracteres que permitimos como argumentos da linha de comando, simplesmente `t` nesse exemplo. Ele é seguido por dois-pontos, indicando que a opção aceita um argumento. Nos programas que aceitam mais de uma opção, elas são concatenadas; por exemplo, a Figura 29.7 passa `0i:l:v` para indicar que aceita quatro opções; `i` e `l` aceitam um argumento e `0` e `v` não aceitam. Essa função trabalha com quatro variáveis globais definidas incluindo `<unistd.h>`.

```
extern char   *optarg;
extern int    optind, opterr, optopt;
```

Antes de chamar `getopt`, configuramos `opterr` como 0 para evitar que a função grave mensagens de erro no erro-padrão se ocorrer um erro, pois queremos tratar deles. O POSIX afirma que, se o primeiro caractere do terceiro argumento for dois-pontos, ele também impedirá que a função grave no erro-padrão, mas nem todas as implementações suportam isso.

Abertura do soquete `PF_KEY`

1–8 Primeiro, abrimos um soquete `PF_KEY`. Isso exige privilégios de sistema, como descrito anteriormente, visto que permite acesso a material de manipulação de chaves sensível.

Construção da solicitação `SADB_DUMP`

9–15 Primeiro, zeramos a estrutura `sadb_msg` para poder pular a inicialização dos campos que queremos que permaneçam zerados. Preenchemos cada campo restante na estrutura `sadb_msg` individualmente. Todas as mensagens nos soquetes abertos, com `PF_KEY_V2` como o terceiro argumento, também devem utilizar `PF_KEY_V2` como a versão da mensagem. O tipo da mensagem é `SADB_DUMP`. Configuramos o comprimento como o comprimento do cabeçalho básico sem extensões, visto que a mensagem de dump não aceita extensões. Por fim, configuramos o ID do processo (*process ID* – PID) como o nosso próprio PID, uma vez que todas as mensagens devem ser identificadas pelo PID do remetente.

Exibição e gravação de uma mensagem `SADB_DUMP`

16–18 Exibimos a mensagem utilizando nossa rotina da `print_sadb_msg`. Não mostraremos essa rotina visto que é longa e desinteressante, mas ela é incluída no código-fonte livremente disponível. Ela aceita uma mensagem gravada ou recebida de um soquete de gerenciamento

de chaves e imprime todas as informações sobre a mensagem de uma forma legível por humanos. Em seguida, gravamos a mensagem no soquete.

Leitura das respostas

19–30 Fazemos o loop, lendo e imprimindo as respostas com a nossa função `print_sadb_msg`. A última mensagem na seqüência do dump tem um número de seqüência de mensagem de zero, assim, utilizamos esse número como nossa indicação de "fim de arquivo".

Fechamento do soquete `PF_KEY`

31 Por fim, fechamos o soquete que abrimos.

Tratamento dos argumentos da linha de comando

38–48 A função `main` tem pouco trabalho a fazer. Esse programa aceita um único argumento opcional, que é o tipo da SA para fazer o dump. Por padrão, o tipo é `SADB_SATYPE_UNSPEC`, que faz o dump de todas as SAs de qualquer tipo. Especificando um argumento da linha de comando, o usuário pode selecionar em qual tipo das SAs fazer o dump. Esse programa utiliza nossa função `getsatypebyname`, que retorna o valor do tipo para uma string de texto.

Chamada à rotina `sadb_dump`

49 Por fim, chamamos a função `sadb_dump` que definimos anteriormente para fazer todo o trabalho.

Execução de exemplo

O seguinte é uma execução de exemplo do programa de dump em um sistema com duas SAs estáticas.

```
macosx % dump
Sending dump message:
SADB Message Dump, errno 0, satype Unspecified, seq 0, pid 20623

Messages returned:
 SADB Message Dump, errno 0, satype IPsec AH, seq 1, pid 20623
   SA: SPI=258 Replay Window=0 State=Mature
   Authentication Algorithm: HMAC-MD5
   Encryption Algorithm: None
 [unknown extension 19]
 Current lifetime:
   0 allocations, 0 bytes
   added at Sun May 18 16:28:11 2003, never used
 Source address: 2.3.4.5/128 (IP proto 255)
 Dest address: 6.7.8.9/128 (IP proto 255)
 Authentication key, 128 bits: 0x20202020202020202020202020202020
SADB Message Dump, errno 0, satype IPsec AH, seq 0, pid 20623
 SA: SPI=257 Replay Window=0 State=Mature
   Authentication Algorithm: HMAC-MD5
   Encryption Algorithm: None
 [unknown extension 19]
 Current lifetime:
   0 allocations, 0 bytes
   added at Sun May 18 16:26:24 2003, never used
 Source address:   1.2.3.4/128 (IP proto 255)
 Dest address:   5.6.7.8/128 (IP proto 255)
 Authentication key, 128 bits: 0x10101010101010100101010101010101
```

19.4 Criando uma associação de segurança estática (SA)

O método mais simples e direto de adicionar uma SA é enviar uma mensagem SADB_ADD com todos os parâmetros preenchidos, presumivelmente especificados manualmente. Embora a especificação manual do material de manipulação de chaves não leve facilmente a alterações de chave, que são cruciais para evitar ataques de criptoanálise, ela é bem fácil de configurar: Alice e Bob concordam com uma chave e os algoritmos a utilizar fora da banda e prosseguem para utilizá-los. Mostramos os procedimentos necessários para criar e enviar uma mensagem SADB_ADD.

A mensagem SADB_ADD requer três extensões: SA, endereço e chave. Ela também pode conter opcionalmente outras extensões: tempo de vida, identidade e grau de sigilo. Primeiro, descrevemos as extensões requeridas. A extensão da SA é descrita pela estrutura sadb_sa, mostrada na Figura 19.6.

```
struct sadb_sa {
  u_int16_t sadb_sa_len;      /* comprimento da extensão / 8 */
  u_int16_t sadb_sa_exttype;  /* SADB_EXT_SA */
  u_int32_t sadb_sa_spi;      /* Security Parameters Index (SPI) */
  u_int8_t sadb_sa_replay;    /* tamanho da janela de replay, ou zero */
  u_int8_t sadb_sa_state;     /* estado da AS, veja Figura 19.7 */
  u_int8_t sadb_sa_auth;      /* algoritmo de autenticação, veja
                                 Figura 19.8 */
  u_int8_t sadb_sa_encrypt;   /* algoritmo de cifragem, veja Figura 19.8 */
  u_int32_t sadb_sa_flags;    /* máscara de bits de flags */
};
```

Figura 19.6 Extensão da SA.

O campo sadb_sa_spi contém o *Security Parameters Index* ou SPI. Esse valor, combinado com o endereço de destino e o protocolo em utilização (por exemplo, IPSec AH), identifica unicamente uma SA. Ao receber um pacote, esse valor é utilizado para pesquisar esse pacote na SA; ao enviar um pacote, esse valor é inserido no pacote para a outra extremidade utilizar. Ele não tem nenhum outro significado, portanto, esses valores podem ser alocados seqüencialmente, aleatoriamente ou utilizando qualquer método que o sistema de destino preferir. O campo sadb_sa_reply especifica o tamanho da janela para proteção contra replay. Como o uso de chaves estática impede a proteção contra replay, configuraremos isso como zero. O campo sadb_sa_state varia durante o ciclo de vida de uma SA criada dinamicamente, utilizando os valores na Figura 19.7. Entretanto, SAs criadas manualmente passam todo seu tempo no estado SADB_SASTATE_MATURE. Veremos os outros estados na Seção 19.5.

Os campos sadb_sa_auth e sadb_sa_encrypt especificam os algoritmos de autenticação e de cifragem para essa SA. Valores possíveis para esses campos estão listados na Figura 19.8. Há somente um valor de flag atualmente definido para o campo sadb_sa_flags, SADB_SAFLAGS_PFS. Esse flag requer uma *segurança perfeita de encaminhamento*, isto é, o valor dessa chave não deve ser dependente de quaisquer chaves anteriores ou de alguma chave-mestra. O valor desse flag é utilizado ao solicitar chaves a partir de uma aplicação de gerenciamento de chaves e não é utilizado ao adicionar associações estáticas.

As próximas extensões requeridas para um comando SADB_ADD são os endereços. Endereços de origem e de destino, especificados com SADB_EXT_ADDRESS_SRC e SADB_EXT_ADDRESS_DST, respectivamente, são requeridos. Um endereço de proxy, especificado com SADB_EXT_ADDRESS_PROXY, é opcional. Para mais detalhes sobre os ende-

Estado da SA	Descrição	Pode ser utilizado?
SADB_SASTATE_LARVAL	Em processo de criação	Não
SADB_SASTATE_MATURE	Completamente formada	Sim
SADB_SASTATE_DYING	Tempo de vida flexível expirou	Sim
SADB_SASTATE_DEAD	Tempo de vida rígido expirou	Não

Figura 19.7 Possíveis estados das SAs.

Algoritmo	Descrição	Referência
SADB_AALG_NONE	Nenhuma autenticação	
SADB_AALG_MD5HMAC	HMAC-MD5-96	RFC 2403
SADB_AALG_SHA1HMAC	HMAC-SHA-1-96	RFC 2404
SADB_EALG_NONE	Nenhuma cifragem	
SADB_EALG_DESCBC	DES-CBC	RFC2405
SADB_EALG_3DESCBC	3DES-CBC	RFC1851
SADB_EALG_NULL	NULL	RFC2410

Figura 19.8 Algoritmos de autenticação e de cifragem.

reços de proxy, consulte a RFC 2367 (McDonald, Metz e Phan, 1998). Os endereços são especificados utilizando uma extensão `sadb_address`, mostrada na Figura 19.9. O campo `sadb_address_exttype` determina o tipo de endereço que essa extensão fornece. O campo `sadb_address_proto` especifica a correspondência do protocolo IP com essa SA ou 0 para corresponder a todos os protocolos. O campo `sadb_address_prefixlen` descreve o prefixo do endereço que é significativo. Isso permite a uma SA corresponder a mais de um endereço. Um `sockaddr` da família apropriada (por exemplo, `sockaddr_in`, `sockaddr_in6`) segue a estrutura `sadb_address`. O número da porta nesse `sockaddr` é significativo somente se o `sadb_address_proto` especificar um protocolo que suporta números de porta (por exemplo, `IPPROTO_TCP`).

```
struct sadb_address {
  u_int16_t sadb_address_len;      /* comprimento da extensão + endereço
                                      / 8 */
  u_int16_t sadb_address_exttype;  /* SADB_EXT_ADDRESS_{SRC,DST,PROXY} */
  u_int8_t  sadb_address_proto;    /* protocolo IP, ou 0 para todos */
  u_int8_t  sadb_address_prefixlen; /* nº bits significativos no endereço */
  u_int16_t sadb_address_reserved; /* reservado para extensão */
};
                                   /* seguido pelo sockaddr apropriado */
```

Figura 19.9 Extensão de endereço.

As extensões finais requeridas para a mensagem `SADB_ADD` são a autenticação e as chaves de cifragem, especificadas com as extensões `SADB_EXT_KEY_AUTH` e `SADB_EXT_KEY_ENCRYPT`, representadas por uma estrutura `sadb_key` (Figura 19.10). A extensão de chave é muito simples e direta; o membro `sadb_key_exttype` define se ela é uma chave de autenticação ou de cifragem, o membro `sadb_key_bits` especifica o número de bits na chave e a própria chave segue a estrutura `sadb_key`.

```
struct sadb_key {
  u_int16_t sadb_key_len;       /* comprimento da extensão + chave / 8 */
  u_int16_t sadb_key_exttype;   /* SADB_EXT_KEY_{AUTH,ENCRYPT} */
  u_int16_t sadb_key_bits;      /* nº de bits na chave */
  u_int16_t sadb_key_reserved;  /* reservado para extensão */
};
                                /* seguido pelos dados da chave */
```

Figura 19.10 Extensão de chave.

key/add.c

```
33 void
34 sadb_add(struct sockaddr *src, struct sockaddr *dst, int type, int alg,
35           int spi, int keybits, unsigned char *keydata)
36 {
37     int    s;
38     char   buf[4096], *p;        /* XXX */
39     struct sadb_msg *msg;
40     struct sadb_sa *saext;
41     struct sadb_address *addrext;
42     struct sadb_key *keyext;
43     int    len;
44     int    mypid;

45     s = Socket(PF_KEY, SOCK_RAW, PF_KEY_V2);

46     mypid = getpid();

47     /* Constrói e grava a solicitação SADB_ADD */
48     bzero(&buf, sizeof(buf));
49     p = buf;
50     msg = (struct sadb_msg *) p;
51     msg->sadb_msg_version = PF_KEY_V2;
52     msg->sadb_msg_type = SADB_ADD;
53     msg->sadb_msg_satype = type;
54     msg->sadb_msg_pid = getpid();
55     len = sizeof(*msg);
56     p += sizeof(*msg);

57     saext = (struct sadb_sa *) p;
58     saext->sadb_sa_len = sizeof(*saext) / 8;
59     saext->sadb_sa_exttype = SADB_EXT_SA;
60     saext->sadb_sa_spi = htonl(spi);
61     saext->sadb_sa_replay = 0;  /* sem proteção contra replay com chaves
       estáticas */
62     saext->sadb_sa_state = SADB_SASTATE_MATURE;
63     saext->sadb_sa_auth = alg;
64     saext->sadb_sa_encrypt = SADB_EALG_NONE;
65     saext->sadb_sa_flags = 0;
66     len += saext->sadb_sa_len * 8;
67     p += saext->sadb_sa_len * 8;

68     addrext = (struct sadb_address *) p;
69     addrext->sadb_address_len = (sizeof(*addrext) + salen(src) + 7) / 8;
70     addrext->sadb_address_exttype = SADB_EXT_ADDRESS_SRC;
71     addrext->sadb_address_proto = 0;  /* qualquer protocolo */
72     addrext->sadb_address_prefixlen = prefix_all(src);
73     addrext->sadb_address_reserved = 0;
74     memcpy(addrext + 1, src, salen(src));
```

Figura 19.11 Programa para emitir o comando `SADB_ADD` no soquete de gerenciamento de chaves (*continua*).

```
 75      len += addrext->sadb_address_len * 8;
 76      p += addrext->sadb_address_len * 8;

 77      addrext = (struct sadb_address *) p;
 78      addrext->sadb_address_len = (sizeof(*addrext) + salen(dst) + 7) / 8;
 79      addrext->sadb_address_exttype = SADB_EXT_ADDRESS_DST;
 80      addrext->sadb_address_proto = 0;  /* qualquer protocolo */
 81      addrext->sadb_address_prefixlen = prefix_all(dst);
 82      addrext->sadb_address_reserved = 0;
 83      memcpy(addrext + 1, dst, salen(dst));
 84      len += addrext->sadb_address_len * 8;
 85      p += addrext->sadb_address_len * 8;

 86      keyext = (struct sadb_key *) p;
 87      /* "+7" trata dos requisitos de alinhamento */
 88      keyext->sadb_key_len = (sizeof(*keyext) + (keybits / 8) + 7) / 8;
 89      keyext->sadb_key_exttype = SADB_EXT_KEY_AUTH;
 90      keyext->sadb_key_bits = keybits;
 91      keyext->sadb_key_reserved = 0;
 92      memcpy(keyext + 1, keydata, keybits / 8);
 93      len += keyext->sadb_key_len * 8;
 94      p += keyext->sadb_key_len * 8;

 95      msg->sadb_msg_len = len / 8;
 96      printf("Sending add message:\n");
 97      print_sadb_msg(buf, len);
 98      Write(s, buf, len);

 99      printf("\nReply returned:\n");
100      /* Lê e imprime resposta SADB_ADD, descartando quaisquer outras */
101      for ( ; ; ) {
102          int       msglen;
103          struct sadb_msg *msgp;

104          msglen = Read(s, &buf, sizeof(buf));
105          msgp = (struct sadb_msg *) &buf;
106          if (msgp->sadb_msg_pid == mypid && msgp->sadb_msg_type == SADB_ADD) {
107             print_sadb_msg(msgp, msglen);
108             break;
109          }
110      }
111      close(s);
112  }
```

key/add.c

Figura 19.11 Programa para emitir o comando SADB_ADD no soquete de gerenciamento de chaves (*continuação*).

Mostramos nosso programa para adicionar uma entrada estática ao SADB na Figura 19.11.

Abrindo o soquete PF_KEY e salvando o PID

55–56 Como fizemos anteriormente, abrimos um soquete PF_KEY e salvamos nosso PID para utilização posterior.

Construindo um cabeçalho de mensagem comum

47–56 Construímos o cabeçalho de mensagem comum para a mensagem SADB_ADD. Não configuramos o elemento sadb_msg_len até um pouco antes de escrevermos a mensagem, uma vez que ele deve refletir o comprimento total da mensagem. A variável len mantém um comprimento variável da mensagem e o ponteiro p sempre aponta para o primeiro byte não-utilizado no buffer.

Acrescentando uma extensão da SA

57–67 Em seguida, adicionamos a extensão da SA requerida (Figura 19.6). O campo `sadb_sa_spi` deve estar na ordem de bytes de rede, assim, chamamos `htonl` no valor da ordem de host que foi passado para a função. Desativamos a proteção contra replay e configuramos o estado da SA (Figura 19.7) como `SADB_SASTATE_MATURE`. Configuramos o algoritmo de autenticação como o valor do algoritmo especificado na linha de comando e não especificamos nenhuma cifragem com `SADB_EALG_NONE`.

Acrescentando um endereço de origem

68–76 Adicionamos o endereço de origem à mensagem como uma extensão `SADB_EXT_ADDRESS_SRC`. Configuramos o protocolo como 0, para informar que essa associação se aplica a todos os protocolos. Configuramos o comprimento do prefixo como o comprimento apropriado da versão IP, isto é, 32 bits para o IPv4 e 128 bits para o IPv6. O cálculo do campo de comprimento adiciona 7 antes de dividir por 8, o que assegura que o comprimento reflete o preenchimento requerido para definir um limite de 64 bits como exigido por todas as extensões `PF_KEY`. O `sockaddr` é copiado depois do cabeçalho da extensão.

Acrescentando um endereço de destino

77–85 O endereço de destino é adicionado como uma extensão `SADB_EXT_ADDRESS_DST` exatamente da mesma maneira como o endereço de origem.

Acrescentando uma chave

86–94 Adicionamos uma chave de autenticação à mensagem como uma extensão `SADB_EXT_KEY_AUTH`. Calculamos o campo de comprimento da mesma maneira como os endereços, para adicionar o preenchimento requerido para a chave de comprimento variável. Configuramos o número de bits e copiamos os dados da chave para seguir o cabeçalho da extensão.

Gravando uma mensagem

95–98 Imprimimos a mensagem com nossa função `print_sadb_msg` e a gravamos no soquete.

Lendo a resposta

99–111 Lemos as mensagens no soquete até recebermos uma mensagem endereçada ao nosso PID, uma mensagem `SADB_ADD`. Em seguida, imprimimos essa mensagem com a função `print_sadb_msg` e terminamos.

Exemplo

Executamos nosso programa para enviar uma mensagem `SADB_ADD` ao tráfego entre 127.0.0.1 e 127.0.0.1; em outras palavras, no sistema local.

```
macosx % add 127.0.0.1 127.0.0.1 HMAC-SHA-1-96 160 \
                                 0123456789abcdef0123456789abcdef01234567

Sending add message:
SADB Message Add, errno 0, satype IPsec AH, seq 0, pid 6246
 SA: SPI=39030 Replay Window=0 State=Mature
   Authentication Algorithm: HMAC-SHA-1
   Encryption Algorithm: None
 Source address:  127.0.0.1/32
 Dest address:  127.0.0.1/32
 Authentication key, 160 bits: 0x0123456789abcdef0123456789abcdef01234567
```

```
Reply returned:
SADB Message Add, errno 0, satype IPsec AH, seq 0, pid 6246
 SA: SPI=39030 Replay Window=0 State=Mature
   Authentication Algorithm: HMAC-SHA-1
   Encryption Algorithm: None
 Source address:  127.0.0.1/32
 Dest address:  127.0.0.1/32
```

Observe que a resposta ecoa a solicitação sem a chave. Isso ocorre porque a resposta é enviada a todos os soquetes PF_KEY, mas diferentes soquetes PF_KEY podem pertencer a soquetes em diferentes domínios de proteção e dados chaveados não devem ultrapassar os domínios de proteção. Depois de adicionar a SA ao banco de dados, emitimos um ping a 127.0.0.1 para que a SA seja utilizada, e fazemos então um dump do banco de dados para ver o que foi adicionado.

```
macosx % dump
Sending dump message:
SADB Message Dump, errno 0, satype Unspecified, seq 0, pid 6283

Messages returned:
SADB Message Dump, errno 0, satype IPsec AH, seq 0, pid 6283
 SA: SPI=39030 Replay Window=0 State=Mature
   Authentication Algorithm: HMAC-SHA-1
   Encryption Algorithm: None
 [unknown extension 19]
 Current lifetime:
   36 allocations, 0 bytes
   added at Thu Jun 5 21:01:31 2003, first used at Thu Jun 5 21:15:07 2003
 Source address: 127.0.0.1/128 (IP proto 255)
 Dest address: 127.0.0.1/128 (IP proto 255)
 Authentication key, 160 bits: 0x0123456789abcdef0123456789abcdef01234567
```

Vemos que nesse dump o kernel alterou nosso protocolo IP de zero para 255. Isso é um artefato dessa implementação, não uma propriedade geral dos soquetes PF_KEY. Além disso, vemos que o kernel alterou o comprimento do prefixo de 32 para 128. Isso parece ser uma questão confusa entre o IPv4 e o IPv6 dentro do kernel. O kernel retorna uma extensão (numerada como 19) que nosso programa de dump não entende. Extensões desconhecidas são puladas utilizando o campo de comprimento. Uma extensão de tempo de vida (Figura 19.12) é retornada com as informações sobre o tempo de vida atual do SA.

```
struct sadb_lifetime {
  u_int16_t  sadb_lifetime_len;        /* comprimento da extensão / 8 */
  u_int16_t  sadb_lifetime_exttype;    /* SADB_EXT_LIFETIME_{SOFT,HARD,
                                          CURRENT} */
  u_int32_t  sadb_lifetime_allocations; /* n° conexões, extremidades ou
                                          fluxos */
  u_int64_t  sadb_lifetime_bytes;      /* n° bytes */
  u_int64_t  sadb_lifetime_addtime;    /* tempo da conexão, ou tempo desde
                                          a criação até a expiração */
  u_int64_t  sadb_lifetime_usetime;    /* tempo primeiro utilizado, ou
                                          tempo desde primeiro uso até
                                          expiração */
};
```

Figura 19.12 Extensão de tempo de vida.

Há três extensões diferentes de tempo de vida. As extensões SADB_LIFETIME_SOFT e SADB_LIFETIME_HARD especificam respectivamente os tempos de vida flexível e rígido para uma SA. O kernel envia uma mensagem SADB_EXPIRE quando o tempo de vida flexível (soft) tiver sido alcançado; a SA não será utilizada depois que seu tempo de vida rígido (hard) tiver sido alcançado. A extensão SADB_LIFETIME_CURRENT é retornada nas respostas SADB_DUMP, SADB_EXPIRE e SADB_GET para descrever os valores da associação atual.

19.5 Mantendo SAs dinamicamente

Para melhor segurança, uma troca periódica de chaves é requerida. Normalmente, isso é realizado por um protocolo como o IKE (RFC 2409 [Harkins e Carrel, 1998]).

> No momento em que este livro estava sendo escrito, o grupo de trabalho IETF do IPSec estava trabalhando em um substituto para o IKE.

Para aprender quando uma SA é requerida entre um novo par de hosts, um daemon se registra no kernel utilizando a mensagem SADB_REGISTER, especificando o tipo da SA que ele pode tratar no campo sadb_msg_satype a partir dos valores na Figura 19.3. Se um daemon puder tratar múltiplos tipos de SA, ele envia múltiplas mensagens SADB_REGISTER, cada uma registrando um único tipo. Na sua mensagem de resposta SADB_REGISTER, o kernel inclui uma extensão de algoritmos suportados, indicando quais mecanismos de cifragem e/ou de autenticação são suportados com quais comprimentos de chave. A extensão para os algoritmos suportados é descrita por uma estrutura sadb_supported, mostrada na Figura 19.13; ela simplesmente contém uma série descrições dos algoritmos de cifragem ou de autenticação nas estruturas sadb_alg que se seguem ao cabeçalho da extensão.

```
struct sadb_supported {
  u_int16_t sadb_supported_len;      /* comprimento da extensão +
                                        algoritmos / 8 */
  u_int16_t sadb_supported_exttype;  /* SADB_EXT_SUPPORTED_{AUTH,ENCRYPT} */
  u_int32_t sadb_supported_reserved; /* reservado para expansão futura */
};
                                     /* seguido pela lista de algoritmos */

struct sadb_alg {
  u_int8_t sadb_alg_id;             /* ID do algoritmo ID na Figura 19.8 */
  u_int8_t sadb_alg_ivlen;           /* comprimento IV length, ou zero */
  u_int16_t sadb_alg_minbits;        /* comprimento de chave mínimo */
  u_int16_t sadb_alg_maxbits;        /* comprimento de chave máximo */
  u_int16_t sadb_alg_reserved;       /* reservado para expansão futura */
};
```

Figura 19.13 Extensão para os algoritmos suportados.

Uma estrutura sadb_alg segue o cabeçalho sadb_supported da extensão para cada algoritmo suportado pelo sistema. A Figura 19.14 mostra uma possível resposta a uma mensagem registrada no tipo da SA SADB_SATYPE_ESP.

Nosso primeiro programa de exemplo, mostrado na Figura 19.15, simplesmente registra no kernel um dado mecanismo e imprime a resposta dos algoritmos suportados.

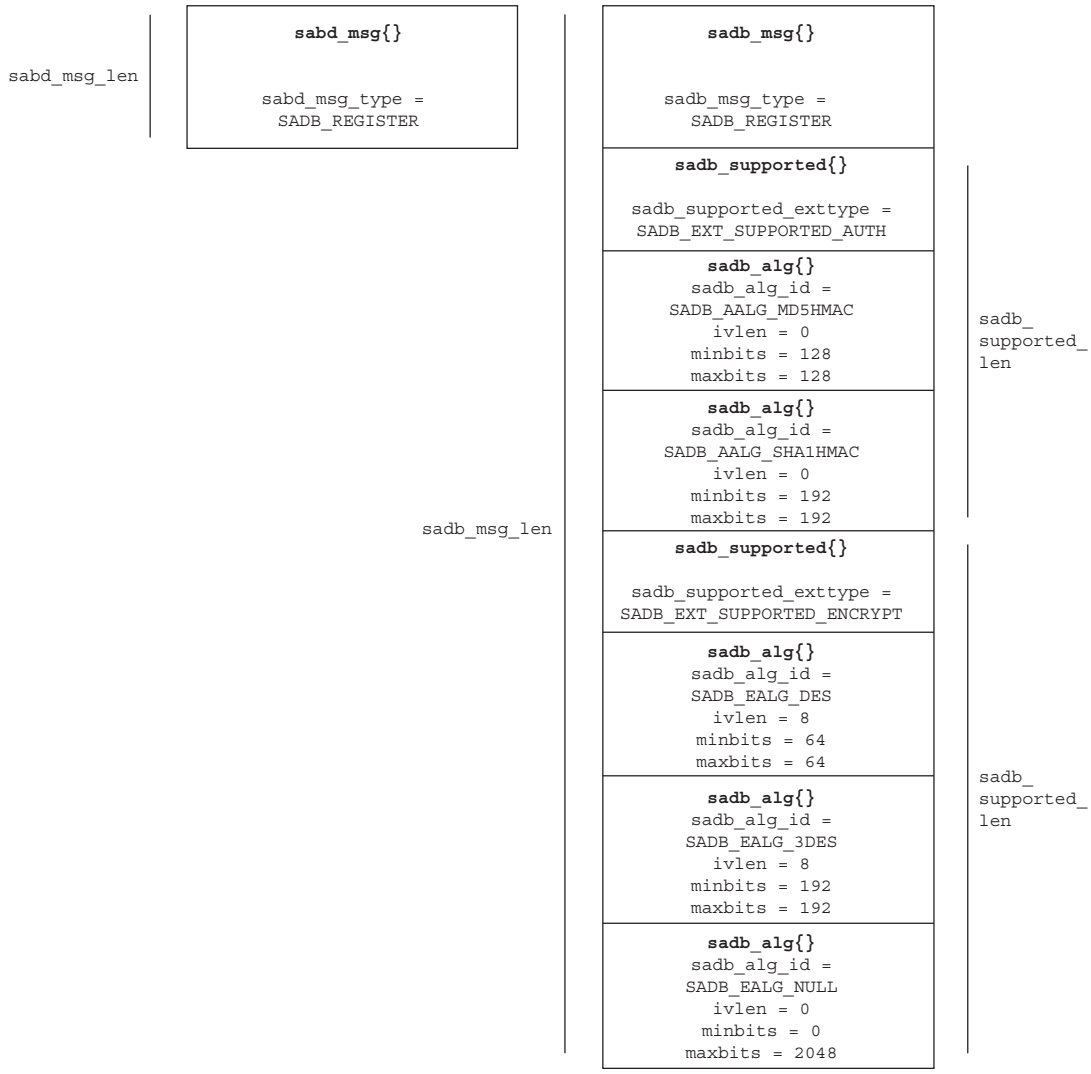


Figura 19.14 Os dados retornados do kernel ao comando `SADB_REGISTER`.

key/register.c

```
 1 void
 2 sadb_register(int type)
 3 {
 4     int     s;
 5     char    buf[4096];         /* XXX */
 6     struct sadb_msg msg;
 7     int     goteof;
 8     int     mypid;

 9     s = Socket(PF_KEY, SOCK_RAW, PF_KEY_V2);

10     mypid = getpid();

11     /* Constrói e grava a solicitação SADB_REGISTER */
```

Figura 19.15 Programa para registrar em um soquete de gerenciamento de chaves (*continua*).

```
12      bzero(&msg, sizeof(msg));
13      msg.sadb_msg_version = PF_KEY_V2;
14      msg.sadb_msg_type = SADB_REGISTER;
15      msg.sadb_msg_satype = type;
16      msg.sadb_msg_len = sizeof(msg) / 8;
17      msg.sadb_msg_pid = mypid;
18      printf("Sending message:\n");
19      print_sadb_msg(&msg, sizeof(msg));
20      Write(s, &msg, sizeof(msg));

21      printf("\nReply returned:\n");
22      /* Lê e imprime resposta SADB_REGISTER, descartando quaisquer outras */
23      for ( ; ; ) {
24          int    msglen;
25          struct sadb_msg *msgp;

26          msglen = Read(s, &buf, sizeof(buf));
27          msgp = (struct sadb_msg *) &buf;
28          if (msgp->sadb_msg_pid == mypid &&
29              msgp->sadb_msg_type == SADB_REGISTER) {
30              print_sadb_msg(msgp, msglen);
31              break;
32          }
33      }
34      close(s);
35  }
```

key/register.c

Figura 19.15 Programa para registrar em um soquete de gerenciamento de chaves (*continuação*).

Abrindo o soquete `PF_KEY`

1–9 Abrimos o soquete `PF_KEY`.

Armazenando o PID

10 Como as mensagens serão endereçadas a nós utilizando nosso PID, nós as armazenamos para comparação posterior.

Criando a mensagem `SADB_REGISTER`

11–17 Como ocorre com `SADB_DUMP`, a mensagem `SADB_REGISTER` não requer nenhuma extensão. Zeramos a mensagem e então preenchemos os campos individuais necessários.

Exibindo e gravando a mensagem no soquete

18–20 Exibimos a mensagem que estamos enviando utilizando nossa função `print_sadb_msg` e enviamos a mensagem ao soquete.

Esperando a resposta

23–33 Lemos as mensagens no soquete e esperamos a resposta da nossa mensagem registradora. A resposta é endereçada ao nosso PID e é uma mensagem `SADB_REGISTER`. Ela contém uma lista dos algoritmos suportados, que imprimimos com a nossa função `print_sadb_msg`.

Exemplo

Executamos o programa `register` em um sistema que suporta vários outros protocolos além daqueles descritos na RFC 2367.

```
macosx % register -t ah
Sending register message:
SADB Message Register, errno 0, satype IPsec AH, seq 0, pid 20746
```

```
Reply returned:
SADB Message Register, errno 0, satype IPsec AH, seq 0, pid 20746
 Supported authentication algorithms:
   HMAC-MD5 ivlen 0 bits 128-128
   HMAC-SHA-1 ivlen 0 bits 160-160
   Keyed MD5 ivlen 0 bits 128-128
   Keyed SHA-1 ivlen 0 bits 160-160
   Null ivlen 0 bits 0-2048
   SHA2-256 ivlen 0 bits 256-256
   SHA2-384 ivlen 0 bits 384-384
   SHA2-512 ivlen 0 bits 512-512
 Supported encryption algorithms:
   DES-CBC ivlen 8 bits 64-64
   3DES-CBC ivlen 8 bits 192-192
   Null ivlen 0 bits 0-2048
   Blowfish-CBC ivlen 8 bits 40-448
   CAST128-CBC ivlen 8 bits 40-128
   AES ivlen 16 bits 128-256
```

Quando o kernel precisa se comunicar com um peer e a política de segurança informa que uma SA é exigida, mas não há uma disponível, o kernel envia uma mensagem `SADB_ACQUIRE` aos soquetes de gerenciamento de chaves que registraram o tipo da SA exigida, contendo uma proposta de extensão que descreve os algoritmos e o comprimento da chave proposto pelo kernel. A proposta pode ser uma combinação daquilo suportado pelo sistema e uma política pré-configurada que limita o que é permitido nessa comunicação. A proposta é uma lista de algoritmos, comprimentos de chave e tempos de vida, na ordem de preferência. Quando um daemon de gerenciamento de chaves recebe uma mensagem `SADB_ACQUIRE`, ele realiza os procedimentos necessários para escolher uma chave que se ajuste a uma das combinações propostas pelo kernel e instala essa chave no kernel. Ele utiliza a mensagem `SADB_GETSPI` para solicitar que o kernel selecione um SPI a partir de um intervalo desejado. A resposta do kernel à mensagem `SADB_GETSPI` inclui a criação de uma SA no estado embrionário. O daemon então negocia os parâmetros de segurança com a extremidade remota utilizando o SPI fornecido pelo kernel e utiliza a mensagem `SADB_UPDATE` para completar a SA e faz com que ela entre no estado maduro. Geralmente, SAs criadas dinamicamente têm um tempo de vida flexível e um tempo de vida rígido (*soft and hard lifetime*) associados a elas. Quando um desses tempos de vida expira, o kernel envia uma mensagem `SADB_EXPIRE`, indicando qual tempo de vida (flexível ou rígido) expirou. Se o tempo de vida flexível expirou, a SA entrou no estado moribundo, durante o qual ainda pode ser utilizada, mas uma nova SA deverá ser obtida. Se o tempo de vida rígido expirou, a SA entrou no estado morto, no qual não mais é utilizada para propósitos de segurança e será removida do SADB.

19.6 Resumo

Soquetes de gerenciamento de chaves são utilizados para comunicar SAs ao kernel, aos daemons de chave de gerenciamento e a outros consumidores de segurança, como os daemons de roteamento. As SAs podem ser instaladas estática ou dinamicamente via um protocolo de negociação de chave. As chaves dinâmicas podem ter tempos de vida associados; quando o tempo de vida flexível é alcançado, o daemon de gerenciamento de chaves é informado. Se uma SA não for substituída antes de o tempo de vida rígido ser alcançado, a SA não mais poderá ser utilizada.

Dez mensagens são trocadas entre o processo e o kernel nos soquetes de gerenciamento de chaves. Cada tipo de mensagem associa extensões, algumas obrigatórias e outras opcionais. Cada mensagem enviada por um processo é ecoada a todos os outros soquetes de gerenciamento de chaves abertos, removendo quaisquer extensões que contenham dados sigilosos.

Exercícios

19.1 Escreva um programa que abra um soquete `PF_KEY` e faça o dump de todas as mensagens que ele recebe.

19.2 Pesquise o novo protocolo que o grupo de trabalho do IETF IPSec criou para substituir o IKE visitando sua página na Web: `http://www.ietf.org/html.charters/ipsec-charter.html`

Capítulo **20**

Broadcast

20.1 Visão geral

Neste capítulo descreveremos o *broadcasting* e, no capítulo seguinte, o *multicasting*. Todos os exemplos vistos até agora lidaram com o *unicasting*: um processo que se comunica exatamente com um único outro processo. De fato, o TCP funciona somente com endereços de unicast, embora o UDP e o IP brutos suportem outros paradigmas. A Figura 20.1 mostra uma comparação dos diferentes tipos de endereçamento.

O IPv6 adicionou o *anycasting* à arquitetura de endereçamento. Uma versão IPv4 do anycasting, que nunca foi amplamente utilizada, é descrita na RFC 1546 (Partridge, Mendez e Milliken, 1993). O anycasting do IPv6 é definido na RFC 3513 (Hinden e Deering, 2003). O anycasting permite endereçar um sistema (em geral, o mais "próximo" por alguma métrica) entre um conjunto de sistemas que normalmente fornecem serviços idênticos. Com uma configuração de roteamento apropriada, os hosts podem fornecer serviços de anycasting tanto no IPv4 como no IPv6 injetando o mesmo endereço no protocolo de roteamento em múltiplas localizações. Entretanto, o anycasting da RFC 3513 permite apenas a roteadores ter endereços de anycast; hosts não podem fornecer serviços de anycasting. No momento em que este livro estava sendo escrito, não havia nenhuma API definida para utilizar endereços de anycast. Há trabalhos em progresso para refinar a arquitetura anycast do IPv6; os hosts talvez sejam capazes de, no futuro, fornecer serviços de anycasting dinamicamente.

Tipo	IPv4	IPv6	TCP	UDP	nº de interfaces IP identificadas	nº de interfaces IP entregues a
Unicast	•	•	•	•	Uma	Uma
Anycast	*	•	Ainda não	•	Um conjunto	Uma no conjunto
Multicast	Opcional	•		•	Um conjunto	Todas no conjunto
Broadcast	•			•	Todas	Todas

Figura 20.1 Diferentes formas de endereçamento.

Os pontos importantes na Figura 20.1 são:

- O suporte de multicasting é opcional no IPv4, mas obrigatório no IPv6.
- O suporte de broadcasting não é fornecido no IPv6. Qualquer aplicação IPv4 que utilize broadcasting deve ser recodificada para que o IPv6 em vez disso utilize o multicasting.
- Broadcasting e multicasting requerem transporte de datagrama como o UDP ou o IP bruto; eles não podem funcionar com o TCP.

Uma utilização do broadcasting é localizar um servidor na sub-rede local quando se supõe que esteja nela, mas seu endereço IP unicast não é conhecido. Às vezes, isso é chamado de *descoberta de recurso*. Uma outra utilização é minimizar o tráfego de rede em uma rede local quando há múltiplos clientes que se comunicam com um único servidor. Há vários exemplos de aplicações de Internet que utilizam o broadcasting para esse propósito. Alguns desses exemplos também podem utilizar o multicasting.

- ARP – Apesar de esse protocolo residir abaixo do IPv4 e não de uma aplicação do usuário, ele transmite uma solicitação na sub-rede local que diz "Por favor, sistema com o endereço a.b.c.d, identifique-se e informe seu endereço de hardware". O ARP utiliza broadcast na camada de enlace, não na camada IP, mas é um exemplo de uma utilização de broadcasting.
- DHCP – O cliente supõe que um servidor ou um retransmissor (*relay*) esteja na sub-rede local e envia sua solicitação ao endereço de broadcast (freqüentemente 255.255.255.255, uma vez que o cliente ainda não conhece seu endereço IP, a máscara de sub-rede ou o endereço de broadcast limitado da sub-rede).
- Network Time Protocol (NTP) – Em um cenário comum, um cliente NTP é configurado para utilizar o endereço IP de um ou mais servidores; o cliente consulta seqüencialmente (*polls*) os servidores em alguma freqüência (a cada 64 segundos ou mais). O cliente atualiza seu relógio utilizando algoritmos sofisticados com base na data/hora retornada pelos servidores e no RTT para os mesmos. Mas, em uma LAN de broadcast, em vez de fazer com que cada um dos clientes consulte seqüencialmente um único servidor, o servidor pode transmitir a data/hora atual a cada 64 segundos a todos os clientes na sub-rede local, reduzindo a quantidade de tráfego na rede.
- Daemons de roteamento – O mais antigo daemon de roteamento, `routed`, que implementa a versão 1 do RIP, transmite sua tabela de roteamento em uma LAN. Isso permite a todos os outros roteadores conectados à LAN receberem esses anúncios de roteamento, sem que cada roteador precise ser configurado com os endereços IP de todos os seus roteadores vizinhos. Esse recurso também pode ser utilizado por hosts na LAN que recebem esses anúncios de roteamento e atualizam suas tabelas de roteamento dessa maneira. A versão 2 do RIP permite o uso de multicast ou broadcast.

Precisamos observar que o multicasting pode substituir os dois usos do broadcasting (descoberta de recurso e redução do tráfego de rede). Descreveremos os problemas com broadcasting mais adiante neste e no próximo capítulo.

20.2 Endereços de broadcast

Se indicarmos um endereço IPv4 como {*subnetid*, *hostid*}, em que *subnetid* representa os bits que são abrangidos pela máscara de rede (ou o prefixo CIDR) e *hostid* representa os bits que não são abrangidos, teremos dois tipos de endereços de broadcast. Identificamos um campo que contém todos os bits em um como –1.

1. Endereço de broadcast direcionado à sub-rede: {*subnetid*, –1} – Endereça todas as interfaces na sub-rede especificada. Por exemplo, se tivéssemos o 192.168.42/24 na

sub-rede, então 192.168.42.255 seria o endereço de broadcast direcionado à sub-rede para todas as interfaces na sub-rede 192.168.42/24.

Normalmente, os roteadores não encaminham essas transmissões de broadcast (páginas 226 e 227 do TCPv2). Na Figura 20.2, mostramos um roteador conectado às sub-redes 192.168.42/24 e 192.168.123/24.

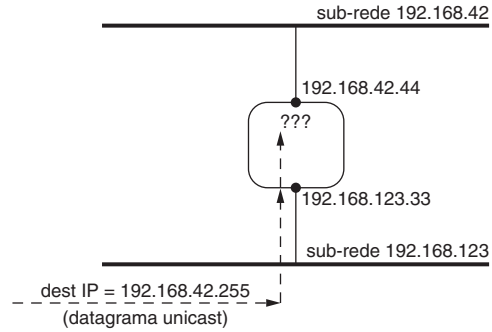


Figura 20.2 Um roteador encaminha uma transmissão de broadcast direcionada à sub-rede?

O roteador recebe um datagrama IP unicast na sub-rede 192.168.123/24 com um endereço de destino de 192.168.42.255 (o endereço de broadcast direcionado à sub-rede de uma outra interface). Normalmente, o roteador não encaminha o datagrama à sub-rede 192.168.42/24. Alguns sistemas têm uma opção de configuração que permite que broadcasts direcionados à sub-rede sejam encaminhados (Apêndice E do TCPv1).

> Encaminhar broadcasts direcionados à sub-rede permite uma classe de ataques de recusa de serviço chamada ataques de "amplificação"; por exemplo, enviar uma solicitação de eco de ICMP a um endereço de broadcast direcionado à sub-rede pode fazer com que múltiplas respostas sejam enviadas a uma única solicitação. Combinado com um endereço IP de origem forjado, isso resulta em um ataque de utilização da largura de banda contra o sistema da vítima, portanto, é aconselhável deixar essa opção de configuração desativada.
>
> Por essa razão, não é aconselhável projetar uma aplicação que conta com o encaminhamento de broadcasts direcionados à sub-rede, exceto em um ambiente controlado, em que você sabe que é seguro ativá-los.

2. Endereço de broadcast limitado: {–1, –1, –1} ou 255.255.255.255 – Datagramas destinados a esse endereço nunca devem ser encaminhados por um roteador.

   > 255.255.255.255 deve ser utilizado como o endereço de destino durante o processo de bootstrap por aplicações como BOOTP e DHCP que ainda não conhecem o endereço IP do nó.
   >
   > A pergunta é: O que um host faz quando uma aplicação envia um datagrama UDP a 255.255.255.255? A maioria dos hosts permite isso (assumindo que o processo configurou a opção de soquete `SO_BROADCAST`) e converte o endereço de destino no endereço de broadcast direcionado à sub-rede da interface de saída. Freqüentemente, é necessário acessar o enlace de dados diretamente (Capítulo 29) para enviar um pacote a 255.255.255.255.
   >
   > Uma outra pergunta é: O que um host multihomed faz quando a aplicação envia um datagrama UDP a 255.255.255.255? Alguns sistemas enviam uma única transmissão de broadcast na interface primária (a primeira interface que foi configurada) com o endereço IP de destino configurado como o endereço de broadcast direcionado à sub-rede dessa interface (página 736 do TCPv2). Outros sistemas enviam uma cópia do datagrama a partir de cada

interface capaz de broadcast. A Seção 3.3.6 da RFC 1122 (Braden, 1989) não tem uma posição firme quanto a essa questão. Para portabilidade, porém, se a aplicação precisar enviar um broadcast a partir de todas as interfaces capazes de broadcast, ela deverá obter a configuração da interface (Seção 17.6) e utilizar um `sendto` para cada interface capaz de broadcast com o destino configurado como esse endereço de broadcast da interface.

20.3 Unicast *versus* broadcast

Antes de examinar o broadcasting, vamos nos certificar de que entendemos os passos que acontecem quando um datagrama UDP é enviado a um endereço de unicast. A Figura 20.3 mostra três hosts em uma Ethernet.

O endereço da sub-rede da Ethernet é 192.168.42/24 com 24 bits na máscara de rede, deixando 8 bits para o ID do host. A aplicação no host à esquerda chama `sendto` em um soquete UDP, enviando um datagrama a 192.168.42.3, porta 7433. A camada UDP prefixa um cabeçalho UDP e passa o datagrama UDP para a camada IP. O IP prefixa um cabeçalho IPv4, determina a interface de saída e, no caso de uma Ethernet, o ARP é invocado para mapear o endereço IP de destino para o endereço Ethernet correspondente: `00:0a:95:79:bc:b4`. O pacote é então enviado como um quadro Ethernet com esse endereço de 48 bits como o endereço Ethernet de destino. O campo do tipo de quadro do quadro Ethernet será `0x0800`, especificando um pacote IPv4. O tipo de quadro para um pacote IPv6 é `0x86dd`.

A interface Ethernet no host na parte central vê o quadro passando e compara o endereço Ethernet de destino com seu próprio endereço Ethernet `00:04:ac:17:bf:38`). Como eles são diferentes, a interface ignora o quadro. Com um quadro unicast, não há nenhum overhead para esse host. A interface ignora o quadro.

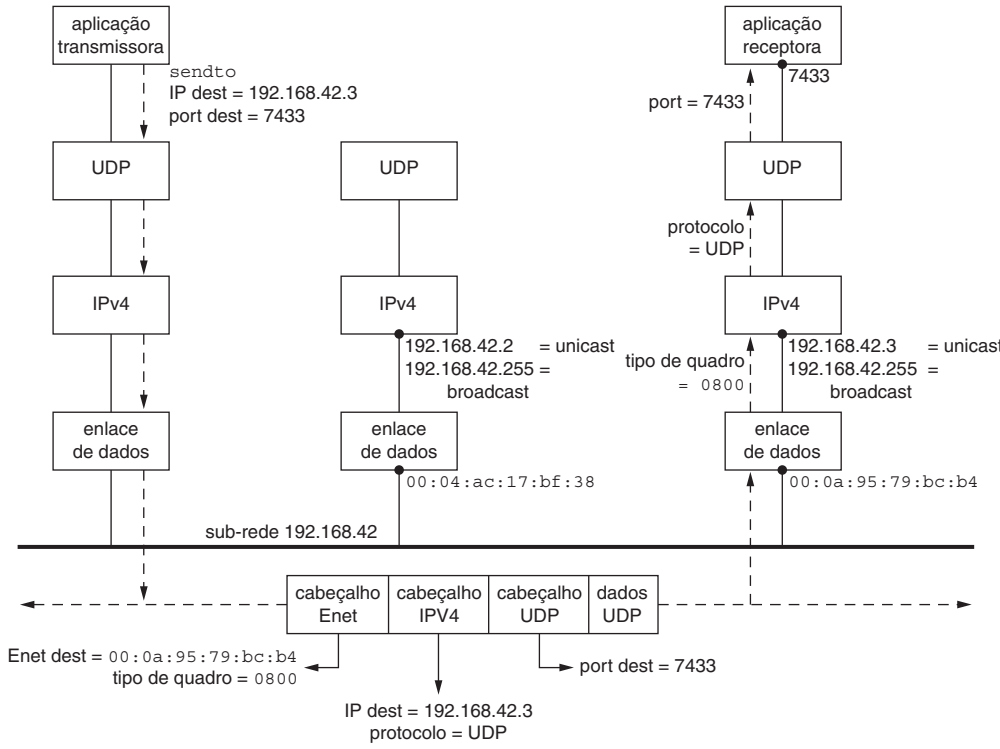


Figura 20.3 Exemplo unicast de um datagrama UDP.

A interface Ethernet no host à direita também vê a passagem do quadro e, quando compara o endereço Ethernet de destino com seu próprio endereço Ethernet, eles são iguais. Essa interface lê o quadro inteiro, provavelmente gera uma interrupção de hardware quando o quadro está completo e o driver de dispositivo lê o quadro a partir da memória da interface. Como o tipo de quadro é `0x0800`, o pacote é colocado na fila de entrada do IP.

Quando a camada IP processa o pacote, primeiro ela compara o endereço IP de destino (192.168.42.3) com todos os seus próprios endereços IP. (Lembre-se de que um host pode ser multihomed. Lembre-se também da nossa discussão sobre o modelo de sistema de extremidade forte e o modelo de sistema de extremidade fraca na Seção 8.8.) Como o endereço de destino é um dos endereços IP do próprio host, o pacote é aceito.

A camada IP examina então o campo de protocolo no cabeçalho IPv4, seu valor será 17 para o UDP. O datagrama IP é passado para o UDP. A camada UDP examina a porta de destino (e, possivelmente, também a porta de origem se o soquete UDP estiver conectado) e, no nosso exemplo, coloca o datagrama na fila de recebimento do soquete apropriada. O processo é acordado, se necessário, para ler o datagrama recém-recebido.

O ponto-chave nesse exemplo é que um datagrama IP de unicast é recebido somente pelo host especificado pelo endereço IP de destino. Nenhum outro host na sub-rede é afetado.

Agora, vamos considerar um exemplo semelhante, na mesma sub-rede, mas com a aplicação transmissora escrevendo um datagrama UDP no endereço de broadcast direcionado à sub-rede: 192.168.42.255. A Figura 20.4 mostra o arranjo.

Quando o host à esquerda envia o datagrama, ele percebe que o endereço IP de destino é o endereço de broadcast direcionado à sub-rede e o mapeia para o endereço Ethernet com 48 bits em um: `ff:ff:ff:ff:ff:ff`. Isso faz com que *todas* as interfaces Ethernet na sub-rede recebam o quadro. Os dois hosts à direita dessa figura que estão executando o IPv4 rece-

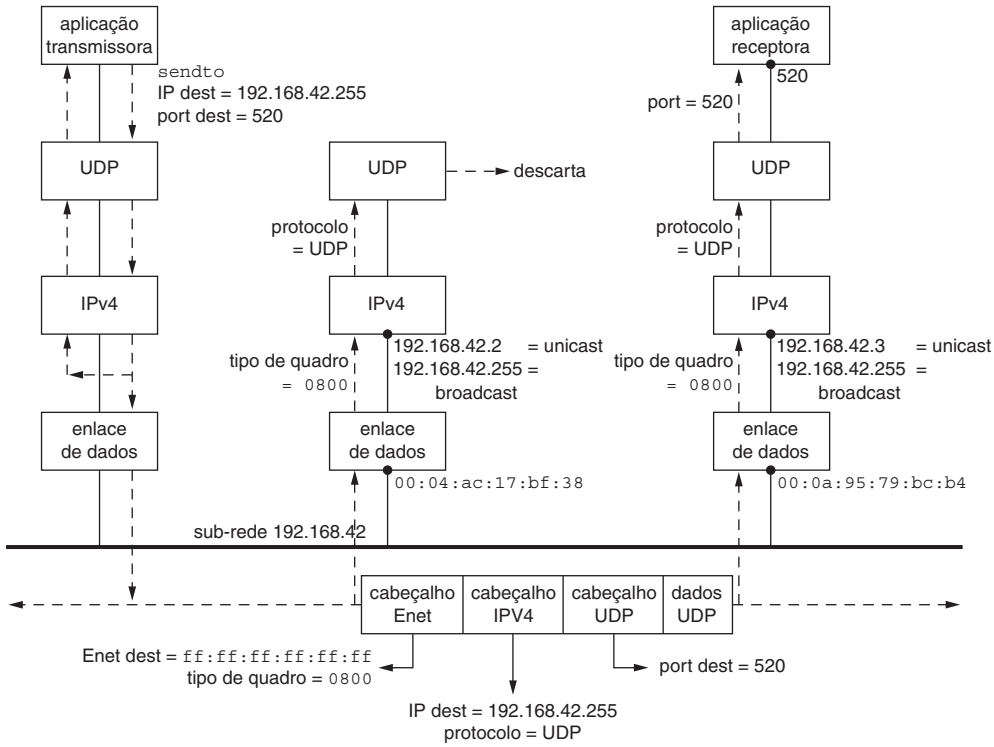


Figura 20.4 Exemplo de um datagrama UDP de broadcast.

berão o quadro. Como o tipo do quadro Ethernet é `0x0800`, os dois hosts passam o pacote para camada IP. Uma vez que o endereço IP de destino corresponde ao endereço de broadcast para cada um dos dois hosts e o campo do protocolo é 17 (UDP), os dois hosts passam o pacote para o UDP.

O host à direita passa o datagrama UDP para a aplicação que se vinculou à porta UDP 520. Nada especial precisa ser feito por uma aplicação para receber um datagrama UDP de broadcast: ela apenas cria um soquete UDP e o vincula ao número da porta da aplicação. (Assumimos que endereço IP vinculado é `INADDR_ANY`, o que é típico.)

No host na parte central, nenhuma aplicação se vinculou à porta UDP 520. O código UDP do host descarta então o datagrama recebido. Esse host *não* deve enviar um ICMP "port unreachable", pois fazer isso poderia gerar uma *inundação de pacotes boadcast (broadcast storm)*: uma condição em que uma grande quantidade de hosts na sub-rede gera uma resposta quase ao mesmo tempo, fazendo com que a rede torne-se inutilizável durante um período de tempo. Além disso, não está claro o que o host de envio faria com um erro de ICMP; e se alguns receptores reportarem erros e outros não?

Nesse exemplo, também mostramos o datagrama que é gerado pelo host à esquerda sendo entregue ao próprio host. Essa é uma propriedade das transmissões de broadcast: por definição, uma transmissão de broadcast passa por cada host na sub-rede, o que inclui o remetente (páginas 109 e 110 do TCPv2). Também assumimos que a aplicação transmissora vinculou-se à porta em que está enviando (520), de modo que receberá uma cópia de cada datagrama de broadcast que enviar. (Em geral, porém, não há nenhum requisito para que um processo vincule uma porta UDP à qual ele envia datagramas.)

> Nesse exemplo, mostramos um loopback lógico realizado pela camada IP ou pela camada do enlace de dados que faz uma cópia (páginas 109 e 110 do TCPv2) e a envia ao topo da pilha de protocolos. Uma rede poderia utilizar um loopback físico, mas isso pode levar a problemas no caso de falhas da rede (como uma Ethernet não-terminada).

Esse exemplo mostra o problema fundamental com o broadcasting: cada host IPv4 na sub-rede que não está participando na aplicação deve processar completamente o datagrama UDP de broadcast até o topo da pilha de protocolos, através de e incluindo a camada UDP, antes de descartar o datagrama. (Lembre-se da nossa discussão na Figura 8.21.) Além disso, cada host não-IP na sub-rede (digamos um host executando o IPX da Novell) também deve receber o quadro completo na camada do enlace de dados antes de descartá-lo (assumindo que o host não suporta o tipo de quadro, que seria `0x0800` para um datagrama IPv4). Para aplicações que geram datagramas IP a uma velocidade alta (áudio ou vídeo, por exemplo), esse processamento desnecessário pode afetar seriamente esses outros hosts na sub-rede. Veremos no próximo capítulo a maneira como o multicasting evita, até certo ponto, esse problema.

> Nossa escolha da porta 520 do UDP na Figura 20.4 é intencional. Essa é a porta utilizada pelo daemon `routed` para trocar pacotes RIP. Todos os roteadores em uma sub-rede que utilizam a versão 1 do RIP enviarão um datagrama UDP de broadcast a cada 30 segundos. Se houver 200 sistemas na sub-rede, incluindo dois roteadores que utilizam o RIP, 198 hosts terão de processar (e descartar) esses datagramas de broadcast a cada 30 segundos, supondo que nenhum deles está executando o `routed`. A versão 2 do RIP utiliza o multicast para evitar esse mesmo problema.

20.4 Função `dg_cli` utilizando broadcast

Novamente modificamos nossa função `dg_cli`, desta vez permitindo que ela transmita no servidor de data/hora UDP-padrão (Figura 2.18) e imprima todas as respostas. A única alteração que fazemos na função `main` (Figura 8.7) é alterar o número da porta de destino para 13.

```
servaddr.sin_port = htons(13);
```

Primeiro, compilamos essa função `main` modificada com a função `dg_cli` não-modificada da Figura 8.8 e a executamos no host `freebsd`.

```
freebsd % udpcli01 192.168.42.255
hi
sendto error: Permission denied
```

O argumento da linha de comando é o endereço de broadcast direcionado à sub-rede para a Ethernet secundária. Digitamos uma linha de entrada, o programa chama `sendto` e o erro `EACCES` é retornado. A razão pela qual recebemos o erro é que não temos permissão de enviar um datagrama a um endereço de broadcast de destino a menos que o kernel seja informado explicitamente de que estaremos transmitindo por broadcast. Fazemos isso configurando a opção de soquete `SO_BROADCAST` (Seção 7.5).

> Implementações derivadas do Berkeley implementam essa verificação de sanidade. O Solaris 2.5, por outro lado, aceita o datagrama destinado ao endereço de broadcast mesmo se não especificarmos a opção de soquete. A especificação POSIX requer que a opção do soquete SO_BROADCAST seja configurada para enviar um pacote de broadcast.
>
> O broadcasting era uma operação privilegiada no 4.2BSD e não havia a opção do soquete `SO_BROADCAST`. Essa opção foi adicionada ao 4.3BSD e qualquer processo tinha permissão de configurá-la.

Agora modificamos nossa função `dg_cli` como mostrado na Figura 20.5. Essa versão configura a opção do soquete `SO_BROADCAST` e imprime todas as respostas recebidas dentro de cinco segundos.

Alocação de espaço ao endereço do servidor, configuração da opção de soquete

11–13 `malloc` aloca espaço ao endereço do servidor a ser retornado por `recvfrom`. A opção do soquete `SO_BROADCAST` é configurada e um handler de sinal é instalado para `SIGALRM`.

Leitura da linha, envio ao soquete, leitura de todas as respostas

14–24 Os dois passos a seguir, `fgets` e `sendto`, são semelhantes a versões anteriores dessa função. Mas, como estamos enviando um datagrama de broadcast, poderemos receber múltiplas respostas. Chamamos `recvfrom` em um loop e imprimimos todas as respostas recebidas dentro de cinco segundos. Depois de cinco segundos, `SIGALRM` é gerado, nosso handler de sinal é chamado e `recvfrom` retorna o erro `EINTR`.

Impressão de cada resposta recebida

25–29 Para cada resposta recebida, chamamos `sock_ntop_host`, que, no caso do IPv4, retorna uma string que contém o endereço IP de pontos decimais do servidor. Isso é impresso juntamente com a resposta do servidor.

Se executarmos o programa especificando o endereço de broadcast direcionado à sub-rede 192.168.42.255, veremos o seguinte:

```
freebsd % udpcli01 192.168.42.255
hi
from 192.168.42.2: Sat Aug 2 16:42:45 2003
from 192.168.42.1: Sat Aug 2 14:42:45 2003
from 192.168.42.3: Sat Aug 2 14:42:45 2003
hello
from 192.168.42.3: Sat Aug 2 14:42:57 2003
from 192.168.42.2: Sat Aug 2 16:42:57 2003
from 192.168.42.1: Sat Aug 2 14:42:57 2003
```

A cada vez, precisamos digitar uma linha de entrada para gerar o datagrama UDP de saída. A cada vez, recebemos três respostas, e elas incluem o host de envio. Como dissemos an-

teriormente, o destino de um datagrama de broadcast são *todos* os hosts na rede conectada, incluindo o remetente. Cada resposta é unicast porque o endereço de origem da solicitação, utilizado por cada servidor como o endereço de destino da resposta, é um endereço de unicast.

Todos os sistemas informam a mesma data/hora porque todos executam o NTP.

bcast/dgclibcast1.c

```
 1 #include   "unp.h"
 2 static void recvfrom_alarm(int);
 3 void
 4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 5 {
 6     int    n;
 7     const int on = 1;
 8     char   sendline[MAXLINE], recvline[MAXLINE + 1];
 9     socklen_t len;
10     struct sockaddr *preply_addr;
11     preply_addr = Malloc(servlen);
12     Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
13     Signal(SIGALRM, recvfrom_alarm);
14     while (Fgets(sendline, MAXLINE, fp) != NULL) {
15         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
16         alarm(5);
17         for ( ; ; ) {
18             len = servlen;
19             n = recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
20             if (n < 0) {
21                 if (errno == EINTR)
22                     break;     /* esperou tempo demais pelas respostas */
23                 else
24                     err_sys("recvfrom error");
25             } else {
26                 recvline[n] = 0;   /* termina com nulo */
27                 printf("from %s: %s",
28                        Sock_ntop_host(preply_addr, len), recvline);
29             }
30         }
31     }
32     free(preply_addr);
33 }
34 static void
35 recvfrom_alarm(int signo)
36 {
37     return;                  /* apenas interrompe recvfrom() */
38 }
```

bcast/dgclibcast1.c

Figura 20.5 Função `dg_cli` que transmite por broadcast.

Fragmentação de IP e broadcasts

Kernels derivados do Berkeley não permitem que um datagrama de broadcast seja fragmentado. Se o tamanho de um datagrama IP que está sendo enviado a um endereço de broadcast exceder o MTU da interface de saída, `EMSGSIZE` é retornado (páginas 233 e 234 do TCPv2). Essa é uma decisão política que existia desde o 4.2BSD. Não há nada que impeça um kernel

de fragmentar um datagrama de broadcast, mas a sensação é de que o broadcasting impõe carga suficiente na rede da maneira como é, assim não há necessidade de multiplicar essa carga pelo número de fragmentos.

Podemos ver esse cenário com nosso programa na Figura 20.5. Redirecionamos a entrada-padrão de um arquivo que contém uma linha de 2.000 bytes, o que exigirá fragmentação em uma Ethernet.

```
freebsd % udpcli01 192.168.42.255 < 2000line
sendto error: Message too long
```

AIX, FreeBSD e MacOS implementam essa limitação. Linux, Solaris e HP-UX fragmentam datagramas enviados a um endereço de broadcast. Para portabilidade, porém, uma aplicação que precisa transmitir deve determinar o MTU da interface de saída utilizando o `SIOCGIFMTU ioctl` e então subtrair os comprimentos do cabeçalho de transporte e do IP para determinar o tamanho máximo do payload. Alternativamente, ela pode selecionar um MTU comum, como o 1500 da Ethernet, e utilizá-lo como uma constante.

20.5 Condições de corrida

Uma *condição de corrida (race condition)* normalmente ocorre quando múltiplos processos acessam os dados que são compartilhados entre eles, mas o resultado correto depende da ordem de execução dos processos. Como a ordem de execução dos processos nos sistemas Unix tradicionais depende de vários fatores que podem variar entre execuções, às vezes o resultado está correto, porém, outras vezes, está errado. Os tipos mais difíceis de condições de corrida a depurar são aqueles em que o resultado apenas ocasionalmente está errado. Discutiremos esses tipos de condições de corrida no Capítulo 26, quando discutirmos as variáveis de exclusão mútua e as variáveis de condição. As condições de corrida sempre são uma preocupação na programação de threads, visto que um grande volume de dados é compartilhado entre todos os threads (por exemplo, todas as variáveis globais).

Freqüentemente, há condições de corrida de um tipo diferente ao lidar com sinais. O problema ocorre porque um sinal normalmente pode ser entregue a qualquer momento enquanto o programa está em execução. O POSIX permite *bloquear* a entrega de um sinal, mas isso freqüentemente é pouco utilizado quando realizamos operações de E/S.

Vamos examinar esse problema utilizando um exemplo. Há uma condição de corrida na Figura 20.5; pare por alguns minutos e veja se você pode encontrá-la. (*Dica*: Onde podemos estar executando quando o sinal é entregue?) Você também pode forçar a condição a ocorrer desta maneira: altere o argumento de `alarm` de 5 para 1 e adicione `sleep(1)` antes de `printf`.

Quando fazemos essas alterações na função e então digitamos a primeira linha de entrada, a linha é enviada como um broadcast e configuramos o `alarm` para um segundo no futuro. Bloqueamos na chamada a `recvfrom` e a primeira resposta então chega ao nosso soquete, provavelmente dentro de alguns milissegundos. A resposta é retornada por `recvfrom`, mas então dormimos por um segundo. Respostas adicionais são recebidas e colocadas no nosso buffer de recebimento do soquete. Mas, enquanto estamos adormecidos, o timer `alarm` expira e o sinal `SIGALRM` é gerado: nosso handler de sinal é chamado, apenas retorna e interrompe o `sleep` em que estamos bloqueados. Em seguida, fazemos um loop e lemos as respostas enfileiradas com uma pausa de um segundo toda vez que imprimimos uma resposta. Quando tivermos lido todas as respostas, bloqueamos novamente na chamada a `recvfrom`, mas o timer não está em execução. Portanto, bloquearemos eternamente em `recvfrom`. O problema fundamental é que a nossa intenção era o handler de sinal interromper uma `recvfrom` bloqueada, mas o sinal pode ser entregue a qualquer momento e podemos estar executando em qualquer lugar no loop `for` infinito quando o sinal é entregue.

Agora, examinaremos quatro soluções diferentes para esse problema: uma solução incorreta e três diferentes soluções corretas.

Bloqueando e desbloqueando o sinal

Nossa primeira solução (incorreta) reduz a possibilidade de erro bloqueando a entrega do sinal enquanto estamos executando o restante do loop `for`. A Figura 20.6 mostra a nova versão.

bcast/dgclibcast3.c

```
 1 #include   "unp.h"
 2 static void recvfrom_alarm(int);
 3 void
 4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 5 {
 6     int    n;
 7     const int on = 1;
 8     char   sendline[MAXLINE], recvline[MAXLINE + 1];
 9     sigset_t sigset_alrm;
10     socklen_t len;
11     struct sockaddr *preply_addr;
12     preply_addr = Malloc(servlen);
13     Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
14     Sigemptyset(&sigset_alrm);
15     Sigaddset(&sigset_alrm, SIGALRM);
16     Signal(SIGALRM, recvfrom_alarm);
17     while (Fgets(sendline, MAXLINE, fp) != NULL) {
18         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
19         alarm(5);
20         for ( ; ; ) {
21             len = servlen;
22             Sigprocmask(SIG_UNBLOCK, &sigset_alrm, NULL);
23             n = recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
24             Sigprocmask(SIG_BLOCK, &sigset_alrm, NULL);
25             if (n < 0) {
26                 if (errno == EINTR)
27                     break;     /* esperou tempo demais pelas respostas */
28                 else
29                     err_sys("recvfrom error");
30             } else {
31                 recvline[n] = 0;   /* termina com nulo */
32                 printf("from %s: %s",
33                        Sock_ntop_host(preply_addr, len), recvline);
34             }
35         }
36     }
37     free(preply_addr);
38 }
39 static void
40 recvfrom_alarm(int signo)
41 {
42     return;                  /* apenas interrompe recvfrom() */
43 }
```

bcast/dgclibcast3.c

Figura 20.6 Bloqueio dos sinais enquanto executa dentro do loop `for` (solução incorreta).

Declaração de um conjunto de sinais e inicialização

14–15 Declaramos um conjunto de sinais e o inicializamos para o conjunto vazio (`sigemptyset`) e então ativamos o bit correspondente a `SIGALRM` (`sigaddset`).

Desbloqueio e bloqueio do sinal

21–24 Antes de chamar `recvfrom`, desbloqueamos o sinal (de modo que possa ser entregue enquanto estamos bloqueados) e então o bloqueamos logo que `recvfrom` retorna. Se o sinal for gerado (isto é, o timer expirar) enquanto está bloqueado, o kernel lembra-se desse fato, mas não pode entregar o sinal (isto é, chamar nosso handler de sinal) até que seja desbloqueado. Essa é a diferença fundamental entre a *geração* de um sinal e sua *entrega*. O Capítulo 10 da APUE fornece detalhes adicionais sobre todas essas facetas do tratamento de sinal do POSIX.

Se compilarmos e executarmos esse programa, aparentemente ele irá funcionar bem, mas a maioria dos programas com uma condição de corrida também funciona na maior parte do tempo! Ainda há um problema: o desbloqueamento do sinal, a chamada a `recvfrom` e o bloqueamento do sinal são chamadas de sistema independentes. Suponha que `recvfrom` retorne com o datagrama da resposta final e que o sinal seja entregue entre o `recvfrom` e o bloqueamento do sinal. A próxima chamada a `recvfrom` será bloqueada eternamente. Diminuímos a possibilidade, mas o problema ainda existe.

Uma variação dessa solução é fazer com que o handler de sinal configure um flag global quando o sinal é entregue.

```
static void
recvfrom_alarm(int signo)
{
   had_alarm = 1;
   return;
}
```

O flag é inicializado como 0 toda vez que `alarm` é chamado. Nossa função `dg_cli` verifica esse flag antes de chamar `recvfrom` e não o chama se o flag for não-zero.

```
for ( ; ; ) {
   len = servlen;
   Sigprocmask(SIG_UNBLOCK, &sigset_alrm, NULL);
   if (had_alarm == 1)
      break;
   n = recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
```

Se o sinal tiver sido gerado no momento em que foi bloqueado (depois do retorno anterior de `recvfrom`), e quando ele é desbloqueado nessa parte do código, será entregue antes que `sigprocmask` retorne, ligando nosso flag. Mas há ainda um pequeno intervalo de tempo entre o teste do flag e a chamada a `recvfrom` quando o sinal pode ser gerado e entregue e, se isso acontecer, a chamada a `recvfrom` será bloqueada para sempre (naturalmente, supondo que nenhuma resposta adicional é recebida).

Bloqueando e desbloqueando o sinal com `pselect`

Uma solução correta é utilizar `pselect` (Seção 6.9), como mostrado na Figura 20.7.

bcast/dgclibcast4.c

```
1 #include   "unp.h"
2 static void recvfrom_alarm(int);
3 void
```

Figura 20.7 Bloqueando e desbloqueando o sinal com `pselect` (*continua*).

```
 4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 5 {
 6     int    n;
 7     const int on = 1;
 8     char   sendline[MAXLINE], recvline[MAXLINE + 1];
 9     fd_set rset;
10     sigset_t sigset_alrm, sigset_empty;
11     socklen_t len;
12     struct sockaddr *preply_addr;

13     preply_addr = Malloc(servlen);

14     Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

15     FD_ZERO(&rset);

16     Sigemptyset(&sigset_empty);
17     Sigemptyset(&sigset_alrm);
18     Sigaddset(&sigset_alrm, SIGALRM);

19     Signal(SIGALRM, recvfrom_alarm);

20     while (Fgets(sendline, MAXLINE, fp) != NULL) {
21         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

22         Sigprocmask(SIG_BLOCK, &sigset_alrm, NULL);
23         alarm(5);
24         for ( ; ; ) {
25            FD_SET(sockfd, &rset);
26            n = pselect(sockfd + 1, &rset, NULL, NULL, NULL, &sigset_empty);
27            if (n < 0) {
28                if (errno == EINTR)
29                    break;
30                else
31                    err_sys("pselect error");
32            } else if (n != 1)
33                err_sys("pselect error: returned %d", n);

34            len = servlen;
35            n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
36            recvline[n] = 0;  /* termina com nulo */
37            printf("from %s: %s",
38                   Sock_ntop_host(preply_addr, len), recvline);
39        }
40    }
41    free(preply_addr);
42 }

43 static void
44 recvfrom_alarm(int signo)
45 {
46    return;                  /* apenas interrompe recvfrom() */
47 }
```

bcast/dgclibcast4.c

Figura 20.7 Bloqueando e desbloqueando o sinal com `pselect` (*continuação*).

22–33 Bloqueamos `SIGALRM` e chamamos `pselect`. O argumento final para `pselect` é um ponteiro para nossa variável `sigset_empty`, que é um conjunto de sinais sem nenhum sinal bloqueado, isto é, todos os sinais estão desbloqueados. `pselect` salvará a máscara de sinais atual (que tem `SIGALRM` bloqueado), testará os descritores especificados e bloqueará, se necessário, com a máscara de sinais configurada como o conjunto vazio. Antes de retornar, a máscara de sinais do processo é redefinida para seu valor quando `pselect` é chamada. O ponto-chave para `pselect` é que a configuração da máscara de sinais, o teste dos descritores e a reinicialização da máscara de sinais são operações atômicas em relação ao processo chamador.

34–38 Se nosso soquete for legível, chamamos `recvfrom`, sabendo que ela não bloqueará.

Como mencionamos na Seção 6.9, `pselect` é nova na especificação POSIX; entre todos os sistemas na Figura 1.16, somente o FreeBSD e o Linux suportam essa função. Contudo, a Figura 20.8 mostra uma implementação, incorreta mas simples. Nossa razão para mostrar essa implementação incorreta é demonstrar os três procedimentos envolvidos: configurar a máscara do sinal para o valor especificado pelo chamador, salvar a máscara atual, testar os descritores e redefinir a máscara de sinais.

lib/pselect.c

```
 9 #include   "unp.h"

10 int
11 pselect(int nfds, fd_set *rset, fd_set *wset, fd_set *xset,
12         const struct timespec *ts, const sigset_t *sigmask)
13 {
14     int    n;
15     struct timeval tv;
16     sigset_t savemask;

17     if (ts != NULL) {
18         tv.tv_sec = ts->tv_sec;
19         tv.tv_usec = ts->tv_nsec / 1000;        /* nanosec -> microsec */
20     }

21     sigprocmask(SIG_SETMASK, sigmask, &savemask);  /* máscara do chamador */
22     n = select(nfds, rset, wset, xset, (ts == NULL) ? NULL : &tv);
23     sigprocmask(SIG_SETMASK, &savemask, NULL); /* restaura máscara */

24     return (n);
25 }
```

lib/pselect.c

Figura 20.8 Implementação incorreta, mas simples, de `pselect`.

Utilizando `sigsetjmp` e `siglongjmp`

Uma outra maneira correta de resolver nosso problema é não utilizar a capacidade de um handler de sinal de interromper uma chamada de sistema bloqueada, mas, em vez disso, chamar `siglongjmp` a partir do handler de sinal. Essa técnica é chamada de *goto não-local* porque podemos utilizá-la para pular de uma função para outra. A Figura 20.9 demonstra a técnica.

bcast/dgclibcast5.c

```
 1 #include    "unp.h"
 2 #include    <setjmp.h>

 3 static void recvfrom_alarm(int);
 4 static sigjmp_buf jmpbuf;

 5 void
 6 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 7 {
 8     int    n;
 9     const int on = 1;
10     char   sendline[MAXLINE], recvline[MAXLINE + 1];
11     socklen_t len;
12     struct sockaddr *preply_addr;

13     preply_addr = Malloc(servlen);
```

Figura 20.9 Utilização de `sigsetjmp` e `siglongjmp` a partir do handler de sinal (*continua*).

```
14      Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
15      Signal(SIGALRM, recvfrom_alarm);
16      while (Fgets(sendline, MAXLINE, fp) != NULL) {
17          Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
18          alarm(5);
19          for ( ; ; ) {
20              if (sigsetjmp(jmpbuf, 1) != 0)
21                  break;
22              len = servlen;
23              n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
24              recvline[n] = 0;        /* termina com nulo */
25              printf("from %s: %s",
26                     Sock_ntop_host(preply_addr,len), recvline);
27          }
28      }
29      free(preply_addr);
30  }
31  static void
32  recvfrom_alarm(int signo)
33  {
34      siglongjmp(jmpbuf, 1);
35  }
```

bcast/dgclibcast5.c

Figura 20.9 Utilização de `sigsetjmp` e `siglongjmp` a partir do handler de sinal (*continuação*).

Alocação do buffer de jump

4 Alocamos um buffer de jump que será utilizado pela nossa função e por seu handler de sinal.

Chamada a `sigsetjmp`

20–23 Quando chamamos `sigsetjmp` diretamente da nossa função `dg_cli`, ela estabelece o buffer de jump e retorna 0. Prosseguimos e chamamos `recvfrom`.

Tratando `SIGALRM` e chamando `siglongjmp`

31–35 Quando o sinal é entregue, chamamos `siglongjmp`. Isso faz com que o `sigsetjmp` na função `dg_cli` retorne com um valor de retorno igual ao segundo argumento (1), que deve ser um valor não-zero. Isso fará com que o loop `for` em `dg_cli` termine.

Utilizar `sigsetjmp` e `siglongjmp` dessa maneira garante que não estaremos bloqueados eternamente em `recvfrom` devido a um sinal entregue em um momento inoportuno. Entretanto, isso introduz um outro problema potencial: se o sinal for entregue enquanto `printf` está no meio da sua saída, iremos, efetivamente, pular fora do meio da `printf` e voltar à nossa `sigsetjmp`. Isso pode, por exemplo, deixar `printf` com estruturas de dados privados inconsistentes. Para evitar isso, devemos combinar o sinal bloqueador e desbloqueador da Figura 20.6 com a técnica de goto não-local. Isso torna essa solução difícil de manejar, uma vez que o sinal bloqueador pode ocorrer durante qualquer função que talvez se comporte de maneira inapropriada como resultado de ser interrompida no meio da execução.

Usando comunicação entre processos do handler de sinal para a função

Ainda há uma outra maneira correta de resolver nosso problema. Em vez de fazer com que o handler de sinal apenas retorne e, esperançosamente, interrompa uma `recvfrom` bloqueada, fazemos com que ele utilize comunicação entre processos (IPC) para notificar nossa função

`dg_cli` de que o timer expirou. Isso é quase semelhante à proposta que fizemos anteriormente para o handler de sinal de configurar a variável global `had_alarm` quando o timer tiver expirado, pois essa variável global foi utilizada como uma forma de IPC (memória compartilhada entre nossa função e o handler de sinal). O problema dessa solução, porém, foi que a nossa função tinha de testar essa variável e isso resultava em problemas de sincronização se o sinal fosse entregue quase ao mesmo tempo.

O que utilizamos na Figura 20.10 é um pipe dentro do nosso processo, com o handler de sinal gravando um byte no pipe quando o timer expira e nossa função `dg_cli` lendo esse byte para saber quando terminar seu loop `for`. O que torna essa solução tão boa é que testar a legibilidade do pipe ocorre com `select`. Testamos se o soquete ou o pipe está legível.

Criação de um pipe

15 Criamos um pipe Unix normal e dois descritores retornam. `pipefd[0]` é a extremidade de leitura e `pipefd[1]` é a extremidade de gravação.

> Também poderíamos utilizar `socketpair` e obter um pipe full-duplex. Em alguns sistemas, notavelmente o SVR4, um pipe Unix normal sempre é full-duplex, e podemos ler e escrever de uma extremidade qualquer.

Chamada a `select` tanto no soquete como na extremidade de leitura do pipe

23–30 Chamamos `select` para `sockfd`, o soquete, e para `pipefd[0]`, a extremidade de leitura do pipe.

47–52 Quando `SIGALRM` é entregue, nosso handler de sinal grava um byte no pipe, tornando a extremidade de leitura legível. Nosso handler de sinal também retorna, possivelmente interrompendo `select`. Portanto, se `select` retornar `EINTR`, ignoramos o erro, sabendo que a extremidade de leitura do pipe também estará legível e que isso terminará o loop `for`.

Leitura do pipe com `read`

39–42 Quando a extremidade de leitura do pipe está legível, chamamos `read` para ler e ignorar o byte nulo que o handler de sinal gravou. Mas isso informa que o timer expirou, assim utilizamos `break` para interromper o loop `for` infinito.

bcast/dgclibcast6.c

```
 1 #include "unp.h"

 2 static void recvfrom_alarm(int);
 3 static int pipefd[2];

 4 void
 5 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 6 {
 7     int    n, maxfdp1;
 8     const int on = 1;
 9     char   sendline[MAXLINE], recvline[MAXLINE + 1];
10     fd_set rset;
11     socklen_t len;
12     struct sockaddr *preply_addr;

13     preply_addr = Malloc(servlen);

14     Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

15     Pipe(pipefd);
16     maxfdp1 = max(sockfd, pipefd[0]) + 1;

17     FD_ZERO(&rset);

18     Signal(SIGALRM, recvfrom_alarm);
```

Figura 20.10 Utilizando um pipe como IPC do handler de sinal para nossa função (*continua*).

```
19      while (Fgets(sendline, MAXLINE, fp) != NULL) {
20          Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
21          alarm(5);
22          for ( ; ; ) {
23              FD_SET(sockfd, &rset);
24              FD_SET(pipefd[0], &rset);
25              if ( (n = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
26                  if (errno == EINTR)
27                      continue;
28                  else
29                      err_sys("select error");
30              }
31              if (FD_ISSET(sockfd, &rset)) {
32                  len = servlen;
33                  n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr,
34                               &len);
35                  recvline[n] = 0;   /* termina com nulo */
36                  printf("from %s: %s",
37                         Sock_ntop_host(preply_addr,len), recvline);
38              }
39              if (FD_ISSET(pipefd[0], &rset)) {
40                  Read(pipefd[0], &n, 1); /* timer expirado */
41                  break;
42              }
43          }
44      }
45      free(preply_addr);
46  }
47  static void
48  recvfrom_alarm(int signo)
49  {
50      Write(pipefd[1], "", 1);    /* grava um byte nulo no pipe */
51      return;
52  }
```

bcast/dgclibcast6.c

Figura 20.10 Utilizando um pipe como IPC do handler de sinal para nossa função (*continuação*).

20.6 Resumo

O broadcasting envia um datagrama que todos os hosts na sub-rede conectada recebem. A desvantagem quanto ao broadcasting é que cada host na sub-rede deve processar o datagrama, até a camada UDP no caso de um datagrama de UDP, mesmo se o host não estiver participando na aplicação. Para aplicações com altas taxas de dados, como áudio ou vídeo, isso pode impor uma carga excessiva de processamento sobre esses hosts. Veremos no próximo capítulo que o multicasting resolve esse problema porque somente os hosts que estão interessados na aplicação recebem o datagrama.

Utilizando uma versão do nosso cliente de eco UDP que envia um broadcast ao servidor de data/hora e então imprime todas as respostas recebidas dentro de cinco segundos, examinamos condições de corrida com o sinal `SIGALRM`. Como o uso da função `alarm` e do sinal `SIGALRM` é uma maneira comum de determinar um tempo-limite em uma operação de leitura, esse erro sutil é comum nas aplicações de rede. Mostramos uma maneira incorreta de resolver o problema e três maneiras corretas:

- Utilizando `pselect`
- Utilizando `sigsetjmp` e `siglongjmp`
- Utilizando IPC (em geral um pipe) do handler de sinal para o loop principal

Exercícios

20.1 Execute o cliente UDP utilizando a função `dg_cli` que transmite por broadcast (Figura 20.5). Quantas respostas você recebe? As respostas sempre estão na mesma ordem? Os hosts na sua rede têm relógios sincronizados?

20.2 Coloque algumas `printf` na Figura 20.10 depois que `select` retorna para ver se ela retorna um erro ou legibilidade para um dos dois descritores. Quando `alarm` expira, o seu sistema retorna `EINTR` ou legibilidade no pipe?

20.3 Execute uma ferramenta como `tcpdump`, se disponível, e procure pacotes de broadcast na sua rede local; `tcpdump ether broadcast` é o comando `tcpdump`. A quais conjuntos de protocolos as transmissões de broadcast pertencem?

Capítulo 21

Multicast

21.1 Visão geral

Como mostrado na Figura 20.1, um endereço unicast identifica uma *única* interface IP, um endereço de broadcast identifica *todas* as interfaces IP na sub-rede e um endereço multicast identifica um *conjunto* de interfaces IP. O unicast e o broadcast são os extremos do espectro de endereçamento (um ou todos) e o propósito do multicast é permitir o endereçamento de algo entre os dois. Um datagrama multicast deve ser recebido somente pelas interfaces interessadas por ele, isto é, pelas interfaces nos hosts que executam as aplicações que desejam participar do grupo multicast. Além disso, normalmente o broadcast está limitado a uma rede local, enquanto o multicast pode ser utilizado em uma rede local ou em uma WAN. De fato, as aplicações utilizam o multicast em um subconjunto da Internet diariamente.

As adições aos soquetes API para suportar o multicast são simples; elas abrangem nove opções de soquete: três que afetam o envio de datagramas UDP a um endereço multicast e seis que afetam a recepção pelo host dos datagramas multicast.

21.2 Endereços de multicast

Ao descrever endereços multicast, precisamos distinguir entre o IPv4 e o IPv6.

Endereços IPv4 da classe D

Os endereços da classe D, no intervalo 224.0.0.0 a 239.255.255.255, são os endereços multicast no IPv4 (Figura A.3). Os 28 bits de ordem inferior de um endereço da classe D formam o *ID de grupo* do multicast e o endereço de 32 bits é chamado de *endereço de grupo*.

A Figura 21.1 mostra como os endereços multicast IP são mapeados para endereços multicast Ethernet. Esse mapeamento para endereços multicast IPv4 está descrito na RFC 1112 (Deering, 1989) para Ethernets, na RFC 1390 (Katz, 1993) para redes FDDI e na RFC 1469 (Pusateri, 1993) para redes token-ring. Também mostramos o mapeamento para endereços multicast IPv6 para permitir uma comparação fácil dos endereços Ethernet resultantes.

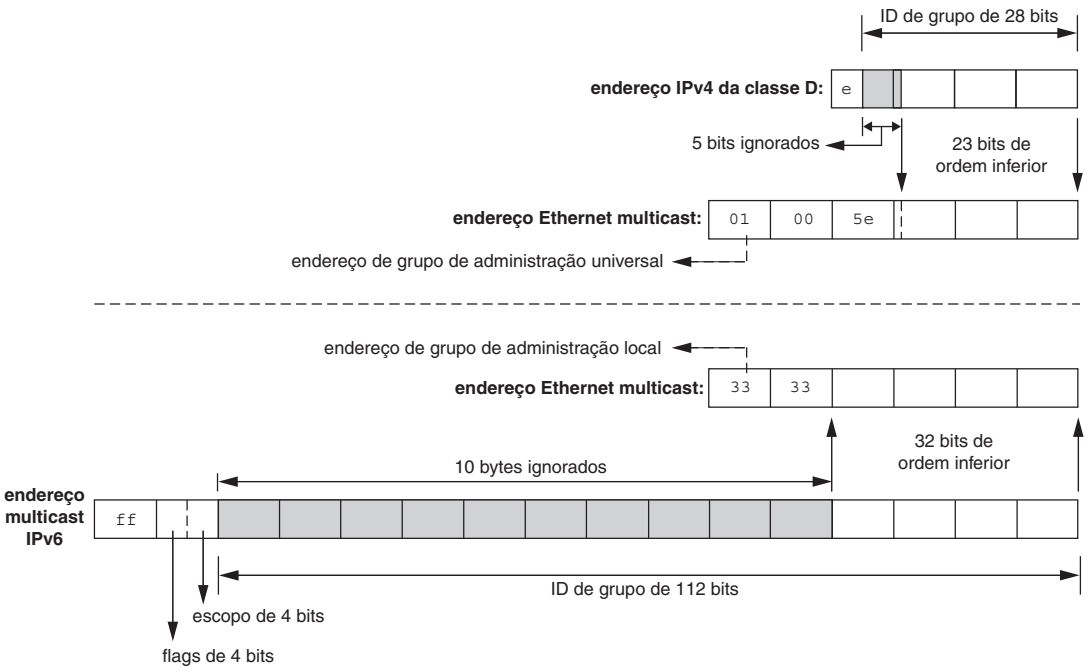


Figura 21.1 Mapeamento do endereço multicast IPv4 e IPv6 para endereços Ethernet.

Considerando somente o mapeamento IPv4, os 24 bits de ordem superior do endereço Ethernet sempre são `01:00:5e`. O bit seguinte sempre é 0 e os 23 bits de ordem inferior são copiados dos 23 bits de ordem inferior do endereço do grupo multicast. Os 5 bits de ordem superior do endereço de grupo são ignorados no mapeamento. Isso significa que 32 endereços multicast mapeiam para um único endereço Ethernet: o mapeamento não é de um para um.

Os 2 bits de ordem inferior do primeiro byte do endereço Ethernet identificam o endereço como um endereço de grupo *universalmente administrado*, o que significa que os 24 bits de ordem superior foram atribuídos pelo IEEE e os endereços de grupo são reconhecidos e tratados especialmente pelas interfaces receptoras.

Há alguns endereços multicast IPv4 especiais:

- 224.0.0.1 é o grupo *all-hosts*. Todos os nós de multicast capazes (hosts, roteadores, impressoras, etc.) em uma sub-rede devem se associar a esse grupo em todas as interfaces capazes de multicast. (Discutiremos o que significa associar-se a um grupo multicast mais adiante.)
- 224.0.0.2 é o grupo de *todos os roteadores*. Todos os roteadores capazes de multicast em uma sub-rede devem se associar a esse grupo em todas as interfaces capazes de multicast.

O intervalo entre 224.0.0.0 e 224.0.0.255 (que também podemos escrever como 224.0.0.0/24) é chamado *enlace local (link local)*. Esses endereços são reservados à descoberta da topologia de baixo nível ou a protocolos de manutenção, e datagramas destinados a qualquer um desses endereços nunca são encaminhados por um roteador multicast. Discutiremos o escopo dos vários endereços multicast IPv4 em detalhes depois de examinarmos os endereços multicast IPv6.

Endereços multicast IPv6

O byte de ordem superior de um endereço multicast IPv6 tem o valor `ff`. A Figura 21.1 mostra o mapeamento de um endereço multicast IPv6 de 16 bytes para um endereço Ethernet de 6 bytes. Os 32 bits de ordem inferior do endereço de grupo são copiados para os 32 bits de ordem inferior do endereço Ethernet. Os 2 bytes de ordem superior do endereço Ethernet são `33:33`. Esse mapeamento para Ethernet é especificado na RFC 2464 (Crawford, 1998a), o mesmo mapeamento para FDDI está na RFC 2467 (Crawford, 1998b) e o mapeamento para token-ring está na RFC 2470 (Crawford, Narten e Thomas, 1998).

Os dois bits de ordem inferior do primeiro byte do endereço Ethernet especificam o endereço como um endereço de grupo *localmente administrado*, o que significa que não há nenhuma garantia de que o endereço seja único para o IPv6. Pode haver outros conjuntos de protocolos além do IPv6 que compartilham a rede e utilizam os mesmos dois bytes de ordem superior do endereço Ethernet. Como mencionamos anteriormente, os endereços de grupo são reconhecidos e tratados especialmente pelas interfaces receptoras.

Dois formatos são definidos para endereços multicast IPv6, como mostrado na Figura 21.2. Se o flag *P* for 0, o flag *T* irá diferenciar entre um grupo multicast *bem-conhecido* (um valor de 0) e um grupo multicast *transitório* (um valor de 1). Um valor *P* de 1 designa um endereço multicast atribuído com base em um prefixo de unicast (definido na RFC 3306 [Haberman e Thaler, 2002]). Se o flag *P* for 1, o flag *T* também deverá ser 1 (isto é, os endereços multicast baseados em unicast sempre são transitórios), e os campos *prefix* e *plen* são configurados como o comprimento do prefixo e o valor do prefixo de unicast, respectivamente. Os dois bits superiores desse campo são reservados. Endereços multicast IPv6 também têm um campo *scope (escopo)* de 4 bits que discutiremos mais adiante. A RFC 3307 (Haberman, 2002) descreve o mecanismo de alocação para os 32 bits de ordem inferior de um endereço de grupo IPv6 (o *ID de grupo*), independentemente da configuração do flag *P*.

Há alguns endereços multicast IPv6 especiais:

- `ff01::1` e `ff02::1` são grupos *all-nodes (todos os nós)* no escopo de uma interface e enlace local. Todos os nós (hosts, roteadores, impressoras, etc.) em uma sub-rede devem se associar a esses grupos em todas as interfaces capazes de multicast. Isso é semelhante ao endereço multicast IPv4 224.0.0.1. Entretanto, como o multicast é uma parte integral do IPv6, diferentemente do IPv4, isso não é opcional.

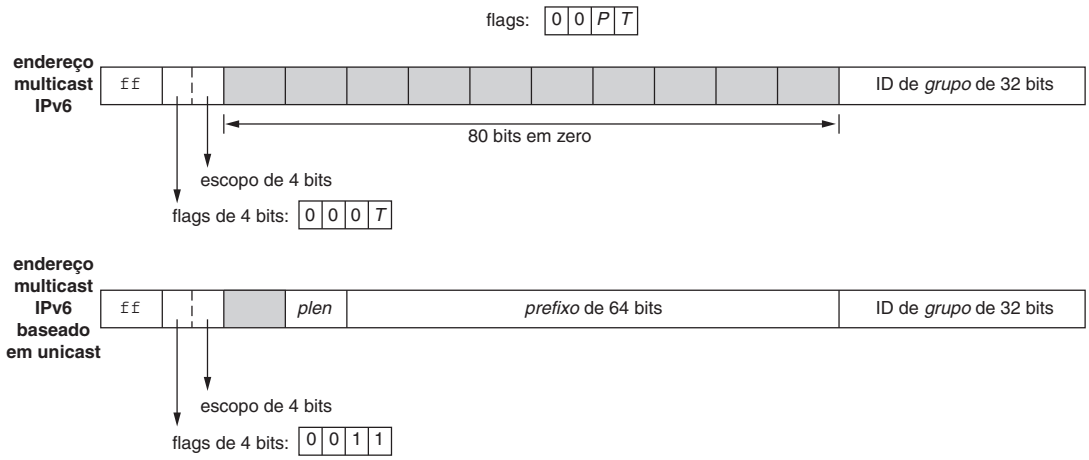


Figura 21.2 Formato dos endereços multicast IPv6.

Embora o grupo IPv4 seja chamado *all-hosts* (todos os hosts) e o grupo IPv6 seja chamado *all-nodes* (todos os nós), ambos servem para o mesmo propósito. O grupo foi renomeado no IPv6 para tornar claro que se destina a endereçar roteadores, impressoras e quaisquer outros dispositivos IP na sub-rede, bem como hosts.

- `ff01::2`, `ff02::2` e `ff05::2` são os grupos *all-routers* (todos os roteadores) nos escopos de uma interface local, enlace local e site local. Todos os roteadores em uma sub-rede devem se associar a esses grupos em todas as interfaces capazes de multicast. Isso é semelhante ao endereço multicast IPv4 224.0.0.2.

O escopo dos endereços multicast

Os endereços multicast IPv6 têm um campo de *escopo* explícito de 4 bits que especifica a distância que o pacote multicast irá percorrer. Os pacotes IPv6 também têm um campo de *limite de hop* que limita o número de vezes que o pacote é encaminhado por um roteador. Os valores a seguir foram atribuídos ao campo de escopo:

1: interface local
2: enlace local
4: administração local
5: site local
8: organização local
14: global

Os valores restantes não estão atribuídos ou estão reservados. Um datagrama da interface local não deve ser gerado por uma interface e um datagrama do enlace local nunca deve ser encaminhado por um roteador. O que define uma região de administração, um site ou uma organização é responsabilidade dos administradores dos roteadores multicast nesse site ou organização. Endereços multicast IPv6 que diferem somente em escopo representam grupos diferentes.

O IPv4 não tem um campo de escopo separado para pacotes multicast. Historicamente, o campo TTL do IPv4 no cabeçalho IP tem funcionado também como um campo de escopo multicast: um TTL de 0 significa interface local; 1 significa um enlace local; até 32 significa site local; até 64 significa região local; até 128 continente local (o que significa evitar enlaces de velocidade baixa ou altamente congestionados, intercontinentais ou não); e até 255 são irrestritos em escopo (globais). Esse uso duplo do campo TTL tem levado a dificuldades, como detalhado na RFC 2365 (Meyer, 1998).

Embora a utilização do campo TTL do IPv4 para determinar o escopo seja uma prática aceita e recomendável, o escopo administrativo é preferível quando possível. Isso define o intervalo 239.0.0.0 a 239.255.255.255 como o *espaço de multicast IPv4 administrativamente com escopo* (RFC 2365 [Meyer, 1998]). Esse é o ponto máximo do espaço do endereço multicast. Os endereços nesse intervalo são atribuídos localmente por uma organização, mas não há garantias de serem únicos por todos os limites organizacionais. Uma organização deve configurar seus roteadores de borda (roteadores multicast na periferia da organização) para não encaminhar pacotes multicast destinados a qualquer um desses endereços.

Os endereços multicast IPv4 de escopo administrativo são divididos em escopo local e escopo da organização local, sendo o primeiro semelhante (mas não semanticamente equivalente) ao escopo de site local do IPv6. Resumimos as diferentes regras de scoping na Figura 21.3.

Sessões de multicast

Especialmente no caso de multimídia em streaming, a combinação de um endereço multicast IP (IPv4 ou IPv6) e uma porta da camada de transporte (em geral UDP) é denominada *sessão*. Por exemplo, uma teleconferência de áudio/vídeo pode abranger duas sessões; uma pa-

Escopo	Escopo do IPv6	IPv4	
		Escopo de TTL	Escopo administrativo
Interface local	1	0	
Enlace local	2	1	224.0.0.0 a 224.0.0.255
Site local	5	< 32	239.255.0.0 a 239.255.255.255
Organização local	8		239.192.0.0 a 239.195.255.255
Global	14	≤ 255	224.0.1.0 a 238.255.255.255

Figura 21.3 O escopo dos endereços multicast IPv4 e IPv6.

ra áudio e outra para vídeo. Essas sessões quase sempre utilizam diferentes portas e, às vezes, também diferentes grupos para flexibilidade de escolha durante o recebimento. Por exemplo, um cliente pode escolher receber somente a sessão de áudio e outro pode escolher a sessão de áudio e vídeo. Se as sessões utilizassem o mesmo endereço de grupo, essa escolha não seria possível.

21.3 Multicast *versus* broadcast em uma rede local

Agora retornamos aos exemplos nas Figuras 20.3 e 20.4 para mostrar o que acontece no caso de multicast. Utilizamos o IPv4 para o exemplo mostrado na Figura 21.4, mas os passos são semelhantes para o IPv6.

A aplicação receptora no host mais à direita inicia e cria um soquete UDP, vincula a porta 123 ao soquete e então se associa ao grupo multicast 224.0.1.1. Veremos a seguir que essa operação de "associação" ("*join*") ocorre chamando `setsockopt`. Quando isso acontece, a

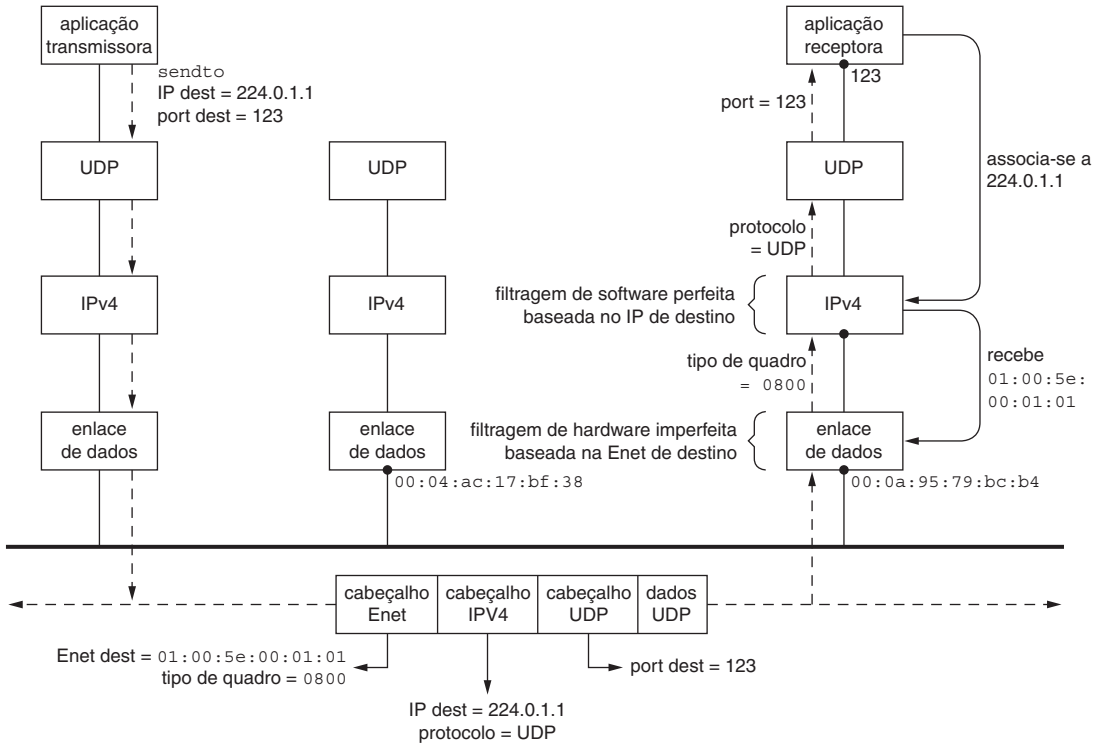


Figura 21.4 Exemplo de multicast de um datagrama UDP.

camada IPv4 salva as informações internamente e então informa o enlace de dados apropriado a receber os quadros Ethernet destinados a `01:00:5e:00:01:01` (Seção 12.11 do TCPv2). Esse é o endereço Ethernet correspondente ao endereço multicast ao qual a aplicação acabou de se associar utilizando o mapeamento que mostramos na Figura 21.1.

O próximo passo é a aplicação transmissora no host mais à esquerda criar um soquete UDP e enviar um datagrama a 224.0.1.1, porta 123. Nada especial é necessário para enviar um datagrama multicast: a aplicação não precisa se associar ao grupo multicast. O host transmissor converte o endereço IP no endereço Ethernet de destino correspondente e o quadro é enviado. Observe que o quadro contém tanto o endereço Ethernet de destino (examinado pelas interfaces) como o endereço IP de destino (examinado pelas camadas IP).

Assumimos que o host no meio não seja um capaz de multicast IPv4 (uma vez que o suporte para multicast IPv4 é opcional). Esse host ignora o quadro completamente porque: (i) o endereço Ethernet de destino não corresponde ao endereço da interface; (ii) o endereço Ethernet de destino não é o endereço Ethernet de broadcast; e (iii) a interface não foi instruída a receber nenhum endereço de grupo (aqueles com o bit de ordem inferior do byte de ordem superior configurado como 1, como na Figura 21.1).

O quadro é recebido pelo enlace de dados à direita com base no que chamamos *filtragem imperfeita*, que é feita pela interface utilizando o endereço Ethernet de destino. Dizemos que é imperfeita porque normalmente, quando a interface é instruída a receber quadros destinados a um endereço Ethernet multicast específico, ela também pode receber quadros destinados a outros endereços Ethernet multicast.

> Quando informadas a receber quadros destinados a um endereço Ethernet multicast específico, várias placas de interface Ethernet atuais aplicam uma função de hash ao endereço, calculando um valor entre 0 e 511. Um dos 512 bits em um array é então ativado. Quando um quadro passa pelo cabo destinado a um endereço multicast, a mesma função de hash é aplicada pela interface ao endereço de destino (o primeiro campo no quadro), calculando um valor entre 0 e 511. Se o bit correspondente no array estiver ativado, o quadro é recebido; caso contrário, é ignorado. Placas de interface mais antigas reduzem o tamanho do array de bits de 512 para 64, aumentando a probabilidade de que uma interface receberá os quadros nos quais não está interessada. Ao longo do tempo, à medida que um número cada vez maior de aplicações utilizar o multicast, esse tamanho provavelmente aumentará ainda mais. Algumas placas de interface atuais já têm uma filtragem perfeita (a capacidade de ignorar datagramas destinados a todos os endereços, exceto aqueles endereços multicast desejados). Outras placas de interface não têm absolutamente nenhuma filtragem de multicast e, quando informadas a receber um endereço multicast específico, devem receber todos os quadros multicast (às vezes chamado modo *multicast promíscuo*). Uma placa de interface popular faz uma filtragem perfeita em 16 endereços multicast e também tem uma tabela de hash de 512 bits. Uma outra faz uma filtragem perfeita em 80 endereços multicast, mas então tem de entrar no modo promíscuo de multicast. Mesmo se a interface realizar uma filtragem perfeita, a filtragem perfeita de software na camada IP ainda é requerida porque o mapeamento do endereço multicast do IP para o endereço de hardware não é de um para um.

Supondo que o enlace de dados à direita receba o quadro, uma vez que o tipo de quadro Ethernet é IPv4, o pacote é passado para a camada IP. Como o pacote recebido era destinado a um endereço multicast IP, a camada IP compara esse endereço contra todos os endereços multicast aos quais as aplicações nesse host se associaram. Chamamos isso de *filtragem perfeita,* uma vez que está completamente baseada no endereço da classe D de 32 bits no cabeçalho IPv4. Nesse exemplo, o pacote é aceito pela camada IP e passado para a camada UDP que, por sua vez, passa o datagrama para o soquete que está vinculado à porta 123.

Há três cenários que não mostramos na Figura 21.4:

1. Um host que executa uma aplicação que se associou ao endereço multicast 225.0.1.1. Como os cinco bits superiores do endereço de grupo são ignorados no mapeamento para o endereço Ethernet, essa interface do host também irá receber os quadros com

um endereço Ethernet de destino de `01:00:5e:00:01:01`. Nesse caso, o pacote será descartado pela filtragem perfeita na camada IP.

2. Um host executando uma aplicação que se associou a algum grupo multicast cujo endereço Ethernet correspondente seja um que a interface recebe quando está programada para receber `01:00:5e:00:01:01` (isto é, a placa de interface realiza uma filtragem imperfeita). Esse quadro será descartado pela camada do enlace de dados ou pela camada IP.
3. Um pacote destinado ao mesmo grupo, 224.0.1.1, mas a uma porta diferente, digamos 4000. O host mais à direita na Figura 21.4 continua a receber o pacote, que é aceito pela camada IP, mas, assumindo que não haja um soquete vinculado à porta 4000, o pacote será descartado pela camada UDP.

   Isso demonstra que, para um processo receber um datagrama multicast, o processo deve se associar ao grupo e vincular a porta.

21.4 Multicast em uma WAN

O multicast em uma única rede local, como discutido na seção anterior, é simples. Um host envia um pacote multicast e qualquer outro host interessado o recebe. O benefício do multicast em relação ao broadcast é reduzir a carga sobre todos os hosts não-interessados nos pacotes multicast.

O multicast é também benéfico em redes geograficamente distribuídas (*wide area network* – WAN). Considere a WAN mostrada na Figura 21.5, que mostra cinco redes locais conectadas com cinco roteadores multicast.

Em seguida, suponha que algum programa é iniciado nos cinco hosts (digamos que um programa que escuta uma sessão de áudio de multicast) e esses cinco programas se associam a um dado grupo multicast. Cada um dos cinco hosts se associa então a esse grupo multicast. Também supomos que todos os roteadores multicast estão se comunicando com seus roteadores multicast vizinhos utilizando um *protocolo de roteamento multicast*, que simplesmente denominamos *MRP* (*multicast routing protocol*). Mostramos isso na Figura 21.6.

Quando um processo em um host se associa a um grupo multicast, esse host envia uma mensagem IGMP a quaisquer roteadores multicast conectados, informando-os que o host acabou de se associar a esse grupo. Os roteadores multicast trocam então essas informações utilizando o MRP de modo que cada roteador multicast saiba o que fazer se receber um pacote destinado ao endereço multicast.

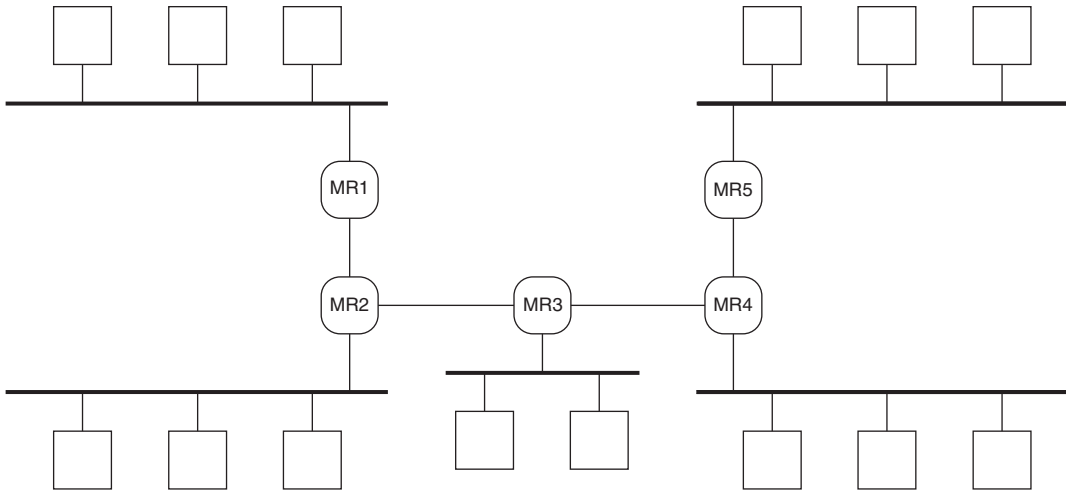


Figura 21.5 Cinco redes locais conectadas com cinco roteadores multicast.

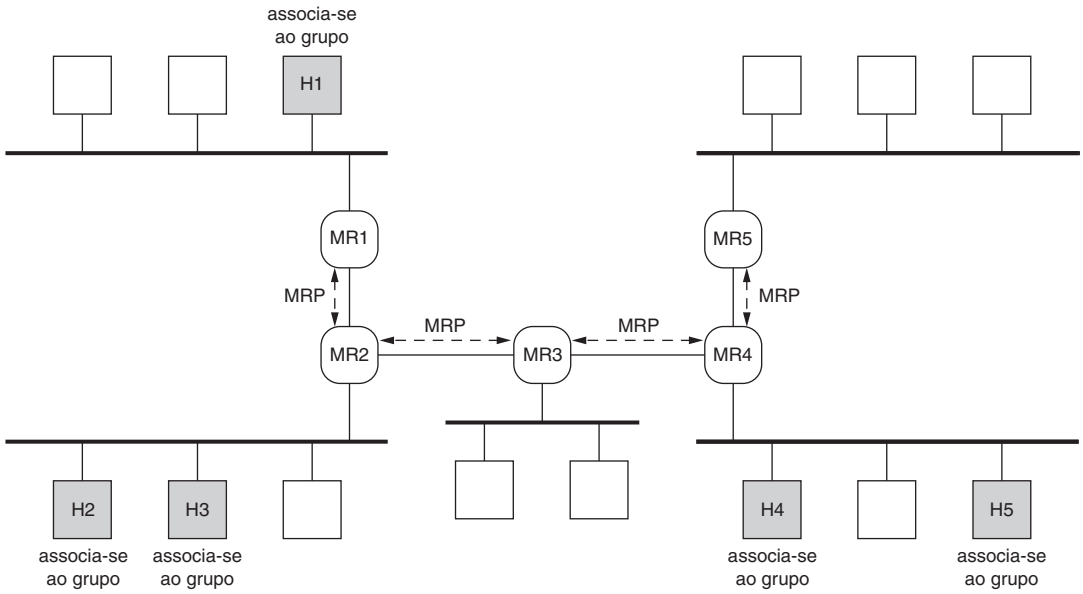


Figura 21.6 Cinco hosts se associam a um grupo multicast em uma WAN.

O roteamento multicast ainda é um tópico de pesquisa e, por si só, poderia facilmente ocupar um livro inteiro.

Agora, supomos que um processo no host na parte superior esquerda inicia o envio de pacotes destinados ao endereço multicast. Digamos que esse processo está enviando os pacotes de áudio que os receptores multicast estão esperando receber. Mostramos esses pacotes na Figura 21.7.

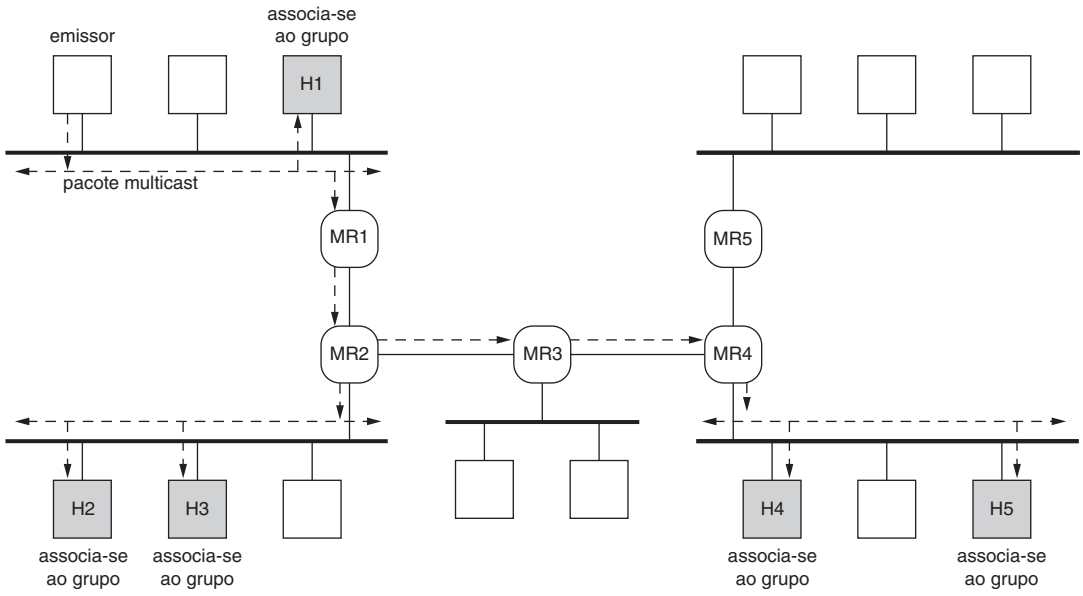


Figura 21.7 Enviando pacotes multicast em uma WAN.

Podemos acompanhar os passos à medida que os pacotes multicast vão do emissor a todos os receptores:

- Os pacotes são transmitidos por multicast na parte superior esquerda da rede local pelo emissor. O receptor H1 recebe esses pacotes (uma vez que se associou ao grupo), assim como o MR1 (uma vez que um roteador multicast deve receber todos pacotes multicast).
- O MR1 encaminha o pacote multicast ao MR2, porque o MRP informou ao MR1 que o MR2 precisa receber os pacotes destinados a esse grupo.
- O MR2 transmite por multicast o pacote a sua rede local anexada, visto que os hosts H2 e H3 pertencem ao grupo. Ele também faz uma cópia do pacote e o envia ao MR3.

  Fazer uma cópia do pacote, como o MR2 faz aqui, é algo único no encaminhamento de multicast. Um pacote unicast nunca é duplicado quando é encaminhado pelos roteadores.
- O MR3 envia o pacote multicast ao MR4, mas não transmite por multicast uma cópia na sua rede local anexada porque supomos que nenhum host na rede local se associou ao grupo.
- O MR4 transmite por multicast o pacote a sua rede local anexada, visto que os hosts H4 e H5 pertencem ao grupo. Ele não faz uma cópia e a envia ao MR5 porque nenhum dos hosts anexados à rede local do MR5 pertence ao grupo e o MR4 sabe disso com base nas informações do roteamento multicast trocadas com o MR5.

Duas alternativas menos desejáveis ao multicast em uma WAN são a *inundação de broadcast (broadcast flooding)* e o envio de cópias individuais a cada receptor. No primeiro caso, os pacotes seriam difundidos por broadcast pelo emissor e cada roteador difundiria por broadcast os pacotes de cada uma das suas interfaces, exceto a interface receptora. É óbvio que isso aumenta o número de hosts e roteadores não-interessados que devem lidar com o pacote.

No segundo caso, o emissor precisa conhecer o endereço IP de todos os receptores e enviar uma cópia a cada um. Com os cinco receptores que mostramos na Figura 21.7, isso iria requerer cinco pacotes na rede local do emissor, quatro pacotes indo do MR1 ao MR2 e dois pacotes indo do MR2 ao MR3 ao MR4. Agora, simplesmente imagine uma situação com um milhão de receptores!

21.5 Multicast de origem específica

O multicast em uma WAN tem sido difícil de instalar por várias razões. O maior problema é que o *MRP*, descrito na Seção 21.4, precisa obter os dados de todos os emissores, que poderiam estar localizados em qualquer lugar na rede, para todos os receptores, os quais, igualmente, poderiam estar localizados em um lugar qualquer. Um outro grande problema é a alocação do endereço multicast: não há endereços multicast IPv4 suficientes para atribuí-los estaticamente a todas as pessoas que querem um, como é feito com endereços unicast. Para enviar multicast na WAN sem entrar em conflito com os outros emissores de multicast, você precisa de um endereço único, porém, ainda não há um mecanismo global de alocação de endereços multicast.

O multicast de origem específica, (*source-specific multicast* – SSM) (Holbrook e Cheriton, 1999), fornece uma solução pragmática para esses problemas. Ele combina o endereço de grupo com um endereço de origem do sistema, o que resolve os problemas da seguinte maneira:

- Os receptores fornecem o endereço de origem do emissor para os roteadores como parte da associação ao grupo. Isso remove o problema do "*rendezvous*" na rede, pois esta agora sabe exatamente onde o emissor está. Mas isso mantém as propriedades de escalonamento por não exigir que o emissor saiba quem são todos os receptores. Isso simplifica imensamente os protocolos de roteamento multicast.

- Isso redefine o identificador como sendo simplesmente um endereço de grupo multicast para uma combinação de uma origem unicast e um destino multicast (que o SSM agora chama de *canal*). Isso significa que a origem pode selecionar qualquer endereço multicast desde que ele se torne a combinação (de origem ou destino) que deve ser única; e a origem já o torna único. Uma sessão SSM é a combinação entre origem, destino e porta.

O SSM também fornece um certo grau de anti-spoofing, isto é, é mais difícil para a origem 2 transmitir no canal da origem 1, visto que o canal da origem 1 inclui o endereço da origem 1. O spoofing ainda é possível, naturalmente, mas é muito mais difícil.

21.6 Opções de soquete multicast

O suporte API para o multicast tradicional requer somente cinco novas opções de soquete. O suporte à filtragem de origem, requerido para o *SSM*, adiciona outros quatro. A Figura 21.8 mostra as três opções de soquete relacionadas a não-associações, acrescido do tipo de dados do argumento esperado na chamada a `getsockopt` ou `setsockopt`. A Figura 21.9 mostra as seis opções de soquete relacionadas com associações a IPv4, IPv6 e para a versão da API independente do protocolo IP. Um ponteiro para uma variável do tipo de dados mostrado é o quarto argumento para `getsockopt` e `setsockopt`. Todas essas nove opções são válidas com `setsockopt`, mas as seis que se associam a (*join*) e saem de (*leave*) um grupo de multicast ou origem não são permitidas com `getsockopt`.

> As opções de loopback e TTL do IPv4 recebem um argumento `u_char`, enquanto as opções de loopback e limite de hop do IPv6 recebem um argumento `int` e um `u_int`, respectivamente. Um erro de programação comum com as opções de multicast do IPv4 é chamar `setsockopt` com um argumento `int` para especificar o TTL ou o loopback (o

Comando	Tipo de dados	Descrição
IP_MULTICAST_IF	struct in_addr	Especifica a interface-padrão para transmissões multicast saintes
IP_MULTICAST_TTL	u_char	Especifica o TTL para transmissões multicast saintes
IP_MULTICAST_LOOP	u_char	Ativa ou desativa o loopback das transmissões multicast saintes
IPV6_MULTICAST_IF	u_int	Especifica a interface-padrão para transmissões multicast saintes
IPV6_MULTICAST_HOPS	int	Especifica o limite de hop para transmissões multicasts saintes
IPV6_MULTICAST_LOOP	u_int	Ativa ou desativa o loopback das transmissões multicast saintes

Figura 21.8 Opções de soquete multicast.

Comando	Tipo de dados	Descrição
IP_ADD_MEMBERSHIP	struct ip_mreq	Associa-se a um grupo multicast
IP_DROP_MEMBERSHIP	struct ip_mreq	Sai de um grupo multicast
IP_BLOCK_SOURCE	struct ip_mreq_source	Bloqueia uma origem em um grupo associado
IP_UNBLOCK_SOURCE	struct ip_mreq_source	Desbloqueia uma origem anteriormente bloqueada
IP_ADD_SOURCE_MEMBERSHIP	struct ip_mreq_source	Associa-se a um grupo de origem específica
IP_DROP_SOURCE_MEMBERSHIP	struct ip_mreq_source	Sai de um grupo de origem específica
IPV6_JOIN_GROUP	struct ipv6_mreq	Associa-se a um grupo multicast
IPV6_LEAVE_GROUP	struct ipv6_mreq	Sai de um grupo multicast
MCAST_JOIN_GROUP	struct group_req	Associa-se a um grupo multicast
MCAST_LEAVE_GROUP	struct group_req	Sai de um grupo multicast
MCAST_BLOCK_SOURCE	struct group_source_req	Bloqueia uma origem em um grupo associado
MCAST_UNBLOCK_SOURCE	struct group_source_req	Desbloqueia uma origem anteriormente bloqueada
MCAST_JOIN_SOURCE_GROUP	struct group_source_req	Associa-se a um grupo de origem específica
MCAST_LEAVE_SOURCE_GROUP	struct group_source_req	Sai de um grupo de origem específica

Figura 21.9 As opções de soquete para composição de grupo multicast.

que não é permitido; páginas 354 e 355 do TCPv2), uma vez que a maioria das outras opções de soquete na Figura 7.1 tem argumentos do tipo inteiro. A alteração no IPv6 torna-as mais consistentes com outras opções.

Agora, descreveremos cada uma dessas nove opções de soquete em mais detalhes. Observe que as nove opções são conceitualmente idênticas entre o IPv4 e o IPv6; somente o nome e o tipo de argumento são diferentes.

`IP_ADD_MEMBERSHIP, IPV6_JOIN_GROUP, MCAST_JOIN_GROUP`

Esta opção associa um grupo multicast de qualquer origem em uma interface local especificada. Especificamos a interface local com um dos seus endereços unicast para o IPv4 ou com o índice de interface para o IPv6 e a API independente de protocolo. As três estruturas a seguir são utilizadas ao associar-se a um grupo ou ao sair de um grupo:

```
struct ip_mreq {
  struct in_addr   imr_multiaddr;        /* endereço multicast IPv4 classe D */
  struct in_addr   imr_interface;        /* endereço IPv4 da interface local */
};

struct ipv6_mreq {
  struct in6_addr  ipv6mr_multiaddr;     /* endereço IPv6 multicast */
  unsigned int     ipv6mr_interface;     /* índice de interface ou 0 */
};

struct group_req {
  unsigned int             gr_interface; /* índice de interface ou 0 */
  struct sockaddr_storage  gr_group;     /* endereço multicast IPv4 ou IPv6 */
}
```

Se a interface local for especificada como o endereço curinga para o IPv4 (`INADDR_ANY`) ou como um índice de 0 para o IPv6, uma única interface local é então escolhida pelo kernel.

Dizemos que um host pertence a um dado grupo multicast em uma dada interface, se um ou mais processos pertencerem atualmente a esse grupo nessa interface.

Mais de uma associação é permitida em um dado soquete, mas cada uma delas deve ser com um endereço multicast diferente, ou com o mesmo endereço multicast mas em uma interface diferente das associações anteriores a esse endereço nesse soquete. Isso pode ser utilizado em um host multihomed em que, por exemplo, um soquete é criado e, então, para cada interface, uma associação é realizada com um dado endereço multicast.

Lembre-se, da Figura 21.3, de que endereços multicast IPv6 têm um campo de escopo explícito como parte do endereço. Como observamos, os endereços multicast IPv6 que diferem somente em escopo representam grupos diferentes. Portanto, se uma implementação de NTP quisesse receber todos os pacotes NTP, independentemente do escopo, ela teria de se associar a `ff01::101` (interface local), `ff02::101` (enlace local), `ff05::101` (site local), `ff08::101` (organização local) e `ff0e::101` (global). Todas as associações poderiam ser realizadas em um único soquete e a opção de soquete `IPV6_PKTINFO` poderia ser configurada (Seção 22.8) para fazer com que `recvmsg` retornasse o endereço de destino de cada datagrama.

A opção de soquete independente de protocolo IP (`MCAST_JOIN_GROUP`) é a mesma da opção IPv6, exceto que utiliza um `sockaddr_storage` em vez de `in6_addr` para passar o endereço de grupo para o kernel. Um `sockaddr_storage` (Figura 3.5) é suficientemente grande para armazenar qualquer tipo de endereço que o sistema suporta.

A maioria das implementações tem um limite quanto ao número de associações permitido por soquete. Esse limite é especificado como `IP_MAX_MEMBERSHIPS` (20 para im-

plementações derivadas do Berkeley), mas algumas implementações ampliaram essa limitação ou aumentaram o número máximo.

Se a interface à qual se associar não estiver especificada, os kernels derivados do Berkeley pesquisam o endereço multicast na tabela de roteamento do IP normal e utilizam a interface resultante (página 357 do TCPv2). Alguns sistemas instalam uma rota para todos os endereços multicast (isto é, uma rota com um destino de 224.0.0.0/8 para o IPv4) na inicialização para tratar esse cenário.

A alteração foi feita no IPv6 e as opções independentes de protocolo utilizam um índice de interface para especificar a interface, em vez do endereço unicast local utilizado com o IPv4 para permitir associações em interfaces não-numeradas e extremidades do túnel.

A definição da API de multicast IPv6 original utilizava `IPV6_ADD_MEMBERSHIP` em vez de `IPV6_JOIN_GROUP`. A API, do contrário, é a mesma. Nossa função `mcast_join`, descrita mais adiante, oculta essa diferença.

`IP_DROP_MEMBERSHIP, IPV6_LEAVE_GROUP, MCAST_LEAVE_GROUP`

Esta opção permite sair de um grupo multicast de qualquer origem em uma interface local especificada. As mesmas estruturas que acabamos de mostrar para associar-se a um grupo são utilizadas com essa opção de soquete. Se a interface local não estiver especificada (isto é, o valor é `INADDR_ANY` para o IPv4 ou tem um índice de interface de 0 para o IPv6), a primeira composição (*membership*) de grupo multicast que corresponda é o grupo que será abandonado (e a composição do grupo alterada para refletir a saída desse processo).

Se um processo associar-se a um grupo, mas nunca sair dele explicitamente, quando o soquete é fechado (explicitamente ou no término do processo), o processo será retirado da composição do grupo automaticamente. É possível que múltiplos soquetes em um host se associem ao mesmo grupo, nesse caso, o host permanece um membro desse grupo até que o último soquete deixe o grupo.

A definição da API de multicast IPv6 original utilizava `IPV6_DROP_MEMBERSHIP` em vez do `IPV6_LEAVE_GROUP`. A API, ao contrário, é a mesma. Nossa função `mcast_leave`, descrita mais adiante, oculta essa diferença.

`IP_BLOCK_SOURCE, MCAST_BLOCK_SOURCE`

Bloqueia a recepção do tráfego nesse soquete a partir de uma origem dada uma composição de grupo de qualquer origem existente em uma interface local especificada. Se todos os soquetes associados bloquearam a mesma origem, o sistema pode informar aos roteadores que esse tráfego não é desejável, possivelmente afetando o roteamento multicast na rede. Por exemplo, isso pode ser utilizado para ignorar o tráfego de falsos emissores. Especificamos a interface local com um dos seus endereços unicast para o IPv4 ou com o índice de interface para a API independente de protocolo. As duas estruturas a seguir são utilizadas ao bloquear ou desbloquear uma origem:

```
struct ip_mreq_source {
  struct in_addr   imr_multiaddr;       /* endereço multicast IPv4 classe D */
  struct in_addr   imr_sourceaddr;      /* endereço de origem IPv4 */
  struct in_addr   imr_interface;       /* endereço IPv4 da interface local */
};
struct group_source_req {
  unsigned int            gsr_interface; /* índice de interface ou 0 */
  struct sockaddr_storage gsr_group;     /* endereço multicast IPv4 ou IPv6 */
  struct sockaddr_storage gsr_source;    /* endereço de origem IPv4 ou IPv6 */
}
```

Se a interface local estiver especificada como o endereço curinga para o IPv4 (`INADDR_ANY`) ou como um índice de 0 para a API independente de protocolo, a interface local será então escolhida pelo kernel para corresponder à primeira composição nesse soquete para o grupo dado.

A solicitação de bloqueio de origem modifica uma composição de grupo existente, portanto, o grupo precisa já ter sido associado à interface especificada com uma das opções `IP_ADD_MEMBERSHIP`, `IPV6_JOIN_GROUP` ou `MCAST_JOIN_GROUP`.

`IP_UNBLOCK_SOURCE, MCAST_UNBLOCK_SOURCE`

Desbloqueia uma origem anteriormente bloqueada. Os argumentos devem ser os mesmos de uma solicitação `IP_BLOCK_SOURCE` ou `MCAST_BLOCK_SOURCE` anterior nesse soquete.

Se a interface local estiver especificada como o endereço curinga para o IPv4 (`INADDR_ANY`) ou como um índice de 0 para a API independente de protocolo, a primeira origem bloqueada correspondente é então desbloqueada.

`IP_ADD_SOURCE_MEMBERSHIP, MCAST_JOIN_SOURCE_GROUP`

Associa um grupo de origem específica a uma interface local especificada. As mesmas estruturas recém-mostradas para bloquear ou desbloquear origens são utilizadas com essa opção de soquete. O grupo não deve ter sido ainda associado utilizando a interface de qualquer origem (`IP_ADD_MEMBERSHIP`, `IPV6_JOIN_GROUP` ou `MCAST_JOIN_GROUP`).

Se a interface local estiver especificada como o endereço curinga para o IPv4 (`INADDR_ANY`), ou como um índice de 0 para a API independente de protocolo, ela é então escolhida pelo kernel.

`IP_DROP_SOURCE_MEMBERSHIP, MCAST_LEAVE_SOURCE_GROUP`

Esta opção permite sair de um grupo de origem específica em uma interface local especificada. As mesmas estruturas recém-mostradas para associar um grupo de origem específica são utilizadas com essa opção de soquete. Se a interface local não estiver especificada (isto é, o valor é `INADDR_ANY` para o IPv4 ou tiver um índice de interface de 0 para a API independente de protocolo), a primeira composição específica da origem correspondente é escolhida.

Se um processo se associar a um grupo de origem específica, mas nunca sair do grupo explicitamente, quando o soquete for fechado (explicitamente ou no término do processo), o processo será retirado automaticamente da composição. É possível que múltiplos processos em um host se associem ao mesmo grupo de origem específica; nesse caso, o host permanece como um membro desse grupo até que o último processo saia do grupo.

`IP_MULTICAST_IF, IPV6_MULTICAST_IF`

Especifica a interface para datagramas multicast saintes enviados a esse soquete. Essa interface é especificada como uma estrutura `in_addr` para o IPv4 ou como um índice de interface para o IPv6. Se o valor especificado for `INADDR_ANY` para o IPv4 ou um índice de interface de 0 para o IPv6, é removida qualquer interface atribuída anteriormente por essa opção de soquete e o sistema escolherá a interface toda vez que um datagrama for enviado.

Tenha cuidado em distinguir entre a interface local especificada (ou escolhida) quando um processo se associa a um grupo (a interface na qual os datagramas multicast entrantes serão recebidos) e a interface local especificada (ou escolhida) quando é gerada a saída de um datagrama multicast.

Kernels derivados do Berkeley escolhem a interface default para um datagrama multicast sainte pesquisando na tabela normal de roteamento do IP uma rota para o endereço multicast de destino, e a interface correspondente é utilizada. Essa mesma técnica é utilizada para escolher a interface receptora se o processo não especificar uma ao associar-se a um grupo. A suposição é de que, se há uma rota para um dado endereço multicast (talvez a ro-

ta default na tabela de roteamento), a interface resultante deverá então ser utilizada para entrada e saída.

`IP_MULTICAST_TTL, IPV6_MULTICAST_HOPS`

Configura o TTL do IPv4 ou o limite de hop do IPv6 para datagramas multicast saintes. Se isso não for especificado, ambos assumirão 1 por default, o que restringe o datagrama à sub-rede local.

`IP_MULTICAST_LOOP, IPV6_MULTICAST_LOOP`

Ativa ou desativa o loopback local dos datagramas multicast. Por default, o loopback é ativado: uma cópia de cada datagrama multicast enviado por um processo no host também será retornada e processada como um datagrama recebido por esse host, se o host pertencer a esse grupo multicast na interface de saída.

Isso é semelhante ao broadcasting quando vimos que broadcasts enviados por um host também são processadas como um datagrama recebido nesse host (Figura 20.4). (Com o broadcasting, não há nenhuma maneira de desativar esse loopback.) Isso significa que, se um processo pertencer ao grupo multicast ao qual está enviando datagramas, ele receberá suas próprias transmissões.

O loopback descrito aqui é um loopback interno realizado na camada IP ou em uma mais alta. Se a interface escutar suas próprias transmissões, a RFC 1112 (Deering, 1989) requer que o driver descarte essas cópias. Essa RFC também afirma que a opção de loopback assume o default de ON (ligado) como "uma otimização de desempenho para protocolos da camada superior que restringe a composição de um grupo a um processo por host (como um protocolo de roteamento)".

Os primeiros seis pares de opções de soquete (`ADD_MEMBERSHIP/JOIN_GROUP`, `DROP_MEMBERSHIP/LEAVE_GROUP`, `BLOCK_SOURCE`, `UNBLOCK_SOURCE`, `ADD_SOURCE_MEMBERSHIP/JOIN_SOURCE_GROUP` e `DROP_SOURCE_MEMBERSHIP/LEAVE_SOURCE_GROUP`) afetam o *recebimento* de datagramas multicast, enquanto os últimos três pares afetam o *envio* de datagramas multicast (interface de saída, TTL ou limite de hop e loopback). Mencionamos anteriormente que nada especial é requerido para enviar um datagrama multicast. Se nenhuma opção de soquete multicast estiver especificada antes do envio de um datagrama multicast, a interface para o datagrama sainte será escolhida pelo kernel, o TTL ou o limite de hop será 1 e será feito o loopback em uma cópia.

Para receber um datagrama multicast, um processo deve se associar ao grupo multicast e também vincular (`bind`) um soquete UDP ao número da porta que será utilizado como o número da porta de destino para datagramas enviados ao grupo. As duas operações são distintas e ambas são requeridas. Associar-se ao grupo instrui a camada IP do host e a camada de enlace de dados a receber datagramas multicast enviados a esse grupo. Vincular-se à porta é a maneira como a aplicação especifica ao UDP que ela quer receber datagramas enviados a essa porta. Algumas aplicações também utilizam `bind` para vincular o endereço multicast ao soquete, além da porta. Isso impede que quaisquer outros datagramas que poderiam ser recebidos nessa porta para outros endereços unicast, broadcast ou multicast sejam enviados ao soquete.

Historicamente, a interface do serviço multicast exigia que somente *algum* soquete no host se associasse ao grupo multicast, não necessariamente o soquete que se vincula à porta e então recebe os datagramas multicast. Há o potencial, porém, com essas implementações, de que datagramas multicast sejam enviados a aplicações que não estão cientes de multicast. Kernels multicast mais recentes agora requerem que o processo vincule-se à porta e configure todas as opções de soquete multicast para o soquete, sendo a última uma indicação de que a aplicação é "ciente" de multicast. A opção de soquete multicast mais comum a configurar é a associação do grupo. O Solaris difere ligeiramente e somen-

te entrega datagramas multicast recebidos para um soquete que tenha se associado ao grupo e se vinculado à porta. Para portabilidade, todas as aplicações de multicast devem se associar ao grupo e se vincular à porta.

A interface do serviço multicast mais recente exige que a camada IP somente entregue pacotes multicast para um soquete se esse soquete tiver sido associado ao grupo e/ou origem aplicável. Isso foi introduzido no IGMPv3 (RFC 3376 [Cain *et al.*, 2002]) para permitir filtragem de origem e multicast de origem específica. Isso reforça o requisito de associação ao grupo, mas relaxa o requisito de vinculação ao endereço do grupo. Contudo, para máxima portabilidade tanto para interfaces de serviço multicast novas como para as antigas, as aplicações devem se associar ao grupo e se vincular ao endereço do mesmo.

Alguns hosts mais antigos capazes de multicast não permitem utilizar `bind` para vincular um endereço multicast a um soquete. Para portabilidade, uma aplicação talvez deseje ignorar um erro de `bind` para um endereço multicast e tentar novamente utilizando `INADDR_ANY` ou `in6addr_any`.

21.7 `mcast_join` e funções relacionadas

Embora as opções de soquete multicast para o IPv4 sejam semelhantes àquelas para o IPv6, há muitas diferenças que levam o código independente de protocolo que utiliza o multicast a

```
#include "unp.h"

int mcast_join(int sockfd, const struct sockaddr *grp, socklen_t grplen,
               const char *ifname, u_int ifindex);

int mcast_leave(int sockfd, const struct sockaddr *grp, socklen_t grplen);

int mcast_block_source(int sockfd,
               const struct sockaddr *src, socklen_t srclen,
               const struct sockaddr *grp, socklen_t grplen);

int mcast_unblock_source(int sockfd,
               const struct sockaddr *src, socklen_t srclen,
               const struct sockaddr *grp, socklen_t grplen);

int mcast_join_source_group(int sockfd,
               const struct sockaddr *src, socklen_t srclen,
               const struct sockaddr *grp, socklen_t grplen,
               const char *ifname, u_int ifindex);

int mcast_leave_source_group(int sockfd,
               const struct sockaddr *src, socklen_t srclen,
               const struct sockaddr *grp, socklen_t grplen);

int mcast_set_if(int sockfd, const char *ifname, u_int ifindex);

int mcast_set_loop(int sockfd, int flag);

int mcast_set_ttl(int sockfd, int ttl);
```

Todas retornam: 0 se OK, –1 em erro

```
int mcast_get_if(int sockfd);
```

Retorna: índice de interface não-negativo se OK, –1 em erro

```
int mcast_get_loop(int sockfd);
```

Retorna: flag do loopback atual se OK, –1 em erro

```
int mcast_get_ttl(int sockfd);
```

Retorna: TTL atual ou limite de hop se OK, –1 em erro

tornar-se complicado com uma grande quantidade de `#ifdefs`. Uma solução melhor é ocultar as diferenças dentro destas oito funções:

`mcast_join` cria uma associação ao grupo multicast de qualquer origem cujo endereço IP é contido dentro da estrutura de endereço de soquete apontada por *grp* e cujo comprimento é especificado por *grplen*. Podemos especificar a interface à qual associar o grupo pelo nome desta (um *ifname* não-nulo) ou por um índice de interface não-zero (*ifindex*). Se nenhum for especificado, o kernel escolhe a interface na qual o grupo é associado. Lembre-se de que, com o IPv6, a interface é especificada para a opção de soquete pelo seu índice. Se um nome for especificado para um soquete IPv6, chamamos `if_nametoindex` para obter o índice. Com a opção de soquete IPv4, a interface é especificada pelo seu endereço unicast IP. Se um nome for especificado para um soquete IPv4, chamamos `ioctl` com uma solicitação de `SIOCGIFADDR` para obter o endereço unicast IP para a interface. Se um índice for especificado para um soquete IPv4, primeiro chamamos `if_indextoname` para obter o nome e então processamos o nome da maneira recém-descrita.

> Um nome de interface, como `le0` ou `ether0`, é normalmente a maneira como os usuários especificam interfaces, e não com o endereço IP ou com o índice. Por exemplo, `tcpdump` é um dos poucos programas que deixam o usuário especificar uma interface; e sua opção `-i` aceita um nome de interface como o argumento.

`mcast_leave` permite sair do grupo multicast cujo endereço IP está contido dentro da estrutura de endereço de soquete apontada por *grp*. Observe que `mcast_leave` não aceita uma especificação de interface; ele sempre exclui da primeira composição correspondente. Isso simplifica a API da biblioteca, mas significa que programas que requerem controle direto da composição por interface precisam utilizar a API `setsockopt` diretamente.

`mcast_block_source` bloqueia a recepção em um dado soquete da origem e do grupo cujos endereços IP estejam contidos dentro das estruturas de endereço de soquete apontadas por *src* e *grp*, respectivamente, e cujos comprimentos sejam especificados por *srclen* e *grplen*. `mcast_join` já deve ter sido chamada nesse soquete para o grupo dado.

`mcast_unblock_source` desbloqueia a recepção de tráfego de uma dada origem para um dado grupo. Os argumentos *src*, *srclen*, *grp* e *grplen* devem ser os mesmos de uma chamada anterior a `mcast_block_source`.

`mcast_join_source_group` realiza associação ao grupo de origem específica em que os endereços IP da origem e do grupo estão contidos dentro das estruturas de endereço de soquete apontadas por *src* e *grp*, respectivamente, e cujos comprimentos são especificados por *srclen* e *grplen*. Podemos especificar a interface à qual associar o grupo pelo nome desta (um *ifname* não-nulo) ou por um índice de interface não-zero (*ifindex*). Se nenhum for especificado, o kernel escolhe a interface em que o grupo está associado.

`mcast_leave_source_group` permite sair do grupo multicast de origem específica cujos endereços IP da origem e do grupo estão contidos dentro da estrutura de endereço de soquete apontada por *src* e *grp*, respectivamente, e cujos comprimentos são especificados por *srclen* e *grplen*. Como ocorre com `mcast_leave`, `mcast_leave_source_group` não aceita uma especificação de interface; sempre exclui da primeira composição correspondente.

`mcast_set_if` configura o índice de interface default para datagramas multicast saintes. Se *ifindex* for maior que 0, ele especificará o índice da interface; caso contrário, se *ifname* for não-nulo, ele especificará o nome da interface. Para o IPv6, o nome é mapeado para um índice utilizando `if_nametoindex`. Para o IPv4, o mapeamento de um nome ou um índice para o endereço unicast IP da interface ocorre da maneira como descrita para `mcast_join`.

`mcast_set_loop` configura a opção de loopback como 0 ou 1 e `mcast_set_ttl` configura o TTL do IPv4 ou o limite de hop do IPv6. As três funções `mcast_get_`*XXX* retornam o valor correspondente.

Exemplo: Função `mcast_join`

A Figura 21.10 mostra a primeira das nossas três funções `mcast_join`. Essa terceira função mostra como uma API independente de protocolo pode ser simples e direta.

Tratando o índice

9–17 Se o chamador tiver fornecido um índice, nós o utilizamos diretamente. Caso contrário, se o chamador tiver fornecido um nome de interface, o índice é obtido chamando `if_nametoindex`. Caso contrário, a interface é configurada como 0, instruindo o kernel a escolhê-la.

Copiando o endereço e chamando `setsockopt`

18–22 O endereço de soquete do chamador é copiado diretamente para o campo grupo da solicitação. Lembre-se de que o campo grupo é um `sockaddr_storage`, portanto, ele é suficientemente grande para tratar qualquer tipo de endereço de soquete que o sistema suporte. Entretanto, para se prevenir contra estouros de buffer causados por uma programação relapsa, verificamos o tamanho de `sockaddr` e retornamos `EINVAL` se for muito grande.

23–24 `setsockopt` realiza a associação. O argumento *level (nível)* para `setsockopt` é determinado utilizando a família de endereços do grupo e nossa função `family_to_level`. Alguns sistemas suportam uma não-correspondência entre *level* e a família de endereços do soquete, por exemplo, utilizando `IPPROTO_IP` com `MCAST_JOIN_GROUP`, mesmo com um soquete `AF_INET6`, mas nem todos a suportam, assim transformamos a família de endereços no nível apropriado. Não mostramos essa função trivial, mas o código-fonte está livremente disponível (consulte o Prefácio).

lib/mcast_join.c

```
 1 #include    "unp.h"
 2 #include    <net/if.h>

 3 int
 4 mcast_join(int sockfd, const SA *grp, socklen_t grplen,
 5            const char *ifname, u_int ifindex)
 6 {
 7 #ifdef MCAST_JOIN_GROUP
 8     struct group_req req;
 9     if (ifindex > 0) {
10         req.gr_interface = ifindex;
11     } else if (ifname != NULL) {
12         if ( (req.gr_interface = if_nametoindex(ifname)) == 0) {
13             errno = ENXIO;    /* nome não encontrado */
14             return (-1);
15         }
16     } else
17         req.gr_interface = 0;
18     if (grplen > sizeof(req.gr_group)) {
19         errno = EINVAL;
20         return -1;
21     }
22     memcpy(&req.gr_group, grp, grplen);
23     return (setsockopt(sockfd, family_to_level(grp->sa_family),
24                        MCAST_JOIN_GROUP,&req, sizeof(req)));
25 #else
```

lib/mcast_join.c

Figura 21.10 Associa a um grupo multicast: versão independente do IP.

A Figura 21.11 mostra o segundo terço da função `mcast_join`, que trata soquetes IPv4.

Tratando o índice

33–38 O endereço multicast IPv4 na estrutura de endereço de soquete é copiado para uma estrutura `ip_mreq`. Se um índice foi especificado, `if_indextoname` é chamado, armazenando o nome na nossa estrutura `ifreq`. Se isso for bem-sucedido, ramificamos na frente para emitir a `ioctl`.

Tratando o nome

39–46 O nome do chamador é copiado para uma estrutura `ifreq`; e uma função `ioctl` `SIOCGIFADDR` retorna o endereço unicast associado a esse nome. Em caso de sucesso, o endereço IPv4 é copiado para o membro `imr_interface` da estrutura `ip_mreq`.

Especificando o padrão

47–48 Se um índice e um nome não foram especificados, a interface é configurada como o endereço curinga, instruindo o kernel a escolhê-la.

49–50 `setsockopt` realiza a associação.

lib/mcast_join.c

```
26    switch (grp->sa_family) {
27    case AF_INET:{
28            struct ip_mreq mreq;
29            struct ifreq ifreq;

30            memcpy(&mreq.imr_multiaddr,
31                   &((const struct sockaddr_in *) grp)->sin_addr,
32                   sizeof(struct in_addr));

33            if (ifindex > 0) {
34               if (if_indextoname(ifindex, ifreq.ifr_name) == NULL) {
35                   errno = ENXIO; /* índice não encontrado */
36                   return (-1);
37               }
38               goto doioctl;
39            } else if (ifname != NULL) {
40               strncpy(ifreq.ifr_name, ifname, IFNAMSIZ);
41              doioctl:
42               if (ioctl(sockfd, SIOCGIFADDR, &ifreq) < 0)
43                   return (-1);
44               memcpy(&mreq.imr_interface,
45                      &((struct sockaddr_in *) &ifreq.ifr_addr)->sin_addr,
46                      sizeof(struct in_addr));
47            } else
48               mreq.imr_interface.s_addr = htonl(INADDR_ANY);

49            return (setsockopt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
50                               &mreq, sizeof(mreq)));
51        }
```

lib/mcast_join.c

Figura 21.11 Associação a um grupo multicast: soquete IPv4.

A parte final da função, que trata dos soquetes IPv6, é mostrada na Figura 21.12.

Copiando o endereço

55–57 Primeiro, o endereço multicast IPv6 é copiado da estrutura de endereço de soquete para a estrutura `ipv6_mreq`.

Tratando o índice, nome ou padrão

58–66 Se um índice foi especificado, ele será armazenado no membro `ipv6mr_interface`; se um nome foi especificado, o índice será obtido chamando `if_nametoindex`; caso contrário, o índice da interface será configurado como 0 para `setsockopt`, instruindo o kernel a escolher a interface.

67–68 O grupo é associado.

lib/mcast_join.c

```
52 #ifdef IPV6
53     case AF_INET6:{
54             struct ipv6_mreq mreq6;

55             memcpy(&mreq6.ipv6mr_multiaddr,
56                    &((const struct sockaddr_in6 *) grp)->sin6_addr,
57                    sizeof(struct in6_addr));

58             if (ifindex > 0) {
59                mreq6.ipv6mr_interface = ifindex;
60             } else if (ifname != NULL) {
61                if ( (mreq6.ipv6mr_interface = if_nametoindex(ifname)) == 0) {
62                    errno = ENXIO; /* nome não encontrado */
63                    return (-1);
64                }
65             } else
66                mreq6.ipv6mr_interface = 0;

67             return (setsockopt(sockfd, IPPROTO_IPV6, IPV6_JOIN_GROUP,
68                                &mreq6,sizeof(mreq6)));
69         }
70 #endif

71     default:
72         errno = EAFNOSUPPORT;
73         return (-1);
74     }
75 #endif
76 }
```

lib/mcast_join.c

Figura 21.12 Associação a um grupo multicast: soquete IPv6.

Exemplo: Função `mcast_set_loop`

A Figura 21.13 mostra nossa função `mcast_set_loop`.

Como o argumento é um descritor de soquete e não uma estrutura de endereço de soquete, chamamos nossa função `sockfd_to_family` para obter a família de endereços do soquete. A opção de soquete apropriada é configurada.

Não mostramos o código-fonte para todas as funções `mcast_`*XXX* remanescentes, mas ele está livremente disponível (consulte o Prefácio).

lib/mcast_set_loop.c

```
1 #include   "unp.h"

2 int
3 mcast_set_loop(int sockfd, int onoff)
4 {
5     switch (sockfd_to_family(sockfd)) {
6     case AF_INET:{
```

Figura 21.13 Configuração da opção multicast loopback (*continua*).

```
 7            u_char flag;
 8            flag = onoff;
 9            return (setsockopt(sockfd, IPPROTO_IP, IP_MULTICAST_LOOP,
10                               &flag, sizeof(flag)));
11        }
12 #ifdef IPV6
13    case AF_INET6:{
14             u_int flag;
15             flag = onoff;
16             return (setsockopt(sockfd, IPPROTO_IPV6, IPV6_MULTICAST_LOOP,
17                               &flag, sizeof(flag)));
18        }
19 #endif
20    default:
21        errno = EAFNOSUPPORT;
22        return (-1);
23    }
24 }
```

lib/mcast_set_loop.c

Figura 21.13 Configuração da opção multicast loopback (*continuação*).

21.8 Função `dg_cli` utilizando multicast

Modificamos nossa função `dg_cli` da Figura 20.5 simplesmente removendo a chamada a `setsockopt`. Como dissemos anteriormente, nenhuma das opções de soquete multicast precisa ser configurada para enviar um datagrama multicast se as configurações default para a interface de saída, o TTL e a opção de loopback forem aceitáveis. Executamos um servidor de eco UDP modificado que se associa ao grupo all-hosts e então executa nosso programa especificando o grupo all-hosts como o endereço de destino.

```
macosx % udpcli01 224.0.0.1
hi there
from 172.24.37.78: hi there                    MacOS X
from 172.24.37.94: hi there                    FreeBSD
```

Obtemos uma resposta de ambos os sistemas na sub-rede. Cada um deles está executando os servidores de eco de multicast. Cada resposta é unicast porque o endereço de origem da solicitação, utilizado por cada servidor como o endereço de destino da resposta, é um endereço de unicast.

Fragmentação de IP e multicasts

Mencionamos no final da Seção 20.4 que a maioria dos sistemas não permite a fragmentação de um datagrama de broadcast por decisão política. Não há problemas em utilizar fragmentação com multicast, como podemos facilmente verificar utilizando o mesmo arquivo com uma linha de 2.000 bytes.

```
macosx % udpcli01 224.0.0.1 < 2000line
from 172.24.37.78: xxxxxxxxxx[...]
from 172.24.37.94: xxxxxxxxxx[...]
```

21.9 Recebendo anúncios da sessão da infra-estrutura IP multicast

A infra-estrutura IP multicast é a parte da Internet com multicast interdomínio ativado. O multicast não é ativado na Internet inteira; a infra-estrutura IP multicast começou a existir em 1992 como o "MBone", uma rede de overlay, passando a ser instalada como parte da infra-es-

trutura da Internet em 1998. O multicast está amplamente instalado dentro das empresas, mas fazer parte da infra-estrutura IP multicast interdomínio é menos comum.

Para receber uma conferência multimídia na infra-estrutura IP multicast, um site precisa conhecer apenas o endereço multicast da conferência e as portas UDP para os fluxos de dados da mesma (por exemplo, áudio e vídeo). O *Session Announcement Protocol*, ou SAP (RFC 2974 [Handley, Perkins e Whelan, 2000]), descreve a maneira como isso é feito (os cabeçalhos de pacote e a freqüência com que esses anúncios são transmitidos por multicast para a infra-estrutura IP multicast), e o *Session Description Protocol*, ou SDP (RFC 2327 [Handley e Jacobson, 1998]), descreve o conteúdo desses anúncios (a maneira como os endereços multicast e os números de porta UDP são especificados). Um site que deseja anunciar uma sessão na infra-estrutura IP multicast envia periodicamente um pacote multicast que contém uma descrição da sessão a um grupo multicast e uma porta UDP bem-conhecidos. Os sites na infra-estrutura IP multicast executam um programa chamado `sdr` para receber esses anúncios. Esse programa faz várias coisas: ele não apenas recebe os anúncios de sessão, mas também fornece uma interface interativa com o usuário que exibe as informações e deixa o usuário enviar os anúncios.

Nesta seção, desenvolveremos um programa simples que recebe apenas esses anúncios de sessão para mostrar um exemplo de um programa receptor de multicast simples. Nosso objetivo é mostrar a simplicidade de um receptor de multicast, sem nos aprofundarmos nos detalhes dessa aplicação.

A Figura 21.14 mostra nosso programa `main` que recebe anúncios SAP/SDP periódicos.

Nome e porta bem-conhecidos

2–3 O endereço multicast atribuído a anúncios SAP é 224.2.127.254 e seu nome é `sap.mcast.net`. Todos os endereços multicast bem-conhecidos (consulte `http://www.iana.org/assignments/multicast-addresses`) aparecem no DNS sob a hierarquia `mcast.net`. A porta UDP bem-conhecida é 9875.

mysdr/main.c

```
 1 #include    "unp.h"

 2 #define SAP_NAME  "sap.mcast.net" /* nome de grupo-padrão e porta-padrão */
 3 #define SAP_PORT  "9875"

 4 void   loop(int, socklen_t);

 5 int
 6 main(int argc, char **argv)
 7 {
 8     int    sockfd;
 9     const int on = 1;
10     socklen_t salen;
11     struct sockaddr *sa;

12     if (argc == 1)
13         sockfd = Udp_client(SAP_NAME, SAP_PORT, (void **) &sa, &salen);
14     else if (argc == 4)
15         sockfd = Udp_client(argv[1], argv[2], (void **) &sa, &salen);
16     else
17         err_quit("usage: mysdr <mcast-addr> <port#> <interface-name>");

18     Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
19     Bind(sockfd, sa, salen);

20     Mcast_join(sockfd, sa, salen, (argc == 4) ? argv[3] : NULL, 0);

21     loop(sockfd, salen);         /* recebe e imprime */

22     exit(0);
23 }
```

mysdr/main.c

Figura 21.14 Programa `main` para receber anúncios SAP/SDP.

Criação do soquete UDP

12–17 Chamamos nossa função `udp_client` para pesquisar o nome e a porta e ela preenche a estrutura de endereço de soquete apropriada. Utilizamos os padrões se nenhum argumento de linha de comando estiver especificado; caso contrário, selecionamos o endereço multicast, a porta e o nome de interface a partir dos argumentos de linha de comando.

Utilização de `bind` para vincular-se à porta

18–19 Configuramos a opção de soquete `SO_REUSEADDR` para permitir que múltiplas instâncias desse programa executem em um host e utilizamos `bind` para vincular a porta ao soquete. Vinculando o endereço multicast ao soquete, evitamos que este receba quaisquer outros datagramas UDP que poderiam ser recebidos na porta. Não é obrigatório vincular esse endereço multicast, mas isso fornece uma filtragem pelo kernel dos pacotes em que não estamos interessados.

Associando-se a grupo de multicast

20 Chamamos nossa função `mcast_join` para fazer a associação ao grupo. Se o nome da interface foi especificado como um argumento de linha de comando, ele é passado para a nossa função; do contrário, deixamos que o kernel escolha a interface a que o grupo está associado.

21 Chamamos nossa função `loop`, mostrada na Figura 21.15, para ler e imprimir todos os anúncios.

mysdr/loop.c

```
 1 #include   "mysdr.h"

 2 void
 3 loop(int sockfd, socklen_t salen)
 4 {
 5     socklen_t len;
 6     ssize_t n;
 7     char   *p;
 8     struct sockaddr *sa;
 9     struct sap_packet {
10         uint32_t sap_header;
11         uint32_t sap_src;
12         char   sap_data[BUFFSIZE];
13     } buf;

14     sa = Malloc(salen);

15     for ( ; ; ) {
16         len = salen;
17         n = Recvfrom(sockfd, &buf, sizeof(buf) - 1, 0, sa, &len);
18         ((char *) &buf)[n] = 0; /* termina com nulo */
19         buf.sap_header = ntohl(buf.sap_header);

20         printf("From %s hash 0x%04x\n", Sock_ntop(sa, len),
21                buf.sap_header & SAP_HASH_MASK);
22         if (((buf.sap_header & SAP_VERSION_MASK) >> SAP_VERSION_SHIFT) > 1) {
23             err_msg("... version field not 1 (0x%08x)", buf.sap_header);
24             continue;
25         }
26         if (buf.sap_header & SAP_IPV6) {
27             err_msg("... IPv6");
28             continue;
29         }
30         if (buf.sap_header & (SAP_DELETE | SAP_ENCRYPTED | SAP_COMPRESSED)) {
31             err_msg("... can't parse this packet type (0x%08x)",
32                     buf.sap_header);
```

Figura 21.15 Loop que recebe e imprime os anúncios SAP/SDP (*continua*).

```
33            continue;
34        }
35        p = buf.sap_data + ((buf.sap_header & SAP_AUTHLEN_MASK)
36                            >> SAP_AUTHLEN_SHIFT);
37        if (strcmp(p, "application/sdp") == 0)
38            p += 16;
39        printf("%s\n", p);
40    }
41 }
```

mysdr/loop.c

Figura 21.15 Loop que recebe e imprime os anúncios SAP/SDP (*continuação*).

Formato de pacote

9–13 `sap_packet` descreve o pacote SDP: um cabeçalho SAP de 32 bits, seguido por um endereço de origem de 32 bits, seguido pelo anúncio real. O anúncio é constituído simplesmente de linhas de texto ISO 8859–1 e não deve exceder 1.024 bytes. Somente um anúncio de sessão é permitido em cada datagrama UDP.

Leitura do datagrama UDP, impressão do emissor e do conteúdo

15–21 `recvfrom` espera o próximo datagrama UDP destinado ao nosso soquete. Quando um datagrama chegar, posicionamos um byte nulo no final do buffer, corrigimos a ordem de byte do campo de cabeçalho e imprimimos a origem do pacote e o sinal numérico do SAP.

Verificação do cabeçalho SAP

22–34 Verificamos o cabeçalho SAP para ver se ele é um tipo que tratamos. Não tratamos pacotes SAP com endereços IPv6 no cabeçalho, nem pacotes compactados ou criptografados.

Localização do começo do anúncio e impressão

35–39 Pulamos quaisquer dados de autenticação que possam estar presentes, pulamos o tipo de conteúdo do pacote se estiver presente e então imprimimos o conteúdo do pacote.

A Figura 21.16 mostra uma saída típica do nosso programa.

```
freebsd % mysdr
From 128.223.83.33:1028 hash 0x0000
v=0
o=- 60345 0 IN IP4 128.223.214.198
s=UO Broadcast - NASA Videos - 25 Years of Progress
i=25 Years of Progress, parts 1-13. Broadcast with Cisco System's
  IP/TV using MPEG1 codec (6 hours 5 Minutes; repeats) More information
  about IP/TV and the client needed to view this program is available
  from http://videolab.uoregon.edu/download.html
u=http://videolab.uoregon.edu/
e=Hans Kuhn <multicast@lists.uoregon.edu>
p=Hans Kuhn <541/346-1758>
b=AS:1000
t=0 0
a=type:broadcast
a=tool:IP/TV Content Manager 3.2.24
a=x-iptv-file:1 name y:25yop1234567890123.mpg
m=video 63096 RTP/AVP 32 31 96
c=IN IP4 224.2.245.25/127
a=framerate:30
a=rtpmap:96 WBIH/90000
a=x-iptv-svr:video blaster2.uoregon.edu file 1 loop
m=audio 31954 RTP/AVP 14 96 0 3 5 97 98 99 100 101 102 10 11 103 104 105 106
```

Figura 21.16 Anúncio SAP/SDP típico (*continua*).

```
c=IN IP4 224.2.216.85/127
a=rtpmap:96 X-WAVE/8000
a=rtpmap:97 L8/8000/2
a=rtpmap:98 L8/8000
a=rtpmap:99 L8/22050/2
a=rtpmap:100 L8/22050
a=rtpmap:101 L8/11025/2
a=rtpmap:102 L8/11025
a=rtpmap:103 L16/22050/2
a=rtpmap:104 L16/22050
a=rtpmap:105 L16/11025/2
a=rtpmap:106 L16/11025
a=x-iptv-svr:audio blaster2.uoregon.edu file 1 loop
```

Figura 21.16 Anúncio SAP/SDP típico (*continuação*).

Esse anúncio descreve a cobertura da NASA quanto à infra-estrutura IP multicast em uma missão de lançamento espacial. A descrição da sessão SDP consiste em várias linhas na forma

tipo=*valor*

na qual o *tipo* sempre é um caractere que diferencia letras maiúsculas e minúsculas. O *valor* é uma string de texto estruturada que depende do *tipo*. Espaços não são permitidos no sinal de igual. `v=0` é a versão.

`o=` é a origem. – indica nenhum nome de usuário particular, `60345` é o ID de sessão, `0` é o número da versão desse anúncio, `IN` é o tipo de rede, `IP4` é o tipo de endereço e `128.223.214.198` é o endereço. As cinco tuplas que consistem no nome do usuário, ID de sessão, tipo de rede, tipo de endereço e endereço formam um identificador globalmente único para a sessão.

`s=` define o nome da sessão e `i=` são as informações sobre a sessão. Quebramos a linha do último a cada 80 caracteres. `u=` fornece um Uniform Resource Identifier (URI) para outras informações sobre a sessão e `e=` e `p=` fornecem o endereço do correio eletrônico e o número de telefone da pessoa responsável pela conferência.

`b=` fornece uma medida da largura de banda esperada para essa sessão. `t=` fornece o tempo inicial e final, os dois em unidades NTP, que são segundos desde 1º de janeiro de 1900, UTC. Nesse caso, essa sessão é "permanente"; sem nenhum tempo inicial ou final particular; portanto, o tempo inicial e o final são especificados como 0.

As linhas `a=` são atributos; cada um da sessão, se eles aparecerem antes de quaisquer linhas `m=`, ou da mídia, se aparecerem depois de uma linha `m=`.

As linhas `m=` são os anúncios sobre a mídia. A primeira dessas duas linhas especifica que o vídeo está na porta 63096 e seu formato é RTP, utilizando o Audio/Video Profile, ou AVP, com tipos de payload possíveis de 32, 31 ou 96 (que são MPEG, H.261 e WBIH, respectivamente). A linha `c=` que vem em seguida fornece as informações sobre a conexão que, nesse exemplo, especifica que é baseada em IP, utilizando o IPv4, com um endereço multicast de 224.2.245.25 e um TTL de 127. Embora essas linhas sejam separadas por uma barra, como o formato de prefixo CIDR, elas não são concebidas para representar um prefixo e uma máscara.

A próxima linha `m=` especifica que o áudio está na porta 31954 e poderia estar em qualquer um dos vários tipos de payload RTP/AVP, alguns dos quais são padrão e outros que são especificados abaixo utilizando atributos `a=rtpmap:`. A linha `c=` que vem em seguida fornece as informações sobre a conexão para o áudio que, nesse exemplo, especifica que é baseado em IP, utilizando o IPv4, com um endereço multicast de 224.2.216.85 e um TTL de 127.

21.10 Enviando e recebendo

O programa de anúncio de sessão da infra-estrutura IP multicast na seção anterior recebia apenas datagramas multicast. Agora desenvolveremos um programa simples que envia e recebe

datagramas multicast. Nosso programa consiste em duas partes. A primeira envia um datagrama multicast a um grupo específico a cada cinco segundos; o datagrama contém o hostname e o ID de processo do emissor. A segunda é um loop infinito que se associa ao grupo multicast para o qual a primeira parte está enviando e imprime cada datagrama recebido (contendo o hostname e o ID de processo do emissor). Isso permite iniciar o programa em múltiplos hosts em uma rede local e verificar facilmente qual host está recebendo os datagramas de quais emissores.

A Figura 21.17 mostra a função `main` do nosso programa.

mcast/main.c

```
 1 #include   "unp.h"
 2 void   recv_all(int, socklen_t);
 3 void   send_all(int, SA *, socklen_t);
 4 int
 5 main(int argc, char **argv)
 6 {
 7     int    sendfd, recvfd;
 8     const int on = 1;
 9     socklen_t salen;
10     struct sockaddr *sasend, *sarecv;
11     if (argc != 3)
12         err_quit("usage: sendrecv <IP-multicast-address> <port#>");
13     sendfd = Udp_client(argv[1], argv[2], (void **) &sasend, &salen);
14     recvfd = Socket(sasend->sa_family, SOCK_DGRAM, 0);
15     Setsockopt(recvfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
16     sarecv = Malloc(salen);
17     memcpy(sarecv, sasend, salen);
18     Bind(recvfd, sarecv, salen);
19     Mcast_join(recvfd, sasend, salen, NULL, 0);
20     Mcast_set_loop(sendfd, 0);
21     if (Fork() == 0)
22         recv_all(recvfd, salen);    /* filho -> recebe */
23     send_all(sendfd, sasend, salen);    /* pai -> envia */
24 }
```

mcast/main.c

Figura 21.17 Criação de soquetes, bifurcação com `fork` e início do emissor e do receptor.

Criamos dois soquetes, um para envio e outro para recebimento. Queremos que o soquete receptor utilize `bind` para vincular-se ao grupo e à porta multicast, digamos a 239.255.1.2 porta 8888. (Lembre-se de que poderíamos simplesmente utilizar `bind` para nos vincularmos ao endereço IP curinga e à porta 8888, mas vincular-se ao endereço multicast evita que o soquete receba quaisquer outros datagramas que poderiam chegar à porta 8888.) Em seguida, queremos que o soquete receptor associe-se ao grupo multicast. O soquete de envio enviará os datagramas para esse mesmo endereço e porta multicast, digamos a 239.255.1.2 porta 8888. Mas, se tentarmos utilizar um único soquete para envio e recebimento, o endereço do protocolo de origem será 239.255.1.2:8888 a partir do `bind` (utilizando a notação `netstat`) e o endereço do protocolo de destino para o `sendto` também será 239.255.1.2:8888. Entretanto, agora o endereço do protocolo de origem que está vinculado ao soquete torna-se o endereço IP de origem do datagrama UDP, a RFC 1122 (Braden, 1989) proíbe que um datagrama IP tenha um endereço IP de origem que seja um endereço multicast ou broadcast (consulte também o Exercício 21.2). Portanto, devemos criar dois soquetes: um para envio e outro para recebimento.

Criação de um soquete de envio

13 Nossa função `udp_client` cria o soquete de envio, processa os dois argumentos de linha de comando que especificam o endereço multicast e o número de porta. Essa função também retorna uma estrutura de endereço de soquete que está pronta para chamadas a `sendto` juntamente com seu comprimento de endereço de soquete.

Criação de um soquete receptor e utilização de `bind` para vincular-se ao endereço e à porta multicast

14–18 Criamos o soquete receptor utilizando a mesma família de endereços utilizada para o soquete de envio. Configuramos a opção de soquete `SO_REUSEADDR` para permitir que múltiplas instâncias desse programa executem ao mesmo tempo em um host. Em seguida, alocamos espaço para uma estrutura de endereço de soquete para esse soquete, copiamos seu conteúdo da estrutura de endereço de soquete de envio (cujo endereço e porta foram selecionados a partir dos argumentos de linha de comando) e utilizamos `bind` para vincular o endereço e a porta multicast para o soquete receptor.

Associação ao grupo multicast e desativação do loopback

19–20 Chamamos nossa função `mcast_join` para nos associarmos ao grupo multicast no soquete receptor e nossa função `mcast_set_loop` para desativar o recurso de loopback no soquete de envio. Para a associação, especificamos o nome da interface como um ponteiro nulo e o índice da interface como 0, instruindo o kernel a escolher a interface.

Bifurcação com `fork` e chamada às funções apropriadas

21–23 Bifurcamos com `fork`, o filho torna-se então o loop de recebimento; e o pai torna-se o loop de envio. Nossa função `send_all`, que envia um datagrama multicast a cada cinco segundos, é mostrada na Figura 21.18. A função `main` passa como argumentos o descritor de soquete, um ponteiro para uma estrutura de endereço de soquete que contém o destino e a porta multicast e o comprimento da estrutura.

Obtenção de um hostname e formação do conteúdo do datagrama

9–11 Obtemos o hostname a partir da função `uname`, construímos a linha de saída que contém o hostname e o ID de processo.

Envio do datagrama e então dormir

12–15 Enviamos um datagrama e então chamamos `sleep` por cinco segundos.

A função `recv_all`, que é o loop infinito de recebimento, é mostrada na Figura 21.19.

Alocação da estrutura de endereço de soquete

9 Uma estrutura de endereço de soquete é alocada para receber o endereço de protocolo do emissor para cada chamada a `recvfrom`.

Leitura e impressão dos datagramas

10–15 Cada datagrama é lido por `recvfrom`, terminado por caractere nulo e impresso.

mcast/send.c

```
1 #include   "unp.h"
2 #include   <sys/utsname.h>

3 #define SENDRATE  5      /* envia um datagrama a cada cinco segundos */
```

Figura 21.18 Envio de um datagrama multicast a cada cinco segundos (*continua*).

```
 4 void
 5 send_all(int sendfd, SA *sadest, socklen_t salen)
 6 {
 7     char   line[MAXLINE];        /* hostname e ID do processo */
 8     struct utsname myname;

 9     if(uname(&myname) < 0)
10         err_sys("uname error");;
11     snprintf(line, sizeof(line), "%s, %d\n", myname.nodename, getpid());

12     for ( ; ; ) {
13         Sendto(sendfd, line, strlen(line), 0, sadest, salen);

14         sleep(SENDRATE);
15     }
16 }
```

mcast/send.c

Figura 21.18 Envio de um datagrama multicast a cada cinco segundos (*continuação*).

mcast/recv.c

```
 1 #include   "unp.h"

 2 void
 3 recv_all(int recvfd, socklen_t salen)
 4 {
 5     int    n;
 6     char   line[MAXLINE + 1];
 7     socklen_t len;
 8     struct sockaddr *safrom;

 9     safrom = Malloc(salen);

10     for ( ; ; ) {
11         len = salen;
12         n = Recvfrom(recvfd, line, MAXLINE, 0, safrom, &len);

13         line[n] = 0;             /* termina com nulo */
14         printf("from %s: %s", Sock_ntop(safrom, len), line);
15     }
16 }
```

mcast/recv.c

Figura 21.19 Recebimento de todos os datagramas multicast para um grupo ao qual nos associamos.

Exemplo

Executamos esse programa nos nossos dois sistemas, freebsd4 e macosx. Vemos que cada sistema verifica os pacotes que o outro está enviando.

```
freebsd4 % sendrecv 239.255.1.2 8888
from 172.24.37.78:51297: macosx, 21891
from 172.24.37.78:51297: macosx, 21891
from 172.24.37.78:51297: macosx, 21891
from 172.24.37.78:51297: macosx, 21891

macosx % sendrecv 239.255.1.2 8888
from 172.24.37.94.1215: freebsd4, 55372
from 172.24.37.94.1215: freebsd4, 55372
from 172.24.37.94.1215: freebsd4, 55372
from 172.24.37.94.1215: freebsd4, 55372
```

21.11 Simple Network Time Protocol (SNTP)

O NTP é um protocolo sofisticado para sincronizar relógios dentro de uma WAN ou de uma rede local e, freqüentemente, pode alcançar a precisão de milissegundos. A RFC 1305 (Mills, 1992) descreve esse protocolo em detalhes e a RFC 2030 (Mills, 1996) descreve o SNTP, uma versão simplificada, mas compatível com o protocolo NTP, concebida para hosts que não precisam da complexidade de uma implementação NTP completa. É comum que alguns hosts em uma rede local sincronizem seus clocks pela Internet com outros hosts NTP e então redistribuam essa data/hora na rede local utilizando broadcast ou multicast.

Nesta seção, desenvolveremos um cliente SNTP que escuta broadcasts ou multicasts de NTP em todas as redes conectadas e então imprime a diferença de data/hora entre o pacote NTP e a hora atual do dia do host. Não tentamos ajustar a hora do dia, visto que isso requer privilégios de superusuário.

O arquivo `ntp.h`, mostrado na Figura 21.20, contém algumas definições básicas sobre o formato do pacote NTP.

ssntp/ntp.h

```
 1 #define JAN_1970  2208988800UL  /* 1970 - 1900 em segundos */

 2 struct l_fixedpt {              /* formato de ponto fixo de 64 bits */
 3     uint32_t int_part;
 4     uint32_t fraction;
 5 };

 6 struct s_fixedpt {              /* formato de ponto fixo de 32 bits */
 7     uint16_t int_part;
 8     uint16_t fraction;
 9 };

10 struct ntpdata {                /* cabeçalho NTP */
11     u_char status;
12     u_char stratum;
13     u_char ppoll;
14     int    precision:8;
15     struct s_fixedpt distance;
16     struct s_fixedpt dispersion;
17     uint32_t refid;
18     struct l_fixedpt reftime;
19     struct l_fixedpt org;
20     struct l_fixedpt rec;
21     struct l_fixedpt xmt;
22 };

23 #define VERSION_MASK     0x38
24 #define MODE_MASK        0x07

25 #define MODE_CLIENT      3
26 #define MODE_SERVER      4
27 #define MODE_BROADCAST   5
```

ssntp/ntp.h

Figura 21.20 Cabeçalho `ntp.h`: formato e definições do pacote NTP.

2-22 `l_fixedpt` define os valores de ponto fixo de 64 bits utilizados pelo NTP para registros de data/hora (*timestamps*) e `s_fixedpt` define os valores de ponto fixo de 32 bits que também são utilizados pelo NTP. A estrutura `ntpdata` é o formato do pacote NTP de 48 bytes.

A Figura 21.21 mostra a função `main`.

Obtenção do endereço multicast IP

12–14 Quando o programa é executado, o usuário deve especificar o endereço multicast a que se associar como o argumento de linha de comando. Com o IPv4, isso seria 224.0.1.1 ou o nome `ntp.mcast.net`. Com o IPv6, isso seria `ff05::101` para o NTP de escopo do site local. Nossa função `udp_client` aloca espaço para uma estrutura de endereço de soquete do tipo correto (IPv4 ou IPv6) e armazena o endereço multicast e a porta nessa estrutura. Se esse programa for executado em um host que não suporta multicast, um endereço IP qualquer pode ser especificado, uma vez que somente a família de endereços e a porta são utilizadas a partir dessa estrutura. Observe que nossa função `udp_client` não utiliza `bind` para vincular o endereço ao soquete; ela apenas cria o soquete e preenche a estrutura de endereço de soquete.

ssntp/main.c

```
 1 #include   "sntp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    sockfd;
 6     char   buf[MAXLINE];
 7     ssize_t n;
 8     socklen_t salen, len;
 9     struct ifi_info *ifi;
10     struct sockaddr *mcastsa, *wild, *from;
11     struct timeval now;

12     if (argc != 2)
13         err_quit("usage: ssntp <IPaddress>");

14     sockfd = Udp_client(argv[1], "ntp", (void **) &mcastsa, &salen);

15     wild = Malloc(salen);
16     memcpy(wild, mcastsa, salen);   /* copia família e porta */
17     sock_set_wild(wild, salen);
18     Bind(sockfd, wild, salen);   /* vincula curinga */

19 #ifdef MCAST
20         /* obtém a lista de interfaces e processa cada uma */
21     for (ifi = Get_ifi_info(mcastsa->sa_family, 1); ifi != NULL;
22          ifi = ifi->ifi_next) {
23         if (ifi->ifi_flags & IFF_MULTICAST) {
24             Mcast_join(sockfd, mcastsa, salen, ifi->ifi_name, 0);
25             printf("joined %s on %s\n",
26                    Sock_ntop(mcastsa, salen), ifi->ifi_name);
27         }
28     }
29 #endif

30     from = Malloc(salen);
31     for ( ; ; ) {
32         len = salen;
33         n = Recvfrom(sockfd, buf, sizeof(buf), 0, from, &len);
34         Gettimeofday(&now, NULL);
35         sntp_proc(buf, n, &now);
36     }
37 }
```

ssntp/main.c

Figura 21.21 Função `main`.

Vinculando o endereço curinga ao soquete

15–18 Alocamos espaço para uma outra estrutura de endereço de soquete e a preenchemos copiando a estrutura que foi preenchida por `udp_client`. Isso configura a família e a porta do endereço. Chamamos nossa função `sock_set_wild` para configurar o endereço IP como o curinga e então chamamos `bind`.

Obtendo a lista de interfaces

20–22 Nossa função `get_ifi_info` retorna as informações sobre todas as interfaces e endereços. A família de endereços que solicitamos é escolhida a partir da estrutura de endereço de soquete que foi preenchida por `udp_client` com base no argumento da linha de comando.

Ingressando em grupo de multicast

23–27 Chamamos nossa função `mcast_join` para associar o grupo multicast especificado pelo argumento da linha de comando para cada interface capaz de multicast. Todas essas associações são feitas em um dos soquetes que esse programa utiliza. Como dissemos anteriormente, costuma haver um limite de associações `IP_MAX_MEMBERSHIPS` (em geral 20) por soquete, mas poucos hosts multihomed têm esse número de interfaces.

Leitura e processamento de todos os pacotes NTP

30–36 Uma outra estrutura de endereço de soquete é alocada para armazenar o endereço retornado por `recvfrom`, o programa entra em um loop infinito lendo todos os pacotes NTP que o host recebe e chamando nossa função `sntp_proc` (descrita a seguir) para processar o pacote. Como o soquete foi vinculado ao endereço curinga e o grupo multicast foi associado a todas as interfaces capazes de multicast, o soquete deve receber qualquer pacote NTP via unicast, broadcast ou multicast que o host receba. Antes de chamar `sntp_proc`, chamamos `gettimeofday` para buscar a data/hora atual, pois `sntp_proc` calcula a diferença entre a data/hora no pacote e a data/hora atual.

Nossa função `sntp_proc`, mostrada na Figura 21.22, processa o pacote NTP real.

Validando o pacote

10–21 Primeiro, verificamos o tamanho do pacote e então imprimimos a versão, o modo e o estrato do servidor. Se o modo é `MODE_CLIENT`, o pacote é uma solicitação do cliente, não uma resposta do servidor, e o ignoramos.

Obtenção da data/hora de transmissão do pacote NTP

22–33 O campo no pacote NTP em que estamos interessados é `xmt`, o registro de data/hora de transmissão, que é o tempo no formato de ponto fixo de 64 bits com o qual o pacote foi enviado pelo servidor. Como os registros de data/hora do NTP contam segundos (começando em 1900) e os registros de data/hora do Unix também (começando em 1970), primeiro subtraímos `JAN_1970` (o número de segundos nesses 70 anos) da parte inteira.

A parte fracionária é um inteiro sem sinal de 32 bits entre 0 e 4.294.967.295, inclusive. Isso é copiado de um inteiro de 32 bits (`useci`) para uma variável de ponto flutuante de dupla precisão (`usecf`) e então dividido por 4.294.967.296 (2^{32}). O resultado é maior ou igual a 0,0 e menor que 1,0. Multiplicamos esse número por 1.000.000, o número de microssegundos em um segundo, armazenando o resultado como um inteiro sem sinal de 32 bits na variável `useci`. Esse é o número de microssegundos e estará entre 0 e 999.999 (veja o Exercício 21.5). Convertemos em microssegundos porque o registro de data hora do Unix retornado por `gettimeofday` é retornado como dois inteiros: o número de segundos a partir de 1º janeiro de 1970, UTC, juntamente com o número de microssegundos. Em seguida, calculamos e imprimimos a diferença entre a hora do dia do host e a hora do dia do servidor NTP, em microssegundos.

ssntp/sntp_proc.c

```
 1 #include   "sntp.h"

 2 void
 3 sntp_proc(char *buf, ssize_t n, struct timeval *nowptr)
 4 {
 5     int    version, mode;
 6     uint32_t nsec, useci;
 7     double usecf;
 8     struct timeval diff;
 9     struct ntpdata *ntp;

10     if (n < (ssize_t) sizeof(struct ntpdata)) {
11         printf("\npacket too small: %d bytes\n", n);
12         return;
13     }

14     ntp = (struct ntpdata *) buf;
15     version = (ntp->status & VERSION_MASK) >> 3;
16     mode = ntp->status & MODE_MASK;
17     printf("\nv%d, mode %d, strat %d, ", version, mode, ntp->stratum);
18     if (mode == MODE_CLIENT) {
19         printf("client\n");
20         return;
21     }

22     nsec = ntohl(ntp->xmt.int_part) - JAN_1970;
23     useci = ntohl(ntp->xmt.fraction);    /* fração inteira de 32 bits */
24     usecf = useci;              /* fração inteira -> double */
25     usecf /= 4294967296.0;      /* divide por 2**32 -> [0, 1.0) */
26     useci = usecf * 1000000.0;  /* fração -> partes por milhão */

27     diff.tv_sec = nowptr->tv_sec - nsec;
28     if ( (diff.tv_usec = nowptr->tv_usec - useci) < 0) {
29         diff.tv_usec += 1000000;
30         diff.tv_sec--;
31     }
32     useci = (diff.tv_sec * 1000000) + diff.tv_usec;   /* dif. em
                                                            microssegundos */
33     printf("clock difference = %d usec\n", useci);
34 }
```

ssntp/sntp_proc.c

Figura 21.22 Função `sntp_proc`: processa o pacote NTP.

Uma das coisas que o nosso programa não leva em consideração é o retardo de rede entre o servidor e o cliente. Mas supomos que os pacotes NTP normalmente são recebidos como um broadcast ou multicast em uma rede local, nesse caso, o retardo de rede deve ser de apenas alguns milissegundos.

Se executarmos esse programa no nosso host `macosx` com um servidor NTP no nosso host `freebsd4`, que está transmitindo pacotes NTP por multicast para a Ethernet a cada 64 segundos, teremos a seguinte saída:

```
macosx # ssntp 224.0.1.1
joined 224.0.1.1.123 on lo0
joined 224.0.1.1.123 on en1

v4, mode 5, strat 3, clock difference = 661 usec

v4, mode 5, strat 3, clock difference = -1789 usec

v4, mode 5, strat 3, clock difference = -2945 usec

v4, mode 5, strat 3, clock difference = -3689 usec
```

```
v4, mode 5, strat 3, clock difference = -5425 usec
v4, mode 5, strat 3, clock difference = -6700 usec
v4, mode 5, strat 3, clock difference = -8520 usec
```

Para executar nosso programa, primeiro terminamos o servidor NTP normal em execução nesse host a fim de que, quando nosso programa iniciar, a data/hora esteja bem próxima da data/hora do servidor. Verificamos que esse host perdeu 9181 microssegundos nos 384 segundos em que executamos o programa ou cerca de 2 segundos em 24 horas.

21.12 Resumo

Uma aplicação multicast inicia associando-se ao grupo multicast atribuído a ela. Isso instrui a camada IP a se associar ao grupo que, por sua vez, instrui a camada de enlace de dados a receber os quadros multicast enviados ao endereço multicast da camada de hardware correspondente. O multicast tira proveito da filtragem de hardware presente na maioria das placas de interface e, quanto melhor a filtragem, menor o número recebido de pacotes indesejáveis. Utilizar essa filtragem de hardware reduz a carga sobre todos os outros hosts que não estão participando na aplicação.

Transmitir por multicast em uma WAN requer roteadores capazes de multicast e um protocolo de roteamento multicast. Até que todos os roteadores na Internet tornem-se capazes de multicast, o multicast está disponível apenas para um subconjunto de usuários da Internet. Utilizamos o termo "infra-estrutura IP multicast" para descrever o conjunto de todos os sistemas capazes de multicast na Internet.

Nove opções de soquete fornecem a API para multicast:

- Associa um grupo multicast de qualquer origem em uma interface
- Sai de um grupo multicast
- Bloqueia uma origem em um grupo associado
- Desbloqueia uma origem bloqueada
- Associa um grupo multicast de origem específica em uma interface
- Sai de um grupo multicast de origem específica
- Configura a interface default para multicasts saintes
- Configura o TTL ou o limite de hop para multicasts saintes
- Ativa ou desativa o loopback de multicasts

Os primeiros seis são para recebimento e os últimos três são para envio. Há uma grande diferença entre as opções de soquete IPv4 e as opções de soquete IPv6, o que faz com que o código de multicast independente de protocolo torne-se abarrotado com `#ifdefs` muito rapidamente. Desenvolvemos nossas 12 funções próprias, todas iniciando com `mcast_`, que podem ajudar a escrever aplicações multicast que funcionam com o IPv4 ou o IPv6.

Exercícios

21.1 Construa o programa mostrado na Figura 20.9 e o execute especificando um endereço IP na linha de comando de 224.0.0.1. O que acontece?

21.2 Modifique o programa no exemplo anterior para utilizar `bind` para vincular 224.0.0.1 e a porta 0 ao seu soquete. Execute-o. Você tem permissão para utilizar `bind` para vincular um endereço multicast ao soquete? Se você tiver uma ferramenta como `tcpdump`, observe os pacotes na rede. Qual é o endereço IP de origem do datagrama que você envia?

21.3 Uma das maneiras de determinar quais hosts na sua sub-rede são capazes de multicast é emitir um `ping` ao grupo all-hosts: 224.0.0.1. Experimente isso.

21.4 Uma das maneiras de determinar se seu host está conectado à infra-estrutura IP multicast é executar o nosso programa da Seção 21.9, esperar alguns minutos e verificar se algum anúncio de sessão aparece. Experimente isso e veja se você recebe algum anúncio.

21.5 Examine os cálculos na Figura 21.22 quando a parte fracionária do registro de data/hora do NTP é 1.073.741.824 (um quarto de 2^{32}).

Refaça esses cálculos para a maior fração inteira possível (2^{32}-1).

21.6 Modifique a implementação de `mcast_set_if` para o IPv4 a fim de lembrar o nome de cada interface para a qual ele obtém o endereço IP com o objetivo de evitar chamar `ioctl` novamente para essa interface.

Capítulo 22

Soquetes UDP Avançados

22.1 Visão geral

Este capítulo apresentará vários tópicos que influenciam o comportamento das aplicações que utilizam soquetes UDP. O primeiro é determinar o endereço de destino de um datagrama UDP e a interface em que o datagrama foi recebido, pois um soquete vinculado a uma porta UDP faz com que o endereço curinga possa receber datagramas unicast, broadcast e multicast em qualquer interface.

O TCP é um protocolo de fluxo de bytes (*byte-stream*) que utiliza uma janela móvel; não há, portanto, nada como limite de registro ou permissão para que o emissor inunde o receptor com dados. Com o UDP, porém, cada operação de entrada corresponde a um datagrama UDP (um registro); surge, assim, um problema sobre o que acontece quando o datagrama recebido for maior que o buffer de entrada da aplicação.

O UDP é não-confiável, mas há aplicações em que faz sentido utilizá-lo em vez do TCP. Discutiremos o que acontece quando o UDP pode ser utilizado no lugar do TCP. Nessas aplicações UDP, devemos incluir alguns recursos para compensar a não-confiabilidade do UDP: um tempo-limite e retransmissão, para tratar datagramas perdidos, e números de seqüência, para fazer a correspondência entre respostas e solicitações. Desenvolveremos um conjunto de funções que pode ser chamado a partir das nossas aplicações UDP para tratar desses detalhes.

Se a implementação não suportar a opção de soquete `IP_RECVDSTADDR`, uma maneira de determinar o endereço IP de destino de um datagrama UDP é vincular todos os endereços de interface e utilizar `select`.

A maioria dos servidores UDP é iterativa, mas há aplicações que trocam múltiplos datagramas UDP entre o cliente e o servidor exigindo alguma forma de concorrência. O TFTP é o exemplo comum, e discutiremos como isso ocorre, com ou sem `inetd`.

O tópico final discute as informações "por pacote" que podem ser especificadas como dados auxiliares para um datagrama IPv6: o endereço IP de origem, a interface de envio, o limite de hops de saída e o endereço do próximo hop. Informações semelhantes podem ser retornadas com um datagrama IPv6 utilizando: o endereço IP de destino, a interface recebida e o limite de hops recebido.

22.2 Recebendo flags, endereço IP de destino e índice de interfaces

Historicamente, `sendmsg` e `recvmsg` eram utilizadas somente para passar descritores para soquetes de domínio Unix (Seção 15.7) e, mesmo assim, isso era raro. Mas o uso dessas duas funções está aumentando por duas razões:

1. O membro `msg_flags`, adicionado à estrutura `msghdr` no 4.3BSD Reno, retorna flags para a aplicação. Resumimos esses flags na Figura 14.7.
2. Dados auxiliares são utilizados para passar um número cada vez maior de informações entre a aplicação e o kernel. Veremos no Capítulo 27 que o IPv6 mantém essa tendência.

Como um exemplo de `recvmsg`, escreveremos uma função chamada `recvfrom_flags` semelhante a `recvfrom`, mas que também retorna o seguinte:

- O valor de `msg_flags` retornado
- O endereço de destino do datagrama recebido (da opção de soquete `IP_RECVDSTADDR`)
- O índice da interface em que o datagrama foi recebido (a opção de soquete `IP_RECVIF`)

Para retornar os dois últimos itens, definimos a seguinte estrutura no nosso cabeçalho `unp.h`:

```
struct unp_in_pktinfo {
   struct in_addr  ipi_addr;     /* endereço IPv4 de destino */
   int             ipi_ifindex;  /* índice da interface recebida */
};
```

Propositadamente, escolhemos o nome da estrutura e os nomes dos membros como semelhantes à estrutura `in6_pktinfo` IPv6 que retorna os mesmos dois itens para um soquete IPv6 (Seção 22.8). Nossa função `recvfrom_flags` receberá um ponteiro para uma estrutura `unp_in_pktinfo` como argumento e, se esse ponteiro for não-nulo, retornaremos a estrutura por meio dele.

Um problema de *design* dessa estrutura é o que retornar se as informações de `IP_RECVDSTADDR` não estiverem disponíveis (isto é, se a implementação não suportar a opção de soquete). O índice da interface é fácil de tratar porque um valor de 0 pode indicar que ele não é conhecido. Mas todos os valores de 32 bits para um endereço IP são válidos. O que escolhemos é retornar um valor de todos zeros (0.0.0.0) como o endereço de destino quando o valor real não está disponível. Embora esse seja um endereço IP válido, nunca é permitido como o endereço IP de destino (RFC 1122 [Braden, 1989]); ele é válido somente como o endereço IP de origem quando um host inicializa e ainda não conhece seu endereço IP.

> Infelizmente, kernels derivados do Berkeley aceitam datagramas IP destinados a 0.0.0.0 (páginas 218 e 219 do TCPv2). Esses são broadcasts obsoletos gerados por kernels derivados do 4.2BSD.

Agora, mostramos a primeira metade da nossa função `recvfrom_flags` na Figura 22.1. Essa função é concebida para ser utilizada com um soquete UDP.

advio/recvfromflags.c

```
 1 #include   "unp.h"
 2 #include   <sys/param.h>    /* macro ALIGN para a macro CMSG_NXTHDR() */

 3 ssize_t
 4 recvfrom_flags(int fd, void *ptr, size_t nbytes, int *flagsp,
 5                 SA *sa, socklen_t *salenptr, struct unp_in_pktinfo *pktp)
 6 {
 7     struct msghdr msg;
```

Figura 22.1 Função `recvfrom_flags`: chama `recvmsg` (*continua*).

```
 8     struct iovec iov[1];
 9     ssize_t n;

10 #ifdef HAVE_MSGHDR_MSG_CONTROL
11     struct cmsghdr *cmptr;
12     union {
13         struct cmsghdr cm;
14         char   control[CMSG_SPACE(sizeof(struct in_addr)) +
15                        CMSG_SPACE(sizeof(struct unp_in_pktinfo))];
16     } control_un;

17     msg.msg_control = control_un.control;
18     msg.msg_controllen = sizeof(control_un.control);
19     msg.msg_flags = 0;
20 #else
21     bzero(&msg, sizeof(msg)); /* certifica-se de que msg_accrightslen = 0 */
22 #endif

23     msg.msg_name = sa;
24     msg.msg_namelen = *salenptr;
25     iov[0].iov_base = ptr;
26     iov[0].iov_len = nbytes;
27     msg.msg_iov = iov;
28     msg.msg_iovlen = 1;

29     if ( (n = recvmsg(fd, &msg, *flagsp)) < 0)
30         return (n);

31     *salenptr = msg.msg_namelen;   /* passa os resultados de volta */
32     if (pktp)
33         bzero(pktp, sizeof(struct unp_in_pktinfo)); /* 0.0.0.0, interface = 0 */
```

advio/recvfromflags.c

Figura 22.1 Função `recvfrom_flags`: chama `recvmsg` (*continuação*).

Inclusão de arquivos

1–2 A macro `CMSG_NXTHDR` requer o cabeçalho `<sys/param.h>`.

Argumentos de funções

3–5 Os argumentos de função são semelhantes a `recvfrom`, exceto que o quarto argumento agora é um ponteiro para um flag de inteiros (de modo que possamos retornar os flags retornados por `recvmsg`) e o sétimo é novo: este argumento é um ponteiro para uma estrutura `unp_in_pktinfo` que conterá o endereço IPv4 de destino do datagrama recebido e o índice da interface em que o datagrama foi recebido.

Diferenças na implementação

10–22 Ao lidar com a estrutura `msghdr` e as diversas constantes `MSG_`*xxx*, encontramos muitas diferenças entre as várias implementações. Nossa maneira de tratar essas diferenças é utilizar o recurso de inclusão condicional do C (`#ifdef`). Se a implementação suportar o membro `msg_control`, o espaço é alocado para armazenar os valores retornados pelas opções de soquete `IP_RECVDSTADDR` e `IP_RECVIF` e os membros apropriados são inicializados.

Preenchimento da estrutura `msghdr` e chamada a `recvmsg`

23–33 Uma estrutura `msghdr` é preenchida e `recvmsg` é chamada. Os valores dos membros `msg_namelen` e `msg_flags` devem ser passados de volta para o chamador; eles são argumentos do tipo valor-resultado. Também inicializamos a estrutura `unp_in_pktinfo` do chamador, configurando o endereço IP como 0.0.0.0 e o índice da interface como 0.

A Figura 22.2 mostra a segunda metade da nossa função.

advio/recvfromflags.c

```
34 #ifndef HAVE_MSGHDR_MSG_CONTROL
35    *flagsp = 0;                  /* passa os resultados de volta */
36    return (n);
37 #else
38    *flagsp = msg.msg_flags;      /* passa os resultados de volta */
39    if (msg.msg_controllen < sizeof(struct cmsghdr) ||
40        (msg.msg_flags & MSG_CTRUNC) || pktp == NULL)
41        return (n);
42    for (cmptr = CMSG_FIRSTHDR(&msg); cmptr != NULL;
43        cmptr = CMSG_NXTHDR(&msg, cmptr)) {
44 #ifdef IP_RECVDSTADDR
45        if (cmptr->cmsg_level == IPPROTO_IP &&
46            cmptr->cmsg_type == IP_RECVDSTADDR) {
47            memcpy(&pktp->ipi_addr, CMSG_DATA(cmptr),
48                   sizeof(struct in_addr));
49            continue;
50        }
51 #endif
52 #ifdef IP_RECVIF
53        if (cmptr->cmsg_level == IPPROTO_IP && cmptr->cmsg_type == IP_RECVIF) {
54            struct sockaddr_dl *sdl;
55            sdl = (struct sockaddr_dl *) CMSG_DATA(cmptr);
56            pktp->ipi_ifindex = sdl->sdl_index;
57            continue;
58        }
59 #endif
60        err_quit("unknown ancillary data, len = %d, level = %d, type = %d",
61                 cmptr->cmsg_len, cmptr->cmsg_level, cmptr->cmsg_type);
62    }
63    return (n);
64 #endif /* HAVE_MSGHDR_MSG_CONTROL */
65 }
```

advio/recvfromflags.c

Figura 22.2 Função `recvfrom_flags`: retorna flags e endereço de destino.

34–37 Se a implementação não suportar o membro `msg_control`, simplesmente configuramos os flags retornados como 0 e retornamos. O restante da função trata das informações de `msg_control`.

Retorno se não houver nenhuma informação de controle

38–41 Retornamos o valor de `msg_flags` e então retornarmos ao chamador se: (i) não houver nenhuma informação de controle; (ii) se as informações de controle estiverem truncadas; ou (iii) se o chamador não quiser que uma estrutura `unp_in_pktinfo` retorne.

Processamento dos dados auxiliares

42–43 Processamos qualquer número de objetos dados auxiliares utilizando as macros `CMSG_FIRSTHDR` e `CMSG_NXTHDR`.

Processamento de `IP_RECVDSTADDR`

44–51 Se o endereço IP de destino for retornado como informação de controle (Figura 14.9), ele será retornado ao chamador.

Processamento de IP_RECVIF

52–59 Se o índice da interface recebida for retornado como informação de controle, ele será retornado ao chamador. A Figura 22.3 mostra o conteúdo do objeto dados auxiliares que retorna.

Lembre-se da estrutura de endereço de soquete de enlace de dados na Figura 18.1. Os dados retornados no objeto dados auxiliares são uma dessas estruturas, mas os três comprimentos são 0 (comprimento do nome, comprimento do endereço e comprimento do seletor). Portanto, não há necessidade de nenhum dos dados que se seguem a esses comprimentos; assim, o tamanho da estrutura deve ser de 8 bytes, não os 20 que mostramos na Figura 18.1. As informações que retornamos constituem o índice da interface.

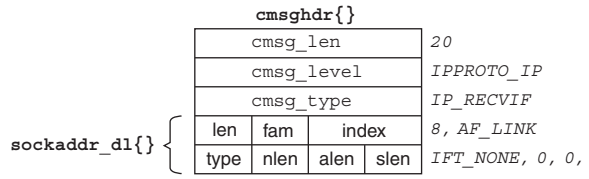


Figura 22.3 Objeto dados auxiliares retornado para IP_RECVIF.

Exemplo: Impressão do endereço IP de destino e o flag do datagrama truncado

Para testar nossa função, modificamos nossa função dg_echo (Figura 8.4) para chamar recvfrom_flags em vez de recvfrom. Mostramos essa nova versão da dg_echo na Figura 22.4.

Alterando MAXLINE

2–3 Removemos a definição existente de MAXLINE que ocorre no nosso cabeçalho unp.h e a redefinimos como 20. Fazemos isso para ver o que acontece quando recebemos um datagrama UDP maior que o buffer que passamos para a função de entrada (nesse caso, recvmsg).

Configurando as opções de soquete IP_RECVDSTADDR e IP_RECVIF

14–21 Se a opção de soquete IP_RECVDSTADDR estiver definida, ela está ativada. De maneira semelhante, a opção de soquete IP_RECVIF é ativada.

advio/dgechoaddr.c

```
 1 #include   "unpifi.h"
 2 #undef MAXLINE
 3 #define MAXLINE 20       /* para ver o truncamento do datagrama */
 4 void
 5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 6 {
 7     int    flags;
 8     const int on = 1;
 9     socklen_t len;
10     ssize_t n;
11     char   mesg[MAXLINE], str[INET6_ADDRSTRLEN], ifname[IFNAMSIZ];
12     struct in_addr in_zero;
13     struct unp_in_pktinfo pktinfo;
14 #ifdef IP_RECVDSTADDR
```

Figura 22.4 A função dg_echo que chama nossa função recvfrom_flags (*continua*).

```
15      if (setsockopt(sockfd, IPPROTO_IP, IP_RECVDSTADDR, &on, sizeof(on)) < 0)
16          err_ret("setsockopt of IP_RECVDSTADDR");
17  #endif
18  #ifdef IP_RECVIF
19      if (setsockopt(sockfd, IPPROTO_IP, IP_RECVIF, &on, sizeof(on)) < 0)
20          err_ret("setsockopt of IP_RECVIF");
21  #endif
22      bzero(&in_zero, sizeof(struct in_addr)); /* endereço IPv4 com tudo 0 */

23      for ( ; ; ) {
24          len = clilen;
25          flags = 0;
26          n = Recvfrom_flags(sockfd, mesg, MAXLINE, &flags,
27                             pcliaddr, &len, &pktinfo);
28          printf("%d-byte datagram from %s", n, Sock_ntop(pcliaddr, len));
29          if (memcmp(&pktinfo.ipi_addr, &in_zero, sizeof(in_zero)) != 0)
30              printf(", to %s", Inet_ntop(AF_INET, &pktinfo.ipi_addr,
31                                          str,sizeof(str)));
32          if (pktinfo.ipi_ifindex > 0)
33              printf(", recv i/f = %s",
34                     If_indextoname(pktinfo.ipi_ifindex, ifname));
35  #ifdef MSG_TRUNC
36          if (flags & MSG_TRUNC)
37              printf(" (datagram truncated)");
38  #endif
39  #ifdef MSG_CTRUNC
40          if (flags & MSG_CTRUNC)
41              printf(" (control info truncated)");
42  #endif
43  #ifdef MSG_BCAST
44          if (flags & MSG_BCAST)
45              printf(" (broadcast)");
46  #endif
47  #ifdef MSG_MCAST
48          if (flags & MSG_MCAST)
49              printf(" (multicast)");
50  #endif
51          printf("\n");

52          Sendto(sockfd, mesg, n, 0, pcliaddr, len);
53      }
54  }
```

advio/dgechoaddr.c

Figura 22.4 A função `dg_echo` que chama nossa função `recvfrom_flags` (*continuação*).

Leitura do datagrama, impressão do endereço IP de origem e a porta

24–28 O datagrama é lido chamando `recvfrom_flags`. O endereço IP de origem e a porta de resposta do servidor são convertidos em um formato de apresentação por `sock_ntop`.

Impressão do endereço IP de destino

29–31 Se o endereço IP retornado não for 0, ele será convertido em um formato de apresentação por `inet_ntop` e impresso.

Impressão do nome da interface recebida

32–34 Se o índice da interface retornado não for 0, seu nome será obtido chamando `if_indextoname` e impresso.

Teste de vários flags

35–51 Testamos quatro flags adicionais e imprimimos uma mensagem se algum estiver ativado.

22.3 Truncamento de datagrama

Nos sistemas derivados do BSD, quando um datagrama UDP chegar e for maior que o buffer da aplicação, `recvmsg` configura o flag `MSG_TRUNC` no membro `msg_flags` da estrutura `msghdr` (Figura 14.7). Todas as implementações derivadas do Berkeley que suportam a estrutura `msghdr` com o membro `msg_flags` fornecem essa notificação.

> Esse é um exemplo de um flag que deve retornar do kernel para o processo. Mencionamos na Seção 14.3 que um dos problemas de *design* com as funções `recv` e `recvfrom` é que seu argumento *flags* é um inteiro, o que permite que flags sejam passados do processo para o kernel, mas não vice-versa.

Infelizmente, nem todas as implementações tratam um datagrama UDP maior que o esperado dessa maneira. Há três possíveis cenários:

1. Descartar os bytes excessivos e retornar o flag `MSG_TRUNC` para a aplicação. Isso requer que a aplicação chame `recvmsg` para receber o flag.
2. Descartar os bytes excessivos, sem informar à aplicação.
3. Manter os bytes excessivos e retorná-los nas operações de leitura subseqüentes no soquete.

   > A especificação POSIX determina o primeiro tipo de comportamento: descartar os bytes excessivos e configurar o flag `MSG_TRUNC`. Distribuições iniciais do SVR4 exibiam o terceiro tipo de comportamento.

Como há variações na maneira como as implementações tratam os datagramas que são maiores que o buffer de recebimento da aplicação, é possível detectar o problema alocando um buffer para a aplicação que seja um byte maior que o maior datagrama que a aplicação deve receber. Se, alguma vez, for recebido um datagrama cujo comprimento seja igual a esse buffer, considere isso um erro.

22.4 Quando utilizar o UDP em vez do TCP

Nas Seções 2.3 e 2.4, descrevemos as diferenças mais importantes entre o UDP e o TCP. Como o TCP é confiável enquanto o UDP não é, surge a pergunta: quando devemos utilizar o UDP em vez do TCP e por quê? Primeiro, listamos as vantagens do UDP:

- Como mostramos na Figura 20.1, o UDP suporta broadcast e multicast. De fato, o UDP *deve* ser utilizado se a aplicação utilizar broadcast ou multicast. Discutimos esses dois modos de endereçamento nos Capítulos 20 e 21.
- O UDP não tem nenhuma configuração de conexão ou desconexão. Com referência à Figura 2.5, o UDP requer somente dois pacotes para trocar uma solicitação e uma resposta (assumindo que o tamanho de cada uma seja menor que o MTU mínimo entre os dois sistemas finais). O TCP requer cerca de 10 pacotes, assumindo que uma nova conexão TCP é estabelecida para cada troca de solicitação e resposta.

  Também importante na análise desse número de pacotes é o número de viagens de ida e volta dos pacotes requerido para obter a resposta. Isso se torna importante se a latência exceder a largura da banda, como descrito no Apêndice A do TCPv3. Esse texto mostra que o *tempo de transação* mínimo para uma solicitação e resposta do UDP é RTT + tempo de processamento do servidor (*server processing time* – SPT). Com o TCP, porém, se uma nova conexão TCP for utilizada para solicitação e resposta, o tempo mínimo de transação é $2 \times \text{RTT} + \text{SPT}$, um RTT maior que o tempo de UDP.

Deve ser óbvio, com relação ao segundo ponto, que, se uma conexão TCP for utilizada para múltiplas trocas de solicitações e respostas, o custo do estabelecimento da conexão e da desconexão é amortizado por todas as solicitações e respostas; e esse *design* é geralmente me-

lhor do que utilizar uma nova conexão para cada solicitação e resposta. Contudo, há aplicações que utilizam uma nova conexão TCP para cada solicitação e resposta (por exemplo, as versões mais antigas do HTTP) e há aplicações nas quais o cliente e o servidor trocam uma solicitação e resposta (por exemplo, o DNS) e, então, talvez não se comuniquem mais durante horas ou dias.

Agora, listaremos os recursos do TCP que não são fornecidos pelo UDP, o que significa que uma aplicação por si só deverá fornecê-los, se forem necessários para ela. Utilizamos o qualificador "necessário" porque nem todos os recursos são necessários para todas as aplicações. Por exemplo, segmentos descartados talvez não precisem ser retransmitidos a uma aplicação de áudio em tempo real, se o receptor puder interpolar os dados ausentes. Além disso, para transações simples de solicitação e resposta, o controle do fluxo em janelas talvez não seja necessário se as duas extremidades concordarem antecipadamente quanto ao tamanho da maior solicitação e resposta.

- Reconhecimentos positivos, retransmissão de pacotes perdidos, detecção de duplicados e seqüenciamento de pacotes reordenados pela rede – O TCP reconhece todos os dados, permitindo que pacotes perdidos sejam detectados. A implementação desses dois recursos requer que cada segmento de dados TCP contenha um número de seqüência que possa então ser reconhecido. Ela também requer que o TCP faça uma estimativa quanto ao valor do tempo-limite de uma retransmissão da conexão e que esse valor seja atualizado continuamente à medida que o tráfego de rede entre os dois sistemas finais muda.
- Controle de fluxo com janelas – Um TCP receptor informa ao emissor quanto espaço em buffer ele alocou para os dados de recebimento, e o emissor não pode exceder esse espaço. Isto é, a quantidade de dados não-reconhecidos no emissor nunca pode exceder à janela anunciada do receptor.
- Inicialização lenta e prevenção de congestionamento – Essa é uma forma de controle de fluxo imposta pelo emissor para determinar a capacidade atual da rede e tratar períodos de congestionamento. Todos os TCPs atuais devem suportar esses dois recursos e, por experiência, sabemos (antes de esses algoritmos terem sido implementados no final da década de 1980) que os protocolos que não "retrocedem" perante o congestionamento simplesmente o tornam pior (ver, por exemplo, Jacobson [1988]).

Em resumo, podemos dar as seguintes recomendações:

- O UDP *deve* ser utilizado para aplicações broadcast ou multicast. Qualquer forma de controle de erros desejada deve ser adicionada aos clientes e servidores, mas freqüentemente as aplicações utilizam broadcast ou multicast quando alguma quantidade (presumidamente pequena) de erros é aceitável (como pacotes perdidos para áudio ou vídeo). Aplicações multicast que exigem uma entrega confiável foram construídas (por exemplo, para transferência de arquivos multicast), mas devemos decidir se o ganho de desempenho ao utilizar o multicasting (enviar um pacote para *N* destinos *versus* enviar *N* cópias do pacote por *N* conexões do TCP) compensa a complexidade adicional requerida dentro da aplicação para fornecer comunicações confiáveis.
- O UDP *pode* ser utilizado para aplicações simples de solicitação e resposta, mas a detecção de erros deve então ser construída na aplicação. Minimamente, isso envolve reconhecimentos, tempos-limite e retransmissão. Com freqüência, o controle de fluxo não é um problema para solicitações e respostas com um tamanho razoável. Forneceremos um exemplo desses recursos em uma aplicação UDP na Seção 22.5. Aqui, os fatores a serem considerados são a freqüência da comunicação entre o cliente e o servidor (uma conexão TCP poderia permanecer ativa entre os dois?) e o volume de dados trocado (se múltiplos pacotes normalmente forem requeridos, o custo do estabelecimento da conexão TCP e de sua desconexão não se torna oneroso).

- O UDP *não deve* ser utilizado para transferência de dados em volume (por exemplo, transferência de arquivos). A razão é que o controle de fluxo em janelas, a prevenção de congestionamento e a inicialização lenta devem ser construídos na aplicação, juntamente com os recursos do parágrafo anterior, o que significa que estaremos reinventando o TCP dentro da aplicação. Devemos deixar os fornecedores focalizarem em um melhor desempenho do TCP e concentrar nossos esforços na própria aplicação.

Há exceções a essas regras, especialmente quanto às aplicações existentes. Por exemplo, o TFTP utiliza o UDP para transferência de dados em volume. O UDP foi escolhido pelo TFTP porque é mais simples de implementar que o TCP no código de bootstrap (por exemplo, 800 linhas de código C para UDP *versus* 4.500 linhas para TCP no TCPv2,) e porque o TFTP é utilizado somente para inicializar sistemas em uma rede local, não para transferência de dados em volume por redes remotas. Mas isso requer que o TFTP inclua seu próprio campo de número de seqüência para reconhecimentos, juntamente com um tempo-limite e capacidade de retransmissão.

O NFS é uma outra exceção à regra: ele também utiliza o UDP para transferência de dados em volume (embora algumas pessoas talvez sustentem que ele, na verdade, seja uma aplicação de solicitação e resposta, apesar de utilizar grandes solicitações e respostas). Isso é parcialmente verdade, pois, em meados da década de 1980, quando o NFS foi projetado, as implementações UDP eram mais rápidas que o TCP, e o NFS era utilizado somente em redes locais, em que a perda de pacotes costuma ser bem menor do que em redes remotas. Mas como o NFS começou a ser utilizado em WANs no início da década de 1990 e como as implementações TCP ultrapassaram o UDP em termos do desempenho na transferência de dados em volume, a versão 3 do NFS foi projetada para suportar TCP e a maioria dos fornecedores agora fornece o NFS sobre UDP e TCP. Um raciocínio semelhante (o UDP sendo mais rápido que o TCP em meados da década de 1980, juntamente com a predominância de LANs em relação a WANs) levou o precursor do pacote DCE RPC (o pacote Apollo NCS) a também escolher o UDP em vez do TCP, embora as implementações atuais suportem tanto o UDP como o TCP.

Talvez sejamos tentados a dizer que o uso do UDP está diminuindo se comparado com o TCP, com boas implementações TCP tão rápidas quanto a rede atual e com menos desenvolvedores de aplicação querendo reinventar o TCP dentro de suas aplicações UDP. Mas o aumento previsto para a próxima década nas aplicações de multimídia provavelmente será no uso do UDP, uma vez que multimídia normalmente implica transmissões por multicasting, o que requer o UDP.

22.5 Adicionando confiabilidade a uma aplicação UDP

Se formos utilizar o UDP para uma aplicação de solicitação e resposta, como mencionado na seção anterior, *deveremos* adicionar dois recursos ao nosso cliente:

1. Tempo-limite e retransmissão para tratar os datagramas que são descartados
2. Números de seqüência de modo que o cliente possa verificar se uma resposta é para a solicitação apropriada

Esses dois recursos fazem parte da maioria das aplicações UDP existentes que utilizam o paradigma simples de solicitação e resposta: por exemplo, resolvedores de DNS, agentes de SNMP, TFTP e RPC. Não estamos tentando utilizar o UDP para transferência de dados em volume; nossa intenção é a de que uma aplicação envie uma solicitação e espere uma resposta.

> Por definição, um datagrama é não-confiável; portanto, propositadamente não chamamos isso "serviço confiável de datagrama". De fato, o termo "datagrama confiável" é um oximoro. O que estamos mostrando é uma aplicação que acrescenta confiabilidade a um serviço de datagrama não-confiável (UDP).

Adicionar números de seqüência é simples. O cliente prefixa um número de seqüência a cada solicitação e o servidor deve ecoar esse número de volta ao cliente na resposta. Isso permite que cliente verifique se uma dada resposta é para a solicitação que foi emitida.

O método antiquado de tratar o tempo-limite e a retransmissão era enviar uma solicitação e esperar *n* segundos. Se nenhuma resposta tivesse sido recebida, devia-se retransmitir e esperar outros *n* segundos. Depois de algumas repetições, a aplicação desistia. Esse é um exemplo de um timer de retransmissão linear. (A Figura 6.8 do TCPv1 mostra um exemplo de um cliente TFTP que utiliza essa técnica. Vale observar que muitos clientes TFTP ainda utilizam esse método.)

O problema dessa técnica é que a quantidade de tempo requerida para um datagrama fazer uma viagem de ida e volta em uma rede varia de frações de um segundo em uma rede local a vários segundos em uma rede remota. Os fatores que afetam o RTT são distância, velocidade de rede e congestionamento. Além disso, o RTT entre um cliente e um servidor pode mudar rapidamente com o tempo, à medida que as condições de rede mudam. Devemos utilizar um tempo-limite e um algoritmo de retransmissão que leve em conta os RTTs reais que medimos em conjunto com as alterações no RTT ao longo do tempo. Muito trabalho tem sido focado nessa área, principalmente com relação ao TCP, mas as mesmas idéias são empregadas em qualquer aplicação de rede.

Queremos calcular o RTO para utilizar em cada pacote que enviamos. Para isso, medimos o RTT: o tempo real da viagem de ida e volta para um pacote. Cada vez que medimos um RTT, atualizamos dois estimadores estatísticos: *srtt* é o estimador RTT suavizado e *rttvar* é o estimador suavizado de desvio médio. O último é uma boa aproximação do desvio-padrão, porém é mais fácil de calcular, visto que não envolve uma raiz quadrada. Dados esses dois estimadores, o *RTO* a utilizar é o *srtt* mais quatro vezes o *rttvar*. Jacobson (1988) fornece todos os detalhes desses cálculos, que podemos resumir nas quatro equações a seguir:

$$delta = RTT\ medido - srtt$$
$$srtt \leftarrow srtt + g \times delta$$
$$rttvar \leftarrow rttvar + h(\,|delta| - rttvar\,)$$
$$RTO = srtt + 4 \times rttvar$$

delta é a diferença entre o RTT medido e o estimador RTT suavizado atual (*srtt*). *g* é o ganho aplicado ao estimador RTT e é igual a 1/8. *h* é o ganho aplicado ao estimador de desvio médio e é igual a 1/4.

> Os dois ganhos e o multiplicador 4 no cálculo do *RTO* são propositadamente potências de 2, de modo que possam ser calculados utilizando as operações de deslocamento em vez de multiplicação ou divisão. De fato, a implementação do kernel TCP (Seção 25.7 do TCPv2) normalmente é realizada com uma aritmética de ponto fixo para aumentar a velocidade de cálculo, mas, para simplificar, utilizamos cálculos de pontos flutuantes no nosso código que se segue.

Uma outra observação feita em Jacobson (1988) é que, quando o timer de retransmissão expira, um *backoff exponencial* deve ser utilizado para o próximo *RTO*. Por exemplo, se o nosso primeiro *RTO* for 2 segundos e a resposta não tiver sido recebida nesse tempo, o próximo *RTO* será então 4 segundos. Se ainda não houver nenhuma resposta, o *RTO* seguinte será 8 segundos, 16 e assim por diante.

Os algoritmos de Jacobson informam como calcular o *RTO* a cada vez que medimos um RTT e como aumentar o *RTO* quando retransmitimos. Mas há um problema quando temos de retransmitir um pacote e então receber uma resposta. Isso é denominado *problema da ambigüidade de retransmissão*. A Figura 22.5 mostra os três possíveis cenários em que nosso timer de retransmissão expira:

- A solicitação é perdida

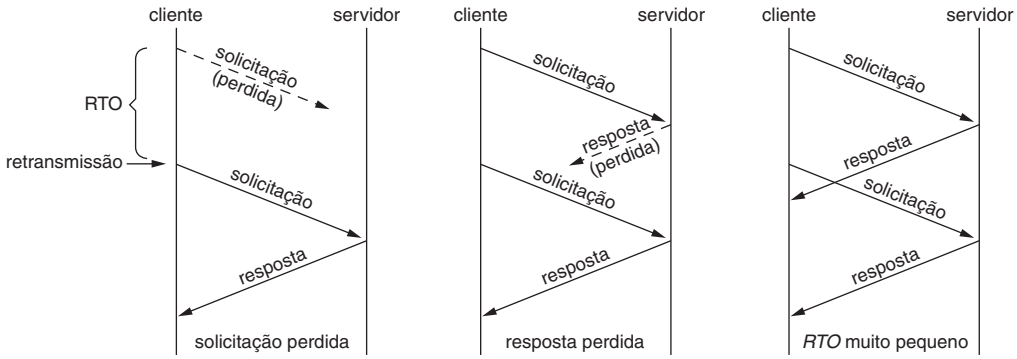


Figura 22.5 Os três cenários em que o timer de retransmissão expira.

- A resposta é perdida
- O *RTO* é muito pequeno

Quando o cliente recebe uma resposta a uma solicitação retransmitida, ele não pode determinar a solicitação a que corresponde a resposta. No exemplo à direita, a resposta corresponde à solicitação original, enquanto nos dois outros exemplos a resposta corresponde à solicitação retransmitida.

O *algoritmo de Karn* (Karn e Partridge, 1991) trata esse cenário com as seguintes regras, que se aplicam sempre que uma resposta é recebida para uma solicitação retransmitida:

- Se um RTT foi medido, não o utilize para atualizar os estimadores, uma vez que não sabemos a qual solicitação a resposta corresponde.
- Como essa resposta chegou antes de o nosso timer de retransmissão expirar, reutilize esse *RTO* para o próximo pacote. Somente quando recebemos uma resposta a uma solicitação que não foi retransmitida é que atualizaremos os estimadores RTT e recalcularemos o *RTO*.

Não é difícil levar o algoritmo de Karn em consideração ao codificar nossas funções RTT, mas, como veremos, há uma solução melhor e mais elegante. Essa solução é proveniente das extensões TCP para "conexões de alta velocidade" (redes com uma largura de banda alta, um RTT longo ou ambos), que estão documentadas na RFC 1323 (Jacobson, Braden e Borman, 1992). Além de prefixar um número de seqüência para cada solicitação, que o servidor deve ecoar de volta, também prefixamos um *registro de data/hora* que o servidor também deve ecoar. Toda vez que enviamos uma solicitação, armazenamos a data/hora atual no registro de data/hora. Quando uma resposta for recebida, calculamos o RTT desse pacote como o tempo atual menos o registro de data/hora que foi ecoado pelo servidor na resposta. Como cada solicitação transporta um registro de data/hora que é ecoado pelo servidor, podemos calcular o RTT de *todas* as respostas que recebemos. Não há absolutamente mais nenhuma ambigüidade. Além disso, uma vez que tudo o que servidor faz é ecoar o registro de data/hora do cliente, o cliente pode utilizar quaisquer unidades que desejar para os registros de data/hora, e não há absolutamente nenhum requisito para que ele e o servidor tenham relógios sincronizados.

Exemplo

Agora, agruparemos tudo isso em um exemplo. Iniciamos com a função `main` do nosso cliente UDP da Figura 8.7 e apenas alteramos o número da porta de `SERV_PORT` para 7 (o servidor de eco-padrão, Figura 2.18).

A Figura 22.6 é a função `dg_cli`. A única alteração na Figura 8.8 é substituir as chamadas a `sendto` e `recvfrom` por uma chamada a nossa nova função, `dg_send_recv`.

Antes de mostrar nossa função `dg_send_recv` e as funções RTT que ela chama, a Figura 22.7 mostra um esboço da maneira como adicionamos confiabilidade a um cliente UDP. Todas as funções que começam com `rtt_` serão mostradas brevemente.

rtt/dg_cli.c

```
 1 #include   "unp.h"
 2 ssize_t Dg_send_recv(int, const void *, size_t, void *, size_t,
 3                       const SA *, socklen_t);
 4 void
 5 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 6 {
 7     ssize_t n;
 8     char   sendline[MAXLINE], recvline[MAXLINE + 1];
 9     while (Fgets(sendline, MAXLINE, fp) != NULL) {
10         n = Dg_send_recv(sockfd, sendline, strlen(sendline),
11                          recvline, MAXLINE, pservaddr, servlen);
12         recvline[n] = 0;          /* termina com nulo */
13         Fputs(recvline, stdout);
14     }
15 }
```

rtt/dg_cli.c

Figura 22.6 Função `dg_cli` que chama a nossa função `dg_send_recv`.

```
static sigjmp_buf jmpbuf;

{
    . . .
    forma a solicitação

    signal(SIGALRM, sig_alrm);   /* estabelece o handler de sinal */
    rtt_newpack();               /* inicializa o contador rexmt como 0 */
sendagain:
    sendto();

    alarm(rtt_start());          /* configura o alarme como RTO segundos */
    if (sigsetjmp(jmpbuf, 1) != 0) {
        if (rtt_timeout())       /* RTO duplo, retransmitido o suficiente? */
            desiste
        goto sendagain;          /* retransmite */
    }
    do {
        recvfrom();
    } while (wrong sequence#);

    alarm(0);                    /* desativa o alarme */
    rtt_stop();                  /* calcula o RTT e atualiza os estimadores */
    processa a resposta
    . . .
}

void
sig_alrm(int signo)
{
    siglongjmp(jmpbuf, 1);
}
```

Figura 22.7 Esboço das funções RTT e quando são chamadas.

Quando uma resposta é recebida, mas o número da seqüência não é o esperado, chamamos `recvfrom` novamente, mas não retransmitimos a solicitação e não reiniciamos o timer de retransmissão que está em execução. Observe no exemplo mais à direita, na Figura 22.5, que a resposta final da solicitação retransmitida estará no buffer de recebimento do soquete na próxima vez que cliente enviar uma nova solicitação. Isso é adequado uma vez que o cliente irá ler essa resposta, observar que o número da seqüência não é o esperado, descartar a resposta e chamar `recvfrom` novamente.

Chamamos `sigsetjmp` e `siglongjmp` para evitar a condição de concorrência com o sinal `SIGALRM` que descrevemos na Seção 20.5.

A Figura 22.8 mostra a primeira metade da nossa função `dg_send_recv`.

1–5 Incluímos um novo cabeçalho, `unprtt.h`, mostrado na Figura 22.10, que define a estrutura `rtt_info` que mantém as informações sobre o RTT para um cliente. Definimos uma dessas estruturas e muitas outras variáveis.

Definindo as estruturas `msghdr` e a estrutura `hdr`

6–10 Queremos ocultar do chamador o fato de que prefixamos um número de seqüência e um registro de data/hora em cada pacote. A maneira mais fácil de fazer isso é utilizar `writev` e gravar nosso cabeçalho (a estrutura `hdr`), seguido pelos dados do chamador, como um único datagrama UDP. Lembre-se de que a saída para `writev` em um soquete de datagrama é um único datagrama. Isso é mais fácil do que forçar o chamador a alocar espaço na frente do seu buffer para nossa utilização e é também mais rápido do que copiar nosso cabeçalho e os dados do chamador para um buffer (que teríamos de alocar) para um único `sendto`. Mas, como estamos utilizando UDP e temos de especificar um endereço de destino, devemos utilizar a capacidade `iovec` do `sendmsg` e `recvmsg`, em vez de `sendto` e `recvfrom`. Lembre-se, da Seção 14.5, de que alguns sistemas têm uma estrutura `msghdr` mais recente para dados auxiliares, enquanto sistemas mais antigos continuam a ter os membros de direitos de acesso no final da estrutura. Para evitar complicar o código com `#ifdef`s para tratar essas diferenças, declaramos duas estruturas `msghdr` como `static`, forçando suas inicializações para todos os bits em zero pelo C e então simplesmente ignorando os membros não-utilizados no final das estruturas.

Inicializando na primeira vez em que somos chamados

20–24 Na primeira vez em que somos chamados, chamamos a função `rtt_init`.

Preenchendo as estruturas `msghdr`

25–41 Preenchemos as duas estruturas `msghdr` utilizadas para saída e entrada. Incrementamos o número de seqüência de envio desse pacote, mas não configuramos o registro de data/hora de envio até enviarmos o pacote (como ele talvez seja retransmitido e cada retransmissão precisa do registro de data/hora atual).

A segunda metade da função, juntamente com o handler de sinal `sig_alrm`, é mostrada na Figura 22.9.

rtt/dg_send_recv.c

```
1 #include   "unprtt.h"
2 #include   <setjmp.h>

3 #define RTT_DEBUG

4 static struct rtt_info rttinfo;
5 static int rttinit = 0;
6 static struct msghdr msgsend, msgrecv;   /* assumido init como 0 */
7 static struct hdr {
8     uint32_t seq;                 /* n° de seqüência */
```

Figura 22.8 Função `dg_send_recv`: primeira metade (*continua*).

```
 9     uint32_t ts;                 /* registro de data/hora ao enviar */
10 } sendhdr, recvhdr;

11 static void sig_alrm(int signo);
12 static sigjmp_buf jmpbuf;

13 ssize_t
14 dg_send_recv(int fd, const void *outbuff, size_t outbytes,
15              void *inbuff, size_t inbytes,
16              const SA *destaddr, socklen_t destlen)
17 {
18     ssize_t n;
19     struct iovec iovsend[2], iovrecv[2];

20     if (rttinit == 0) {
21         rtt_init(&rttinfo);      /* primeira vez que somos chamados */
22         rttinit = 1;
23         rtt_d_flag = 1;
24     }

25     sendhdr.seq++;
26     msgsend.msg_name = destaddr;
27     msgsend.msg_namelen = destlen;
28     msgsend.msg_iov = iovsend;
29     msgsend.msg_iovlen = 2;
30     iovsend[0].iov_base = &sendhdr;
31     iovsend[0].iov_len = sizeof(struct hdr);
32     iovsend[1].iov_base = outbuff;
33     iovsend[1].iov_len = outbytes;

34     msgrecv.msg_name = NULL;
35     msgrecv.msg_namelen = 0;
36     msgrecv.msg_iov = iovrecv;
37     msgrecv.msg_iovlen = 2;
38     iovrecv[0].iov_base = &recvhdr;
39     iovrecv[0].iov_len = sizeof(struct hdr);
40     iovrecv[1].iov_base = inbuff;
41     iovrecv[1].iov_len = inbytes;
```

rtt/dg_send_recv.c

Figura 22.8 Função `dg_send_recv`: primeira metade (*continuação*).

rtt/dg_send_recv.c

```
42     Signal(SIGALRM, sig_alrm);
43     rtt_newpack(&rttinfo);       /* inicialize para este pacote */

44   sendagain:
45     sendhdr.ts = rtt_ts(&rttinfo);
46     Sendmsg(fd, &msgsend, 0);

47     alarm(rtt_start(&rttinfo));  /* calcula o valor do tempo-limite
                                       e do timer de inicialização */

48     if (sigsetjmp(jmpbuf, 1) != 0) {
49         if (rtt_timeout(&rttinfo) < 0) {
50             err_msg("dg_send_recv: no response from server, giving up");
51             rttinit = 0;   /* reinicia no caso de sermos chamados de novo */
52             errno = ETIMEDOUT;
53             return (-1);
54         }
55         goto sendagain;
56     }

57     do {
58         n = Recvmsg(fd, &msgrecv, 0);
```

Figura 22.9 Função `dg_send_recv`: segunda metade (*continua*).

```
59      } while (n < sizeof(struct hdr) || recvhdr.seq != sendhdr.seq);

60      alarm(0);               /* interrompe o timer SIGALRM */
61          /* calcula e armazena os novos valores do estimador de RTT */
62      rtt_stop(&rttinfo, rtt_ts(&rttinfo) - recvhdr.ts);

63      return (n - sizeof(struct hdr));     /* tamanho de retorno do
                                                datagrama recebido */
64  }

65  static void
66  sig_alrm(int signo)
67  {
68      siglongjmp(jmpbuf, 1);
69  }
```

rtt/dg_send_recv.c

Figura 22.9 Função `dg_send_recv`: segunda metade (*continuação*).

Estabelecendo o handler de sinal

42–43 Um handler de sinal é estabelecido para `SIGALRM` e `rtt_newpack` configura o contador de retransmissão como 0.

Enviando o datagrama

45–47 O registro de data/hora atual é obtido por `rtt_ts` e armazenado na estrutura `hdr` prefixada com os dados do usuário. Um único datagrama UDP é enviado por `sendmsg`. O `rtt_start` retorna o número de segundos desse tempo-limite e o `SIGALRM` é agendado chamando `alarm`.

Estabelecendo o buffer de salto

48 Estabelecemos um buffer de salto para o nosso handler de sinal com `sigsetjmp`. Esperamos o próximo datagrama chegar chamando `recvmsg`. (Discutimos o uso de `sigsetjmp` e `siglongjmp` juntamente com `SIGALRM` na Figura 20.9.) Se o timer de alarme expirar, `sigsetjmp` retorna 1.

Tratando o tempo-limite

49–55 Quando um tempo-limite ocorre, `rtt_timeout` calcula o próximo *RTO* (o backoff exponencial) e retorna –1 se devemos desistir ou 0 se devemos retransmitir. Se desistirmos, configuramos `errno` como `ETIMEDOUT` e o retornamos ao chamador.

Chamando `recvmsg`, comparando os números de seqüência

57–59 Esperamos um datagrama chegar chamando `recvmsg`. Quando ele retorna, seu comprimento deve ter pelo menos o tamanho da nossa estrutura `hdr` e seu número de seqüência deve ser igual ao número da seqüência que foi enviado. Se uma das comparações for falsa, `recvmsg` é chamada novamente.

Desativando o alarme e atualizando os estimadores RTT

60–62 Quando a resposta esperada é recebida, o `alarm` pendente é desativado e `rtt_stop` atualiza os estimadores RTT. `rtt_ts` retorna o registro de data/hora atual e o registro de data/hora proveniente do datagrama recebido é subtraído dele, fornecendo o RTT.

Handler `SIGALRM`

65–69 `siglongjmp` é chamado, fazendo com que o `sigsetjmp` em `dg_send_recv` retorne 1.

Agora vemos as várias funções RTT que foram chamadas por `dg_send_recv`. A Figura 22.10 mostra o cabeçalho `unprtt.h`.

lib/unprtt.h

```
 1 #ifndef __unp_rtt_h
 2 #define __unp_rtt_h

 3 #include   "unp.h"

 4 struct rtt_info {
 5     float  rtt_rtt;        /* RTT mais recentemente medido, em segundos */
 6     float  rtt_srtt;       /* estimador de RTT suavizado, em segundos */
 7     float  rtt_rttvar;     /* desvio médio suavizado, em segundos */
 8     float  rtt_rto;        /* RTO atual para atualizar, em segundos */
 9     int    rtt_nrexmt;     /* nº de vezes retransmitido: 0, 1, 2, ... */
10     uint32_t rtt_base;     /* nº de segundos desde 1/1/1970 no início */
11 };

12 #define RTT_RXTMIN     2   /* mínimo valor do tempo-limite de retransmis-
                               são, em segundos */
13 #define RTT_RXTMAX    60   /* máximo valor do tempo-limite de retransmis-
                               são, em segundos */
14 #define RTT_MAXNREXMT  3   /* nº máximo de vezes a retransmitir */

15               /* protótipos de função */
16 void   rtt_debug(struct rtt_info *);
17 void   rtt_init(struct rtt_info *);
18 void   rtt_newpack(struct rtt_info *);
19 int    rtt_start(struct rtt_info *);
20 void   rtt_stop(struct rtt_info *, uint32_t);
21 int    rtt_timeout(struct rtt_info *);
22 uint32_t rtt_ts(struct rtt_info *);

23 extern int rtt_d_flag;    /* pode ser configurado como não-zero para
                              info de addl */

24 #endif /* __unp_rtt_h */
```

lib/unprtt.h

Figura 22.10 Cabeçalho `unprtt.h`.

Estrutura `rtt_info`

4–11 Essa estrutura contém as variáveis necessárias para calcular a data/hora dos pacotes entre um cliente e um servidor. As primeiras quatro variáveis são provenientes das equações dadas um pouco antes do começo desta seção.

12–14 Essas constantes definem os tempos-limite mínimo e máximo de retransmissão e o número máximo de vezes que retransmitimos.

A Figura 22.11 mostra uma macro e as nossas duas primeiras funções RTT.

lib/rtt.c

```
 1 #include   "unprtt.h"

 2 int     rtt_d_flag = 0;     /* flag debug; pode ser configurado por chamador */

 3 /*
 4  * Calcula o valor RTO com base nos estimadores atuais:
 5  *     RTT suavizado mais 4 vezes o desvio
 6  */
 7 #define RTT_RTOCALC(ptr) ((ptr)->rtt_srtt + (4.0 * (ptr)->rtt_rttvar))

 8 static float
 9 rtt_minmax(float rto)
10 {
11     if (rto < RTT_RXTMIN)
```

Figura 22.11 Macro `RTT_RTOCALC` e as funções `rtt_minmax` e `rtt_init` (*continua*).

```
12          rto = RTT_RXTMIN;
13      else if (rto > RTT_RXTMAX)
14          rto = RTT_RXTMAX;
15      return (rto);
16  }

17  void
18  rtt_init(struct rtt_info *ptr)
19  {
20      struct timeval tv;

21      Gettimeofday(&tv, NULL);
22      ptr->rtt_base = tv.tv_sec; /* n° de segundos desde 1/1/1970 no início */

23      ptr->rtt_rtt = 0;
24      ptr->rtt_srtt = 0;
25      ptr->rtt_rttvar = 0.75;
26      ptr->rtt_rto = rtt_minmax(RTT_RTOCALC(ptr));
27      /* primeiro RTO em (srtt + (4 * rttvar)) = 3 segundos */
28  }
```

lib/rtt.c

Figura 22.11 Macro `RTT_RTOCALC` e as funções `rtt_minmax` e `rtt_init` (*continuação*).

3–7 A macro `RTT_RTOCALC` calcula o *RTO* como o estimador RTT mais quatro vezes o estimador de desvio médio.

8–16 `rtt_minmax` assegura que o *RTO* esteja entre os limites superior e inferior no cabeçalho `unprtt.h`.

17–28 `rtt_init` é chamado por `dg_send_recv` na primeira vez que qualquer pacote é enviado. `gettimeofday` retorna a data e hora atual na mesma estrutura `timeval` que vimos com `select` (Seção 6.3). Salvamos somente o número atual de segundos a partir da "Época Unix", que é 00:00:00 de 1º de janeiro de 1970, UTC. O RTT medido é configurado como 0 e o RTT e os estimadores de desvio médio são configurados como 0 e 0,75, respectivamente, fornecendo um *RTO* inicial de 3 segundos.

A Figura 22.12 mostra as próximas três funções RTT.

lib/rtt.c

```
34  uint32_t
35  rtt_ts(struct rtt_info *ptr)
36  {
37      uint32_t ts;
38      struct timeval tv;

39      Gettimeofday(&tv, NULL);
40      ts = ((tv.tv_sec - ptr->rtt_base) * 1000) + (tv.tv_usec / 1000);
41      return (ts);
42  }

43  void
44  rtt_newpack(struct rtt_info *ptr)
45  {
46      ptr->rtt_nrexmt = 0;
47  }

48  int
49  rtt_start(struct rtt_info *ptr)
50  {
51      return ((int) (ptr->rtt_rto + 0.5));    /* arredonda float para int */
52          /* valor de retorno pode ser utilizado como: alarm(rtt_start(&foo)) */
53  }
```

lib/rtt.c

Figura 22.12 Funções `rtt_ts`, `rtt_newpack` e `rtt_start`.

34–42 `rtt_ts` retorna o registro de data/hora atual para que o chamador o armazene como um inteiro de 32 bits sem sinal no datagrama sendo enviado. Obtemos a data/hora atual a partir de `gettimeofday` e então subtraímos o número de segundos quando `rtt_init` for chamada (o valor salvo em `rtt_base`). Convertemos esse número em milissegundos e também convertemos o valor em microssegundos retornado por `gettimeofday` em milissegundos. O registro de data/hora torna-se então a soma desses dois valores em milissegundos.

A diferença entre as duas chamadas a `rtt_ts` é o número de milissegundos entre elas. Porém, armazenamos os registros de data/hora em milissegundos em um inteiro de 32 bits sem sinal em vez de em uma estrutura `timeval`.

43–47 `rtt_newpack` configura apenas o contador de retransmissão como 0. Essa função deve ser chamada sempre que um novo pacote é enviado pela primeira vez.

48–53 `rtt_start` retorna o *RTO* atual em segundos. O valor de retorno pode então ser utilizado como o argumento para `alarm`.

`rtt_stop`, mostrada na Figura 22.13, é chamada depois que uma resposta é recebida para atualizar o estimadores RTT e calcular o novo *RTO*.

lib/rtt.c

```
62 void
63 rtt_stop(struct rtt_info *ptr, uint32_t ms)
64 {
65     double delta;

66     ptr->rtt_rtt = ms / 1000.0; /* RTT medido em segundos */

67     /*
68      * Atualiza nossos estimadores de RTT e desvio médio de RTT.
69      * Veja o artigo de JACOBSON de 1988 em SIGCOMM, Apêndice A, para detalhes.
70      * Utilizamos ponto flutuante aqui para simplificar.
71      */

72     delta = ptr->rtt_rtt - ptr->rtt_srtt;
73     ptr->rtt_srtt += delta / 8; /* g = 1/8 */

74     if (delta < 0.0)
75         delta = -delta;          /* |delta| */

76     ptr->rtt_rttvar += (delta - ptr->rtt_rttvar) / 4;   /* h = 1/4 */

77     ptr->rtt_rto = rtt_minmax(RTT_RTOCALC(ptr));
78 }
```

lib/rtt.c

Figura 22.13 Função `rtt_stop`: atualiza os estimadores RTT e calcula um novo *RTO*.

62–78 O segundo argumento é o RTT medido, obtido pelo chamador subtraindo o registro de data/hora recebido na resposta do registro de data/hora atual (`rtt_ts`). As equações no começo desta seção são então aplicadas, armazenando os novos valores para `rtt_srtt`, `rtt_rttvar` e `rtt_rto`.

A função final, `rtt_timeout`, é mostrada na Figura 22.14. Essa função será chamada quando o timer de retransmissão expirar.

lib/rtt.c

```
83 int
84 rtt_timeout(struct rtt_info *ptr)
85 {
86     ptr->rtt_rto *= 2;           /* próximo RTO */

87     if (++ptr->rtt_nrexmt > RTT_MAXNREXMT)
88         return (-1);             /* tempo para desistir desse pacote */
89     return (0);
90 }
```

lib/rtt.c

Figura 22.14 Função `rtt_timeout`: aplica o backoff exponencial.

86 O atual *RTO* é dobrado: esse é o backoff exponencial.

87–89 Se atingirmos o número máximo de retransmissões, –1 é retornado para informar ao chamador que ele deve desistir; caso contrário, 0 é retornado.

Como exemplo, nosso cliente foi executado duas vezes em dois diferentes servidores de eco na Internet na manhã de um dia útil. Quinhentas linhas foram enviadas a cada servidor. Oito pacotes foram perdidos no primeiro servidor e 16 no segundo servidor. Entre os 16 perdidos no segundo servidor, um foi perdido duas vezes em seqüência: isto é, o pacote teve de ser retransmitido duas vezes antes de uma resposta ter sido recebida. Todos os outros pacotes perdidos foram tratados com uma única retransmissão. Poderíamos verificar se esses pacotes foram realmente perdidos imprimindo o número da seqüência de cada pacote recebido. Se um pacote estiver apenas atrasado e não perdido, depois da retransmissão, duas respostas serão recebidas pelo cliente: uma correspondendo à transmissão original que foi adiada e outra corresponde à retransmissão. Observe que, ao retransmitirmos, não somos capazes de dizer se foi a solicitação do cliente ou a resposta do servidor que foi descartada.

> Para a primeira edição deste livro, o autor escreveu um servidor UDP que descartava aleatoriamente pacotes para testar esse cliente. Isso não é mais necessário; tudo o que precisamos fazer é executar o cliente em um servidor na Internet e praticamente teremos garantias de que ocorrerá alguma perda de pacote!

22.6 Vinculando endereços de interface

Uma utilização comum da nossa função `get_ifi_info` é com aplicações UDP que precisam monitorar todas as interfaces em um host para saber quando um datagrama chega e em que interface ele chega. Isso permite ao programa receptor saber o endereço de destino do datagrama UDP, uma vez que esse endereço é o que determina o soquete para o qual um datagrama é entregue, mesmo que o host não suporte a opção de soquete `IP_RECVDSTADDR`.

> Lembre-se da nossa discussão no final da Seção 22.2. Se o host empregar o modelo comum de sistema final fraco, o endereço IP de destino pode diferir do endereço IP da interface receptora. Nesse caso, tudo o que podemos determinar é o endereço de destino do datagrama, que não precisa ser um endereço atribuído à interface receptora. Determinar a interface receptora requer a opção de soquete `IP_RECVIF` ou `IPV6_PKTINFO`.

A Figura 22.15 é a primeira parte de um exemplo simples dessa técnica com um servidor UDP que vincula todos os endereços unicast, endereços broadcast e, por fim, o endereço curinga.

Chamada a `get_ifi_info` para obter as informações sobre a interface

11–12 `get_ifi_info` obtém todos os endereços IPv4, incluindo aliases, para todas as interfaces. O programa faz então um loop por cada estrutura `ifi_info` retornada.

Criação de um soquete UDP e chamada a `bind` para vincular o endereço unicast

13–20 Um soquete UDP é criado e o endereço unicast é vinculado a ele. Também configuramos a opção de soquete `SO_REUSEADDR`, visto que estamos nos vinculando à mesma porta (`SERV_PORT`) para todos os endereços IP.

advio/udpserv03.c

```
 1 #include   "unpifi.h"
 2 void   mydg_echo(int, SA *, socklen_t, SA *);
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int    sockfd;
 7     const int on = 1;
 8     pid_t  pid;
 9     struct ifi_info *ifi, *ifihead;
10     struct sockaddr_in *sa, cliaddr, wildaddr;
11     for (ifihead = ifi = Get_ifi_info(AF_INET, 1);
12          ifi != NULL; ifi = ifi->ifi_next) {
13           /* vincula endereço unicast */
14        sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
15        Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
16        sa = (struct sockaddr_in *) ifi->ifi_addr;
17        sa->sin_family = AF_INET;
18        sa->sin_port = htons(SERV_PORT);
19        Bind(sockfd, (SA *) sa, sizeof(*sa));
20        printf("bound %s\n", Sock_ntop((SA *) sa, sizeof(*sa)));
21        if ( (pid = Fork()) == 0) { /* filho */
22           mydg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr), (SA *) sa);
23           exit(0);              /* nunca executado */
24        }
```

advio/udpserv03.c

Figura 22.15 Primeira parte do servidor UDP que vincula todos os endereços.

Nem todas as implementações exigem que essa opção de soquete seja configurada. Implementações derivadas do Berkeley, por exemplo, não exigem essa opção e permitem um novo `bind` de uma porta já vinculada se o novo endereço IP sendo vinculado: (i) não for o curinga e (ii) diferir de todos os endereços IP que já estão vinculados à porta.

Bifurcação (`fork`) do filho para esse endereço

21–24 Um filho é bifurcado com `fork` e a função `mydg_echo` é chamada para ele. Essa função espera que qualquer datagrama chegue nesse soquete e o ecoa de volta ao emissor.

A Figura 22.16 mostra a próxima parte da função `main`, que trata endereços de transmissão.

Vinculação do endereço broadcast

25–42 Se a interface suportar broadcast, um soquete UDP é criado e o endereço broadcast é vinculado a ele. Dessa vez, permitimos que `bind` falhe com um erro de `EADDRINUSE` porque, se uma interface tiver múltiplos endereços (aliases) na mesma sub-rede, cada um dos diferentes endereços unicast terá o mesmo endereço broadcast. Mostramos um exemplo disso na Figura 17.6. Nesse cenário, esperamos que somente o primeiro `bind` seja bem-sucedido.

advio/udpserv03.c

```
25        if (ifi->ifi_flags & IFF_BROADCAST) {
26               /* tenta vincular endereço broadcast */
27           sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
```

Figura 22.16 Segunda parte do servidor UDP que utiliza `bind` para vincular todos os endereços (*continua*).

```
28              Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
29              sa = (struct sockaddr_in *) ifi->ifi_brdaddr;
30              sa->sin_family = AF_INET;
31              sa->sin_port = htons(SERV_PORT);
32              if (bind(sockfd, (SA *) sa, sizeof(*sa)) < 0) {
33                  if (errno == EADDRINUSE) {
34                      printf("EADDRINUSE: %s\n",
35                             Sock_ntop((SA *) sa, sizeof(*sa)));
36                      Close(sockfd);
37                      continue;
38                  } else
39                      err_sys("bind error for %s",
40                              Sock_ntop((SA *) sa, sizeof(*sa)));
41              }
42              printf("bound %s\n", Sock_ntop((SA *) sa, sizeof(*sa)));
43              if ( (pid = Fork()) == 0) { /* filho */
44                  mydg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr),
45                            (SA*) sa);
46                  exit(0);           /* nunca executado */
47              }
48          }
49      }
```

advio/udpserv03.c

Figura 22.16 Segunda parte do servidor UDP que utiliza `bind` para vincular todos os endereços (*continuação*).

Bifurcação (`fork`) do filho

43–47 Um filho é bifurcado e chama a função `mydg_echo`.

A parte final da função `main` é mostrada na Figura 22.17. Esse código utiliza `bind` para vincular o endereço curinga para tratar quaisquer endereços de destino, exceto os unicast e broadcast que já vinculamos. Os únicos datagramas que devem chegar a esse soquete são aqueles destinados ao endereço broadcast limitado (255.255.255.255).

Criação de um soquete e vinculação do endereço curinga

50–62 Um soquete UDP é criado, a opção de soquete `SO_REUSEADDR` é configurada e o endereço IP curinga é vinculado. Um filho é gerado e chama a função `mydg_echo`.

Término da função `main`

63 A função `main` termina e o servidor continua a executar todos os filhos que foram gerados.

advio/udpserv03.c

```
50          /* vincula endereço curinga */
51      sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
52      Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
53      bzero(&wildaddr, sizeof(wildaddr));
54      wildaddr.sin_family = AF_INET;
55      wildaddr.sin_addr.s_addr = htonl(INADDR_ANY);
56      wildaddr.sin_port = htons(SERV_PORT);
57      Bind(sockfd, (SA *) &wildaddr, sizeof(wildaddr));
58      printf("bound %s\n", Sock_ntop((SA *) &wildaddr, sizeof(wildaddr)));
59      if ( (pid = Fork()) == 0) { /* filho */
```

Figura 22.17 Parte final do servidor UDP que utiliza `bind` para vincular todos os endereços (*continua*).

```
60          mydg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr), (SA *) sa);
61          exit(0);                    /* nunca executado */
62      }
63      exit(0);
64  }
```

advio/udpserv03.c

Figura 22.17 Parte final do servidor UDP que utiliza `bind` para vincular todos os endereços (*continuação*).

A função `mydg_echo`, executada por todos os filhos, é mostrada na Figura 22.18.

advio/udpserv03.c

```
65 void
66 mydg_echo(int sockfd, SA *pcliaddr, socklen_t clilen, SA *myaddr)
67 {
68     int    n;
69     char   mesg[MAXLINE];
70     socklen_t len;

71     for ( ; ; ) {
72         len = clilen;
73         n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
74         printf("child %d, datagram from %s", getpid(),
75                Sock_ntop(pcliaddr, len));
76         printf(", to %s\n", Sock_ntop(myaddr, clilen));

77         Sendto(sockfd, mesg, n, 0, pcliaddr, len);
78     }
79 }
```

advio/udpserv03.c

Figura 22.18 Função `mydg_echo`.

Novo argumento

65–66 O quarto argumento para essa função é o endereço IP que foi vinculado ao soquete. Esse soquete deve receber somente datagramas destinados a esse endereço IP. Se o endereço IP for o curinga, o soquete deverá então receber somente os datagramas que não foram correspondidos por algum outro soquete vinculado à mesma porta.

Leitura do datagrama e eco da resposta

71–78 O datagrama é lido com `recvfrom` e enviado de volta ao cliente com `sendto`.

Essa função também imprime o endereço IP do cliente e o endereço IP que foi vinculado ao soquete.

Agora, executamos esse programa no nosso host `solaris` depois de estabelecer um endereço de alias à interface Ethernet `hme0`. O endereço de alias é o host número de 200 em 10.0.0/24.

```
solaris % udpserv03
bound 127.0.0.1:9877                 interface de loopback
bound 10.0.0.200:9877                endereço unicast da interface hme0:1
bound 10.0.0.255:9877                endereço broadcast da interface hme0:1
bound 192.168.1.20:9877              endereço unicast da interface hme0
bound 192.168.1.255:9877             endereço broadcast da interface hme0
bound 0.0.0.0.9877                   curinga
```

Podemos verificar que todos esses soquetes estão vinculados ao endereço IP e à porta indicada utilizando `netstat`.

```
solaris % netstat -na | grep 9877
127.0.0.1.9877                         Idle
10.0.0.200.9877                        Idle
      *.9877                           Idle
192.129.100.100.9877                   Idle
      *.9877                           Idle
      *.9877                           Idle
```

Devemos observar que o nosso *design* de um processo-filho por soquete visa à simplicidade, outros *designs* são possíveis. Por exemplo, para reduzir o número de processos, o próprio programa poderia gerenciar todos os descritores utilizando `select`, nunca chamando `fork`. O problema desse *design* é o aumento da complexidade do código. Embora seja fácil utilizar `select` para todos os descritores, teríamos de manter algum tipo de mapeamento de cada descritor para seu endereço IP vinculado (provavelmente um array de estruturas), de modo que seja possível imprimir o endereço IP de destino quando um datagrama é lido a partir de um soquete. Costuma ser mais simples utilizar um único processo ou um único thread para uma operação ou descritor em vez de fazer com que um único processo multiplexe diferentes operações ou descritores.

22.7 Servidores UDP concorrentes

A maioria dos servidores UDP é iterativa: o servidor espera uma solicitação do cliente, lê a solicitação, processa-a, envia a resposta de volta e então espera a próxima solicitação. Mas, quando o processamento da solicitação leva muito tempo, alguma forma de concorrência é desejável.

A definição de "muito tempo" é tudo o que for considerado como muito tempo de espera para um outro cliente enquanto o cliente atual está sendo atendido. Por exemplo, se duas solicitações de cliente chegarem dentro de 10 milisegundos uma da outra e se levar uma média de 5 segundos para atender um cliente, o segundo cliente terá então de esperar aproximadamente 10 segundos pela sua resposta, em vez de cerca de 5 segundos se a solicitação fosse tratada logo que chegasse.

Com o TCP, é simples bifurcar um novo filho com `fork` (ou criar um novo thread, como veremos no Capítulo 26) e deixar que ele trate o novo cliente. O que simplifica essa concorrência de servidor quando o TCP é utilizado é que cada conexão de cliente é única: o par de soquetes TCP é único para cada conexão. Mas, com o UDP, devemos lidar com dois tipos de servidores:

1. Um servidor UDP simples que lê a solicitação do cliente, envia uma resposta e então conclui a operação com o cliente. Nesse cenário, o servidor que lê a solicitação do cliente pode bifurcar um filho com `fork` e deixar que ele trate a solicitação. A "solicitação", isto é, o conteúdo do datagrama e da estrutura do endereço de soquete que contém o endereço de protocolo do cliente, é passado para o filho na sua imagem de memória proveniente de `fork`. O filho envia então sua resposta diretamente ao cliente.
2. Um servidor UDP que troca múltiplos datagramas com o cliente. O problema é que o único número de porta que o cliente conhece para o servidor é sua porta bem-conhecida. O cliente envia o primeiro datagrama da sua solicitação a essa porta, mas como o servidor distingue entre os datagramas subseqüentes provenientes desse cliente e as novas solicitações? A solução típica é fazer com que o servidor crie um novo soquete para cada cliente, utilize `bind` para vincular uma porta efêmera a esse soquete e use esse soquete para todas as suas respostas. Isso exige que o cliente examine o número da porta da primeira resposta do servidor e envie os datagramas subseqüentes dessa solicitação a essa porta.

Um exemplo do segundo tipo de servidor UDP é o TFTP. Transferir um arquivo utilizando TFTP normalmente requer muitos datagramas (centenas ou milhares, dependendo do tamanho do arquivo), porque o protocolo envia somente 512 bytes por datagrama. O cliente envia um datagrama à porta bem-conhecida do servidor (69), especificando o arquivo a enviar ou receber. O servidor lê a solicitação, mas envia sua resposta a partir de um outro soquete que ele cria e vincula a uma porta efêmera. Todos os datagramas subseqüentes entre o cliente e o servidor para esse arquivo utilizam o novo soquete. Isso permite ao servidor TFTP principal continuar a tratar outras solicitações do cliente que chegam à porta 69, enquanto essa transferência de arquivos acontece (talvez em segundos ou mesmo minutos).

Se assumirmos um servidor TFTP independente (isto é, não invocado por `inetd`), teremos o cenário mostrado na Figura 22.19. Assumimos que a porta efêmera vinculada pelo filho ao seu novo soquete seja 2134.

Se `inetd` for utilizado, o cenário envolve mais um passo. Lembre-se, da Figura 13.6, de que a maioria dos servidores UDP especifica o *wait-flag* como `wait`. Na nossa descrição seguindo a Figura 13.10, dissemos que isso faz com que o `inetd` pare de selecionar no soquete até que seu filho termine, permitindo ao filho ler o datagrama que chegou ao soquete. A Figura 22.20 mostra os passos envolvidos.

O servidor TFTP que é o filho de `inetd` chama `recvfrom` e lê a solicitação do cliente. Ele então bifurca um filho próprio com `fork` e esse filho irá processar a solicitação do cliente. O servidor TFTP então chama `exit`, enviando `SIGCHLD` ao `inetd`, que informa `inetd` a chamar novamente `select` no soquete vinculado à porta 69 UDP.

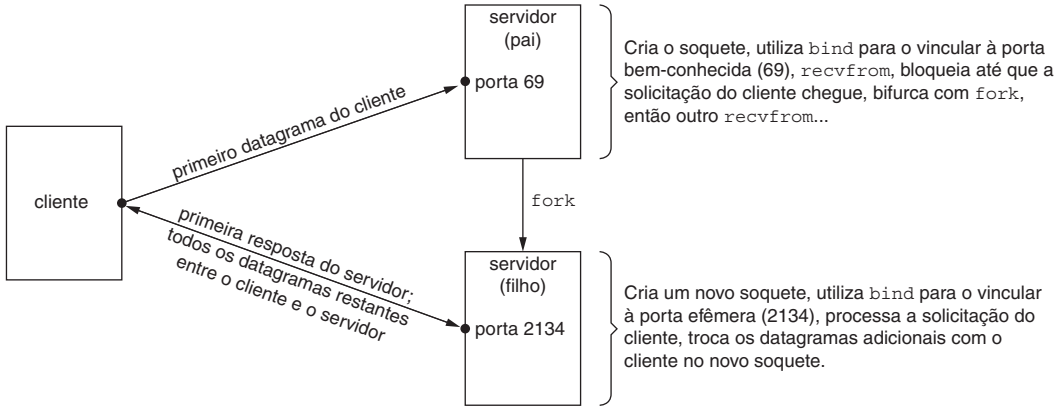


Figura 22.19 Processos que fazem parte de um servidor UDP concorrente e independente.

22.8 Informações do pacote IPv6

O IPv6 permite a uma aplicação especificar até cinco informações para um datagrama sainte:

1. Endereço IPv6 de origem
2. Índice da interface de saída
3. Limite de hops de saída
4. Endereço do próximo hop
5. Classe do tráfego de saída

Essas informações são enviadas como dados auxiliares com `sendmsg`. Valores "persistentes" podem ser configurados para o soquete, de modo que sejam aplicados a cada pacote

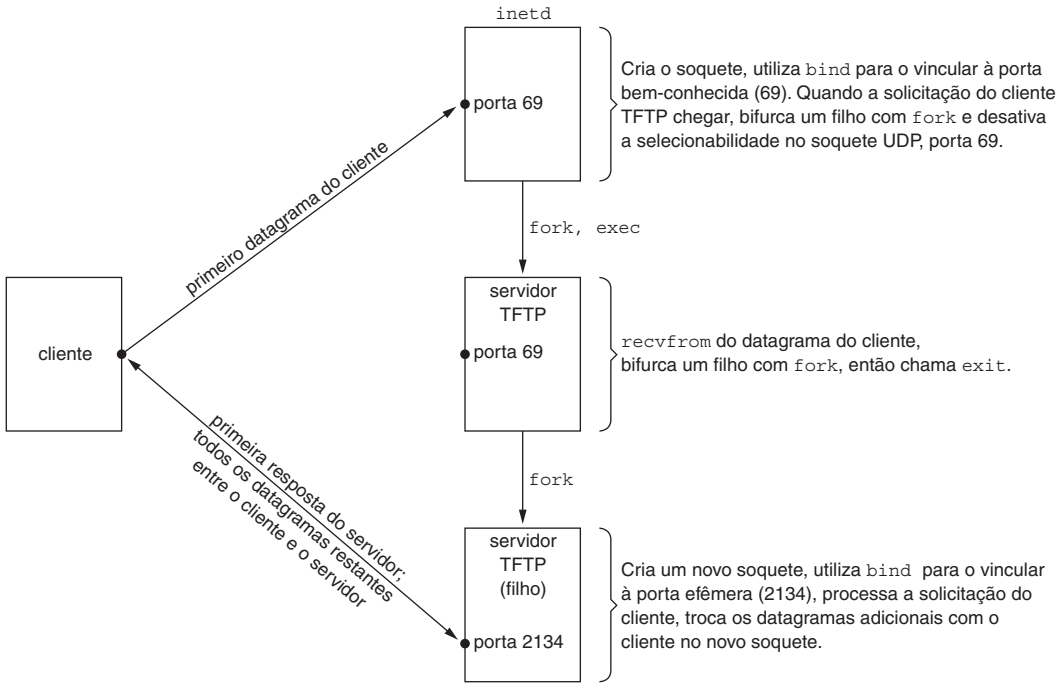


Figura 22.20 O servidor UDP concorrente invocado por `inetd`.

enviado, como descrito na Seção 27.7. Quatro informações semelhantes podem ser retornadas a um pacote recebido, as quais retornam como dados auxiliares com `recvmsg`:

1. Endereço IPv6 de destino
2. Índice da interface de entrada
3. Limite de hops entrante
4. Classe do tráfego entrante

A Figura 22.21 resume o conteúdo dos dados auxiliares, que discutiremos mais adiante.

Uma estrutura `in6_pktinfo` contém o endereço IPv6 de origem e o índice da interface de saída para um datagrama sainte ou o endereço IPv6 de destino e o índice da interface de entrada para um datagrama recebido.

```
struct in6_pktinfo {
   struct in6_addr ipi6_addr;      /* endereço IPv6 de origem/destino */
   int             ipi6_ifindex;   /* índice da interface de envio/
                                      recebimento */
};
```

Essa estrutura é definida incluindo o cabeçalho `<netinet/in.h>`. Na estrutura `cmsghdr` que contém esses dados auxiliares, o membro `cmsg_level` será `IPPROTO_IPV6`, o membro `cmsg_type` será `IPV6_PKTINFO` e o primeiro byte de dados será o primeiro byte da estrutura `in6_pktinfo`. No exemplo na Figura 22.21, supomos não haver preenchimento entre a estrutura `cmsghdr` e os dados, e 4 bytes para um inteiro.

Para especificar essas informações para um dado pacote, especifique apenas as informações de controle como dados auxiliares para `sendmsg`. Para especificar essas informações para todos os pacotes enviados em um soquete, configure a opção de soquete `IPV6_PKTIN-`

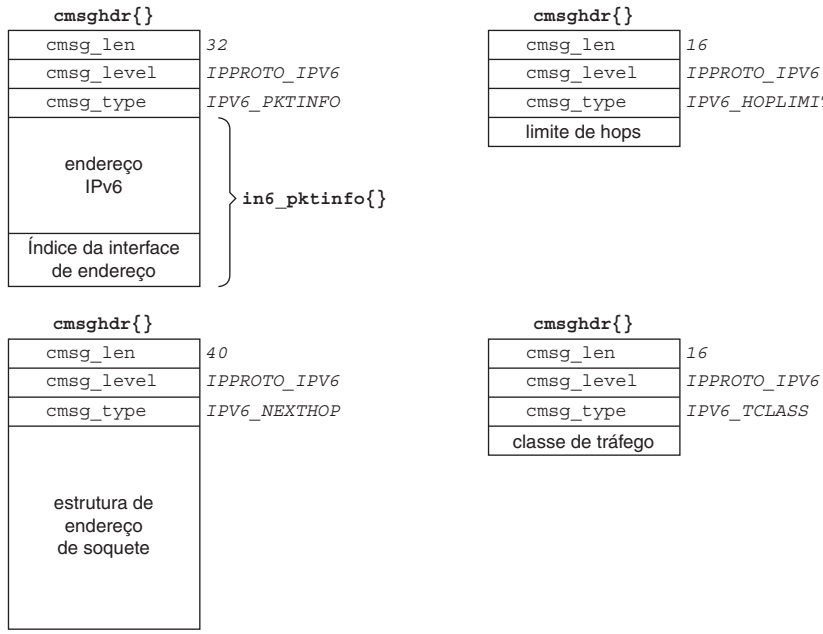


Figura 22.21 Dados auxiliares para informações do pacote IPv6.

FO com o `in6_pktinfo` como o valor de opção da maneira descrita na Seção 27.7. Essas informações retornam como dados auxiliares por `recvmsg` somente se a aplicação tiver a opção de soquete `IPV6_RECVPKTINFO` ativada.

Interface de saída e entrada

Interfaces em um nó IPv6 são identificadas por inteiros positivos, como discutido na Seção 18.6. Lembre-se de que não é atribuído um índice de 0 a nenhuma interface. Ao especificar a interface de saída, se o valor `ipi6_ifindex` for 0, o kernel escolherá a interface de saída. Se a aplicação especificar uma interface de saída para um pacote de multicast, a interface especificada pelos dados auxiliares sobrescreverá qualquer interface especificada pela opção de soquete `IPV6_MULTICAST_IF` apenas para esse datagrama.

Endereços IPv6 de origem e destino

O endereço IPv6 de origem normalmente é especificado chamando `bind`. Fornecer o endereço de origem junto com os dados talvez requeira menos overhead. Essa opção também permite que um servidor garanta que o endereço de origem da sua resposta seja igual ao endereço de destino da solicitação do cliente, um recurso que alguns clientes requerem e que é mais difícil de realizar com o IPv4 (Exercício 22.4).

Ao especificar o endereço IPv6 de origem como dados auxiliares, se o membro `ipi6_addr` da estrutura `in6_pktinfo` for `IN6ADDR_ANY_INIT`, então: (i) se um endereço estiver atualmente vinculado ao soquete, ele será utilizado como o endereço de origem; ou (ii) se nenhum endereço estiver atualmente vinculado ao soquete, o kernel escolherá o endereço de origem. Se o membro `ipi6_addr` não for o endereço especificado, mas o soquete já vinculou um endereço de origem, o valor `ipi6_addr` irá sobrescrever o endereço de origem já vinculado apenas para essa operação de saída. O kernel verificará se o endereço de origem solicitado é de fato um endereço unicast atribuído ao nó.

Quando a estrutura `in6_pktinfo` é retornada como dados auxiliares por `recvmsg`, o membro `ipi6_addr` contém o endereço IPv6 de destino proveniente do pacote recebido. Isso é semelhante em conceito à opção de soquete `IP_RECVDSTADDR` do IPv4.

Especificando e recebendo o limite de hops

O limite de hops de saída normalmente é especificado com a opção de soquete `IPV6_UNICAST_HOPS` para datagramas de unicast (Seção 7.8) ou com a opção de soquete `IPV6_MULTICAST_HOPS` para datagramas multicast (Seção 21.6). Especificar o limite de hops como dados auxiliares permite sobrescrever o padrão do kernel ou um valor previamente especificado, para um destino unicast ou multicast, para uma única operação de saída. Retornar o limite de hops recebido é útil para programas como o `traceroute` e para uma classe de aplicações IPv6 que precisa verificar se o limite de hops recebido é 255 (por exemplo, se o pacote não foi encaminhado).

O limite de hops recebido é retornado como dados auxiliares por `recvmsg` somente se a aplicação ativou a opção de soquete `IPV6_RECVHOPLIMIT`. Na estrutura `cmsghdr` que contém esses dados auxiliares, o membro `cmsg_level` será `IPPROTO_IPV6`, o membro `cmsg_type` será `IPV6_HOPLIMIT` e o primeiro byte de dados será o primeiro byte do inteiro (4 bytes) limite de hops. Mostramos isso na Figura 22.21. Perceba que o valor retornado como dados auxiliares é o valor real proveniente do datagrama recebido, enquanto o valor retornado por um `getsockopt` da opção `IPV6_UNICAST_HOPS` é o valor-padrão que o kernel utilizará para datagramas saintes no soquete.

Para controlar o limite de hops de saída para um dado pacote, apenas especifique as informações de controle como dados auxiliares para `sendmsg`. Os valores normais para o limite de hops estão entre 0 e 255, inclusive, mas, se o valor de inteiro for –1, isso informa o kernel a utilizar seu default.

> O limite de hops não está contido na estrutura `in6_pktinfo` pela seguinte razão: alguns servidores UDP querem responder a solicitações do cliente enviando a resposta à mesma interface em que a solicitação foi recebida, com o endereço IPv6 de origem da resposta igual ao endereço IPv6 de destino da solicitação. Para fazer isso, a aplicação pode ativar somente a opção de soquete `IPV6_RECVPKTINFO` e então utilizar as informações de controle recebidas provenientes de `recvmsg` como as informações de controle de saídas para `sendmsg`. A aplicação não precisa absolutamente examinar ou modificar a estrutura `in6_pktinfo`. Mas, se o limite de hops estiver contido nessa estrutura, a aplicação teria de analisar sintaticamente as informações de controle recebidas e alterar o membro-limite de hops, uma vez que o limite de hops recebido não é o valor desejado para um pacote sainte.

Especificando o endereço do próximo hop

O objeto dos dados auxiliares `IPV6_NEXTHOP` especifica o próximo hop do datagrama como uma estrutura do endereço de soquete. Na estrutura `cmsghdr` que contém esses dados auxiliares, o membro `cmsg_level` é `IPPROTO_IPV6`, o membro `cmsg_type` é `IPV6_NEXTHOP` e o primeiro byte de dados é o primeiro byte da estrutura do endereço de soquete.

Na Figura 22.21, mostramos um exemplo desse objeto dos dados auxiliares, supondo que a estrutura do endereço de soquete seja uma estrutura `sockaddr_in6` de 28 bytes. Nesse caso, o nó identificado por esse endereço deve ser um vizinho do host de envio. Se esse endereço for igual ao endereço IPv6 de destino do datagrama, isso é então equivalente à opção de soquete `SO_DONTROUTE` existente. O endereço do próximo hop pode ser configurado para todos os pacotes em um soquete configurando a opção de soquete `IPV6_NEXTHOP` com o `sockaddr_in6` como o valor de opção, como descrito na Seção 27.7. Configurar essa opção requer privilégios de superusuário.

Especificando e recebendo a classe de tráfego

O objeto dados auxiliares `IPV6_TCLASS` especifica a classe de tráfego para o datagrama. Na estrutura `cmsghdr` que contém esses dados auxiliares, o membro `cmsg_level` será `IPPROTO_IPV6`, o membro `cmsg_type` será `IPV6_TCLASS` e o primeiro byte de dados será o primeiro byte do inteiro (4 bytes) da classe de tráfego. Mostramos isso na Figura 22.21. Como descrito na Seção A.3, a classe de tráfego é composta pelos campos *DSCP* e *ECN*. Esses campos devem ser configurados juntos. O kernel pode mascarar ou ignorar o valor especificado pelo usuário se precisar controlar o valor (por exemplo, se o kernel implementar o ECN, poderá configurar os bits deste para seu próprio valor desejado, ignorando os dois bits especificados com a opção `IPV6_TCLASS`). A classe de tráfego especificada pode estar no intervalo normal entre 0 e 255 ou –1 para permitir ao kernel utilizar o valor default.

Para especificar a classe de tráfego para um dado pacote, inclua os dados auxiliares com esse pacote. Para especificar a classe de tráfego para todos os pacotes em um soquete, especifique a classe de tráfego como um inteiro para a opção de soquete `IPV6_TCLASS`, como descrito na Seção 27.7. A classe de tráfego recebida é retornada como dados auxiliares por `recvmsg` somente se a aplicação tiver a opção de soquete `IPV6_RECVTCLASS` ativada.

22.9 Controle do MTU do caminho IPv6

O IPv6 oferece às aplicações vários controles para a descoberta do MTU de caminho (Seção 2.11). Os padrões são apropriados para a ampla maioria das aplicações, mas os programas de uso especial talvez queiram modificar o comportamento da descoberta do MTU de caminho. Quatro opções de soquete são oferecidas para esse propósito.

Enviando com um MTU mínimo

Ao realizar a descoberta do MTU do caminho, os pacotes normalmente são fragmentados utilizando o MTU da interface de saída ou o MTU do caminho, o que for menor. O IPv6 define um MTU mínimo de 1.280 bytes, que deve ser suportado por todos os caminhos. Fragmentar com esse MTU mínimo desperdiça oportunidades para enviar pacotes maiores (o que é mais eficiente), mas evita as desvantagens da descoberta do MTU do caminho (pacotes descartados e retardos enquanto o MTU está sendo descoberto).

Duas classes de aplicações talvez queiram utilizar o MTU mínimo: aquelas que utilizam o multicast (para evitar uma implosão de mensagens ICMP de "pacotes muito grandes") e aquelas que realizam transações breves para uma grande quantidade de destinos (como o DNS). Aprender o MTU para uma sessão de multicast talvez não seja o suficiente para compensar o custo do recebimento e processamento de milhões de mensagens ICMP de "pacotes muito grandes", e geralmente aplicações como o DNS não conversam com o mesmo servidor com uma freqüência suficiente para tornar vantajoso o risco do custo dos pacotes descartados.

O uso do MTU mínimo é controlado pela opção de soquete `IPV6_USE_MIN_MTU`. Ela tem três valores definidos: –1, o padrão, utiliza o MTU mínimo para destinos multicast, mas realiza a descoberta do MTU de caminho para destinos unicast; 0 realiza a descoberta do MTU de caminho para todos os destinos; e 1 utiliza o MTU mínimo para todos os destinos.

`IPV6_USE_MIN_MTU` também pode ser enviado como dados auxiliares. Na estrutura `cmsghdr` que contém esses dados auxiliares, o membro `cmsg_level` será `IPPROTO_IPV6`, o membro `cmsg_type` será `IPV6_USE_MIN_MTU` e o primeiro byte de dados será o primeiro byte do valor de inteiro (4 bytes).

Recebendo indicações de alteração no MTU do caminho

Para receber notificações de alteração no MTU do caminho, uma aplicação pode ativar a opção de soquete `IPV6_RECVPATHMTU`. Esse flag permite a recepção do MTU do caminho

como dados auxiliares a qualquer momento que mudar. `recvmsg` retornará um datagrama de comprimento zero, mas haverá dados auxiliares que indicam o MTU do caminho. Na estrutura `cmsghdr` que contém esses dados auxiliares, o membro `cmsg_level` será `IPPROTO_IPV6`, o membro `cmsg_type` será `IPV6_PATHMTU` e o primeiro byte de dados será o primeiro byte de uma estrutura `ip6_mtuinfo`. Essa estrutura contém o destino no qual o MTU do caminho mudou e o novo valor do MTU de caminho em bytes.

```
struct ip6_mtuinfo {
    struct sockaddr_in6 ip6m_addr;     /* endereço de destino */
    uint32_t            ip6m_mtu;      /* MTU do caminho na ordem de byte
                                          do host */
};
```

Essa estrutura é definida incluindo o cabeçalho `<netinet/in.h>`.

Determinando o MTU do caminho atual

Se uma aplicação não estiver monitorando com a opção `IPV6_RECVPATHMTU`, ela poderá determinar o MTU do caminho atual de um soquete *conectado* com a opção de soquete `IPV6_PATHMTU`. Essa é uma opção "get-only" ("somente obtenção"), que retorna uma estrutura `ip6_mtuinfo` (veja acima) que contém o MTU do caminho atual. Se nenhum MTU de caminho tiver sido determinado, ele retorna o MTU da interface de saída. O valor do endereço retornado é indefinido.

Evitando a fragmentação

Por padrão, a pilha IPv6 fragmentará os pacotes saintes para o MTU do caminho. Uma aplicação como o `traceroute` talvez não queira essa fragmentação automática, para descobrir o MTU do caminho por conta própria. A opção de soquete `IPV6_DONTFRAG` é utilizada para desativar a fragmentação automática; um valor de 0 (default) permite a fragmentação automática, enquanto um valor de 1 a desativa.

Se a fragmentação automática estiver desativada, uma chamada a `send` fornecendo um pacote que requer fragmentação *poderia* retornar `EMSGSIZE`; entretanto, não é exigido que a implementação forneça isso. A única maneira de determinar se um pacote requer fragmentação é utilizar a opção `IPV6_RECVPATHMTU`, que já foi descrita.

`IPV6_DONTFRAG` também poder ser enviado como dados auxiliares. Na estrutura `cmsghdr` que contém esses dados auxiliares, o membro `cmsg_level` será `IPPROTO_IPV6`, o membro `cmsg_type` será `IPV6_DONTFRAG` e o primeiro byte de dados será o primeiro byte do valor inteiro (4 bytes).

22.10 Resumo

Há aplicações que querem conhecer o endereço IP de destino e a interface recebida para um datagrama UDP. As opções de soquete `IP_RECVDSTADDR` e `IP_RECVIF` podem ser ativadas para retornarem essas informações como dados auxiliares com cada datagrama. Informações semelhantes, juntamente com o limite de hops recebido, podem ser retornadas para soquetes IPv6 ativando a opção de soquete `IPV6_RECVPKTINFO`.

Apesar de todos os recursos fornecidos pelo TCP que não são fornecidos pelo UDP, há momentos para utilizar o UDP. O UDP deve ser utilizado para broadcast ou multicast e pode ser utilizado para cenários simples de solicitação e resposta (mas alguma forma de confiabilidade deve então ser adicionada à aplicação). O UDP não deve ser utilizado para transferência de dados em grande volume.

Adicionamos confiabilidade ao nosso cliente UDP na Seção 22.5 para detectar pacotes perdidos utilizando um tempo-limite e retransmissão. Modificamos nosso tempo-limite de re-

transmissão dinamicamente adicionando um registro de data/hora a cada pacote e monitorando dois estimadores: o RTT e seu desvio médio. Também adicionamos um número de seqüência para verificar se uma resposta dada era a esperada. Nosso cliente ainda empregou um protocolo "pare e espere" simples, mas esse é o tipo de aplicação com o qual o UDP pode ser utilizado.

Exercícios

22.1 Na Figura 22.18, por que há duas chamadas a `printf`?

22.2 `dg_send_recv` pode (Figuras 22.8 e 22.9) retornar 0?

22.3 Recodifique `dg_send_recv` para utilizar `select` e seu timer em vez de utilizar `alarm`, `SIGALRM`, `sigsetjmp` e `siglongjmp`.

22.4 Como um servidor IPv4 pode garantir que o endereço de origem da sua resposta seja igual ao endereço de destino da solicitação do cliente (por exemplo, funcionalidade semelhante à fornecida pela opção de soquete `IPV6_PKTINFO`)?

22.5 A função `main` na Seção 22.6 é dependente de protocolo no IPv4. Recodifique essa função para ser independente de protocolo. Exija que o usuário especifique um ou dois argumentos de linha de comando, o primeiro sendo um endereço IP opcional (por exemplo, 0.0.0.0 ou 0::0) e o segundo um número de porta requerido. Em seguida, chame `udp_client` para obter apenas a família de endereço, o número de porta e o comprimento da estrutura do endereço de soquete.

O que acontece se você chamar `udp_client`, como sugerido, sem especificar um argumento *hostname* porque `udp_client` não especifica a dica `AI_PASSIVE` para `getaddrinfo`?

22.6 Execute o cliente na Figura 22.6 para um servidor de eco pela Internet depois de modificar as funções RTT para imprimir cada RTT. Além disso, modifique a função `dg_send_recv` para imprimir cada número de seqüência recebido. Plote os RTTs resultantes juntamente com o estimadores para o RTT e seu desvio médio.

Capítulo 23

Soquetes Avançados de SCTP

23.1 Visão geral

Neste capítulo, analisaremos o SCTP em mais detalhes, examinando um maior número de recursos e opções de soquete que ele oferece. Discutiremos alguns tópicos, incluindo controle de detecção de falha, dados não-ordenados e notificações. Também forneceremos exemplos de códigos para que o leitor possa verificar como utilizar alguns recursos avançados do SCTP.

O SCTP é um protocolo orientado a mensagens, entregando mensagens parciais ou completas ao usuário. As mensagens parciais só serão entregues se a aplicação optar por enviar grandes mensagens (por exemplo, maior que a metade do tamanho do buffer de soquete) ao seu peer. Quando mensagens parciais são entregues, o SCTP nunca agrupará duas mensagens parciais. Uma aplicação receberá uma mensagem completa em uma operação de recebimento ou receberá uma mensagem em várias operações de recebimento consecutivas. Ilustraremos um método de lidar com esse mecanismo de entrega parcial por meio de um exemplo de função utilitária..

Os servidores SCTP podem ser iterativos ou concorrentes, dependendo do estilo de interface que o desenvolvedor de aplicações escolher. O SCTP também fornece um método de extrair uma associação a partir de um soquete no estilo "um para muitos" para um soquete no estilo "um para um" separado. Esse método permite a construção de um servidor que é iterativo e concorrente.

23.2 Um servidor no estilo um para muitos com autofechamento

Lembre-se do programa servidor que escrevemos no Capítulo 10. Esse programa não monitora nenhuma associação. O servidor depende do cliente para fechar a associação, removendo assim o estado de associação. Mas depender do cliente para isso introduz uma fraqueza: o que acontece se um cliente abrir uma associação e nunca enviar nenhum dado? Seriam alocados recursos para um cliente que nunca irá utilizá-los. Essa dependência poderia introduzir um ataque de recusa de serviço acidental na nossa implementação SCTP para clientes preguiçosos. Para evitar esse problema, foi adicionado ao SCTP um recurso de *autofechamento*.

O autofechamento deixa que uma extremidade do SCTP especifique um número máximo de segundos que uma associação pode permanecer desocupada. Considera-se que uma associação está desocupada quando ela não está transmitindo dados do usuário em nenhuma direção. Se uma associação estiver desocupada além desse tempo máximo, a associação é automaticamente fechada pela implementação SCTP.

Ao utilizar essa opção, tenha cuidado em escolher um valor para o autofechamento. O servidor não deve selecionar um valor muito pequeno, caso contrário ele próprio talvez se encontre na situação de precisar enviar dados a uma associação que foi fechada. Há um overhead extra ao reabrir a associação para enviar os dados de volta ao cliente e é improvável que o cliente tenha executado um `listen` para ativar as associações entrantes. A Figura 23.1 revisita o código do nosso servidor e insere as chamadas necessárias para torná-lo resistente a associações desocupadas ou caducas. Como descrito na Seção 7.10, o autofechamento assume o default de desativado e deve ser explicitamente ativado com a opção de soquete `SCTP_AUTOCLOSE`.

sctp/sctpserv04.c

```
14     if (argc == 2)
15         stream_increment = atoi(argv[1]);
16     sock_fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
17     close_time = 120;
18     Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_AUTOCLOSE,
19                &close_time, sizeof(close_time));
20     bzero(&servaddr, sizeof(servaddr));
21     servaddr.sin_family = AF_INET;
22     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
23     servaddr.sin_port = htons(SERV_PORT);
```

sctp/sctpserv04.c

Figura 23.1 Um servidor ativando o autofechamento.

Configuração da opção de autofechamento

17–19 O servidor seleciona um valor de 120 segundos para desativar associações desocupadas e coloca esse valor na variável `close_time`. Em seguida, chama a opção de soquete que configura o tempo do autofechamento. Todo o código restante no servidor permanece inalterado.

Agora, o SCTP fechará automaticamente as associações que permanecem desocupadas por mais de dois minutos. Forçando a associação a fechar automaticamente, reduzimos a quantidade de recursos do servidor consumida por clientes preguiçosos.

23.3 Entrega parcial

A entrega parcial será utilizada pela implementação SCTP sempre que uma "grande" mensagem é recebida, "grande" significando que a pilha do SCTP considera que ela não tem os recursos para dedicar à mensagem. As seguintes considerações serão feitas pela implementação SCTP receptora antes de iniciar essa API:

- A quantidade de espaço em buffer sendo consumida pela mensagem deve atender ou exceder algum limiar.
- A pilha pode ser entregue somente a partir do início da mensagem seqüencialmente até o primeiro pedaço que esteja ausente.

- Depois de invocada, nenhuma outra mensagem pode ser disponibilizada ao usuário até que a mensagem atual tenha sido integralmente recebida e passada para o usuário. Isso significa que uma mensagem grande bloqueia todas as outras mensagens que normalmente poderiam ser entregues, incluindo aquelas em outros fluxos.

A implementação KAME do SCTP utiliza um limiar que é a metade do buffer de recebimento do soquete. Quando este livro estava sendo escrito, o buffer de recebimento default para a pilha era 131.072 bytes. Portanto, sem alterar o `SO_RCVBUF`, uma única mensagem deve ser maior que 65.536 bytes antes que a API de entrega parcial seja invocada. Para estender ainda mais a nova versão do servidor na Seção 10.2, escrevemos uma função utilitária que empacota a chamada de função `sctp_recvmsg`. Em seguida, criamos um servidor modificado que utiliza a nossa nova função. A Figura 23.2 mostra nossa função empacotadora que trata a API de entrega parcial.

Preparação do buffer estático

12–15 Se o buffer estático da função não foi alocado, aloque-o e configure estado associado a ele.

Leitura da mensagem

16–18 Leia a primeira mensagem utilizando a função `sctp_recvmsg`.

Tratando o erro da leitura inicial

19–22 Se `sctp_recvmsg` retornar um erro ou um EOF, nós o passamos diretamente para o chamador.

Enquanto houver outros dados para essa mensagem

23–24 Enquanto os flags da mensagem mostrarem que a função não recebeu uma mensagem completa, colete mais dados. A função inicia calculando quanto ainda permanece no buffer estático.

Verificando a necessidade de aumentar o buffer estático

25–34 Sempre que a função não mais tiver uma quantidade mínima de espaço remanescente no seu buffer de recebimento, ela deverá aumentar o buffer. Fazemos isso utilizando a função `realloc` para alocar o novo tamanho atual ao buffer, mais uma quantidade aumentada e uma cópia dos dados antigos. Se por alguma razão a função não mais puder aumentar seu buffer, ela encerra com um erro.

Recebendo mais dados

35–36 Colete mais dados com a função `sctp_recvmsg`.

Prosseguimento

37–38 A função incrementa o índice do buffer e volta para testar se leu toda a mensagem.

Ao final do loop

39–40 Quando o loop termina, a função copia o número de bytes lido para o ponteiro fornecido pelo chamador e retorna um ponteiro ao buffer alocado.

sctp/sctp_pdapircv.c

```
1 #include   "unp.h"

2 static uint8_t *sctp_pdapi_readbuf = NULL;
3 static int sctp_pdapi_rdbuf_sz = 0;

4 uint8_t *
```

Figura 23.2 Tratando a API de entrega parcial (*continua*).

```
 5 pdapi_recvmsg(int sock_fd,
 6                int *rdlen,
 7                SA *from,
 8                int *from_len, struct sctp_sndrcvinfo *sri, int *msg_flags)
 9 {
10     int    rdsz, left, at_in_buf;
11     int    frmlen = 0;

12     if (sctp_pdapi_readbuf == NULL) {
13         sctp_pdapi_readbuf = (uint8_t *) Malloc(SCTP_PDAPI_INCR_SZ);
14         sctp_pdapi_rdbuf_sz = SCTP_PDAPI_INCR_SZ;
15     }
16     at_in_buf =
17         Sctp_recvmsg(sock_fd, sctp_pdapi_readbuf, sctp_pdapi_rdbuf_sz, from,
18                      from_len,sri, msg_flags);
19     if (at_in_buf < 1) {
20         *rdlen = at_in_buf;
21         return (NULL);
22     }
23     while ((*msg_flags & MSG_EOR) == 0) {
24         left = sctp_pdapi_rdbuf_sz - at_in_buf;
25         if (left < SCTP_PDAPI_NEED_MORE_THRESHOLD) {
26             sctp_pdapi_readbuf =
27                 realloc(sctp_pdapi_readbuf,
28                         sctp_pdapi_rdbuf_sz + SCTP_PDAPI_INCR_SZ);
29             if (sctp_pdapi_readbuf == NULL) {
30                 err_quit("sctp_pdapi ran out of memory");
31             }
32             sctp_pdapi_rdbuf_sz += SCTP_PDAPI_INCR_SZ;
33             left = sctp_pdapi_rdbuf_sz - at_in_buf;
34         }
35         rdsz = Sctp_recvmsg(sock_fd, &sctp_pdapi_readbuf[at_in_buf],
36                             left, NULL, &frmlen, NULL, msg_flags);
37         at_in_buf += rdsz;
38     }
39     *rdlen = at_in_buf;
40     return (sctp_pdapi_readbuf);
41 }
```

sctp/sctp_pdapircv.c

Figura 23.2 Tratando a API de entrega parcial (*continuação*).

Em seguida, modificamos nosso servidor na Figura 23.3 de modo que ele utilize a nova função.

Leitura da mensagem

29–30 Aqui, o servidor chama a nova função utilitária da entrega parcial. O servidor chama essa função depois de anular quaisquer dados antigos que ainda possam estar na variável `sri`.

sctp/sctpserv05.c

```
26     for ( ; ; ) {
27         len = sizeof(struct sockaddr_in);
28         bzero(&sri, sizeof(sri));
29         readbuf = pdapi_recvmsg(sock_fd, &rd_sz,
30                                 (SA *) &cliaddr, &len, &sri, &msg_flags);
31         if (readbuf == NULL)
32             continue;
```

sctp/sctpserv05.c

Figura 23.3 Nosso servidor utilizando a API de entrega parcial.

Verificando se lemos alguma coisa

31–32 Observe que agora o servidor deve fazer um teste de NULL para ver se a leitura foi bem-sucedida. Se não foi bem-sucedida, o servidor simplesmente continua.

23.4 Notificações

Como discutido na Seção 9.14, uma aplicação pode inscrever-se para até sete notificações. Até agora, nossa aplicação ignorou todos os eventos que poderiam ocorrer além do recebimento de novos dados. Os exemplos nesta seção fornecem uma visão geral sobre como receber e interpretar as notificações do SCTP sobre eventos adicionais na camada de transporte. A Figura 23.4 mostra uma função que irá exibir todas as notificações provenientes do transporte. Também modificaremos nosso servidor para ativar todos os eventos; chamaremos essa nova função quando uma notificação é recebida. Observe que o nosso servidor, na verdade, não está utilizando a notificação para qualquer propósito específico.

Lançamento e troca

14–15 A função lança o buffer entrante ao tipo de união geral. Ela desreferencia a estrutura `sn_header` genérica e o tipo genérico `sn_type` e troca esse valor.

Processando a alteração da associação

16–40 Se a função encontrar uma notificação sobre uma alteração da associação no buffer, ela imprimirá o tipo de alteração que ocorreu.

Alteração do endereço do peer

41–66 Se for uma notificação do endereço do peer, a função imprimirá o evento do endereço (depois de decodificá-lo) e o endereço.

Erro remoto

67–71 Se a função encontrar um erro remoto, ela exibirá esse fato e o ID da associação em que o erro ocorreu. A função não se incomoda em decodificar e exibir o erro real informado pelo peer remoto. A informação está disponível no campo `sre_data` da estrutura `sctp_remote_error`.

Falha de envio da mensagem

72–76 Se a função decodificar uma notificação de falha de envio, ela saberá que uma mensagem não foi enviada ao peer. Isso significa que: (i) a associação está caindo e uma notificação de associação virá em seguida (se ainda não chegou), ou (ii) o servidor está utilizando a extensão de confiabilidade parcial e uma mensagem não foi enviada com sucesso (devido a restrições impostas na transferência). Os dados realmente enviados estão disponíveis à função no campo `ssf_data` (que nossa função não examina).

Indicação da camada de adaptação

77–81 Se a função decodificar um indicador de camada de adaptação, ela exibirá o valor de 32 bits passado na mensagem de configuração (INIT ou INIT-ACK).

Notificação de entrega parcial

82–89 Se uma notificação de entrega parcial chegar, a função irá anunciá-la. O único evento definido durante a escrita deste livro era que a entrega parcial era abortada.

sctp/sctp_displayevents.c

```
 1 #include   "unp.h"
 2 void
 3 print_notification(char *notify_buf)
 4 {
 5     union sctp_notification *snp;
 6     struct sctp_assoc_change *sac;
 7     struct sctp_paddr_change *spc;
 8     struct sctp_remote_error *sre;
 9     struct sctp_send_failed *ssf;
10     struct sctp_shutdown_event *sse;
11     struct sctp_adaption_event *ae;
12     struct sctp_pdapi_event *pdapi;
13     const char *str;
14     snp = (union sctp_notification *) notify_buf;
15     switch (snp->sn_header.sn_type) {
16     case SCTP_ASSOC_CHANGE:
17         sac = &snp->sn_assoc_change;
18         switch (sac->sac_state) {
19         case SCTP_COMM_UP:
20             str = "COMMUNICATION UP";
21             break;
22         case SCTP_COMM_LOST:
23             str = "COMMUNICATION LOST";
24             break;
25         case SCTP_RESTART:
26             str = "RESTART";
27             break;
28         case SCTP_SHUTDOWN_COMP:
29             str = "SHUTDOWN COMPLETE";
30             break;
31         case SCTP_CANT_STR_ASSOC:
32             str = "CAN'T START ASSOC";
33             break;
34         default:
35             str = "UNKNOWN";
36             break;
37         }                          /* termina o switch(sac->sac_state) */
38         printf("SCTP_ASSOC_CHANGE: %s, assoc=0x%x\n", str,
39                (uint32_t) sac->sac_assoc_id);
40         break;
41     case SCTP_PEER_ADDR_CHANGE:
42         spc = &snp->sn_paddr_change;
43         switch (spc->spc_state) {
44         case SCTP_ADDR_AVAILABLE:
45             str = "ADDRESS AVAILABLE";
46             break;
47         case SCTP_ADDR_UNREACHABLE:
48             str = "ADDRESS UNREACHABLE";
49             break;
50         case SCTP_ADDR_REMOVED:
51             str = "ADDRESS REMOVED";
52             break;
53         case SCTP_ADDR_ADDED:
54             str = "ADDRESS ADDED";
55             break;
56         case SCTP_ADDR_MADE_PRIM:
57             str = "ADDRESS MADE PRIMARY";
58             break;
59         default:
```

Figura 23.4 Um utilitário de exibição de notificações (*continua*).

```
60              str = "UNKNOWN";
61              break;
62          }                           /* termina o switch(spc->spc_state) */
63          printf("SCTP_PEER_ADDR_CHANGE: %s, addr=%s, assoc=0x%x\n", str,
64                  Sock_ntop((SA *) &spc->spc_aaddr, sizeof(spc->spc_aaddr)),
65                  (uint32_t) spc->spc_assoc_id);
66          break;
67      case SCTP_REMOTE_ERROR:
68          sre = &snp->sn_remote_error;
69          printf("SCTP_REMOTE_ERROR: assoc=0x%x error=%d\n",
70                  (uint32_t) sre->sre_assoc_id, sre->sre_error);
71          break;
72      case SCTP_SEND_FAILED:
73          ssf = &snp->sn_send_failed;
74          printf("SCTP_SEND_FAILED: assoc=0x%x error=%d\n",
75                  (uint32_t) ssf->ssf_assoc_id, ssf->ssf_error);
76          break;
77      case SCTP_ADAPTION_INDICATION:
78          ae = &snp->sn_adaption_event;
79          printf("SCTP_ADAPTION_INDICATION: 0x%x\n",
80                  (u_int) ae->sai_adaption_ind);
81          break;
82      case SCTP_PARTIAL_DELIVERY_EVENT:
83          pdapi = &snp->sn_pdapi_event;
84          if (pdapi->pdapi_indication == SCTP_PARTIAL_DELIVERY_ABORTED)
85              printf("SCTP_PARTIAL_DELIVERY_ABORTED\n");
86          else
87              printf("Unknown SCTP_PARTIAL_DELIVERY_EVENT 0x%x\n",
88                      pdapi->pdapi_indication);
89          break;
90      case SCTP_SHUTDOWN_EVENT:
91          sse = &snp->sn_shutdown_event;
92          printf("SCTP_SHUTDOWN_EVENT: assoc=0x%x\n",
93                  (uint32_t) sse->sse_assoc_id);
94          break;
95      default:
96          printf("Unknown notification event type=0x%x\n",
97                  snp->sn_header.sn_type);
98      }
99  }
```

sctp/sctp_displayevents.c

Figura 23.4 Um utilitário de exibição de notificações (*continuação*).

Notificação de desativação

90–94 Se a função decodificar uma notificação de shutdown, ela saberá que o peer emitiu uma desativação elegante. Essa notificação normalmente é seguida por uma notificação de alteração da associação quando a seqüência de desativação é completada.

A modificação para que o servidor utilize nossa nova função pode ser vista na Figura 23.5.

sctp/sctpserv06.c

```
21      bzero(&evnts, sizeof(evnts));
22      evnts.sctp_data_io_event = 1;
23      evnts.sctp_association_event = 1;
24      evnts.sctp_address_event = 1;
25      evnts.sctp_send_failure_event = 1;
26      evnts.sctp_peer_error_event = 1;
27      evnts.sctp_shutdown_event = 1;
```

Figura 23.5 Um servidor modificado que utiliza notificações (*continua*).

```
28      evnts.sctp_partial_delivery_event = 1;
29      evnts.sctp_adaption_layer_event = 1;
30      Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_EVENTS, &evnts, sizeof(evnts));
31      Listen(sock_fd, LISTENQ);
32      for ( ; ; ) {
33          len = sizeof(struct sockaddr_in);
34          rd_sz = Sctp_recvmsg(sock_fd, readbuf, sizeof(readbuf),
35                               (SA *) &cliaddr, &len, &sri, &msg_flags);
36          if (msg_flags & MSG_NOTIFICATION) {
37              print_notification(readbuf);
38              continue;
39          }
```

sctp/sctpserv06.c

Figura 23.5 Um servidor modificado que utiliza notificações (*continuação*).

Configuração para receber notificações

21–30 Aqui o servidor altera as configurações de evento para receber todas as notificações.

Código de recebimento normal

31–35 Esta seção do código do servidor permanece inalterada.

Tratando a notificação

36–39 Aqui o servidor verifica o campo `msg_flags`. Se o servidor descobrir que os dados são uma notificação, ele chamará a nossa função de exibição `sctp_print_notification` e fará um loop para ler a próxima mensagem.

Executando o código

Iniciamos o cliente e enviamos uma mensagem como a seguir:

```
FreeBSD-lap: ./sctpclient01 10.1.1.5
[0]Hello
From str:1 seq:0 (assoc:c99e15a0):[0]Hello
Control-D
FreeBSD-lap:
```

Ao receber a conexão, a mensagem e o término de conexão, nosso servidor modificado exibe cada evento à medida que eles ocorrem.

```
FreeBSD-lap: ./sctpserv06
SCTP_ADAPTION_INDICATION:0x504c5253
SCTP_ASSOC_CHANGE: COMMUNICATION UP, assoc=c99e2680h
SCTP_SHUTDOWN_EVENT: assoc=c99e2680h
SCTP_ASSOC_CHANGE: SHUTDOWN COMPLETE, assoc=c99e2680h
Control-C
```

Como você pode ver, o servidor agora anuncia os eventos à medida que eles ocorrem no transporte.

23.5 Dados não-ordenados

O SCTP normalmente fornece uma entrega ordenada confiável dos dados, além de um serviço não-ordenado confiável. Uma mensagem com o flag `MSG_UNORDERED` é enviada sem restrições de ordem e pode ser entregue assim que chegar. Dados não-ordenados podem ser enviados em qualquer fluxo. Não é atribuído nenhum número de seqüência ao fluxo. A Figura 23.6 mostra as alterações necessárias ao nosso programa cliente para enviar a solicitação ao servidor de eco com o serviço de dados não-ordenado.

sctp/sctp_strcli_un.c

```
18          out_sz = strlen(sendline);
19          Sctp_sendmsg(sock_fd, sendline, out_sz,
20                       to, tolen, 0, MSG_UNORDERED, sri.sinfo_stream, 0, 0);
```

sctp/sctp_strcli_un.c

Figura 23.6 `sctp_strcli` que envia dados não-ordenados.

Envio de dados utilizando um serviço não-ordenado

18–20 Isso é quase idêntico à função `sctpstr_cli` desenvolvida na Seção 10.4. Na linha 21, vemos uma única alteração: o cliente passa explicitamente o flag `MSG_UNORDERED` para invocar o serviço não-ordenado. Normalmente, todos os dados dentro de um dado fluxo são ordenados com os números da seqüência. O flag `MSG_UNORDERED` faz com que os dados enviados com esse flag sejam enviados de maneira não-ordenada, sem nenhum número de seqüência, podendo ser entregues assim que chegarem, mesmo se outros dados não-ordenados enviados anteriormente no mesmo fluxo ainda não chegaram.

23.6 Vinculando um subconjunto de endereços

Algumas aplicações talvez queiram vincular um subconjunto adequado de endereços IP de uma máquina a um único soquete. No TCP e no UDP, tradicionalmente, não era possível vincular um subconjunto de endereços. A chamada de sistema `bind` permite a uma aplicação vincular um único endereço ou o endereço curinga. Devido a essa restrição, a nova chamada de função `sctp_bindx` é fornecida para permitir a uma aplicação vincular mais de um endereço. Observe que todos os endereços devem utilizar o mesmo número de porta e, se `bind` tiver sido chamada, o número da porta deve ser o mesmo que o fornecido para `bind`. A chamada `sctp_bindx` falhará se uma porta diferente for fornecida. A Figura 23.7 mostra um utilitário que adicionaremos ao nosso servidor que vinculará uma lista de argumentos.

sctp/sctp_bindargs.c

```
 1 #include   "unp.h"
 2 int
 3 sctp_bind_arg_list(int sock_fd, char **argv, int argc)
 4 {
 5     struct addrinfo *addr;
 6     char    *bindbuf, *p, portbuf[10];
 7     int     addrcnt = 0;
 8     int     i;
 9     bindbuf = (char *) Calloc(argc, sizeof(struct sockaddr_storage));
10     p = bindbuf;
11     sprintf(portbuf, "%d", SERV_PORT);
12     for (i = 0; i < argc; i++) {
13         addr = Host_serv(argv[i], portbuf, AF_UNSPEC, SOCK_SEQPACKET);
14         memcpy(p, addr->ai_addr, addr->ai_addrlen);
15         freeaddrinfo(addr);
16         addrcnt++;
17         p += addr->ai_addrlen;
18     }
19     Sctp_bindx(sock_fd, (SA *) bindbuf, addrcnt, SCTP_BINDX_ADD_ADDR);
20     free(bindbuf);
21     return (0);
22 }
```

sctp/sctp_bindargs.c

Figura 23.7 Função para vincular um subconjunto de endereços.

Alocando espaço para argumentos de vinculação

9–10 Nossa função `sctp_bind_arg_list` inicia alocando espaço para os argumentos da vinculação. Observe que a função `sctp_bindx` pode aceitar uma combinação de endereços IPv4 e IPv6. Alocamos espaço suficiente a um `sockaddr_storage` para cada endereço, mesmo que o argumento de lista de endereço para `sctp_bindx` seja uma lista empacotada de endereços (Figura 9.4). Isso resulta em algum desperdício de memória, mas é mais simples do que calcular o espaço exato requerido processando a lista de argumentos duas vezes.

Processamento dos argumentos

11–18 Configuramos o *portbuf* como uma representação ASCII do número da porta, para preparar a chamada à nossa função empacotadora `getaddrinfo`, `host_serv`. Passamos cada endereço e o número da porta para `host_serv`, juntamente com `AF_UNSPEC` para permitir o IPv4 ou o IPv6, e `SOCK_SEQPACKET` para especificar que estamos utilizando o SCTP. Copiamos o primeiro `sockaddr` que retorna e ignoramos quaisquer outros. Como os argumentos para essa função são concebidos para serem strings de endereço literal, em oposição a nomes que poderiam ter múltiplos endereços associados, isso é seguro. Liberamos o valor de retorno de `getaddrinfo`, para incrementar nossa contagem de endereços, e movemos o ponteiro para o próximo elemento no nosso array empacotado de estruturas `sockaddr`.

Chamada à função de vinculação

19 A função agora redefine seu ponteiro na parte superior do buffer de vinculação e chama `sctp_bindx` com o subconjunto de endereços decodificado anteriormente.

Retorno com sucesso

20–21 Se a função alcançar esse ponto, seremos bem-sucedidos, portanto, limpe e retorne.

A Figura 23.8 ilustra o nosso servidor de eco modificado que agora vincula uma lista de endereços passada para a linha de comando. Observe que modificamos o servidor ligeiramente a fim de que ele sempre retorne qualquer mensagem ecoada no fluxo em que chegou.

sctp/sctpserv07.c

```
12      if (argc < 2)
13          err_quit("Error, use %s [list of addresses to bind]\n", argv[0]);
14      sock_fd = Socket(AF_INET6, SOCK_SEQPACKET, IPPROTO_SCTP);

15      if (sctp_bind_arg_list(sock_fd, argv + 1, argc - 1))
16          err_sys("Can't bind the address set");

17      bzero(&evnts, sizeof(evnts));
18      evnts.sctp_data_io_event = 1;
```

sctp/sctpserv07.c

Figura 23.8 Servidor utilizando um conjunto variável de endereços.

Código do servidor utilizando o IPv6

14 Aqui, vemos o servidor em que estamos trabalhando por todo este capítulo, mas com uma pequena modificação. O servidor cria um soquete `AF_INET6`. Dessa maneira, o servidor pode utilizar tanto o IPv4 como o IPv6.

Utilizando a nova função `sctp_bind_arg_list`

15–16 O servidor chama a nova função `sctp_bind_arg_list`, passando a lista de argumentos para ela processar.

23.7 Determinando as informações sobre peer e endereço local

Como o SCTP é um protocolo multihomed, diferentes mecanismos são necessários para descobrir quais endereços estão em utilização tanto nas extremidades remotas como nas extremidades locais de uma associação. Nesta seção, modificaremos nosso cliente para que ele receba a comunicação na notificação. Nosso cliente utilizará essa notificação para exibir os endereços das extremidades locais e remotas da associação. As Figuras 23.9 e 23.10 mostram as modificações no código do nosso cliente. As Figuras 23.11 e 23.12 mostram o novo código que adicionamos ao cliente.

sctp/sctpclient04

```
16    bzero(&evnts, sizeof(evnts));
17    evnts.sctp_data_io_event = 1;
18    evnts.sctp_association_event = 1;
19    Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_EVENTS, &evnts, sizeof(evnts));
20    sctpstr_cli(stdin, sock_fd, (SA *) &servaddr, sizeof(servaddr));
```

sctp/sctpclient04

Figura 23.9 Configuração do cliente para notificações.

Configuração dos eventos e chamada à função de eco

16–20 Vemos uma pequena alteração na rotina principal do nosso cliente. O cliente inscreve-se explicitamente para as notificações de alteração da associação.

Agora, examinaremos as modificações necessárias a `sctpstr_cli` de modo que utilizará a nossa nova rotina de processamento de notificação.

sctp/sctp_strcli1.c

```
21        do {
22            len = sizeof(peeraddr);
23            rd_sz = Sctp_recvmsg(sock_fd, recvline, sizeof(recvline),
24                                 (SA *) &peeraddr, &len, &sri, &msg_flags);
25            if (msg_flags & MSG_NOTIFICATION)
26                check_notification(sock_fd, recvline, rd_sz);
27        } while (msg_flags & MSG_NOTIFICATION);
28        printf("From str:%d seq:%d (assoc:0x%x):",
29                sri.sinfo_stream, sri.sinfo_ssn, (u_int) sri.sinfo_assoc_id);
30        printf("%.*s", rd_sz, recvline);
```

sctp/sctp_strcli1.c

Figura 23.10 `sctp_strcli` que trata as notificações.

Fazendo um loop esperando a mensagem

21–24 Aqui, o cliente configura a variável de comprimento do endereço e chama a função de recebimento para obter a mensagem ecoada proveniente do servidor.

Verificação das notificações

25–26 O cliente agora verifica se a mensagem que ele acabou de ler é uma notificação. Se for, o cliente chama nossa rotina de processamento de notificação mostrada na Figura 23.11.

Fazendo um loop esperando os dados

27 Se a mensagem lida for uma notificação, fique em loop até ler os dados reais.

Exibindo a mensagem

28–30 Em seguida, o cliente exibe a mensagem e volta à parte superior do seu loop de processamento, esperando a entrada do usuário.

Agora, vamos examinar a nova função `sctp_check_notification`, que exibirá os endereços das duas extremidades quando um evento de notificação de associação chegar.

sctp/sctp_check_notify.c

```
 1 #include   "unp.h"

 2 void
 3 check_notification(int sock_fd, char *recvline, int rd_len)
 4 {
 5     union sctp_notification *snp;
 6     struct sctp_assoc_change *sac;
 7     struct sockaddr_storage *sal, *sar;
 8     int      num_rem, num_loc;

 9     snp = (union sctp_notification *) recvline;
10     if (snp->sn_header.sn_type == SCTP_ASSOC_CHANGE) {
11         sac = &snp->sn_assoc_change;
12         if ((sac->sac_state == SCTP_COMM_UP) ||
13             (sac->sac_state == SCTP_RESTART)) {
14            num_rem = sctp_getpaddrs(sock_fd, sac->sac_assoc_id, &sar);
15            printf("There are %d remote addresses and they are:\n", num_rem);
16            sctp_print_addresses(sar, num_rem);
17            sctp_freepaddrs(sar);

18            num_loc = sctp_getladdrs(sock_fd, sac->sac_assoc_id, &sal);
19            printf("There are %d local addresses and they are:\n", num_loc);
20            sctp_print_addresses(sal, num_loc);
21            sctp_freeladdrs(sal);
22         }
23     }

24 }
```

sctp/sctp_check_notify.c

Figura 23.11 Processamento das notificações.

Verificando se é a notificação esperada

9–13 A função lança o buffer de recebimento ao nosso ponteiro de notificação genérico para encontrar o tipo de notificação. Se for a notificação em que a função está interessada, uma notificação de alteração da associação, ela então testa se a notificação é uma associação nova ou uma reiniciada (`SCTP_COMM_UP` ou `SCTP_RESTART`). Ignoramos todas as outras notificações.

Coletando e imprimindo os endereços do peer

14–17 Chamamos `sctp_getpaddrs` para coletar uma lista de endereços remotos. Em seguida, imprimimos o número de endereços e utilizamos a rotina de impressão de endereço, `sctp_print_addresses`, mostrada na Figura 23.12, para exibir os endereços. Quando essa função termina de utilizar o ponteiro de endereço, ela chama `sctp_freepaddrs` para liberar os recursos alocados por `sctp_getpaddrs`.

Coletando e imprimindo os endereços locais

18–21 Chamamos `sctp_getladdrs` para coletar uma lista de endereços locais, então imprimimos o número de endereços e os próprios endereços. Depois que a função terminar de utilizar os endereços, ela chama a função `sctp_freeladdrs` para liberar os recursos alocados por `sctp_getladdrs`.

Por fim, examinamos uma última nova função, `sctp_print_addresses`, que imprimirá uma lista de endereços na forma retornada pelas funções `sctp_getpaddrs` e `sctp_getladdrs`.

sctp/sctp_print_addrs.c

```
 1 #include   "unp.h"

 2 void
 3 sctp_print_addresses(struct sockaddr_storage *addrs, int num)
 4 {
 5     struct sockaddr_storage *ss;
 6     int    i, salen;

 7     ss = addrs;
 8     for (i = 0; i < num; i++) {
 9         printf("%s\n", Sock_ntop((SA *) ss, salen));
10 #ifdef HAVE_SOCKADDR_SA_LEN
11         salen = ss->ss_len;
12 #else
13         switch (ss->ss_family) {
14         case AF_INET:
15             salen = sizeof(struct sockaddr_in);
16             break;
17 #ifdef IPV6
18         case AF_INET6:
19             salen = sizeof(struct sockaddr_in6);
20             break;
21 #endif
22         default:
23             err_quit("sctp_print_addresses: unknown AF");
24             break;
25         }
26 #endif
27         ss = (struct sockaddr_storage *) ((char *) ss + salen);
28     }
29 }
```

sctp/sctp_print_addrs.c

Figura 23.12 Impressão de uma lista de endereços.

Processando cada endereço

7–8 A função faz um loop através de cada endereço com base no número de endereços que o nosso chamador especificou.

Imprimindo o endereço

9 Imprimimos o endereço utilizando nossa função `sock_ntop`. Lembre-se de que essa função imprime qualquer formato de estrutura de endereço de soquete que o sistema suporte.

Determinando o tamanho do endereço

10–26 A lista de endereços é uma lista empacotada, não um array simples de estruturas `sockaddr_storage`. Isso ocorre porque a estrutura `sockaddr_storage` é muito grande e é um desperdício utilizá-la para passar endereços de um lado a outro entre o kernel e o espaço do usuário. Nos sistemas em que a estrutura sockaddr contém seu próprio comprimento, isso é trivial: simplesmente extraia o comprimento a partir da estrutura `sockaddr_storage` atual. Nos outros sistemas, escolhemos o comprimento com base na família de endereços e encerramos com erro se esta for desconhecida.

Movendo o ponteiro de endereço

27 A função agora adiciona o tamanho do endereço ao ponteiro básico para avançar pela lista de endereços.

Executando o código

Executamos nosso cliente modificado contra o servidor desta maneira:

```
FreeBSD-lap: ./sctpclient01 10.1.1.5
[0]Hi
There are 2 remote addresses and they are:
10.1.1.5:9877
127.0.0.1:9877
There are 2 local addresses and they are:
10.1.1.5:1025
127.0.0.1:1025
From str:0 seq:0 (assoc:c99e2680):[0]Hi
Control-D
FreeBSD-lap:
```

23.8 Encontrando um ID de associação dado um endereço IP

Nas alterações recentes que fizemos no nosso cliente na Seção 23.7, o cliente utilizava a notificação de associação para desencadear uma recuperação da lista de endereços. Essa notificação foi bem conveniente, visto que armazenou a identificação da associação no campo `sac_assoc_id`. Mas, se a aplicação não estiver monitorando as identificações da associação e tiver apenas o endereço de um peer, como ela poderá encontrar a identificação da associação? Na Figura 23.13, ilustramos uma função simples que converte um endereço do peer em um ID de associação. O servidor utilizará essa função mais adiante na Seção 23.10.

Inicialização

7–8 Primeiro, nossa função inicializa sua estrutura `sctp_paddrparams`.

Copiando o endereço

9 Copiamos o endereço, utilizando o comprimento que foi passado, para a estrutura `sctp_paddrparams`.

Chamando a opção de soquete

10 A função agora utiliza a opção de soquete `SCTP_PEER_ADDR_PARAMS` para solicitar os parâmetros de endereço do peer. Observe que utilizamos `sctp_opt_info`, em vez de `getsockopt`, uma vez que a opção de soquete `SCTP_PEER_ADDR_PARAMS` requer copiar argumentos para dentro e para fora do kernel. Essa chamada retornará o intervalo do heartbeat atual, o número máximo de retransmissões antes que a implementação SCTP considere o endereço do peer como falho e, acima de tudo, o ID de associação. Observe que não verificamos o valor de retorno, visto que, se a chamada falhar, queremos retornar 0.

sctp/sctp_addr_to_associd.c

```
 1 #include   "unp.h"

 2 sctp_assoc_t
 3 sctp_address_to_associd(int sock_fd, struct sockaddr *sa, socklen_t salen)
 4 {
 5     struct sctp_paddrparams sp;
 6     int    siz;

 7     siz = sizeof(struct sctp_paddrparams);
 8     bzero(&sp, siz);
 9     memcpy(&sp.spp_address, sa, salen);
10     sctp_opt_info(sock_fd, 0, SCTP_PEER_ADDR_PARAMS, &sp, &siz);
11     return (sp.spp_assoc_id);
12 }
```

sctp/sctp_addr_to_associd.c

Figura 23.13 Conversão de um endereço em um ID de associação.

11 A função retorna o ID de associação ao chamador. Observe que, se a chamada falhar, a limpeza anterior da estrutura garantirá que o nosso chamador obterá um 0 como o ID da associação retornado. Um ID de associação em 0 não é permitido e é utilizado também para indicar nenhuma associação pela implementação SCTP.

23.9 O mecanismo de heartbeat e falha de endereço

O SCTP fornece um mecanismo de heartbeat semelhante em conceito à opção keep-alive do TCP. No caso do SCTP, porém, a opção é ativada por default. A aplicação pode controlar o heartbeat e configurar o limiar de erro para um endereço utilizando a mesma opção de soquete que vimos na Seção 23.8. O limiar de erro é o número de heartbeats ausentes ou os tempos-limite de retransmissão que devem ocorrer antes de um endereço de destino ser considerado inacessível. Quando o endereço de destino estiver acessível novamente, detectado pelos heartbeats, o endereço torna-se ativo.

A aplicação pode desativar os heartbeats, mas sem estes o SCTP não tem como detectar se um endereço falho do peer tornou-se acessível novamente. Esses endereços não podem retornar a um estado ativo sem a intervenção do usuário.

O campo de parâmetro heartbeat da estrutura `sctp_paddrparams` é `spp_hbinterval`. Se uma aplicação configurar o campo `spp_hbinterval` como `SCTP_NO_HB` (`0`), os heartbeats serão desativados. Um valor de `SCTP_ISSUE_HB` (`0xffffffff`) solicita um heartbeat por demanda (imediato). Qualquer outro valor configura o retardo do heartbeat em milissegundos. O retardo do heartbeat fornece um retardo configurado entre os heartbeats. Esse valor, adicionado ao valor atual do timer de retransmissão mais uma oscilação aleatória, tornar-se-á a quantidade de tempo entre os heartbeats. Na Figura 23.14 mostramos uma pequena função que irá configurar o retardo do heartbeat, solicitar um heartbeat por demanda ou desativar o heartbeat para o destino especificado. Observe que, deixando o parâmetro de retransmissões, o campo `spp_pathmaxrxt` da estrutura `sctp_paddrparams`, configurado como 0, o valor atual permanece inalterado.

sctp/sctp_modify_hb.c

```
 1 #include   "unp.h"
 2 int
 3 heartbeat_action(int sock_fd, struct sockaddr *sa, socklen_t salen,
 4                   u_int value)
 5 {
 6     struct sctp_paddrparams sp;
 7     int    siz;
 8     bzero(&sp, sizeof(sp));
 9     sp.spp_hbinterval = value;
10     memcpy((caddr_t) & sp.spp_address, sa, salen);
11     Setsockopt(sock_fd, IPPROTO_SCTP,
12                SCTP_PEER_ADDR_PARAMS, &sp, sizeof(sp));
13     return (0);
14 }
```

sctp/sctp_modify_hb.c

Figura 23.14 Função utilitária de controle de heartbeat.

Zerando a estrutura `sctp_paddrparams` e copiando o intervalo

8–9 Zeramos `struct sctp_paddrparams` para assegurar que não iremos alterar nenhum parâmetro que não queremos. Em seguida, copiamos o valor do heartbeat desejado do usuário: `SCTP_ISSUE_HB`, `SCTP_NO_HB` ou um intervalo do heartbeat.

Configurando o endereço

10 A função configura o endereço e o copia para a estrutura `sctp_paddrparams` de modo que a implementação SCTP conhecerá o endereço ao qual desejamos enviar um heartbeat.

Executando a ação

11–12 Por fim, a função emite a chamada à opção de soquete para executar a ação que o usuário solicitou.

23.10 Extraindo uma associação

Focalizamos a interface do estilo um para muitos fornecida pelo SCTP. Essa interface tem várias vantagens em relação ao estilo um para um mais clássico:

- Há somente um descritor de arquivo a manter.
- Ela permite escrever um servidor iterativo simples.
- Ela deixa que uma aplicação envie dados no terceiro e no quarto pacotes do handshake de quatro vias utilizando `sendmsg` ou `sctp_sendmsg` para estabelecer a conexão implicitamente.
- Não há necessidade de monitorar o estado do transporte. Em outras palavras, a aplicação faz uma chamada de recebimento no descritor de soquete e não precisa fazer quaisquer chamadas de função `connect` ou `accept` tradicionais antes de receber as mensagens.

Entretanto, há uma desvantagem importante nesse estilo. Ele torna difícil construir um servidor concorrente (utilizando threads ou bifurcando os filhos). Essa desvantagem conduziu à adição da função `sctp_peeloff`. Essa função aceita um descritor de soquete de um para muitos e um ID de associação e retorna um novo descritor de soquete apenas com essa associação (além de quaisquer notificações e dados enfileirados nessa associação) anexada em um estilo um para um. O soquete original permanece aberto e quaisquer outras associações representadas pelo soquete de um para muitos permanecem inalteradas.

Esse soquete pode então ser passado para um thread ou um processo-filho a fim de executar um servidor concorrente. A Figura 23.15 ilustra uma outra modificação no nosso servidor que processa a primeira mensagem de um cliente, extrai o descritor de soquete do cliente utilizando `sctp_peeloff`, bifurca um filho e chama nossa função `str_echo` do TCP original introduzida na Seção 5.3. Utilizamos o endereço da mensagem recebida para chamar nossa função que obtém o ID da associação (Seção 23.8). O ID da associação também está disponível em `sri.sinfo_assoc_id`; mostramos esse método de determinar o ID da associação do endereço IP para ilustrar um outro método. Depois de bifurcar o filho, nosso servidor faz um loop novamente para processar a próxima mensagem.

sctp/sctpserv_fork.c

```
23      for ( ; ; ) {
24          len = sizeof(struct sockaddr_in);
25          rd_sz = Sctp_recvmsg(sock_fd, readbuf, sizeof(readbuf),
26                               (SA *) &cliaddr, &len, &sri, &msg_flags);
27          Sctp_sendmsg(sock_fd, readbuf, rd_sz,
28                       (SA *) &cliaddr, len,
29                       sri.sinfo_ppid,
30                       sri.sinfo_flags, sri.sinfo_stream, 0, 0);
31          assoc = sctp_address_to_associd(sock_fd, (SA *) &cliaddr, len);
32          if ((int) assoc == 0) {
33              err_ret("Can't get association id");
34              continue;
35          }
```

Figura 23.15 Um servidor SCTP concorrente (*continua*).

```
36          connfd = sctp_peeloff(sock_fd, assoc);
37          if (connfd == -1) {
38              err_ret("sctp_peeloff fails");
39              continue;
40          }
41          if ( (childpid = fork()) == 0) {
42              Close(sock_fd);
43              str_echo(connfd);
44              exit(0);
45          } else {
46              Close(connfd);
47          }
48      }
```

sctp/sctpserv_fork.c

Figura 23.15 Um servidor SCTP concorrente (*continuação*).

Recebendo e processando a primeira mensagem do cliente

26–30 O servidor recebe e processa a primeira mensagem que um cliente envia.

Convertendo o endereço em um ID de associação

31–35 Em seguida, o servidor utiliza a nossa função da Figura 23.13 para converter o endereço em um ID de associação. Se por alguma razão o servidor não puder obter um ID de associação, ele pula essa tentativa de bifurcar um filho e, em vez disso, tentará a próxima mensagem.

Extraindo a associação

36–40 O servidor extrai a associação para seu próprio descritor de soquete com `sctp_peeloff`. Isso resulta em um soquete de um para um que pode ser passado para a nossa versão do TCP anterior de `str_echo`.

Delegando trabalho ao filho

41–47 O servidor bifurca um filho e deixa que ele realize todo o trabalho futuro nesse novo descritor de soquete.

23.11 Controlando a sincronização

O SCTP tem vários controles que são ajustáveis pelo usuário. Todos esses controles avançados são acessados via as opções de soquete que discutimos na Seção 7.10. Nesta seção, destacaremos alguns controles específicos que influenciam o tempo que uma extremidade do SCTP levará para declarar uma falha de associação ou de destino.

Há sete controles específicos que ditam o tempo de detecção de falha no SCTP (Figura 23.16).

campo	Descrição	default	unidade
`srto_min`	Tempo-limite mínimo de retransmissão	1000	milissegundos
`srto_max`	Tempo-limite máximo de retransmissão	60000	milissegundos
`srto_initial`	Tempo-limite inicial de retransmissão	3000	milissegundos
`sinit_max_init_timeo`	Tempo-limite máximo de retransmissão durante INIT	3000	milissegundos
`sinit_max_attempts`	Retransmissões máximas de INIT	8	tentativas
`spp_pathmaxrxt`	Retransmissões máximas por endereço	5	tentativas
`sasoc_asocmaxrxt`	Retransmissões máximas por associação	10	tentativas

Figura 23.16 Campos que controlam a sincronização no SCTP.

Cada um desses parâmetros influencia a rapidez com que o SCTP detectará uma falha ou uma tentativa de retransmissão. Podemos pensar nesses parâmetros como "maçanetas" de controle que encurtam ou prolongam o tempo que leva para uma extremidade detectar uma falha. Primeiro, examinaremos dois cenários:

1. Uma extremidade do SCTP tenta abrir uma associação para um peer desconectado da rede.
2. Duas extremidades do SCTP multihomed estão trocando dados e uma delas foi desconectada no meio da comunicação. Nenhuma mensagem ICMP é recebida devido à filtragem por um firewall.

No Cenário 1, o sistema tentando abrir a conexão primeiro configuraria seu timer RTO com o valor `srto_initial` de 3.000 ms. Depois do tempo-limite, ele iria retransmitir a mensagem INIT e dobrar o timer RTO para 6.000 ms. Esse comportamento continuaria até que ele tivesse enviado `sinit_max_attempts`, ou oito mensagens INIT e, subseqüentemente, expirasse em cada uma das transmissões. A duplicação do timer RTO pararia em `sinit_max_init_timeo` ou 60.000 ms. Portanto, levaria 3+6+12+24+48+60+60+60, ou 273 segundos, para alcançar o ponto em que a implementação SCTP declararia o peer como potencialmente inacessível.

Há um número de ajustes e combinações de ajustes que podemos fazer para encurtar ou prolongar esse tempo. Primeiro, vamos focalizar a influência de dois parâmetros específicos que podemos utilizar para encurtar o tempo de 270 segundos. Uma das alterações que podemos fazer é diminuir o número de retransmissões alterando `sinit_max_attempts`. Uma alteração alternativa que também pode ser feita é reduzir o valor máximo do RTO para INIT alterando `srto_max_init_timeo`. Se reduzirmos o número de tentativas a quatro, nosso tempo de detecção de falhas cai drasticamente para 45 segundos, um sexto daquilo que o valor default nos oferece. Mas esse método tem uma desvantagem: podemos passar por uma situação em que nosso peer está disponível mas, devido à perda na rede ou talvez a uma sobrecarga no peer, declaramos o peer como inacessível.

Outra abordagem é reduzir o `srto_max_init_timeo` a 20 segundos. Isso diminui nosso tempo de detecção de falhas para 121 segundos, menos que a metade do valor original, mas essa alteração também leva a uma relação de compromisso. Se selecionarmos um valor muito baixo, é possível que um retardo excessivo na rede faça com que seja enviado um número bem maior de mensagens INIT do que o necessário.

Agora, vamos voltar nossa atenção ao Cenário 2, em que há dois peers multihomed que se comunicam entre si. Uma extremidade tem os endereços IP-A e IP-B, a outra IP-X e IP-Y. Se uma delas torna-se inacessível (pressupondo que os dados estavam sendo enviados do peer que não estava desconectado), a extremidade emissora verá sucessivos tempos-limite excedidos para cada destino iniciando em um valor de `srto_min` (default de 1 segundo) e dobrando até que ambos os destinos alcancem `srto_max` (default de 60 segundos). A extremidade iria retransmitir até alcançar o `sasoc_asocmaxrxt` máximo da associação (default de 10 retransmissões).

Agora, no nosso cenário, a extremidade emissora veria tempos-limite em 1(IP-A) + 1(IP-B) + 2(IP-A) + 2(IP-B) + 4(IP-A) + 4(IP-B) + 8(IP-A) + 8(IP-B) + 16(IP-A) + 16(IP-B), totalizando 62 segundos. O parâmetro `srto_max` não influencia um peer multihomed se ele permanecer com o seu valor default, uma vez que alcançamos o valor default de `sasoc_asocmaxrxt` antes de alcançarmos `srto_max`. Novamente, focalizamos dois parâmetros que podemos utilizar para afetar esses tempos-limite e a resultante detecção de falhas. Podemos diminuir o número de tentativas alterando o valor `sasoc_asocmaxrxt` (default de 10) ou podemos diminuir o RTO máximo alterando `srto_max` (default de 60 segundos). Se configurarmos nosso tempo `srto_max` como 10 segundos, poderemos diminuir o tempo de detecção em 12 segundos, reduzindo-o a 50 segundos. Uma alternativa, diminuir as re-

transmissões máximas a oito, reduziria o nosso tempo de detecção para 30 segundos. As mesmas preocupações que mencionamos anteriormente também se aplicam a esse cenário: um problema de rede, breve mas superável, ou uma sobrecarga no sistema remoto poderia fazer com que uma conexão funcional fosse derrubada.

Entre as muitas alternativas, não recomendamos reduzir o RTO mínimo (`srto_min`). Na comunicação pela Internet, reduzir esse valor poderia ter conseqüências terríveis pelo fato de que iríamos retransmitir muito mais rapidamente, esgotando a infra-estrutura da Internet. Em uma rede privada, talvez seja aceitável ajustar esse valor para baixo, mas para a maioria das aplicações esse valor não deve ser diminuído.

Cada aplicação, ao fazer esses ajustes de sincronização, deve antes levar em consideração vários fatores:

- Quão rápido sua aplicação precisa detectar uma falha?
- A aplicação será executada em redes privadas onde as condições no caminho geral de uma extremidade a outra são bem-conhecidas e variam menos do que na Internet?
- Quais são as conseqüências de uma falsa detecção de falha?

Somente depois de responder cuidadosamente a essas perguntas uma aplicação será capaz de ajustar adequadamente os parâmetros de sincronização do SCTP.

23.12 Quando utilizar o SCTP em vez do TCP

O SCTP foi originalmente desenvolvido para sinalização do controle de chamadas para permitir o transporte de sinais de telefonia pela Internet. Entretanto, durante o desenvolvimento, seu escopo foi expandido para um protocolo de transporte de uso geral. Ele fornece a maioria dos recursos do TCP e ainda acrescenta um amplo espectro de novos serviços na camada de transporte. São poucas as aplicações que não podem se beneficiar do uso do SCTP. Portanto, quando devemos utilizá-lo? Vamos iniciar listando os benefícios do SCTP:

1. O SCTP é um protocolo que suporta multihoming diretamente. Uma extremidade tira proveito de múltiplas redes em um host para ganhar confiabilidade adicional. Um bônus extra é o fato de que a aplicação não precisa realizar nenhuma ação, além de mudar-se para o SCTP, para automaticamente tirar proveito do serviço multihomed deste. Para mais detalhes sobre o multihoming do SCTP, consulte a Seção 7.4 de Stewart e Xie (2001).
2. O bloqueio de início de linha pode ser eliminado. Uma aplicação pode utilizar uma única associação SCTP e transportar múltiplos elementos de dados em paralelo. Uma perda em um dos fluxos de informações não influenciará nenhum outro fluxo paralelo de informações pela associação (discutimos esse conceito na Seção 10.5).
3. Os limites da mensagem na camada da aplicação são preservados. Muitas aplicações não enviam fluxos de bytes; em vez disso, elas enviam mensagens. O SCTP preserva os limites da mensagem enviada por uma aplicação e, assim, simplifica a tarefa do escritor da aplicação. Não é mais necessário marcar os limites da mensagem dentro de um fluxo de bytes e fornecer um código de tratamento especial para lidar com a reconstrução das mensagens a partir do fluxo de informações no receptor.
4. Um serviço de mensagem não-ordenado é fornecido. Para algumas aplicações, nenhum ordenamento é necessário. No passado, esse tipo de aplicação poderia utilizar o TCP pela sua confiabilidade com a desvantagem de que todos os dados, mesmo os não-ordenados, teriam de ser entregues em ordem. Qualquer perda causaria um bloqueio de início de linha para todas as mensagens subseqüentes que fluíssem pela conexão. Com o SCTP, há um serviço não-ordenado que evita esse problema e permite que uma aplicação tenha suas necessidades atendidas diretamente no transporte.

5. Um serviço parcialmente confiável está disponível em algumas implementações SCTP. Esse recurso permite a um emissor do SCTP especificar o tempo de vida em cada mensagem, utilizando o campo `sinfo_timetolive` da `struct sctp_sndrcvinfo`. (Isso é diferente do TTL do IPv4 ou do limite de hops do IPv6; na verdade, é um intervalo de tempo.) Se as duas extremidades suportarem esse recurso, dados sensíveis à data/hora podem ser descartados pelo transporte em vez de pela aplicação, mesmo se foram transmitidos e perdidos, otimizando assim o transporte dos dados em face de um congestionamento.
6. Um caminho de migração fácil no TCP é fornecido pelo SCTP com sua interface no estilo um para um. Essa interface duplica uma interface TCP típica de tal modo que, com uma ou duas pequenas alterações, uma aplicação TCP pode migrar para o SCTP.
7. Muitos recursos do TCP, como reconhecimento positivo, retransmissão de dados perdidos, novo seqüenciamento de dados, controle de fluxo em janelas, início lento, prevenção de congestionamento e reconhecimentos seletivos, estão incluídos no SCTP, com duas exceções notáveis (o estado meio fechado e dados urgentes).
8. O SCTP fornece muitos ganchos (como visto neste capítulo e na Seção 7.10) para uma aplicação configurar, ajustar o transporte e ter suas necessidades atendidas individualmente por associação. Essa flexibilidade, juntamente com um conjunto geral de bons padrões (para a aplicação que não deseja ajustar o transporte), fornece à aplicação controles não-disponíveis no TCP.

O SCTP não fornece dois recursos do TCP. Um é o estado meio fechado, ou fechado pela metade (*half-closed state*). Entramos nesse estado quando uma aplicação fecha a sua metade da conexão, mas ainda permite ao peer enviar dados a ela (discutimos isso na Seção 6.6). Uma aplicação entra no estado meio fechado para sinalizar ao peer que ela terminou de transmitir os dados. Poucas aplicações utilizam esse recurso, dessa forma, durante o desenvolvimento do SCTP, considerou-se que não valia a pena adicioná-lo ao protocolo. As aplicações que precisam desse recurso e querem mudar para o SCTP precisarão alterar o protocolo da sua camada de aplicação para fornecer esse sinal no fluxo de dados da aplicação. Em algumas instâncias, essa alteração talvez não seja trivial.

Um outro recurso do TCP que o SCTP não fornece são dados urgentes. Utilizar um fluxo SCTP separado para dados urgentes apresenta uma semântica um pouco semelhante, mas não pode replicar o recurso de maneira exata.

Um outro tipo de aplicação que talvez não se beneficie do SCTP é uma aplicação verdadeiramente baseada em fluxo de bytes, como `telnet`, `rlogin`, `rsh` e `ssh`. Para essa aplicação, o TCP pode segmentar o fluxo de bytes em pacotes IP de maneira mais eficiente que o SCTP. O SCTP preservará fielmente os limites da mensagem, o que talvez se equipare a um tamanho que não se ajusta eficientemente nos datagramas IP e, portanto, pode causar mais overheads.

Em resumo, muitas aplicações poderiam considerar a utilização do SCTP à medida que ele se torna disponível para plataformas Unix. Entretanto, os recursos especiais do SCTP demandam muita atenção para verdadeiramente se tirar proveito deles; até que o SCTP se torne onipresente, poderia ser vantajoso simplesmente permanecer com o TCP.

23.13 Resumo

Neste capítulo, examinamos o recurso de autofechamento do SCTP, explorando como ele pode ser utilizado para limitar associações desocupadas em um soquete de um para muitos. Construímos um utilitário simples que uma aplicação pode utilizar para receber grandes mensagens com a API de entrega parcial. Examinamos como uma aplicação pode decodificar os eventos que ocorrem no transporte com um recurso simples que exibe notificações. Examina-

mos brevemente como um usuário pode enviar dados não-ordenados e vincular um subconjunto de endereços. Vimos como adquirir os endereços tanto da extremidade do peer como da extremidade local de uma associação. Também examinamos um método simples que uma aplicação pode utilizar para converter um endereço em um ID de associação.

Heartbeats (chamados keep-alives no TCP) são trocados por default em uma associação SCTP. Examinamos como controlar esse recurso por meio de um pequeno utilitário que construímos. Vimos como extrair uma associação com a chamada de sistema `sctp_peeloff` e ilustramos com um exemplo de servidor que é iterativo e concorrente utilizando essa chamada. Também discutimos as considerações que uma aplicação precisa fazer antes de ajustar os parâmetros de sincronização do SCTP. Concluímos com um exame sobre quando uma aplicação deve considerar a utilização do SCTP.

Exercícios

23.1 Escreva um cliente que possa testar a API de entrega parcial de nosso servidor que desenvolvemos na Seção 23.3.

23.2 Além de enviar uma mensagem muito grande ao servidor ilustrado na Seção 23.3, que outro método pode ser utilizado para obter a API de entrega parcial invocada no nosso servidor?

23.3 Reescreva o servidor da API de entrega parcial para tratar notificações da mesma.

23.4 Quais aplicações se beneficiariam do uso de dados não-ordenados? Quais aplicações não se beneficiariam de dados não-ordenados? Justifique suas escolhas.

23.5 Como você pode testar o servidor de vinculação de subconjunto?

23.6 Suponha que sua aplicação esteja em execução em uma rede privada em que todas as extremidades estão conectadas via uma rede local. Também suponha que todos os seus servidores e clientes estejam em execução em hosts multihomed. Que parâmetros você precisa ajustar para garantir a detecção de uma falha em dois ou menos segundos?

Capítulo **24**

Dados Fora da Banda

24.1 Visão geral

Muitas camadas de transporte têm o conceito de dados *fora da banda*, que às vezes são chamados de *dados apressados (expedited data)*. A idéia é a de que algo importante ocorre em uma extremidade de uma conexão e essa extremidade quer se comunicar com seu peer rapidamente. Neste caso, "rapidamente" significa que essa notificação deve ser enviada antes de quaisquer dados "normais" (às vezes chamados de "dentro da banda") que já estejam enfileirados para serem enviados, o que deve ser feito independentemente de quaisquer problemas de controle de fluxo ou de bloqueio. Isto é, os dados fora da banda são considerados de prioridade mais alta que os dados normais. Em vez de utilizar duas conexões entre o cliente e o servidor, os dados fora da banda são mapeados na conexão existente.

Infelizmente, quando saímos dos conceitos gerais e entramos no mundo real, quase toda camada de transporte tem uma implementação diferente de dados fora da banda. Como um exemplo extremo, o UDP não tem nenhuma implementação de dados fora da banda. Neste capítulo, focalizaremos o modelo de dados fora da banda do TCP, daremos numerosos exemplos de como eles são tratados pela API de soquetes e descreveremos como são utilizados por aplicações como `telnet`, `rlogin` e FTP. Além das aplicações remotas interativas como essas, é raro encontrar qualquer utilização para dados fora da banda.

24.2 Dados TCP fora da banda

O TCP não tem *dados fora da banda* verdadeiros. Em vez disso, o protocolo TCP fornece um *modo urgente*. Suponha que um processo tenha gravado *N* bytes de dados em um soquete TCP e que esses dados sejam enfileirados pelo TCP no buffer de envio do soquete, esperando para serem enviados para o peer. Mostramos isso na Figura 24.1 e rotulamos os bytes de dados de 1 a *N*.

Agora, o processo grava um único byte de dados fora da banda, contendo o caractere ASCII `a`, utilizando a função `send` e o flag `MSG_OOB`.

```
send(fd, "a", 1, MSG_OOB);
```

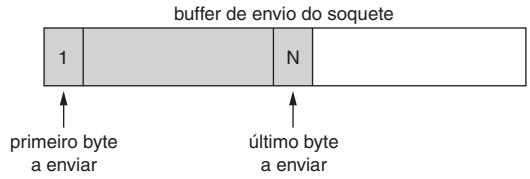


Figura 24.1 Buffer de envio do soquete contendo dados a enviar.

O protocolo TCP coloca os dados na próxima posição disponível no buffer de envio do soquete e configura seu *ponteiro urgente* para que essa conexão seja a próxima localização disponível. Mostramos isso na Figura 24.2 e rotulamos o byte fora da banda como "OOB".

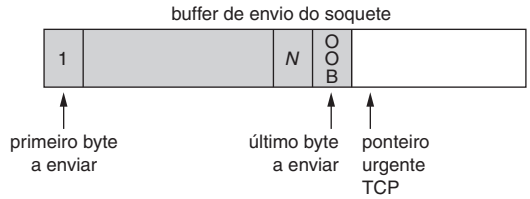


Figura 24.2 Buffer de envio do soquete depois que um byte de dados fora da banda é gravado pela aplicação.

> O ponteiro urgente TCP tem um número de seqüência uma unidade maior que o byte de dados que é gravado com o flag `MSG_OOB`. Conforme discutido nas páginas 292 a 296 do TCPv1, esse é um artefato histórico que agora é emulado por todas as implementações. Contanto que os protocolos TCP emissor e TCP receptor concordem com a interpretação de ponteiro urgente TCP, tudo correrá bem.

Dado o estado do buffer de envio do soquete TCP mostrado na Figura 24.2, o próximo segmento enviado pelo protocolo TCP terá seu flag URG ativado no cabeçalho TCP; e o campo de deslocamento urgente no cabeçalho TCP apontará para o byte seguinte ao byte fora da banda. Mas esse segmento pode ou não conter o byte que rotulamos como OOB. O envio do byte OOB depende do número de bytes à frente dele no buffer de envio do soquete, do tamanho de segmento que o protocolo TCP está enviando para o peer e da janela corrente anunciada pelo peer.

> Utilizamos os termos *ponteiro urgente* e *deslocamento urgente (urgent offset)*. No nível TCP os dois são diferentes. O valor de 16 bits no cabeçalho TCP é chamado de deslocamento urgente e deve ser adicionado ao campo de número de seqüência no cabeçalho para se obter o ponteiro urgente de 32 bits. O protocolo TCP vê o deslocamento urgente somente se outro bit no cabeçalho estiver configurado; e esse bit é chamado de *flag URG*. Da perspectiva da programação, não precisamos nos preocupar com esse detalhe e nos referimos somente ao ponteiro urgente do protocolo TCP.

Essa é uma característica importante do modo urgente do TCP: o cabeçalho TCP indica que o emissor entrou no modo urgente (isto é, o flag URG é ativado junto ao deslocamento urgente), mas o byte de dados real referido pelo ponteiro urgente não precisa ser enviado. De fato, se o protocolo TCP emissor é interrompido pelo controle de fluxo (o buffer de recepção do soquete do receptor está cheio, de modo que seu protocolo TCP anunciou uma janela 0 para o TCP emissor), a notificação urgente é enviada sem quaisquer dados (páginas 1016 e 1017 do TCPv2), como mostraremos nas Figuras 24.10 e 24.11. Essa é uma razão pela qual as aplicações utilizam o modo urgente do protocolo TCP (isto é, dados fora da banda): a notificação urgente é *sempre* enviada para o TCP do peer, mesmo que o fluxo de dados seja interrompido pelo controle de fluxo do protocolo TCP.

O que acontece se enviamos vários bytes de dados fora da banda, como em

```
send(fd, "abc", 3, MSG_OOB);
```

Nesse exemplo, o ponteiro urgente do TCP aponta para uma unidade além do byte final; isto é, o byte final (o `c`) é considerado o byte fora da banda.

Agora que abordamos o envio de dados fora da banda, vejamos isso no lado do receptor:

1. Quando o protocolo TCP recebe um segmento com o flag URG configurado, o ponteiro urgente é examinado para ver se ele referencia *novos* dados fora da banda; isto é, se essa é a primeira vez que o modo urgente do protocolo TCP referenciou esse byte em particular no fluxo de dados do emissor para o receptor. É comum o protocolo TCP emissor transmitir vários segmentos (em geral, em um curto período de tempo) contendo o flag URG, mas com o ponteiro urgente apontando para o mesmo byte de dados. Somente o primeiro desses segmentos faz o processo receptor ser notificado de que novos dados fora da banda chegaram.
2. O processo receptor é notificado quando um novo ponteiro urgente chega. Primeiro, o sinal `SIGURG` é enviado para o proprietário do soquete, supondo que `fcntl` ou `ioctl` tenha sido chamado para estabelecer um proprietário para o soquete (Figura 7.20). Segundo, se o processo é bloqueado em uma chamada a `select`, esperando que esse descritor de soquete tenha uma condição de exceção, `select` retorna.

   Essas duas notificações em potencial para o processo receptor acontecem quando um novo ponteiro urgente chega, independentemente de o byte de dados real apontado pelo ponteiro urgente ter chegado no TCP receptor.

   Há somente uma marca OOB; se um novo byte OOB chega, antes que o byte OOB antigo tenha sido lido, o byte antigo é descartado.
3. Quando o byte de dados real apontado pelo ponteiro urgente chega no TCP receptor, o byte de dados pode ser extraído de fora da banda ou deixado em linha. Por default, a opção de soquete `SO_OOBINLINE` *não* é configurada para um soquete; portanto, o único byte de dados não é colocado no buffer de recepção do soquete. Em vez disso, o byte de dados é colocado em um buffer fora da banda separado, de um byte, para essa conexão (páginas 986 a 988 do TCPv2). A única maneira de o processo ler esse buffer de um byte especial é chamar `recv`, `recvfrom` ou `recvmsg` e especificar o flag `MSG_OOB`. Se um novo byte OOB chega, antes que o byte antigo seja lido, o valor anterior desse buffer é descartado.

   Entretanto, se o processo configura a opção de soquete `SO_OOBINLINE`, o único byte de dados referenciado pelo ponteiro urgente do TCP é deixado no buffer de recepção do soquete normal. Nesse caso, o processo não pode especificar o flag `MSG_OOB` para ler o byte de dados. O processo saberá quando chegar nesse byte de dados, verificando a *marca de fora da banda* dessa conexão, conforme descreveremos na Seção 24.3.

Alguns dos erros a seguir são possíveis:

1. Se o processo solicitar dados fora da banda (por exemplo, especificando o flag `MSG_OOB`), mas o peer não tiver enviado nenhum, `EINVAL` é retornado.
2. Se o processo tiver sido notificado de que o peer enviou um byte fora da banda (por exemplo, por `SIGURG` ou `select`) e tentar lê-lo, mas esse byte ainda não tiver chegado, `EWOULDBLOCK` será retornado. Tudo que o processo pode fazer nesse ponto é ler o buffer de recepção do soquete (possivelmente descartando os dados, caso não tenha espaço para armazená-los), a fim de criar espaço no buffer para que o TCP peer possa enviar o byte fora da banda.

3. Se o processo tentar ler o mesmo byte fora da banda várias vezes, EINVAL é retornado.
4. Se o processo tiver configurado a opção de soquete SO_OOBINLINE e então tentar ler os dados fora da banda especificando MSG_OOB, EINVAL é retornado.

Exemplo simples utilizando SIGURG

Mostraremos agora um exemplo simples de envio e recebimento de dados fora da banda. A Figura 24.3 mostra o programa de envio.

Nove bytes são enviados, com a execução de sleep durante um segundo entre cada operação de saída. O objetivo da pausa é permitir que cada função write ou send seja transmitida como um único segmento TCP e recebida como tal pelo outro lado. Falaremos posteriormente sobre algumas considerações de sincronização com dados fora da banda. Quando executamos esse programa, vemos a saída esperada.

```
macosx % tcpsend01 freebsd4 9999
wrote 3 bytes of normal data
wrote 1 byte of OOB data
wrote 2 bytes of normal data
wrote 1 byte of OOB data
wrote 2 bytes of normal data
```

oob/tcpsend01.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    sockfd;
 6     if(argc != 3)
 7         err_quit("usage: tcpsend01 <host> <port#>");
 8     sockfd = Tcp_connect(argv[1], argv[2]);
 9     Write(sockfd, "123", 3);
10     printf("wrote 3 bytes of normal data\n");
11     sleep(1);
12     Send(sockfd, "4", 1, MSG_OOB);
13     printf("wrote 1 byte of OOB data\n");
14     sleep(1);
15     Write(sockfd, "56", 2);
16     printf("wrote 2 bytes of normal data\n");
17     sleep(1);
18     Send(sockfd, "7", 1, MSG_OOB);
19     printf("wrote 1 byte of OOB data\n");
20     sleep(1);
21     Write(sockfd, "89", 2);
22     printf("wrote 2 bytes of normal data\n");
23     sleep(1);
24     exit(0);
25 }
```

oob/tcpsend01.c

Figura 24.3 Programa de envio fora da banda simples.

A Figura 24.4 é o programa receptor.

oob/tcprecv01.c

```
 1 #include   "unp.h"
 2 int    listenfd, connfd;
 3 void   sig_urg(int);
 4 int
 5 main(int argc, char **argv)
 6 {
 7    int    n;
 8    char   buff[100];
 9    if(argc == 2)
10       listenfd = Tcp_listen(NULL, argv[1], NULL);
11    else if (argc == 3)
12       listenfd = Tcp_listen(argv[1], argv[2], NULL);
13    else
14        err_quit("usage: tcprecv01 [ <host> ] <port#>");
15    connfd = Accept(listenfd, NULL, NULL);
16    Signal(SIGURG, sig_urg);
17    Fcntl(connfd, F_SETOWN, getpid());
18    for ( ; ; ) {
19        if ( (n = Read(connfd, buff, sizeof(buff) - 1)) == 0) {
20            printf("received EOF\n");
21            exit(0);
22        }
23        buff[n] = 0;              /* termina com nulo */
24        printf("read %d bytes: %s\n", n, buff);
25    }
26 }
27 void
28 sig_urg(int signo)
29 {
30    int    n;
31    char   buff[100];
32    printf("SIGURG received\n");
33    n = Recv(connfd, buff, sizeof(buff) - 1, MSG_OOB);
34    buff[n] = 0;                  /* termina com nulo */
35    printf("read %d OOB byte: %s\n", n, buff);
36 }
```

oob/tcprecv01.c

Figura 24.4 Programa receptor fora da banda simples.

Estabelecimento do handler de sinal e do proprietário de soquete

16–17 O handler de sinal para `SIGURG` é estabelecido e `fcntl` configura o proprietário do soquete conectado.

> Observe que não estabelecemos o handler de sinal até que `accept` retorne. Há uma pequena probabilidade de que dados fora da banda possam chegar depois que nosso protocolo TCP conclua o handshake de três vias, mas antes que `accept` retorne, o que perderíamos. Mas se estabelecêssemos o handler de sinal antes de chamar `accept` e também configurássemos o proprietário do soquete receptor (que transporta para o soquete conectado), então, se dados fora da banda chegarem antes de `accept` retornar, nosso handler de sinal ainda não teria um valor para `connfd`. Se esse cenário é importante para a aplicação, ela deve inicializar `connfd` como –1, verificar esse valor no handler de sinal e, se for verdadeiro, basta configurar um flag para o loop principal, a fim de verificar depois de `accept` retornar. Como alternativa, a aplicação poderia bloquear o sinal em torno da chamada a `accept`, mas isso está sujeito a todas as condições de corrida de sinal que discutimos na Seção 20.5.

18–25 O processo lê o soquete, imprimindo cada string retornada por `read`. Quando o emissor termina a conexão, o receptor termina.

handler `SIGURG`

27–36 Nosso handler de sinal chama `printf`, lê o byte fora da banda especificando o flag `MSG_OOB` e, então, imprime os dados retornados. Observe que solicitamos até 100 bytes na chamada a `recv`, mas, conforme veremos brevemente, somente 1 byte é retornado como dado fora da banda.

> Conforme declarado anteriormente, chamar a função insegura `printf` a partir de um handler de sinal não é recomendado. Fazemos isso apenas para ver o que está acontecendo com nossos programas.

Eis a saída de quando executamos o programa receptor e então o programa de envio da Figura 24.3:

```
freebsd4 % tcprecv01 9999
read 3 bytes: 123
SIGURG received
read 1 OOB byte: 4
read 2 bytes: 56
SIGURG received
read 1 OOB byte: 7
read 2 bytes: 89
received EOF
```

Os resultados são conforme o esperado. Cada envio de dados fora da banda pelo emissor gera um sinal `SIGURG` para o receptor, o qual lê então o único byte fora da banda.

Exemplo simples utilizando `select`

Agora, refazemos nosso receptor fora da banda para utilizar `select` em vez do sinal `SIGURG`. A Figura 24.5 é o programa receptor.

15–20 O processo chama `select` enquanto espera por dados normais (o conjunto de leitura, `rset`) ou dados fora da banda (o conjunto de exceção, `xset`). Em cada caso, os dados recebidos são impressos.

Quando executamos esse programa e então o mesmo programa de envio anterior (Figura 24.3), encontramos o seguinte erro:

```
freebsd4 % tcprecv02 9999
read 3 bytes: 123
read 1 OOB byte: 4
recv error: Invalid argument
```

O problema é que `select` indica uma condição de exceção até que o processo leia *além* dos dados fora da banda (páginas 530 a 531 do TCPv2). Não podemos ler os dados fora da banda mais do que uma vez porque, depois que lemos a primeira vez, o kernel limpa o buffer fora da banda de um byte. Quando chamamos `recv` especificando o flag `MSG_OOB` na segunda vez, ele retorna `EINVAL`.

oob/tcprecv02.c

```
1 #include   "unp.h"

2 int
3 main(int argc, char **argv)
4 {
```

Figura 24.5 Programa receptor que usa `select` (incorretamente) para ser notificado de dados fora da banda (*continua*).

```
 5     int    listenfd, connfd, n;
 6     char   buff[100];
 7     fd_set rset, xset;

 8     if(argc == 2)
 9         listenfd = Tcp_listen(NULL, argv[1], NULL);
10     else if (argc == 3)
11         listenfd = Tcp_listen(argv[1], argv[2], NULL);
12     else
13         err_quit("usage: tcprecv02 [ <host> ] <port#>");

14     connfd = Accept(listenfd, NULL, NULL);

15     FD_ZERO(&rset);
16     FD_ZERO(&xset);
17     for ( ; ; ) {
18         FD_SET(connfd, &rset);
19         FD_SET(connfd, &xset);

20         Select(connfd + 1, &rset, NULL, &xset, NULL);

21         if (FD_ISSET(connfd, &xset)) {
22             n = Recv(connfd, buff, sizeof(buff) - 1, MSG_OOB);
23             buff[n] = 0;          /* termina com nulo */
24             printf("read %d OOB byte: %s\n", n, buff);
25         }

26         if (FD_ISSET(connfd, &rset)) {
27             if ( (n = Read(connfd, buff, sizeof(buff) - 1)) == 0) {
28                 printf("received EOF\n");
29                 exit(0);
30             }
31             buff[n] = 0;          /* termina com nulo */
32             printf("read %d bytes: %s\n", n, buff);
33         }
34     }
35 }
```

oob/tcprecv02.c

Figura 24.5 Programa receptor que usa `select` (incorretamente) para ser notificado de dados fora da banda (*continuação*).

A solução é usar `select` para uma condição de exceção somente depois de ler dados normais. A Figura 24.6 é uma modificação da Figura 24.5 que trata desse cenário corretamente.

5 Declaramos uma nova variável chamada `justreadoob`, que indica se acabamos de ler dados fora da banda ou não. Esse flag determina se `select` é usado ou não para uma condição de exceção.

26–27 Quando configuramos o flag `justreadoob`, também devemos limpar o bit para esse descritor no conjunto de exceção.

O programa agora funciona conforme o esperado.

oob/tcprecv03.c

```
 1 #include   "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    listenfd, connfd, n, justreadoob = 0;
 6     char   buff[100];
 7     fd_set rset, xset;

 8     if(argc == 2)
```

Figura 24.6 Modificação da Figura 24.5 para `select`, para uma condição de exceção correta (*continua*).

```
 9          listenfd = Tcp_listen(NULL, argv[1], NULL);
10      else if (argc == 3)
11          listenfd = Tcp_listen(argv[1], argv[2], NULL);
12      else
13          err_quit("usage: tcprecv03 [ <host> ] <port#>");
14      connfd = Accept(listenfd, NULL, NULL);
15      FD_ZERO(&rset);
16      FD_ZERO(&xset);
17      for ( ; ; ) {
18          FD_SET(connfd, &rset);
19          if (justreadoob == 0)
20              FD_SET(connfd, &xset);
21          Select(connfd + 1, &rset, NULL, &xset, NULL);
22          if (FD_ISSET(connfd, &xset)) {
23              n = Recv(connfd, buff, sizeof(buff) - 1, MSG_OOB);
24              buff[n] = 0;  /* termina com nulo */
25              printf("read %d OOB byte: %s\n", n, buff);
26              justreadoob = 1;
27              FD_CLR(connfd, &xset);
28          }
29          if (FD_ISSET(connfd, &rset)) {
30              if ( (n = Read(connfd, buff, sizeof(buff) - 1)) == 0) {
31                  printf("received EOF\n");
32                  exit(0);
33              }
34              buff[n] = 0;          /* termina com nulo */
35              printf("read %d bytes: %s\n", n, buff);
36              justreadoob = 0;
37          }
38      }
39  }
```

oob/tcprecv03.c

Figura 24.6 Modificação da Figura 24.5 para `select`, para uma condição de exceção correta (*continuação*).

24.3 Função `sockatmark`

Sempre que dados fora da banda são recebidos, há uma *marca de fora da banda* associada. Essa é a posição no fluxo de dados normal *no emissor*, quando o processo de envio transmitiu o byte fora da banda. O processo receptor determina se está na marca de fora da banda ou não, chamando a função `sockatmark` enquanto lê do soquete.

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

int sockatmark(int sockfd);
```

Retorna: 1 se estiver na marca de fora da banda, 0 se não estiver na marca, –1 em caso de erro

Essa função é uma invenção do POSIX. O POSIX está substituindo muitas `ioctl`s por funções.

A Figura 24.7 mostra uma implementação dessa função utilizando a `ioctl SIOCATMARK` comumente encontrada.

lib/sockatmark.c

```
1 #include   "unp.h"
2 int
3 sockatmark(int fd)
4 {
5     int    flag;
6     if(ioctl(fd, SIOCATMARK, &flag) < 0)
7         return (-1);
8     return (flag != 0);
9 }
```

lib/sockatmark.c

Figura 24.7 Função `sockatmark` implementada utilizando `ioctl`.

A marca de fora da banda se aplica independentemente de o processo receptor estar recebendo os dados fora da banda em linha (a opção de soquete `SO_OOBINLINE`) ou fora da banda (o flag `MSG_OOB`). Uma utilização comum da marca de fora da banda é o receptor tratar todos os dados como especiais, até que a marca seja ultrapassada.

Exemplo

Mostraremos agora um exemplo simples para ilustrar os dois recursos da marca de fora da banda a seguir:

1. A marca de fora da banda sempre aponta para uma unidade além do byte de dados normais final. Isso significa que, se os dados fora da banda são recebidos em linha, `sockatmark` retorna verdadeiro, caso o próximo byte a ser lido seja o que foi enviado com o flag `MSG_OOB`. Como alternativa, se a opção de soquete `SO_OOBINLINE` não estiver ativada, então `sockatmark` retornará verdadeiro, caso o próximo byte de dados seja o primeiro byte que foi enviado após os dados fora da banda.
2. Uma operação de leitura sempre pára na marca de fora da banda (páginas 519 e 520 do TCPv2). Isto é, se há 100 bytes no buffer de recepção do soquete, mas somente 5 bytes até a marca de fora da banda e o processo executa um comando `read` solicitando 100 bytes, somente os 5 bytes até a marca são retornados. Essa parada forçada na marca serve para permitir que o processo chame `sockatmark` para determinar se o ponteiro de buffer está na marca.

A Figura 24.8 é nosso programa de envio. Ele envia três bytes de dados normais, um byte de dados fora da banda, seguidos por outro byte de dados normais. Não há nenhuma pausa entre cada operação de saída.

A Figura 24.9 é o programa receptor. Esse programa não utiliza o sinal `SIGURG` nem `select`. Em vez disso, ele chama `sockatmark` para determinar quando o byte fora da banda é encontrado.

oob/tcpsend04.c

```
1 #include   "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int    sockfd;
6     if(argc != 3)
7         err_quit("usage: tcpsend04 <host> <port#>");
8     sockfd = Tcp_connect(argv[1], argv[2]);
```

Figura 24.8 Programa de envio (*continua*).

```
 9      Write(sockfd, "123", 3);
10      printf("wrote 3 bytes of normal data\n");

11      Send(sockfd, "4", 1, MSG_OOB);
12      printf("wrote 1 byte of OOB data\n");

13      Write(sockfd, "5", 1);
14      printf("wrote 1 byte of normal data\n");

15      exit(0);
16  }
```

oob/tcpsend04.c

Figura 24.8 Programa de envio (*continuação*).

oob/tcprecv04.c

```
 1 #include   "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5      int     listenfd, connfd, n, on = 1;
 6      char    buff[100];

 7      if(argc == 2)
 8          listenfd = Tcp_listen(NULL, argv[1], NULL);
 9      else if (argc == 3)
10          listenfd = Tcp_listen(argv[1], argv[2], NULL);
11      else
12          err_quit("usage: tcprecv04 [ <host> ] <port#>");

13      Setsockopt(listenfd, SOL_SOCKET, SO_OOBINLINE, &on, sizeof(on));

14      connfd = Accept(listenfd, NULL, NULL);
15      sleep(5);

16      for ( ; ; ) {
17          if (Sockatmark(connfd))
18              printf("at OOB mark\n");

19          if ( (n = Read(connfd, buff, sizeof(buff) - 1)) == 0) {
20              printf("received EOF\n");
21              exit(0);
22          }
23          buff[n] = 0;              /* termina com nulo */
24          printf("read %d bytes: %s\n", n, buff);
25      }
26  }
```

oob/tcprecv04.c

Figura 24.9 Programa receptor que chama `sockatmark`.

Configuração da opção de soquete `SO_OOBINLINE`

13 Queremos receber os dados fora da banda em linha; portanto, devemos configurar a opção de soquete `SO_OOBINLINE`. Mas, se esperarmos até que `accept` retorne e configurarmos a opção no soquete conectado, o handshake de três vias estará concluído e dados fora da banda já poderão ter chegado. Portanto, devemos configurar essa opção para o soquete de recepção, sabendo que todas as opções de soquete transportam do soquete de recepção para o soquete conectado (Seção 7.4).

Dormir (`sleep`) depois da conexão aceita

14–15 O receptor dorme após a conexão ser aceita para permitir que todos os dados do emissor sejam recebidos. Isso nos permite demonstrar que um comando `read` pára na marca de fora da banda, mesmo que dados adicionais estejam no buffer de recepção do soquete.

Leitura de todos os dados do emissor

16–25 O programa chama `read` em um loop, imprimindo os dados recebidos. Mas, antes de chamar `read`, `sockatmark` verifica se o ponteiro de buffer está na marca de fora da banda.

Quando executamos esse programa, obtemos a seguinte saída:

```
freebsd4 % tcprecv04 6666
read 3 bytes: 123
at OOB mark
read 2 bytes: 45
received EOF
```

Mesmo que todos os dados tenham sido recebidos pelo TCP receptor, quando `read` é chamado pela primeira vez (porque o processo receptor chama `sleep`), somente três bytes são retornados, pois a marca é encontrada. O próximo byte lido é o byte fora da banda (com um valor igual a 4), porque dissemos ao kernel para que colocasse os dados fora da banda em linha.

Exemplo

Mostramos agora outro exemplo simples para ilustrar dois recursos adicionais de dados fora da banda, ambos mencionados anteriormente.

1. O protocolo TCP envia notificação de dados fora da banda (seu ponteiro urgente), mesmo que seja impedido de enviar dados pelo controle de fluxo.
2. Um processo receptor pode ser notificado de que o emissor enviou dados fora da banda (com o sinal `SIGURG` ou por meio de `select`) *antes* que estes cheguem. Se o processo chama então `recv`, especificando `MSG_OOB`, e os dados fora da banda não tiverem chegado, um erro de `EWOULDBLOCK` é retornado.

A Figura 24.10 é o programa de envio.

9–19 Esse processo configura o tamanho de seu buffer de envio do soquete como 32.768, grava 16.384 bytes de dados normais e então dorme por 5 segundos. Veremos em breve que o receptor configura o tamanho de seu buffer de recepção do soquete como 4.096, para que essas operações do emissor garantam que o TCP emissor preencha o buffer de recepção do soquete do receptor. O emissor envia então 1 byte de dados fora da banda, seguidos por 1.024 bytes de dados normais, e termina.

oob/tcpsend05.c

```
 1 #include   "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    sockfd, size;
 6     char   buff[16384];

 7     if(argc != 3)
 8         err_quit("usage: tcpsend05 <host> <port#>");

 9     sockfd = Tcp_connect(argv[1], argv[2]);

10     size = 32768;
```

Figura 24.10 Programa de envio (*continua*).

```
11      Setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, &size, sizeof(size));

12      Write(sockfd, buff, 16384);
13      printf("wrote 16384 bytes of normal data\n");
14      sleep(5);

15      Send(sockfd, "a", 1, MSG_OOB);
16      printf("wrote 1 byte of OOB data\n");

17      Write(sockfd, buff, 1024);
18      printf("wrote 1024 bytes of normal data\n");

19      exit(0);
20  }
```

oob/tcpsend05.c

Figura 24.10 Programa de envio (*continuação*).

A Figura 24.11 mostra o programa receptor.

14–20 O processo receptor configura o tamanho do buffer de recepção do soquete ouvinte como 4.096. Esse tamanho será transportado para o soquete conectado, depois que a conexão for estabelecida. Então, o processo aceita (`accepts`) a conexão, estabelece um handler de sinal para `SIGURG` e estabelece o proprietário do soquete. O loop principal chama `pause` em um loop infinito.

22–31 O handler de sinal chama `recv` para ler os dados fora da banda.

Eis a saída do emissor, quando iniciamos o receptor e então o emissor:

```
macosx % tcpsend05 freebsd4 5555
wrote 16384 bytes of normal data
wrote 1 byte of OOB data
wrote 1024 bytes of normal data
```

Como esperado, todos os dados se encaixam no buffer de envio do soquete do emissor e então terminam. Eis a saída do receptor:

```
freebsd4 % tcprecv05 5555
SIGURG received
recv error: Resource temporarily unavailable
```

A string de erro impressa por nossa função `err_sys` corresponde a `EAGAIN`, que é o mesmo que `EWOULDBLOCK` no FreeBSD. O protocolo TCP envia a notificação de fora da banda para a TCP receptora, que gera então o sinal `SIGURG` para o processo receptor. Mas, quando `recv` é chamado especificando o flag `MSG_OOB`, o byte fora da banda não pode ser lido.

oob/tcprecv05.c

```
 1 #include   "unp.h"

 2 int    listenfd, connfd;

 3 void   sig_urg(int);

 4 int
 5 main(int argc, char **argv)
 6 {
 7     int    size;

 8     if(argc == 2)
 9         listenfd = Tcp_listen(NULL, argv[1], NULL);
10     else if (argc == 3)
11         listenfd = Tcp_listen(argv[1], argv[2], NULL);
12     else
13         err_quit("usage: tcprecv05 [ <host> ] <port#>");
```

Figura 24.11 Programa receptor (*continua*).

```
14      size = 4096;
15      Setsockopt(listenfd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));
16      connfd = Accept(listenfd, NULL, NULL);
17      Signal(SIGURG, sig_urg);
18      Fcntl(connfd, F_SETOWN, getpid());
19      for ( ; ; )
20          pause();
21  }
22  void
23  sig_urg(int signo)
24  {
25      int     n;
26      char    buff[2048];
27      printf("SIGURG received\n");
28      n = Recv(connfd, buff, sizeof(buff) - 1, MSG_OOB);
29      buff[n] = 0;                    /* termina nulo */
30      printf("read %d OOB byte\n", n);
31  }
```

oob/tcprecv05.c

Figura 24.11 Programa receptor (*continuação*).

A solução é o receptor dar espaço em seu buffer de recepção do soquete, lendo os dados normais que estão disponíveis. Isso fará com que seu TCP anuncie uma janela diferente de zero para o emissor, que finalmente permitirá que o emissor transmita o byte fora da banda.

> Notamos dois problemas relacionados nas implementações derivadas do Berkeley (páginas 1016 e 1017 do TCPv2). Primeiro, mesmo que o buffer de envio do soquete esteja cheio, um byte fora da banda sempre é aceito pelo kernel do processo para envio ao peer. Segundo, quando o processo envia um byte fora da banda, um segmento TCP é enviado imediatamente, contendo a notificação urgente. Todas as verificações de saída TCP normal (algoritmo de Nagle, impedimento de janela inútil, etc.) são ignoradas.

Exemplo

Nosso próximo exemplo demonstra que há somente uma única marca de fora da banda para uma conexão TCP dada e que, se novos dados fora da banda chegam antes que o processo receptor leia alguns dados fora da banda existentes, a marca anterior é perdida.

A Figura 24.12 é o programa de envio, que é semelhante ao da Figura 24.8, com a adição de outro comando `send` de dados fora da banda, seguido por mais um comando `write` de dados normais.

oob/tcpsend06.c

```
 1  #include   "unp.h"
 2  int
 3  main(int argc, char **argv)
 4  {
 5      int     sockfd;
 6      if(argc != 3)
 7          err_quit("usage: tcpsend06 <host> <port#>");
 8      sockfd = Tcp_connect(argv[1], argv[2]);
 9      Write(sockfd, "123", 3);
10      printf("wrote 3 bytes of normal data\n");
```

Figura 24.12 Enviando dois bytes fora da banda em rápida sucessão (*continua*).

```
11      Send(sockfd, "4", 1, MSG_OOB);
12      printf("wrote 1 byte of OOB data\n");

13      Write(sockfd, "5", 1);
14      printf("wrote 1 byte of normal data\n");

15      Send(sockfd, "6", 1, MSG_OOB);
16      printf("wrote 1 byte of OOB data\n");

17      Write(sockfd, "7", 1);
18      printf("wrote 1 byte of normal data\n");

19      exit(0);
20  }
```

oob/tcpsend06.c

Figura 24.12 Enviando dois bytes fora da banda em rápida sucessão (*continuação*).

Não há nenhuma pausa no envio, permitindo que todos os dados sejam enviados rapidamente para o TCP receptor.

O programa receptor é idêntico ao da Figura 24.9, o qual dorme (`sleep`) por cinco segundos, após aceitar a conexão, para permitir que os dados cheguem em seu TCP. Eis a saída do programa receptor:

```
freebsd4 % tcprecv06 5555
read 5 bytes: 12345
at OOB mark
read 2 bytes: 67
received EOF
```

A chegada do segundo byte fora da banda (o `6`) sobrescreve a marca que foi armazenada quando o primeiro byte fora da banda chegou (o `4`). Como dissemos, há no máximo uma marca de fora da banda por conexão TCP.

24.4 Recapitulação sobre dados TCP fora da banda

Todos os nossos exemplos usando dados fora da banda fornecidos até agora foram simples. Infelizmente, dados de fora da banda tornam-se confusos quando consideramos os problemas de sincronização que podem surgir. O primeiro ponto a considerar é que o conceito de dados fora da banda na realidade transmite três diferentes informações para o receptor:

1. O fato de o emissor ter entrado no modo urgente. O processo receptor pode ser notificado disso com o sinal `SIGURG` ou com `select`. Essa *notificação* é transmitida imediatamente, depois que o emissor envia o byte fora da banda, porque vimos na Figura 24.11 que o TCP envia a notificação mesmo que seja impedido de enviar quaisquer dados para o receptor pelo controle de fluxo. Essa notificação poderia fazer com que o receptor entrasse em algum modo especial de processamento para quaisquer dados subseqüentes que recebesse.
2. A *posição* do byte fora da banda, isto é, para onde foi enviado com relação ao restante dos dados do emissor: a marca de fora da banda.
3. O *valor* real do byte fora da banda. Como o TCP é um protocolo de fluxo de bytes que não interpreta os dados enviados pela aplicação, esse pode ser qualquer valor de 8 bits.

Com o modo urgente do TCP, podemos considerar o flag URG como sendo a notificação, o ponteiro urgente como sendo a marca e o byte de dados como ele próprio.

Os problemas desse conceito de dados fora da banda são os seguintes: (i) há somente um ponteiro urgente TCP por conexão; (ii) há somente uma marca de fora da banda por conexão;

e (iii) há somente um buffer fora da banda de um byte por conexão (o que é um problema somente se os dados não estiverem sendo lidos em linha). Com a Figura 24.12, vimos que uma marca que chega sobrescreve qualquer marca anterior que o processo ainda não tiver encontrado. Se os dados estiverem sendo lidos em linha, os bytes fora da banda anteriores não serão perdidos quando novos dados de fora da banda chegarem, mas a marca será perdida.

Uma utilização comum de dados fora da banda é com `rlogin`, quando o cliente interrompe o programa que está sendo executado no servidor (páginas 393 e 394 do TCPv1). O servidor precisa dizer ao cliente para que descarte toda saída enfileirada, porque até uma janela inteira de saída pode ser enfileirada para envio do servidor para o cliente. O servidor envia um byte especial para o cliente, dizendo para que ele descarregue toda a saída, e esse byte é enviado como dados fora da banda. Quando o cliente recebe o sinal `SIGURG`, apenas lê o soquete até encontrar a marca, descartando tudo até ela. (As páginas 398 a 401 do TCPv1 têm um exemplo dessa utilização de dados fora da banda, junto com a saída de `tcpdump` correspondente.) Nesse cenário, se o servidor enviasse vários bytes fora da banda em rápida sucessão, isso não afetaria o cliente, pois este lê apenas até a marca final, descartando todos os dados.

Em resumo, a utilidade dos dados fora da banda depende do motivo de estarem sendo utilizados pela aplicação. Se o objetivo é dizer ao peer para que descarte os dados normais até a marca, então perder um byte fora da banda intermediário e sua marca correspondente não tem nenhuma conseqüência. Mas, se é importante que nenhum dos bytes fora da banda seja perdido, então os dados devem ser recebidos em linha. Além disso, os bytes de dados que são enviados como dados fora da banda devem ser diferenciados dos dados normais, pois marcas intermediárias podem ser sobrescritas quando uma nova marca é recebida, misturando efetivamente bytes fora da banda com dados normais. O comando `telnet`, por exemplo, envia seus próprios comandos no fluxo de dados normal entre o cliente e o servidor, prefixando-os com um byte de valor 255. (Enviar esse valor como dados exige, então, dois bytes sucessivos de valor 255.) Isso permite diferenciar seus comandos dos dados de usuário normais, mas exige que o cliente e o servidor processem cada byte de dados procurando comandos.

24.5 Resumo

O TCP não tem *dados fora da banda* verdadeiros. Ele fornece um ponteiro urgente, que é enviado no cabeçalho TCP para o peer assim que o emissor entra no modo urgente. A recepção desse ponteiro pela outra extremidade da conexão diz a esse processo que o emissor entrou no modo urgente e o ponteiro aponta para o byte final de dados urgentes. Mas todos os dados ainda estão sujeitos ao controle de fluxo normal do TCP.

A API de soquetes mapeia o modo urgente do TCP para aquilo que é chamado de dados fora da banda. O emissor entra no modo urgente especificando o flag `MSG_OOB` em uma chamada a `send`. O byte de dados final nessa chamada é considerado o byte fora da banda. O receptor é notificado quando seu protocolo TCP recebe um novo ponteiro urgente, por meio do sinal `SIGURG` ou por uma indicação de `select` dizendo que o soquete tem uma condição de exceção pendente. Por default, o protocolo TCP extrai o byte fora da banda do fluxo de dados normal e o coloca em seu próprio buffer fora da banda de um byte, que o processo lê chamando `recv` com o flag `MSG_OOB`. Como alternativa, o receptor pode configurar a opção de soquete `SO_OOBINLINE`, no caso em que o byte fora da banda é deixado no fluxo de dados normal. Independentemente do método utilizado pelo receptor, a camada de soquete mantém uma marca de fora da banda no fluxo de dados e não lerá até ela com uma única operação de entrada. O receptor determina se alcançou a marca chamando a função `sockatmark`.

Os dados fora da banda não são muito utilizados. Os comandos `telnet` e `rlogin` os utilizam, assim como FTP; todos eles os utilizam para notificar a extremidade remota a respeito de uma condição excepcional (por exemplo, interrupção de cliente) e os servidores descartam toda entrada recebida antes da marca de fora da banda.

Exercícios

24.1 Há uma diferença entre a chamada de função única

```
send(fd, "ab", 2, MSG_OOB);
```

e as duas chamadas de função

```
send(fd, "a", 1, MSG_OOB);
send(fd, "b", 1, MSG_OOB);
```

24.2 Refaça a Figura 24.6 para utilizar `poll` em vez de `select`.

Capítulo 25

E/S Dirigida por Sinal

25.1 Visão geral

Quando utiliza E/S dirigida por sinal (*signal-driven I/O*), o kernel nos avisa com um sinal, quando algo acontece em um descritor. Historicamente, isso foi chamado de *E/S assíncrona*, mas a E/S dirigida por sinal que descreveremos não é a E/S assíncrona verdadeira. Esta última é normalmente definida como o processo que executa a operação de E/S (digamos, uma leitura ou uma gravação), com o kernel retornando imediatamente após iniciar a operação de E/S. O processo continua a executar, enquanto a E/S acontece. Alguma forma de notificação é então fornecida para o processo, quando a operação está concluída ou encontra um erro. Na Seção 6.2, comparamos os vários tipos de E/S que estão normalmente disponíveis e mostramos a diferença entre E/S dirigida por sinal e E/S assíncrona.

A E/S não-bloqueadora que descrevemos no Capítulo 16 também não é a E/S assíncrona. No caso da E/S não-bloqueadora, o kernel não retorna após iniciar a operação de E/S; o kernel retorna imediatamente somente se a operação não puder ser concluída sem colocar o processo para dormir.

> O POSIX fornece E/S assíncrona verdadeira com suas funções `aio_`*XXX*. Essas funções permitem que o processo especifique se um sinal é gerado ou não, quando a E/S termina, e qual o sinal a ser gerado.

As implementações derivadas do Berkeley suportam E/S dirigida por sinal para soquetes e dispositivos terminais que utilizam o sinal `SIGIO`. O SVR4 suporta E/S dirigida por sinal para dispositivos STREAMS que utilizam o sinal `SIGPOLL`, o qual então é comparado ao sinal `SIGIO`.

25.2 E/S dirigida por sinal para soquetes

Utilizar E/S dirigida por sinal com um soquete (`SIGIO`) exige que o processo execute os três passos a seguir:

1. Um handler de sinal deve ser estabelecido para o sinal `SIGIO`.

2. O proprietário de soquete deve ser configurado, normalmente com o comando `F_SETOWN` de `fcntl` (Figura 7.20).
3. A E/S dirigida por sinal deve ser ativada para o soquete, normalmente com o comando `F_SETFL` de `fcntl`, para ativar o flag `O_ASYNC` (Figura 7.20).

   O flag `O_ASYNC` é uma adição relativamente tardia na especificação POSIX. Muito poucos sistemas implementaram suporte para o flag. Em vez disso, na Figura 25.4, ativaremos a E/S dirigida por sinal com o flag `ioctl FIOASYNC`. Note a má escolha de nomes do POSIX: o nome `O_SIGIO` teria sido uma escolha melhor para o novo flag.

   Devemos estabelecer o handler de sinal *antes* de configurar o proprietário do soquete. Sob implementações derivadas do Berkeley, a ordem das duas chamadas de função não importa, pois a ação default é ignorar `SIGIO`. Portanto, se invertêssemos a ordem das duas chamadas de função, haveria uma pequena chance de que um sinal pudesse ser gerado após a chamada a `fcntl`, mas antes da chamada a `signal`; se isso acontecesse, o sinal seria apenas descartado. No SVR4, entretanto, `SIGIO` é definido como `SIGPOLL` no cabeçalho `<sys/signal.h>` e a ação default de `SIGPOLL` é terminar o processo. Portanto, no SVR4, queremos garantir que o handler de sinal seja instalado antes de configurarmos o proprietário do soquete.

Embora configurar um soquete para E/S dirigida por sinal seja fácil, a parte difícil é determinar quais condições fazem `SIGIO` ser gerado para o proprietário de soquete. Isso depende do protocolo subjacente.

`SIGIO` com soquetes UDP

Utilizar E/S dirigida por sinal com UDP é simples. O sinal é gerado quando:

- Chega um datagrama para o soquete
- Um erro assíncrono ocorre no soquete

Então, quando capturamos `SIGIO` para um soquete UDP, chamamos `recvfrom` para ler o datagrama que chegou ou para obter o erro assíncrono. Falamos sobre erros assíncronos com referência a soquetes UDP na Seção 8.9. Lembre-se de que eles são gerados apenas se o soquete UDP for conectado.

> `SIGIO` é gerado para essas duas condições pelas chamadas a `sorwakeup` nas páginas 775, 779 e 784 do TCPv2.

`SIGIO` com soquetes TCP

Infelizmente, a E/S dirigida por sinal é quase inútil com um soquete TCP. O problema é que o sinal é gerado com muita freqüência e sua ocorrência não nos diz o que aconteceu. Conforme observado na página 439 do TCPv2, todas as condições a seguir fazem `SIGIO` ser gerado para um soquete TCP (supondo que a E/S dirigida por sinal esteja ativada):

- Uma solicitação de conexão concluída em um soquete de recepção
- Uma solicitação de desconexão foi iniciada
- Uma solicitação de desconexão foi concluída
- Metade de uma conexão foi desligada
- Dados chegaram em um soquete
- Dados foram enviados de um soquete (isto é, o buffer de saída tem espaço livre)
- Ocorreu um erro assíncrono

Por exemplo, se alguém está lendo e gravando em um soquete TCP, `SIGIO` é gerado quando novos dados chegam e quando dados anteriormente gravados são reconhecidos, e não

há nenhuma maneira de distinguir entre os dois no handler de sinal. Se SIGIO é utilizado nesse cenário, o soquete TCP deve ser configurado como não-bloqueador para evitar o bloqueio de um sinal read ou write. Devemos considerar a utilização de SIGIO somente com um soquete TCP de recepção, porque a única condição que gera SIGIO para um soquete de recepção é a conclusão de uma nova conexão.

A única utilização no mundo real de E/S dirigida por sinal com soquetes que os autores puderam encontrar é no servidor de NTP, que utiliza UDP. O loop principal do servidor recebe um datagrama de um cliente e envia uma resposta. Mas há um volume considerável de processamento a fazer para a solicitação de cada cliente (mais do que nosso servidor de eco simples). É importante para o servidor registrar indicações de tempo exatas para cada datagrama recebido, pois esse valor é retornado para o cliente e então utilizado por este para calcular o RTT para o servidor. A Figura 25.1 mostra duas maneiras de construir tal servidor de UDP.

A maioria dos servidores de UDP (incluindo nosso servidor de eco do Capítulo 8) é projetada como mostrado à esquerda dessa figura. Mas o servidor de NTP utiliza a técnica mostrada no lado direito: quando um novo datagrama chega, ele é lido pelo handler SIGIO, que também registra o momento da chegada. Então, o datagrama é colocado em outra fila dentro do processo a partir do qual será removido e processado pelo loop principal de servidor. Embora isso complique o código do servidor, fornece indicações de tempo exatas da chegada de datagramas.

> Lembre-se, da Figura 22.4, de que o processo pode configurar a opção de soquete IP_RECVDSTADDR para receber o endereço de destino de um datagrama de UDP recebido. Alguém poderia argumentar que duas informações adicionais, que também devem ser retornadas para um datagrama de UDP recebido, são uma indicação da interface recebida (que pode diferir do endereço de destino, caso o host empregue o modelo de sistema de lado fraco comum) e do momento em que o datagrama chegou.
>
> Para IPv6, a opção de soquete IPV6_PKTINFO (Seção 22.8) retorna a interface recebida. Para IPv4, discutimos a opção de soquete IP_RECVIF na Seção 22.2.
>
> O FreeBSD também fornece a opção de soquete SO_TIMESTAMP, que retorna o momento em que o datagrama foi recebido como dados auxiliares em uma estrutura timeval. O Linux fornece um ioctl SIOCGSTAMP que retorna uma estrutura timeval contendo o momento em que o datagrama foi recebido.

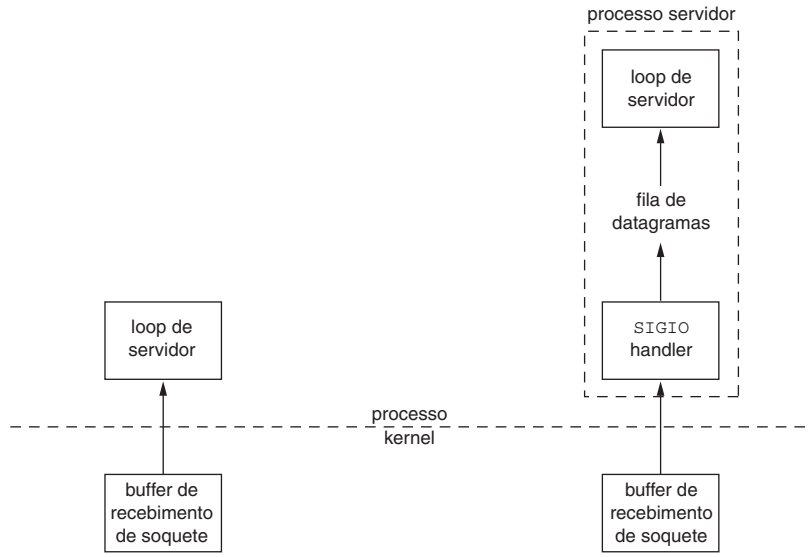


Figura 25.1 Duas maneiras diferentes de construir um servidor de UDP.

25.3 Servidor de eco UDP utilizando SIGIO

Agora, daremos um exemplo semelhante ao lado direito da Figura 25.1: um servidor UDP que utiliza o sinal SIGIO para receber os datagramas que chegam. Esse exemplo também ilustra o uso de sinais confiáveis POSIX.

Não alteramos em nada o cliente das Figuras 8.7 e 8.8, e a função main do servidor não muda em relação à Figura 8.3. As únicas alterações que fazemos aparecem na função dg_echo, que mostraremos nas quatro figuras a seguir. A Figura 25.2 mostra as declarações globais.

sigio/dgecho01.c

```
 1 #include "unp.h"

 2 static int sockfd;

 3 #define QSIZE 8                 /* tamanho da fila de entrada */
 4 #define MAXDG 4096              /* tamanho máximo do datagrama */

 5 typedef struct {
 6     void   *dg_data;            /* ponteiro para o datagrama real */
 7     size_t dg_len;              /* comprimento do datagrama */
 8     struct sockaddr *dg_sa;     /* ponteiro para sockaddr{} com endereço
                                    do cliente */
 9     socklen_t dg_salen;         /* comprimento de sockaddr{} */
10 } DG;
11 static DG dg[QSIZE];           /* fila de datagramas para processar */
12 static long cntread[QSIZE + 1]; /* contador de diagnóstico */

13 static int iget;               /* próximo para o loop principal
                                    processar */
14 static int iput;               /* próximo para o handler de sinal ler */
15 static int nqueue;             /* n° na fila para o loop principal
                                   processar */
16 static socklen_t clilen;       /* comprimento máximo de sockaddr{} */

17 static void sig_io(int);
18 static void sig_hup(int);
```

sigio/dgecho01.c

Figura 25.2 Declarações globais.

Fila de datagramas recebidos

3–12 O handler de sinal SIGIO coloca os datagramas que chegam em uma fila. Essa fila é um array de estruturas DG que tratamos como um buffer circular. Cada estrutura contém um ponteiro para o datagrama recebido, seu comprimento, um ponteiro para uma estrutura de endereço de soquete contendo o endereço de protocolo do cliente e o tamanho do endereço de protocolo. As QSIZE dessas estruturas são alocados e veremos, na Figura 25.4, que a função dg_echo chama malloc para alocar memória para todos os datagramas e estruturas de endereço de soquete.

Também alocamos um contador de diagnóstico, cntread, que examinaremos em breve. A Figura 25.3 mostra o array de estruturas, supondo que a primeira entrada aponta para um datagrama de 150 bytes e o comprimento de sua estrutura de endereço de soquete associada é 16.

Índices de array

13–15 iget é o índice da próxima entrada de array para o loop principal processar e iput é o índice da próxima entrada de array para o handler de sinal armazenar. nqueue é o número total de datagramas na fila para o loop principal processar.

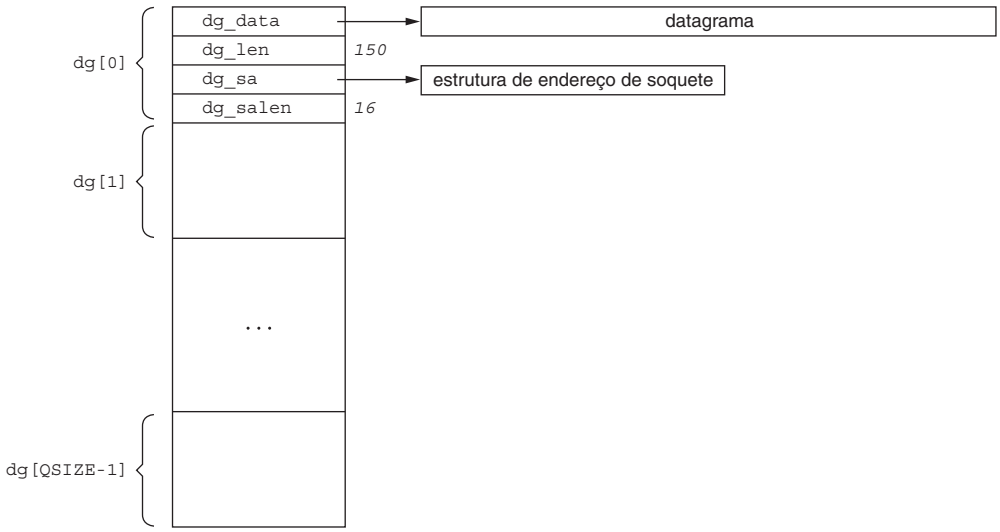


Figura 25.3 Estruturas de dados utilizadas para conter os datagramas recebidos e suas estruturas de endereço de soquete.

A Figura 25.4 mostra o loop de servidor principal, a função `dg_echo`.

sigio/dgecho01.c

```
19 void
20 dg_echo(int sockfd_arg, SA *pcliaddr, socklen_t clilen_arg)
21 {
22     int    i;
23     const int on = 1;
24     sigset_t zeromask, newmask, oldmask;

25     sockfd = sockfd_arg;
26     clilen = clilen_arg;

27     for (i = 0; i < QSIZE; i++) {  /* inicializa fila de buffers */
28         dg[i].dg_data = Malloc(MAXDG);
29         dg[i].dg_sa = Malloc(clilen);
30         dg[i].dg_salen = clilen;
31     }
32     iget = iput = nqueue = 0;

33     Signal(SIGHUP, sig_hup);
34     Signal(SIGIO, sig_io);
35     Fcntl(sockfd, F_SETOWN, getpid());
36     Ioctl(sockfd, FIOASYNC, &on);
37     Ioctl(sockfd, FIONBIO, &on);

38     Sigemptyset(&zeromask);       /* inicializa três conjuntos de sinais */
39     Sigemptyset(&oldmask);
40     Sigemptyset(&newmask);
41     Sigaddset(&newmask, SIGIO);  /* sinaliza que queremos bloquear */

42     Sigprocmask(SIG_BLOCK, &newmask, &oldmask);
43     for ( ; ; ) {
44         while (nqueue == 0)
45             sigsuspend(&zeromask); /* espera por datagrama para processar */

46             /* desbloqueia SIGIO */
```

Figura 25.4 Função `dg_echo`: loop de processamento principal de servidor (*continua*).

```
47          Sigprocmask(SIG_SETMASK, &oldmask, NULL);

48          Sendto(sockfd, dg[iget].dg_data, dg[iget].dg_len, 0,
49                 dg[iget].dg_sa, dg[iget].dg_salen);

50          if (++iget >= QSIZE)
51              iget = 0;

52              /* bloqueia SIGIO */
53          Sigprocmask(SIG_BLOCK, &newmask, &oldmask);
54          nqueue--;
55      }
56  }
```

sigio/dgecho01.c

Figura 25.4 Função `dg_echo`: loop de processamento principal de servidor (*continuação*).

Inicialização da fila de datagramas recebidos

27–32 O descritor de soquete é salvo em uma variável global, pois o handler de sinal precisa dele. A fila de datagramas recebidos é inicializada.

Estabelecimento dos handlers de sinal e configuração dos flags de soquete

33–37 Os handlers de sinal são estabelecidos para `SIGHUP` (que utilizamos para propósitos de diagnóstico) e `SIGIO`. O proprietário de soquete é configurado utilizando `fcntl` e os flags dirigidos por sinal e de E/S não-bloqueadora são configurados utilizando `ioctl`.

> Mencionamos anteriormente que o flag `O_ASYNC` com `fcntl` é a maneira POSIX de especificar E/S dirigida por sinal, mas, como a maioria dos sistemas ainda não o suporta, utilizamos `ioctl` em seu lugar. Embora a maioria dos sistemas suporte o flag `O_NONBLOCK` para configurar E/S não-bloqueadora, mostramos aqui o método `ioctl`.

Inicialização dos conjuntos de sinais

38–41 Três conjuntos de sinais são inicializados: `zeromask` (que nunca muda), `oldmask` (que contém a máscara de sinal antiga, quando bloqueamos `SIGIO`) e `newmask`. `sigaddset` ativa o bit correspondente a `SIGIO` em `newmask`.

Bloqueio de `SIGIO` e espera por algo a fazer

42–45 `sigprocmask` armazena a máscara de sinal atual do processo em `oldmask` e, então, utiliza a função lógica OU com `newmask` na máscara de sinal atual. Isso bloqueia `SIGIO` e retorna a máscara de sinal atual. Então, entramos no loop `for` e testamos o contador `nqueue`. Contanto que esse contador seja 0, não há nada a fazer e podemos chamar `sigsuspend`. Essa função POSIX salva a máscara de sinal atual internamente e, então, a configura com o argumento (`zeromask`). Como `zeromask` é um conjunto de sinais vazio, isso ativa todos os sinais. `sigsuspend` retorna depois que um sinal foi capturado e que o handler de sinal retorna. (Essa é uma função incomum, pois *sempre* retorna um erro, `EINTR`.) Antes de retornar, `sigsuspend` sempre configura a máscara de sinal com seu valor de quando a função foi chamada, que neste caso é o valor de `newmask`; portanto, temos a garantia de que, quando `sigsuspend` retorna, `SIGIO` é bloqueado. Essa é a razão pela qual podemos testar o contador `nqueue`, sabendo que, enquanto o estivermos testando, um sinal `SIGIO` não pode ser enviado.

> Considere o que aconteceria se `SIGIO` não fosse bloqueado enquanto testamos a variável `nqueue`, que é compartilhada entre o loop principal e o handler de sinal. Poderíamos testar `nqueue` e encontrar o valor 0, mas, imediatamente após esse teste, o sinal é enviado e `nqueue` é configurado como 1. Então, chamamos `sigsuspend` e vamos dormir, efetivamente perdendo o sinal. Nunca somos acordados com a chamada de `sigsuspend`, a menos que outro sinal ocorra. Isso é semelhante à condição de concorrência que descrevemos na Seção 20.5.

Desbloqueio de SIGIO e envio da resposta

46–51 Desbloqueamos SIGIO chamando sigprocmask para configurar a máscara de sinal do processo com o valor que foi salvo anteriormente (oldmask). Então, a resposta é enviada por sendto. O índice iget é incrementado e, se seu valor é o número de elementos no array, esse valor é configurado novamente como 0. Tratamos o array como um buffer circular. Observe que não precisamos de SIGIO bloqueado enquanto modificamos iget, pois esse índice é utilizado somente pelo loop principal; ele nunca é modificado pelo handler de sinal.

Bloqueio de SIGIO

52–54 SIGIO é bloqueado e o valor de nqueue é decrementado. Devemos bloquear o sinal enquanto modificamos essa variável, pois ela é compartilhada entre o loop principal e o handler de sinal. Além disso, precisamos de SIGIO bloqueado quando testamos nqueue no início do loop.

Uma técnica alternativa é remover as duas chamadas para sigprocmask que estão dentro do loop for, o que evita desbloquear o sinal e então bloqueá-lo posteriormente. O problema, entretanto, é que isso executa o loop inteiro com o sinal bloqueado, o que diminui a responsividade do handler de sinal. Os datagramas não devem ficar perdidos por causa dessa alteração (supondo que o buffer de recepção do soquete seja suficientemente grande), mas o envio do sinal para o processo será retardado o tempo inteiro durante o qual o sinal estiver bloqueado. Um objetivo ao codificar aplicações que realizam tratamento de sinal deve ser bloquear o sinal por um período mínimo de tempo.

A Figura 25.5 mostra o handler de sinal SIGIO.

sigio/dgecho01.c

```
57 static void
58 sig_io(int signo)
59 {
60     size_t len;
61     int    nread;
62     DG     *ptr;

63     for (nread = 0;;) {
64         if (nqueue >= QSIZE)
65             err_quit("receive overflow");

66         ptr = &dg[iput];
67         ptr->dg_salen = clilen;
68         len = recvfrom(sockfd, ptr->dg_data, MAXDG, 0,
69                        ptr->dg_sa, &ptr->dg_salen);
70         if (len < 0) {
71             if (errno == EWOULDBLOCK)
72                 break;   /* tudo feito; não há mais enfileirados para ler */
73             else
74                 err_sys("recvfrom error");
75         }
76         ptr->dg_len = len;

77         nread++;
78         nqueue++;
79         if (++iput >= QSIZE)
80             iput = 0;

81     }
82     cntread[nread]++; /* histograma de n° de datagramas lidos por sinal */
83 }
```

sigio/dgecho01.c

Figura 25.5 Handler SIGIO.

O problema que encontramos ao codificar esse handler de sinal é que os sinais POSIX normalmente *não* são enfileirados. Isso significa que, se estivermos no handler de sinal, que garante que o sinal é bloqueado, e o sinal ocorrer mais duas vezes, ele será enviado somente mais uma vez.

O POSIX fornece alguns sinais de tempo real que *são* enfileirados, mas outros sinais, como `SIGIO`, normalmente não são enfileirados.

Considere o cenário a seguir: um datagrama chega e o sinal é enviado. O handler de sinal lê o datagrama e o coloca na fila do loop principal. Mas, enquanto o handler de sinal está executando, chegam mais dois datagramas, fazendo o sinal ser gerado mais duas vezes. Como o sinal está bloqueado, quando o handler de sinal retorna, ele é chamado somente mais uma vez. Na segunda vez que o handler de sinal é executado, ele lê o segundo datagrama, mas o terceiro datagrama é deixado na fila de recepção do soquete. Esse terceiro datagrama será lido somente se e quando um quarto datagrama chegar. Quando um quarto datagrama chegar, o terceiro datagrama, não o quarto, será lido e colocado na fila do loop principal.

Como os sinais não são enfileirados, o descritor que é configurado para E/S dirigida por sinal normalmente também é configurado como não-bloqueador. Então, codificamos nosso handler `SIGIO` para ler em um loop, terminando somente quando a leitura retorna `EWOULDBLOCK`.

Verificação de estouro de fila

64–65 Se a fila está cheia, terminamos. Há outras maneiras de tratar disso (por exemplo, buffers adicionais poderiam ser alocados), mas para nosso exemplo simples já terminamos.

Leitura do datagrama

66–76 `recvfrom` é chamado no soquete não-bloqueador. A entrada de array indexada por `iput` é onde o datagrama é armazenado. Se não há nenhum datagrama para ler, `break` sai do loop `for`.

Incrementando os contadores e o índice

77–80 `nread` é um contador de diagnóstico do número de datagramas lidos por sinal. `nqueue` é o número de datagramas para o loop principal processar.

82 Antes que o handler de sinal retorne, ele incrementa o contador correspondente ao número de datagramas lidos por sinal. Imprimimos esse array na Figura 25.6, quando o sinal `SIGHUP` é enviado como informações de diagnóstico.

A função final (Figura 25.6) é o handler de sinal `SIGHUP`, que imprime o array `cntread`. Isso conta o número de datagramas lidos por sinal.

sigio/dgecho01.c

```
84 static void
85 sig_hup(int signo)
86 {
87     int     i;

88     for (i = 0; i <= QSIZE; i++)
89         printf("cntread[%d] = %ld\n", i, cntread[i]);
90 }
```

sigio/dgecho01.c

Figura 25.6 Handler `SIGHUP`.

Para ilustrar que os sinais não são enfileirados e que devemos configurar o soquete como não-bloqueador, além de configurar o flag de E/S orientado por sinal, executaremos esse servidor com seis clientes simultaneamente. Cada cliente envia 3.645 linhas para o servidor ecoar e cada cliente é iniciado a partir de um script de shell em segundo plano, de modo que todos os clientes são iniciados praticamente ao mesmo tempo. Quando todos os clientes tiverem terminado, enviamos o sinal `SIGHUP` para o servidor, fazendo-o imprimir seu array `cntread`.

```
linux % udpserv01
cntread[0] = 0
cntread[1] = 15899
cntread[2] = 2099
cntread[3] = 515
cntread[4] = 57
cntread[5] = 0
cntread[6] = 0
cntread[7] = 0
cntread[8] = 0
```

Na maioria das vezes, o handler de sinal lê somente um datagrama, mas há ocasiões em que mais de um está pronto. O contador não-zero para `cntread[0]` é quando o sinal é gerado enquanto o handler de sinal está executando, mas antes que o handler de sinal retorne, ele lê todos os datagramas pendentes. Quando o handler de sinal é chamado novamente, não resta nenhum datagrama para ler. Por fim, podemos verificar se a soma ponderada dos elementos do array ($15899 \times 1 + 2099 \times 2 + 515 \times 3 + 57 \times 4 = 21870$) é igual a 6 (o número de clientes) vezes 3.645 linhas por cliente.

25.4 Resumo

A E/S dirigida por sinal faz o kernel nos notificar com o sinal `SIGIO`, quando "algo" acontece em um soquete.

- Com um soquete TCP conectado, numerosas condições podem causar essa notificação, tornando esse recurso pouco útil.
- Com um soquete TCP de recepção, essa notificação ocorre quando uma nova conexão está pronta para ser aceita.
- Com UDP, essa notificação significa que chegou um datagrama ou um erro assíncrono; em ambos os casos, chamamos `recvfrom`.

Modificamos nosso servidor de eco UDP para utilizar E/S dirigida por sinal, usando uma técnica semelhante àquela utilizada pelo NTP: ler um datagrama logo que possível, depois que ele chega, para obter uma indicação de tempo exata de sua chegada e, então, enfileirá-lo para processamento posterior.

Exercício

25.1 Um projeto alternativo para o loop da Figura 25.4 é o seguinte:

```
for ( ; ; ) {
    Sigprocmask(SIG_BLOCK, &newmask, &oldmask);
    while (nqueue == 0)
        sigsuspend(&zeromask); /* espera por datagrama para processar */
    nqueue--;

        /* desbloqueia SIGGIO */
    Sigprocmask(SIG_SETMASK, &oldmask, NULL);

    Sendto(sockfd, dg[iget].dg_data, dg[iget].dg_len, 0,
           dg[iget].dg_sa, dg[iget].dg_salen);
    if (++iget >= QSIZE)
        iget = 0;
}
```

Essa modificação é aceitável?

Capítulo **26**

Threads

26.1 Visão geral

No modelo Unix tradicional, quando um processo precisa de algo realizado por outra entidade, ele bifurca um processo-filho e o deixa realizar o processamento. No Unix, a maioria dos servidores de rede é escrita dessa maneira, como vimos em nossos exemplos de servidor concorrente: o pai aceita (`accept`) a conexão, bifurca (`fork`) um filho e este trata do cliente.

Embora esse paradigma tenha servido bem durante muitos anos, há problemas com a bifurcação:

- A bifurcação (`fork`) é cara. A memória é copiada do pai para o filho, todos os descritores são duplicados no filho e assim por diante. As implementações atuais utilizam uma técnica chamada "*copiar ao gravar*"(*copy-on-write*), que evita uma cópia do espaço de dados do pai para o filho, até que este precise de sua própria cópia. Mas, independentemente dessa otimização, a bifurcação é cara.
- É exigido um IPC para passar as informações entre o pai e o filho, *após* a bifurcação. A passagem de informações do pai para o filho *antes* da bifurcação é fácil, desde que o filho comece com uma cópia do espaço de dados do pai e com uma cópia de todos os descritores do pai. Mas retornar informações do filho para o pai dá mais trabalho.

Os threads ajudam em ambos os problemas. Às vezes, eles são chamados de *processos leves*, pois são "mais leves" que um processo. Isto é, a criação de um thread pode ser 10 a 100 vezes mais rápida que a criação de um processo.

Todos os threads dentro de um processo compartilham a mesma memória global. Isso facilita o compartilhamento de informações entre os threads, mas junto com essa facilitação vem o problema da *sincronização*.

Mais que apenas as variáveis globais são compartilhadas. Todos os threads dentro de um processo compartilham o seguinte:

- Instruções do processo
- A maioria dos dados
- Arquivos abertos (por exemplo, descritores)

- Handlers de sinal e disposições de sinal
- Diretório de trabalho atual
- IDs de usuário e de grupo

Mas cada thread tem seu(sua) próprio(a)

- ID de thread
- Conjunto de registradores, incluindo contador do programa e ponteiro da pilha
- Pilha (para variáveis locais e endereços de retorno)
- `errno`
- Máscara de sinal
- Prioridade

> Uma analogia é considerar os handlers de sinal como um tipo de thread, conforme discutimos na Seção 11.18. Isto é, no modelo Unix tradicional, temos o fluxo de execução principal (um thread) e um handler de sinal (outro thread). Se o fluxo principal está no meio de uma atualização de uma lista encadeada, quando um sinal ocorre, e o handler de sinal também tenta atualizar a lista encadeada, normalmente resultam danos. O fluxo principal e o handler de sinal compartilham as mesmas variáveis globais, mas cada um tem sua própria pilha.

Neste texto, abordaremos os threads POSIX, também chamados de *Pthreads*. Eles foram padronizados em 1995, como parte do padrão POSIX.1c, e a maioria das versões do Unix os suportará no futuro. Veremos que todas as funções Pthread começam com `pthread_`. Este capítulo é uma introdução aos threads, para que possamos utilizá-los em nossos programas de rede. Para detalhes adicionais, veja Butenhof (1997).

26.2 Funções de thread básicas: criação e término

Nesta seção, abordaremos cinco funções básicas de thread e, então, as utilizaremos nas próximas duas seções para recodificar nosso cliente/servidor TCP utilizando threads em vez de `fork`.

Função `pthread_create`

Quando um programa é iniciado por `exec`, um thread simples é criado, o chamado *thread inicial* ou *principal*. Threads adicionais são criados por `pthread_create`.

```
#include <pthread.h>

int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*func)(void *), void *arg);
```

Retorna: 0 se OK, valor E*xxx* positivo em caso de erro

Cada thread dentro de um processo é identificado por uma *ID de thread*, cujo tipo de dados é `pthread_t` (freqüentemente um valor `unsigned int`). Na criação bem-sucedida de um novo thread, seu ID é retornado pelo ponteiro *tid*.

Cada thread tem numerosos *atributos*: sua prioridade, seu tamanho de pilha inicial, se deve ser daemon ou não, e assim por diante. Quando um thread é criado, podemos especificar esses atributos inicializando uma variável `pthread_attr_t`, que sobrescreve o default. Normalmente, usamos o default; neste caso, especificamos o argumento *attr* como um ponteiro nulo.

Por fim, quando criamos um thread, especificamos uma função para ele executar. O thread inicia chamando essa função e, então, termina explicitamente (chamando `pthread_exit`) ou implicitamente (permitindo que a função retorne). O endereço da função é especificado como o argumento *func,* e essa função é chamada com um único argumento de ponteiro, *arg*. Se precisarmos de vários argumentos para a função, deveremos empacotá-los em uma estrutura e então passar o endereço dessa estrutura como o único argumento para a função inicial.

Observe as declarações de *func* e *arg*. A função recebe um argumento, um ponteiro genérico (`void *`), e retorna um ponteiro genérico (`void *`). Isso nos permite passar um ponteiro (para qualquer coisa que quisermos) para o thread e permite que o thread retorne um ponteiro (novamente, para qualquer coisa que quisermos).

O valor de retorno das funções Pthread normalmente é 0 em caso de sucesso ou não-zero em caso de erro. Mas, ao contrário das funções de soquete e da maioria das chamadas de sistema, que retornam –1 em caso de erro e configuram `errno` com um valor positivo, as funções Pthread retornam a indicação de erro positiva como valor de retorno da função. Por exemplo, se `pthread_create` não pode criar um novo thread por exceder algum limite do sistema quanto ao número de threads, o valor de retorno da função é `EAGAIN`. As funções Pthread não configuram `errno`. A convenção de 0 para sucesso ou não-zero para erro está bem, desde que todos os valores E*xxx* em `<sys/errno.h>` sejam positivos. Um valor igual a 0 nunca é atribuído a um dos nomes E*xxx*.

Função `pthread_join`

Podemos esperar que determinado thread termine chamando `pthread_join`. Comparando os threads com os processos do Unix, `pthread_create` é semelhante a `fork` e `pthread_join` é semelhante a `waitpid`.

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **status);
```

Retorna: 0 se OK, valor E*xxx* positivo em caso de erro

Devemos especificar o *tid* do thread que queremos esperar. Infelizmente, não há nenhuma maneira de esperar por qualquer um de nossos threads (semelhante a `waitpid` com o argumento de ID de processo –1). Retornaremos a esse problema quando discutirmos a Figura 26.14.

Se o ponteiro *status* é não-nulo, o valor de retorno do thread (um ponteiro para algum objeto) é armazenado no local apontado por *status*.

Função `pthread_self`

Cada thread tem um ID que o identifica dentro de determinado processo. O ID do thread é retornado por `pthread_create` e vimos que ele foi utilizado por `pthread_join`. Um thread busca esse valor para si próprio utilizando `pthread_self`.

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Retorna: ID do thread de chamada

Comparando threads com processos Unix, `pthread_self` é semelhante a `getpid`.

Função `pthread_detach`

Um thread é *associável* (o default) ou *separável*. Quando um thread associável termina, seu ID e seu *status* de saída são mantidos até que outro thread chame `pthread_join`. Mas um thread separável (*detached*) é como um processo daemon: quando termina, todos os seus recursos são liberados e não podemos ficar em espera pelo seu término. Se um thread precisa saber quando outro termina, é melhor deixá-lo como associável (*joinable*).

A função `pthread_detach` altera o thread especificado para que ele seja separável.

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

Retorna: 0 se OK, valor E*xxx* positivo em caso de erro

Essa função é comumente chamada pelo thread que quer se separar, como em

```
pthread_detach(pthread_self());
```

Função `pthread_exit`

Uma maneira de terminar um thread é chamar `pthread_exit`.

```
#include <pthread.h>

void pthread_exit(void *status);
```

Não retorna para o chamador

Se o thread não é separável, seu ID e seu *status* de saída são mantidos para uma função `pthread_join` posterior, executada por algum outro thread no processo chamador.

O ponteiro *status* não deve apontar para um objeto que seja local para o thread chamador, pois esse objeto desaparecerá quando o thread terminar.

Há duas outras maneiras de um thread terminar:

- A função que o iniciou (o terceiro argumento de `pthread_create`) pode retornar. Como essa função deve ser declarada como retornando um ponteiro `void`, esse valor de retorno é o *status* de saída do thread.
- Se a função `main` do processo retorna ou se qualquer thread chama `exit`, o processo termina, incluindo todos os threads.

26.3 Função `str_cli` utilizando threads

Nosso primeiro exemplo utilizando threads é recodificar a função `str_cli` da Figura 16.10, que utiliza `fork`, para utilizar threads. Lembre-se de que fornecemos várias outras versões dessa função: a original, na Figura 5.5, utilizava um protocolo de parada-e-espera, que mostramos estar distante do ótimo para entrada de lote; a Figura 6.13 utilizava E/S bloqueadora e a função `select`; e a versão que inicia com a Figura 16.3 usava E/S não-bloqueadora. A Figura 26.1 mostra o projeto da nossa versão de threads.

A Figura 26.2 mostra a função `str_cli` utilizando threads.

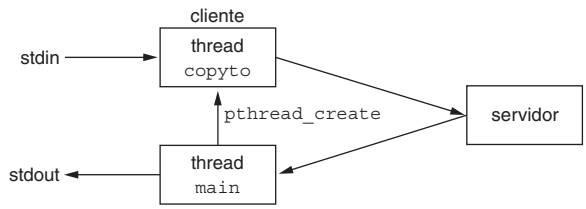


Figura 26.1 Recodificando `str_cli` para utilizar threads.

threads/strclithread.c

```
 1 #include   "unpthread.h"
 2 void   *copyto(void *);
 3 static int sockfd;       /* global para ambos os threads acessarem */
 4 static FILE *fp;
 5 void
 6 str_cli(FILE *fp_arg, int sockfd_arg)
 7 {
 8     char   recvline[MAXLINE];
 9     pthread_t tid;
10     sockfd = sockfd_arg;  /* copia os argumentos p/ variáveis externas */
11     fp = fp_arg;
12     Pthread_create(&tid, NULL, copyto, NULL);
13     while (Readline(sockfd, recvline, MAXLINE) > 0)
14         Fputs(recvline, stdout);
15 }
16 void *
17 copyto(void *arg)
18 {
19     char   sendline[MAXLINE];
20     while (Fgets(sendline, MAXLINE, fp) != NULL)
21         Writen(sockfd, sendline, strlen(sendline));
22     Shutdown(sockfd, SHUT_WR); /* EOF em stdin, envia FIN */
23     return (NULL);
24         /* retorna (isto é, o thread termina) quando EOF em stdin */
25 }
```

threads/strclithread.c

Figura 26.2 Função `str_cli` utilizando threads.

Cabeçalho `unpthread.h`

1 Esta é a primeira vez que encontramos o cabeçalho `unpthread.h`. Ele inclui nosso cabeçalho `unp.h` normal, seguido do cabeçalho POSIX `<pthread.h>` e, então, define os protótipos de função para nossas versões de empacotadoras das funções `pthread_`*XXX* (Seção 1.4), todas as quais iniciam com `Pthread_`.

Salvando argumentos em variáveis externas

10–11 O thread que estamos para criar precisa dos valores dos dois argumentos para `str_cli`: `fp`, o ponteiro `FILE` de E/S-padrão para o arquivo de entrada, e `sockfd`, o soquete TCP conectado ao servidor. Por simplicidade, armazenamos esses dois valores em variáveis externas. Uma técnica alternativa é colocar ambos os valores em uma estrutura e então passar um ponteiro para essa estrutura como argumento do thread que estamos para criar.

Criação de um novo thread

12 O thread é criado e o novo ID de thread é salvo em `tid`. A função executada pelo novo thread é `copyto`. Nenhum argumento é passado para o thread.

Loop principal do thread: copia soquete para a saída-padrão

13–14 O thread principal chama `readline` e `fputs`, copiando do soquete para a saída-padrão.

Término

15 Quando a função `str_cli` retorna, a função `main` termina, chamando `exit` (Seção 5.4). Quando isso acontece, *todos* os threads do processo são terminados. Normalmente, o thread `copyto` já terá terminado quando a função `main` do servidor for concluída. Mas, no caso em que o servidor termina prematuramente (Seção 5.12), chamar `exit` quando a função `main` do servidor for concluída terminará o thread `copyto`, que é o que queremos.

Thread `copyto`

16–25 Esse thread apenas copia da entrada-padrão para o soquete. Quando ele lê um EOF na entrada-padrão, um FIN é enviado através do soquete por `shutdown` e o thread retorna. A função `return` dessa função (que iniciou o thread) termina o thread.

No fim da Seção 16.2, fornecemos medidas para as cinco diferentes técnicas de implementação que utilizamos com nossa função `str_cli`. A versão de threads que acabamos de apresentar executou em 8,5 segundos, sendo ligeiramente mais rápida que a versão que utiliza `fork` (o que esperávamos), mas mais lenta que a versão de E/S não-bloqueadora. Contudo, comparando a complexidade da versão de E/S não-bloqueadora (Seção 16.2) com a simplicidade da versão de threads, ainda recomendamos o uso de threads, em vez da E/S não-bloqueadora.

26.4 Servidor de eco de TCP utilizando threads

Agora, refaremos nosso servidor de eco de TCP da Figura 5.2, utilizando um thread por cliente, em vez de um processo-filho por cliente. Também o tornaremos independente de protocolo, utilizando nossa função `tcp_listen`. A Figura 26.3 mostra o servidor.

Criação do thread

17–21 Quando `accept` retorna, chamamos `pthread_create` em vez de `fork`. O único argumento que passamos para a função `doit` é o descritor de soquete conectado, `connfd`.

> Convertemos o descritor inteiro `connfd` para ser um ponteiro `void`. A linguagem C ANSI não garante que isso funciona. Isso funciona somente nos sistemas em que o tamanho de um inteiro é menor ou igual ao tamanho de um ponteiro. Felizmente, a maioria das implementações do Unix tem essa propriedade (Figura 1.17). Falaremos mais sobre isso em breve.

Função do thread

23–30 `doit` é a função executada pelo thread. O thread se separa desde que não haja nenhuma razão para que o thread principal espere por cada thread que cria. A função `str_echo` não muda em relação à Figura 5.3. Quando essa função retorna, devemos fechar (`close`) o soquete conectado, pois o thread compartilha todos os descritores com o thread principal. Com `fork`, o filho não precisava fechar (`close`) o soquete conectado porque, quando terminava, todos os descritores abertos eram fechados ao término do processo (veja o Exercício 26.2).

Note também que o thread principal não fecha o soquete conectado, o que sempre fizemos com um servidor concorrente que chama `fork`. Isso acontece porque todos os threads

dentro de um processo compartilham os descritores; assim, se o thread principal chamasse `close`, ele terminaria a conexão. A criação de um novo thread não afeta as contagens de referência para descritores abertos, o que é diferente de `fork`.

Há um erro sutil nesse programa, que descreveremos em detalhes na Seção 26.5. Você consegue descobri-lo (veja o Exercício 26.5)?

threads/tcpserv01.c

```
 1 #include   "unpthread.h"
 2 static void *doit(void *);       /* cada thread executa esta função */
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int    listenfd, connfd;
 7     pthread_t tid;
 8     socklen_t addrlen, len;
 9     struct sockaddr *cliaddr;
10     if (argc == 2)
11         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
12     else if (argc == 3)
13         listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
14     else
15         err_quit("usage: tcpserv01 [ <host> ] <service or port>");
16     cliaddr = Malloc(addrlen);
17     for ( ; ; ) {
18         len = addrlen;
19         connfd = Accept(listenfd, cliaddr, &len);
20         Pthread_create(&tid, NULL, &doit, (void *) connfd);
21     }
22 }
23 static void *
24 doit(void *arg)
25 {
26     Pthread_detach(pthread_self());
27     str_echo((int) arg);          /* a mesma função de antes */
28     Close((int) arg);             /* feito com soquete conectado */
29     return (NULL);
30 }
```

threads/tcpserv01.c

Figura 26.3 Servidor de eco de TCP utilizando threads (veja também o Exercício 26.5).

Passando argumentos para novos threads

Mencionamos que, na Figura 26.3, convertemos a variável inteira `connfd` para um ponteiro `void`, mas não é garantido que isso funcione em todos os sistemas. Tratar disso corretamente exige trabalho adicional.

Primeiro, note que não podemos apenas passar o endereço de `connfd` para o novo thread. Isto é, o que segue não funciona:

```
int
main(int argc, char **argv)
{
   int   listenfd, connfd;
   ...
   for ( ; ; ) {
      len = addrlen;
```

```
        connfd = Accept(listenfd, cliaddr, &len);

        Pthread_create(&tid, NULL, &doit, &connfd);
    }
}
static void *
doit(void *arg)
{
    int  connfd;

    connfd = *((int *) arg);
    Pthread_detach(pthread_self());
    str_echo(connfd);   /* a mesma função de antes */
    Close(connfd);      /* feito com soquete conectado */
    return(NULL);
}
```

Da perspectiva da linguagem C ANSI, isso é aceitável: temos a garantia de que podemos converter o ponteiro inteiro para um `void *` e, então, convertemos esse ponteiro de volta para um ponteiro inteiro. O problema é para o que esse ponteiro aponta.

Há uma variável inteira, `connfd`, no thread principal e cada chamada a `accept` a sobrescreve com um novo valor (o descritor conectado). O cenário a seguir pode ocorrer:

- `accept` retorna, `connfd` é armazenado (digamos que o novo descritor seja 5) e o thread principal chama `pthread_create`. O ponteiro para `connfd` (não seu conteúdo) é o argumento final para `pthread_create`.
- Um thread é criado e a função `doit` é programada para começar a executar.
- Outra conexão está pronta e o thread principal é executado novamente (antes do thread recentemente criado). `accept` retorna, `connfd` é armazenado (digamos que o novo descritor agora seja 6) e o thread principal chama `pthread_create`.

Mesmo que dois threads sejam criados, ambos operarão sobre o valor final armazenado em `connfd`, que supomos ser 6. O problema é que vários threads estão acessando uma variável compartilhada (o valor inteiro em `connfd`) sem sincronização. Na Figura 26.3, resolvemos esse problema passando o *valor* de `connfd` para `pthread_create`, em vez de um ponteiro para o valor. Isso está bem, dada a maneira pela qual a linguagem C passa valores inteiros para uma função chamada (uma cópia do valor é colocada na pilha da função chamada).

A Figura 26.4 mostra uma solução melhor para esse problema.

threads/tcpserv02.c

```
 1 #include   "unpthread.h"
 2 static void *doit(void *);       /* cada thread executa esta função */
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int    listenfd, *iptr;
 7     thread_t tid;
 8     socklen_t addrlen, len;
 9     struct sockaddr *cliaddr;
10     if (argc == 2)
11         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
12     else if (argc == 3)
13         listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
```

Figura 26.4 Servidor de eco de TCP utilizando threads com passagem de argumento mais portável (*continua*).

```
14      else
15          err_quit("usage: tcpserv01 [ <host> ] <service or port>");
16      cliaddr = Malloc(addrlen);
17      for ( ; ; ) {
18          len = addrlen;
19          iptr = Malloc(sizeof(int));
20          *iptr = Accept(listenfd, cliaddr, &len);
21          Pthread_create(&tid, NULL, &doit, iptr);
22      }
23  }
24  static void *
25  doit(void *arg)
26  {
27      int     connfd;
28      connfd = *((int *) arg);
29      free(arg);
30      Pthread_detach(pthread_self());
31      str_echo(connfd);           /* a mesma função de antes */
32      Close(connfd);              /* feito com soquete conectado */
33      return (NULL);
34  }
```

threads/tcpserv02.c

Figura 26.4 Servidor de eco de TCP utilizando threads com passagem de argumento mais portável (*continuação*).

17–22 Toda vez que chamamos `accept`, chamamos primeiro `malloc` e alocamos espaço para uma variável inteira, o descritor conectado. Isso fornece a cada thread sua própria cópia do descritor conectado.

28–29 O thread busca o valor do descritor conectado e, então, chama `free` para liberar a memória.

Historicamente, as funções `malloc` e `free` têm sido não-reentrantes. Isto é, chamar qualquer uma das funções a partir de um handler de sinal, enquanto o thread principal está no meio de uma delas, tem sido a receita para um desastre, devido às estruturas de dados estáticas que são manipuladas por essas funções. Como podemos chamar essas duas funções na Figura 26.4? O POSIX exige que ambas, junto com muitas outras, sejam *seguras para threads*. Isso normalmente é feito por alguma forma de sincronização dentro das funções de biblioteca, que é transparente para nós.

Funções seguras para threads

O POSIX.1 exige que todas as funções por ele definidas e pelo padrão ANSI C sejam seguras para threads, com as exceções listadas na Figura 26.5.

Infelizmente, o POSIX não diz nada sobre segurança de thread com relação às funções de API de interligação em rede. As últimas cinco linhas dessa tabela são do Unix 98. Falamos sobre a propriedade não-reentrante de `gethostbyname` e de `gethostbyaddr` na Seção 11.18. Mencionamos que, mesmo que alguns fornecedores tenham definido versões seguras para threads, cujos nomes terminam em `_r`, não há nenhum padrão para essas funções e elas devem ser evitadas. Todas as funções `get`*XXX* não-reentrantes foram resumidas na Figura 11.21.

Vemos na Figura 26.5 que a técnica comum para tornar uma função segura para threads é definir uma nova função, cujo nome termine em `_r`. Duas das funções são seguras para threads somente se o chamador alocar espaço para o resultado e passar esse ponteiro como o argumento da função.

Não precisa ser segura para threads	Deve ser segura para threads	Comentário
asctime	asctime_r	
	ctermid	Segura para threads somente se tiver argumento não-nulo
ctime	ctime_r	
getc_unlocked		
getchar_unlocked		
getgrid	getgrid_r	
getgrnam	getgrnam_r	
getlogin	getlogin_r	
getpwnam	getpwnam_r	
getpwuid	getpwuid_r	
gmtime	gmtime_r	
localtime	localtime_r	
putc_unlocked		
putchar_unlocked		
rand	rand_r	
readdir	readdir_r	
strtok	strtok_r	
	tmpnam	Segura para threads somente se tiver argumento não-nulo
ttyname	ttyname_r	
gethost*XXX*		
getnet*XXX*		
getproto*XXX*		
getserv*XXX*		
inet_ntoa		

Figura 26.5 Funções seguras para threads.

26.5 Dados específicos de thread

Ao se converterem funções existentes para executarem em um ambiente de threads, um problema comum encontrado ocorre devido às variáveis estáticas. Uma função que mantém o estado em um buffer privado ou que retorna um resultado na forma de um ponteiro para um buffer estático não é segura para threads, pois múltiplos threads não podem utilizar o mesmo buffer para conter coisas diferentes ao mesmo tempo. Quando ocorrer esse problema, há várias soluções:

- Utilizar dados específicos de thread. Isso não é trivial e converte a função em uma que funciona somente em sistemas com suporte para threads. A vantagem dessa estratégia é que a seqüência de chamada não muda e todas as alterações entram na função de biblioteca e não nas aplicações que chamam a função. Mostraremos uma versão de `readline` que é segura para threads, utilizando dados específicos de thread, posteriormente nesta seção.
- Alterar a seqüência de chamada de modo que o chamador empacote todos os argumentos em uma estrutura e também armazene nessa estrutura as variáveis estáticas da Figura 3.18. Isso também foi feito, e a Figura 26.6 mostra a nova estrutura e os novos protótipos de função.

```
typedef struct {
    int      read_fd;        /* descritor do chamador para ler */
    char     *read_ptr;      /* buffer do chamador para ler */
    size_t   read_maxlen;    /* número máximo de bytes do chamador para ler */
                 /* os próximos três são utilizados internamente pela função */
    int      rl_cnt;         /* inicializa como 0 */
    char     *rl_bufptr;     /* inicializa como rl_buf */
```

Figura 26.6 Estrutura de dados e protótipo de função para a versão reentrante de `readline` (*continua*).

```
    char     rl_buf[MAXLINE];
} Rline;

void     readline_rinit(int, void *, size_t, Rline *);
ssize_t readline_r(Rline *);
ssize_t Readline_r(Rline *);
```

Figura 26.6 Estrutura de dados e protótipo de função para a versão reentrante de `readline` (*continuação*).

Essas novas funções podem ser utilizadas em sistemas com e sem threads, mas todas as aplicações que chamam `readline` devem mudar.

- Reestruturar a interface para evitar quaisquer variáveis estáticas, para que a função seja segura para threads. Para o exemplo de `readline`, isso seria o equivalente a ignorar os aumentos de velocidade introduzidos na Figura 3.18 e voltar à versão mais antiga da Figura 3.17. Como dissemos que a versão mais antiga era "dolorosamente lenta", usar essa opção nem sempre é viável.

Usar dados específicos de thread é uma técnica comum para tornar uma função existente segura para threads. Antes de descrevermos as funções Pthread que trabalham com dados específicos de thread, descreveremos o conceito e uma *possível* implementação, pois as funções parecem mais complicadas do que realmente são.

> Parte da complicação em muitos textos sobre a utilização de threads é que suas descrições de dados específicos de thread se parecem com o padrão de Pthreads, falando sobre pares de chave-valor e chaves sendo objetos opacos. Descrevemos os dados específicos de thread em termos de *índices* e *ponteiros* porque as implementações comuns utilizam um pequeno índice de inteiros para a chave e o valor associado ao índice é apenas um ponteiro para uma região que o thread aloca (`malloc`).

Cada sistema suporta um número limitado de itens de dados específicos de thread. O POSIX exige que esse limite não seja menor que 128 (por processo) e utilizamos esse limite no exemplo a seguir. O sistema (provavelmente a biblioteca de threads) mantém um array de estruturas por processo, que chamamos de estruturas `Key`, conforme mostramos na Figura 26.7.

O flag na estrutura `Key` indica se esse elemento do array está correntemente em uso e todos os flags são inicializados de forma a "não estarem em uso". Quando um thread chama `pthread_key_create` para criar um novo item de dados específico de thread, o sistema pesquisa seu array de estruturas `Key` e localiza a primeira que não está em uso. Seu índice, de 0 a 127, é chamado de *chave* e é retornado para o thread chamador. Falaremos sobre o "ponteiro destrutor", o outro membro da estrutura `Key`, em breve.

Além do array de estruturas `Key` com abrangência de processo, o sistema mantém numerosas informações sobre cada thread dentro de um processo. Chamamos isso de estrutura

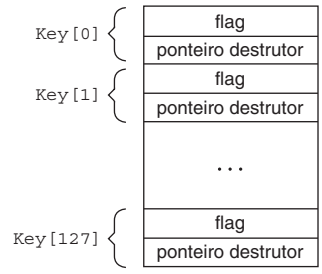


Figura 26.7 Possível implementação de dados específicos de thread.

`Pthread` e parte dessas informações é um array de ponteiros de 128 elementos, o qual chamamos de array `pkey`. Mostramos isso na Figura 26.8.

Todas as entradas no array `pkey` são inicializadas como ponteiros nulos. Esses 128 ponteiros são os "valores" associados a cada uma das 128 "chaves" possíveis no processo.

Quando criamos uma chave com `pthread_key_create`, o sistema nos informa sua chave (índice). Cada thread pode então armazenar um valor (ponteiro) para a chave, e cada thread normalmente obtém o ponteiro chamando `malloc`. Parte da confusão com os dados específicos de thread é porque o ponteiro é o valor no par chave-valor, mas os dados específicos de thread *reais* são o que esse ponteiro aponta.

Agora, vamos ver um exemplo de como os dados específicos de thread são utilizados, supondo que nossa função `readline` utiliza dados específicos de thread para manter o estado por thread através de sucessivas chamadas à função. Brevemente, mostraremos o código para isso, modificando nossa função `readline` para seguir estes passos:

1. Um processo é iniciado e múltiplos threads são criados.
2. Um dos threads será o primeiro a chamar `readline`, que, por sua vez, chama `pthread_key_create`. O sistema localiza a primeira estrutura `Key` não utilizada na Figura 26.7 e retorna seu índice (0-127) para o chamador. Nesse exemplo, supomos um índice igual a 1.

   Utilizaremos a função `pthread_once` para garantir que `pthread_key_create` seja chamada somente pelo primeiro thread a chamar `readline`.
3. `readline` chama `pthread_getspecific` para obter o valor de `pkey[1]` (o "ponteiro" na Figura 26.8, para essa chave igual a 1) para esse thread e o valor retornado é um ponteiro nulo. Portanto, `readline` chama `malloc` para alocar a memória que precisa para manter as informações por thread em sucessivas chamadas de `readline` para esse thread. `readline` inicializa essa memória, conforme for necessário, e chama `pthread_setspecific` para configurar o ponteiro de dados específicos de thread (`pkey[1]`), para que essa chave aponte para a memória que acaba de alocar. Mostramos isso na Figura 26.9, supondo que o thread chamador seja o thread 0 no processo.

   Nessa figura, observamos que a estrutura `Pthread` é mantida pelo sistema (provavelmente a biblioteca de threads), mas os dados específicos de thread reais em que usamos `malloc` são mantidos por nossa função (`readline`, neste caso). Tudo que

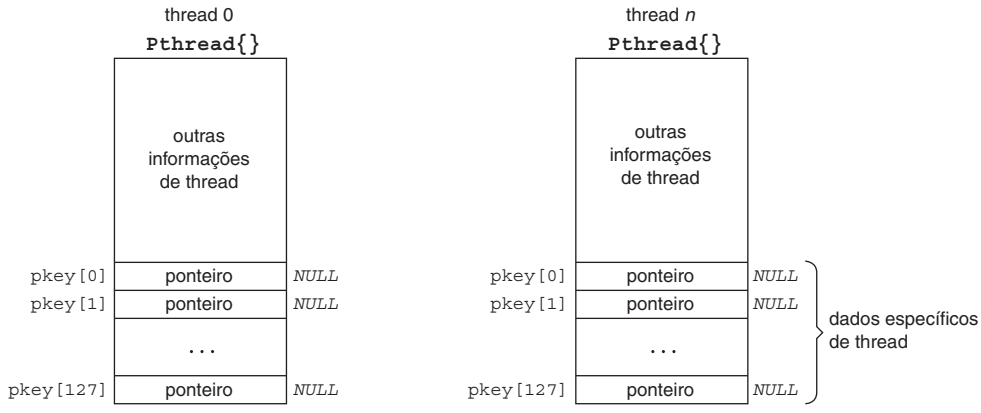


Figura 26.8 Informações mantidas pelo sistema sobre cada thread.

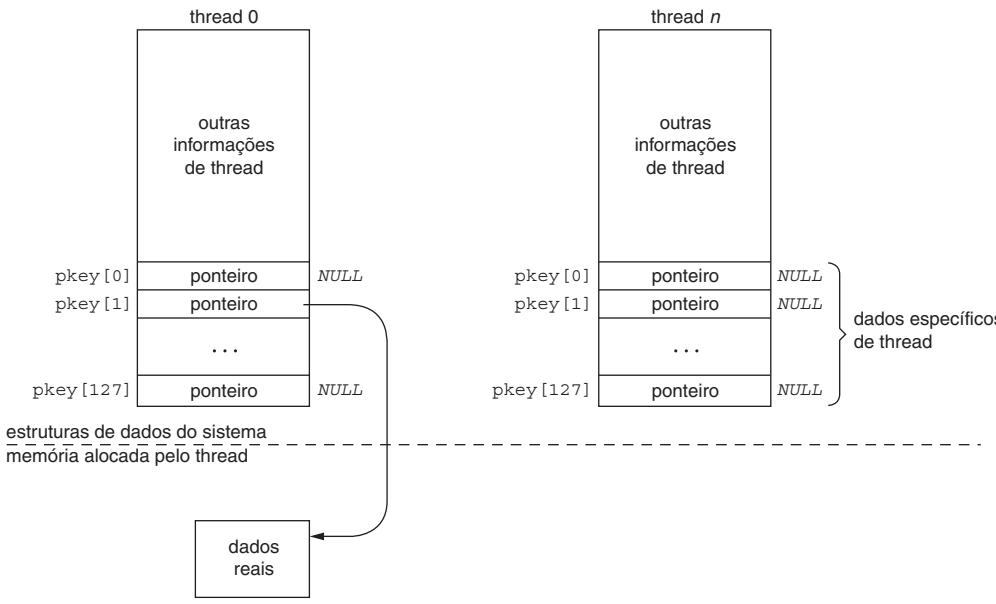


Figura 26.9 Associando uma região alocada usando `malloc` com ponteiro de dados específicos de thread.

`pthread_setspecific` faz é configurar o ponteiro para essa chave na estrutura `Pthread` para apontar para nossa memória alocada. De maneira semelhante, tudo que `pthread_getspecific` faz é retornar esse ponteiro para nós.

4. Outro thread, digamos, o thread *n*, chama `readline`, talvez enquanto o thread 0 ainda está executando dentro de `readline`.

   `readline` chama `pthread_once` para inicializar a chave para esse item de dados específicos de thread, mas, como ela já foi chamada, não será chamada novamente.

5. `readline` chama `pthread_getspecific` para buscar o ponteiro `pkey[1]` para esse thread e um ponteiro nulo é retornado. Então, esse thread chama `malloc`, seguido por `pthread_setspecific`, exatamente como o thread 0, inicializando seus dados específicos de thread para essa chave (1). Mostramos isso na Figura 26.10.

6. O thread *n* continua a executar em `readline`, utilizando e modificando seus próprios dados específicos de thread.

Um item que não tratamos é: o que acontece quando um thread termina? Se o thread tiver chamado nossa função `readline`, essa função alocou uma região da memória que precisa ser liberada. É aí que o "ponteiro destrutor" da Figura 26.7 é utilizado. Quando o thread que cria o item de dados específicos de thread chama `pthread_key_create`, um argumento dessa função é um ponteiro para uma função *destrutora*. Quando um thread termina, o sistema percorre o array `pkey` desse thread, chamando a função destrutora correspondente para cada ponteiro `pkey` não-nulo. O que queremos dizer com "destrutora correspondente" é o ponteiro de função armazenado no array `Key` da Figura 26.7. É assim que os dados específicos de thread são liberados quando um thread termina.

As duas primeiras funções que são normalmente chamadas ao se lidar com dados específicos de thread são `pthread_once` e `pthread_key_create`.

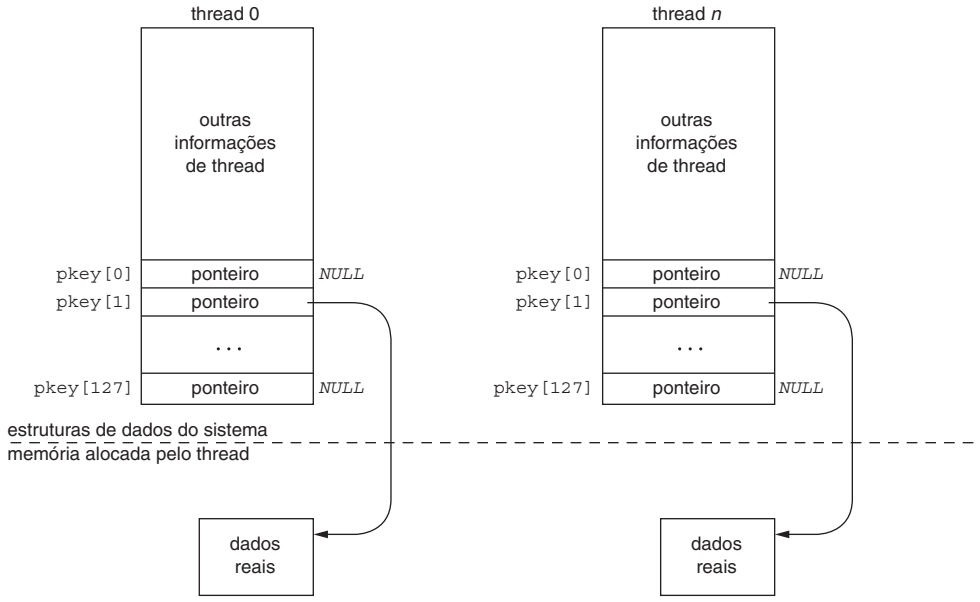


Figura 26.10 Estruturas de dados após o thread *n* inicializar seus dados específicos de thread.

```
#include <pthread.h>

int pthread_once(pthread_once_t *onceptr, void (*init)(void));

int pthread_key_create(pthread_key_t *keyptr, void (*destructor)(void *value));
```

As duas retornam: 0 se OK, valor E*xxx* positivo em caso de erro

`pthread_once` é normalmente chamada sempre que uma função que utiliza dados específicos de thread é chamada, mas `pthread_once` utiliza o valor da variável apontada por *onceptr* para garantir que a função *init* seja chamada somente uma vez por processo.

`pthread_key_create` deve ser chamada somente uma vez para determinada chave dentro de um processo. A chave é retornada por meio do ponteiro *keyptr* e a função *destrutora*, caso o argumento seja um ponteiro não-nulo, será chamada ao término de cada thread, se esse thread tiver armazenado um valor para essa chave.

O uso típico dessas duas funções (ignorando retornos de erro) é como segue:

```
pthread_key_t rl_key;
pthread_once_t rl_once = PTHREAD_ONCE_INIT;

void
readline_destructor(void *ptr)
{
   free(ptr);
}
void
readline_once(void)
{
   pthread_key_create(&rl_key, readline_destructor);
}
ssize_t
readline( ... )
```

```
{
    ...
    pthread_once(&rl_once, readline_once);
    if ( (ptr = pthread_getspecific(rl_key)) == NULL) {
        ptr = Malloc( ... );
        pthread_setspecific(rl_key, ptr);
        /* inicializa a memória apontada por ptr */
    }
    ...
    /* utiliza os valores apontados por ptr */
}
```

Sempre que `readline` é chamada, ela chama `pthread_once`. Essa função utiliza o valor apontado por seu argumento *onceptr* (o conteúdo da variável `rl_once`) para certificar-se de que sua função *init* seja chamada somente uma vez. Essa função de inicialização, `readline_once`, cria a chave de dados específicos de thread que é armazenada em `rl_key` e que `readline` utiliza então nas chamadas a `pthread_getspecific` e `pthread_setspecific`.

As funções `pthread_getspecific` e `pthread_setspecific` são utilizadas buscar e armazenar o valor associado a uma chave. Esse valor é o que chamamos de "ponteiro" na Figura 26.8. Para onde esse ponteiro aponta fica por conta da aplicação, mas normalmente é para memória alocada dinamicamente.

```
#include <pthread.h>

void *pthread_getspecific(pthread_key_t key);
```

Returna: ponteiro para dados específicos de thread (possivelmente um ponteiro nulo)

```
int pthread_setspecific(pthread_key_t key, const void *value);
```

Retorna: 0 se OK, valor E*xxx* positivo em caso de erro

Observe que o argumento para `pthread_key_create` é um ponteiro para a chave (pois essa função armazena o valor atribuído à chave), enquanto os argumentos para as funções `get` e `set` são a própria chave (provavelmente um pequeno índice de inteiros, conforme discutido anteriormente).

Exemplo: Função `readline` utilizando dados específicos de thread

Mostraremos agora um exemplo completo de dados específicos de thread, convertendo a versão otimizada de nossa função `readline` da Figura 3.18 para ser segura para threads, sem alterar a seqüência de chamada.

A Figura 26.11 mostra a primeira parte da função: a variável `pthread_key_t`, a variável `pthread_once_t`, a função `readline_destructor`, a função `readline_once` e nossa estrutura `Rline` que contém todas as informações que devemos manter por thread.

threads/readline.c

```
1 #include   "unpthread.h"
2 static pthread_key_t rl_key;
3 static pthread_once_t rl_once = PTHREAD_ONCE_INIT;
4 static void
5 readline_destructor(void *ptr)
```

Figura 26.11 Primeira parte da função `readline` segura para threads (*continua*).

```
 6 {
 7     free(ptr);
 8 }
 9 static void
10 readline_once(void)
11 {
12     Pthread_key_create(&rl_key, readline_destructor);
13 }
14 typedef struct {
15     int    rl_cnt;              /* inicializa como 0 */
16     char   *rl_bufptr;          /* inicializa como rl_buf */
17     char   rl_buf[MAXLINE];
18 } Rline;
```

threads/readline.c

Figura 26.11 Primeira parte da função `readline` segura para threads (*continuação*).

Função destrutora

4–8 Nossa função destrutora apenas libera a memória que foi alocada para esse thread.

Função de execução única

9–13 Veremos que nossa função de uma única execução é chamada uma única vez por `pthread_once` e apenas cria a chave que é utilizada por `readline`.

Estrutura `Rline`

14–18 Nossa estrutura `Rline` contém as três variáveis que causaram problema, sendo declaradas como `static` na Figura 3.18. Uma dessas estruturas será alocada dinamicamente para cada thread e então liberada pela nossa função destrutora.

A Figura 26.12 mostra a função `readline` real, além da função `my_read` que ela chama. Essa figura é uma modificação da Figura 3.18.

Função `my_read`

19–35 O primeiro argumento da função agora é um ponteiro para a estrutura `Rline` que foi alocada para esse thread (os dados específicos de thread reais).

Alocação de dados específicos de thread

42 Primeiro, chamamos `pthread_once` para que o primeiro thread a chamar `readline` nesse processo chame `readline_once` para criar a chave de dados específicos de thread.

Busca do ponteiro de dados específicos de thread

43–46 `pthread_getspecific` retorna o ponteiro para a estrutura `Rline` para esse thread. Mas, se essa é a primeira vez que esse thread chamou `readline`, o valor de retorno é um ponteiro nulo. Nesse caso, alocamos espaço para uma estrutura `Rline` e o membro `rl_cnt` é inicializado como 0 por `calloc`. Então, armazenamos o ponteiro para esse thread, chamando `pthread_setspecific`. Na próxima vez que esse thread chamar `readline`, `pthread_getspecific` retornará esse ponteiro que acaba de ser armazenado.

threads/readline.c

```
19 static ssize_t
20 my_read(Rline *tsd, int fd, char *ptr)
21 {
22     if (tsd->rl_cnt <= 0) {
```

Figura 26.12 Segunda parte da função `readline` segura para threads (*continua*).

```
23      again:
24        if ( (tsd->rl_cnt = read(fd, tsd->rl_buf, MAXLINE)) < 0) {
25            if (errno == EINTR)
26                goto again;
27            return (-1);
28        } else if (tsd->rl_cnt == 0)
29            return (0);
30        tsd->rl_bufptr = tsd->rl_buf;
31    }
32    tsd->rl_cnt--;
33    *ptr = *tsd->rl_bufptr++;
34    return (1);
35 }
36 ssize_t
37 readline(int fd, void *vptr, size_t maxlen)
38 {
39    size_t n, rc;
40    char   c, *ptr;
41    Rline  *tsd;
42    Pthread_once(&rl_once, readline_once);
43    if ( (tsd = pthread_getspecific(rl_key)) == NULL) {
44        tsd = Calloc(1, sizeof(Rline)); /* inicia como 0 */
45        Pthread_setspecific(rl_key, tsd);
46    }
47    ptr = vptr;
48    for (n = 1; n < maxlen; n++) {
49        if ( (rc = my_read(tsd, fd, &c)) == 1) {
50            *ptr++ = c;
51            if (c == '\n')
52                break;
53        } else if (rc == 0) {
54            *ptr = 0;
55            return (n - 1);       /* EOF, n - 1 bytes lidos */
56        } else
57            return (-1);          /* erro, errno configurado por read() */
58    }
59    *ptr = 0;
60    return (n);
61 }
```

threads/readline.c

Figura 26.12 Segunda parte da função `readline` segura para threads (*continuação*).

26.6 Cliente Web e conexões simultâneas (continuação)

Vamos rever agora o exemplo de cliente da Web da Seção 16.5 e recodificá-lo utilizando threads em vez de `connect` não-bloqueador. Com os threads, podemos deixar os soquetes em seu modo bloqueador-padrão e criar um thread por conexão. Cada thread pode ficar bloqueado em sua chamada a `connect`, enquanto o kernel executará algum outro thread que esteja pronto.

A Figura 26.13 mostra a primeira parte do programa, as variáveis globais e o início da função `main`.

Globais

1–16 Usamos `#include <thread.h>`, além do normal `<pthread.h>`, porque precisamos utilizar threads do Solaris além dos Pthreads, conforme descreveremos em breve.

10 Adicionamos um membro na estrutura `file`: `f_tid`, a ID do thread. O restante desse código é semelhante à Figura 16.15. Com essa versão de threads, não utilizamos `select` e, portanto, não precisamos de quaisquer configurações de descritor nem da variável `maxfd`.

36 A função `home_page` chamada é igual à da Figura 16.16.

threads/web01.c

```
 1 #include   "unpthread.h"
 2 #include   <thread.h>          /* threads Solaris */

 3 #define MAXFILES  20
 4 #define SERV      "80"         /* número da porta ou nome de serviço */

 5 struct file {
 6     char   *f_name;            /* nome de arquivo */
 7     char   *f_host;            /* nome de host ou endereço IP */
 8     int     f_fd;              /* descritor */
 9     int     f_flags;           /* F_xxx abaixo */
10     pthread_t f_tid;           /* ID de thread */
11 } file[MAXFILES];
12 #define F_CONNECTING  1       /* connect() em andamento */
13 #define F_READING     2       /* connect() completo; agora lendo */
14 #define F_DONE        4       /* tudo pronto */

15 #define GET_CMD   "GET %s HTTP/1.0\r\n\r\n"

16 int    nconn, nfiles, nlefttoconn, nlefttoread;

17 void   *do_get_read(void *);
18 void   home_page(const char *, const char *);
19 void   write_get_cmd(struct file *);

20 int
21 main(int argc, char **argv)
22 {
23     int    i, n, maxnconn;
24     pthread_t tid;
25     struct file *fptr;

26     if (argc < 5)
27         err_quit("usage: web <#conns> <IPaddr> <homepage> file1 ...");
28     maxnconn = atoi(argv[1]);
29     nfiles = min(argc - 4, MAXFILES);
30     for (i = 0; i < nfiles; i++) {
31         file[i].f_name = argv[i + 4];
32         file[i].f_host = argv[2];
33         file[i].f_flags = 0;
34     }
35     printf("nfiles = %d\n", nfiles);

36     home_page(argv[2], argv[3]);

37     nlefttoread = nlefttoconn = nfiles;
38     nconn = 0;
```

threads/web01.c

Figura 26.13 Globais e início da função de `main`.

A Figura 26.14 mostra o loop de processamento principal do thread `main`.

threads/web01.c

```
39     while (nlefttoread > 0) {
40         while (nconn < maxnconn && nlefttoconn > 0) {
41                /* localiza um arquivo para ler */
```

Figura 26.14 Loop de processamento principal da função `main` (*continua*).

```
42              for (i = 0; i < nfiles; i++)
43                  if (file[i].f_flags == 0)
44                      break;
45              if (i == nfiles)
46                  err_quit("nlefttoconn = %d but nothing found", nlefttoconn);
47              file[i].f_flags = F_CONNECTING;
48              Pthread_create(&tid, NULL, &do_get_read, &file[i]);
49              file[i].f_tid = tid;
50              nconn++;
51              nlefttoconn--;
52          }
53          if ( (n = thr_join(0, &tid, (void **) &fptr)) != 0)
54              errno = n, err_sys("thr_join error");
55          nconn--;
56          nlefttoread--;
57          printf("thread id %d for %s done\n", tid, fptr->f_name);
58      }
59      exit(0);
60  }
```

threads/web01.c

Figura 26.14 Loop de processamento principal da função `main` (*continuação*).

Criação de outro thread, quando possível

40–52 Se for permitido criar outro thread (`nconn` é menor que `maxnconn`), fazemos isso. A função que cada novo thread executa é `do_get_read` e o argumento é o ponteiro para a estrutura `file`.

Esperando qualquer thread terminar

53–54 Chamamos a função de thread do Solaris `thr_join` com o primeiro argumento igual a 0, para esperar que qualquer um de nossos threads termine. Infelizmente, os Pthreads não fornecem uma maneira de esperar que *qualquer* um de nossos threads termine; a função `pthread_join` nos faz especificar exatamente qual thread queremos esperar. Veremos, na Seção 26.9, que a solução de Pthreads para esse problema é mais complicada, exigindo utilizar uma variável condicional para o thread que está terminando de notificar o thread principal, quando estiver pronto.

> A solução que mostramos, utilizando a função de thread Solaris `thr_join`, não é portável para todos os ambientes. Contudo, queremos mostrar essa versão de nosso exemplo de cliente da Web utilizando threads, sem complicar a discussão com variáveis condicionais e mutexes. Felizmente, podemos misturar Pthreads com threads Solaris sob Solaris.

A Figura 26.15 mostra a função `do_get_read`, que é executada por cada thread. Essa função estabelece a conexão TCP, envia um comando HTTP `GET` para o servidor e lê a resposta deste.

threads/web01.c

```
61 void *
62 do_get_read(void *vptr)
63 {
64     int     fd, n;
65     char    line[MAXLINE];
66     struct file *fptr;
67     fptr = (struct file *) vptr;
```

Figura 26.15 Função `do_get_read` (*continua*).

```
68      fd = Tcp_connect(fptr->f_host, SERV);
69      fptr->f_fd = fd;
70      printf("do_get_read for %s, fd %d, thread %d\n",
71              fptr->f_name, fd, fptr->f_tid);
72      write_get_cmd(fptr);         /* grava o comando GET */
73          /* Lê a resposta do servidor */
74      for ( ; ; ) {
75          if ( (n = Read(fd, line, MAXLINE)) == 0)
76              break;               /* o servidor fechou a conexão */
77          printf("read %d bytes from %s\n", n, fptr->f_name);
78      }
79      printf("end-of-file on %s\n", fptr->f_name);
80      Close(fd);
81      fptr->f_flags = F_DONE;      /* limpa F_READING */
82      return (fptr);               /* termina o thread */
83  }
```

threads/web01.c

Figura 26.15 Função `do_get_read` (*continuação*).

Criação de soquete TCP, estabelecimento da conexão

68–71 Um soquete TCP é criado e uma conexão é estabelecida por nossa função `tcp_connect`. O soquete é um soquete bloqueador normal, de modo que o thread bloqueará na chamada de `connect` até que a conexão seja estabelecida.

Gravação da solicitação no servidor

72 `write_get_cmd` constrói o comando HTTP `GET` e o envia para o servidor. Não mostramos essa função novamente, pois a única diferença em relação à Figura 16.18 é que a versão de threads não chama `FD_SET` e não utiliza `maxfd`.

Leitura da resposta do servidor

73–82 A resposta do servidor é, então, lida. Quando a conexão é fechada pelo servidor, o flag `F_DONE` é configurado e a função retorna, terminando o thread.

Também não mostramos a função `home_page`, pois ela é idêntica à versão mostrada na Figura 16.16.

Retornaremos a este exemplo, substituindo a função Solaris `thr_join` pela solução de Pthreads mais portável, mas devemos primeiro discutir os mutexes e as variáveis condicionais.

26.7 Mutexes: exclusão mútua

Observe, na Figura 26.14, que, quando um thread termina, o loop principal decrementa `nconn` e `nlefttoread`. Poderíamos ter colocado esses dois decrementos na função `do_get_read`, permitindo que cada thread decrementasse esses dois contadores imediatamente, antes de terminar. Mas isso seria um erro de programação concorrente sutil, embora significativo.

O problema de colocar o código na função que cada thread executa é que essas duas variáveis são globais e não específicas do thread. Se um thread está no meio da decrementação de uma variável, ele é suspenso, e se outro thread executa e decrementa a mesma variável, um erro pode resultar. Por exemplo, suponha que o compilador C transforme o operador de decre-

mento em três instruções: carregar da memória para um registrador, decrementar o registrador e armazenar do registrador na memória. Considere o possível cenário a seguir:

1. O thread A está executando e carrega o valor de `nconn` (3) em um registrador.
2. O sistema troca do thread A para o B. Os registradores de A são salvos e os de B são restaurados.
3. O thread B executa as três instruções correspondentes à expressão C `nconn--`, armazenando o novo valor 2.
4. Algum tempo depois, o sistema troca do thread B para o A. Os registradores de A são restaurados e A continua de onde parou, na segunda instrução de máquina da seqüência de três instruções. O valor do registrador é decrementado de 3 para 2 e o valor 2 é armazenado em `nconn`.

O resultado final é que `nconn` é 2, quando deveria ser 1. Isso está errado.

Esses tipos de erros de programação concorrente são difíceis de localizar por várias razões. Primeiro, eles ocorrem raramente. Contudo, são um erro e haverá uma falha.

Segundo, o erro é difícil de reproduzir, pois depende da sincronização não-determinista de muitos eventos. Por último, em alguns sistemas, as instruções de hardware podem ser atômicas; isto é, pode haver uma instrução de hardware para decrementar um inteiro na memória (em vez da seqüência de três instruções que supomos anteriormente) e o hardware não pode ser interrompido durante essa instrução. Mas isso não é garantido por todos os sistemas; portanto, o código funciona em um sistema, mas não em outro.

Chamamos a programação de threads de *programação concorrente* ou *programação paralela*, pois vários threads podem estar executando concomitantemente (em paralelo), acessando as mesmas variáveis. Embora o cenário de erro que acabamos de discutir suponha um sistema de uma única CPU, o potencial para erro também existe se os threads A e B estiverem executando simultaneamente em diferentes CPUs, em um sistema de multiprocessador. Com a programação de Unix normal, não encontramos esses problemas de programação concorrente, porque com `fork` nada além dos descritores é compartilhado entre o pai e o filho. Entretanto, encontraremos esse mesmo tipo de problema ao discutirmos memória compartilhada entre processos.

Podemos demonstrar facilmente esse problema com threads. A Figura 26.17 é um programa simples que cria dois threads e então faz com que cada um deles incremente uma variável global 5.000 vezes.

Agravamos a possibilidade de haver problemas, buscando o valor corrente de `counter`, imprimindo o novo valor e depois armazenando esse novo valor. Se executarmos esse programa, teremos a saída mostrada na Figura 26.16.

```
4: 1
4: 2
4: 3
4: 4
                continua quando o thread 4 executa
4: 517
4: 518
5: 518          agora, o thread 5 executa
5: 519
5: 520
                continua quando o threads 5 executa
5: 926
5: 927
4: 519          agora, o thread 4 executa; o valor armazenado está errado
4: 520
```

Figura 26.16 Saída do programa da Figura 26.17.

threads/example01.c

```
 1 #include   "unpthread.h"
 2 #define NLOOP 5000
 3 int    counter;                 /* incrementado pelos threads */
 4 void   *doit(void *);
 5 int
 6 main(int argc, char **argv)
 7 {
 8     pthread_t tidA, tidB;
 9     Pthread_create(&tidA, NULL, &doit, NULL);
10     Pthread_create(&tidB, NULL, &doit, NULL);
11         /* espera que os dois threads terminem */
12     Pthread_join(tidA, NULL);
13     Pthread_join(tidB, NULL);
14     exit(0);
15 }
16 void *
17 doit(void *vptr)
18 {
19     int    i, val;
20     /*
21      * Cada thread busca, imprime e incrementa o contador NLOOP vezes.
22      * O valor do contador deve aumentar monotonicamente.
23      */
24     for (i = 0; i < NLOOP; i++) {
25         val = counter;
26         printf("%d: %d\n", pthread_self(), val + 1);
27         counter = val + 1;
28     }
29     return (NULL);
30 }
```

threads/example01.c

Figura 26.17 Dois threads que incrementam uma variável global incorretamente.

Observe o erro na primeira vez que o sistema troca do thread 4 para o thread 5: o valor 518 é armazenado por cada thread. Isso acontece numerosas vezes nas 10.000 linhas de saída.

A natureza não-determinista desse tipo de problema também fica evidente se executarmos o programa algumas vezes: a cada vez, o resultado final é diferente da execução anterior do programa. Além disso, se redirecionarmos a saída para um arquivo em disco, às vezes o erro não ocorrerá, pois o programa executará mais rapidamente, fornecendo menos oportunidades de troca entre os threads. O maior número de erros ocorre quando executamos o programa interativamente, escrevendo a saída no terminal (lento), mas salvando a saída em um arquivo, utilizando o programa Unix `script` (discutido em detalhes no Capítulo 19 de APUE).

O problema que acabamos de discutir – vários threads atualizando uma variável compartilhada – é o mais simples. A solução é proteger a variável compartilhada com um *mutex* (que significa "exclusão mútua") e acessá-la somente quando tivermos o mutex. Em termos de Pthreads, um mutex é uma variável de tipo `pthread_mutex_t`. Bloqueamos e desbloqueamos um mutex utilizando as duas funções a seguir:

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mptr);

int pthread_mutex_unlock(pthread_mutex_t *mptr);
```

As duas retornam: 0 se OK, valor E*xxx* positivo em caso de erro

Se tentarmos bloquear um mutex que já está bloqueado por algum outro thread, seremos impedidos até que ele seja desbloqueado.

Se uma variável de mutex é alocada estaticamente, devemos inicializá-la com a constante `PTHREAD_MUTEX_INITIALIZER`. Veremos, na Seção 30.8, que, se alocarmos um mutex na memória compartilhada, deveremos inicializá-lo em tempo de execução chamando a função `pthread_mutex_init`.

> Alguns sistemas (por exemplo, Solaris) definem `PTHREAD_MUTEX_INITIALIZER` como 0; portanto, omitir essa inicialização é aceitável, pois as variáveis alocadas estaticamente são inicializadas como 0 automaticamente. Mas não há nenhuma garantia de que isso seja aceitável e outros sistemas (por exemplo, Digital Unix) definem o inicializador como sendo não-zero.

A Figura 26.18 é uma versão corrigida da Figura 26.17, que utiliza um único mutex para bloquear o contador entre os dois threads.

threads/example02.c

```
 1 #include  "unpthread.h"
 2 #define NLOOP 5000
 3 int counter;                     /* incrementado pelos threads */
 4 pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;
 5 void  *doit(void *);
 6 int
 7 main(int argc, char **argv)
 8 {
 9     pthread_t tidA, tidB;
10     Pthread_create(&tidA, NULL, &doit, NULL);
11     Pthread_create(&tidB, NULL, &doit, NULL);
12         /* espera que os dois threads terminem */
13     Pthread_join(tidA, NULL);
14     Pthread_join(tidB, NULL);
15 exit(0);
16 }
17 void *
18 doit(void *vptr)
19 {
20     int    i, val;
21     /*
22      * Cada thread busca, imprime e incrementa o contador NLOOP vezes.
23      * O valor do contador deve aumentar monotonicamente.
24      */
25     for (i = 0; i < NLOOP; i++) {
26        Pthread_mutex_lock(&counter_mutex);
```

Figura 26.18 Versão corrigida da Figura 26.17 utilizando um mutex para proteger a variável compartilhada (*continua*).

```
27          val = counter;
28          printf("%d: %d\n", pthread_self(), val + 1);
29          counter = val + 1;

30          Pthread_mutex_unlock(&counter_mutex);
31      }

32      return (NULL);
33  }
```

threads/example02.c

Figura 26.18 Versão corrigida da Figura 26.17 utilizando um mutex para proteger a variável compartilhada (*continuação*).

Declaramos um mutex chamado `counter_mutex` que deve ser bloqueado pelo thread antes que este manipule a variável `counter`. Quando executamos esse programa, a saída está sempre correta: o valor é incrementado monotonicamente e o valor final impresso é sempre 10.000.

Qual é a sobrecarga envolvida no bloqueio do mutex? Os programas das Figuras 26.17 e 26.18 foram alterados para fazer loop 50.000 vezes e cronometrados enquanto a saída era dirigida para `/dev/null`. A diferença, em tempo de CPU, da versão incorreta sem mutex para a versão correta que utilizou um mutex foi de 10%. Isso nos diz que o bloqueio com mutex não é uma grande sobrecarga.

26.8 Variáveis de condição

Um mutex serve bem para evitar acesso simultâneo a uma variável compartilhada, mas precisamos de algo mais para nos permitir dormir enquanto esperamos que alguma condição ocorra. Vamos demonstrar isso com um exemplo. Retornamos para nosso cliente Web da Seção 26.6 e substituímos a função `thr_join` do Solaris por `pthread_join`. Mas não podemos chamar a função Pthread até sabermos que um thread terminou. Primeiro, declaramos uma variável global que conta o número de threads terminados e o protege com um mutex.

```
int             ndone;        /* número de threads terminados */
pthread_mutex_t ndone_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Então, exigimos que cada thread incremente esse contador ao terminar, tendo o cuidado de utilizar o mutex associado.

```
void *
do_get_read(void *vptr)
{
    ...

    Pthread_mutex_lock(&ndone_mutex);
    ndone++;
    Pthread_mutex_unlock(&ndone_mutex);

    return(fptr);        /* termina o thread */
}
```

Isso está bem, mas como codificamos o loop principal? Ele precisa bloquear o mutex continuamente e verificar se quaisquer threads terminaram.

```
while (nlefttoread > 0) {
    while (nconn < maxnconn && nlefttoconn > 0) {
          /* localiza um arquivo para ler */
       ...
    }

       /* Vê se um dos threads está pronto */
    Pthread_mutex_lock(&ndone_mutex);
```

```
        if (ndone > 0) {
           for (i = 0; i < nfiles; i++) {
                if (file[i].f_flags & F_DONE) {
                    Pthread_join(file[i].f_tid, (void **) &fptr);

                 /* atualiza file[i] para thread terminado */
                 ...
              }
           }
        }
        Pthread_mutex_unlock(&ndone_mutex);
    }
```

Embora isso esteja certo, significa que o loop principal *nunca* vai dormir; ele apenas repete, verificando `ndone` sempre que percorre o loop. Isso é chamado de *consulta seqüencial* e é considerado um desperdício de tempo de CPU.

Queremos um método para que o loop principal durma até que um de seus threads o notifique de que algo está pronto. Uma *variável condicional*, em conjunto com um mutex, fornece essa facilidade. O mutex fornece exclusão mútua e a variável condicional um mecanismo de sinalização.

Em termos de Pthreads, uma variável condicional é uma variável de tipo `pthread_cond_t`. Elas são utilizadas com as duas funções a seguir:

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *cptr, pthread_mutex_t *mptr);

int pthread_cond_signal(pthread_cond_t *cptr);
```

As duas retornam: 0 se OK, valor E*xxx* positivo em caso de erro

O termo "signal" no nome da segunda função não se refere a um sinal `SIG`*xxx* do Unix.

Um exemplo é a maneira mais fácil de explicar essas funções. Retornando ao nosso exemplo de cliente da Web, o contador `ndone` agora está associado a uma variável condicional e a um mutex.

```
int             ndone;
pthread_mutex_t ndone_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  ndone_cond  = PTHREAD_COND_INITIALIZER;
```

Um thread notifica o loop principal de que está terminando, incrementando o contador enquanto seu bloqueio de mutex é mantido, e sinalizando a variável condicional.

```
Pthread_mutex_lock(&ndone_mutex);
ndone++;
Pthread_cond_signal(&ndone_cond);
Pthread_mutex_unlock(&ndone_mutex);
```

Então, o loop principal bloqueia em uma chamada a `pthread_cond_wait`, esperando para ser sinalizado por um thread que esteja terminando.

```
while (nlefttoread > 0) {
   while (nconn < maxnconn && nlefttoconn > 0) {
         /* localiza o arquivo para ler */
      ...
   }

      /* Espera o thread terminar */
   Pthread_mutex_lock(&ndone_mutex);
   while (ndone == 0)
      Pthread_cond_wait(&ndone_cond, &ndone_mutex);

   for (i = 0; i < nfiles; i++) {
```

```
            if (file[i].f_flags & F_DONE) {
                Pthread_join(file[i].f_tid, (void **) &fptr);

                /* atualiza file[i] para thread terminado */
                ...
            }
        }
        Pthread_mutex_unlock(&ndone_mutex);
    }
```

Observe que a variável `ndone` ainda é verificada somente enquanto o mutex é mantido. Então, se não houver nada a fazer, `pthread_cond_wait` é chamada. Isso coloca o thread chamador para dormir *e* libera o bloqueio de mutex que ele mantém. Além disso, quando o thread retorna de `pthread_cond_wait` (depois que algum outro thread sinalizou para isso), ele mantém novamente o mutex.

Por que um mutex é sempre associado a uma variável condicional? A "condição" normalmente é o valor de alguma variável compartilhada entre os threads. O mutex é exigido para permitir que essa variável seja configurada e testada pelos diferentes threads. Por exemplo, se não tivéssemos o mutex no código do exemplo que acabamos de mostrar, o loop principal o testaria como segue:

```
        /* Espera o thread terminar */
    while (ndone == 0)
        Pthread_cond_wait(&ndone_cond, &ndone_mutex);
```

Mas há uma possibilidade de que o último dos threads incremente `ndone` depois do teste de `ndone == 0`, mas antes da chamada a `pthread_cond_wait`. Se isso acontecer, esse último "sinal" seria perdido e o loop principal bloquearia para sempre, esperando por algo que nunca ocorrerá novamente.

Essa é a mesma razão pela qual `pthread_cond_wait` deve ser chamada com o mutex associado bloqueado e pela qual essa função desbloqueia o mutex e coloca o thread chamador para dormir como uma única operação atômica. Se essa função não desbloqueasse o mutex e depois o bloqueasse novamente ao retornar, o thread teria que desbloquear e bloquear o mutex, e o código seria parecido com o seguinte:

```
        /* Espera o thread terminar */
    Pthread_mutex_lock(&ndone_mutex);
    while (ndone == 0) {
        Pthread_mutex_unlock(&ndone_mutex);
        Pthread_cond_wait(&ndone_cond, &ndone_mutex);
        Pthread_mutex_lock(&ndone_mutex);
    }
```

Mas, novamente, há uma possibilidade de que o thread final possa terminar e incrementar o valor de `ndone` entre a chamada a `pthread_mutex_unlock` e `pthread_cond_wait`.

Normalmente, `pthread_cond_signal` desperta um thread que está esperando na variável condicional. Há ocasiões em que um thread sabe que vários outros devem ser despertados, nesse caso, `pthread_cond_broadcast` despertará *todos* os threads que estão bloqueados na variável condicional.

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cptr);

int pthread_cond_timedwait(pthread_cond_t *cptr, pthread_mutex_t *mptr,
                           const struct timespec *abstime);
```

As duas retornam: 0 se OK, valor E*xxx* positivo em caso de erro

`pthread_cond_timedwait` permite que um thread imponha um limite sobre quanto tempo ele bloqueará. *abstime* é uma estrutura `timespec` (conforme definimos com a função `pselect`, Seção 6.9) que especifica o tempo de sistema quando a função deve retornar, mesmo que a variável condicional ainda não tenha sido sinalizada. Se esse tempo-limite ocorrer, `ETIME` é retornado.

Esse valor é um *tempo absoluto*; não é um *delta de tempo*. Isto é, *abstime* é o tempo do sistema – o número de segundos e nanossegundos passados desde 1º de janeiro de 1970, UTC – quando a função deve retornar. Isso difere de `select` e de `pselect`, que especificam o número de segundos e microssegundos (nanossegundos para `pselect`) até algum tempo no futuro quando a função deve retornar. O procedimento normal é chamar `gettimeofday` para obter o tempo atual (como uma estrutura `timeval`!) e copiá-lo para uma estrutura `timespec`, adicionando o limite de tempo desejado. Por exemplo:

```
struct timeval tv;
struct timespec ts;

if (gettimeofday(&tv, NULL) < 0)
    err_sys("gettimeofday error");
ts.tv_sec = tv.tv_sec + 5;        /* 5 segundos no futuro */
ts.tv_nsec = tv.tv_usec * 1000;   /* microssegundos para nanossegundos */

pthread_cond_timedwait( ... , &ts);
```

A vantagem de utilizar um tempo absoluto em vez de um tempo delta se dá no caso de a função retornar prematuramente (talvez por causa de um sinal capturado), a qual pode ser chamada de novo, sem ter de alterar o conteúdo da estrutura `timespec`. A desvantagem, entretanto, é ter de chamar `gettimeofday` antes que a função possa ser chamada pela primeira vez.

> A especificação POSIX define uma função `clock_gettime` que retorna o tempo atual como uma estrutura `timespec`.

26.9 Cliente Web e conexões simultâneas (continuação)

Agora, recodificaremos nosso cliente Web da Seção 26.6, removendo a chamada para a função `thr_join` do Solaris e substituindo-a por uma chamada para `pthread_join`. Conforme discutido naquela seção, devemos agora especificar exatamente quais threads estamos esperando. Para fazer isso, utilizaremos uma variável condicional, conforme descrito na Seção 26.8.

A única alteração nas globais (Figura 26.13) é adicionar um novo flag e a variável condicional.

```
#define   F_JOINED          8   /* principal usou pthread_join */

int                ndone;       /* número de threads terminados */
pthread_mutex_t    ndone_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t     ndone_cond  = PTHREAD_COND_INITIALIZER;
```

A única alteração na função `do_get_read` (Figura 26.15) é incrementar `ndone` e sinalizar o loop principal antes de o thread terminar.

```
    printf("end-of-file on %s\n", fptr->f_name);
    Close(fd);

    Pthread_mutex_lock(&ndone_mutex);
    fptr->f_flags = F_DONE;     /* limpa F_READING */
    ndone++;
    Pthread_cond_signal(&ndone_cond);
    Pthread_mutex_unlock(&ndone_mutex);

    return(fptr);         /* termina o thread */
}
```

A maioria das alterações está no loop principal (Figura 26.14), cuja nova versão mostramos na Figura 26.19.

threads/web03.c

```
43      while (nlefttoread > 0) {
44          while (nconn < maxnconn && nlefttoconn > 0) {
45                  /* localiza um arquivo para ler */
46              for (i = 0; i < nfiles; i++)
47                  if (file[i].f_flags == 0)
48                      break;
49              if (i == nfiles)
50                  err_quit("nlefttoconn = %d but nothing found", nlefttoconn);
51              file[i].f_flags = F_CONNECTING;
52              Pthread_create(&tid, NULL, &do_get_read, &file[i]);
53              file[i].f_tid = tid;
54              nconn++;
55              nlefttoconn--;
56          }
57              /* Espera o thread terminar */
58          Pthread_mutex_lock(&ndone_mutex);
59          while (ndone == 0)
60              Pthread_cond_wait(&ndone_cond, &ndone_mutex);
61          for (i = 0; i < nfiles; i++) {
62              if (file[i].f_flags & F_DONE) {
63                  Pthread_join(file[i].f_tid, (void **) &fptr);
64                  if (&file[i] != fptr)
65                      err_quit("file[i] != fptr");
66                  fptr->f_flags = F_JOINED; /* limpa F_DONE */
67                  ndone--;
68                  nconn--;
69                  nlefttoread--;
70                  printf("thread %d for %s done\n", fptr->f_tid, fptr->f_name);
71              }
72          }
73          Pthread_mutex_unlock(&ndone_mutex);
74      }
75      exit(0);
76  }
```

threads/web03.c

Figura 26.19 Loop de processamento principal da função `main`.

Criação de outro thread, quando possível

44–56 Esse código não mudou.

Esperando o thread terminar

57–60 Para esperar que um dos threads termine, esperamos que `ndone` seja não-zero. Conforme discutido na Seção 26.8, o teste deve ser feito enquanto o mutex está bloqueado. O sono, ou repouso, é realizado por `pthread_cond_wait`.

Tratando o thread terminado

61–73 Quando um thread tiver terminado, percorremos todas as estruturas `file` para localizar o thread apropriado, chamamos `pthread_join` e então configuramos o novo flag `F_JOINED`.

A Figura 16.20 mostra os tempos de execução para essa versão, junto com os tempos da versão que utiliza `connect` não-bloqueador.

26.10 Resumo

A criação de um novo thread normalmente é mais rápida que a de um novo processo com `fork`. Só isso já pode ser uma vantagem em servidores de rede muito utilizados. Entretanto, a programação com threads é um novo paradigma que exige mais disciplina.

Todos os threads de um processo compartilham variáveis globais e descritores, permitindo que essas informações sejam compartilhadas entre diferentes threads. Mas esse compartilhamento introduz problemas de sincronização e as primitivas de sincronização Pthread que devemos utilizar são mutexes e variáveis condicionais. A sincronização de dados compartilhados é uma parte obrigatória de quase todas as aplicações com threads.

Ao se escrever funções que podem ser chamadas por aplicações com threads, essas funções devem ser seguras para threads. Os dados específicos de thread representam uma técnica que ajuda nisso e mostramos um exemplo com nossa função `readline`.

Retornaremos ao modelo de threads no Capítulo 30, com outro projeto de servidor, no qual o servidor cria um pool de threads ao iniciar. Um thread disponível desse pool trata a próxima solicitação de cliente.

Exercícios

26.1 Compare o uso de descritor em um servidor que utiliza `fork` com um servidor que utiliza thread, supondo que 100 clientes estão sendo atendidos ao mesmo tempo.

26.2 O que acontece na Figura 26.3 se o thread não fecha (`close`) o soquete conectado quando `str_echo` retorna?

26.3 Nas Figuras 5.5 e 6.13, imprimimos "servidor terminou prematuramente" quando esperamos uma linha ecoada do servidor, mas em vez disso recebemos um EOF (lembre-se da Seção 5.12). Modifique a Figura 26.2 para imprimir essa mensagem também, quando for apropriado.

26.4 Modifique as Figuras 26.11 e 26.12 de modo que elas possam compilar em um sistema que não suporte threads.

26.5 Para ver o erro da função `readline` que é utilizada na Figura 26.3, construa aquele programa e inicie o servidor. Então, construa o cliente TCP de eco da Figura 6.13 que funcione corretamente em um modo de lote. Localize um arquivo de texto grande em seu sistema e inicie o cliente três vezes em um modo de lote, lendo o arquivo de texto grande e gravando sua saída em um arquivo temporário. Se possível, execute os clientes em um host diferente do servidor. Se os três clientes terminarem corretamente (freqüentemente eles travam), veja seus arquivos de saída temporários e compare-os com o arquivo de entrada.

Agora, construa uma versão do servidor utilizando a função `readline` correta da Seção 26.5. Execute novamente o teste com três clientes; todos eles devem funcionar agora. Você também deve colocar uma instrução `printf` na função `readline_destructor`, na função `readline_once` e na chamada a `malloc` em `readline`. Isso mostra que a chave é criada somente uma vez, mas a memória é alocada para cada thread, e a função destrutora é chamada para cada thread.

Capítulo 27

Opções IP

27.1 Visão geral

O IPv4 permite até 40 bytes de opções para acompanhar o cabeçalho de 20 bytes corrigido. Embora 10 opções diferentes estejam definidas, a mais comumente utilizada é a de rota da origem. O acesso a essas opções se dá por meio da opção de soquete `IP_OPTIONS` e demonstraremos isso com um exemplo que utiliza roteamento da origem.

O IPv6 permite que cabeçalhos de extensão ocorram entre o cabeçalho IPv6 de 40 bytes corrigido e o cabeçalho de camada de transporte (por exemplo, ICMPv6, TCP ou UDP). Seis cabeçalhos de extensão diferentes estão definidos atualmente. Ao contrário da estratégia do IPv4, o acesso aos cabeçalhos de extensão do IPv6 se dá por meio de uma interface funcional, em vez de obrigar o usuário a entender os detalhes reais de como os cabeçalhos aparecem no pacote IPv6.

27.2 Opções IPv4

Na Figura A.1, mostramos as opções que acompanham o cabeçalho IPv4 de 20 bytes. Conforme se observa lá, o campo de comprimento de cabeçalho de 4 bits limita o tamanho total do cabeçalho IPv4 a 15 palavras de 32 bits (60 bytes); portanto, o tamanho das opções de IP está limitado em 40 bytes. Dez opções diferentes estão definidas para IPv4:

1. NOP: nenhuma operação – Uma opção de um byte geralmente utilizada para preenchimento, para fazer uma opção posterior cair em um limite de quatro bytes.
2. EOL: fim-de-lista – Uma opção de um byte que termina o processamento de opção. Como o tamanho total das opções de IP deve ser um múltiplo de quatro bytes, alguns bytes EOL podem seguir a opção final.
3. LSRR: origem vaga e rota do registro (Seção 8.5 do TCPv1) – Mostraremos um exemplo disso em breve.
4. SSRR: origem estrita e rota do registro (Seção 8.5 do TCPv1) – Mostraremos um exemplo disso em breve.

5. Indicação de tempo – *timestamp* (Seção 7.4 do TCPv1).
6. Rota de registro (Seção 7.3 do TCPv1).
7. Segurança básica (obsoleta).
8. Segurança estendida (obsoleta).
9. Identificador de fluxo (obsoleto).
10. Alerta de roteador – Essa opção está descrita no RFC 2113 (Katz, 1997). Ela é incluída em datagramas de IP que devem ser examinados por todos os roteadores que encaminham o datagrama.

O Capítulo 9 do TCPv2 fornece mais detalhes sobre o processamento de kernel das seis primeiras opções e as seções indicadas no TCPv1 fornecem exemplos de sua utilização.

As funções `getsockopt` e `setsockopt` (com um *nível* de `IPPROTO_IP` e um *optname* de `IP_OPTIONS`) buscam e configuram as opções de IP. O quarto argumento de `getsockopt` e `setsockopt` é um ponteiro para um buffer (cujo tamanho é de 44 bytes ou menos) e o quinto argumento é o tamanho desse buffer. A razão pela qual o tamanho desse buffer para `getsockopt` pode ser quatro bytes maior que o tamanho máximo das opções deve-se à maneira como a opção de rota de origem é tratada, conforme descreveremos em breve. Diferentemente das duas opções de rota de origem, o formato do que entra no buffer é o formato das opções, quando colocadas no datagrama de IP.

Quando as opções de IP forem configuradas utilizando `setsockopt`, as opções especificadas serão então enviadas em todos os datagramas de IP naquele soquete. Isso funciona para TCP, UDP e soquetes de IP brutos. Para limpar essas opções, chame `setsockopt` e especifique um ponteiro nulo como o quarto argumento ou um valor 0 como o quinto argumento (o comprimento).

> A configuração das opções de IP para um soquete de IP bruto não funciona em todas as implementações, caso a opção de soquete `IP_HDRINCL` (que descreveremos no próximo capítulo) também estiver configurada. Muitas implementações derivadas do Berkeley não enviam as opções configuradas com `IP_OPTIONS`, quando `IP_HDRINCL` está ativado, pois a aplicação pode configurar suas próprias opções de IP no cabeçalho de IP que constrói (páginas 1056 e 1057 do TCPv2). Outros sistemas (por exemplo, FreeBSD) permitem que a aplicação especifique opções de IP utilizando a opção de soquete `IP_OPTIONS` ou configurando `IP_HDRINCL` e os incluindo no cabeçalho de IP que constrói, mas não ambos.

Quando `getsockopt` é chamada para buscar as opções de IP para um soquete TCP conectado que foi criado por `accept`, é retornado apenas o inverso da opção de rota de origem recebida com o SYN do cliente para o soquete receptor (página 931 do TCPv2). A rota de origem é automaticamente invertida pelo TCP, porque a rota de origem especificada pelo cliente era deste para o servidor, mas o servidor precisa utilizar o inverso dessa rota em datagramas que envia para o cliente. Se nenhuma rota de origem tiver acompanhado o SYN, então o comprimento do valor-resultado retornado por `getsockopt` por meio de seu quinto argumento será 0. Para todos os outros soquetes TCP e para todos os soquetes UDP e soquetes IP brutos, chamar `getsockopt` para buscar as opções de IP retorna somente uma cópia das opções de IP que foram configuradas por `setsockopt` para o soquete. Observe que, para um soquete IP bruto, o cabeçalho de IP recebido, incluindo todas as opções de IP, é sempre retornado pelas funções de entrada; portanto, as opções de IP recebidas estão sempre disponíveis.

> Os kernels derivados do Berkeley nunca retornaram uma rota de origem recebida, nem quaisquer outras opções de IP, para um soquete UDP. O código mostrado na página 775 do TCPv2, para retornar as opções de IP, existia desde a 4.3BSD Reno, mas sempre foi desativado, pois não funciona. Isso torna impossível para uma aplicação de UDP utilizar o inverso de uma rota recebida para datagramas, de volta para o remetente.

27.3 Opções de rota de origem do IPv4

Uma *rota de origem* é uma lista de endereços IP especificados pelo remetente do datagrama de IP. Se a rota de origem é *estrita*, então o datagrama deve passar por cada nó listado e somente pelos nós listados. Isto é, todos os nós listados na rota de origem devem ser vizinhos. Porém, se a rota de origem é *vaga*, o datagrama deve passar por cada nó listado, mas também pode passar por outros nós que não aparecem na rota de origem.

> O roteamento de origem do IPv4 é controverso. Embora ele possa ser muito útil para depuração de rede, pode ser utilizado para "spoofing de endereço de origem" e outros tipos de ataques. Cheswick, Bellovin e Rubin (2003) defendem a desativação do recurso em todos os seus roteadores e muitas organizações e provedores de serviço fazem isso. Uma utilização legítima do roteamento de origem é para detectar rotas assimétricas que utilizam o programa `traceroute`, como demonstrado nas páginas 108 e 109 do TCPv1, mas com cada vez mais roteadores na Internet desativando o roteamento de origem, até mesmo isso tende a desaparecer. Contudo, especificar e receber rotas de origem faz parte da API de soquetes e precisa ser descrito.

As rotas de origem IPv4 são chamadas de *rotas de origem e registro* (LSRR para a opção vaga e SSRR para a opção estrita), porque quando um datagrama passa por todos os nós listados, cada um substitui seu endereço listado pelo endereço da interface de saída. Isso permite que o receptor pegue essa nova lista e a inverta para seguir o caminho inverso de volta ao remetente. Exemplos dessas duas rotas de origem, junto com a saída de `tcpdump` correspondente, são encontrados na Seção 8.5 do TCPv1.

Especificamos uma rota de origem como um array de endereços IPv4, prefixados com três campos de um byte, como mostrado na Figura 27.1. Esse é o formato do buffer que passaremos para `setsockopt`.

Colocamos um NOP antes da opção de rota de origem, o que faz todos os endereços IP serem alinhados em um limite de quatro bytes. Isso não é exigido, mas não requer nenhum espaço adicional (as opções de IP são sempre preenchidas de modo a serem um múltiplo de quatro bytes) e alinha os endereços.

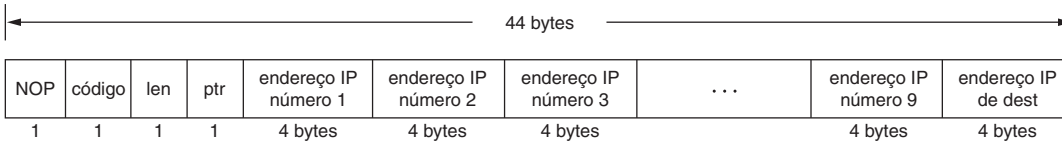


Figura 27.1 Passando uma rota de origem para o kernel.

Nessa figura, mostramos até 10 endereços IP na rota, mas o primeiro endereço listado é removido da opção de rota de origem e torna-se o endereço de destino do datagrama de IP, quando deixa o host de origem. Embora haja espaço para somente nove endereços IP no espaço da opção de IP de 40 bytes (não se esqueça do cabeçalho de opção de 3 bytes que estamos para descrever), há na verdade 10 endereços IP em um cabeçalho IPv4, quando o endereço de destino é incluído.

O *código* é `0x83` para uma opção LSRR ou `0x89` para uma opção SSRR. O item *len* que especificamos é o tamanho da opção em bytes, incluindo o cabeçalho de três bytes e o endereço de destino extra no fim. O valor será 11 para uma rota consistindo em um endereço IP, 15 para uma rota consistindo em dois endereços IP e assim por diante, até um máximo de 43. O NOP não faz parte da opção e não é incluído no campo *len*, mas é incluído no tamanho do buffer que especificamos para `setsockopt`. Quando o primeiro endereço da lista é removido da opção de rota de origem e colocado no campo de endereço de destino do cabeçalho de IP, esse valor de *len* é decrementado por quatro (Figuras 9.32 e 9.33 do TCPv2). *ptr* é um ponte-

ro que contém o deslocamento do próximo endereço IP a ser processado na rota e o inicializamos como 4, o que aponta para o primeiro endereço IP. O valor desse campo aumenta por quatro, enquanto o datagrama é processado em cada nó listado.

Desenvolveremos agora três funções para inicializar, criar e processar uma opção de rota de origem. Nossas funções tratam somente de uma opção de rota de origem. Embora seja possível combinar uma rota de origem com outras opções de IP (como alerta de roteador), tal combinação é rara. A Figura 27.2 é a primeira função, `inet_srcrt_init`, junto com algumas variáveis estáticas que são utilizadas enquanto uma opção está sendo construída.

ipopts/sourceroute.c

```
 1 #include   "unp.h"
 2 #include   <netinet/in_systm.h>
 3 #include   <netinet/ip.h>

 4 static u_char *optr;      /* ponteiro para opções sendo formadas */
 5 static u_char *lenptr;    /* ponteiro para byte de comprimento na opção SRR */
 6 static int ocnt;          /* contagem de n° de endereços */

 7 u_char *
 8 inet_srcrt_init(int type)
 9 {
10     optr = Malloc(44);     /* NOP, código, len, ptr, até 10 endereços */
11     bzero(optr, 44);       /* garante EOLs no fim */
12     ocnt = 0;
13     *optr++ = IPOPT_NOP;   /* NOP para alinhamento */
14     *optr++ = type ? IPOPT_SSRR : IPOPT_LSRR;
15     lenptr = optr++;       /* preenchemos o comprimento posteriormente */
16     *optr++ = 4;           /* deslocamento para o primeiro endereço */

17     return (optr - 4);     /* ponteiro para setsockopt() */
18 }
```

ipopts/sourceroute.c

Figura 27.2 Função `inet_srcrt_init`: inicializa antes de armazenar uma rota de origem.

Inicialização

10–17 Alocamos um buffer de tamanho máximo de 44 bytes e o configuramos como 0. O valor da opção EOL é 0; portanto, isso inicializa a opção inteira com EOL bytes. Então, configuramos o cabeçalho de rota de origem. Conforme mostrado na Figura 27.1, primeiro utilizamos um NOP para alinhamento e, então, o tipo de rota de origem (vaga ou estrita), o comprimento e o ponteiro. Salvamos um ponteiro para o campo *len* e armazenaremos esse valor à medida que cada endereço for adicionado à lista. O ponteiro para a opção é retornado para o chamador e será passado como o quarto argumento para `setsockopt`.

A próxima função, `inet_srcrt_add`, adiciona um endereço IPv4 à rota de origem que está sendo construída.

ipopts/sourceroute.c

```
19 int
20 inet_srcrt_add(char *hostptr)
21 {
22     int     len;
23     struct addrinfo *ai;
24     struct sockaddr_in *sin;

25     if (ocnt > 9)
```

Figura 27.3 Função `inet_srcrt_add`: adiciona um endereço IPv4 a uma rota de origem (*continua*).

```
26          err_quit("too many source routes with: %s", hostptr);

27      ai = Host_serv(hostptr, NULL, AF_INET, 0);
28      sin = (struct sockaddr_in *) ai->ai_addr;
29      memcpy(optr, &sin->sin_addr, sizeof(struct in_addr));
30      freeaddrinfo(ai);

31      optr += sizeof(struct in_addr);
32      ocnt++;
33      len = 3 + (ocnt * sizeof(struct in_addr));
34      *lenptr = len;
35      return (len + 1);           /* tamanho para setsockopt() */
36  }
```

ipopts/sourceroute.c

Figura 27.3 Função `inet_srcrt_add`: adiciona um endereço IPv4 a uma rota de origem (*continuação*).

Argumento

19–20 O argumento aponta para um nome de host ou para um endereço IP decimal com pontos.

Verificação do estouro

25–26 Verificamos que muitos endereços não estão especificados e, então, inicializamos se esse for o primeiro endereço.

Obtenção do endereço IP binário e armazenamento na rota

27–35 Nossa função `host_serv` trata de um nome de host ou de uma string decimal com pontos e armazenamos o endereço binário resultante na lista. Atualizamos o campo *len* e retornamos o tamanho total do buffer (incluindo o NOP) que o chamador deve passar para `setsockopt`.

Quando uma rota de origem recebida é retornada para a aplicação por `getsockopt`, o formato é diferente da Figura 27.1. Mostramos o formato recebido na Figura 27.4.

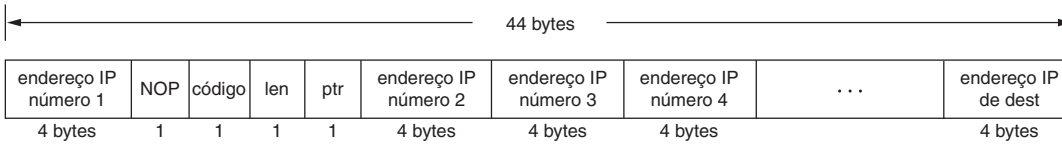


Figura 27.4 Formato da opção de rota de origem retornada por `getsockopt`.

Primeiro, a ordem dos endereços foi invertida pelo kernel, em relação à ordem existente na rota de origem recebida. O que queremos dizer com "invertida" é que, se a rota de origem recebida contém os quatro endereços A, B, C e D, nessa ordem, o inverso dessa rota é D, C, B e, então, A. Os primeiros 4 bytes contêm o primeiro endereço IP da lista, seguido de um NOP de 1 byte (para alinhamento), seguido do cabeçalho de opção de rota de origem de 3 bytes, seguido dos endereços IP restantes. Até nove endereços IP podem vir após o cabeçalho de 3 bytes e o campo *len* no cabeçalho retornado terá um valor máximo de 39. Como o NOP está sempre presente, o comprimento retornado por `getsockopt` sempre será um múltiplo de 4 bytes.

> O formato mostrado na Figura 27.4 é definido em `<netinet/ip_var.h>` com a seguinte estrutura:
>
> ```
> #define MAX_IPOPTLEN    40
>
> struct ipoption {
>   struct in_addr ipopt_dst; /* dst do primeiro hop se rota de origem */
>   char           ipopt_list[MAX_IPOPTLEN]; /* opções adequadas */
> };
> ```
>
> Na Figura 27.5, verificamos que é muito fácil analisarmos sintaticamente os dados nós mesmos, em vez de utilizarmos essa estrutura.

ipopts/sourceroute.c

```
37 void
38 inet_srcrt_print(u_char *ptr, int len)
39 {
40     u_char c;
41     char   str[INET_ADDRSTRLEN];
42     struct in_addr hop1;

43     memcpy(&hop1, ptr, sizeof(struct in_addr));
44     ptr += sizeof(struct in_addr);

45     while ( (c = *ptr++) == IPOPT_NOP) ;   /* pula todos os NOPs seguintes */

46     if (c == IPOPT_LSRR)
47         printf("received LSRR: ");
48     else if (c == IPOPT_SSRR)
49         printf("received SSRR: ");
50     else {
51         printf("received option type %d\n", c);
52         return;
53     }
54     printf("%s ", Inet_ntop(AF_INET, &hop1, str, sizeof(str)));

55     len = *ptr++ - sizeof(struct in_addr);  /* subtrai endereço IP de dest */
56     ptr++;                          /* pula o ponteiro */
57     while (len > 0) {
58         printf("%s ", Inet_ntop(AF_INET, ptr, str, sizeof(str)));
59         ptr += sizeof(struct in_addr);
60         len -= sizeof(struct in_addr);
61     }
62     printf("\n");
63 }
```

ipopts/sourceroute.c

Figura 27.5 Função `inet_srcrt_print`: imprime uma rota de origem recebida.

Esse formato retornado difere daquele que passamos para `setsockopt`. Se quiséssemos converter o formato da Figura 27.4 para o da Figura 27.1, teríamos de trocar os primeiros 4 bytes com os 4 bytes seguintes e somar 4 ao campo de comprimento. Felizmente, não precisamos fazer isso, pois as implementações derivadas do Berkeley utilizam automaticamente o inverso de uma rota de origem recebida para um soquete TCP. Isto é, as informações mostradas na Figura 27.4 são retornadas por `getsockopt` apenas para nossa informação. Não precisamos chamar `setsockopt` para dizer ao kernel que utilize essa rota para datagramas de IP enviados na conexão TCP; o kernel faz isso automaticamente. Veremos um exemplo disso em breve, com nosso servidor de TCP.

A próxima de nossas funções de rota de origem pega uma rota de origem recebida, no formato mostrado na Figura 27.4, e imprime as informações. Mostramos nossa função `inet_srcrt_print` na Figura 27.5.

Salvando o primeiro endereço IP, pulando todos os NOPs

43–45 O primeiro endereço IP no buffer é salvo e todos os NOPs seguintes são pulados.

Verificação de opção de rota de origem

46–62 Imprimimos somente as informações de uma rota de origem e, a partir do cabeçalho de três bytes, verificamos o *código*, buscamos o item *len* e pulamos o *ptr*. Então, imprimimos todos os endereços IP que vêm após o cabeçalho de três bytes, exceto o endereço IP de destino.

Exemplo

Agora, modificaremos nosso cliente de eco TCP, para especificar uma rota de origem, e nosso servidor de eco TCP, para imprimir uma rota de origem recebida. A Figura 27.6 é nosso cliente.

Processamento dos argumentos de linha de comando

12–26 Chamamos nossa função `inet_srcrt_init` para inicializar a rota de origem, com o tipo de rota especificado pela função `-g` (vaga) ou pela opção `-G` (estrita).

27–33 Se o ponteiro `ptr` estiver configurado, uma opção de rota de origem foi especificada e, assim, adicionamos todos os hops intermediários especificados na rota de origem que alocamos acima com nossa função `inet_srcrt_add`. Se `ptr` não estiver configurado, mas existir mais de um argumento restante na linha de comando, o usuário especificou uma rota sem especificar se ela é vaga ou estrita; portanto, saímos com um erro.

ipopts/tcpcli01.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    c, sockfd, len = 0;
 6     u_char *ptr = NULL;
 7     struct addrinfo *ai;
 8     if(argc < 2)
 9        err_quit("usage: tcpcli01 [ -[gG] <hostname> ... ] <hostname>");
10     opterr = 0;                /* não quer getopt() gravando em stderr */
11     while ( (c = getopt(argc, argv, "gG")) != -1) {
12         switch (c) {
13         case 'g':              /* rota de origem vaga */
14             if (ptr)
15                 err_quit("can't use both -g and -G");
16             ptr = inet_srcrt_init(0);
17             break;
18         case 'G':              /* rota de origem estrita */
19             if (ptr)
20                 err_quit("can't use both -g and -G");
21             ptr = inet_srcrt_init(1);
22             break;
23         case '?':
24             err_quit("unrecognized option: %c", c);
25         }
26     }
27     if (ptr)
28         while (optind < argc - 1)
29             len = inet_srcrt_add(argv[optind++]);
30     else if (optind < argc - 1)
31         err_quit("need -g or -G to specify route");
32     if (optind != argc - 1)
33         err_quit("missing <hostname>");
34     ai = Host_serv(argv[optind], SERV_PORT_STR, AF_INET, SOCK_STREAM);
35     sockfd = Socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
36     if (ptr) {
37         len = inet_srcrt_add(argv[optind]); /* dest no fim */
```

Figura 27.6 Cliente de eco TCP que especifica uma rota de origem (*continua*).

```
38          Setsockopt(sockfd, IPPROTO_IP, IP_OPTIONS, ptr, len);
39          free(ptr);
40      }
41      Connect(sockfd, ai->ai_addr, ai->ai_addrlen);
42      str_cli(stdin, sockfd);     /* faz tudo */
43      exit(0);
44  }
```

ipopts/tcpcli01.c

Figura 27.6 Cliente de eco TCP que especifica uma rota de origem (*continuação*).

Tratando do endereço de destino e criação do soquete

34–35 O argumento final da linha de comando é o nome de host ou endereço decimal com pontos do servidor e nossa função `host_serv` o processa. Não podemos chamar nossa função `tcp_connect` porque devemos especificar a rota de origem entre as chamadas a `socket` e `connect`. Esta última inicia o handshake de três vias e queremos que o SYN inicial e todos os pacotes subseqüentes utilizem essa rota de origem.

36–42 Se uma rota de origem estiver especificada, devemos adicionar o endereço IP do servidor no fim da lista de endereços IP (Figura 27.1). `setsockopt` instala a rota de origem para esse soquete. Então, chamamos `connect`, seguida de nossa função `str_cli` (Figura 5.5).

Nosso servidor de TCP é quase idêntico ao código mostrado na Figura 5.12, com as seguintes alterações. Primeiro, alocamos espaço para as opções.

```
int      len;
u_char   *opts;
opts = Malloc(44);
```

Então, buscamos as opções de IP depois da chamada à função `accept`, mas antes da chamada a `fork`.

```
len = 44;
Getsockopt(connfd, IPPROTO_IP, IP_OPTIONS, opts, &len);
if (len > 0) {
    printf("received IP options, len = %d\n", len);
    inet_srcrt_print(opts, len);
}
```

Se o SYN recebido do cliente não contiver quaisquer opções de IP, a variável *len* conterá 0 no retorno de `getsockopt` (trata-se de um argumento de valor-resultado). Conforme mencionado anteriormente, não precisamos fazer nada para que o TCP utilize o inverso da rota de origem recebida: isso é feito automaticamente pelo TCP (página 931 do TCPv2). Tudo que estamos fazendo, chamando `getsockopt`, é obter uma cópia da rota de origem invertida. Se não quisermos que o TCP utilize essa rota, chamamos `setsockopt` depois que `accept` retornar, especificando um quinto argumento (o comprimento) igual a 0, o que remove todas as opções de IP correntemente em utilização. A rota de origem já foi utilizada pelo TCP para o segundo segmento do handshake de três vias (Figura 2.5), mas, se removermos as opções, o IP utilizará a rota que calcular para futuros pacotes para esse cliente.

Agora, mostraremos um exemplo de nosso cliente/servidor, quando especificamos uma rota de origem. Executamos nosso cliente no host `freebsd`, como segue:

```
freebsd4 % tcpcli01 -g macosx freebsd4 macosx
```

Após a configuração apropriada tratar das rotas de origem e encaminhar o IP, isso envia os datagramas de IP de `freebsd4` para o host `macosx`, de volta para `freebsd4` e, por fim, para o host `macosx`, que está executando o servidor. Os dois sistemas, `freebsd4` e `macosx`, devem encaminhar e aceitar datagramas roteados da origem para que esse exemplo funcione.

Quando a conexão é estabelecida no servidor, ele gera a seguinte saída:

```
macosx % tcpserv01
received IP options, len = 16
received LSRR: 172.24.37.94 172.24.37.78 172.24.37.94
```

O primeiro endereço IP impresso é o primeiro hop do caminho inverso (`freebsd4`, como mostrado na Figura 27.4) e os dois endereços seguintes estão na ordem utilizada pelo servidor para enviar datagramas de volta para o cliente. Se observarmos a troca do cliente/servidor utilizando `tcpdump`, poderemos ver a opção de rota de origem em cada datagrama nas duas direções.

> Infelizmente, a operação da opção de soquete `IP_OPTIONS` nunca foi documentada; portanto, podem ser encontradas variações em sistemas que não são derivados do código-fonte Berkeley. Por exemplo, no Solaris 2.5, o primeiro endereço retornado no buffer por `getsockopt` (Figura 27.4) não é o do primeiro hop da rota de retorno, mas o endereço do peer. Contudo, a rota invertida utilizada pelo TCP é correta. Além disso, o Solaris 2.5 precede todas as opções de rota de origem com quatro NOPs, limitando a opção a oito endereços IP, em vez do limite real de nove.

Excluindo a Rota de Origem Recebida

Infelizmente, as rotas de origem apresentam uma brecha de segurança para programas que realizam autenticação utilizando somente endereços IP (agora reconhecidamente inadequados). Se um hacker enviar pacotes com um endereço confiável como a origem, mas com seu próprio endereço na rota de origem, os pacotes de retorno que utilizarem a rota de origem inversa irão para o hacker, sem envolver o sistema listado como a origem original. A partir do lançamento do Net/1 (1989), os servidores `rlogind` e `rshd` tiveram um código semelhante ao seguinte:

```
u_char   buf[44];
char     lbuf[BUFSIZ];
int      optsize;

optsize = sizeof(buf);
if (getsockopt(0, IPPROTO_IP, IP_OPTIONS,
               buf, &optsize) == 0 && optsize != 0) {
   /* formata as opções como números hexadecimais para imprimir em lbuf[] */
   syslog(LOG_NOTICE,
          "Connection received using IP options (ignored):%s", lbuf);
   setsockopt(0, IPPROTO_IP, IP_OPTIONS, NULL, 0);
}
```

Se uma conexão chega com quaisquer opções de IP (o valor de `optsize` retornado por `getsockopt` é não-zero), uma mensagem é registrada utilizando `syslog` e `setsockopt` é chamada para limpar as opções. Isso impede que quaisquer segmentos TCP futuramente enviados nessa conexão utilizem o inverso da rota de origem recebida. Essa técnica é agora conhecida como sendo inadequada, pois, quando a aplicação recebe a conexão, o handshake de três vias do TCP está completo e o segundo segmento (o SYN-ACK do servidor, na Figura 2.5) já seguiu o inverso da rota de origem, de volta para o cliente (ou pelo menos para um dos hops intermediários listados na rota de origem, que é onde o hacker está localizado). Como o hacker viu números de seqüência do TCP nas duas direções, mesmo que mais nenhum pacote seja enviado com a rota de origem, ele ainda poderá enviar pacotes para o servidor com o número de seqüência correto.

A única solução para esse problema em potencial é proibir todas as conexões de TCP que chegam com uma rota de origem, quando você estiver utilizando o endereço IP de origem para alguma forma de validação (como fazem `rlogind` e `rshd`). Substitua a chamada a `setsockopt`, no fragmento de código que acabamos de mostrar, por um fechamento da conexão recém-aceita e uma terminação do servidor recentemente gerado. Dessa maneira, o se-

gundo segmento do handshake de três vias já foi enviado, mas a conexão não deverá ser deixada aberta.

27.4 Cabeçalhos de extensão do IPv6

Não mostramos quaisquer opções com o cabeçalho IPv6 na Figura A.2 (ele tem sempre 40 bytes de comprimento), mas um cabeçalho IPv6 pode ser seguido pelos seguintes *cabeçalhos de extensão* opcionais:

1. Opções de hop por hop devem vir imediatamente após o cabeçalho IPv6 de 40 bytes. Não existem opções hop por hop correntemente definidas que sejam usáveis por uma aplicação.
2. Nenhuma opção de destino que seja usável por uma aplicação está correntemente definida.
3. O cabeçalho de roteamento é uma opção de roteamento de origem, conceitualmente semelhante ao que descrevemos para o IPv4, na Seção 27.3.
4. O cabeçalho de fragmentação é gerado automaticamente por um host que fragmenta um datagrama de IPv6 e, então, é processado pelo destino final, ao montar os fragmentos.
5. O uso do AH (Authentication Header) está documentado no RFC 2402 (Kent e Atkinson, 1998b).
6. O uso do cabeçalho ESP (encapsulating security payload) está documentado na RFC 2406 (Kent e Atkinson, 1998c).

Dissemos que o cabeçalho de fragmentação é tratado inteiramente pelo kernel e os cabeçalhos AH e ESP são tratados automaticamente também pelo kernel, com base no SADB e no SPDB, que são mantidos utilizando soquetes `PF_KEY` (Capítulo 19). Isso deixa as três primeiras opções, que discutiremos nas próximas duas seções. A API para especificar essas opções é definida pela RFC 3542 (Stevens *et al*., 2003).

27.5 Opções hop por hop do IPv6 e opções de destino

As opções de hop por hop e de destino têm um formato semelhante, mostrado na Figura 27.7. O campo de *próximo cabeçalho* de 8 bits identifica o cabeçalho seguinte que vem após esse cabeçalho de extensão. O *comprimento de extensão de cabeçalho* de 8 bits é o comprimento desse cabeçalho de extensão, em unidades de 8 bytes, mas não incluindo os primeiros 8 bytes. Por exemplo, se esse cabeçalho de extensão ocupa 8 bytes, então o comprimento de sua extensão de cabeçalho é 0; se esse cabeçalho de extensão ocupa 16 bytes, então o comprimento de sua extensão de cabeçalho é 1, e assim por diante. Esses dois cabeçalhos são preenchidos de modo a serem um múltiplo de 8 bytes, com a opção `pad1` ou com a opção `padN`, que será descrita em breve.

O cabeçalho de opções de hop por hop e o cabeçalho de opções de destino contêm qualquer número de opções individuais, as quais têm o formato mostrado na Figura 27.8.

Isso é chamado de *codificação TLV*, pois cada opção aparece com seu tipo, comprimento e valor. O campo de *tipo*, de 8 bits, identifica o tipo de opção. Além disso, os dois bits de ordem superior especificam o que um nó IPv6 faz com essa opção, caso não a entenda:

`00` Pula essa opção e continua o processamento do cabeçalho.

`01` Descarta o pacote.

`10` Descarta o pacote e envia um erro tipo 2 de problema de parâmetro ICMP (Figura A.16) para o remetente, independentemente de o destino do pacote ser ou não um endereço de multicast.

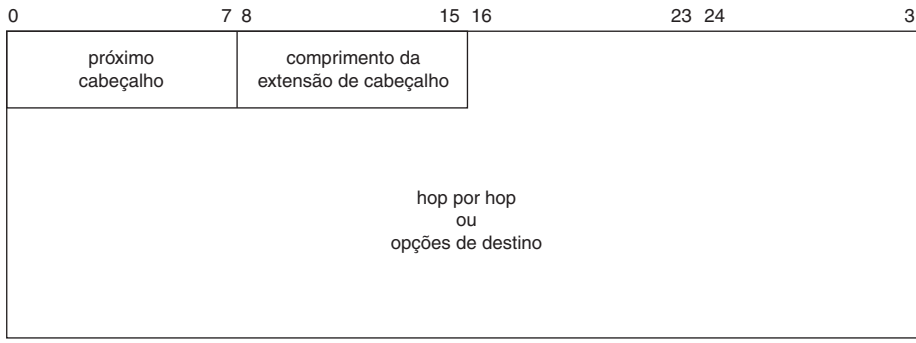


Figura 27.7 Formato das opções de hop por hop e de destino.

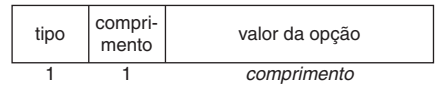


Figura 27.8 Formato das opções de hop por hop individual e de destino.

`11` Descarta o pacote e envia um erro tipo 2 de problema de parâmetro ICMP (Figura A.16) para o remetente. Esse erro é enviado somente se o destino do pacote não é um endereço de multicast.

O próximo bit de ordem superior especifica se os dados da opção mudam na rota:

`0` Os dados da opção não mudam na rota.

`1` Os dados da opção podem mudar na rota.

Os 5 bits de ordem inferior especificam, então, a opção. Observe que todos os 8 bits compõem o código da opção; os 5 bits de ordem inferior sozinhos não identificam a opção. Entretanto, atribuições de valor de opção são feitas para manter os 5 bits de ordem inferior únicos pelo máximo de tempo possível.

O campo de *comprimento,* de 8 bits, especifica o comprimento dos dados da opção em bytes. O campo de tipo e esse campo de comprimento não são incluídos nesse comprimento.

As duas opções de preenchimento estão definidas na RFC 2460 (Deering e Hinden, 1998) e podem ser utilizadas no cabeçalho de opções de hop por hop ou no cabeçalho de opções de destino. O *comprimento de payload jumbo*, uma opção de hop por hop, está definido na RFC 2675 (Borman, Deering e Hinden, 1999) e é gerado quando necessário e processado quando recebido inteiramente pelo kernel. O *alerta de roteador*, uma opção de hop por hop, está descrito para IPv6 na RFC 2711 (Partridge e Jackson, 1999) e é semelhante ao alerta de roteador IPv4. Mostramos isso na Figura 27.9. Outras opções também estão definidas, por exemplo, para Mobile-IPv6, mas não as mostraremos aqui.

O byte `pad1` é a única opção sem comprimento e valor. Ele fornece 1 byte de preenchimento. A opção `pad`*N* é utilizada quando 2 ou mais bytes de preenchimento são exigidos. Para 2 bytes de preenchimento, o comprimento dessa opção seria 0 e a opção consistiria somente no campo de tipo e no campo de comprimento. Para 3 bytes de preenchimento, o comprimento seria 1, e 1 byte de 0 viria após esse comprimento. A opção comprimento de payload jumbo fornece um comprimento de datagrama de 32 bits e é utilizada quando o campo de comprimento de payload de 16 bits da Figura A.2 é inadequado. A opção de alerta de roteador indica que esse pacote deve ser interceptado por certos roteadores ao longo do caminho; o valor na opção de alerta de roteador indica quais roteadores devem estar interessados.

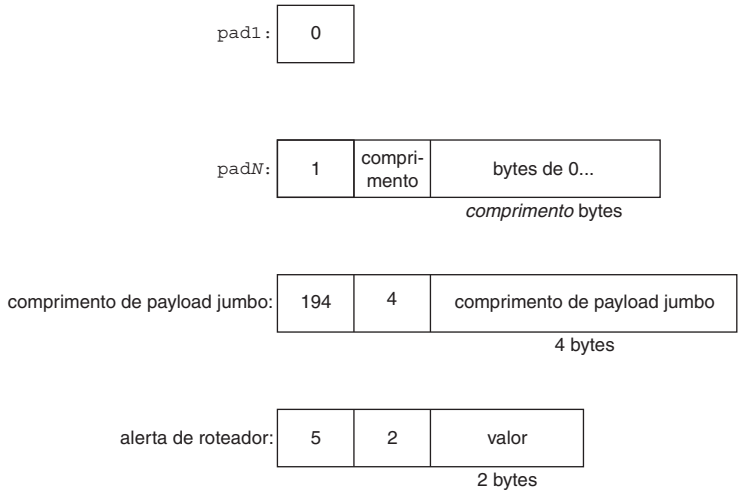


Figura 27.9 Opções de hop por hop do IPv6.

Mostramos as opções de preenchimento porque cada opção de hop por hop e de destino também tem um *requisito de alinhamento* associado, escrito como *x*n + *y*. Isso significa que a opção deve aparecer em um inteiro múltiplo de *x* bytes, a partir do início do cabeçalho, mais *y* bytes. Por exemplo, o requisito de alinhamento da opção de payload jumbo é 4n + 2 e isso é obrigar o valor da opção de 4 bytes (o comprimento de payload jumbo) a estar em um limite de 4 bytes. A razão pela qual o valor *y* é 2 para essa opção deve-se aos 2 bytes que aparecem no começo de cada cabeçalho de opções de hop por hop e de destino (Figura 27.8). O requisito de alinhamento da opção de alerta de roteador é 2n + 0, para obrigar o valor da opção de 2 bytes a estar em um limite de 2 bytes.

As opções de hop por hop e de destino normalmente são especificadas como dados auxiliares com `sendmsg`, e retornadas como dados auxiliares por `recvmsg`. Nada de especial precisa ser feito pela aplicação para enviar uma ou ambas dessas opções; basta especificá-las em uma chamada para `sendmsg`. Para receber essas opções, a opção de soquete correspondente deve estar ativada: `IPV6_RECVHOPOPTS` para as opções de hop por hop e `IPV6_RECVDSTOPTS` para as opções de destino. Por exemplo, para ativar as duas opções para serem retornadas,

```
const int on = 1;

setsockopt(sockfd, IPPROTO_IPV6, IPV6_RECVHOPOPTS, &on, sizeof(on));
setsockopt(sockfd, IPPROTO_IPV6, IPV6_RECVDSTOPTS, &on, sizeof(on));
```

A Figura 27.10 mostra o formato dos objetos de dados auxiliares utilizados para enviar e receber as opções de hop por hop e de destino.

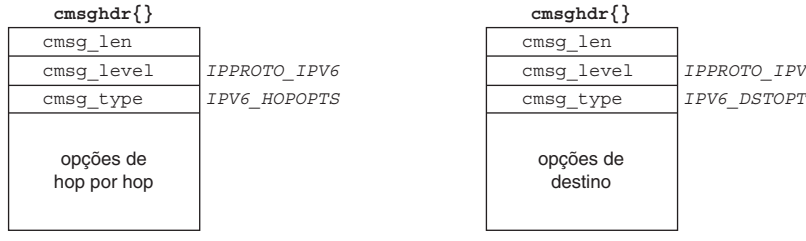


Figura 27.10 Objetos de dados auxiliares para opções de hop por hop e de destino.

O conteúdo real do cabeçalho de opção IPv6 é passado entre o usuário e o kernel como a parte `cmsg_data` desses objetos. Para reduzir a duplicação de código, sete funções são definidas para criar e processar essas seções de dados. As quatro funções a seguir constroem uma opção para enviar:

```
#include <netinet/in.h>

int inet6_opt_init(void *extbuf, socklen_t extlen);
```

Retorna: o número de bytes exigidos para conter o cabeçalho de extensão vazio, –1 em caso de erro

```
int inet6_opt_append(void *extbuf, socklen_t extlen,
                     int offset, uint8_t type, socklen_t len,
                     uint_t align, void **databufp);
```

Retorna: comprimento atualizado do cabeçalho de extensão total depois de adicionar a opção, –1 em caso de erro

```
int inet6_opt_finish(void *extbuf, socklen_t extlen,
                     int offset);
```

Retorna: comprimento atualizado do cabeçalho terminado de extensão, –1 em caso de erro

```
int inet6_opt_set_val(void *databuf, int offset,
                      const void *val, socklen_t vallen);
```

Retorna: novo deslocamento dentro de *databuf*

`inet6_opt_init` retorna o número de bytes exigido para conter um cabeçalho de extensão vazio. Se o argumento *extbuf* não é `NULL`, ele inicializa o cabeçalho de extensão. Ele falha e retorna –1 se o argumento *extbuf* é fornecido, mas o argumento *extlen* não é múltiplo de 8. (Todos os cabeçalhos de opções de hop por hop e de destino IPv6 devem ser múltiplos de 8 bytes.)

`inet6_opt_append` retorna o comprimento total atualizado do cabeçalho de extensão, depois de acrescentar a opção especificada. Se o argumento *extbuf* não é `NULL`, ele também inicializa a opção e insere todo o preenchimento necessário. Ele falha e retorna –1 se a nova opção não se ajusta no buffer fornecido. O argumento *offset* é o comprimento total geral corrente e deve ser o valor de retorno de uma chamada anterior a `inet6_opt_init` ou `inet6_opt_append`. Os argumentos *type* e *len* são o tipo e o comprimento da opção e são copiados diretamente para o cabeçalho de opção. O argumento *align* especifica o requisito de alinhamento; isto é, o *x* da função *x*n + *y*. O valor de *y* é derivado de *align* e *len*; portanto, ele não precisa ser especificado explicitamente. O argumento *databufp* é o endereço para um ponteiro que será preenchido com a localização do valor da opção; o chamador pode então copiar o valor da opção para essa localização, utilizando a função `inet6_opt_set_val` ou qualquer outro método.

`inet6_opt_finish` é chamado para completar um cabeçalho de extensão, adicionando todo o preenchimento necessário para tornar o cabeçalho global um múltiplo de 8 bytes. Como antes, se o argumento *extbuf* não é `NULL`, o preenchimento é inserido no buffer; caso contrário, a função simplesmente calcula o comprimento atualizado. Assim como acontece com `inet6_opt_append`, o argumento *deslocamento* é o comprimento total geral corrente, o valor de retorno de uma função `inet6_opt_init` ou `inet6_opt_append` anterior. `inet6_opt_finish` retorna o comprimento total do cabeçalho completado ou –1, caso o preenchimento exigido não se ajuste no buffer fornecido.

`inet6_opt_set_val` copia um valor de opção para o buffer de dados retornado por `inet6_opt_append`. O argumento *databuf* é o ponteiro retornado de `inet6_opt_append`. *offset* é um comprimento em execução dentro dessa opção; deve ser inicializado como 0 para cada opção e, então, será o valor de retorno da função `inet6_opt_set_val` anterior, quando a opção é construída. Os argumentos *val* e *vallen* especificam o valor a copiar para o buffer de valor da opção.

A utilização esperada dessas funções é fazer duas passagens pela lista de opções que você pretende inserir: a primeira para calcular o comprimento desejado e a segunda para construir realmente a opção em um buffer apropriadamente dimensionado. Durante a primeira passagem, chamamos `inet6_opt_init` e `inet6_opt_append` uma vez para cada opção que acrescentaremos e `inet6_opt_finish`, passando NULL e 0 para os argumentos *extbuf* e *extlen*, respectivamente. Então, alocamos dinamicamente o buffer de opção, utilizando o tamanho retornado por `inet6_opt_finish`, e passaremos esse buffer como o argumento *extbuf* durante a segunda passagem. Durante a segunda passagem, chamamos `inet6_opt_init` e `inet6_opt_append`, ou copiando os dados manualmente ou utilizando `inet6_opt_set_val` para cada valor de opção. Por fim, chamamos `inet6_opt_finish`. Como alternativa, podemos alocar previamente um buffer, que deve ser grande o bastante para nossas opções desejadas, e pular a primeira passagem; entretanto, isso é vulnerável a falhas, caso uma alteração nas opções desejadas estoure o buffer previamente alocado.

As três funções restantes processam uma opção recebida:

```
#include <netinet/in.h>

int inet6_opt_next(const void *extbuf, socklen_t extlen, int offset,
                   uint8_t *typep, socklen_t *lenp, void **databufp);

                Retorna: o deslocamento da próxima opção, –1 no fim das opções ou em caso de erro

int inet6_opt_find(const void *extbuf, socklen_t extlen, int offset,
                   uint8_t type, socklen_t *lenp, void **databufp);

                Retorna: o deslocamento da próxima opção, –1 no fim das opções ou em caso de erro

int inet6_opt_get_val(const void *databuf, int offset,
                      void *val, socklen_t vallen);

                                        Retorna: novo deslocamento dentro de databuf
```

`inet6_opt_next` processa a próxima opção em um buffer. *extbuf* e *extlen* especificam o buffer que contém o cabeçalho. Assim como acontece com `inet6_opt_append`, o *offset* é um deslocamento global no buffer. Ele é 0 para a primeira chamada de `inet6_opt_next` e, então, é o valor de retorno da chamada anterior para chamadas futuras. *typep*, *lenp* e *databufp* retornam o tipo, o comprimento e o valor da opção, respectivamente. `inet6_opt_next` retorna –1, caso o cabeçalho seja malformado ou se tiver alcançado o fim do buffer.

`inet6_opt_find` é semelhante à função anterior, mas permite que o chamador especifique o tipo da opção a procurar (o argumento *tipo*), em vez de sempre retornar a próxima opção.

`inet6_opt_get_val` é utilizada para extrair valores de uma opção, utilizando o ponteiro *databuf* retornado por uma chamada de `inet6_opt_next` ou `inet6_opt_find` anterior. Assim como acontece com `inet6_opt_set_val`, o argumento *deslocamento* deve iniciar em 0 para cada opção e, então, ser o valor de retorno de uma chamada anterior a `inet6_opt_get_val`.

27.6 Cabeçalho de roteamento IPv6

O cabeçalho de roteamento IPv6 é utilizado para roteamento de origem no IPv6. Os dois primeiros bytes do cabeçalho de roteamento são os mesmos que mostramos na Figura 27.7: um campo de *próximo* cabeçalho, seguido de um *comprimento de extensão de cabeçalho*. Os próximos dois bytes especificam o *tipo de roteamento* e o número de *segmentos restantes* (isto é, quantos nós listados ainda precisam ser visitados). Somente um tipo de cabeçalho de roteamento é especificado, o tipo 0, e mostramos seu formato na Figura 27.11.

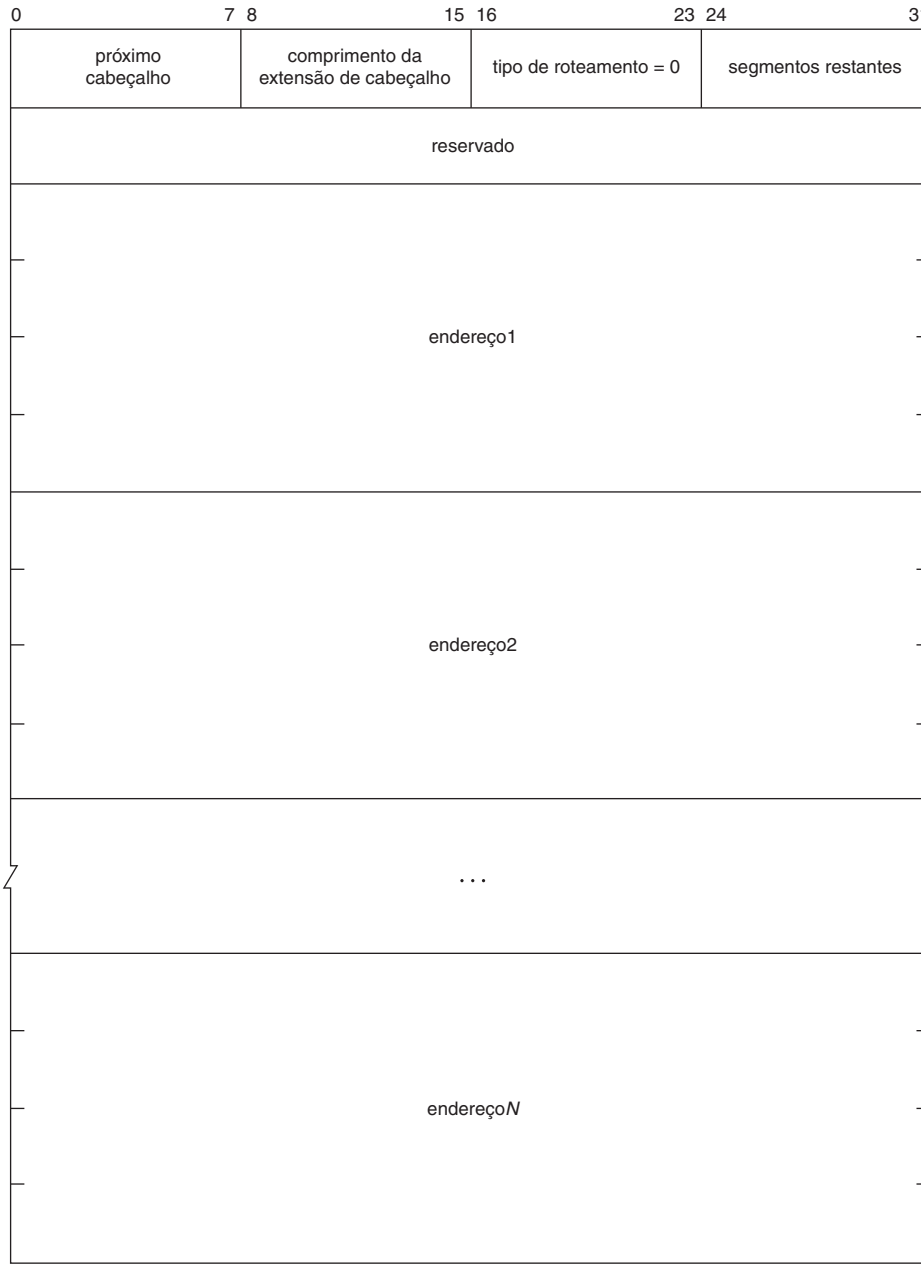


Figura 27.11 Cabeçalho de roteamento IPv6.

Um número ilimitado de endereços pode aparecer no cabeçalho de roteamento (limitado somente pelo comprimento do pacote) e os *segmentos restantes* devem ser iguais ou menores que o número de endereços no cabeçalho. A RFC 2460 (Deering e Hinden, 1998) especifica os detalhes de como o cabeçalho é processado quando o pacote viaja para o destino final, junto com um exemplo detalhado.

O cabeçalho de roteamento é normalmente especificado como dados auxiliares com `sendmsg` e retornado também como dados auxiliares por `recvmsg`. Nada de especial precisa ser feito pela aplicação para enviar o cabeçalho: basta especificá-lo como dados auxiliares em uma chamada a `sendmsg`. Para receber o cabeçalho de roteamento, a opção de soquete `IPV6_RECVRTHDR` deve estar ativada, como em

```
const int on = 1;

setsockopt(sockfd, IPPROTO_IPV6, IPV6_RECVRTHDR, &on, sizeof(on));
```

A Figura 27.12 mostra o formato do objeto de dados auxiliar utilizado para enviar e receber o cabeçalho de roteamento. Seis funções são definidas para criar e processar o cabeçalho de roteamento. As três funções a seguir constroem uma opção para envio:

```
#include <netinet/in.h>

socklen_t inet6_rth_space(int type, int segments);

                                Retorna: número positivo de bytes, se estiver OK, 0 em caso de erro

void *inet6_rth_init(void *rthbuf, socklen_t rthlen,
                     int type, int segments);

                                Retorna: ponteiro não-nulo se OK, NULL em erro

int inet6_rth_add(void *rthbuf, const struct in6_addr *addr);

                                Retorna: 0 se OK, –1 em erro
```

`inet6_rth_space` retorna o número de bytes exigidos para conter um cabeçalho de roteamento do *type* especificado (normalmente especificado como `IPV6_RTHDR_TYPE_0`) com o número especificado de *segments*.

`inet6_rth_init` inicializa o buffer apontado por *rthbuf*, para conter um cabeçalho de roteamento do tipo (*type*) especificado e o número especificado de segmentos (*segments*). O valor de retorno é o ponteiro para o buffer, o qual é então utilizado como um argumento para a próxima função. `inet6_rth_init` retorna `NULL` caso um erro ocorra, por exemplo, quando o buffer fornecido não for suficientemente grande.

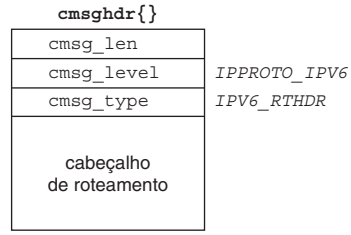


Figura 27.12 Objeto de dados auxiliar para cabeçalho de roteamento IPv6.

`inet6_rth_add` adiciona o endereço de IPv6 apontado por `addr` no fim do cabeçalho de roteamento que está sendo construído. Quando bem-sucedido, o membro `segleft` do cabeçalho de roteamento é atualizado para levar em conta o novo endereço.

As três funções a seguir lidam com um cabeçalho de roteamento recebido:

```
#include <netinet/in.h>

int inet6_rth_reverse(const void *in, void *out);
                                                    Retorna: 0 se OK, –1 em erro

int inet6_rth_segments(const void *rthbuf);
          Retorna: o número de segmentos no cabeçalho de roteamento se OK, –1 em caso de erro

struct in6_addr *inet6_rth_getaddr(const void *rthbuf, int index);
                                          Retorna: ponteiro não-nulo se OK, NULL em erro
```

`inet6_rth_reverse` pega um cabeçalho de roteamento que foi recebido (apontado por *in*) e cria um novo cabeçalho de roteamento (no buffer apontado por *out*), que envia datagramas ao longo do inverso desse caminho. O inverso pode ocorrer no local; isto é, os ponteiros *in* e *out* podem apontar para o mesmo buffer.

`inet6_rth_segments` retorna o número de segmentos no cabeçalho de roteamento descrito por *rthbuf*. Quando bem-sucedido, o valor de retorno é maior que zero.

`inet6_rth_getaddr` retorna um ponteiro para o endereço IPv6 especificado pelo índice (*index*) no cabeçalho de roteamento descrito por *rthbuf*. O *index* deve ter um valor entre zero e um a menos que o valor retornado por `inet6_rth_segments`, inclusive.

Para demonstrar essas opções, criamos um cliente e um servidor de UDP. O cliente, mostrado na Figura 27.13, aceita uma rota de origem na linha de comando, como o cliente TCP de IPv4 que mostramos na Figura 27.6; o servidor imprime a rota de origem recebida e a inverte para enviar de volta ao cliente.

Criação da rota de origem

11–21 Se mais de um argumento foi fornecido, todos os argumentos, menos o último, formam a rota de origem. Primeiro, determinamos o espaço que o cabeçalho de rota exigirá, com `inet6_rth_space`; então, alocamos o espaço necessário com `malloc`. Inicializamos o buffer alocado com `inet6_rth_init`. Então, para cada endereço na rota de origem, o convertemos para a forma numérica utilizando `host_serv` e o adicionamos na rota de origem utilizando `inet6_rth_add`. Isso é muito parecido com nosso cliente TCP de IPv4, exceto que, em vez de nossas próprias funções auxiliares, essas funções de biblioteca são fornecidas pelo sistema.

Pesquisa do destino e criação do soquete

22–23 Utilizamos `host_serv` para pesquisar o destino e criamos um soquete para utilizar.

Configuração da opção persistente `IPV6_RTHDR` e chamada à função de trabalho

24–27 Conforme veremos na Seção 27.7, em vez de enviar os mesmos dados auxiliares com cada pacote, podemos utilizar `setsockopt` para aplicar o cabeçalho de roteamento em cada pacote da sessão. Somente configuramos essa opção se `ptr` for não-nulo, significando que alocamos um cabeçalho de rota anteriormente. Por fim, chamamos a função de trabalho, `dg_cli`, que definimos na Figura 8.8.

ipopts/udpcli01.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    c, sockfd, len = 0;
 6     u_char *ptr = NULL;
 7     void   *rth;
 8     struct addrinfo *ai;
 9     if(argc < 2)
10         err_quit("usage: udpcli01 [ <hostname> ... ] <hostname>");
11     if (argc > 2) {
12         int i;
13         len = Inet6_rth_space(IPV6_RTHDR_TYPE_0, argc - 2);
14         ptr = Malloc(len);
15         Inet6_rth_init(ptr, len, IPV6_RTHDR_TYPE_0, argc - 2);
16         for (i = 1; i < argc - 1; i++) {
17             ai = Host_serv(argv[i], NULL, AF_INET6, 0);
18             Inet6_rth_add(ptr,
19                           &((struct sockaddr_in6 *) ai->ai_addr)->sin6_addr);
20         }
21     }
22     ai = Host_serv(argv[argc - 1], SERV_PORT_STR, AF_INET6, SOCK_DGRAM);
23     sockfd = Socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
24     if (ptr) {
25         Setsockopt(sockfd, IPPROTO_IPV6, IPV6_RTHDR, ptr, len);
26         free(ptr);
27     }
28     dg_cli(stdin, sockfd, ai->ai_addr, ai->ai_addrlen); /* faz tudo */
29     exit(0);
30 }
```

ipopts/udpcli01.c

Figura 27.13 O cliente UDP de IPv6 com rota de origem.

Nosso servidor é o mesmo servidor de UDP simples de antes: abre um soquete e chama `dg_echo`. A configuração é simples; portanto, não a mostraremos. Em vez disso, a Figura 27.14 mostra nossa função `dg_echo`, que imprime a rota de origem, se uma foi recebida, e a inverte para utilização no retorno do pacote.

ipopts/dgechoprintroute.c

```
 1 #include   "unp.h"
 2 void
 3 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 4 {
 5     int    n;
 6     char   mesg[MAXLINE];
 7     int    on;
 8     char   control[MAXLINE];
 9     struct msghdr msg;
10     struct cmsghdr *cmsg;
11     struct iovec iov[1];
12     on = 1;
13     Setsockopt(sockfd, IPPROTO_IPV6, IPV6_RECVRTHDR, &on, sizeof(on));
```

Figura 27.14 Função `dg_echo` que imprime e inverte rota de origem IPv6 (*continua*).

```
14      bzero(&msg, sizeof(msg));
15      iov[0].iov_base = mesg;
16      msg.msg_name = pcliaddr;
17      msg.msg_iov = iov;
18      msg.msg_iovlen = 1;
19      msg.msg_control = control;
20      for ( ; ; ) {
21          msg.msg_namelen = clilen;
22          msg.msg_controllen = sizeof(control);
23          iov[0].iov_len = MAXLINE;
24          n = Recvmsg(sockfd, &msg, 0);

25          for (cmsg = CMSG_FIRSTHDR(&msg); cmsg != NULL;
26               cmsg = CMSG_NXTHDR(&msg, cmsg)) {
27              if (cmsg->cmsg_level == IPPROTO_IPV6 &&
28                  cmsg->cmsg_type == IPV6_RTHDR) {
29                  inet6_srcrt_print(CMSG_DATA(cmsg));
30                  Inet6_rth_reverse(CMSG_DATA(cmsg), CMSG_DATA(cmsg));
31              }
32          }

33          iov[0].iov_len = n;
34          Sendmsg(sockfd, &msg, 0);
35      }
36  }
```

ipopts/dgechoprintroute.c

Figura 27.14 Função `dg_echo` que imprime e inverte rota de origem IPv6 (*continuação*).

Ativação da opção `IPV6_RECVRTHDR` e configuração da estrutura `msghdr`

12–13 Para obter a rota de origem recebida, devemos configurar a opção de soquete `IPV6_RECVRTHDR`. Também devemos utilizar `recvmsg`; portanto, configuramos os campos imutáveis de uma estrutura `msghdr`.

Configuração de campos modificáveis e chamada a `recvmsg`

21–24 Configuramos os campos de comprimento com os tamanhos apropriados e chamamos `recvmsg`.

Localização e processamento do cabeçalho de rota

25–32 Fazemos um loop pelos dados auxiliares, utilizando `CMSG_FIRSTHDR` e `CMSG_NXTHDR`. Mesmo que estejamos esperando somente uma porção de dados auxiliares, é considerada uma boa prática fazer um loop como esse. Se encontrarmos um cabeçalho de roteamento, o imprimimos com nossa função `inet6_srcrt_print` (Figura 27.15). Então, invertemos a rota com `inet6_rth_reverse` para que possamos utilizá-la para retornar o pacote pelo mesmo caminho. Nesse caso, `inet6_rth_reverse` funciona na rota local, de modo que podemos utilizar os mesmos dados auxiliares para enviar o pacote de retorno.

ipopts/sourceroute6.c

```
1 #include   "unp.h"

2 void
3 inet6_srcrt_print(void *ptr)
4 {
5     int    i, segments;
6     char   str[INET6_ADDRSTRLEN];
```

Figura 27.15 Função `inet6_srcrt_print`: imprime uma rota de origem de IPv6 recebida (*continua*).

```
 7     segments = Inet6_rth_segments(ptr);
 8     printf("received source route: ");
 9     for (i = 0; i < segments; i++)
10         printf("%s ", Inet_ntop(AF_INET6, Inet6_rth_getaddr(ptr, i),
11                                 str,sizeof(str)));
12     printf("\n");
13 }
```

ipopts/sourceroute6.c

Figura 27.15 Função `inet6_srcrt_print`: imprime uma rota de origem de IPv6 recebida (*continuação*).

Eco do pacote

33–34 Configuramos o comprimento dos dados a enviar e utilizamos `sendmsg` para retornar o pacote.

Nossa função `inet6_srcrt_print` é quase trivial, graças às funções auxiliares de rota IPv6.

Determinação do número de segmentos na rota

7 Primeiro, utilizamos `inet6_rth_segments` para determinar o número de segmentos presentes na rota.

Fazendo um loop em cada segmento

9–11 Fazemos loop por todos os segmentos, chamando `inet6_rth_getaddr` para cada um e convertendo o endereço para a forma de apresentação, utilizando `inet_ntop`.

Nossos cliente e servidor que tratam de rotas de origem de IPv6 não precisam saber como a rota de origem é formatada no pacote. As funções de biblioteca que a API fornece ocultam os detalhes do formato do pacote, apesar de nos fornecerem toda a flexibilidade que tínhamos quando construímos a opção a partir do zero no IPv4.

27.7 Opções IPv6 persistentes

Descrevemos o uso de dados auxiliares com `sendmsg` e `recvmsg` para enviar e receber sete objetos de dados auxiliares diferentes:

1. Informações de pacote IPv6: a estrutura `in6_pktinfo` contendo o endereço de destino e o índice de interface enviado ou o endereço de origem e o índice de interface recebido (Figura 22.21)
2. O limite de hop enviado ou recebido (Figura 22.21)
3. O endereço do próximo hop (Figura 22.21)
4. A classe de tráfego enviada ou recebida (Figura 22.21)
5. Opções de hop por hop (Figura 27.10)
6. Opções de destino (Figura 27.10)
7. Cabeçalho de roteamento (Figura 27.12)

Resumimos os valores de `cmsg_level` e `cmsg_type` para esses objetos, junto com os valores para o outro objeto de dados auxiliares, na Figura 14.11.

Quando o mesmo valor vai ser utilizado para todos os pacotes enviados em um soquete, em vez de enviar essas opções em cada chamada a `sendmsg`, podemos configurar as opções de soquete correspondentes. Estas utilizam as mesmas constantes dos dados auxiliares; isto é,

o nível de opção é sempre IPPROTO_IPV6 e o nome da opção é IPV6_DSTOPTS, IPV6_HOPLIMIT, IPV6_HOPOPTS, IPV6_NEXTHOP, IPV6_PKTINFO, IPV6_RTHDR ou IPV6_TCLASS. Mas essas opções persistentes podem ser sobrescritas de acordo com o pacote, para um soquete UDP ou para um soquete IPv6 bruto, especificando-se dados auxiliares em uma chamada a sendmsg. Se quaisquer dados auxiliares forem especificados em uma chamada a sendmsg, as opções persistentes correspondentes não serão enviadas com esse datagrama.

O conceito de opções persistentes também pode ser utilizado com TCP, pois os dados auxiliares nunca são enviados nem recebidos por sendmsg ou recvmsg em um soquete TCP. Em vez disso, uma aplicação de TCP pode configurar a opção de soquete correspondente e especificar qualquer um dos sete tipos de opção mencionados no começo desta seção. Esses objetos afetam então todos os pacotes enviados nesse soquete. Entretanto, a retransmissão de pacotes que foram originalmente enviados quando outras (ou nenhuma) opções persistentes foram configuradas pode utilizar as opções originais ou as novas opções persistentes.

Não há nenhuma maneira de recuperar as opções recebidas via TCP, pois não há nenhum relacionamento entre os pacotes recebidos e as operações de recepção do usuário.

27.8 API histórica avançada para IPv6

A RFC 2292 (Stevens e Thomas, 1998) define uma versão anterior da API descrita aqui, que está implementada e é distribuída em alguns sistemas. Nessa versão anterior, as funções para lidar com opções de destino e de hop por hop são: inet6_option_space, inet6_option_init, inet6_option_append, inet6_option_alloc, inet6_option_next e inet6_option_find. Essas funções lidavam diretamente com objetos struct cmsghdr, supondo que todas as opções estavam contidas nos dados auxiliares. As funções de cabeçalho de roteamento dessa API eram: inet6_rthdr_space, inet6_rthdr_init, inet6_rthdr_add, inet6_rthdr_lasthop, inet6_rthdr_reverse, inet6_rthdr_segments, inet6_rthdr_getaddr e inet6_ rthdr_getflags. Essas funções também operam diretamente sobre objetos de dados auxiliares struct cmsghdr.

Nessa API, as opções persistentes eram configuradas com a opção de soquete IPV6_PKTOPTIONS. Os objetos de dados auxiliares que seriam passados para sendmsg eram, em vez disso, configuradas como a parte dos dados da opção de soquete IPV6_PKTOPTIONS. Nessa API, as opções de soquete IPV6_DSTOPTS, IPV6_HOPOPTS e IPV6_RTHDR eram valores de flag para solicitar a recepção dos respectivos cabeçalhos por meio de dados auxiliares.

Para obter informações adicionais sobre essas operações, consulte as Seções 4 a 8 da RFC 2292 (Stevens e Thomas, 1998).

27.9 Resumo

A mais comumente utilizada das 10 opções de IPv4 definidas é a rota de origem, mas sua utilização está diminuindo atualmente, devido às preocupações com a segurança. O acesso às opções de cabeçalho IPv4 se dá por intermédio da opção de soquete IP_OPTIONS.

O IPv6 define seis cabeçalhos de extensão. O acesso aos cabeçalhos de extensão do IPv6 se dá por meio de uma interface funcional, tornando óbvia a necessidade de entender seu formato real no pacote. Esses cabeçalhos de extensão são gravados como dados auxiliares com sendmsg e retornados também como dados auxiliares com recvmsg.

Exercícios

27.1 Em nosso exemplo de rota de origem IPv4, no final da Seção 27.3, o que mudará se especificarmos cada nó intermediário no cliente com a opção `-G`, em vez da opção `-g`?

27.2 O comprimento do buffer especificado para `setsockopt` para a opção de soquete `IP_OPTIONS` deve ser um múltiplo de 4 bytes. O que faríamos se não colocássemos um NOP no começo do buffer, como mostrado na Figura 27.1?

27.3 Como `ping` recebe uma rota de origem, quando a opção de rota de registro de IP é utilizada (descrito na Seção 7.3 do TCPv1)?

27.4 No exemplo de código do servidor `rlogind`, no final da Seção 27.3, que limpa uma rota de origem recebida, por que o argumento descritor de soquete de `getsockopt` e `setsockopt` é 0?

27.5 Durante anos muitos, o código que mostramos no final da Seção 27.3, que limpa uma rota de origem recebida, era parecido com o seguinte:

```
optsize = 0;
setsockopt(0, IPPROTO_IP, IP_OPTIONS, NULL, &optsize);
```

O que há de errado com esse código? Isso importa?

Capítulo **28**

Soquetes Brutos

28.1 Visão geral

Os soquetes brutos oferecem três recursos não fornecidos pelos soquetes TCP e UDP normais:

- Permitem ler e gravar pacotes ICMPv4, IGMPv4 e ICMPv6. O programa `ping`, por exemplo, envia solicitações de eco ICMP e recebe respostas de eco ICMP. (Desenvolveremos nossa própria versão do programa `ping` na Seção 28.5.) O daemon de roteamento de multicast, `mrouted`, envia e recebe pacotes IGMPv4.

  Essa capacidade também possibilita que aplicações construídas utilizando ICMP ou IGMP sejam tratadas inteiramente como processos de usuário, em vez de colocar mais código no kernel. O daemon de descoberta de roteador (`in.rdisc` no Solaris 2.x; o Apêndice F do TCPv1 descreve como obter o código-fonte para uma versão publicamente disponível), por exemplo, é construído dessa maneira. Ele processa duas mensagens ICMP (anúncio e solicitação de roteador) sobre as quais o kernel nada sabe.
- Com um soquete bruto, um processo pode ler e gravar datagramas IPv4 com um campo de protocolo IPv4 que não é processado pelo kernel. Lembre-se do campo de protocolo IPv4 de 8 bits da Figura A.1. A maioria dos kernels processa somente datagramas contendo os valores 1 (ICMP), 2 (IGMP), 6 (TCP) e 17 (UDP). Mas muitos outros valores são definidos para o campo de protocolo: o registro "Números de Protocolo" do IANA lista todos os valores. Por exemplo, o protocolo de roteamento OSPF não TCP nem UDP, mas IP diretamente, configurando o campo de protocolo do datagrama IP como 89. O programa `gated`, que implementa OSPF, deve utilizar um soquete bruto para ler e gravar esses datagramas IP, pois eles contêm um campo de protocolo sobre o qual o kernel nada sabe. Essa capacidade também persiste no IPv6.
- Com um soquete bruto, um processo pode construir seu próprio cabeçalho IPv4 utilizando a opção de soquete `IP_HDRINCL`. Isso pode ser utilizado, por exemplo, para construir pacotes UDP e TCP, e mostraremos um exemplo na Seção 29.7.

Este capítulo descreve a criação e a entrada e saída de soquete bruto. Também desenvolveremos versões dos programas `ping` e `traceroute` que funcionam tanto com o IPv4 como com o IPv6.

28.2 Criação de soquetes brutos

Os passos envolvidos na criação de um soquete bruto são os seguintes:

1. A função `socket` cria um soquete bruto quando o segundo argumento é `SOCK_RAW`. O terceiro argumento (o protocolo) normalmente é não-zero. Por exemplo, para criar um soquete bruto IPv4, escreveríamos

```
int  sockfd;

sockfd = socket(AF_INET, SOCK_RAW, protocol);
```

   em que *protocol* é uma das constantes `IPPROTO_xxx`, definidas pela inclusão do cabeçalho `<netinet/in.h>`, como `IPPROTO_ICMP`.

   Somente o superusuário pode criar um soquete bruto. Isso impede que os usuários normais escrevam seus próprios datagramas IP para a rede.

2. A opção de soquete `IP_HDRINCL` pode ser configurada como segue:

```
const int on = 1;

if (setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0)
error
```

   Descreveremos o efeito dessa opção de soquete na próxima seção.

3. A função `bind` pode ser chamada no soquete bruto, mas isso é raro. Essa função configura somente o endereço local: não há nenhum conceito de número de porta com um soquete bruto. Com relação à saída, chamar `bind` configura o endereço IP de origem que será utilizado para datagramas enviados no soquete bruto (mas somente se a opção de soquete `IP_HDRINCL` não estiver configurada). Se a função `bind` não é chamada, o kernel configura o endereço IP de origem com o endereço IP primário da interface enviada.

4. A função `connect` pode ser chamada no soquete bruto, mas isso é raro. Essa função configura somente o endereço estrangeiro: novamente, não há nenhum conceito de número de porta com um soquete bruto. Com relação à saída, chamar `connect` nos permite chamar `write` ou `send`, em vez de `sendto`, pois o endereço IP de destino já está especificado.

28.3 Saída de soquete bruto

A saída em um soquete bruto é governada pelas seguintes regras:

- A saída normal é realizada chamando-se `sendto` ou `sendmsg` e especificando-se o endereço IP de destino. As funções `write`, `writev` ou `send` também podem ser chamadas, se o soquete tiver sido conectado.
- Se a opção `IP_HDRINCL` não estiver configurada, o endereço inicial dos dados para o kernel enviar especifica o primeiro byte após o cabeçalho IP, pois o kernel construirá o cabeçalho IP e o colocará como prefixo dos dados do processo. O kernel configura o campo de protocolo do cabeçalho IPv4 que constrói com o terceiro argumento da chamada de `socket`.
- Se a opção `IP_HDRINCL` estiver configurada, o endereço inicial dos dados para o kernel enviar especifica o primeiro byte do cabeçalho IP. O volume de dados a gravar deve incluir o tamanho do cabeçalho IP do chamador. O processo constrói o cabeçalho IP inteiro, exceto: (i) o campo de identificação IPv4 pode ser configurado como 0, o que diz ao kernel para que configure esse valor; (ii) o kernel sempre calcula e armazena a soma de

verificação (*checksum*) do cabeçalho IPv4; e (iii) opções de IP podem ou não ser incluídas; veja a Seção 27.2.

- O kernel fragmenta os pacotes brutos que excedem o MTU da interface enviada.

> Os soquetes brutos são documentados para fornecer uma interface idêntica à que um protocolo teria se fosse residente no kernel (McKusick *et al*., 1996). Infelizmente, isso significa que certos trechos da API são dependentes do kernel do SO, especificamente com relação à ordenação de byte dos campos no cabeçalho IP. Em muitos kernels derivados do Berkeley, todos os campos estão em ordem de byte de rede, exceto `ip_len` e `ip_off`, que estão em ordem de byte do host (páginas 233 e 1057 do TCPv2). No Linux e no OpenBSD, entretanto, todos os campos devem estar na ordem de byte de rede.
>
> A opção de soquete `IP_HDRINCL` foi introduzida com o 4.3BSD Reno. Antes disso, a única maneira para uma aplicação especificar seu próprio cabeçalho IP em pacotes enviados em um soquete de IP bruto era aplicar um patch de kernel que foi introduzido em 1988, por Van Jacobson, para suportar `traceroute`. Esse patch exigia que a aplicação criasse um soquete de IP bruto especificando um *protocolo* de `IPPROTO_RAW`, que tem um valor igual a 255 (que é um valor reservado e nunca deve aparecer como campo de protocolo em um cabeçalho IP).
>
> As funções que realizam entrada e saída em soquetes brutos são algumas das mais simples do kernel. Por exemplo, no TCPv2, cada função exige cerca de 40 linhas de código C (páginas 1054 a 1057), comparadas com a entrada de TCP, com cerca de 2.000 linhas, e com a saída de TCP, com cerca de 700 linhas.

Nossa descrição da opção de soquete `IP_HDRINCL` é para o 4.4BSD. As versões anteriores, como Net/2, preenchiam mais campos no cabeçalho IP, quando essa opção era configurada.

Com o IPv4, é responsabilidade do processo de usuário calcular e configurar todas as somas de verificação de cabeçalho contidas no que vier após o cabeçalho IPv4. Por exemplo, em nosso programa `ping` (Figura 28.14), devemos calcular a soma de verificação ICMPv4 e armazená-la no cabeçalho ICMPv4, antes de chamar `sendto`.

Diferenças do IPv6

Existem algumas diferenças nos soquetes brutos IPv6 (RFC 3542 [Stevens *et al*., 2003]):

- Todos os campos nos cabeçalhos de protocolo enviados ou recebidos em um soquete bruto IPv6 estão na ordem de byte de rede.
- No IPv6, não há nada semelhante à opção de soquete `IP_HDRINCL` do IPv4. Pacotes IPv6 completos (incluindo o cabeçalho IPv6 ou cabeçalhos de extensão) não podem ser lidos nem gravados em um soquete bruto IPv6. Quase todos os campos em um cabeçalho IPv6 e todos os cabeçalhos de extensão estão disponíveis para a aplicação por intermédio de opções de soquete ou dados auxiliares (veja o Exercício 28.1). Se uma aplicação precisar ler ou gravar datagramas IPv6 completos, o acesso a enlace de dados (descrito no Capítulo 29) deve ser utilizado.
- As somas de verificação em soquetes IPv6 brutos são tratadas diferentemente, conforme será descrito em breve.

Opção de soquete `IPV6_CHECKSUM`

Para um soquete bruto ICMPv6, o kernel sempre calcula e armazena a soma de verificação no cabeçalho ICMPv6. Isso é diferente de um soquete bruto ICMPv4, no qual a aplicação deve fazer isso sozinha (compare as Figuras 28.14 e 28.16). Embora o ICMPv4 e o ICMPv6 exijam que o remetente calcule a soma de verificação, o ICMPv6 inclui um pseudocabeçalho em sua soma de verificação (discutiremos o conceito de pseudocabeçalho quando calcularmos a

soma de verificação de UDP, na Figura 29.14). Um dos campos nesse pseudocabeçalho é o endereço IPv6 de origem e normalmente a aplicação permite que o kernel escolha esse valor. Para evitar que a aplicação tenha que tentar escolher esse endereço somente para calcular a soma de verificação, é mais fácil permitir que o kernel calcule a soma de verificação.

Para outros soquetes de IPv6 brutos (isto é, aqueles criados com um terceiro argumento para `socket`, diferente de `IPPROTO_ICMPV6`), uma opção de soquete diz ao kernel se deve calcular e armazenar uma soma de verificação em pacotes enviados e verificar essa soma em pacotes recebidos. Por padrão, essa opção é desativada, devendo ser ativada por meio da configuração da opção com um valor não-negativo, como em

```
int  offset = 2;

if (setsockopt(sockfd, IPPROTO_IPV6, IPV6_CHECKSUM,
               &offset, sizeof(offset)) < 0)
   error
```

Isso não apenas ativa as somas de verificação nesse soquete, como também informa ao kernel sobre o deslocamento de byte da soma de verificação de 16 bits: 2 bytes a partir do início dos dados da aplicação, nesse exemplo. Para desativar a opção, ela deve ser configurada como –1. Quando ativada, o kernel calculará e armazenará a soma de verificação para pacotes enviados no soquete e também verificará as somas de verificação dos pacotes recebidos no soquete.

28.4 Entrada de soquete bruto

A primeira pergunta que devemos responder com relação à entrada de soquete bruto é: Quais datagramas IP recebidos o kernel passa para soquetes brutos? As seguintes regras se aplicam:

- Os pacotes UDP e TCP recebidos *nunca* são passados para um soquete bruto. Se um processo quer ler datagramas IP contendo pacotes UDP ou TCP, os pacotes devem ser lidos na camada de enlace de dados, conforme será descrito no Capítulo 29.
- A *maioria* dos pacotes ICMP é passada para um soquete bruto depois que o kernel terminou o processamento da mensagem ICMP. As implementações derivadas do Berkeley passam todos os pacotes ICMP recebidos para um soquete bruto, diferentes da solicitação de eco, da solicitação de indicação de tempo e da solicitação de máscara de endereço (página 302 e 303 do TCPv2). Essas três mensagens ICMP são processadas inteiramente pelo kernel.
- *Todos* os pacotes IGMP são passados para um soquete bruto depois que o kernel terminou o processamento da mensagem IGMP.
- *Todos* os datagramas IP com um campo de protocolo que o kernel não entende são passados para um soquete bruto. O único processamento de kernel feito nesses pacotes é a verificação mínima de alguns campos de cabeçalho IP: a versão de IP, a soma de verificação de cabeçalho IPv4, o comprimento do cabeçalho e o endereço IP de destino (páginas 213 a 220 do TCPv2).
- Se o datagrama chega em fragmentos, nada é passado para um soquete bruto até que todos os fragmentos tenham chegado e tenham sido montados.

Quando o kernel tem um datagrama IP para passar aos soquetes brutos, todos os soquetes brutos de todos os processos são examinados, em busca de todos os soquetes correspondentes. Uma cópia do datagrama IP é distribuída para *cada* soquete correspondente. Os testes a seguir são realizados para cada soquete bruto e somente se todos os três testes forem verdadeiros é que o datagrama será distribuído para o soquete:

- Se um *protocolo* não-zero é especificado, quando o soquete bruto é criado (o terceiro argumento de `socket`), então o campo de protocolo do datagrama recebido deve corresponder a esse valor ou o datagrama não será distribuído para esse soquete.

- Se um endereço IP local é limitado ao soquete bruto por meio de `bind`, então o endereço IP de destino do datagrama recebido deve corresponder a esse endereço de limite ou o datagrama não será distribuído para esse soquete.
- Se um endereço IP estrangeiro foi especificado para o soquete bruto por meio de `connect`, então o endereço IP de origem do datagrama recebido deve corresponder a esse endereço conectado ou o datagrama não será distribuído para esse soquete.

Observe que, se um soquete bruto é criado com um *protocolo* igual a 0 e nem `bind` nem `connect` é chamado, então esse soquete recebe uma cópia de *cada* datagrama bruto que o kernel passa para soquetes brutos.

Quando um datagrama recebido é passado para um soquete IPv4 bruto, o datagrama inteiro, incluindo o cabeçalho IP, é passado para o processo. Para um soquete bruto IPv6, somente o payload (isto é, nenhum cabeçalho IPv6 nem quaisquer cabeçalhos de extensão) é passado para o soquete (por exemplo, Figuras 28.11 e 28.22).

> No cabeçalho IPv4 passado para a aplicação, `ip_len`, `ip_off` e `ip_id` são ordenados pelo byte de host e `ip_len` contém somente o comprimento do payload de IP (com o comprimento do cabeçalho IP subtraído), mas os campos restantes são ordenados pelo byte de rede. No Linux, todos os campos são deixados em ordem de byte de rede.
>
> Conforme anteriormente mencionado, a interface de soquete bruto é definida de modo a fornecer uma interface idêntica àquela que um protocolo teria se fosse residente no kernel; portanto, os conteúdos dos campos são dependentes do kernel do SO.
>
> Mencionamos, na seção anterior, que todos os campos em um datagrama recebidos em um soquete bruto IPv6 são deixados em ordem de byte de rede.

Filtragem de tipo de ICMPv6

Um soquete bruto ICMPv4 recebe a maioria das mensagens ICMPv4 recebidas pelo kernel. Mas o ICMPv6 é um superconjunto do ICMPv4, incluindo a funcionalidade de ARP e IGMP (Seção 2.2). Portanto, um soquete bruto ICMPv6 pode receber muito mais pacotes, comparado a um soquete bruto ICMPv4. Mas a maioria das aplicações que utilizam um soquete bruto está interessada somente em um pequeno subconjunto de todas as mensagens ICMP.

Para reduzir o número de pacotes passados do kernel para a aplicação, através de um soquete bruto ICMPv6, é fornecido um filtro especificado pela aplicação. Um filtro é declarado com um tipo de dados `struct icmp6_filter`, que é definido pela inclusão de `<netinet/icmp6.h>`. O filtro atual para um soquete bruto ICMPv6 é configurado e buscado utilizando-se `setsockopt` e `getsockopt`, com um *nível* `IPPROTO_ICMPV6` e um *optname* `ICMP6_FILTER`.

Seis macros operam na estrutura `icmp6_filter`.

```
#include <netinet/icmp6.h>
void ICMP6_FILTER_SETPASSALL(struct icmp6_filter *filt);
void ICMP6_FILTER_SETBLOCKALL(struct icmp6_filter *filt);
void ICMP6_FILTER_SETPASS(int msgtype, struct icmp6_filter *filt);
void ICMP6_FILTER_SETBLOCK(int msgtype, struct icmp6_filter *filt);
int  ICMP6_FILTER_WILLPASS(int msgtype, const struct icmp6_filter *filt);
int  ICMP6_FILTER_WILLBLOCK(int msgtype, const struct icmp6_filter *filt);
                As duas retornam: 1 se o filtro passar (bloquear) o tipo de mensagem; 0, caso contrário
```

O argumento *filt* para todas as macros é um ponteiro para uma variável `icmp6_filter`, que é modificada pelas primeiras quatro macros e examinada pelas duas macros finais. O argumento *msgtype* é um valor entre 0 e 255 e especifica o tipo de mensagem ICMP.

A macro `SETPASSALL` especifica que todos os tipos de mensagem devem ser passados para a aplicação, enquanto as macros `SETBLOCKALL` especificam que nenhum tipo de men-

sagem deve ser passado. Por padrão, quando um soquete bruto ICMPv6 é criado, todos os tipos de mensagem ICMPv6 são passados para a aplicação.

A macro `SETPASS` permite que um tipo de mensagem específico seja passado para a aplicação, enquanto a macro `SETBLOCK` bloqueia um tipo de mensagem específico. A macro `WILLPASS` retorna 1, se o tipo de mensagem especificado é passado pelo filtro, ou 0, caso contrário; a macro `WILLBLOCK` retorna 1, se o tipo de mensagem especificado é bloqueado pelo filtro, ou 0, caso contrário.

Como um exemplo, considere a seguinte aplicação, que quer receber somente anúncios de roteador ICMPv6:

```
struct icmp6_filter myfilt;

fd = Socket(AF_INET6, SOCK_RAW, IPPROTO_ICMPV6);

ICMP6_FILTER_SETBLOCKALL(&myfilt);
ICMP6_FILTER_SETPASS(ND_ROUTER_ADVERT, &myfilt);
Setsockopt(fd, IPPROTO_ICMPV6, ICMP6_FILTER, &myfilt, sizeof(myfilt));
```

Primeiro, bloqueamos todos os tipos de mensagem (pois o padrão é passá-los todos) e, então, passamos somente anúncios de roteador. Apesar de nossa utilização do filtro, a aplicação deve estar preparada para receber todos os tipos de pacotes ICMPv6, pois aqueles que chegam entre `socket` e `setsockopt` serão adicionados na fila de recebimento. A opção `ICMP6_FILTER` é simplesmente uma otimização.

28.5 Programa `ping`

Nesta seção, desenvolveremos e apresentaremos uma versão do programa `ping` que funciona tanto com o IPv4 como com o IPv6. Desenvolveremos nosso próprio programa, em vez de apresentar o código-fonte publicamente disponível, por duas razões. Primeira: o programa `ping` publicamente disponível sofre de um mal de programação comum, conhecido como *excesso de recursos*: ele suporta dezenas de opções diferentes. Nosso objetivo ao examinar um programa `ping` é entender os conceitos e técnicas de programação de redes sem sermos distraídos por todas essas opções. Nossa versão de `ping` suporta somente uma opção e é aproximadamente cinco vezes menor que a versão pública. Segunda: a versão pública funciona somente com o IPv4 e queremos mostrar uma que também suporte o IPv6.

O funcionamento de `ping` é extremamente simples: uma solicitação de eco ICMP é enviada para algum endereço IP e esse nó responde com um eco ICMP. Essas duas mensagens ICMP são suportadas tanto sob o IPv4 como sob o IPv6. A Figura 28.1 mostra o formato das mensagens ICMP.

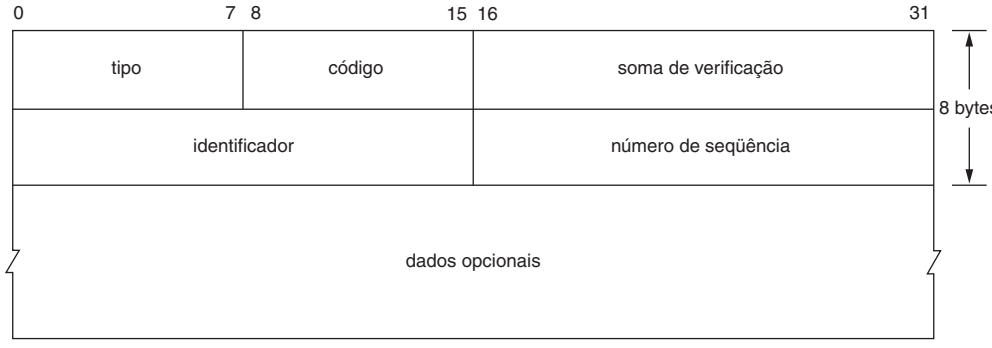


Figura 28.1 Formato da solicitação de eco ICMPv4 e ICMPv6 e mensagens de resposta de eco.

As Figuras A.15 e A.16 mostram os valores de *tipo* para essas mensagens e também que o *código* é 0. Veremos que configuramos o *identificador* com o PID do processo `ping` e incrementamos o *número de seqüência* por um, para cada pacote que enviamos. Armazenamos a indicação de tempo de 8 bytes, de quando o pacote é enviado como *dados opcionais*. As regras do ICMP exigem que o *identificador*, o *número de seqüência* e todos os *dados opcionais* sejam retornados na resposta de eco. Armazenar a indicação de tempo no pacote nos permite calcular o RTT, quando a resposta é recebida.

A Figura 28.2 mostra alguns exemplos de nosso programa. O primeiro utiliza IPv4 e o segundo, IPv6. Observe que fizemos nosso programa `ping` configurar o ID de usuário, pois ele recebe privilégios de superusuário para criar um soquete bruto.

A Figura 28.3 é uma visão geral das funções que compreendem nosso programa `ping`.

O programa opera em duas partes: uma metade lê tudo que é recebido em um soquete bruto, imprimindo as respostas de eco ICMP, e a outra metade envia uma solicitação de eco ICMP uma vez por segundo. A segunda metade é ativada por um sinal `SIGALRM` uma vez por segundo.

A Figura 28.4 mostra nosso cabeçalho `ping.h` que é incluído por todos os nossos arquivos de programa.

```
freebsd % ping www.google.com
PING www.google.com (216.239.57.99): 56 data bytes
64 bytes from 216.239.57.99: seq=0, ttl=53, rtt=5.611 ms
64 bytes from 216.239.57.99: seq=1, ttl=53, rtt=5.562 ms
64 bytes from 216.239.57.99: seq=2, ttl=53, rtt=5.589 ms
64 bytes from 216.239.57.99: seq=3, ttl=53, rtt=5.910 ms

freebsd % ping www.kame.net
PING orange.kame.net (2001:200:0:4819:203:47ff:fea5:3085): 56 data bytes
64 bytes from 2001:200:0:4819:203:47ff:fea5:3085: seq=0, hlim=52, rtt=422.066 ms
64 bytes from 2001:200:0:4819:203:47ff:fea5:3085: seq=1, hlim=52, rtt=417.398 ms
64 bytes from 2001:200:0:4819:203:47ff:fea5:3085: seq=2, hlim=52, rtt=416.528 ms
64 bytes from 2001:200:0:4819:203:47ff:fea5:3085: seq=3, hlim=52, rtt=429.192 ms
```

Figura 28.2 Exemplo de saída de nosso programa `ping`.

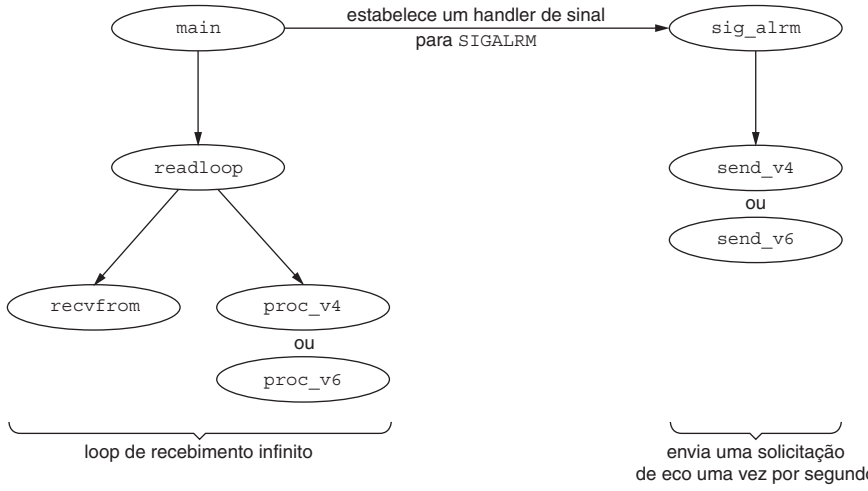


Figura 28.3 Visão geral das funções em nosso programa `ping`.

ping/ping.h

```
 1 #include   "unp.h"
 2 #include   <netinet/in_systm.h>
 3 #include   <netinet/ip.h>
 4 #include   <netinet/ip_icmp.h>

 5 #define BUFSIZE   1500

 6            /* globais */
 7 char   sendbuf[BUFSIZE];

 8 int    datalen;      /* número de bytes de dados após o cabeçalho ICMP */
 9 char  *host;
10 int    nsent;        /* adiciona 1 para cada sendto() */
11 pid_t  pid;          /* nosso PID */
12 int    sockfd;
13 int    verbose;

14            /* protótipos de função */
15 void   init_v6(void);
16 void   proc_v4(char *, ssize_t, struct msghdr *, struct timeval *);
17 void   proc_v6(char *, ssize_t, struct msghdr *, struct timeval *);
18 void   send_v4(void);
19 void   send_v6(void);
20 void   readloop(void);
21 void   sig_alrm(int);
22 void   tv_sub(struct timeval *, struct timeval *);

23 struct proto {
24    void   (*fproc) (char *, ssize_t, struct msghdr *, struct timeval *);
25    void   (*fsend) (void);
26    void   (*finit) (void);
27    struct sockaddr *sasend;    /* sockaddr{} para enviar, de getaddrinfo */
28    struct sockaddr *sarecv;    /* sockaddr{} para receber */
29    socklen_t salen;            /* comprimento de sockaddr{}s */
30    int    icmpproto;           /* Valor de IPPROTO_xxx para ICMP */
31 } *pr;

32 #ifdef IPV6

33 #include   <netinet/ip6.h>
34 #include   <netinet/icmp6.h>

35 #endif
```

ping/ping.h

Figura 28.4 Cabeçalho `ping.h`.

Inclusão dos cabeçalhos IPv4 e ICMPv4

1–22 Incluímos os cabeçalhos IPv4 e ICMPv4 básicos, definimos algumas variáveis globais e nossos protótipos de função.

Definição da estrutura `proto`

23–31 Utilizamos a estrutura `proto` para tratar das diferenças entre IPv4 e IPv6. Essa estrutura contém dois ponteiros de função, dois ponteiros para estruturas de endereço de soquete, o tamanho das estruturas de endereço de soquete e o valor de protocolo para ICMP. O ponteiro global `pr` apontará para uma das estruturas que inicializaremos para IPv4 ou IPv6.

Inclusão dos cabeçalhos IPv6 e ICMPv6

32–35 Incluímos dois cabeçalhos que definem as estruturas e constantes IPv6 e ICMPv6 (RFC 3542 [Stevens *et al*., 2003]).

A função `main` aparece na Figura 28.5.

ping/main.c

```
 1 #include   "ping.h"

 2 struct proto proto_v4 =
 3     { proc_v4, send_v4, NULL, NULL, NULL, 0, IPPROTO_ICMP };

 4 #ifdef IPV6
 5 struct proto proto_v6 =
 6     { proc_v6, send_v6, init_v6, NULL, NULL, 0, IPPROTO_ICMPV6 };
 7 #endif

 8 int    datalen = 56;      /* dados que entram na solicitação de eco ICMP */

 9 int
10 main(int argc, char **argv)
11 {
12     int    c;
13     struct addrinfo *ai;
14     char   *h;

15     opterr = 0;           /* não quer getopt() gravando em stderr */
16     while ( (c = getopt(argc, argv, "v")) != -1) {
17         switch (c) {
18         case 'v':
19             verbose++;
20             break;

21         case '?':
22             err_quit("unrecognized option: %c", c);
23         }
24     }

25     if (optind != argc - 1)
26         err_quit("usage: ping [ -v ] <hostname>");
27     host = argv[optind];

28     pid = getpid() & 0xffff;      /* O campo de ID ICMP é de 16 bits */
29     Signal(SIGALRM, sig_alrm);

30     ai = Host_serv(host, NULL, 0, 0);

31     h = Sock_ntop_host(ai->ai_addr, ai->ai_addrlen);
32     printf("PING %s (%s): %d data bytes\n",
33             ai->ai_canonname ? ai->ai_canonname : h, h, datalen);

34         /* inicializa de acordo com o protocolo */
35     if (ai->ai_family == AF_INET) {
36         pr = &proto_v4;
37 #ifdef IPV6
38     } else if (ai->ai_family == AF_INET6) {
39         pr = &proto_v6;
40         if (IN6_IS_ADDR_V4MAPPED(&(((struct sockaddr_in6 *)
41                                     ai->ai_addr)->sin6_addr)))
42             err_quit("cannot ping IPv4-mapped IPv6 address");
43 #endif
44     } else
45         err_quit("unknown address family %d", ai->ai_family);

46     pr->sasend = ai->ai_addr;
47     pr->sarecv = Calloc(1, ai->ai_addrlen);
48     pr->salen = ai->ai_addrlen;

49     readloop();

50     exit(0);
51 }
```

ping/main.c

Figura 28.5 Função `main`.

Definição das estruturas `proto` para IPv4 e IPv6

2–7 Definimos uma estrutura `proto` para IPv4 e IPv6. Os ponteiros de estrutura de endereço de soquete são inicializados como ponteiros nulos, pois ainda não sabemos se utilizaremos IPv4 ou IPv6.

Comprimento dos dados opcionais

8 Configuramos o volume de dados opcionais que é enviado com a solicitação de eco ICMP como 56 bytes. Isso gerará um datagrama IPv4 de 84 bytes (cabeçalho IPv4 de 20 bytes e cabeçalho ICMP de 8 bytes) ou um datagrama IPv6 de 104 bytes. Todos os dados que acompanham uma solicitação de eco devem ser enviados de volta na resposta de eco. Armazenaremos o momento no qual enviamos uma solicitação de eco nos primeiros 8 bytes dessa área de dados e, então, utilizaremos isso para calcular e imprimir o RTT, quando a resposta de eco for recebida.

Tratamento das opções de linha de comando

15–24 A única opção de linha de comando que suportamos é `–v`, que causará a impressão da maioria das mensagens ICMP recebidas. (Não imprimimos respostas de eco pertencentes à outra cópia de `ping` que está em execução.) Um handler de sinal é estabelecido para `SIGALRM` e veremos que esse sinal é gerado uma vez por segundo e faz uma solicitação de eco ICMP ser enviada.

Processamento do argumento de nome de host

31–48 Uma string de nome de host ou de endereço IP é um argumento obrigatório que é processado por nossa função `host_serv`. A estrutura `addrinfo` retornada contém a família de protocolos, `AF_INET` ou `AF_INET6`. Inicializamos a variável global `pr` com a estrutura `proto` correta. Também nos certificamos de que um endereço IPv6 não é realmente um endereço IPv6 mapeado em IPv4, chamando `IN6_IS_ADDR_V4MAPPED`, pois, mesmo que o endereço retornado seja IPv6, os pacotes IPv4 serão enviados para o host. (Poderíamos trocar e utilizar IPv4 quando isso acontecesse.) A estrutura de endereço de soquete que já foi alocada pela função `getaddrinfo` é utilizada para envio e outra estrutura de endereço de soquete do mesmo tamanho é alocada para recebimento.

49 É na função `readloop` que o processamento acontece. Mostraremos isso na Figura 28.6.

Criação do soquete

12–13 Um soquete bruto do protocolo apropriado é criado. A chamada a `setuid` configura nosso ID de usuário efetivo como nosso ID de usuário real, no caso de o programa ter configurado o ID de usuário, em vez de ser executado por root. O programa deve ter privilégios de superusuário para criar o soquete bruto, mas, agora que o soquete está criado, podemos abandonar os privilégios extras. É sempre melhor abandonar um privilégio extra quando ele não é mais necessário, apenas para a eventualidade de o programa ter um bug latente que alguém poderia explorar.

ping/readloop.c

```
1 #include   "ping.h"

2 void
3 readloop(void)
4 {
5     int     size;
6     char    recvbuf[BUFSIZE];
7     char    controlbuf[BUFSIZE];
8     struct msghdr msg;
```

Figura 28.6 Função `readloop` (*continua*).

```
 9      struct iovec iov;
10      ssize_t n;
11      struct timeval tval;

12      sockfd = Socket(pr->sasend->sa_family, SOCK_RAW, pr->icmpproto);
13      setuid(getuid());     /* não precisamos mais de permissões especiais */
14      if (pr->finit)
15          (*pr->finit) ();

16      size = 60 * 1024;     /* OK, se setsockopt falhar */
17      setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));

18      sig_alrm(SIGALRM);    /* envia o primeiro pacote */

19      iov.iov_base = recvbuf;
20      iov.iov_len = sizeof(recvbuf);
21      msg.msg_name = pr->sarecv;
22      msg.msg_iov = &iov;
23      msg.msg_iovlen = 1;
24      msg.msg_control = controlbuf;
25      for ( ; ; ) {
26          msg.msg_namelen = pr->salen;
27          msg.msg_controllen = sizeof(controlbuf);
28          n = recvmsg(sockfd, &msg, 0);
29          if (n < 0) {
30              if (errno == EINTR)
31                  continue;
32              else
33                  err_sys("recvmsg error");
34          }
35          Gettimeofday(&tval, NULL);
36          (*pr->fproc) (recvbuf, n, &msg, &tval);
37      }
38  }
```

ping/readloop.c

Figura 28.6 Função `readloop` (*continuação*).

Realização da inicialização específica do protocolo

14–15 Se o protocolo especificou uma função de inicialização, a chamamos. Mostramos a função de inicialização de IPv6 na Figura 28.10.

Configuração do tamanho do buffer de recebimento do soquete

16–17 Tentamos configurar o tamanho do buffer de recebimento do soquete como 61.440 bytes (60 x 1024), que deve ser maior que o default. Fazemos isso para o caso de o usuário usar `ping` no endereço de broadcast IPv4 ou em um endereço de multicast, um dos quais pode gerar uma grande quantidade de respostas. Tornando o buffer maior, há uma chance menor de que o buffer de recebimento do soquete estoure.

Envio do primeiro pacote

18 Chamamos nosso handler de sinal, que veremos enviar um pacote e agendar um sinal `SIGALRM` para um segundo no futuro. Não é comum ver um handler de sinal chamado diretamente, como fizemos aqui, mas isso é aceitável. Um handler de sinal é somente uma função C, mesmo que normalmente seja chamada de forma assíncrona.

Configuração de `msghdr` para `recvmsg`

19–24 Configuramos os campos inalteráveis nas estruturas `msghdr` e `iovec` que passaremos para `recvmsg`.

O loop infinito lendo todas as mensagens ICMP

25–37 O loop principal do programa é um loop infinito que lê todos os pacotes retornados no soquete ICMP bruto. Chamamos `gettimeofday` para registrar o momento em que o pacote foi recebido e, então, chamamos a função de protocolo apropriada (`proc_v4` ou `proc_v6`) para processar a mensagem ICMP.

A Figura 28.7 mostra a função `tv_sub`, que subtrai duas estruturas `timeval`, armazenando o resultado na primeira estrutura.

A Figura 28.8 mostra a função `proc_v4`, que processa todas as mensagens ICMPv4 recebidas. Talvez você queira consultar a Figura A.1, que mostra o formato do cabeçalho IPv4. Perceba também que, quando a mensagem ICMPv4 é recebida pelo processo no soquete bruto, o kernel já verificou se os campos básicos dos cabeçalhos IPv4 e ICMPv4 são válidos (páginas 214 e 311 do TCPv2).

lib/tv_sub.c

```
 1 #include   "unp.h"
 2 void
 3 tv_sub(struct timeval *out, struct timeval *in)
 4 {
 5     if((out->tv_usec -= in->tv_usec) < 0) {      /* out -= in */
 6         --out->tv_sec;
 7         out->tv_usec += 1000000;
 8     }
 9     out->tv_sec -= in->tv_sec;
10 }
```

lib/tv_sub.c

Figura 28.7 Função `tv_sub`: subtrai duas estruturas `timeval`.

ping/proc_v4.c

```
 1 #include   "ping.h"
 2 void
 3 proc_v4(char *ptr, ssize_t len, struct msghdr *msg, struct timeval *tvrecv)
 4 {
 5     int    hlen1, icmplen;
 6     double rtt;
 7     struct ip *ip;
 8     struct icmp *icmp;
 9     struct timeval *tvsend;
10     ip = (struct ip *) ptr;       /* início do cabeçalho IP */
11     hlen1 = ip->ip_hl << 2;       /* comprimento do cabeçalho IP */
12     if (ip->ip_p != IPPROTO_ICMP)
13         return;                   /* não ICMP */
14     icmp = (struct icmp *) (ptr + hlen1);    /* início do cabeçalho ICMP */
15     if ( (icmplen = len - hlen1) < 8)
16         return;                   /* pacote malformado */
17     if (icmp->icmp_type == ICMP_ECHOREPLY) {
18         if (icmp->icmp_id != pid)
19             return;             /* não é uma resposta para nosso ECHO_REQUEST */
20     if (icmplen < 16)
21             return;             /* não há dados suficientes para utilizar */
22     tvsend = (struct timeval *) icmp->icmp_data;
23     tv_sub(tvrecv, tvsend);
```

Figura 28.8 Função `proc_v4`: processa mensagem ICMPv4 (*continua*).

```
24      rtt = tvrecv->tv_sec * 1000.0 + tvrecv->tv_usec / 1000.0;

25      printf("%d bytes from %s: seq=%u, ttl=%d, rtt=%.3f ms\n",
26             icmplen, Sock_ntop_host(pr->sarecv, pr->salen),
27             icmp->icmp_seq, ip->ip_ttl, rtt);

28      } else if (verbose) {
29         printf(" %d bytes from %s: type = %d, code = %d\n",
30                icmplen, Sock_ntop_host(pr->sarecv, pr->salen),
31                icmp->icmp_type, icmp->icmp_code);
32      }
33  }
```

ping/proc_v4.c

Figura 28.8 Função `proc_v4`: processa mensagem ICMPv4 (*continuação*).

Obtenção do ponteiro para o cabeçalho ICMP

10–16 O campo de comprimento de cabeçalho IPv4 é multiplicado por 4, fornecendo o tamanho do cabeçalho IPv4 em bytes. (Lembre-se de que um cabeçalho IPv4 pode conter opções.) Isso nos permite configurar `icmp` de modo que aponte para o começo do cabeçalho ICMP. Certificamo-nos de que o protocolo de IP é ICMP e de que há dados suficientes ecoados para vermos a indicação de tempo que incluímos na solicitação de eco. A Figura 28.9 mostra os vários cabeçalhos, ponteiros e comprimentos utilizados pelo código.

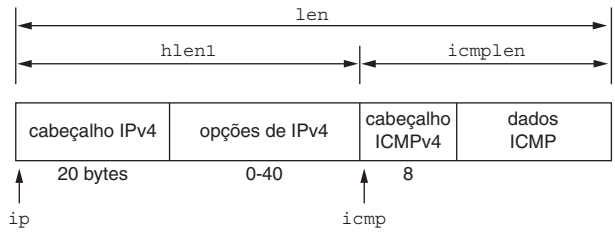


Figura 28.9 Cabeçalhos, ponteiros e comprimentos no processamento da resposta ICMPv4.

Verificação da resposta de eco ICMP

17–21 Se a mensagem é uma resposta de eco ICMP, então devemos verificar o campo identificador para ver se essa resposta está na resposta a uma solicitação que nosso processo enviou. Se o programa `ping` está executando várias vezes nesse host, cada processo obtém uma cópia de todas as mensagens ICMP recebidas.

22–27 Calculamos o RTT subtraindo o momento em que a mensagem foi enviada (contido na parte opcional dos dados da resposta ICMP) do momento atual (apontado pelo argumento de função `tvrecv`). O RTT é convertido de microssegundos para milissegundos e impresso, junto com o campo de número de seqüência e o TTL recebido. O campo de número de seqüência permite que o usuário veja se pacotes foram eliminados, reordenados ou duplicados, e o TTL fornece uma indicação do número de hops entre os dois hosts.

Impressão de todas as mensagens ICMP recebidas se a opção prolixa for especificada

28–32 Se o usuário especificou a opção de linha de comando `-v` (*verbose*), imprimimos os campos de tipo e código de todas as outras mensagens ICMP recebidas.

O processamento de mensagens ICMPv6 é tratado pela função `proc_v6`, mostrada na Figura 28.12. Ela é semelhante à função `proc_v4`; entretanto, como os soquetes IPv6 bru-

tos não retornam o cabeçalho IPv6, ela recebe o limite de hop como dados auxiliares. Isso foi configurado utilizando-se a função `init_v6`, mostrada na Figura 28.10.

A função `init_v6` prepara o soquete para uso.

ping/init_v6.c

```
 1 void
 2 init_v6()
 3 {
 4 #ifdef IPV6
 5     int    on = 1;

 6     if(verbose == 0) {
 7         /*instala um filtro que passa somente ICMP6_ECHO_REPLY, a menos
            que seja prolixo */
 8         struct icmp6_filter myfilt;
 9         ICMP6_FILTER_SETBLOCKALL(&myfilt);
10         ICMP6_FILTER_SETPASS(ICMP6_ECHO_REPLY, &myfilt);
11         setsockopt(sockfd, IPPROTO_IPV6, ICMP6_FILTER, &myfilt,
12                    sizeof(myfilt));
13         /* ignora retorno de erro; o filtro é uma otimização */
14     }

15     /* ignora o erro retornado abaixo; simplesmente não receberemos o
           limite de hop */
16 #ifdef IPV6_RECVHOPLIMIT
17     /* RFC 3542 */
18     setsockopt(sockfd, IPPROTO_IPV6, IPV6_RECVHOPLIMIT, &on, sizeof(on));
19 #else
20     /* RFC 2292 */
21     setsockopt(sockfd, IPPROTO_IPV6, IPV6_HOPLIMIT, &on, sizeof(on));
22 #endif
23 #endif
24 }
```

ping/init_v6.c

Figura 28.10 Função `init_v6`: inicializa soquete ICMPv6.

Configuração do filtro de recebimento ICMPv6

6–14 Se a opção de linha de comando –v não foi especificada, instale um filtro que bloqueie todos os tipos de mensagem ICMP, exceto para a resposta de eco esperada. Isso reduz o número de pacotes recebidos no soquete.

Solicitação de dados auxiliares de `IPV6_HOPLIMIT`

15–22 A API para solicitar o recebimento do limite de hop com pacotes recebidos mudou com o passar do tempo. Preferimos a API mais recente: configurar a opção de soquete `IPV6_RECVHOPLIMIT`. Entretanto, se a constante para essa opção não estiver definida, podemos tentar a API mais antiga: configurar `IPV6_HOPLIMIT` como uma opção. Não verificamos o valor de retorno de `setsockopt`, pois o programa ainda pode fazer trabalho útil sem receber o limite de hop.

A função `proc_v6` (Figura 28.12) processa pacotes recebidos.

Obtenção de um ponteiro para o cabeçalho ICMPv6

11–13 O cabeçalho ICMPv6 são os dados retornados pela operação de recebimento. (Lembre-se de que o cabeçalho IPv6 e os cabeçalhos de extensão, se houver, nunca são retornados como dados normais, mas como dados auxiliares.) A Figura 28.11 mostra os vários cabeçalhos, ponteiros e comprimentos utilizados pelo código.

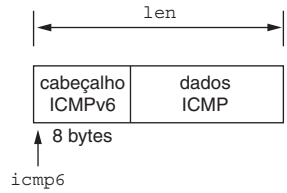


Figura 28.11 Cabeçalhos, ponteiros e comprimentos no processamento da resposta ICMPv6.

Verificação da resposta de eco ICMP

14–37 Se o tipo de mensagem ICMP é uma resposta de eco, verificamos o campo identificador para ver se a resposta é dirigida a nós. Se for, calculamos o RTT e então o imprimimos, junto com o número de seqüência e o limite de hop do IPv6. Obtemos o limite de hop a partir dos dados auxiliares de `IPV6_HOPLIMIT`.

ping/proc_v6.c

```
 1 #include   "ping.h"

 2 void
 3 proc_v6(char *ptr, ssize_t len, struct msghdr *msg, struct timeval *tvrecv)
 4 {
 5 #ifdef IPV6
 6     double rtt;
 7     struct icmp6_hdr *icmp6;
 8     struct timeval *tvsend;
 9     struct cmsghdr *cmsg;
10     int  hlim;

11     icmp6 = (struct icmp6_hdr *) ptr;
12     if (len < 8)
13         return;           /* pacote malformado */

14     if (icmp6->icmp6_type == ICMP6_ECHO_REPLY) {
15         if (icmp6->icmp6_id != pid)
16             return;       /* não é uma resposta para nosso ECHO_REQUEST */
17         if (len < 16)
18             return;       /* não há dados suficientes para utilizar */

19         tvsend = (struct timeval *) (icmp6 + 1);
20         tv_sub(tvrecv, tvsend);
21         rtt = tvrecv->tv_sec * 1000.0 + tvrecv->tv_usec / 1000.0;

22         hlim = -1;
23         for (cmsg = CMSG_FIRSTHDR(msg); cmsg != NULL;
24              cmsg = CMSG_NXTHDR(msg, cmsg)) {
25            if (cmsg->cmsg_level == IPPROTO_IPV6
26                & cmsg->cmsg_type == IPV6_HOPLIMIT) {
27                hlim = *(u_int32_t *) CMSG_DATA(cmsg);
28                break;
29            }
30         }
31         printf("%d bytes from %s: seq=%u, hlim=",
32                len, Sock_ntop_host(pr->sarecv, pr->salen), icmp6->icmp6_seq);
33         if (hlim == -1)
34             printf("???");        /* dados auxiliares ausentes */
35         else
36             printf("%d", hlim);
37         printf(", rtt=%.3f ms\n", rtt);
38     } else if (verbose) {
```

Figura 28.12 Função `proc_v6`: processa mensagem ICMPv6 recebida (*continua*).

```
39          printf(" %d bytes from %s: type = %d, code = %d\n",
40                 len, Sock_ntop_host(pr->sarecv, pr->salen),
41                 icmp6->icmp6_type, icmp6->icmp6_code);
42      }
43  #endif /* IPV6 */
44  }
```

ping/proc_v6.c

Figura 28.12 Função `proc_v6`: processa mensagem ICMPv6 recebida (*continuação*).

Impressão de todas as mensagens ICMP recebidas se a opção prolixa for especificada

38–41 Se o usuário especificou a opção de linha de comando –v, imprimimos os campos de tipo e código de todas as outras mensagens ICMP recebidas.

Nosso handler de sinal para o sinal `SIGALRM` é a função `sig_alrm`, mostrada na Figura 28.13. Vimos, na Figura 28.6, que nossa função `readloop` chama esse handler de sinal uma vez, no início, para enviar o primeiro pacote. Essa função apenas chama a função dependente do protocolo para enviar uma solicitação de eco ICMP (`send_v4` ou `send_v6`) e, então, agenda outro sinal `SIGALRM` para um segundo no futuro.

A função `send_v4`, mostrada na Figura 28.14, constrói uma mensagem de solicitação de eco ICMPv4 e a grava no soquete bruto.

ping/sig_alrm.c

```
1 #include   "ping.h"

2 void
3 sig_alrm(int signo)
4 {
5     (*pr->fsend) ();

6     alarm(1);
7     return;
8 }
```

ping/sig_alrm.c

Figura 28.13 Função `sig_alrm`: handler de sinal `SIGALRM`.

ping/send_v4.c

```
 1 #include   "ping.h"

 2 void
 3 send_v4(void)
 4 {
 5     int    len;
 6     struct icmp *icmp;

 7     icmp = (struct icmp *) sendbuf;
 8     icmp->icmp_type = ICMP_ECHO;
 9     icmp->icmp_code = 0;
10     icmp->icmp_id = pid;
11     icmp->icmp_seq = nsent++;
12     memset(icmp->icmp_data, 0xa5, datalen); /* preenche com padrão */
13     Gettimeofday((struct timeval *) icmp->icmp_data, NULL);

14     len = 8 + datalen;  /* cabeçalho e dados ICMP da soma de verificação */
15     icmp->icmp_cksum = 0;
16     icmp->icmp_cksum = in_cksum((u_short *) icmp, len);

17     Sendto(sockfd, sendbuf, len, 0, pr->sasend, pr->salen);
18 }
```

ping/send_v4.c

Figura 28.14 Função `send_v4`: constrói uma mensagem de solicitação de eco ICMPv4 e a envia.

Construção da mensagem ICMPv4

7–13 A mensagem ICMPv4 é construída. O campo identificador é configurado com nosso PID e o campo de número de seqüência com o `nsent` global, que então é incrementado para o próximo pacote. Armazenamos o padrão `0xa5` na parte dos dados da mensagem ICMP. A hora do dia atual é então armazenada no começo da parte dos dados.

Cálculo da soma de verificação de ICMP

14–16 Para calcular a soma de verificação de ICMP, configuramos o campo de soma de verificação como 0 e chamamos a função `in_cksum`, armazenando o resultado no campo de soma de verificação. A soma de verificação de ICMPv4 é calculada a partir do cabeçalho ICMPv4 e de todos os dados que o seguem.

Envio do datagrama

17 A mensagem ICMP é enviada no soquete bruto. Como não configuramos a opção de soquete `IP_HDRINCL`, o kernel constrói o cabeçalho IPv4 e o coloca como prefixo em nosso buffer.

A soma de verificação de Internet é o complemento de um da soma do complemento de um dos valores de 16 bits a terem soma de verificação. Se o comprimento dos dados é um número ímpar, então 1 byte em 0 é anexado logicamente no fim dos dados, apenas para o cálculo da soma de verificação. Antes de calcular a soma de verificação, o próprio campo de soma de verificação é configurado como 0. Esse algoritmo é utilizado para as somas de verificação de IPv4, ICMPv4, IGMPv4, ICMPv6, UDP e TCP. A RFC 1071 (Braden, Borman e Partridge, 1988) contém informações adicionais e alguns exemplos numéricos.

A Seção 8.7 do TCPv2 fala sobre esse algoritmo com mais detalhe e mostra uma implementação mais eficiente. Nossa função `in_cksum`, mostrada na Figura 28.15, calcula a soma de verificação.

libfree/in_cksum.c

```
 1 uint16_t
 2 in_cksum(uint16_t * addr, int len)
 3 {
 4     int    nleft = len;
 5     uint32_t sum = 0;
 6     uint16_t *w = addr;
 7     uint16_t answer = 0;

 8     /*
 9      * Nosso algoritmo é simples, usando um acumulador de 32 bits (soma),
10      * adicionamos palavras seqüenciais de 16 bits a ele e, no fim, recolhemos
11      * todos os bits de vai-um (carry) dos 16 bits superiores nos 16 bits
12      * inferiores. */
13     while (nleft > 1) {
14         sum += *w++;
15         nleft -= 2;
16     }

17         /* limpa um byte ímpar, se necessário */
18     if (nleft == 1) {
19         *(unsigned char *) (&answer) = *(unsigned char *) w;
20         sum += answer;
21     }

22         /* adiciona os bits de vai-um dos 16 bits superiores nos 16 bits inferiores */
23     sum = (sum >> 16) + (sum & 0xffff); /* adiciona 16 sup aos 16 inf */
24     sum += (sum >> 16);           /* adiciona o vai-um */
25     answer = ~sum;                /* trunca em 16 bits */
26     return (answer);
27 }
```

libfree/in_cksum.c

Figura 28.15 Função `in_cksum`: calcula a soma de verificação Internet.

Algoritmo de soma de verificação de Internet

1-27 O primeiro loop `while` calcula a soma de todos os valores de 16 bits. Se o comprimento é ímpar, então o byte final é adicionado à soma. O algoritmo que mostramos na Figura 28.15 é o simples. O kernel freqüentemente tem um algoritmo de soma de verificação especialmente otimizado, devido ao grande volume de cálculos de soma de verificação realizados pelo kernel.

Esta função foi extraída da versão de domínio público de `ping` cujo autor é Mike Muuss.

A função final para nosso programa `ping` é `send_v6`, mostrada na Figura 28.16, que constrói e envia uma solicitação de eco ICMPv6.

A função `send_v6` é semelhante a `send_v4`, mas note que ela não calcula a soma de verificação de ICMPv6. Conforme mencionamos anteriormente neste capítulo, como a soma de verificação de ICMPv6 utiliza o endereço de origem do cabeçalho IPv6 em seu cálculo, o kernel a calcula para nós, depois de escolher o endereço de origem.

ping/send_v6.c

```
 1 #include   "ping.h"

 2 void
 3 send_v6()
 4 {
 5 #ifdef IPV6
 6     int    len;
 7     struct icmp6_hdr *icmp6;

 8     icmp6 = (struct icmp6_hdr *) sendbuf;
 9     icmp6->icmp6_type = ICMP6_ECHO_REQUEST;
10     icmp6->icmp6_code = 0;
11     icmp6->icmp6_id = pid;
12     icmp6->icmp6_seq = nsent++;
13     memset((icmp6 + 1), 0xa5, datalen); /* preenche com padrão */
14     Gettimeofday((struct timeval *) (icmp6 + 1), NULL);

15     len = 8 + datalen;           /* cabeçalho ICMPv6 de 8 bytes */

16     Sendto(sockfd, sendbuf, len, 0, pr->sasend, pr->salen);
17         /* o kernel calcula e armazena a soma de verificação para nós */
18 #endif /* IPV6 */
19 }
```

ping/send_v6.c

Figura 28.16 Função `send_v6`: constrói e envia uma mensagem de solicitação de eco ICMPv6.

28.6 Programa `traceroute`

Nesta seção, desenvolveremos nossa própria versão do programa `traceroute`. Assim como o programa `ping` que desenvolvemos na seção anterior, desenvolveremos e apresentaremos nossa própria versão, em vez de apresentar a versão publicamente disponível. Faremos isso porque precisamos de uma versão que suporte tanto o IPv4 como o IPv6, e não queremos nos distrair com uma grande quantidade de opções que não são pertinentes à nossa discussão sobre programação de redes.

O `traceroute` permite determinar o caminho que os datagramas IP seguem de nosso host até algum outro destino. Seu funcionamento é simples e o Capítulo 8 do TCPv1 o aborda em detalhes, com numerosos exemplos de sua utilização. O `traceroute` usa o campo TTL do IPv4 ou o campo de limite de hop do IPv6 e duas mensagens ICMP. Ele começa enviando um datagrama UDP para o destino com um TTL (ou limite de hop) igual a 1. Esse datagrama faz com que o roteador do primeiro hop retorne um erro ICMP de "tempo excedido

em trânsito". Então, o TTL é aumentado de um e outro datagrama UDP é enviado, o qual localiza o próximo roteador no caminho. Quando o datagrama UDP alcança o destino final, o objetivo é fazer esse host retornar um erro ICMP "porta inacessível". Isso é feito por meio do envio do datagrama UDP para uma porta aleatória que não esteja (espera-se) em utilização naquele host.

As versões anteriores do `traceroute` eram capazes de configurar o campo de TTL no cabeçalho IPv4, somente configurando a opção de soquete `IP_HDRINCL` e, então, construindo seu próprio cabeçalho IPv4. Os sistemas atuais, entretanto, fornecem uma opção de soquete `IP_TTL` que permite especificar o TTL a ser utilizado para datagramas de saída. (Essa opção de soquete foi introduzida com a versão de 4.3BSD Reno.) É mais fácil configurar essa opção de soquete do que construir um cabeçalho IPv4 completo (embora mostremos como construir cabeçalhos IPv4 e UDP, na Seção 29.7). A opção de soquete IPv6 `IPV6_UNICAST_HOPS` permite controlar o campo de limite de hop para datagramas IPv6.

A Figura 28.17 mostra nosso cabeçalho `trace.h`, que todos os nossos arquivos de programa incluem.

1–11 Incluímos os cabeçalhos IPv4 padrão que definem as estruturas e constantes IPv4, ICMPv4 e UDP. A estrutura `rec` define a parte dos dados do datagrama UDP que enviamos, mas veremos que nunca precisamos examinar esses dados. Eles são enviados principalmente para propósitos de depuração.

Definição da estrutura `proto`

32–43 Assim como no caso do nosso programa `ping` da seção anterior, tratamos das diferenças de protocolo entre IPv4 e IPv6 definindo uma estrutura `proto` que contém ponteiros de função, ponteiros para estruturas de endereço de soquete e outras constantes que diferem entre as duas versões de IP. A variável global `pr` será configurada de forma a apontar para uma dessas estruturas que é inicializada para IPv4 ou IPv6, depois que o endereço de destino é processado pela função `main` (pois é ele que especifica se utilizamos IPv4 ou IPv6).

Inclusão dos cabeçalhos IPv6

44–47 Incluímos os cabeçalhos que definem as estruturas e constantes IPv6 e ICMPv6.

traceroute/trace.h

```
 1 #include   "unp.h"
 2 #include   <netinet/in_systm.h>
 3 #include   <netinet/ip.h>
 4 #include   <netinet/ip_icmp.h>
 5 #include   <netinet/udp.h>

 6 #define BUFSIZE   1500

 7 struct rec {                    /* de dados de UDP de saída */
 8     u_short rec_seq;            /* número de seqüência */
 9     u_short rec_ttl;            /* TTL do pacote deixado com */
10     struct timeval rec_tv;      /* tempo restante do pacote */
11 };

12            /*globais */
13 char    recvbuf[BUFSIZE];
14 char    sendbuf[BUFSIZE];

15 int     datalen;         /* n° de bytes de dados após o cabeçalho ICMP */
16 char   *host;
17 u_short sport, dport;
18 int     nsent;                  /* adiciona 1 para cada sendto() */
```

Figura 28.17 Cabeçalho `trace.h` (*continua*).

```
19 pid_t   pid;                     /* nosso PID */
20 int     probe, nprobes;
21 int     sendfd, recvfd;          /* envia no soquete UDP, lê no soquete
                                     ICMP bruto */
22 int     ttl, max_ttl;
23 int     verbose;

24            /* protótipos de função */
25 const char *icmpcode_v4(int);
26 const char *icmpcode_v6(int);
27 int    recv_v4(int, struct timeval *);
28 int    recv_v6(int, struct timeval *);
29 void   sig_alrm(int);
30 void   traceloop(void);
31 void   tv_sub(struct timeval *, struct timeval *);

32 struct proto {
33    const char *(*icmpcode) (int);
34    int    (*recv) (int, struct timeval *);
35    struct sockaddr *sasend;  /* sockaddr{} para enviar, de getaddrinfo */
36    struct sockaddr *sarecv;  /* sockaddr{} para receber */
37    struct sockaddr *salast;  /* último sockaddr{} para receber */
38    struct sockaddr *sabind;  /* sockaddr{} para vincular porta de origem */
39    socklen_t salen;          /* comprimento de sockaddr{}s */
40    int    icmpproto;         /* Valor de IPPROTO_xxx para ICMP */
41    int    ttllevel;          /* nível de setsockopt() para configurar TTL */
42    int    ttloptname;        /* nome setsockopt() para configurar TTL */
43 } *pr;

44 #ifdef IPV6

45 #include   <netinet/ip6.h>
46 #include   <netinet/icmp6.h>

47 #endif
```

traceroute/trace.h

Figura 28.17 Cabeçalho `trace.h` (*continuação*).

A função `main` é mostrada na Figura 28.18. Ela processa os argumentos de linha de comando, inicializa o ponteiro `pr` para IPv4 ou IPv6 e chama nossa função `traceloop`.

Definição das estruturas `proto`

2–9 Definimos as duas estruturas `proto`, uma para IPv4 e a outra para IPv6, embora os ponteiros para as estruturas de endereço de soquete não sejam alocados até o fim dessa função.

Configuração dos valores default

10–13 O TTL máximo ou limite de hop que o programa utiliza tem como default o valor 30, embora forneçamos a opção de linha de comando –`m` para permitir ao usuário alterar isso. Para cada TTL, enviamos três pacotes de sondagem, mas isso poderia ser alterado com outra opção de linha de comando. A porta de destino inicial é 32768 + 666, que será incrementada por 1 toda vez enviarmos um datagrama UDP. Esperamos que essas portas não estejam em uso no host de destino, quando os datagramas finalmente alcançarem o destino, mas não há nenhuma garantia disso.

Processamento dos argumentos de linha de comando

19–37 A opção de linha de comando –`v` faz com que a maioria das mensagens ICMP recebidas seja impressa.

traceroute/main.c

```
 1 #include   "trace.h"

 2 struct proto proto_v4 = { icmpcode_v4, recv_v4, NULL, NULL, NULL, NULL, 0,
 3     IPPROTO_ICMP, IPPROTO_IP, IP_TTL
 4 };

 5 #ifdef IPV6
 6 struct proto proto_v6 = { icmpcode_v6, recv_v6, NULL, NULL, NULL, NULL, 0,
 7     IPPROTO_ICMPV6, IPPROTO_IPV6, IPV6_UNICAST_HOPS
 8 };
 9 #endif

10 int     datalen = sizeof(struct rec);/* defaults */
11 int     max_ttl = 30;
12 int     nprobes = 3;
13 u_short dport = 32768 + 666;

14 int
15 main(int argc, char **argv)
16 {
17     int    c;
18     struct addrinfo *ai;
19     char   *h;

20     opterr = 0;                 /* não quer getopt() gravando em stderr */
21     while ( (c = getopt(argc, argv, "m:v")) != -1) {
22         switch (c) {
23         case 'm':
24             if ( (max_ttl = atoi(optarg)) <= 1)
25                 err_quit("invalid -m value");
26             break;

27         case 'v':
28             verbose++;
29             break;

30         case '?':
31             err_quit("unrecognized option: %c", c);
32         }
33     }

34     if (optind != argc - 1)
35         err_quit("usage: traceroute [ -m <maxttl> -v ] <hostname>");
36     host = argv[optind];

37     pid = getpid();
38     Signal(SIGALRM, sig_alrm);

39     ai = Host_serv(host, NULL, 0, 0);

40     h = Sock_ntop_host(ai->ai_addr, ai->ai_addrlen);
41     printf("traceroute to %s (%s): %d hops max, %d data bytes\n",
42            ai->ai_canonname ? ai->ai_canonname : h, h, max_ttl, datalen);

43     /* inicializa de acordo com o protocolo */
44     if (ai->ai family == AF_INET) {
45         pr = &proto_v4;
46 #ifdef IPV6
47     } else if (ai->ai_family == AF_INET6) {
48         pr = &proto_v6;
49         if (IN6_IS_ADDR_V4MAPPED
50             (&(((struct sockaddr_in6 *) ai->ai_addr)->sin6_addr)))
51             err_quit("cannot traceroute IPv4-mapped IPv6 address");
52 #endif
```

Figura 28.18 Função `main` do programa `traceroute` (*continua*).

```
53      } else
54          err_quit("unknown address family %d", ai->ai_family);

55      pr->sasend = ai->ai_addr;   /* contém endereço de destino */
56      pr->sarecv = Calloc(1, ai->ai_addrlen);
57      pr->salast = Calloc(1, ai->ai_addrlen);
58      pr->sabind = Calloc(1, ai->ai_addrlen);
59      pr->salen = ai->ai_addrlen;

60      traceloop();

61      exit(0);
62  }
```

traceroute/main.c

Figura 28.18 Função `main` do programa `traceroute` (*continuação*).

Processamento do argumento de nome de host ou de endereço IP e término da inicialização

38–58 O nome de host ou endereço IP de destino é processado por nossa função `host_serv`, retornando um ponteiro para uma estrutura `addrinfo`. Dependendo do tipo de endereço retornado, IPv4 ou IPv6, terminamos a inicialização da estrutura `proto`, armazenamos o ponteiro na variável global `pr` e alocamos estruturas de endereço de soquete adicionais do tamanho correto.

59 A função `traceloop`, mostrada na Figura 28.19, envia os datagramas e lê as mensagens ICMP retornadas. Esse é o loop principal do programa.

Em seguida, examinamos nossa função `traceloop`, mostrada na Figura 28.19.

Criação do soquete bruto

9–10 Precisamos de dois soquetes: um bruto, no qual lemos todas as mensagens de ICMP retornadas, e um UDP, no qual enviamos os pacotes de sondagem com os TTLs crescentes. Após criarmos o soquete bruto, redefinimos nosso ID de usuário efetivo como nosso ID de usuário real, pois não precisamos mais de privilégios de superusuário.

Configuração do filtro de recebimento ICMPv6

11–20 Se estivermos rastreando a rota para um endereço IPv6 e a opção de linha de comando –v não tiver sido especificada, instalamos um filtro que bloqueia todos os tipos de mensagem ICMP, exceto aquelas esperadas: "tempo excedido" ou "destino inacessível". Isso reduz o número de pacotes recebidos no soquete.

Criação do soquete UDP e vinculação à porta de origem

21–25 Usamos `bind` em uma porta de origem para o soquete UDP que é utilizado para envio, usando os 16 bits de ordem inferior de nosso PID com o bit de ordem superior configurado como 1. Como é possível que várias cópias do programa `traceroute` estejam em execução em dado momento, precisamos de uma maneira de determinar se uma mensagem ICMP recebida foi gerada em resposta a um de nossos datagramas ou a um datagrama enviado por outra cópia do programa. Utilizamos a porta de origem no cabeçalho UDP para identificar o processo que está fazendo o envio, pois a mensagem ICMP retornada é obrigada a incluir o cabeçalho UDP do datagrama que causou o erro de ICMP.

Estabelecimento de um handler de sinal para `SIGALRM`

26 Estabelecemos nossa função `sig_alrm` como o handler de sinal para `SIGALRM`, pois, sempre que enviamos um datagrama UDP, esperamos durante três segundos por uma mensagem ICMP, antes de enviarmos a próxima sondagem.

traceroute/traceloop.c

```
 1 #include   "trace.h"

 2 void
 3 traceloop(void)
 4 {

 5     int    seq, code, done;
 6     double rtt;
 7     struct rec *rec;
 8     struct timeval tvrecv;

 9     recvfd = Socket(pr->sasend->sa_family, SOCK_RAW, pr->icmpproto);
10     setuid(getuid()); /* não precisa mais de permissões especiais */

11 #ifdef IPV6
12     if (pr->sasend->sa_family == AF_INET6 && verbose == 0) {
13         struct icmp6_filter myfilt;
14         ICMP6_FILTER_SETBLOCKALL(&myfilt);
15         ICMP6_FILTER_SETPASS(ICMP6_TIME_EXCEEDED, &myfilt);
16         ICMP6_FILTER_SETPASS(ICMP6_DST_UNREACH, &myfilt);
17         setsockopt(recvfd, IPPROTO_IPV6, ICMP6_FILTER,
18                    &myfilt, sizeof(myfilt));
19     }
20 #endif

21     sendfd = Socket(pr->sasend->sa_family, SOCK_DGRAM, 0);

22     pr->sabind->sa_family = pr->sasend->sa_family;
23     sport = (getpid() & 0xffff) | 0x8000;   /* nossa porta UDP de origem # */
24     sock_set_port(pr->sabind, pr->salen, htons(sport));
25     Bind(sendfd, pr->sabind, pr->salen);

26     sig_alrm(SIGALRM);

27     seq = 0;
28     done = 0;
29     for (ttl = 1; ttl <= max_ttl && done == 0; ttl++) {
30         Setsockopt(sendfd, pr->ttllevel, pr->ttloptname, &ttl, sizeof(int));
31         bzero(pr->salast, pr->salen);

32         printf("%2d ", ttl);
33         fflush(stdout);

34         for (probe = 0; probe < nprobes; probe++) {
35             rec = (struct rec *) sendbuf;
36             rec->rec_seq = ++seq;
37             rec->rec_ttl = ttl;
38             Gettimeofday(&rec->rec_tv, NULL);

39             sock_set_port(pr->sasend, pr->salen, htons(dport + seq));
40             Sendto(sendfd, sendbuf, datalen, 0, pr->sasend, pr->salen);

41             if ( (code = (*pr->recv) (seq, &tvrecv)) == -3)
42                 printf(" *"); /* tempo-limite, nenhuma resposta */
43             else {
44                 char   str[NI_MAXHOST];

45                 if (sock_cmp_addr(pr->sarecv, pr->salast, pr->salen) != 0) {
46                     if (getnameinfo(pr->sarecv, pr->salen, str, sizeof(str),
47                                     NULL, 0, 0) == 0)
48                         printf(" %s (%s)", str,
49                                Sock_ntop_host(pr->sarecv, pr->salen));
50                     else
51                         printf(" %s", Sock_ntop_host(pr->sarecv, pr->salen));
52                     memcpy(pr->salast, pr->sarecv, pr->salen);
```

Figura 28.19 Função `traceloop`: loop de processamento principal (*continua*).

```
53                }
54                tv_sub(&tvrecv, &rec->rec_tv);
55                rtt = tvrecv.tv_sec * 1000.0 + tvrecv.tv_usec / 1000.0;
56                printf(" %.3f ms", rtt);

57                if (code == -1) /* porta inacessível; no destino */
58                    done++;
59                else if (code >= 0)
60                    printf(" (ICMP %s)", (*pr->icmpcode) (code));
61            }
62            fflush(stdout);
63        }
64        printf("\n");
65    }
66 }
```

traceroute/traceloop.c

Figura 28.19 Função `traceloop`: loop de processamento principal (*continuação*).

Loop principal; configuração do TTL ou limite de hop e envio de três sondagens

27–38 O loop principal da função é um loop `for` duplamente aninhado. O loop externo inicia o TTL ou limite de hop em 1 e o aumenta por 1, enquanto o loop interno envia três sondagens (datagramas UDP) para o destino. Sempre que o TTL muda, chamamos `setsockopt` para configurar o novo valor utilizando a opção de soquete `IP_TTL` ou `IPV6_UNICAST_HOPS`.

Sempre que fazemos o loop externo, inicializamos em 0 a estrutura de endereço de soquete apontada por `salast`. Essa estrutura será comparada com a de endereço de soquete retornada por `recvfrom`, quando a mensagem ICMP for lida, e se elas forem diferentes, o endereço IP da nova estrutura será impresso. Utilizando-se essa técnica, o endereço IP correspondente à primeira sondagem de cada TTL é impresso e, se o endereço IP mudar para determinado valor de TTL (digamos, uma rota muda enquanto estamos executando o programa), então o novo endereço IP será impresso.

Configuração da porta de destino e envio do datagrama UDP

39–40 Sempre que um pacote de sondagem é enviado, a porta de destino na estrutura de endereço de soquete `sasend` é alterada, chamando nossa função `sock_set_port`. A razão para mudar a porta para cada sondagem é que, quando alcançamos o destino final, todas as três sondagens são enviadas para uma porta diferente, e espera-se que pelo menos uma delas não esteja em uso. `sendto` envia o datagrama UDP.

Leitura da mensagem ICMP

41–42 Uma de nossas funções, `recv_v4` ou `recv_v6`, chama `recvfrom` para ler e processar as mensagens ICMP retornadas. Essas duas funções retornam –3 se um tempo-limite ocorrer (nos dizendo para enviar outra sondagem, caso não tenhamos enviado três para esse TTL), –2 se um erro ICMP "tempo excedido em trânsito" for recebido, –1 se um erro ICMP "porta inacessível" for recebido (o que significa que alcançamos o destino final) ou o código ICMP não-negativo, se algum outro erro ICMP de destino inacessível for recebido.

Impressão da resposta

43–63 Conforme mencionamos anteriormente, se essa é a primeira resposta para determinado TTL, ou se o endereço IP do nó que está enviando a mensagem ICMP tiver mudado para esse TTL, imprimimos o nome de host e o endereço IP, ou apenas o endereço IP (caso a chamada a `getnameinfo` não retorne o nome de host). O RTT é calculado como a diferença de tempo desde quando enviamos a sondagem até o tempo em que a mensagem ICMP foi retornada e impressa.

Nossa função `recv_v4` é mostrada na Figura 28.20.

traceroute/recv_v4.c

```
 1 #include   "trace.h"
 2 extern int gotalarm;
 3 /*
 4  * Retorna: -3 no caso de tempo-limite
 5  *          -2 no caso de tempo de ICMP excedido em trânsito (o chamador continua)
 6  *          -1 no caso de porta ICMP inacessível (o chamador terminou)
 7  *       > = 0 o valor de retorno é algum outro código ICMP inacessível
 8  */
 9 int
10 recv_v4(int seq, struct timeval *tv)
11 {
12     int    hlen1, hlen2, icmplen, ret;
13     socklen_t len;
14     ssize_t n;
15     struct ip *ip, *hip;
16     struct icmp *icmp;
17     struct udphdr *udp;
18     gotalarm = 0;
19     alarm(3);
20     for ( ; ; ) {
21         if (gotalarm)
22             return (-3);          /* o alarme expirou */
23         len = pr->salen;
24         n = recvfrom(recvfd, recvbuf, sizeof(recvbuf), 0, pr->sarecv, &len);
25         if (n < 0) {
26             if (errno == EINTR)
27                 continue;
28             else
29                 err_sys("recvfrom error");
30         }
31         ip = (struct ip *) recvbuf; /* inicia do cabeçalho IP */
32         hlen1 = ip->ip_hl << 2; /* comprimento do cabeçalho IP */
33         icmp = (struct icmp *) (recvbuf + hlen1);   /* início do cabeçalho
                                                        ICMP */
34         if ( (icmplen = n - hlen1) < 8)
35             continue;       /* insuficiente para ver o cabeçalho ICMP */
36         if (icmp->icmp_type == ICMP_TIMXCEED &&
37             icmp->icmp_code == ICMP_TIMXCEED_INTRANS) {
38             if (icmplen < 8 + sizeof(struct ip))
39                 continue;    /* dados insuficientes para ver o IP interno */
40             hip = (struct ip *) (recvbuf + hlen1 + 8);
41             hlen2 = hip->ip_hl << 2;
42             if (icmplen < 8 + hlen2 + 4)
43                 continue;    /* dados insuficientes para ver portas UDP */
44             udp = (struct udphdr *) (recvbuf + hlen1 + 8 + hlen2);
45             if (hip->ip_p == IPPROTO_UDP &&
46                 udp->uh_sport == htons(sport) &&
47                 udp->uh_dport == htons(dport + seq)) {
48                 ret = -2;      /* atingimos um roteador intermediário */
49                 break;
50             }
```

Figura 28.20 Função `recv_v4`: lê e processa mensagens ICMPv4 (*continua*).

```
51          } else if (icmp->icmp_type == ICMP_UNREACH) {
52              if (icmplen < 8 + sizeof(struct ip))
53                  continue;  /* dados insuficientes para ver o IP interno */

54              hip = (struct ip *) (recvbuf + hlen1 + 8);
55              hlen2 = hip->ip_hl << 2;
56              if (icmplen < 8 + hlen2 + 4)
57                  continue;  /* dados insuficientes para ver portas UDP */

58              udp = (struct udphdr *) (recvbuf + hlen1 + 8 + hlen2);
59              if (hip->ip_p == IPPROTO_UDP &&
60                  udp->uh_sport == htons(sport) &&
61                  udp->uh_dport == htons(dport + seq)) {
62                  if (icmp->icmp_code == ICMP_UNREACH_PORT)
63                      ret = -1;   /* alcançou o destino */
64                  else
65                      ret = icmp->icmp_code; /* 0, 1, 2, ... */
66                  break;
67              }
68          }
69          if (verbose) {
70              printf(" (from %s: type = %d, code = %d)\n",
71                     Sock_ntop_host(pr->sarecv, pr->salen),
72                     icmp->icmp_type, icmp->icmp_code);
73          }
74          /* Algum outro erro de ICMP, recvfrom() novamente */
75      }
76      alarm(0);                   /* não deixa o alarme executando */
77      Gettimeofday(tv, NULL);     /* obtém o tempo de chegada do pacote */
78      return (ret);
79  }
```

traceroute/recv_v4.c

Figura 28.20 Função `recv_v4`: lê e processa mensagens ICMPv4 (*continuação*).

Configuração do alarme e leitura de cada mensagem ICMP

19–30 Um alarme é configurado para três segundos no futuro e a função entra em um loop que chama `recvfrom`, lendo cada mensagem ICMPv4 retornada no soquete bruto.

> Essa função evita a condição de corrida que descrevemos na Seção 20.5, utilizando um flag global.

Obtenção do ponteiro para o cabeçalho ICMP

31–35 `ip` aponta para o início do cabeçalho IPv4 (lembre-se de que uma leitura em um soquete bruto sempre retorna o cabeçalho IP) e `icmp` aponta para o início do cabeçalho ICMP. A Figura 28.21 mostra os vários cabeçalhos, ponteiros e comprimentos utilizados pelo código.

Processamento da mensagem ICMP de "tempo excedido em trânsito"

36–50 Se a mensagem ICMP é de "tempo excedido em trânsito", possivelmente se trata de uma resposta para uma de nossas sondagens. `hip` aponta para o cabeçalho IPv4 que é retornado na mensagem ICMP, após o cabeçalho ICMP de 8 bytes. `udp` aponta para o cabeçalho UDP que se segue. Se a mensagem ICMP foi gerada por um datagrama UDP e se as portas de origem e de destino desse datagrama são os valores que enviamos, então essa é uma resposta de um roteador intermediário para nossa sondagem.

Processamento da mensagem ICMP "porta inacessível"

51–68 Se a mensagem ICMP é "destino inacessível", então examinamos o cabeçalho UDP retornado na mensagem ICMP para ver se ela é uma resposta para nossa sondagem. Se assim for, e

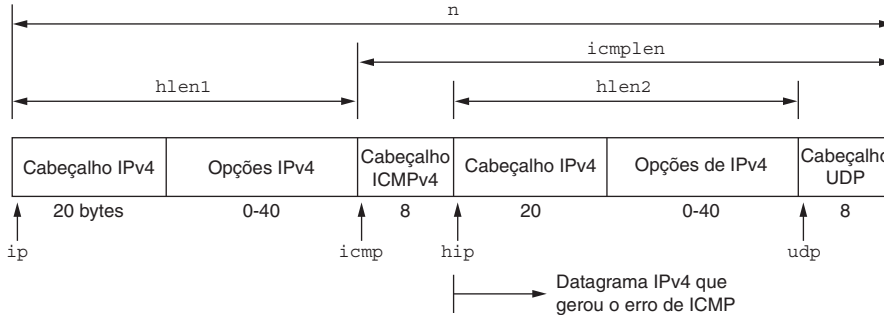


Figura 28.21 Cabeçalhos, ponteiros e comprimentos no processamento do erro ICMPv4.

se o código de ICMP for "porta inacessível", retornamos –1, pois alcançamos o destino final. Se a mensagem ICMP é de uma de nossas sondagens, mas não é uma "porta inacessível", então esse valor de código ICMP é retornado. Um exemplo comum disso é um firewall retornando algum outro código inacessível para o host de destino que estamos investigando.

Tratando outras mensagens ICMP

69–73 Todas as outras mensagens ICMP são impressas, se o flag –v foi especificado.

A próxima função, `recv_v6`, é mostrada na Figura 28.24 e é a equivalente IPv6 da função descrita anteriormente. Essa função é quase idêntica a `recv_v4`, exceto quanto aos diferentes nomes de constante e de membro de estrutura. Além disso, o cabeçalho IPv6 não faz parte dos dados recebidos em um soquete bruto IPv6; os dados iniciam com o cabeçalho ICMPv6. A Figura 28.22 mostra os vários cabeçalhos, ponteiros e comprimentos utilizados pelo código.

Definimos duas funções, `icmpcode_v4` e `icmpcode_v6`, que podem ser chamadas na parte inferior da função `traceloop` para imprimir uma string de descrição correspondente a um erro ICMP "destino inacessível". A Figura 28.25 mostra somente a função para IPv6. A função para IPv4 é semelhante, embora mais longa, pois há mais códigos ICMPv4 de "destino inacessível" (Figura A.15).

A função final em nosso programa `traceroute` é nosso handler `SIGALRM`, a função `sig_alrm` mostrada na Figura 28.23. Tudo que essa função faz é retornar, causando o retorno do erro `EINTR` de `recvfrom` em `recv_v4` ou `recv_v6`.

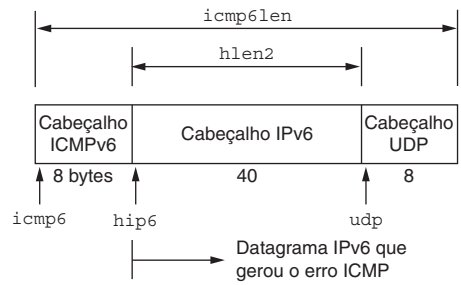


Figura 28.22 Cabeçalhos, ponteiros e comprimentos no processamento de erro ICMPv6.

traceroute/sig_alrm.c

```
1 #include   "trace.h"
2 int    gotalarm;
3 void
4 sig_alrm(int signo)
5 {
6     gotalarm = 1;   /* configura flag para anotar que o alarme ocorreu */
7     return;         /* e interrompe recvfrom () */
8 }
```

traceroute/sig_alrm.c

Figura 28.23 Função `sig_alrm`.

traceroute/recv_v6.c

```
 1 #include   "trace.h"
 2 extern int gotalarm;
 3 /*
 4  * Retorna: -3 no caso de tempo-limite
 5  *          -2 no caso de tempo de ICMP excedido em trânsito (o chamador continua)
 6  *          -1 no caso de porta ICMP inacessível (o chamador terminou)
 7  *       >= 0 o valor de retorno é algum outro código ICMP inacessível
 8  */
 9 int
10 recv_v6(int seq, struct timeval *tv)
11 {
12 #ifdef IPV6
13     int    hlen2, icmp6len, ret;
14     ssize_t n;
15     socklen_t len;
16     struct ip6_hdr *hip6;
17     struct icmp6_hdr *icmp6;
18     struct udphdr *udp;
19     gotalarm = 0;
20     alarm(3);
21     for ( ; ; ) {
22         if (gotalarm)
23             return (-3);          /* o alarme expirou */
24         len = pr->salen;
25         n = recvfrom(recvfd, recvbuf, sizeof(recvbuf), 0, pr->sarecv, &len);
26         if (n < 0) {
27             if (errno == EINTR)
28                 continue;
29             else
30                 err_sys("recvfrom error");
31         }
32         icmp6 = (struct icmp6_hdr *) recvbuf; /* cabeçalho ICMP */
33         if ( (icmp6len = n) < 8)
34             continue;             /* insuficiente para ver o cabeçalho ICMP */
35         if (icmp6->icmp6_type == ICMP6_TIME_EXCEEDED &&
36             icmp6->icmp6_code == ICMP6_TIME_EXCEED_TRANSIT) {
37             if (icmp6len < 8 + sizeof(struct ip6_hdr) + 4)
38                 continue;         /* dados insuficientes para ver o
                                     cabeçalho interno */
39             hip6 = (struct ip6_hdr *) (recvbuf + 8);
40             hlen2 = sizeof(struct ip6_hdr);
41             udp = (struct udphdr *) (recvbuf + 8 + hlen2);
42             if (hip6->ip6_nxt == IPPROTO_UDP &&
```

Figura 28.24 Função `recv_v6`: lê e processa mensagens ICMPv6 (*continua*).

```
43                udp->uh_sport == htons(sport) &&
44                udp->uh_dport == htons(dport + seq))
45                ret = -2;          /* atingimos um roteador intermediário */
46            break;
47        } else if (icmp6->icmp6_type == ICMP6_DST_UNREACH) {
48            if (icmp6len < 8 + sizeof(struct ip6_hdr) + 4)
49                continue;          /* dados insuficientes p/ ver o
                                        cabeçalho interno */
50            hip6 = (struct ip6_hdr *) (recvbuf + 8);
51            hlen2 = sizeof(struct ip6_hdr);
52            udp = (struct udphdr *) (recvbuf + 8 + hlen2);
53            if (hip6->ip6_nxt == IPPROTO_UDP &&
54                udp->uh_sport == htons(sport) &&
55                udp->uh_dport == htons(dport + seq)) {
56                if (icmp6->icmp6_code == ICMP6_DST_UNREACH_NOPORT)
57                    ret = -1; /* alcançou o destino */
58                else
59                    ret = icmp6->icmp6_code; /* 0, 1, 2, ... */
60                break;
61            }
62        } else if (verbose) {
63            printf(" (from %s: type = %d, code = %d)\n",
64                   Sock_ntop_host(pr->sarecv, pr->salen),
65                   icmp6->icmp6_type, icmp6->icmp6_code);
66        }
67        /* Algum outro erro de ICMP, recvfrom() novamente */
68    }
69    alarm(0);                      /* não deixa o alarme executando */
70    Gettimeofday(tv, NULL);        /* obtém o tempo de chegada do pacote */
71    return (ret);
72 #endif
73 }
```

traceroute/recv_v6.c

Figura 28.24 Função `recv_v6`: lê e processa mensagens ICMPv6 (*continuação*).

traceroute/icmpcode_v6.c

```
 1 #include   "trace.h"
 2 const char *
 3 icmpcode_v6(int code)
 4 {
 5 #ifdef IPV6
 6     static char errbuf[100];
 7     switch (code) {
 8     case ICMP6_DST_UNREACH_NOROUTE:
 9         return ("no route to host");
10     case ICMP6_DST_UNREACH_ADMIN:
11         return ("administratively prohibited");
12     case ICMP6_DST_UNREACH_NOTNEIGHBOR:
13         return ("not a neighbor");
14     case ICMP6_DST_UNREACH_ADDR:
15         return ("address unreachable");
16     case ICMP6_DST_UNREACH_NOPORT:
17         return ("port unreachable");
18     default:
19         sprintf(errbuf, "[unknown code %d]", code);
20         return errbuf;
21     }
22 #endif
23 }
```

traceroute/icmpcode_v6.c

Figura 28.25 Retorna a string correspondente a um código inacessível ICMPv6.

Exemplo

Primeiro mostramos um exemplo usando IPv4.

```
freebsd % traceroute www.unpbook.com
traceroute to www.unpbook.com (206.168.112.219): 30 hops max, 24 data bytes
 1 12.106.32.1 (12.106.32.1) 0.799 ms 0.719 ms 0.540 ms
 2 12.124.47.113 (12.124.47.113) 1.758 ms 1.760 ms 1.839 ms
 3 gbr2-p27.sffca.ip.att.net (12.123.195.38) 2.744 ms 2.575 ms 2.648 ms
 4 tbr2-p012701.sffca.ip.att.net (12.122.11.85) 3.770 ms 3.689 ms 3.848 ms
 5 gbr3-p50.dvmco.ip.att.net (12.122.2.66) 26.202 ms 26.242 ms 26.102 ms
 6 gbr2-p20.dvmco.ip.att.net (12.122.5.26) 26.255 ms 26.194 ms 26.470 ms
 7 gar2-p370.dvmco.ip.att.net (12.123.36.141) 26.443 ms 26.310 ms 26.427 ms
 8 att-46.den.internap.ip.att.net (12.124.158.58) 26.962 ms 27.130 ms
                                                                  27.279 ms
 9 border10.ge3-0-bbnet2.den.pnap.net (216.52.40.79) 27.285 ms 27.293 ms
                                                                  26.860 ms
10 coop-2.border10.den.pnap.net (216.52.42.118) 28.721 ms 28.991 ms
                                                                  30.077 ms
11 199.45.130.33 (199.45.130.33) 29.095 ms 29.055 ms 29.378 ms
12 border-to-141-netrack.boulder.co.coop.net (207.174.144.178) 30.875 ms
                                                        29.747 ms 30.142 ms
13 linux.unpbook.com (206.168.112.219) 31.713 ms 31.573 ms 33.952 ms
```

Colocamos quebra de linha nas linhas longas para obtermos uma saída mais legível.

Eis um exemplo usando IPv6.

```
freebsd % traceroute www.kame.net
traceroute to orange.kame.net (2001:200:0:4819:203:47ff:fea5:3085):
            30 hops max, 24 data bytes
 1 3ffe:b80:3:9ad1::1 (3ffe:b80:3:9ad1::1) 107.437 ms 99.341 ms 103.477 ms
 2 Viagenie-gw.int.ipv6.ascc.net (2001:288:3b0::55)
            105.129 ms 89.418 ms 90.016 ms
 3 gw-Viagenie.int.ipv6.ascc.net (2001:288:3b0::54)
            302.300 ms 291.580 ms 289.839 ms
 4 c7513-gw.int.ipv6.ascc.net (2001:288:3b0::c)
            296.088 ms 298.600 ms 292.196 ms
 5 m160-c7513.int.ipv6.ascc.net (2001:288:3b0::1e)
            296.266 ms 314.878 ms 302.429 ms
 6 m20jp-m160tw.int.ipv6.ascc.net (2001:288:3b0::1b)
            327.637 ms 326.897 ms 347.062 ms
 7 hitachi1.otemachi.wide.ad.jp (2001:200:0:1800::9c4:2)
            420.140 ms 426.592 ms 422.756 ms
 8 pc3.yagami.wide.ad.jp (2001:200:0:1c04::1000:2000)
            415.471 ms 418.308 ms 461.654 ms
 9 gr2000.k2c.wide.ad.jp (2001:200:0:8002::2000:1)
            416.581 ms 422.430 ms 427.692 ms
10 2001:200:0:4819:203:47ff:fea5:3085 (2001:200:0:4819:203:47ff:fea5:3085)
            417.169 ms 434.674 ms 424.037 ms
```

Colocamos quebra de linha nas linhas longas para obtermos uma saída mais legível.

28.7 Um daemon de mensagem ICMP

Receber erros assíncronos de ICMP em um soquete UDP foi e continua a ser um problema. Os erros de ICMP são recebidos pelo kernel, mas raramente são entregues à aplicação que precisa conhecê-los. Na API de soquetes, vimos que é necessário conectar o soquete UDP a um endereço IP para se receberem esses erros (Seção 8.11). A razão dessa limitação é que o único erro retornado de `recvfrom` é um código `errno` inteiro e, se a aplicação envia datagramas para vários destinos e, então, chama `recvfrom`, essa função não pode informar a ela qual datagrama encontrou um erro.

Nesta seção, forneceremos uma solução que não exige quaisquer alterações do kernel. Forneceremos um daemon de mensagem ICMP, `icmpd`, que cria dois soquetes brutos: um ICMPv4 e outro ICMPv6, e recebe todas as mensagens de ICMP que o kernel passa para eles. Ele também cria um soquete de fluxo de domínio Unix, o vincula (`bind`) ao nome de caminho `/tmp/icmpd` e espera por conexões de cliente nesse nome de caminho. Mostraremos isso na Figura 28.26.

Uma aplicação UDP (que é um cliente do daemon) primeiro cria seu soquete UDP, o soquete no qual ela quer receber erros assíncronos. A aplicação deve vincular-se (`bind`) a uma porta efêmera nesse soquete, por razões que discutiremos posteriormente. Então, ela cria um soquete de domínio Unix e se conecta ao nome de caminho bem-conhecido desse daemon. Mostraremos isso na Figura 28.27.

Em seguida, a aplicação "passa" seu soquete UDP para o daemon, através da conexão de domínio Unix, utilizando *passagem de descritor*, conforme descrevemos na Seção 15.7. Isso fornece ao daemon uma cópia do soquete, de modo que ele pode chamar `getsockname` e obter o número da porta vinculada ao soquete. Mostraremos essa passagem do soquete na Figura 28.28.

Depois que o daemon obtém o número da porta vinculada ao soquete UDP, ele fecha sua cópia do soquete, levando-nos de volta ao arranjo mostrado na Figura 28.27.

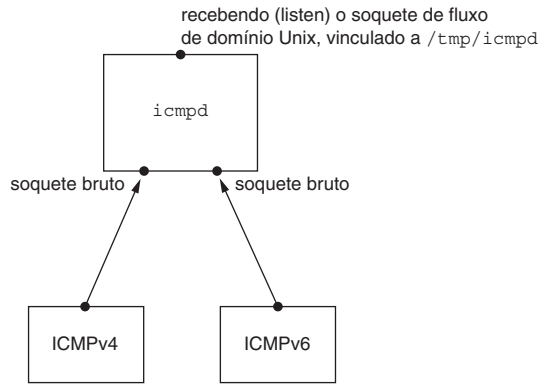


Figura 28.26 Daemon `icmpd`: soquetes iniciais criados.

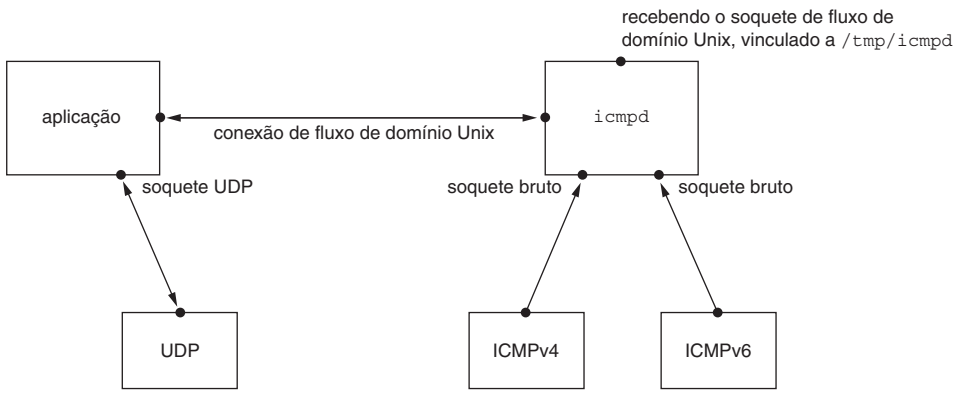


Figura 28.27 A aplicação cria seu soquete UDP e uma conexão de domínio Unix com o daemon.

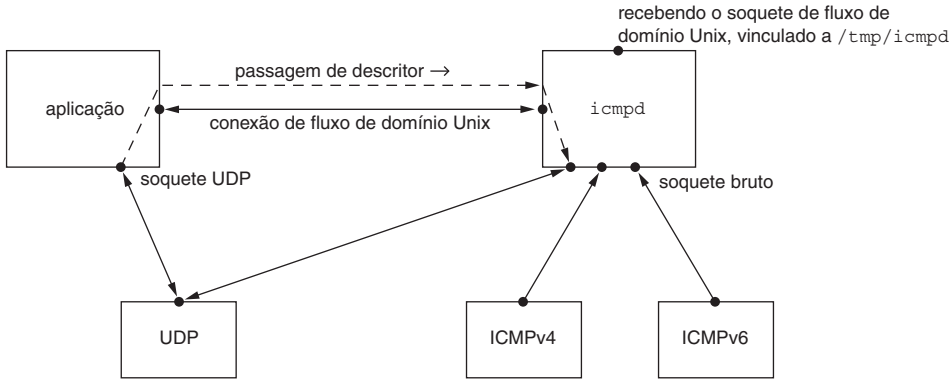


Figura 28.28 Passando o soquete UDP para o daemon através da conexão de domínio Unix.

> Se o host suportasse passagem de credencial (Seção 15.8), a aplicação também poderia enviar suas credenciais para o daemon. Este, então, poderia verificar se esse usuário deve ter acesso a tal facilidade.

Desse ponto em diante, todos os erros de ICMP que o daemon recebe em resposta aos datagramas UDP enviados da porta vinculada ao soquete UDP da aplicação fazem com que o daemon envie uma mensagem (que descreveremos em breve) através do soquete de domínio Unix para a aplicação. Esta, portanto, deve utilizar `select` ou `poll`, esperando dados no soquete UDP ou no soquete de domínio Unix.

Agora, veremos o código-fonte de uma aplicação que utiliza esse daemon e, então, o próprio daemon. Iniciamos com a Figura 28.29, nosso cabeçalho que é incluído pela aplicação e pelo daemon.

icmpd/unpicmpd.h

```
 1 #ifndef __unpicmp_h
 2 #define __unpicmp_h

 3 #include   "unp.h"

 4 #define ICMPD_PATH      "/tmp/icmpd"      /* nome de caminho bem-conhecido
                                              do servidor */

 5 struct icmpd_err {
 6     int    icmpd_errno;        /* EHOSTUNREACH, EMSGSIZE, ECONNREFUSED */
 7     char   icmpd_type;         /* tipo ICMPv[46] real */
 8     char   icmpd_code;         /* código ICMPv[46] real */
 9     socklen_t icmpd_len;       /* comprimento de sockaddr{} que segue */
10     struct sockaddr_storage icmpd_dest;  /* sockaddr_storage trata
                                              qualquer tamanho */
11 };

12 #endif /* __unpicmp_h */
```

icmpd/unpicmpd.h

Figura 28.29 Cabeçalho `unpicmpd.h`.

4–11 Definimos o nome de caminho bem-conhecido do servidor e a estrutura `icmpd_err` que é passada do servidor para a aplicação sempre que é recebida uma mensagem ICMP que deve ser passada para essa aplicação.

6–8 Um problema é que os tipos de mensagem ICMPv4 diferem numericamente (e, às vezes, conceitualmente) dos tipos de mensagem ICMPv6 (Figuras A.15 e A.16). Os valores de *tipo* e *código* ICMP reais são retornados, mas também os mapeamos em um valor de `errno` (`icm-`

pd_errno), de maneira semelhante às colunas finais das Figuras A.15 e A. 16. A aplicação pode lidar com esse valor, em vez dos valores ICMPv4 ou ICMPv6 dependentes de protocolo. A Figura 28.30 mostra as mensagens de ICMP que são tratadas, além de seu mapeamento em um valor errno.

O daemon retorna cinco tipos de erros de ICMP.

- "Porta inacessível", indicando que nenhum soquete está vinculado à porta de destino no endereço IP de destino.
- "Pacote grande demais", que é utilizado com a descoberta do MTU do caminho. Atualmente, não há nenhuma API definida para permitir a uma aplicação UDP realizar a descoberta do MTU do caminho. O que freqüentemente acontece em kernels que suportam a descoberta do MTU do caminho para UDP é que o recebimento desse erro de ICMP faz com que o kernel, em sua tabela de roteamento, registre o novo valor do MTU do caminho, mas a aplicação UDP que enviou o datagrama que foi descartado não é notificada. Em vez disso, ela deve exceder o tempo-limite e retransmitir o datagrama, assim o kernel encontrará o novo (e menor) MTU em sua tabela de roteamento e, então, fragmentará o datagrama. Passar esse erro de volta para a aplicação permite que ela retransmita mais cedo e, talvez, reduza o tamanho dos datagramas que envia.
- O erro "tempo excedido" é visto normalmente com um código igual a 0, indicando que o TTL do IPv4 ou o limite de hop do IPv6 chegaram a 0. Isso costuma indicar um loop de roteamento, que poderia ser um erro transitório.
- "Extinções de origem" do ICMPv4, embora tornadas obsoletas pela RFC 1812 (Backer, 1995), podem ser enviadas por roteadores (ou por hosts mal-configurados, atuando como roteadores). Elas indicam que um pacote foi descartado e, portanto, nós as tratamos como uma mensagem de "destino inacessível". Observe que o IPv6 não tem um erro de "extinção de origem".
- Todas as outras mensagens de destino inacessível indicam que um pacote foi descartado.

10 O membro icmpd_dest é uma estrutura de endereço de soquete contendo o endereço IP de destino e a porta do datagrama que gerou o erro de ICMP. Esse membro será uma estrutura sockaddr_in para IPv4 ou sockaddr_in6 para IPv6. Se a aplicação está enviando datagramas para vários destinos, ela provavelmente tem uma estrutura de endereço de soquete por destino. Retornando essas informações em uma estrutura de endereço de soquete, a aplicação pode compará-la com suas próprias estruturas para localizar a que causou o erro. Trata-se de uma estrutura sockaddr_storage para permitir o armazenamento de qualquer tipo de sockaddr que o sistema suporte.

Cliente de Eco UDP que utiliza nosso daemon icmpd

Agora, modificaremos nosso cliente de eco UDP, a função dg_cli, para utilizar nosso daemon icmpd. A Figura 28.31 mostra a primeira metade da função.

2–3 Os argumentos da função são iguais a todas as versões anteriores dessa função.

icmpd_errno	Erro ICMPv4	Erro ICMPv6
ECONNREFUSED	porta inacessível	porta inacessível
EMSGSIZE	fragmentação necessária, mas DF configurado	pacote grande demais
EHOSTUNREACH	tempo excedido	tempo excedido
EHOSTUNREACH	extinção da origem	
EHOSTUNREACH	Todos os outros destinos inacessíveis	Todos os outros destinos inacessíveis

Figura 28.30 Mapeamento de icmpd_errno de erros ICMPv4 e ICMPv6.

icmpd/dgcli01.c

```
 1 #include   "unpicmpd.h"

 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int    icmpfd, maxfdp1;
 6     char   sendline[MAXLINE], recvline[MAXLINE + 1];
 7     fd_set rset;
 8     ssize_t n;
 9     struct timeval tv;
10     struct icmpd_err icmpd_err;
11     struct sockaddr_un sun;

12     Sock_bind_wild(sockfd, pservaddr->sa_family);

13     icmpfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
14     sun.sun_family = AF_LOCAL;
15     strcpy(sun.sun_path, ICMPD_PATH);
16     Connect(icmpfd, (SA *) &sun, sizeof(sun));
17     Write_fd(icmpfd, "1", 1, sockfd);
18     n = Read(icmpfd, recvline, 1);
19     if (n != 1 || recvline[0] != '1')
20         err_quit("error creating icmp socket, n = %d, char = %c",
21                  n, recvline[0]);

22     FD_ZERO(&rset);
23     maxfdp1 = max(sockfd, icmpfd) + 1;
```

icmpd/dgcli01.c

Figura 28.31 Primeira metade da aplicação `dg_cli`.

Endereço curinga para `bind` e porta efêmera

12 Chamamos nossa função `sock_bind_wild` para vincular (`bind`) o endereço IP curinga e uma porta efêmera ao soquete UDP. Fazemos isso para que a cópia desse soquete, que passamos para o daemon, tenha vínculo com uma porta, pois o daemon precisa conhecer essa porta.

> O daemon também poderia fazer esse vínculo se uma porta local ainda não estivesse vinculada ao soquete que ele recebe, mas isso não funciona em todos os ambientes. Certas implementações de SVR4, como o Solaris 2.5, nas quais os soquetes não fazem parte do kernel, têm um bug quando um processo vincula uma porta a um soquete compartilhado; o outro processo com uma cópia desse soquete recebe erros estranhos quando tenta utilizar o soquete. A solução fácil é exigir que a aplicação se vincule à porta local, antes de passar o soquete para o daemon.

Estabelecimento da conexão do domínio Unix com o daemon

13–16 Criamos um soquete `AF_LOCAL` e o conectamos ao nome de caminho bem-conhecido do daemon.

Envio do soquete UDP para o daemon, espera pela resposta do daemon

17–21 Chamamos nossa função `write_fd`, da Figura 15.13, para enviar uma cópia de nosso soquete UDP para o daemon. Também enviamos um único byte de dados, o caractere "1", porque algumas implementações não gostam de passar um descritor sem dados. O daemon envia de volta um único byte de dados, consistindo no caractere "1", para indicar sucesso. Qualquer outra resposta indica um erro.

22–23 Inicializamos um conjunto de descritores e calculamos o primeiro argumento de `select` (o máximo dos dois descritores, mais um).

A última metade do nosso cliente é mostrada na Figura 28.32. Esse é o loop que lê uma linha da entrada-padrão, a envia para o servidor, lê de volta a resposta deste e escreve essa resposta na saída-padrão.

icmpd/dgcli01.c

```
24     while (Fgets(sendline, MAXLINE, fp) != NULL) {
25         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

26         tv.tv_sec = 5;
27         tv.tv_usec = 0;
28         FD_SET(sockfd, &rset);
29         FD_SET(icmpfd, &rset);
30         if ( (n = Select(maxfdp1, &rset, NULL, NULL, &tv)) == 0) {
31             fprintf(stderr, "socket timeout\n");
32             continue;
33         }

34         if (FD_ISSET(sockfd, &rset)) {
35             n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
36             recvline[n] = 0;      /* termina com nulo */
37             Fputs(recvline, stdout);
38         }

39         if (FD_ISSET(icmpfd, &rset)) {
40             if ( (n = Read(icmpfd, &icmpd_err, sizeof(icmpd_err))) == 0)
41                 err_quit("ICMP daemon terminated");
42             else if (n != sizeof(icmpd_err))
43                 err_quit("n = %d, expected %d", n, sizeof(icmpd_err));
44             printf("ICMP error: dest = %s, %s, type = %d, code = %d\n",
45                    Sock_ntop(&icmpd_err.icmpd_dest, icmpd_err.icmpd_len),
46                    strerror(icmpd_err.icmpd_errno),
47                    icmpd_err.icmpd_type, icmpd_err.icmpd_code);
48         }
49     }
50 }
```

icmpd/dgcli01.c

Figura 28.32 Última metade da aplicação `dg_cli`.

Chamada a `select`

26–33 Como estamos chamando `select`, podemos facilmente colocar um tempo-limite em nossa espera para a resposta do servidor de eco. Configuramos isso como cinco segundos, ativamos os dois descritores para legibilidade e chamamos `select`. Se um tempo-limite ocorrer, imprimimos uma mensagem e voltamos para o início do loop.

Impressão da resposta do servidor

34–38 Se um datagrama é retornado pelo servidor, o imprimimos na saída-padrão.

Tratamento do erro de ICMP

39–48 Se nossa conexão de domínio Unix com o daemon `icmpd` é legível, tentamos ler uma estrutura `icmpd_err`. Se isso tiver sucesso, imprimimos as informações relevantes retornadas pelo daemon.

> `strerror` é um exemplo de uma função simples, quase trivial, que deveria ser mais portável do que é. Primeiro, a linguagem C ANSI não diz nada sobre um retorno de erro da função. A página man do Solaris diz que a função retorna um ponteiro nulo, se o argumento está fora do intervalo. Mas isso significa que um código como
>
> ```
> printf("%s", strerror(arg));
> ```

está incorreto, porque `strerror` pode retornar um ponteiro nulo. Mas a implementação FreeBSD, junto com todas as implementações de código-fonte que os autores puderam encontrar, tratam um argumento inválido retornando um ponteiro para uma string como, por exemplo, "Erro desconhecido". Isso faz sentido e significa que o código acima está correto. Mas o POSIX muda isso e diz que, como nenhum valor de retorno está reservado para indicar um erro, se o argumento estiver fora do intervalo, a função configurará `errno` como `EINVAL`. (O POSIX não diz nada sobre o ponteiro retornado no caso de um erro.) Isso significa que um código adaptado deve configurar `errno` como 0, chamar `strerror`, testar se `errno` é igual a `EINVAL` e imprimir alguma outra mensagem no caso de um erro.

Exemplos de Cliente de Eco de UDP

Agora, mostraremos alguns exemplos desse cliente, antes de vermos o código-fonte do daemon. Primeiro, enviamos datagramas para um endereço IP que não está conectado à Internet.

```
freebsd % udpcli01 192.0.2.5 echo
hi there
socket timeout
and hello
socket timeout
```

Supomos que `icmpd` esteja em execução e esperamos que os erros de ICMP de "host inacessível" sejam retornados por algum roteador, mas nenhum é recebido. Em vez disso, nossa aplicação excede o tempo-limite. Mostramos isso para reiterar que um tempo-limite ainda é exigido e que a geração de mensagens de ICMP, como "host inacessível", não pode ocorrer.

Nosso próximo exemplo envia um datagrama para o servidor de eco-padrão em um host que não está executando o servidor. Recebemos um erro ICMPv4 de "porta inacessível", conforme o esperado.

```
freebsd % udpcli01 aix-4 echo
hello, world
ICMP error: dest = 192.168.42.2:7, Connection refused, type = 3, code = 3
```

Tentamos novamente com IPv6 e recebemos um erro ICMPv6 de "porta inacessível", conforme o esperado.

```
freebsd % udpcli01 aix-6 echo
hello, world
ICMP error: dest = [3ffe:b80:1f8d:2:204:acff:fe17:bf38]:7,
                                    Connection refused, type = 1, code = 4
```

Colocamos quebra de linha na linha longa para legibilidade.

Daemon `icmpd`

Iniciamos a descrição de nosso daemon `icmpd` com o cabeçalho `icmpd.h`, mostrado na Figura 28.33.

Array `client`

2-17 Como o daemon pode tratar qualquer número de clientes, utilizamos um array de estruturas `client` para manter as informações sobre cada cliente. Isso é semelhante às estruturas de dados que utilizamos na Seção 6.8. Além do descritor da conexão de domínio Unix com o cliente, também armazenamos a família de endereço do soquete UDP do cliente (`AF_INET` ou `AF_INET6`) e o número da porta vinculada a esse soquete. Também declaramos os protótipos de função e as variáveis globais compartilhadas por essas funções.

icmpd/icmpd.h

```
 1 #include   "unpicmpd.h"

 2 struct client {
 3     int    connfd;   /* Soquete de fluxo de domínio Unix para o cliente */
 4     int    family;   /* AF_INET ou AF_INET6 */
 5     int    lport;    /* porta local vinculada ao soquete UDP do cliente */
 6     /* ordenado por byte de rede */
 7 } client[FD_SETSIZE];

 8                           /*globais */
 9 int    fd4, fd6, listenfd, maxi, maxfd, nready;
10 fd_set rset, allset;
11 struct sockaddr_un cliaddr;

12                   /* protótipos de função */
13 int    readable_conn(int);
14 int    readable_listen(void);
15 int    readable_v4(void);
16 int    readable_v6(void);
```

icmpd/icmpd.h

Figura 28.33 Cabeçalho `icmpd.h` do daemon `icmpd`.

A Figura 28.34 mostra a primeira metade da função `main`.

icmpd/icmpd.c

```
 1 #include   "icmpd.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    i, sockfd;
 6     struct sockaddr_un sun;

 7     if(argc != 1)
 8        err_quit("usage: icmpd");

 9     maxi = -1;                    /* índice para o array client[] */
10     for (i = 0; i < FD_SETSIZE; i++)
11         client[i].connfd = -1;    /* -1 indica entrada disponível */
12     FD_ZERO(&allset);

13     fd4 = Socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
14     FD_SET(fd4, &allset);
15     maxfd = fd4;

16 #ifdef IPV6
17     fd6 = Socket(AF_INET6, SOCK_RAW, IPPROTO_ICMPV6);
18     FD_SET(fd6, &allset);
19     maxfd = max(maxfd, fd6);
20 #endif

21     listenfd = Socket(AF_UNIX, SOCK_STREAM, 0);
22     sun.sun_family = AF_LOCAL;
23     strcpy(sun.sun_path, ICMPD_PATH);
24     unlink(ICMPD_PATH);
25     Bind(listenfd, (SA *) &sun, sizeof(sun));
26     Listen(listenfd, LISTENQ);
27     FD_SET(listenfd, &allset);
28     maxfd = max(maxfd, listenfd);
```

icmpd/icmpd.c

Figura 28.34 Primeira metade da função `main`: cria soquetes.

Inicialização do array `client`

9–10 O array `client` é inicializado configurando o membro de soquete conectado como –1.

Criação de soquetes

12–28 Três soquetes são criados: um soquete ICMPv4 bruto, um soquete ICMPv6 bruto e um soquete de fluxo de domínio Unix. Desfazemos a conexão (`unlink`) de qualquer instância anteriormente existente do soquete de domínio Unix, vinculamos (`bind`) seu nome de caminho bem-conhecido ao soquete e chamamos `listen`. Esse é o soquete ao qual os clientes se conectam através de `connect`. O descritor máximo também é calculado para `select` e uma estrutura de endereço de soquete é alocada para chamadas a `accept`.

A Figura 28.35 mostra a segunda metade da função `main`, que é um loop infinito que chama `select`, esperando que qualquer um dos descritores do daemon seja legível.

icmpd/icmpd.c

```
29      for ( ; ; ) {
30          rset = allset;
31          nready = Select(maxfd + 1, &rset, NULL, NULL, NULL);

32          if (FD_ISSET(listenfd, &rset))
33              if (readable_listen() <= 0)
34                  continue;

35          if (FD_ISSET(fd4, &rset))
36              if (readable_v4() <= 0)
37                  continue;

38  #ifdef IPV6
39          if (FD_ISSET(fd6, &rset))
40              if (readable_v6() <= 0)
41                  continue;
42  #endif

43          for (i = 0; i <= maxi; i++) { /* verifica dados de todos os clientes */
44              if ( (sockfd = client[i].connfd) < 0)
45                  continue;
46              if (FD_ISSET(sockfd, &rset))
47                  if (readable_conn(i) <= 0)
48                      break;          /* mais nenhum descritor legível */
49          }
50      }
51      exit(0);
52  }
```

icmpd/icmpd.c

Figura 28.35 Segunda metade da função `main`: trata de descritor legível.

Verificação do soquete ouvinte de domínio Unix

32–34 O soquete ouvinte de domínio Unix é testado primeiro e, se estiver pronto, `readable_listen` é chamado. A variável `nready`, o número de descritores que `select` retorna como legíveis, é uma variável global. Cada uma de nossas funções `readable_`*XXX* decrementa essa variável e retorna seu novo valor como o valor de retorno da função. Quando esse valor chega a 0, todos os descritores legíveis foram processados e `select` é chamado novamente.

Verificação dos soquetes ICMP brutos

35–42 Ambos os soquetes brutos ICMPv4 e ICMPv6 são testados, nessa ordem.

Verificação dos soquetes de domínio Unix conectados

43–49 Em seguida, verificamos se um dos soquetes de domínio Unix conectados é legível. A legibilidade em qualquer um desses soquetes significa que o cliente ou enviou um descritor ou terminou.

A Figura 28.36 mostra a função `readable_listen`, chamada quando o soquete ouvinte do daemon é legível. Isso indica uma nova conexão de cliente.

7–25 A conexão é aceita e a primeira entrada disponível no array `client` é utilizada. O código dessa função foi copiado do início da Figura 6.22. Se uma entrada não pudesse ser encontrada no array de cliente, simplesmente fecharíamos a nova conexão de cliente e continuaríamos a atender nossos clientes atuais.

icmpd/readable_listen.c

```
 1 #include   "icmpd.h"

 2 int
 3 readable_listen(void)
 4 {
 5     int    i, connfd;
 6     socklen_t clilen;

 7     clilen = sizeof(cliaddr);
 8     connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

 9         /*localiza a primeira estrutura client[] disponível */
10     for (i = 0; i < FD_SETSIZE; i++)
11         if (client[i].connfd < 0) {
12             client[i].connfd = connfd; /* salva o descritor */
13             break;
14         }
15     if (i == FD_SETSIZE) {
16         close(connfd);            /* não pode tratar o novo cliente, */
17         return (--nready);        /* fecha a nova conexão de forma rude */
18     }
19     printf("new connection, i = %d, connfd = %d\n", i, connfd);

20     FD_SET(connfd, &allset);      /* adiciona novo descritor a configurar */
21     if (connfd > maxfd)
22         maxfd = connfd;           /* para select() */
23     if (i > maxi)
24         maxi = i;                 /* índice máximo no array client[] */

25     return (--nready);
26 }
```

icmpd/readable_listen.c

Figura 28.36 Tratando as novas conexões de cliente.

Quando um soquete conectado está legível, nossa função `readable_conn` é chamada (Figura 28.37). Seu argumento é o índice desse cliente no array `client`.

icmpd/readable_conn.c

```
 1 #include   "icmpd.h"

 2 int
 3 readable_conn(int i)
 4 {
 5     int    unixfd, recvfd;
 6     char   c;
 7     ssize_t n;
```

Figura 28.37 Leitura dos dados e um possível descritor do cliente (*continua*).

```
 8      socklen_t len;
 9      struct sockaddr_storage ss;

10      unixfd = client[i].connfd;
11      recvfd = -1;
12      if ( (n = Read_fd(unixfd, &c, 1, &recvfd)) == 0) {
13          err_msg("client %d terminated, recvfd = %d", i, recvfd);
14          goto clientdone;          /* o cliente provavelmente terminou */
15      }

16          /* dados de cliente; deve ser descritor */
17      if (recvfd < 0) {
18          err_msg("read_fd did not return descriptor");
19          goto clienterr;
20  }
```

icmpd/readable_conn.c

Figura 28.37 Leitura dos dados e um possível descritor do cliente (*continuação*).

Leitura dos dados do cliente e possivelmente um descritor

13–18 Chamamos nossa função `read_fd`, da Figura 15.11, para ler os dados e, possivelmente, um descritor. Se o valor de retorno é 0, o cliente fechou sua extremidade da conexão, possivelmente terminando.

> Uma decisão de projeto era se deveríamos utilizar um soquete de fluxo de domínio Unix entre a aplicação e o daemon ou um soquete de datagrama de domínio Unix. O soquete UDP da aplicação pode ser passado por meio de qualquer tipo de soquete de domínio Unix. O motivo de utilizarmos um soquete de fluxo foi para detectar quando um cliente terminou. Quando um cliente termina, todos os seus descritores são fechados automaticamente, incluindo sua conexão de domínio Unix com o daemon, que diz ao daemon para que remova esse cliente do array `client`. Se tivéssemos utilizado um soquete de datagrama, não saberíamos quando o cliente terminou.

16–20 Se o cliente não fechou a conexão, então esperamos um descritor.

A segunda metade da nossa função `readable_conn` é mostrada na Figura 28.38.

Obtenção do número da porta vinculada ao soquete UDP

21–25 `getsockname` é chamada para que o daemon possa obter o número da porta vinculada ao soquete. Como não sabemos o tamanho do buffer a alocar para a estrutura de endereço de soquete, utilizamos uma estrutura `sockaddr_storage`, que é suficientemente grande e apropriadamente alinhada para armazenar qualquer estrutura de endereço de soquete que o sistema suporte.

26–33 A família de endereços do soquete é armazenada na estrutura `client`, junto com o número da porta. Se o número da porta é 0, chamamos nossa função `sock_bind_wild` para vincular o endereço de curinga e uma porta efêmera com o soquete, mas, conforme mencionamos anteriormente, isso não funciona em algumas implementações de SVR4.

icmpd/readable_conn.c

```
21      len = sizeof(ss);
22      if (getsockname(recvfd, (SA *) &ss, &len) < 0) {
23          err_ret("getsockname error");
24          goto clienterr;
25      }

26      client[i].family = ss.ss_family;
27      if ((client[i].lport = sock_get_port((SA *) &ss, len)) == 0) {
28          client[i].lport = sock_bind_wild(recvfd, client[i].family);
29          if (client[i].lport <= 0) {
```

Figura 28.38 Obtenção do número da porta que o cliente vinculou ao seu soquete UDP (*continua*).

```
30            err_ret("error binding ephemeral port");
31            goto clienterr;
32        }
33    }
34    Write(unixfd, "1", 1);  /* diz ao cliente que tudo está OK */
35    Close(recvfd);          /* tudo pronto com o soquete UDP do cliente */
36    return (--nready);
37  clienterr:
38    Write(unixfd, "0", 1);  /* diz ao cliente que um erro ocorreu */
39  clientdone:
40    Close(unixfd);
41    if (recvfd >= 0)
42        Close(recvfd);
43    FD_CLR(unixfd, &allset);
44    client[i].connfd = -1;
45    return (--nready);
46 }
```

icmpd/readable_conn.c

Figura 28.38 Obtenção do número da porta que o cliente vinculou ao seu soquete UDP (*continuação*).

Indicando sucesso para o cliente

34 Um byte consistindo no caractere "1" é enviado de volta para o cliente.

Fechando `(close)` o soquete UDP do cliente

35 Terminamos com o soquete UDP do cliente e o fechamos. Esse descritor foi passado para nós pelo cliente e, portanto, é uma cópia; então, o soquete UDP está ainda aberto no cliente.

Tratando de erros e da terminação do cliente

37–45 Se um erro ocorre, um byte "0" é gravado de volta no cliente. Quando o cliente termina, nossa extremidade da conexão de domínio Unix é fechada e o descritor é removido do conjunto de descritores de `select`. O membro `connfd` da estrutura `client` é configurado como –1, indicando que está disponível.

Nossa função `readable_v4` é chamada quando o soquete ICMPv4 bruto é legível. Mostramos a primeira metade na Figura 28.39. Esse código é semelhante ao de ICMPv4 mostrado anteriormente nas Figuras 28.8 e 28.20.

icmpd/readable_v4.c

```
 1 #include   "icmpd.h"
 2 #include   <netinet/in_systm.h>
 3 #include   <netinet/ip.h>
 4 #include   <netinet/ip_icmp.h>
 5 #include   <netinet/udp.h>
 6 int
 7 readable_v4(void)
 8 {
 9     int    i, hlen1, hlen2, icmplen, sport;
10     char   buf[MAXLINE];
11     char   srcstr[INET_ADDRSTRLEN], dststr[INET_ADDRSTRLEN];
12     ssize_t n;
13     socklen_t len;
14     struct ip *ip, *hip;
15     struct icmp *icmp;
16     struct udphdr *udp;
17     struct sockaddr_in from, dest;
```

Figura 28.39 Tratamento do datagrama ICMPv4 recebido, primeira metade (*continua*).

```
18    struct icmpd_err icmpd_err;
19    len = sizeof(from);
20    n = Recvfrom(fd4, buf, MAXLINE, 0, (SA *) &from, &len);
21    printf("%d bytes ICMPv4 from %s:", n, Sock_ntop_host((SA *) &from, len));
22    ip = (struct ip *) buf;       /* inicia do cabeçalho IP */
23    hlen1 = ip->ip_hl << 2;       /* comprimento do cabeçalho IP */
24    icmp = (struct icmp *) (buf + hlen1);    /* início do cabeçalho ICMP */
25    if ( (icmplen = n - hlen1) < 8)
26        err_quit("icmplen (%d) < 8", icmplen);
27    printf(" type = %d, code = %d\n", icmp->icmp_type, icmp->icmp_code);
```

icmpd/readable_v4.c

Figura 28.39 Tratamento do datagrama ICMPv4 recebido, primeira metade (*continuação*).

Essa função imprime algumas informações sobre cada mensagem ICMPv4 recebida. Isso foi feito para depuração no desenvolvimento desse daemon e poderia ser gerado na saída com base em um argumento de linha de comando.

A Figura 28.40 mostra a última metade de nossa função `readable_v4`.

icmpd/readable_v4.c

```
28    if (icmp->icmp_type == ICMP_UNREACH ||
29        icmp->icmp_type == ICMP_TIMXCEED ||
30        icmp->icmp_type == ICMP_SOURCEQUENCH) {
31        if (icmplen < 8 + 20 + 8)
32           err_quit("icmplen (%d) < 8 + 20 + 8", icmplen);
33        hip = (struct ip *) (buf + hlen1 + 8);
34        hlen2 = hip->ip_hl << 2;
35        printf("\tsrcip = %s, dstip = %s, proto = %d\n",
36              Inet_ntop(AF_INET, &hip->ip_src, srcstr, sizeof(srcstr)),
37              Inet_ntop(AF_INET, &hip->ip_dst, dststr, sizeof(dststr)),
38              hip->ip_p);
39        if (hip->ip_p == IPPROTO_UDP) {
40           udp = (struct udphdr *) (buf + hlen1 + 8 + hlen2);
41           sport = udp->uh_sport;
42               /* localiza o soquete de domínio Unix do cliente,
                   envia cabeçalhos */
43           for (i = 0; i <= maxi; i++) {
44               if (client[i].connfd >= 0 &&
45                   client[i].family == AF_INET &&
46                   client[i].lport == sport) {
47                   bzero(&dest, sizeof(dest));
48                   dest.sin_family = AF_INET;
49 #ifdef HAVE_SOCKADDR_SA_LEN
50                   dest.sin_len = sizeof(dest);
51 #endif
52                   memcpy(&dest.sin_addr, &hip->ip_dst,
53                          sizeof(struct in_addr));
54                   dest.sin_port = udp->uh_dport;
55                   icmpd_err.icmpd_type = icmp->icmp_type;
56                   icmpd_err.icmpd_code = icmp->icmp_code;
57                   icmpd_err.icmpd_len = sizeof(struct sockaddr_in);
58                   memcpy(&icmpd_err.icmpd_dest, &dest, sizeof(dest));
59                       /* tipo de conversão & código para valor de errno
                            razoável */
```

Figura 28.40 Tratamento do datagrama ICMPv4 recebido, segunda metade (*continua*).

```
60                  icmpd_err.icmpd_errno = EHOSTUNREACH; /* default */
61                  if (icmp->icmp_type == ICMP_UNREACH) {
62                      if (icmp->icmp_code == ICMP_UNREACH_PORT)
63                          icmpd_err.icmpd_errno = ECONNREFUSED;
64                      else if (icmp->icmp_code == ICMP_UNREACH_NEEDFRAG)
65                          icmpd_err.icmpd_errno = EMSGSIZE;
66                  }
67                  Write(client[i].connfd, &icmpd_err, sizeof(icmpd_err));
68              }
69          }
70      }
71  }
72  return (--nready);
73 }
```

icmpd/readable_v4.c

Figura 28.40 Tratamento do datagrama ICMPv4 recebido, segunda metade (*continuação*).

Verificação do tipo de mensagem, notificação à aplicação

29–31 As únicas mensagens ICMPv4 que passamos para a aplicação são "destino inacessível", "tempo excedido" e "extinção de origem" (Figura 28.30).

Verificação de erro de UDP, localização do cliente

34–42 `hip` aponta para o cabeçalho IP que é retornado após o cabeçalho ICMP. Esse é o cabeçalho IP do datagrama que omitiu o erro de ICMP. Verificamos se esse datagrama IP é UDP e, então, buscamos o número da porta UDP de origem do cabeçalho UDP que vem após o cabeçalho IP.

43–55 Uma pesquisa é feita em todas as estruturas `client`, em busca de uma família de endereços e de uma porta correspondentes. Se uma correspondência for encontrada, uma estrutura de endereço de soquete IPv4 é construída, contendo o endereço IP e a porta de destino do datagrama UDP que causou o erro.

Construção da estrutura `icmpd_err`

56–70 É construída uma estrutura `icmpd_err`, que é enviada para o cliente através da sua conexão de domínio Unix. O tipo e o código da mensagem ICMPv4 são primeiramente mapeados em um valor `errno`, conforme descrito na Figura 28.30.

Os erros de ICMPv6 são tratados por nossa função `readable_v6`, cuja primeira metade é mostrada na Figura 28.41. O tratamento de ICMPv6 é semelhante ao código das Figuras 28.12 e 28.24.

icmpd/readable_v6.c

```
 1 #include   "icmpd.h"
 2 #include   <netinet/in_systm.h>
 3 #include   <netinet/ip.h>
 4 #include   <netinet/ip_icmp.h>
 5 #include   <netinet/udp.h>

 6 #ifdef IPV6
 7 #include   <netinet/ip6.h>
 8 #include   <netinet/icmp6.h>
 9 #endif

10 int
11 readable_v6(void)
12 {
13 #ifdef IPV6
```

Figura 28.41 Tratamento do datagrama ICMPv6 recebido, primeira metade (*continua*).

```
14      int     i, hlen2, icmp6len, sport;
15      char    buf[MAXLINE];
16      char    srcstr[INET6_ADDRSTRLEN], dststr[INET6_ADDRSTRLEN];
17      ssize_t n;
18      socklen_t len;
19      struct ip6_hdr *ip6, *hip6;
20      struct icmp6_hdr *icmp6;
21      struct udphdr *udp;
22      struct sockaddr_in6 from, dest;
23      struct icmpd_err icmpd_err;

24      len = sizeof(from);
25      n = Recvfrom(fd6, buf, MAXLINE, 0, (SA *) &from, &len);

26      printf("%d bytes ICMPv6 from %s:", n, Sock_ntop_host((SA *) &from, len));

27      icmp6 = (struct icmp6_hdr *) buf; /* início do cabeçalho ICMPv6 */
28      if ( (icmp6len = n) < 8)
29          err_quit("icmp6len (%d) < 8", icmp6len);

30      printf(" type = %d, code = %d\n", icmp6->icmp6_type, icmp6->icmp6_code);
```

icmpd/readable_v6.c

Figura 28.41 Tratamento do datagrama ICMPv6 recebido, primeira metade (*continuação*).

A segunda metade de nossa função `readable_v6` é mostrada na Figura 28.42. Esse código é semelhante ao da Figura 28.40: ele verifica o tipo de erro de ICMP, verifica se o datagrama que o causou era UDP e, então, constrói a estrutura `icmpd_err`, que é enviada para o cliente.

icmpd/readable_v6.c

```
31      if (icmp6->icmp6_type == ICMP6_DST_UNREACH ||
32          icmp6->icmp6_type == ICMP6_PACKET_TOO_BIG ||
33          icmp6->icmp6_type == ICMP6_TIME_EXCEEDED) {
34          if (icmp6len < 8 + 8)
35              err_quit("icmp6len (%d) < 8 + 8", icmp6len);

36          hip6 = (struct ip6_hdr *) (buf + 8);
37          hlen2 = sizeof(struct ip6_hdr);
38          printf("\tsrcip = %s, dstip = %s, next hdr = %d\n",
39                 Inet_ntop(AF_INET6, &hip6->ip6_src, srcstr, sizeof(srcstr)),
40                 Inet_ntop(AF_INET6, &hip6->ip6_dst, dststr, sizeof(dststr)),
41                 hip6->ip6_nxt);
42          if (hip6->ip6_nxt == IPPROTO_UDP) {
43              udp = (struct udphdr *) (buf + 8 + hlen2);
44              sport = udp->uh_sport;

45                  /* localiza o soquete de domínio Unix do cliente,
                       envia cabeçalhos */
46              for (i = 0; i <= maxi; i++) {
47                  if (client[i].connfd >= 0 &&
48                      client[i].family == AF_INET6 &&
49                      client[i].lport == sport) {

50                      bzero(&dest, sizeof(dest));
51                      dest.sin6_family = AF_INET6;
52  #ifdef HAVE_SOCKADDR_SA_LEN
53                      dest.sin6_len = sizeof(dest);
54  #endif
55                      memcpy(&dest.sin6_addr, &hip6->ip6_dst,
56                             sizeof(struct in6_addr));
57                      dest.sin6_port = udp->uh_dport;
```

Figura 28.42 Tratamento do datagrama ICMPv6 recebido, segunda metade (*continua*).

```
58                  icmpd_err.icmpd_type = icmp6->icmp6_type;
59                  icmpd_err.icmpd_code = icmp6->icmp6_code;
60                  icmpd_err.icmpd_len = sizeof(struct sockaddr_in6);
61                  memcpy(&icmpd_err.icmpd_dest, &dest, sizeof(dest));
62                      /* tipo de conversão & código para valor de errno
                         razoável */
63                  icmpd_err.icmpd_errno = EHOSTUNREACH; /* default */
64                  if (icmp6->icmp6_type == ICMP6_DST_UNREACH &&
65                      icmp6->icmp6_code == ICMP6_DST_UNREACH_NOPORT)
66                      icmpd_err.icmpd_errno = ECONNREFUSED;
67                  if (icmp6->icmp6_type == ICMP6_PACKET_TOO_BIG)
68                      icmpd_err.icmpd_errno = EMSGSIZE;
69                  Write(client[i].connfd, &icmpd_err, sizeof(icmpd_err));
70              }
71          }
72      }
73  }
74  return (--nready);
75 #endif
76 }
```

icmpd/readable_v6.c

Figura 28.42 Tratamento do datagrama ICMPv6 recebido, segunda metade (*continuação*).

28.8 Resumo

Os soquetes brutos fornecem três recursos:

- Podemos ler e gravar pacotes ICMPv4, IGMPv4 e ICMPv6.
- Podemos ler e gravar datagramas IP com um campo de protocolo que o kernel não trata.
- Podemos construir nosso próprio cabeçalho IPv4, normalmente utilizado para propósitos de diagnóstico (ou por hackers, infelizmente).

Duas ferramentas de diagnóstico comumente utilizadas, `ping` e `traceroute`, usam soquetes brutos e desenvolvemos nossas próprias versões de ambas, que suportam IPv4 e IPv6. Também desenvolvemos nosso próprio daemon `icmpd`, que dá acesso aos erros de ICMP para um soquete UDP. Esse exemplo também forneceu uma amostra da passagem de descritor através de um soquete de domínio Unix, entre um cliente e um servidor não relacionados.

Exercícios

28.1 Dissemos que quase todos os campos em um cabeçalho IPv6 e todos os cabeçalhos de extensão estão disponíveis para a aplicação por meio de opções de soquete ou dados auxiliares. Quais informações em um datagrama IPv6 *não* estão disponíveis para uma aplicação?

28.2 O que acontece na Figura 28.40 se, por alguma razão, o cliente pára de ler a partir de sua conexão de domínio Unix com o daemon `icmpd` e uma grande quantidade de erros de ICMP chega ao cliente? Qual é a solução mais fácil?

28.3 Se especificarmos o endereço de broadcast dirigido à sub-rede como nosso programa `ping`, isso funcionará. Isto é, uma solicitação de eco ICMP de broadcast é enviada como um broadcast de camada de enlace, mesmo que não configuremos a opção de soquete `SO_BROADCAST`. Por quê?

28.4 O que acontece com nosso programa `ping` se o utilizarmos no grupo de multicast de todos os hosts, 224.0.0.1 em um host distribuído por vários lugares?

Capítulo 29

Acesso ao Enlace de Dados

29.1 Visão geral

Fornecer acesso à camada de enlace de dados para uma aplicação é uma característica poderosa que está disponível nos sistemas operacionais mais atuais. Isso oferece os seguintes recursos:

- Capacidade de observar os pacotes recebidos pela camada de enlace de dados, permitindo que programas, como o `tcpdump`, sejam executados em sistemas normais de computadores (em oposição aos dispositivos de hardware dedicados a observar pacotes). Quando combinado com a capacidade da interface de rede de entrar em um *modo promíscuo*, isso permite que uma aplicação observe todos os pacotes presentes no cabo local e não apenas aqueles destinados ao host em que o programa está executando.

  Essa capacidade é menos útil em redes comutadas (*switched networks*), que têm se tornado bastante comuns. Isso porque o comutador (*switch*) passa o tráfego para uma porta somente se ele for endereçado aos dispositivos ligado(s) a essa porta (unicast, multicast ou broadcast). Para monitorar tráfego transportado em outras portas do comutador, a porta deste deve ser configurada para receber outro tráfego, o que é freqüentemente chamado de *modo monitor* ou *espelhamento de porta*. Observe que muitos dispositivos que não se poderia esperar que fossem comutadores na verdade o são; por exemplo, um hub de 10/100Mbps de velocidade dual normalmente é um comutador de duas portas: uma para os sistemas de 100 Mbps e a outra para os de 10 Mbps.

- Capacidade de executar certos programas como aplicações normais, em vez de como parte do kernel. Por exemplo, a maioria das versões de Unix de um servidor de RARP é constituída de aplicações normais que lêem solicitações de RARP do enlace de dados (as solicitações de RARP não são datagramas IP) e, então, escrevem a resposta de volta no enlace de dados.

Os três métodos comuns para acessar a camada de enlace de dados no Unix são a filtragem de pacotes BSD (BPF), a interface de provedor de enlace de dados SVR4 (Datalink Provider Interface – DLPI) e a interface `SOCK_PACKET` do Linux. Apresentaremos uma visão

geral desses três métodos e então descreveremos a `libpcap`, a biblioteca de captura de pacotes disponível publicamente. Essa biblioteca trabalha com todos os três métodos e utilizá-la torna nossos programas independentes do acesso ao enlace de dados real fornecido pelo SO. Descreveremos essa biblioteca desenvolvendo um programa que envia consultas de DNS para um servidor de nomes (construímos nossos próprios datagramas UDP e os gravamos em um soquete bruto) e lê a resposta utilizando `libpcap` para determinar se o servidor de nomes permite somas de verificação UDP.

29.2 Filtro de pacotes BSD (BSD Packet Filter – BPF)

A implementação 4.4BSD e muitas outras derivadas do Berkeley suportam BPF, o filtro de pacotes BSD. A implementação do BPF está descrita no Capítulo 31 do TCPv2. A história do BPF, uma descrição da pseudomáquina BPF e uma comparação com o filtro de pacotes SunOS 4.1.x NIT é fornecida em McCanne e Jacobson (1993).

Cada driver de enlace de dados chama o BPF imediatamente antes que um pacote seja transmitido e imediatamente o recebimento de um pacote, como mostrado nas Figura 29.1.

Exemplos dessas chamadas para uma interface Ethernet aparecem na Figuras 4.11 e 4.19 do TCPv2. A razão para chamar o BPF logo que possível após a recepção e o mais tarde possível antes da transmissão é fornecer indicações de tempo (*timestamps*) exatas.

Embora não seja difícil fornecer uma derivação no enlace de dados para capturar todos os pacotes, o poder do BPF está em sua capacidade de filtragem. Cada aplicação que abre um dispositivo BPF pode carregar seu próprio filtro, o qual, então, é aplicado pelo BPF a cada pacote. Enquanto alguns filtros são simples (o filtro "`udp or tcp`" recebe somente pacotes UDP ou TCP), outros podem examinar campos nos cabeçalhos de pacote para certos valores. Por exemplo:

```
tcp and port 80 and tcp[13:1] & 0x7 != 0
```

foi utilizado no Capítulo 14 do TCPv3 para coletar somente segmentos de TCP na porta 80, que tinham os flags SYN, FIN ou RST ativos. A expressão `tcp[13:1]` refere-se ao valor de 1 byte que inicia no deslocamento de byte 13 a partir do começo do cabeçalho TCP.

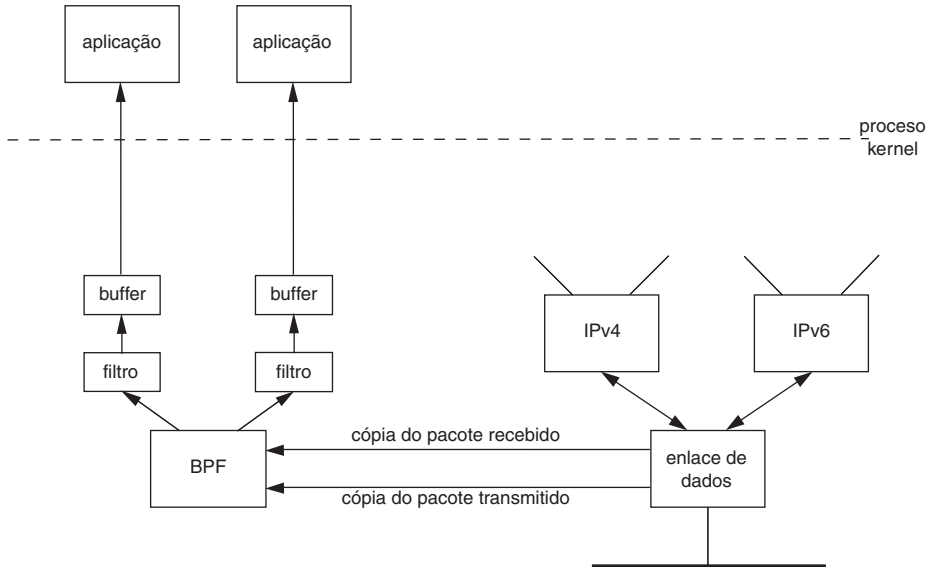


Figura 29.1 Captura de pacote utilizando BPF.

O BPF implementa uma máquina de filtro baseado em registrador que aplica filtros específicos da aplicação a cada pacote recebido. Embora seja possível escrever programas de filtro na linguagem de máquina dessa pseudomáquina (o que está descrito na página man do BPF), a interface mais simples é compilar strings ASCII (como aquela que começa com `tcp` que acabamos de mostrar) nessa linguagem de máquina, utilizando a função `pcap_compile` que descreveremos na Seção 29.7.

Três técnicas são utilizadas pelo BPF para reduzir seus overheads:

- A filtragem do BPF se dá dentro do kernel, o que minimiza o volume de dados copiados do BPF para a aplicação. Essa cópia, do espaço do kernel para o espaço do usuário, é dispendiosa. Se cada pacote fosse copiado, o BPF poderia ter problemas para acompanhar os enlaces de dados rápidos.
- Somente uma parte de cada pacote é passada pelo BPF para a aplicação. Isso é chamado de *comprimento de instantâneo* (*snapshot length*) ou *snaplen*. A maioria das aplicações precisa somente dos cabeçalhos de pacote e não dos dados do pacote. Isso também reduz o volume de dados copiados pelo BPF na aplicação. O `tcpdump`, por exemplo, tem como padrão o valor 96 para isso, o que permite um cabeçalho Ethernet de 14 bytes, um cabeçalho IPv6 de 40 bytes, um cabeçalho TCP de 20 bytes e 22 bytes de dados. Porém, imprimir informações adicionais para outros protocolos (por exemplo, DNS e NFS) exige que o usuário aumente esse valor, quando a função `tcpdump` está sendo executada.
- O BPF armazena em buffer os dados destinados a uma aplicação e esse buffer é copiado para a aplicação somente quando está cheio ou quando o *tempo-limite de leitura* expira. Esse valor de tempo-limite pode ser especificado pela aplicação. `tcpdump`, por exemplo, configura o tempo-limite como 1.000 ms, enquanto o daemon de RARP o configura como 0 (pois há poucos pacotes RARP e o servidor de RARP precisa enviar uma resposta assim que receber a solicitação). O propósito do armazenamento em buffer é reduzir o número de chamadas de sistema. O mesmo número de pacotes ainda é copiado entre o BPF e a aplicação, mas cada chamada de sistema tem um overhead e reduzir o número dessas chamadas sempre reduz o overhead. (A Figura 3.1 da APUE compara o overhead da chamada de sistema `read`, por exemplo, ao ler determinado arquivo em trechos de diferentes tamanhos, variando entre 1 e 131.072 bytes.)

  Embora mostremos somente um único buffer na Figura 29.1, o BPF mantém dois buffers para cada aplicação e preenche um deles enquanto o outro está sendo copiado para a aplicação. Essa é a técnica de *buffer duplo*-padrão.

Na Figura 29.1, mostramos somente a recepção de pacotes do BPF: os pacotes recebidos pelo enlace de dados a partir dos níveis mais baixos (a rede) e os pacotes recebidos pelo enlace de dados vindos dos níveis acima (IP). A aplicação também pode gravar no BPF, fazendo os pacotes serem enviados para fora do enlace de dados, mas a maioria das aplicações somente lê do BPF. Não há nenhuma razão para escrever no BPF para enviar datagramas IP, pois a opção de soquete `IP_HDRINCL` permite escrever qualquer tipo de datagrama de IP desejado, incluindo o cabeçalho de IP. (Mostramos um exemplo disso na Seção 29.7.) A única razão para escrever no BPF é enviar nossos próprios pacotes de rede que não sejam datagramas IP. O daemon RARP faz isso, por exemplo, para enviar suas respostas RARP, que não são datagramas IP.

Para acessar o BPF, devemos abrir (`open`) um dispositivo BPF que não esteja correntemente aberto. Por exemplo, poderíamos tentar usar `/dev/bpf0` e, se o retorno de erro for `EBUSY`, então poderíamos tentar usar `/dev/bpf1`, e assim por diante. Uma vez que um dispositivo esteja aberto, quase uma dezena de comandos `ioctl` configura as características do dispositivo: carregamento do filtro, configuração do tempo-limite de leitura, configuração do tamanho do buffer, anexação de um enlace de dados no dispositivo BPF, permissão do modo promíscuo e assim por diante. Então, a E/S é realizada utilizando `read` e `write`.

29.3 Datalink Provider Interface (DLPI)

O SVR4 fornece acesso ao enlace de dados por meio da DLPI. A DLPI é uma interface independente de protocolo projetada pela AT&T, que se liga ao serviço fornecido pela camada de enlace de dados (Unix Internacional, 1991). O acesso à DLPI é feito por meio do envio e da recepção de mensagens STREAMS.

Há dois estilos de DLPI. Em um deles, há um único dispositivo a abrir e a interface desejada é especificada utilizando uma solicitação DLPI `DL_ATTACH_REQ`. No outro, a aplicação simplesmente abre (`open`) o dispositivo (por exemplo, `le0`). Mas, para uma operação eficiente, normalmente dois módulos STREAMS adicionais são colocados no fluxo: `pfmod`, que realiza filtragem de pacotes dentro do kernel, e `bufmod`, que armazena em buffer os dados destinados à aplicação. Mostramos isso na Figura 29.2.

Conceitualmente, isso é semelhante ao que descrevemos na seção anterior para o BPF: `pfmod` suporta filtragem dentro do kernel, utilizando uma pseudomáquina, e `bufmod` reduz o volume de dados e o número de chamadas de sistema, suportando um comprimento instantâneo e um tempo-limite de leitura.

Uma diferença interessante, entretanto, é o tipo de pseudomáquina suportada pelos filtros BPF e `pfmod`. O filtro BPF é um grafo acíclico direcionado de fluxo de controle (CFG), enquanto o `pfmod` utiliza uma árvore de expressão booleana. O primeiro é mapeado naturalmente em código para uma máquina de registradores, enquanto o último é mapeado naturalmente em código para uma máquina de pilha (McCanne e Jacobson, 1993). Esse artigo mostra que a implementação de CFG utilizada pelo BPF normalmente é de 3 a 20 vezes mais rápida que a árvore de expressão booleana, dependendo da complexidade do filtro.

Outra diferença é que o BPF sempre toma a decisão de filtragem antes de copiar o pacote, a fim de não copiar pacotes que o filtro descartará. Dependendo da implementação da DLPI, o pacote pode ser copiado de forma a fornecê-lo ao `pfmod`, que pode então descartá-lo.

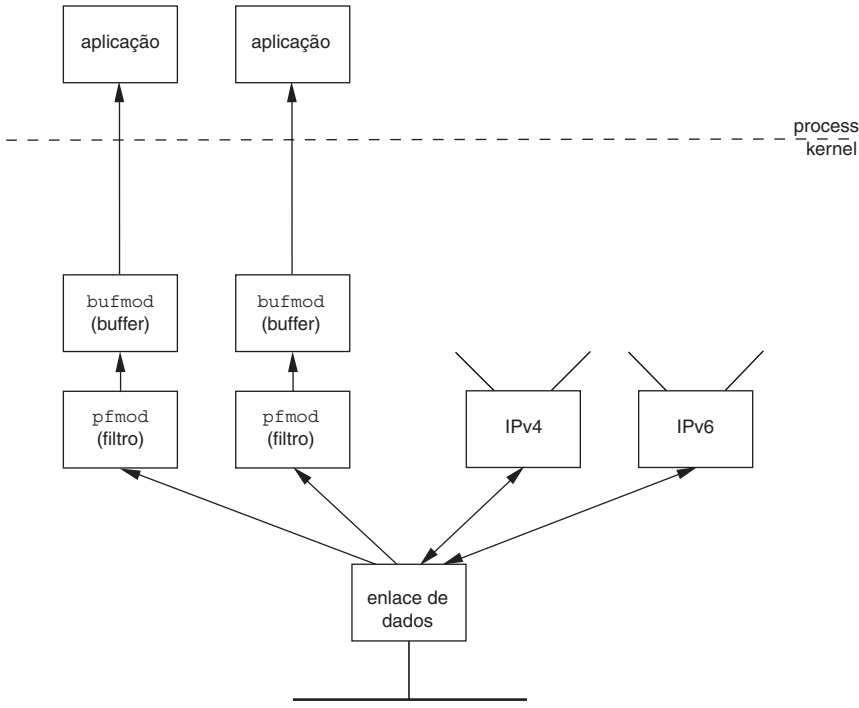


Figura 29.2 Captura de pacote utilizando DLPI, `pfmod` e `bufmod`.

29.4 Linux: SOCK_PACKET e PF_PACKET

Há dois métodos de recepção de pacotes da camada de enlace de dados no Linux. O método original, que é mais amplamente disponível, mas menos flexível, é criar um soquete do tipo SOCK_PACKET. O método mais recente, que introduz mais filtragem e recursos de desempenho, é criar um soquete da família PF_PACKET. Para qualquer um deles, devemos ter privilégios suficientes (semelhante a criar um soquete bruto) e o terceiro argumento de socket deve ser um valor diferente de zero, especificando o tipo de quadro Ethernet. Ao se utilizar soquetes PF_PACKET, o segundo argumento de socket pode ser SOCK_DGRAM, para pacotes "processados" com o cabeçalho da camada de enlace removido; ou SOCK_RAW, para o pacote da camada de enlace completo. Os soquetes SOCK_PACKET retornam somente o pacote da camada de enlace completo. Por exemplo, para receber todos os quadros do enlace de dados, escrevemos

```
fd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));      /* sistemas mais recentes */
```

ou

```
fd = socket(AF_INET, SOCK_PACKET, htons(ETH_P_ALL));     /* sistemas mais antigos */
```

Isso retornaria quadros para todos os protocolos que o enlace de dados recebe.

Se quiséssemos somente quadros IPv4, a chamada seria

```
fd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_IP));       /* sistemas mais recentes */
```

ou

```
fd = socket(AF_INET, SOCK_PACKET, htons(ETH_P_IP));      /* sistemas mais antigos */
```

Outras constantes para o último argumento são ETH_P_ARP e ETH_P_IPV6, por exemplo.

A especificação de um protocolo ETH_P_*xxx* informa ao enlace de dados quais tipos de quadro devem passar para o soquete, ao serem recebidos pelo enlace de dados. Se o enlace de dados suporta um modo promíscuo (por exemplo, Ethernet), então o dispositivo também deve ser colocado em um modo promíscuo, se desejado. Isso é feito com uma opção de soquete PACKET_ADD_MEMBERSHIP, utilizando uma estrutura packet_mreq especificando uma interface e uma ação PACKET_MR_PROMISC. Isso é feito em sistemas mais antigos, em vez de se usar ioctl de SIOCGIFFLAGS, para buscar os flags, configurar o flag IFF_PROMISC e, então, armazenar os flags com SIOCSIFFLAGS. Infelizmente, com esse método, vários ouvintes promíscuos podem interferir uns nos outros e um programa com bugs pode deixar o modo promíscuo ativo, mesmo depois de terminar.

Algumas diferenças ficam evidentes ao se comparar esse recurso do Linux com o BPF e com o DLPI:

1. O recurso do Linux não fornece nenhum armazenamento em buffer do kernel e a filtragem do kernel está disponível somente nos sistemas mais novos (por intermédio da opção de soquete SO_ATTACH_FILTER). Há um buffer normal de recepção de soquete, mas vários quadros não podem ser armazenados em conjunto no buffer e passados para a aplicação com um único comando read. Isso aumenta o overhead envolvido na cópia de quantidades potencialmente volumosas de dados do kernel para a aplicação.
2. SOCK_PACKET não fornece nenhuma filtragem por dispositivo. (Os soquetes PF_PACKET podem ser vinculados a um dispositivo chamando-se bind.) Se ETH_P_IP for especificado na chamada de socket, então todos os pacotes IPv4 de todos os dispositivos (Ethernet, links de PPP, links de SLIP e o dispositivo de loopback, por exemplo) são passados para o soquete. Uma estrutura de endereço de

soquete genérica é retornada por `recvfrom` e o membro `sa_data` contém o nome do dispositivo (por exemplo, `eth0`). A aplicação deve então descartar dados de qualquer dispositivo no qual não esteja interessada. O problema, novamente, é que dados demais podem ser retornados para a aplicação, os quais podem atrapalhar quando for necessário um monitoramento de uma rede de alta velocidade.

29.5 `libpcap`: biblioteca de captura de pacotes

A biblioteca de captura de pacotes, `libpcap`, fornece acesso independente da implementação ao recurso de captura de pacotes subjacente fornecido pelo SO. Atualmente, ela suporta somente a leitura de pacotes (embora adicionar algumas linhas de código à biblioteca permita que se escrevam também pacotes de enlace de dados em alguns sistemas). Veja, na próxima seção, uma descrição de outra biblioteca que suporta não apenas gravação de pacotes de enlace de dados, mas também a construção de pacotes arbitrários.

Atualmente, existe suporte para BPF sob kernels derivados do Berkeley, DLPI sob HP-UX e Solaris 2.x, NIT sob SunOS 4.1.x, os soquetes `SOCK_PACKET` e `PF_PACKET` do Linux e alguns outros sistemas operacionais. Essa biblioteca é utilizada por `tcpdump`. A biblioteca abrange aproximadamente 25 funções, mas, em vez de apenas descrevê-las, mostraremos a utilização real das funções comuns em um exemplo completo, em uma seção posterior. Todas as funções de biblioteca iniciam com o prefixo `pcap_`. A página man de `pcap` descreve essas funções com mais detalhes.

A biblioteca está publicamente disponível no endereço `http://www.tcpdump.org/`.

29.6 `libnet`: biblioteca de criação e injeção de pacotes

`libnet` fornece uma interface para construir e injetar pacotes arbitrários na rede. Ela fornece modos de acesso tanto de soquete bruto como de enlace de dados, de maneira independente da implementação.

A biblioteca oculta muitos dos detalhes da criação de cabeçalhos IP e UDP ou TCP, e fornece acesso simples e portável para gravação de pacotes de enlace de dados e brutos. Assim como acontece com `libpcap`, a biblioteca é composta de um grande número de funções. Mostraremos como utilizar um pequeno grupo das funções para acessar soquetes brutos, no exemplo da seção a seguir, bem como o código exigido para utilizar soquetes brutos diretamente, para comparação. Todas as funções de biblioteca iniciam com o prefixo `libnet_`; a página man e o manual on-line de `libnet` descrevem essas funções com mais detalhes.

A biblioteca está disponível publicamente no endereço `http://www.packetfactory.net/libnet/`. O manual on-line está no endereço `http://www.packetfactory.net/libnet/manual/`. Quando este livro estava sendo escrito, o único manual disponível era para a versão obsoleta 1.0; a versão suportada, 1.1, tem uma API *significativamente* diferente. Este exemplo utiliza a API 1.1.

29.7 Examinando o campo UDP de soma de verificação

Agora, desenvolveremos um exemplo que envia um datagrama UDP contendo uma consulta de DNS para um servidor de nomes e lê a resposta utilizando a biblioteca de captura de pacote. O objetivo do exemplo é determinar se o servidor de nomes calcula uma soma de verificação (*checksum*) UDP ou não. Com o IPv4, o cálculo de uma soma de verificação UDP é opcional. Os sistemas mais atuais ativam essas somas de verificação por default, mas, infelizmente, os sistemas mais antigos, notadamente o SunOS 4.1.x, desativam essas somas de veri-

ficação por default. Atualmente, todos os sistemas, especialmente aqueles que executam um servidor de nomes, *sempre* devem executar com somas de verificação UDP ativadas, pois datagramas corrompidos podem corromper o banco de dados do servidor.

> A ativação ou desativação de somas de verificação UDP normalmente é feita para todo o sistema, conforme descrito no Apêndice E do TCPv1.

Construiremos nosso próprio datagrama UDP (a consulta de DNS) e o gravaremos em um soquete bruto. Também mostraremos o mesmo código utilizando `libnet`. Poderíamos utilizar um soquete de UDP normal para enviar a consulta, mas queremos mostrar como se utiliza a opção de soquete `IP_HDRINCL` para construir um datagrama de IP completo.

Nunca podemos obter a soma de verificação UDP ao ler de um soquete UDP normal e nunca podemos ler pacotes UDP nem TCP utilizando um soquete bruto (Seção 28.4). Portanto, devemos utilizar o recurso de captura de pacotes para obter por inteiro o datagrama UDP que contém a resposta do servidor de nomes.

Também examinaremos o campo de soma de verificação UDP no cabeçalho UDP. Se for 0, o servidor não tem somas de verificação UDP ativadas.

A Figura 29.3 resume a operação de nosso programa.

Gravamos nossos próprios datagramas UDP no soquete bruto e lemos de volta as respostas utilizando `libpcap`. Note que o UDP também recebe a resposta do servidor de nomes e responderá com "porta inacessível" de ICMP, pois nada sabe sobre o número da porta de origem que nossa aplicação escolhe. O servidor de nomes ignorará esse erro de ICMP. Também observamos que é mais difícil escrever um programa de teste dessa forma, que utiliza TCP, mesmo que possamos escrever facilmente nossos próprios segmentos de TCP, pois qualquer resposta aos segmentos de TCP que gerarmos normalmente fará com que nosso TCP responda com um RST para qualquer um que tenhamos enviado o segmento.

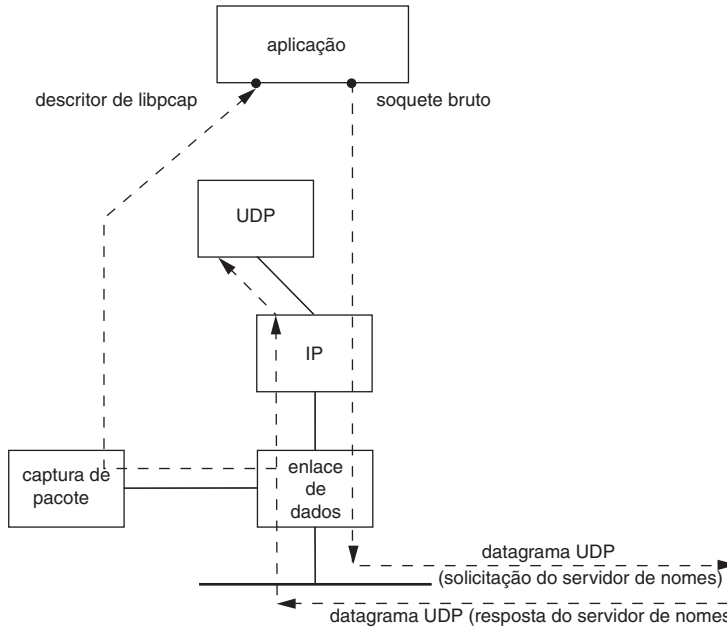


Figura 29.3 Nossa aplicação para verificar se um servidor de nomes tem as somas de verificação UDP ativadas.

Uma maneira de contornar isso é enviar segmentos de TCP com um endereço IP de origem que pertença à sub-rede anexada, mas que não esteja atribuído a algum outro nó. Adicione uma entrada de ARP ao host emissor para esse novo endereço IP, para que o host emissor responda às solicitações ARP para o novo endereço, mas não configure o novo endereço IP como um alias. Isso fará com que a pilha de IP no host emissor descarte os pacotes recebidos para esse novo endereço IP, supondo que o host emissor não esteja agindo como um roteador.

A Figura 29.4 é um resumo das funções que compreendem nosso programa.

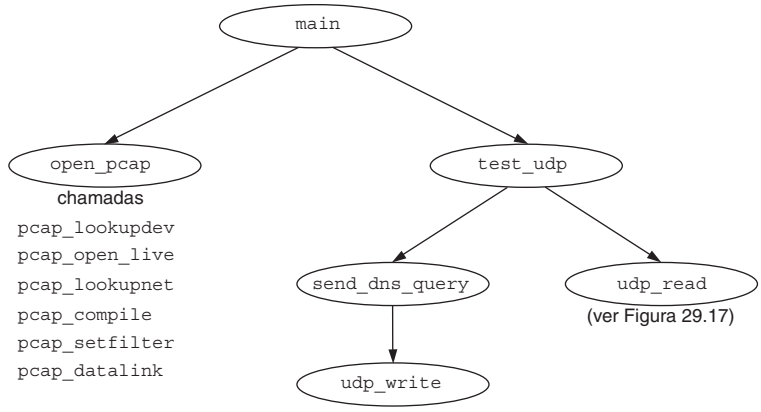


Figura 29.4 Resumo das funções de nosso programa `udpcksum`.

A Figura 29.5 mostra o cabeçalho `udpcksum.h`, que inclui nosso cabeçalho `unp.h` básico, junto com os vários cabeçalhos de sistema que são necessários para acessar as definições de estrutura dos cabeçalhos de pacote IP e UDP.

udpcksum/udpcksum.h

```
 1 #include   "unp.h"
 2 #include   <pcap.h>

 3 #include   <netinet/in_systm.h>  /* exigido pelo ip.h */
 4 #include   <netinet/in.h>
 5 #include   <netinet/ip.h>
 6 #include   <netinet/ip_var.h>
 7 #include   <netinet/udp.h>
 8 #include   <netinet/udp_var.h>
 9 #include   <net/if.h>
10 #include   <netinet/if_ether.h>

11 #define TTL_OUT   64             /* TTL enviado */

12                  /* declara variáveis globais */
13 extern struct sockaddr *dest, *local;
14 extern socklen_t destlen, locallen;
15 extern int datalink;
16 extern char *device;
17 extern pcap_t *pd;
18 extern int rawfd;
19 extern int snaplen;
20 extern int verbose;
21 extern int zerosum;

22                  /* protótipos de função */
```

Figura 29.5 Cabeçalho `udpcksum.h` (*continua*).

```
23 void   cleanup(int);
24 char  *next_pcap(int *);
25 void   open_output(void);
26 void   open_pcap(void);
27 void   send_dns_query(void);
28 void   test_udp(void);
29 void   udp_write(char *, int);
30 struct udpiphdr *udp_read(void);
```

udpcksum/udpcksum.h

Figura 29.5 Cabeçalho `udpcksum.h` (*continuação*).

3–10 Cabeçalhos de Internet adicionais são exigidos para lidar com os campos de cabeçalho IP e UDP.
11–29 Definimos algumas variáveis globais e protótipos para nossas próprias funções, que mostraremos em breve.

A primeira parte da função `main` aparece na Figura 29.6.

udpcksum/main.c

```
 1 #include   "udpcksum.h"

 2           /* define variáveis globais */
 3 struct sockaddr *dest, *local;
 4 struct sockaddr_in locallookup;
 5 socklen_t destlen, locallen;

 6 int     datalink;        /* de pcap_datalink(), em <net/bpf.h> */
 7 char   *device;          /* dispositivo pcap */
 8 pcap_t *pd;              /* ponteiro de estrutura de captura de pacote */
 9 int     rawfd;           /* soquete bruto para gravação */
10 int     snaplen = 200;   /* volume de dados a capturar */
11 int     verbose;
12 int     zerosum;         /* envia consulta UDP sem soma de verificação */

13 static void usage(const char *);

14 int
15 main(int argc, char *argv[])
16 {
17     int    c, lopt = 0;
18     char  *ptr, localname[1024], *localport;
19     struct addrinfo *aip;
```

udpcksum/main.c

Figura 29.6 Função `main`: definições.

A próxima parte da função `main`, mostrada na Figura 29.7, processa os argumentos de linha de comando.

udpcksum/main.c

```
20     opterr = 0;               /* não quer getopt() gravando em stderr */
21     while ( (c = getopt(argc, argv, "0i:l:v")) != -1) {
22         switch (c) {

23         case '0':
24             zerosum = 1;
25             break;

26         case 'i':
27             device = optarg;    /* dispositivo de pcap */
28             break;

29         case 'l':               /* endereço IP local e porta #: a.b.c.d.p */
```

Figura 29.7 Função `main`: processa argumentos de linha de comando (*continua*).

```
30              if ( (ptr = strrchr(optarg, '.')) == NULL)
31                  usage("invalid -l option");

32              *ptr++ = 0;         /* nulo substitui o ponto final */
33              localport = ptr;    /* nome de serviço ou número da porta */
34              strncpy(localname, optarg, sizeof(localname));
35              lopt = 1;
36              break;

37          case 'v':
38              verbose = 1;
39              break;

40          case '?':
41              usage("unrecognized option");
42          }
43      }
```

udpcksum/main.c

Figura 29.7 Função `main`: processa argumentos de linha de comando (*continuação*).

Processamento das opções de linha de comando

20–25 Chamamos `getopt` para processar os argumentos de linha de comando. A opção `–0` nos permite enviar nossa consulta de UDP sem soma de verificação UDP, para ver se o servidor trata disso de forma diferente de um datagrama com soma de verificação.

26–28 A opção `–i` nos permite especificar a interface em que vamos receber a resposta do servidor. Se isso não for especificado, a biblioteca de captura de pacote escolherá uma, o que talvez não seja correto em um host distribuído por vários lugares. Essa é uma maneira pela qual a leitura de um dispositivo de captura de pacote difere da leitura de um soquete normal: com um soquete, podemos usar um curinga para endereço local, possibilitando a recepção de pacotes que chegam em qualquer interface, mas com um dispositivo de captura de pacote recebemos os pacotes que chegam a apenas uma interface.

> Observamos que o recurso `SOCK_PACKET` do Linux não limita sua captura de enlace de dados a um único dispositivo. Contudo, `libpcap` fornece essa filtragem com base em seu default ou em nossa opção `–i`.

29–36 A opção `–l` nos permite especificar o endereço IP de origem e o número da porta. A porta (ou um nome de serviço) é obtida como a string após o ponto final e o endereço IP de origem é obtido como tudo que vem antes do ponto final.

A última parte da função `main` aparece na Figura 29.8.

udpcksum/main.c

```
44      if (optind != argc - 2)
45          usage("missing <host> and/or <serv>");

46          /* converte nome de destino e serviço */
47      aip = Host_serv(argv[optind], argv[optind + 1], AF_INET, SOCK_DGRAM);
48      dest = aip->ai_addr;          /* não faz freeaddrinfo() */
49      destlen = aip->ai_addrlen;

50      /*
51       * Precisa do endereço IP local p/ end IP de origem p/ datagramas UDP.
52       * Não pode especificar 0 e deixa escolha de IP, pois precisamos conhecê-lo
53       * para que o pseudocabeçalho calcule a soma de verificação UDP.
54       * Se a opção -l é fornecida, então utiliza esses valores; caso contrário,
55       * conecta um soquete UDP com o destino para determinar o endereço
56       * de origem correto.
57       */
58      if (lopt) {
```

Figura 29.8 Função `main`: converte nomes de host e nomes de serviço; cria soquete (*continua*).

```
59              /* converte nome local e serviço */
60          aip = Host_serv(localname, localport, AF_INET, SOCK_DGRAM);
61          local = aip->ai_addr; /* não faz freeaddrinfo() */
62          locallen = aip->ai_addrlen;
63      } else {
64          int    s;
65          s = Socket(AF_INET, SOCK_DGRAM, 0);
66          Connect(s, dest, destlen);
67          /* o kernel escolhe o endereço local correto para dest */
68          locallen = sizeof(locallookup);
69          local = (struct sockaddr *) &locallookup;
70          Getsockname(s, local, &locallen);
71          if (locallookup.sin_addr.s_addr == htonl(INADDR_ANY))
72              err_quit("Can't determine local address - use -l\n");
73          close(s);
74      }
75      open_output();          /* abre a saída, ou soquete bruto ou libnet */
76      open_pcap();            /* abre dispositivo de captura de pacote */
77      setuid(getuid());       /* não precisa mais de privilégios de superusuário */
78      Signal(SIGTERM, cleanup);
79      Signal(SIGINT, cleanup);
80      Signal(SIGHUP, cleanup);
81      test_udp();
82      cleanup(0);
83  }
```

udpcksum/main.c

Figura 29.8 Função `main`: converte nomes de host e nomes de serviço; cria soquete (*continuação*).

Processamento do nome de destino e da porta

46–49 Verificamos que permanecem exatamente dois argumentos de linha de comando: o nome de host de destino e o nome do serviço. Chamamos `host_serv` para convertê-los em uma estrutura de endereço de soquete, cujo ponteiro salvamos em `dest`.

Processamento do nome local e da porta

50–74 Se for especificado na linha de comando, fazemos então a mesma conversão do nome de host local e da porta, salvando o ponteiro para a estrutura de endereço de soquete em `local`. Caso contrário, determinamos o endereço IP local a ser utilizado, conectando um soquete de datagrama ao destino e armazenando o endereço local resultante em `local`. Como vamos construir nossos próprios cabeçalhos IP e UDP, devemos conhecer o endereço IP de origem quando escrevermos o datagrama UDP. Não podemos deixá-lo como 0 e permitir que o IP escolha o endereço, pois o endereço faz parte do pseudocabeçalho UDP (que descreveremos em breve) que devemos utilizar para o cálculo da soma de verificação UDP.

Criação do soquete bruto e abertura do dispositivo de captura de pacote

75–76 A função `open_output` prepara o método de saída, sejam soquetes brutos ou `libnet`. A função `open_pcap` abre o dispositivo de captura de pacote; mostraremos essa função a seguir.

Alteração das permissões e estabelecimento dos handlers de sinal

77–80 Precisamos de privilégios de superusuário para criar um soquete bruto. Normalmente, precisamos de privilégios de superusuário para abrir o dispositivo de captura de pacote, mas isso depende da implementação. Por exemplo, com o BPF, o administrador pode configurar as per-

missões dos dispositivos `/dev/bpf` com o que for desejado para aquele sistema. Agora, abandonaremos essas permissões adicionais, supondo que o arquivo de programa seja set-user-ID. Se o processo tem privilégios de superusuário, chamar `setuid` configura nosso ID de usuário real, ID de usuário efetivo e set-user-ID salvo com nosso ID de usuário real (`getuid`). Estabelecemos handlers de sinal, para o caso de o usuário terminar o programa antes de ele ter terminado.

Realização do teste e limpeza

81–82 A função `test_udp` (Figura 29.10) realiza o teste e então retorna. `cleanup` (Figura 29.18) imprime estatísticas de resumo da biblioteca de captura de pacote e termina o processo.

A Figura 29.9 mostra a função `open_pcap`, que chamamos a partir da função `main`, para abrir o dispositivo de captura de pacote.

udpcksum/pcap.c

```
 1 #include   "udpcksum.h"

 2 #define CMD   "udp and src host %s and src port %d"

 3 void
 4 open_pcap(void)
 5 {
 6     uint32_t localnet, netmask;
 7     char   cmd[MAXLINE], errbuf[PCAP_ERRBUF_SIZE],
 8         str1[INET_ADDRSTRLEN], str2[INET_ADDRSTRLEN];
 9     struct bpf_program fcode;

10     if (device == NULL) {
11         if ( (device = pcap_lookupdev(errbuf)) == NULL)
12             err_quit("pcap_lookup: %s", errbuf);
13     }
14     printf("device = %s\n", device);

15         /* codificado: promisc = 0, to_ms=500 */
16     if ( (pd = pcap_open_live(device, snaplen, 0, 500, errbuf)) == NULL)
17         err_quit("pcap_open_live: %s", errbuf);

18     if (pcap_lookupnet(device, &localnet, &netmask, errbuf) < 0)
19         err_quit("pcap_lookupnet: %s", errbuf);
20     if (verbose)
21         printf("localnet = %s, netmask = %s\n",
22                Inet_ntop(AF_INET, &localnet, str1, sizeof(str1)),
23                Inet_ntop(AF_INET, &netmask, str2, sizeof(str2)));

24     snprintf(cmd, sizeof(cmd), CMD,
25              Sock_ntop_host(dest, destlen),
26              ntohs(sock_get_port(dest, destlen)));
27     if (verbose)
28         printf("cmd = %s\n", cmd);
29     if (pcap_compile(pd, &fcode, cmd, 0, netmask) < 0)
30         err_quit("pcap_compile: %s", pcap_geterr(pd));

31     if (pcap_setfilter(pd, &fcode) < 0)
32         err_quit("pcap_setfilter: %s", pcap_geterr(pd));

33     if ( (datalink = pcap_datalink(pd)) < 0)
34         err_quit("pcap_datalink: %s", pcap_geterr(pd));
35     if (verbose)
36         printf("datalink = %d\n", datalink);
37 }
```

udpcksum/pcap.c

Figura 29.9 Função `open_pcap`: abre e inicializa dispositivo de captura de pacote.

Escolha do dispositivo de captura de pacote

10-14 Se o dispositivo de captura de pacote não foi especificado (a função opção de linha de comando –i), então pcap_lookupdev escolhe um dispositivo. Ele emite SIOCGIFCONF ioctl e escolhe o dispositivo de numeração mais baixa que esteja ativo, mas não o loopback. Muitas das funções de biblioteca de pcap preenchem uma string de erro, se um erro ocorre. O único argumento dessa função é um array que é preenchido com uma string de erro.

Abertura do dispositivo

15-17 pcap_open_live abre o dispositivo. O termo "live" refere-se a um dispositivo real sendo aberto, em vez de um arquivo contendo pacotes salvos anteriormente. O primeiro argumento é o nome do dispositivo, o segundo é o número de bytes a salvar por pacote (snaplen, que inicializamos como 200 na Figura 29.6), o terceiro é um flag promíscuo, o quarto é um valor de tempo-limite, em milissegundos, e o quinto é um ponteiro para um array de mensagem de erro.

Se o flag promíscuo estiver configurado, a interface é colocada no modo promíscuo, fazendo com que receba todos os pacotes que passam no cabo. Esse é o modo normal para tcpdump. Para nosso exemplo, entretanto, as respostas do servidor de DNS serão enviadas para nosso host.

O argumento de tempo-limite refere-se a um tempo-limite de leitura. Em vez de fazer o dispositivo retornar um pacote para o processo sempre que um pacote é recebido (o que poderia ser ineficiente, ativando uma grande quantidade de cópias de pacotes individuais do kernel para o processo), um pacote é retornado somente quando o buffer de leitura do dispositivo está cheio ou quando o tempo-limite de leitura expira. Se o tempo-limite de leitura é configurado como 0, cada pacote é retornado assim que é recebido.

Obtenção do endereço de rede e da máscara de sub-rede

18-23 pcap_lookupnet retorna o endereço de rede e a máscara de sub-rede do dispositivo de captura de pacote. Devemos especificar a máscara de sub-rede na chamada a pcap_compile que vem a seguir, pois o filtro de pacotes precisa disso para determinar se um endereço IP é um endereço de broadcast dirigido à sub-rede.

Compilação do filtro de pacotes

24-30 pcap_compile recebe uma string de filtro (que construímos no array cmd) e a compila em um programa de filtragem (armazenado em fcode). Isso selecionará os pacotes que queremos receber.

Carregamento do programa de filtragem

31-32 pcap_setfilter pega o programa de filtragem que acabamos de compilar e o carrega no dispositivo de captura de pacote. Isso inicia a captura dos pacotes que selecionamos com o filtro.

Determinação do tipo de enlace de dados

33-36 pcap_datalink retorna o tipo de enlace de dados do dispositivo de captura de pacote. Precisamos disso ao receber pacotes para determinar o tamanho do cabeçalho de enlace de dados que estará no começo de cada pacote que lemos (Figura 29.15).

Depois de chamar open_pcap, a função main chama test_udp, que mostramos na Figura 29.10. Essa função envia uma consulta de DNS e lê a resposta do servidor.

udpcksum/udpcksum.c

```
12 void
13 test_udp(void)
14 {
15     volatile int nsent = 0, timeout = 3;
16     struct udpiphdr *ui;

17     Signal(SIGALRM, sig_alrm);

18     if (sigsetjmp(jmpbuf, 1)) {
19         if (nsent >= 3)
20             err_quit("no response");
21         printf("timeout\n");
22         timeout *= 2;             /* backoff exponencial: 3, 6, 12 */
23     }
24     canjump = 1;                  /* siglongjmp agora está OK */

25     send_dns_query();
26     nsent++;

27     alarm(timeout);
28     ui = udp_read();
29     canjump = 0;
30     alarm(0);

31     if (ui->ui_sum == 0)
32         printf("UDP checksums off\n");
33     else
34         printf("UDP checksums on\n");
35     if (verbose)
36         printf("received UDP checksum = %x\n", ntohs(ui->ui_sum));
37 }
```

udpcksum/udpcksum.c

Figura 29.10 Função `test_udp`: envia consultas e lê respostas.

Variáveis `volatile`

50 Queremos que as duas variáveis automáticas, `nsent` e `timeout`, mantenham seus valores depois de um `siglongjmp` do handler de sinal de volta para essa função. Uma implementação pode restaurar variáveis automáticas de volta ao valor que tinham quando `sigsetjmp` foi chamado (página 178 de APUE), mas adicionar o qualificador `volatile` evita que isso ocorra.

Estabelecimento do handler de sinal e do buffer de salto

52–53 Um handler de sinal é estabelecido para `SIGALRM` e `sigsetjmp` estabelece um buffer de salto para `siglongjmp`. (Essas duas funções estão descritas em detalhes na Seção 10.15 de APUE.) O segundo argumento de valor 1 para `sigsetjmp` diz a esse comando para que salve a máscara de sinal corrente, pois chamaremos `siglongjmp` a partir do nosso handler de sinal.

Tratamento de `siglongjmp`

54–58 Esse código é executado somente quando `siglongjmp` é chamado a partir do nosso handler de sinal. Isto indica que um tempo-limite excedeu: enviamos uma solicitação e nunca recebemos uma resposta. Se já tivermos enviado três solicitações, terminamos. Caso contrário, imprimimos uma mensagem e multiplicamos o valor do tempo-limite por 2. Isso é um *backoff exponencial*, que descrevemos na Seção 22.5. O primeiro tempo-limite será de 3 segundos, então 6 e depois 12.

A razão de utilizarmos `sigsetjmp` e `siglongjmp` nesse exemplo, em vez de apenas capturar `EINTR` (como na Figura 14.1), é porque as funções de leitura da biblioteca de captu-

ra de pacote (que são chamadas por nossa função `udp_read`) reiniciam uma operação `read` quando `EINTR` é retornado. Como não queremos modificar as funções de biblioteca para retornar esse erro, nossa única solução é capturar o sinal `SIGALRM` e executar um comando goto não-local, retornando o controle para nosso código, em vez do código da biblioteca.

Envio da consulta de DNS e leitura da resposta

60–65 `send_dns_query` (Figura 29.12) envia uma consulta de DNS a um servidor de nomes. `udp_read` (Figura 29.15) lê a resposta. Chamamos `alarm` para evitar que a leitura seja bloqueada eternamente. Se o período de tempo-limite especificado (em segundos) expira, `SIGALRM` é gerado e nosso handler de sinal chama `siglongjmp`.

Examinando a soma de verificação UDP recebida

66–71 Se a soma de verificação UDP recebida for 0, o servidor não calculou nem enviou uma soma de verificação.

A Figura 29.11 mostra nosso handler de sinal, `sig_alrm`, que trata do sinal `SIGALRM`.

udpcksum/udpcksum.c

```
 1 #include   "udpcksum.h"
 2 #include   <setjmp.h>

 3 static sigjmp_buf jmpbuf;
 4 static int canjump;

 5 void
 6 sig_alrm(int signo)
 7 {
 8     if(canjump == 0)
 9        return;
10     siglongjmp(jmpbuf, 1);
11 }
```

udpcksum/udpcksum.c

Figura 29.11 Função `sig_alrm`: trata do sinal `SIGALRM`.

8–10 O flag `canjump` foi configurado na Figura 29.10, depois que o buffer de salto foi inicializado por `sigsetjmp`. Se o flag foi configurado, chamamos `siglongjmp`, que faz o fluxo de controle agir como se o sinal `sigsetjmp` na Figura 29.10 tivesse retornado com um valor igual a 1.

A Figura 29.12 mostra a função `send_dns_query`, que envia uma consulta de UDP para um servidor de DNS utilizando um soquete bruto. Essa função constrói os dados da aplicação, uma consulta de DNS.

udpcksum/senddnsquery-raw.c

```
 6 void
 7 send_dns_query(void)
 8 {
 9     size_t nbytes;
10     char  *buf, *ptr;

11     buf = Malloc(sizeof(struct udpiphdr) + 100);
12     ptr = buf + sizeof(struct udpiphdr);  /* deixa espaço para cabeçalhos IP/UDP */

13     *((uint16_t *) ptr) = htons(1234);   /* identificação */
14     ptr += 2;
15     *((uint16_t *) ptr) = htons(0x0100);         /* flags: recursividade
                                                     desejada */
```

Figura 29.12 Função `send_dns_query`: envia uma consulta para um servidor de DNS (*continua*).

```
16      ptr += 2;
17      *((uint16_t *) ptr) = htons(1);      /* # perguntas */
18      ptr += 2;
19      *((uint16_t *) ptr) = 0;     /* # RRs de resposta */
20      ptr += 2;
21      *((uint16_t *) ptr) = 0;     /* # RRs de autoridade */
22      ptr += 2;
23      *((uint16_t *) ptr) = 0;     /* # RRs adicionais */
24      ptr += 2;

25      memcpy(ptr, "\001a\014root-servers\003net\000", 20);
26      ptr += 20;
27      *((uint16_t *) ptr) = htons(1); /* tipo de consulta = A */
28      ptr += 2;
29      *((uint16_t *) ptr) = htons(1); /* classe de consulta = 1 (end IP) */
30      ptr += 2;

31      nbytes = (ptr - buf) - sizeof(struct udpiphdr);
32      udp_write(buf, nbytes);
33      if (verbose)
34          printf("sent: %d bytes of data\n", nbytes);
35  }
```

udpcksum/senddnsquery-raw.c

Figura 29.12 Função `send_dns_query`: envia uma consulta para um servidor de DNS (*continuação*).

Alocação do buffer e inicialização do ponteiro

11–12 Utilizamos `malloc` para alocar `buf` com espaço para um cabeçalho IP de 20 bytes, um cabeçalho UDP de 8 bytes e 100 bytes de dados de usuário. `ptr` é configurado para apontar para o primeiro byte de dados de usuário.

Construção da consulta de DNS

13–24 Entender os detalhes do datagrama UDP construído por essa função exige uma compreensão do formato da mensagem de DNS. Isso é encontrado na Seção 14.3 do TCPv1. Configuramos o campo de identificação como 1234, os flags como 0, o número de perguntas como 1 e o número de registros de recurso de resposta (RRs), o número de RRs de autoridade e o número de RRs adicionais como 0.

25–30 Formamos a única pergunta que segue a mensagem: uma consulta A para os endereços IP do host `a.root-servers.net`. Esse nome de domínio é armazenado em 20 bytes e consiste em quatro rótulos: o rótulo de 1 byte `a`, o rótulo de 12 bytes `root-servers` (lembre-se de que `\014` é uma constante de caractere octal), o rótulo de 3 bytes `net` e o rótulo-raiz, cujo comprimento é 0. O tipo de consulta (chamada de consulta A) e a classe de consulta também são 1.

Escrita do datagrama UDP

31–32 Essa mensagem consiste em 36 bytes de dados de usuário (oito campos de 2 bytes e o nome de domínio de 20 bytes), mas calculamos o comprimento da mensagem a partir do ponteiro corrente dentro do buffer subtraindo do mesmo o início do buffer, para não termos que alterar uma constante, caso alteremos o formato da mensagem que estamos enviando. Chamamos nossa função `udp_write` para construir os cabeçalhos UDP e IP e gravamos o datagrama de IP em nosso soquete bruto.

A Figura 29.13 mostra a função `open_output` para utilização com soquetes brutos.

Declaração do descritor de soquete bruto

2 Declaramos uma variável global para conter o descritor do soquete bruto.

udpcksum/udpwrite.c

```
 2 int    rawfd;            /* soquete bruto para gravação */

 3 void
 4 open_output(void)
 5 {
 6     int    on = 1;
 7     /*
 8      * Precisamos de um soquete bruto para gravarmos nossos próprios
          datagramas IP.
 9      * O processo deve ter privilégios de superusuário para criar esse
          soquete.
10      * Também deve configurar IP_HDRINCL, para poder gravar nossos
          próprios cabeçalhos       IP.
11      */

12     rawfd = Socket(dest->sa_family, SOCK_RAW, 0);

13     Setsockopt(rawfd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));
14 }
```

udpcksum/udpwrite.c

Figura 29.13 Função `open_output`: prepara um soquete bruto.

Criação do soquete bruto e ativação do `IP_HDRINCL`

7–13 Criamos um soquete bruto e ativamos a opção de soquete `IP_HDRINCL`. Essa opção nos permite gravar datagramas IP completos, incluindo o cabeçalho IP.

A Figura 29.14 mostra nossa função, `udp_write`, que constrói os cabeçalhos IP e UDP e, então, grava o datagrama no soquete bruto.

udpcksum/udpwrite.c

```
19 void
20 udp_write(char *buf, int userlen)
21 {
22     struct udpiphdr *ui;
23     struct ip *ip;

24         /* preenche e faz a soma de verificação do cabeçalho UDP */
25     ip = (struct ip *) buf;
26     ui = (struct udpiphdr *) buf;
27     bzero(ui, sizeof(*ui));
28             /* adiciona 8 em userlen para comprimento do pseudocabeçalho */
29     ui->ui_len = htons((uint16_t) (sizeof(struct udphdr) + userlen));
30             /* então, adiciona 28 para o comprimento do datagrama de IP */
31     userlen += sizeof(struct udpiphdr);

32     ui->ui_pr = IPPROTO_UDP;
33     ui->ui_src.s_addr = ((struct sockaddr_in *) local)->sin_addr.s_addr;
34     ui->ui_dst.s_addr = ((struct sockaddr_in *) dest)->sin_addr.s_addr;
35     ui->ui_sport = ((struct sockaddr_in *) local)->sin_port;
36     ui->ui_dport = ((struct sockaddr_in *) dest)->sin_port;
37     ui->ui_ulen = ui->ui_len;
38     if (zerosum == 0) {
39 #if 1                              /* muda se for 0 para Solaris 2.x, x < 6 */
40         if ( (ui->ui_sum = in_cksum((u_int16_t *) ui, userlen)) == 0)
41             ui->ui_sum = 0xffff;
42 #else
43         ui->ui_sum = ui->ui_len;
44 #endif
45     }
```

Figura 29.14 Função `udp_write`: constrói cabeçalhos UDP e IP e grava datagrama de IP no soquete bruto (*continua*).

```
46          /* preenche o restante do cabeçalho IP; */
47          /* ip_output() calcula e armazena a soma de verificação do
            cabeçalho IP */
48      ip->ip_v = IPVERSION;
49      ip->ip_hl = sizeof(struct ip) >> 2;
50      ip->ip_tos = 0;
51 #if defined(linux) || defined(__OpenBSD__)
52      ip->ip_len = htons(userlen);      /* ordem de byte de rede */
53 #else
54      ip->ip_len = userlen;         /* ordem de byte de host */
55 #endif
56      ip->ip_id = 0;                /* permite que IP configure isso */
57      ip->ip_off = 0;               /* deslocamento de flag, flags MF e DF */
58      ip->ip_ttl = TTL_OUT;

59      Sendto(rawfd, buf, userlen, 0, dest, destlen);
60  }
```

udpcksum/udpwrite.c

Figura 29.14 Função `udp_write`: constrói cabeçalhos UDP e IP e grava datagrama de IP no soquete bruto (*continuação*).

Inicialização dos ponteiros de cabeçalho de pacote

24–26 `ip` aponta para o começo do cabeçalho IP (uma estrutura `ip`) e `ui` aponta para o mesmo local, mas a estrutura `udpiphdr` são os cabeçalhos IP e UDP combinados.

Zerando o cabeçalho

27 Configuramos a área de cabeçalho explicitamente com zeros, para evitar considerar na soma de verificação quaisquer dados que possam ter restado no buffer.

> Versões anteriores desse código configuram explicitamente cada elemento de `struct udpiphdr` como zero; entretanto, essa estrutura contém alguns detalhes de implementação, de modo que ela pode ser diferente de um sistema para outro. Esse é um problema típico de portabilidade, ao se construir cabeçalhos explicitamente.

Atualização dos comprimentos

28–31 `ui_len` é o comprimento de UDP: o número de bytes de dados de usuário, mais o tamanho do cabeçalho UDP (8 bytes). `userlen` (o número de bytes de dados de usuário que segue o cabeçalho UDP) é incrementado por 28 (20 bytes para o cabeçalho IP e 8 bytes para o cabeçalho UDP) para refletir o tamanho total do datagrama de IP.

Preenchimento do cabeçalho UDP e cálculo da soma de verificação UDP

32–45 Quando a soma de verificação UDP é calculada, isso inclui não apenas o cabeçalho UDP e dados de UDP, mas também campos do cabeçalho IP. Esses campos adicionais do cabeçalho IP formam o que é chamado de *pseudocabeçalho*. A inclusão do pseudocabeçalho fornece a verificação adicional de que, se a soma de verificação está correta, então o datagrama foi entregue para o host correto e para o código de protocolo correto. Essas instruções inicializam os campos no cabeçalho IP que formam o pseudocabeçalho. O código é um tanto obtuso, mas está explicado na Seção 23.6 do TCPv2. O resultado é o armazenamento da soma de verificação UDP no membro `ui_sum`, caso o flag `zerosum` (o argumento de linha de comando `–0`) não seja configurado.

Se a soma de verificação calculada é 0, então o valor `0xffff` é armazenado. Em aritmética de complemento de um, os dois valores são os mesmos, mas o UDP configura a soma de verificação como 0 para indicar que o emissor não armazenou uma soma de verificação UDP. Observe que não testamos se uma soma de verificação calculada é igual a 0 na Figura 28.14, porque a soma de verificação ICMPv4 é exigida: o valor 0 não indica a ausência de uma soma de verificação.

Observamos que o Solaris 2.x, para x < 6, tem um bug com relação às somas de verificação para segmentos TCP ou datagramas UDP enviados em um soquete bruto com a opção de soquete `IP_HDRINCL` configurada. O kernel calcula a soma de verificação e devemos configurar o campo `ui_sum` com o comprimento de UDP.

Preenchimento do cabeçalho IP

46–59 Como configuramos a opção de soquete `IP_HDRINCL`, devemos preencher a maioria dos campos no cabeçalho IP. (A Seção 28.3 discute essas gravações em um soquete bruto, quando essa opção de soquete é configurada.) Configuramos o campo de identificação como 0 (`ip_id`), o que diz ao IP para que configure esse campo. O IP também calcula a soma de verificação de cabeçalho IP. `sendto` grava o datagrama IP.

Observe que configuramos o campo `ip_len` na ordem de byte do host ou de rede, dependendo do SO que estejamos utilizando. Esse é um problema típico de portabilidade ao se utilizar soquetes brutos.

A próxima função é `udp_read`, mostrada na Figura 29.15, que foi chamada na Figura 29.10.

udpcksum/udpread.c

```
 7 struct udpiphdr *
 8 udp_read(void)
 9 {
10     int    len;
11     char  *ptr;
12     struct ether_header *eptr;

13     for ( ; ; ) {
14         ptr = next_pcap(&len);

15         switch (datalink) {
16         case DLT_NULL:            /* cabeçalho de loopback = 4 bytes */
17             return (udp_check(ptr + 4, len - 4));

18         case DLT_EN10MB:
19             eptr = (struct ether_header *) ptr;
20             if (ntohs(eptr->ether_type) != ETHERTYPE_IP)
21                 err_quit("Ethernet type %x not IP", ntohs(eptr->ether_type));
22             return (udp_check(ptr + 14, len - 14));

23         case DLT_SLIP:            /* cabeçalho SLIP = 24 bytes */
24             return (udp_check(ptr + 24, len - 24));

25         case DLT_PPP:             /* cabeçalho PPP = 24 bytes */
26             return (udp_check(ptr + 24, len - 24));

27         default:
28             err_quit("unsupported datalink (%d)", datalink);
29         }
30     }
31 }
```

udpcksum/udpread.c

Figura 29.15 Função `udp_read`: lê o próximo pacote do dispositivo de captura de pacote.

14–29 Nossa função `next_pcap` (Figura 29.16) retorna o próximo pacote do dispositivo de captura de pacote. Como os cabeçalhos de enlace de dados diferem dependendo do tipo de dispositivo, desviamos com base no valor retornado pela função `pcap_datalink`.

Esses deslocamentos mágicos de 4, 14 e 24 são mostrados na Figura 31.9 de TCPv2. Os deslocamentos de 24 bytes mostrados para SLIP e PPP servem para o BSD/OS 2.1.

Apesar de ter o qualificador "10MB" no nome `DLT_EN10MB`, esse tipo de enlace de dados também é utilizado para Ethernet de 100 Mbits/seg.

Nossa função udp_check (Figura 29.19) examina o pacote e verifica os campos nos cabeçalhos IP e UDP.

A Figura 29.16 mostra a função next_pcap, que retorna o próximo pacote do dispositivo de captura de pacote.

udpcksum/pcap.c

```
38 char *
39 next_pcap(int *len)
40 {
41     char   *ptr;
42     struct pcap_pkthdr hdr;

43         /* mantém-se em loop até que o pacote esteja pronto */
44     while ( (ptr = (char *) pcap_next(pd, &hdr)) == NULL) ;

45     *len = hdr.caplen;           /* comprimento capturado */
46     return (ptr);
47 }
```

udpcksum/pcap.c

Figura 29.16 Função next_pcap: retorna o próximo pacote.

43–44 Chamamos a função de biblioteca pcap_next, que retorna o próximo pacote ou NULL, caso um tempo-limite ocorra. Se o tempo-limite ocorrer, simplesmente fazemos um loop e chamamos pcap_next novamente. Um ponteiro para o pacote é o valor de retorno da função e o segundo argumento aponta para uma estrutura pcap_pkthdr, que também é preenchida no retorno.

```
struct pcap_pkthdr {
    struct timeval ts;      /* indicação de tempo */
    bpf_u_int32    caplen;  /* comprimento da parte capturada */
    bpf_u_int32    len;     /* comprimento desse pacote (fora do cabo) */
};
```

A indicação de tempo é a de quando o dispositivo de captura de pacote lê o pacote, em oposição à sua entrega real para o processo, a qual poderia ser feita posteriormente. caplen é o volume de dados que foram capturados (lembre-se de que configuramos nossa variável snaplen como 200 na Figura 29.6 e, então, esse foi o segundo argumento de pcap_open_live na Figura 29.9). O propósito do recurso de captura de pacote é capturar os cabeçalhos de pacote e não todos os dados em cada pacote. len é o comprimento total do pacote no cabo. caplen sempre será menor ou igual a len.

45–46 O comprimento capturado é retornado pelo argumento de ponteiro e o valor de retorno da função é o ponteiro para o pacote. Lembre-se de que o "ponteiro para o pacote" aponta para o cabeçalho de enlace de dados, que é o cabeçalho Ethernet de 14 bytes, no caso de um quadro Ethernet, ou um cabeçalho de pseudo-enlace de 4 bytes, no caso da interface de loopback.

Se examinarmos a implementação de pcap_next na biblioteca, veremos que ela mostra a divisão de trabalho entre as diferentes funções. Mostramos isso na Figura 29.17. Nossa aplicação chama as funções pcap_ e algumas dessas funções são independentes de dispositivo, enquanto outras são dependentes do tipo de dispositivo de captura de pacote. Por exemplo, mostramos que a implementação BPF chama read, enquanto a DLPI chama getmsg e a Linux chama recvfrom.

Nossa função udp_check verifica numerosos campos nos cabeçalhos IP e UDP. Isso está mostrado na Figura 29.19. Devemos fazer essas verificações porque, quando o pacote é passado a nós pelo dispositivo de captura de pacote, a camada IP ainda não o viu. Isso difere de um soquete bruto.

44–61 O comprimento do pacote deve incluir pelo menos os cabeçalhos IP e UDP. A versão de IP é verificada, junto com o comprimento do cabeçalho IP e a soma de verificação do cabeçalho

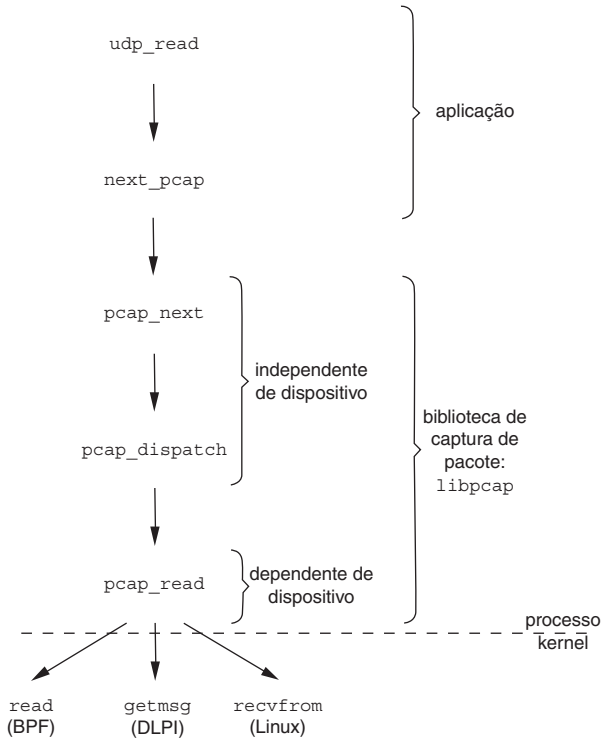


Figura 29.17 Organização das chamadas de função para ler da biblioteca de captura de pacote.

IP. Se o campo de protocolo indica um datagrama UDP, a função retorna o ponteiro para o cabeçalho IP/UDP combinado. Caso contrário, o programa termina, pois o filtro de captura de pacote que especificamos em nossa chamada a `pcap_setfilter` na Figura 29.9 não deve retornar nenhum outro tipo de pacote.

A função `cleanup` mostrada na Figura 29.18 é chamada pela função `main` imediatamente antes que o programa termine e também como handler de sinal, caso o usuário cancele o programa (Figura 29.8).

udpcksum/cleanup.c

```
 2 void
 3 cleanup(int signo)
 4 {
 5     struct pcap_stat stat;

 6     putc('\n', stdout);

 7     if(verbose) {
 8         if (pcap_stats(pd, &stat) < 0)
 9             err_quit("pcap_stats: %s\n", pcap_geterr(pd));
10         printf("%d packets received by filter\n", stat.ps_recv);
11         printf("%d packets dropped by kernel\n", stat.ps_drop);
12     }

13     exit(0);
14 }
```

udpcksum/cleanup.c

Figura 29.18 Função `cleanup`.

udpcksum/udpread.c

```
38 struct udpiphdr *
39 udp_check(char *ptr, int len)
40 {
41     int    hlen;
42     struct ip *ip;
43     struct udpiphdr *ui;

44     if (len < sizeof(struct ip) + sizeof(struct udphdr))
45         err_quit("len = %d", len);

46         /* verificação mínima de cabeçalho IP */
47     ip = (struct ip *) ptr;
48     if (ip->ip_v != IPVERSION)
49         err_quit("ip_v = %d", ip->ip_v);
50     hlen = ip->ip_hl << 2;
51     if (hlen < sizeof(struct ip))
52         err_quit("ip_hl = %d", ip->ip_hl);
53     if (len < hlen + sizeof(struct udphdr))
54         err_quit("len = %d, hlen = %d", len, hlen);

55     if ( (ip->ip_sum = in_cksum((uint16_t *) ip, hlen)) != 0)
56         err_quit("ip checksum error");

57     if (ip->ip_p == IPPROTO_UDP) {
58         ui = (struct udpiphdr *) ip;
59         return (ui);
60     } else
61         err_quit("not a UDP packet");
62 }
```

udpcksum/udpread.c

Figura 29.19 Função `udp_check`: verifica campos em cabeçalhos IP e UDP.

Busca e impressão das estatísticas de captura de pacote

7–12 `pcap_stats` busca as estatísticas de captura de pacote: o número total de pacotes recebidos pelo filtro e o número de pacotes omitidos pelo kernel.

Exemplo

Primeiro, executamos nosso programa com a opção de linha de comando `–0` para verificar se o servidor de nomes responde aos datagramas que chegam sem soma de verificação. Também especificamos o flag `–v`.

```
macosx # udpcksum -i en1 -0 -v bridget.rudoff.com domain
device = en1
localnet = 172.24.37.64, netmask = 255.255.255.224
cmd = udp and src host 206.168.112.96 and src port 53
datalink = 1
sent: 36 bytes of data
UDP checksums on
received UDP checksum = 9d15

3 packets received by filter
0 packets dropped by kernel
```

Em seguida, executamos nosso programa para um servidor de nomes local (nosso sistema `freebsd4`), que não tem somas de verificação UDP ativadas. (Observe que é cada vez mais raro encontrar um servidor de nomes sem somas de verificação UDP ativadas.)

```
macosx # udpcksum -i en1 -v freebsd4.unpbook.com domain
device = en1
localnet = 172.24.37.64, netmask = 255.255.255.224
```

```
cmd = udp and src host 172.24.37.94 and src port 53
datalink = 1
sent: 36 bytes of data
UDP checksums off
received UDP checksum = 0

3 packets received by filter
0 packets dropped by kernel
```

Funções de saída `libnet`

Agora, mostraremos versões de `open_output` e `send_dns_query` que utilizam `libnet` em vez de soquetes brutos. Conforme veremos, `libnet` cuida de muitos detalhes, incluindo os problemas de portabilidade com somas de verificação e ordem de byte de cabeçalho IP mencionados. A função `open_output` para `libnet` aparece na Figura 29.20.

udpcksum/senddnsquery-libnet.c

```
 7 static libnet_t *l;              /* descritor de libnet */

 8 void
 9 open_output(void)
10 {
11     char    errbuf[LIBNET_ERRBUF_SIZE];

12     /* Inicializa libnet com um soquete IPv4 bruto */
13     l = libnet_init(LIBNET_RAW4, NULL, errbuf);
14     if (l == NULL) {
15         err_quit("Can't initialize libnet: %s", errbuf);
16     }
17 }
```

udpcksum/senddnsquery-libnet.c

Figura 29.20 Função `open_output`: prepara para utilizar `libnet`.

Declaração do descritor `libnet`

7 `libnet` utiliza um tipo opaco, `libnet_t`, como um vínculo com a biblioteca. A função `libnet_init` retorna um ponteiro `libnet_t`, o qual é então passado a outras funções `libnet` para indicar qual instância é desejada. Dessa maneira, ela é semelhante aos descritores de soquete e pcap.

Inicialização de `libnet`

12–16 Chamamos a função `libnet_init`, solicitando para que abra um soquete IPv4 bruto, fornecendo `LIBNET_RAW4` como seu primeiro argumento. Se um erro é encontrado, `libnet_init` retorna um erro em seu argumento *errbuf*, o qual imprimimos caso `libnet_init` retorne `NULL`.

A função `send_dns_query` para `libnet` aparece na Figura 29.21. Compare-a com as funções `send_dns_query` (Figura 29.12) e `udp_write` (Figura 29.14) para soquetes brutos.

Construção da consulta de DNS

25–32 Construímos primeiro a parte da consulta do pacote de DNS, assim como nas linhas 25 a 30 da Figura 29.12.

34–40 Então, chamamos a função `libnet_build_dnsv4`, que aceita cada campo do pacote de DNS como um argumento de função separado. Só precisamos conhecer o layout da parte da consulta; os detalhes de como montar o cabeçalho de pacote DNS são resolvidos automaticamente.

udpcksum/senddnsquery-libnet.c

```
18 void
19 send_dns_query(void)
20 {
21     char   qbuf[24], *ptr;
22     u_int16_t one;
23     int    packet_size = LIBNET_UDP_H + LIBNET_DNSV4_H + 24;
24     static libnet_ptag_t ip_tag, udp_tag, dns_tag;

25     /* constrói a parte da consulta de pacote de DNS */
26     ptr = qbuf;
27     memcpy(ptr, "\001a\014root-servers\003net\000", 20);
28     ptr += 20;
29     one = htons(1);
30     memcpy(ptr, &one, 2);         /* tipo de consulta = A */
31     ptr += 2;
32     memcpy(ptr, &one, 2);         /* classe de consulta = 1 (end IP) */

33     /* constrói o pacote de DNS */
34     dns_tag = libnet_build_dnsv4(1234 /* identificação */ ,
35                                  0x0100 /* flags: recursividade desejada */ ,
36                                  1 /* # perguntas */ , 0 /* # RRs de
                                    resposta */ ,
37                                  0 /* # RRs de autoridade */ ,
38                                  0 /* # RRs adicionais */ ,
39                                  qbuf /* consulta */ ,
40                                  24 /* comprimento da consulta */ , l,
                                    dns_tag);
41     /* constrói o cabeçalho UDP */
42     udp_tag = libnet_build_udp(((struct sockaddr_in *) local)->
43                                sin_port /* porta de origem */ ,
44                                ((struct sockaddr_in *) dest)->
45                                sin_port /* porta de dest */ ,
46                                packet_size /* comprimento */ , 0 /* soma
                                  de verif. */ ,
47                                NULL /* payload */ , 0 /* comprimento do
                                  payload */ ,
48                                l,udp_tag);
49     /* Como especificamos a soma de verificação como 0, libnet calculará a
        soma de */
50     /* verificação UDP automaticamente. Desative, caso o usuário não a
        queira. */
51     if (zerosum)
52         if (libnet_toggle_checksum(l, udp_tag, LIBNET_OFF) < 0)
53             err_quit("turning off checksums: %s\n", libnet_geterror(l));
54     /* constrói o cabeçalho IP */
55     ip_tag = libnet_build_ipv4(packet_size + LIBNET_IPV4_H /* len */,
56             0 /* tos */, 0 /* ID de IP */, 0 /* fragmento */,
57             TTL_OUT /* ttl */, IPPROTO_UDP /* protocolo */,
58             0 /* soma de verificação */,
59             ((struct sockaddr_in *) local)->sin_addr.s_addr /* origem */,
60             ((struct sockaddr_in *) dest)->sin_addr.s_addr /* dest */,
61             NULL /* payload */, 0 /* comprimento do payload */, l, ip_tag);

62     if (libnet_write(l) < 0) {
63         err_quit("libnet_write: %s\n", libnet_geterror(l));
64     }
65     if (verbose)
66         printf("sent: %d bytes of data\n", packet_size);
67 }
```

udpcksum/senddnsquery-libnet.c

Figura 29.21 Função `send_dns_query` utilizando `libnet`: envia uma consulta a um servidor de DNS.

Preenchimento do cabeçalho UDP e planejamento do cálculo da soma de verificação UDP

42–48 De maneira semelhante, construímos o cabeçalho UDP chamando a função `libnet_build_udp`. Isso também aceita cada campo de cabeçalho como um argumento de função separado. Ao se passar um campo de soma de verificação como 0, `libnet` calcula automaticamente a soma de verificação para ele. Isso é comparável às linhas 29 a 45 da Figura 29.14.

49–52 Se o usuário solicitou que a soma de verificação não fosse calculada, devemos desativar o cálculo da soma de verificação especificamente.

Preenchimento do cabeçalho IP

53–65 Para completar o pacote, construímos o cabeçalho IPv4 utilizando a função `libnet_build_ipv4`. Assim como acontece com outras funções `libnet_build`, fornecemos somente o conteúdo do campo e a função `libnet` monta o cabeçalho para nós. Isso é comparável às linhas 46 a 58 da Figura 29.14.

> Observe que `libnet` cuida automaticamente do fato de o campo `ip_len` estar em ordem de byte de rede ou não. Esse é um exemplo de melhora de portabilidade obtida com o uso de `libnet`.

Escrita do datagrama UDP

66–70 Chamamos a função `libnet_write` para escrever o datagrama montado na rede.

Observe que a versão `libnet` de `send_dns_query` tem somente 67 linhas, enquanto a versão de soquete bruto (`send_dns_query` e `udp_write` combinadas) tem 96 linhas e contém pelo menos duas "pegadinhas" de portabilidade.

29.8 Resumo

Com soquetes brutos, temos a capacidade de ler e gravar datagramas IP que o kernel não entende e, com o acesso à camada de enlace de dados, podemos estender essa capacidade para ler e gravar *qualquer* tipo de quadro de enlace de dados e não apenas datagramas IP. `tcpdump` é provavelmente o programa que acessa a camada de enlace de dados diretamente mais utilizado.

Diferentes sistemas operacionais possuem diferentes maneiras de acessar a camada de enlace de dados. Vimos o BPF derivado do Berkeley, a DLPI do SVR4 e o `SOCK_PACKET` do Linux. Mas podemos ignorar todas as suas diferenças e ainda escrever código portável utilizando a biblioteca de captura de pacote disponível livremente, `libpcap`.

Escrever datagramas brutos pode ser diferente em diferentes sistemas. A biblioteca `libnet` disponível livremente oculta essas diferenças e fornece uma interface para gerar saída por meio de soquetes brutos e diretamente no enlace de dados.

Exercícios

29.1 Qual é o propósito do flag `canjump` na Figura 29.11?

29.2 Em nosso programa `udpcksum`, as respostas de erro comuns são "porta inacessível" (o destino não está executando um servidor de nomes) ou "host inacessível" do ICMP. Em qualquer caso, não precisamos esperar por um tempo-limite de nossa função `udp_read`, na Figura 29.10, porque o erro ICMP é basicamente uma resposta a nossa consulta de DNS. Modifique o programa para capturar esses erros de ICMP.

Capítulo **30**

Alternativas de Projeto de Cliente/Servidor

30.1 Visão geral

Temos várias escolhas para o tipo de controle de processos a utilizar ao escrever um servidor Unix:

- Nosso primeiro servidor, Figura 1.9, foi um *servidor iterativo*, o qual nem sempre é recomendado, pois não pode processar um cliente pendente até que tenha atendido completamente o cliente atual.
- A Figura 5.2 foi nosso primeiro *servidor concorrente* e chamava `fork` para gerar um processo-filho para cada cliente. Tradicionalmente, a maioria dos servidores Unix cai nessa categoria.
- Na Seção 6.8, desenvolvemos uma versão diferente de nosso servidor TCP, consistindo em um único processo que utiliza `select` para tratar de qualquer número de clientes.
- Na Figura 26.3, modificamos nosso servidor concorrente para criar um thread por cliente, em vez de um processo por cliente.

Há duas outras modificações no projeto de um servidor concorrente, que veremos neste capítulo:

- *Pré-bifurcação (preforking)* faz o servidor chamar `fork` ao iniciar, criando um pool de processos-filhos. Um processo do pool correntemente disponível trata de cada solicitação de cliente.
- *Pré-threading (prethreading)* faz o servidor criar um pool de threads disponíveis ao iniciar e um thread desse pool trata de cada cliente.

Há muitos detalhes na bifurcação prévia e no pré-*threading*, que examinaremos neste capítulo: E se não houver processos ou threads suficientes no pool? E se houver processos ou threads demais no pool? Como o pai e seus filhos ou os threads podem ser sincronizados uns com os outros?

Em geral, os clientes são mais fáceis de escrever do que os servidores, porque há menos controle de processos em um cliente. Contudo, já examinamos várias maneiras de escrever nosso cliente de eco simples e as resumimos na Seção 30.2.

Neste capítulo, veremos nove projetos diferentes de servidor e executaremos cada servidor para o mesmo cliente. Nosso cenário de cliente/servidor é típico da Web: o cliente envia uma pequena solicitação para o servidor e este responde com dados de volta para o cliente. Já discutimos alguns dos servidores em detalhes (por exemplo, o servidor concorrente com um `fork` por cliente), enquanto os servidores de bifurcação prévia e de pré-threading são novidades e, portanto, discutidos em detalhes neste capítulo.

Executaremos várias instâncias de um cliente para cada servidor, medindo o tempo de CPU exigido para atender um número fixo de solicitações de cliente. Em vez de dispersar todas as nossas medidas de tempo de CPU por todo o capítulo, as resumimos na Figura 30.1 e nos referiremos a ela ao longo do capítulo. Observamos que os valores nessa figura medem o tempo de CPU exigido *somente para controle de processos* e o servidor iterativo é nossa linha de base que subtraímos do tempo de CPU real, porque um servidor iterativo não tem nenhum overhead de controle de processos. Incluímos o tempo de linha de base de 0.0 nessa figura, para reiterar esse ponto. Utilizamos o termo *tempo de CPU para controle de processos* neste capítulo, para denotar essa diferença da linha de base para determinado sistema.

Todas essas medidas de tempo do servidor foram obtidas por meio da execução do cliente mostrado na Figura 30.3 em dois hosts diferentes, na mesma sub-rede do servidor. Para todos os testes, os dois clientes geraram cinco filhos para criar cinco conexões simultâneas com o servidor, para um máximo de 10 conexões simultâneas no servidor, a qualquer momento. Cada cliente solicitou 4.000 bytes do servidor através da conexão. Para esses testes envolvendo um servidor pré-bifurcado ou um servidor com pré-threading, o servidor criou 15 filhos ou 15 threads quando iniciou.

Alguns projetos de servidor envolvem a criação de um pool de processos-filhos ou um pool de threads. Um item a considerar nesses casos é a distribuição das solicitações de cliente para o pool disponível. A Figura 30.2 resume essas distribuições e discutiremos cada coluna na seção apropriada.

Linha	Descrição do servidor	Tempo de CPU para controle de processos, segundos (diferença em relação à linha de base)
0	Servidor iterativo (medida da linha de base; nenhum controle de processos)	0,0
1	Servidor concorrente, um `fork` por solicitação de cliente	20,90
2	Bifurcação prévia com cada filho chamando `accept`	1,80
3	Bifurcação prévia com bloqueio de arquivo para proteger `accept`	2,07
4	Bifurcação prévia com bloqueio mutex de thread para proteger `accept`	1,75
5	Bifurcação prévia com o pai passando o descritor de soquete para o filho	2,58
6	Servidor concorrente, cria um thread para cada solicitação de cliente	0,99
7	Pré-*threading* com bloqueio mutex para proteger `accept`	1,93
8	Pré-*threading* com o thread principal chamando `accept`	2,05

Figura 30.1 Comparações de sincronização dos vários servidores discutidos neste capítulo.

30.2 Alternativas para clientes TCP

Já examinamos vários projetos de cliente, mas vale resumir seus pontos fortes e fracos:

- A Figura 5.5 era o cliente TCP básico. Há dois problemas nesse programa. Primeiro, enquanto ele está bloqueado esperando pela entrada do usuário, não vê eventos de rede, como o peer fechando a conexão. Além disso, opera em um modo de parada e espera, tornando-se ineficiente para processamento em lotes.

nº do filho ou nº do thread	nº de clientes atendidos			
	Pré-bifurcado, nenhum bloqueio em torno de `accept` (linha 2)	Pré-bifurcado, bloqueio de arquivo em torno de `accept` (linha 3)	Pré-bifurcado, passagem de descritor (linha 5)	Pré-threading, bloqueio de thread em torno de `accept` (linha 7)
0	333	347	1006	333
1	340	328	950	323
2	335	332	720	333
3	335	335	582	328
4	332	338	485	329
5	331	340	457	322
6	333	335	385	324
7	333	343	250	360
8	332	324	105	341
9	331	315	32	348
10	334	326	14	358
11	333	340	9	331
12	334	330	4	321
13	332	331	1	329
14	332	336	0	320
	5000	5000	5000	5000

Figura 30.2 Número de clientes ou threads atendidos por cada um dos 15 filhos ou threads.

- A Figura 6.9 foi a próxima iteração e, utilizando `select`, o cliente era notificado de eventos de rede, enquanto esperava pela entrada do usuário. Entretanto, esse programa não tratava do modo de lote corretamente. A Figura 6.13 corrigiu esse problema, utilizando a função `shutdown`.
- A Figura 16.3 iniciou a apresentação de nosso cliente utilizando E/S não-bloqueadora.
- O primeiro de nossos clientes que foi além do projeto de processo único ou thread único foi a Figura 16.10, que utilizava `fork`, com um processo tratando dos dados do cliente para o servidor e o outro tratando dos dados do servidor para o cliente.
- A Figura 26.2 utilizou dois threads, em vez de dois processos.

No fim da Seção 16.2, resumimos as diferenças das medidas de tempo entre essas várias versões. Conforme notamos lá, embora a versão não-bloqueadora de E/S fosse a mais rápida, o código era mais complexo e utilizar ou dois processos ou dois threads simplifica-o.

30.3 Cliente TCP de teste

A Figura 30.3 mostra o cliente que utilizaremos para testar todas as variações de nosso servidor.

10–12 Sempre que executamos o cliente, especificamos o nome de host ou endereço IP do servidor, a porta do servidor, o número de filhos do cliente a bifurcar (permitindo-nos iniciar múltiplas conexões com o mesmo servidor concorrentemente), o número de solicitações que cada filho deve enviar para o servidor e o número de bytes a solicitar que servidor retorne a cada vez.

17–30 O pai chama `fork` para cada filho e cada filho estabelece o número especificado de conexões com o servidor. Em cada conexão, o filho envia uma linha especificando o número de bytes para o servidor retornar e, então, lê esse volume de dados do servidor. O pai somente espera que todos os filhos terminem. Observe que o cliente fecha cada conexão TCP; portanto, o estado TIME_WAIT do TCP ocorre no cliente e não no servidor. Essa é uma diferença entre nossas conexões cliente/servidor e as HTTP normais.

Quando medimos os vários servidores neste capítulo, executamos o cliente como

```
% client 192.168.1.20 8888 5 500 4000
```

server/client.c

```
 1 #include   "unp.h"
 2 #define MAXN  16384      /* nº máx de bytes a solicitar do servidor */
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int    i, j, fd, nchildren, nloops, nbytes;
 7     pid_t  pid;
 8     ssize_t n;
 9     char     request[MAXLINE], reply[MAXN];
10     if (argc != 6)
11         err_quit("usage: client <hostname or IPaddr> <port> <#children> "
12                  "<#loops/child> <#bytes/request>");
13     nchildren = atoi(argv[3]);
14     nloops = atoi(argv[4]);
15     nbytes = atoi(argv[5]);
16     snprintf(request, sizeof(request), "%d\n", nbytes); /* nova linha no
                                                            fim */
17     for (i = 0; i < nchildren; i++) {
18         if ( (pid = Fork()) == 0) { /* filho */
19             for (j = 0; j < nloops; j++) {
20                 fd= Tcp_connect(argv[1], argv[2]);
21                 Write(fd, request, strlen(request));
22                 if ( (n = Readn(fd, reply, nbytes)) != nbytes)
23                     err_quit("server returned %d bytes", n);
24                 Close(fd);      /* TIME_WAIT no cliente, não no servidor */
25             }
26             printf("child %d done\n", i);
27             exit(0);
28         }
29         /* o pai volta ao loop para bifurcar, fork(), novamente */
30     }
31     while (wait(NULL) > 0)     /* agora o pai espera por todos os filhos */
32         ;
33     if (errno != ECHILD)
34         err_sys("wait error");
35     exit(0);
36 }
```

server/client.c

Figura 30.3 Programa cliente TCP para testar nossos vários servidores.

Isso cria 2.500 conexões TCP com o servidor: 500 conexões de cada um dos cinco filhos. Em cada conexão, 5 bytes são enviados do cliente para o servidor ("`4000\n`") e 4.000 bytes são transferidos do servidor de volta para o cliente. Executamos o cliente a partir de dois hosts diferentes para o mesmo servidor, fornecendo um total de 5.000 conexões TCP, com um máximo de 10 conexões simultâneas no servidor em dado momento.

> Existem benchmarks sofisticados para testar vários servidores da Web. Um deles é chamado WebStone e está disponível no endereço `http://www.mindcraft.com/webstone`. Entretanto, não precisamos de nada sofisticado assim para fazer algumas comparações gerais das várias alternativas de projeto de servidor que examinaremos neste capítulo.

Apresentaremos agora os nove diferentes projetos de servidor.

30.4 Servidor TCP iterativo

Um servidor TCP iterativo processa completamente a solicitação de cada cliente, antes de passar para o próximo cliente. Os servidores TCP iterativos são raros, mas mostramos um na Figura 1.9: um servidor de data/hora simples.

Entretanto, temos uma utilização para um servidor iterativo, na comparação dos vários servidores deste capítulo. Se executarmos o cliente como

```
% client 192.168.1.20 8888 1 5000 4000
```

para um servidor iterativo, obteremos o mesmo número de conexões TCP (5.000) e o mesmo volume de dados transferidos através de cada conexão. Mas, como o servidor é iterativo, *não há nenhum controle de processos* realizado por ele. Isso nos fornece uma medida de referência (linha de base) do tempo de CPU exigido para tratar desse número de clientes, que podemos então subtrair de todas as outras medidas de servidor. Da perspectiva do controle de processos, o servidor iterativo é o mais rápido possível, porque não realiza nenhum controle de processos. Então, comparamos as *diferenças* a partir dessa linha de base na Figura 30.1.

Não mostramos nosso servidor iterativo, pois ele é uma modificação simples do servidor concorrente que apresentaremos na próxima seção.

30.5 Servidor TCP concorrente, um filho por cliente

Tradicionalmente, um servidor TCP concorrente chama `fork` para gerar um filho para tratar de cada cliente. Isso permite que o servidor trate de inúmeros clientes ao mesmo tempo, um cliente por processo. O único limite sobre o número de clientes é o limite do SO sobre o número de processos-filhos para o ID de usuário sob o qual o servidor está executando. A Figura 5.12 é um exemplo de servidor concorrente e a maioria dos servidores TCP é escrita dessa maneira.

O problema desses servidores concorrentes é a quantidade de tempo de CPU exigida para bifurcar (`fork`) um filho para cada cliente. No final da década de 1980, quando um servidor ocupado tratava de centenas ou talvez até de milhares de clientes por dia, isso era aceitável. Mas a explosão da Web mudou essa atitude. Os servidores da Web ocupados medem o número de conexões TCP por dia na casa dos milhões. Isso serve para um host individual e os sites ocupados executam múltiplos hosts, distribuindo a carga entre eles. (A Seção 14.2 do TCPv3 fala sobre uma maneira comum de distribuir essa carga utilizando o que é chamado de "rodízio de DNS".) As seções posteriores descreverão várias técnicas que evitam a bifurcação por cliente acarretada por um servidor concorrente, mas os servidores concorrentes ainda são comuns.

A Figura 30.4 mostra a função `main` de nosso servidor TCP concorrente.

server/serv01.c

```
 1 #include   "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    listenfd, connfd;
 6     pid_t  childpid;
 7     void   sig_chld(int), sig_int(int), web_child(int);
 8     socklen_t clilen, addrlen;
 9     struct sockaddr *cliaddr;

10     if (argc == 2)
11         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
12     else if (argc == 3)
```

Figura 30.4 Função `main` para servidor TCP concorrente (*continua*).

```
13          listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
14      else
15          err_quit("usage: serv01 [ <host> ] <port#>");
16      cliaddr = Malloc(addrlen);

17      Signal(SIGCHLD, sig_chld);
18      Signal(SIGINT, sig_int);

19      for ( ; ; ) {
20          clilen = addrlen;
21          if ( (connfd = accept(listenfd, cliaddr, &clilen)) < 0) {
22              if (errno == EINTR)
23                  continue;          /* de volta para for()*/
24              else
25                  err_sys("accept error");
26          }

27          if ( (childpid = Fork()) == 0) { /* processo-filho */
28              Close(listenfd);       /* fecha o soquete ouvinte */
29              web_child(connfd);     /* solicitação de processo */
30              exit(0);
31          }
32          Close(connfd);             /* pai fecha o soquete conectado */
33      }
34  }
```

server/serv01.c

Figura 30.4 Função `main` para servidor TCP concorrente (*continuação*).

Essa função é semelhante à Figura 5.12: ela chama `fork` para cada conexão de cliente e trata dos sinais `SIGCHLD` dos filhos que terminam. Entretanto, tornamos essa função independente de protocolo, chamando nossa função `tcp_listen`. Não mostramos o handler de sinal `sig_chld`: ele é o mesmo da Figura 5.11, com o comando `printf` removido.

Também capturamos o sinal `SIGINT`, gerado quando digitamos nossa chave de interrupção no terminal. Digitamos essa chave depois que o cliente termina, para imprimir o tempo de CPU exigido para o programa. A Figura 30.5 mostra o handler de sinal. Esse é um exemplo de handler de sinal que não retorna.

server/serv01.c

```
35  void
36  sig_int(int signo)
37  {
38      void    pr_cpu_time(void);

39      pr_cpu_time();
40      exit(0);
41  }
```

server/serv01.c

Figura 30.5 Handler de sinal para `SIGINT`.

A Figura 30.6 mostra a função `pr_cpu_time`, que é chamada pelo handler de sinal.

server/pr_cpu_time.c

```
 1  #include   "unp.h"
 2  #include   <sys/resource.h>

 3  #ifndef HAVE_GETRUSAGE_PROTO
 4  int     getrusage(int, struct rusage *);
 5  #endif
```

Figura 30.6 Função `pr_cpu_time`: imprime o tempo de CPU total (*continua*).

```
 6 void
 7 pr_cpu_time(void)
 8 {
 9     double user, sys;
10     struct rusage myusage, childusage;

11     if (getrusage(RUSAGE_SELF, &myusage) < 0)
12         err_sys("getrusage error");
13     if (getrusage(RUSAGE_CHILDREN, &childusage) < 0)
14         err_sys("getrusage error");

15     user = (double) myusage.ru_utime.tv_sec +
16         myusage.ru_utime.tv_usec / 1000000.0;
17     user += (double) childusage.ru_utime.tv_sec +
18         childusage.ru_utime.tv_usec / 1000000.0;
19     sys = (double) myusage.ru_stime.tv_sec +
20         myusage.ru_stime.tv_usec / 1000000.0;
21     sys += (double) childusage.ru_stime.tv_sec +
22         childusage.ru_stime.tv_usec / 1000000.0;

23     printf("\nuser time = %g, sys time = %g\n", user, sys);
24 }
```

server/pr_cpu_time.c

Figura 30.6 Função `pr_cpu_time`: imprime o tempo de CPU total (*continuação*).

A função `getrusage` é chamada duas vezes para retornar a utilização de recursos do processo chamador (`RUSAGE_SELF`) e de todos os filhos terminados desse processo (`RUSAGE_CHILDREN`). Os valores impressos são o tempo total do usuário (tempo de CPU gasto no processo de usuário) e o tempo total do sistema (tempo de CPU gasto dentro do kernel, executando em nome do processo chamador).

A função `main` da Figura 30.4 chama a função `web_child` para tratar de cada solicitação de cliente. A Figura 30.7 mostra essa função.

server/web_child.c

```
 1 #include   "unp.h"

 2 #define MAXN  16384     /* n° máx de bytes que o cliente pode solicitar */

 3 void
 4 web_child(int sockfd)
 5 {
 6     int     ntowrite;
 7     ssize_t nread;
 8     char    line[MAXLINE], result[MAXN];

 9     for ( ; ; ) {
10         if ( (nread = Readline(sockfd, line, MAXLINE)) == 0)
11             return;       /* conexão fechada pela outra extremidade */

12             /* a linha do cliente especifica o n° de bytes a gravar de volta */
13         ntowrite = atol(line);
14         if ((ntowrite <= 0) || (ntowrite > MAXN))
15             err_quit("client request for %d bytes", ntowrite);

16         Writen(sockfd, result, ntowrite);
17     }
18 }
```

server/web_child.c

Figura 30.7 Função `web_child` para tratar de cada solicitação do cliente.

Depois que o cliente estabelece a conexão com o servidor, ele escreve uma única linha especificando o número de bytes que este deve retornar. Isso é bastante semelhante ao HTTP: o

cliente envia uma pequena solicitação e o servidor responde com as informações desejadas (freqüentemente um arquivo HTML ou uma imagem GIF, por exemplo). No caso do HTTP, o servidor normalmente fecha a conexão após enviar de volta os dados solicitados, embora as versões mais recentes estejam utilizando *conexões persistentes*, mantendo a conexão TCP aberta para solicitações adicionais do cliente. Em nossa função `web_child`, o servidor permite solicitações adicionais do cliente, mas vimos na Figura 30.3 que nosso cliente envia somente uma solicitação por conexão e, então, fecha a conexão.

A linha 1 da Figura 30.1 mostra o resultado da medição de tempo para esse servidor concorrente. Quando comparamos com as linhas subseqüentes nessa figura, vemos que o servidor concorrente exige maior tempo de CPU, que é o que esperamos com um `fork` por cliente.

> Um projeto de servidor que não medimos neste capítulo é aquele ativado por `inetd`, que abordamos na Seção 13.5. Da perspectiva de controle de processos, um servidor ativado por `inetd` envolve um `fork` e um `exec`; portanto, o tempo de CPU será ainda maior do que os tempos mostrados na linha 1 da Figura 30.1.

30.6 Servidor TCP pré-bifurcado, sem bloqueio de `accept`

Nosso primeiro dos servidores TCP "aprimorados" utiliza uma técnica chamada *pré-bifurcação.* Em vez de gerar um `fork` por cliente, o servidor bifurca previamente certo número de filhos ao iniciar e, então, os filhos estão prontos para atender os clientes à medida que cada conexão de cliente chegar. A Figura 30.8 mostra um cenário onde o pai bifurcou previamente *N* filhos e dois clientes estão correntemente conectados.

A vantagem dessa técnica é que novos clientes podem ser tratados sem o custo de um `fork` feito pelo pai. A desvantagem é que o pai precisa adivinhar quantos filhos deve pré-bifurcar ao iniciar. Se, a qualquer momento, o número de clientes for igual ao de filhos, os clientes adicionais serão ignorados até que um filho esteja disponível. Mas lembre-se, da Seção 4.5, de que os clientes não são completamente ignorados. O kernel completará o handshake de três vias para todos os clientes adicionais, até o acúmulo de `listen` para esse soquete, e, então, passará as conexões completadas para o servidor, quando chamar `accept`. Mas a aplicação cliente pode notar uma degradação no tempo de resposta, pois, mesmo que sua conexão possa retornar imediatamente, sua primeira solicitação talvez não seja tratada pelo servidor por algum tempo.

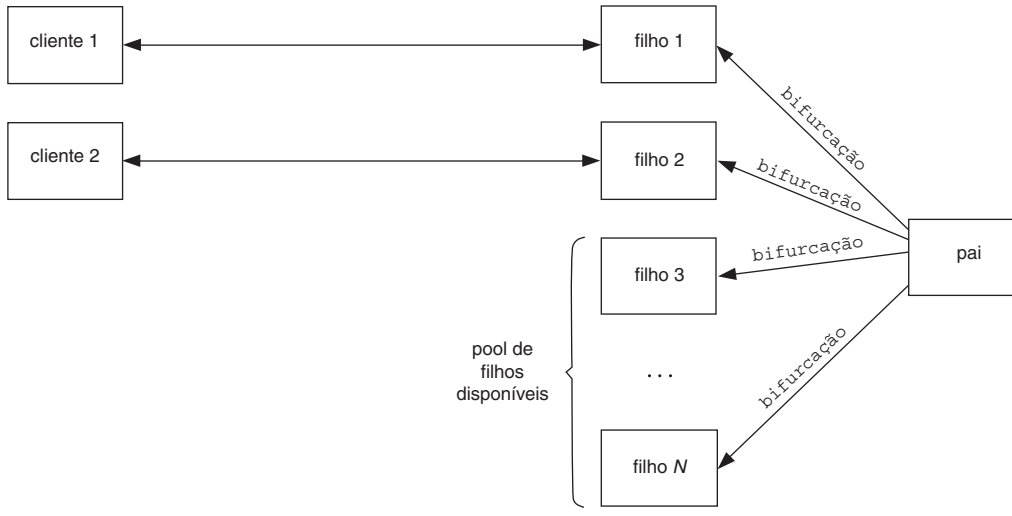


Figura 30.8 Bifurcação prévia de filhos pelo servidor.

Com alguma codificação extra, o servidor sempre pode tratar da carga de cliente. O que o pai deve fazer é monitorar continuamente o número de filhos disponíveis e, se esse valor cair abaixo de algum limite, ele deve bifurcar filhos adicionais. Além disso, se o número de filhos disponíveis exceder outro limite, o pai pode terminar alguns dos filhos em excesso, pois, conforme veremos mais adiante neste capítulo, ter filhos disponíveis demais também pode degradar o desempenho.

Mas, antes de nos preocuparmos com esses aprimoramentos, vamos examinar a estrutura básica desse tipo de servidor. A Figura 30.9 mostra a função `main` da primeira versão de nosso servidor pré-bifurcado.

11–18 Um argumento adicional de linha de comando é o número de filhos a pré-bifurcar. Um array é alocado para conter as PIDs dos filhos, as quais precisamos quando o programa termina, para permitir que a função `main` termine todos os filhos.

19–20 Cada filho é criado por `child_make`, que examinaremos na Figura 30.11.

Nosso handler de sinal para `SIGINT`, que mostramos na Figura 30.10, difere da Figura 30.5.

30–34 `getrusage` relata a utilização de recursos dos filhos *terminados*; portanto, devemos terminar todos os filhos antes de chamar `pr_cpu_time`. Fazemos isso enviando `SIGTERM` para cada filho e, então, esperamos (`wait`) por todos os filhos.

A Figura 30.11 mostra a função `child_make`, que é chamada por `main` para criar cada filho.

7–9 `fork` cria cada filho e somente o pai retorna. O filho chama a função `child_main`, que mostramos na Figura 30.12 e que é um loop infinito.

server/serv02.c

```
 1 #include   "unp.h"
 2 static int nchildren;
 3 static pid_t *pids;
 4 int
 5 main(int argc, char **argv)
 6 {
 7     int    listenfd, i;
 8     socklen_t addrlen;
 9     void   sig_int(int);
10     pid_t  child_make(int, int, int);
11     if (argc == 3)
12         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
13     else if (argc == 4)
14         listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
15     else
16         err_quit("usage: serv02 [ <host> ] <port#> <#children>");
17     nchildren = atoi(argv[argc - 1]);
18     pids = Calloc(nchildren, sizeof(pid_t));
19     for (i = 0; i < nchildren; i++)
20         pids[i] = child_make(i, listenfd, addrlen); /* o pai retorna */
21     Signal(SIGINT, sig_int);
22     for ( ; ; )
23         pause();                    /* tudo feito pelos filhos */
24 }
```

server/serv02.c

Figura 30.9 Função `main` do servidor pré-bifurcado.

server/serv02.c

```
25 void
26 sig_int(int signo)
27 {
28     int    i;
29     void   pr_cpu_time(void);
30         /* termina todos os filhos */
31     for (i = 0; i < nchildren; i++)
32         kill(pids[i], SIGTERM);
33     while (wait(NULL) > 0)       /* espera por todos os filhos */
34         ;
35     if (errno != ECHILD)
36         err_sys("wait error");
37     pr_cpu_time();
38     exit(0);
39 }
```

server/serv02.c

Figura 30.10 Handler de sinal para `SIGINT`.

server/child02.c

```
 1 #include   "unp.h"
 2 pid_t
 3 child_make(int i, int listenfd, int addrlen)
 4 {
 5     pid_t  pid;
 6     void   child_main(int, int, int);
 7     if ( (pid = Fork()) > 0)
 8         return (pid);             /* pai */
 9     child_main(i, listenfd, addrlen);    /* nunca retorna */
10 }
```

server/child02.c

Figura 30.11 Função `child_make`: cria cada filho.

server/child02.c

```
11 void
12 child_main(int i, int listenfd, int addrlen)
13 {
14     int    connfd;
15     void   web_child(int);
16     socklen_t clilen;
17     struct sockaddr *cliaddr;
18     cliaddr = Malloc(addrlen);
19     printf("child %ld starting\n", (long) getpid());
20     for ( ; ; ) {
21         clilen = addrlen;
22         connfd = Accept(listenfd, cliaddr, &clilen);
23         web_child(connfd);        /* processa a solicitação */
24         Close(connfd);
25     }
26 }
```

server/child02.c

Figura 30.12 Função `child_main`: loop infinito executado por cada filho.

20–25 Cada filho chama `accept` e, quando `accept` retorna, a função `web_child` (Figura 30.7) trata da solicitação do cliente. O filho continua nesse loop até que seja terminado pelo pai.

Implementação 4.4BSD

Se você nunca viu esse tipo de organização (vários processos chamando `accept` no mesmo descritor ouvinte), provavelmente está se perguntando como isso pode funcionar. Vale uma breve divagação sobre como isso é implementado nos kernels derivados do Berkeley (por exemplo, conforme apresentado no TCPv2).

O pai cria o soquete ouvinte antes de gerar quaisquer filhos e, se você se lembra, todos os descritores são duplicados em cada filho, sempre que `fork` é chamado. A Figura 30.13 mostra a organização das estruturas `proc` (uma por processo), a estrutura `file` para o descritor ouvinte e a estrutura `socket`.

Os descritores são apenas um índice em um array na estrutura `proc` que referenciam uma estrutura `file`. Uma das propriedades da duplicação de descritores no filho, que ocorre com `fork`, é que determinado descritor no filho referencia a mesma estrutura `file` no pai. Cada estrutura `file` tem uma contagem de referência que inicia em 1, quando o arquivo ou soquete é aberto, e é incrementada por 1 sempre que `fork` é chamado ou sempre que o descritor é duplicado (`dup`). Em nosso exemplo com *N* filhos, a contagem de referência na estrutura `file` seria $N + 1$ (não esqueça o pai, que ainda tem o descritor ouvinte aberto, mesmo que nunca chame `accept`).

Quando o programa inicia, *N* filhos são criados e todos chamam `accept` e são colocados para dormir pelo kernel (linha 140, página 458 do TCPv2). Quando a primeira conexão de cliente chega, todos os *N* filhos são acordados. Isso porque todos os *N* foram dormir no mesmo "canal de espera", o membro `so_timeo` da estrutura `socket`, pois compartilham o mesmo descritor ouvinte, que aponta para a mesma estrutura `socket`. Mesmo que todos os *N* sejam acordados, o primeiro deles a executar obterá a conexão e os $N - 1$ restantes voltarão a dormir, porque, quando cada um dos $N - 1$ restantes executar a instrução da linha 135 da página 458 do TCPv2, o comprimento da fila será 0, pois o primeiro filho a executar já pegou a conexão.

Às vezes, isso é chamado de problema de *estouro da manada*, pois todos os *N* são acordados, mesmo que somente um obtenha a conexão. Contudo, o código funciona, com o efeito colateral no desempenho de acordar processos demais sempre que uma conexão está pronta para ser aceita. Agora, mediremos esse efeito no desempenho.

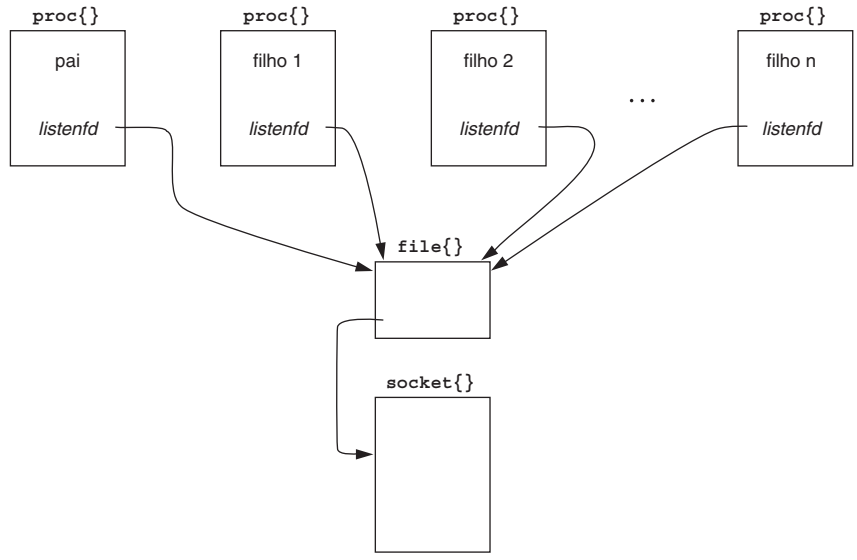


Figura 30.13 Organização das estruturas `proc`, `file` e `socket`.

Efeito causado pela existência de filhos demais

O tempo de CPU de 1,8 para o servidor na linha 2 da Figura 30.1 refere-se aos 15 filhos e a um máximo de 10 clientes simultâneos. Podemos medir o efeito do problema do estouro da manada apenas aumentando o número de filhos para o mesmo número máximo de clientes (10). Não mostramos os resultados do aumento do número de filhos, porque os resultados individuais do teste não são de interesse. Como qualquer número maior que 10 introduz filhos supérfluos, o problema do estouro da manada piora e os resultados da medição de tempo aumentam.

> Alguns kernels Unix têm uma função, freqüentemente chamada de `wakeup_one`, que acorda somente o processo que esteja esperando por algum evento, em vez de acordar todos os processos que estejam esperando pelo evento (Schimmel, 1994).

Distribuição das conexões para os filhos

O próximo item a examinar é a distribuição das conexões de cliente para o pool de filhos disponíveis que são bloqueados na chamada a `accept`. Para coletar essas informações, modificamos a função `main` para alocar um array de contadores de inteiro longo na memória compartilhada, um contador por filho. Isso é feito com o seguinte:

```
long *cptr, *meter(int);    /* para contar o número de clientes/filho */
cptr = meter(nchildren);    /* antes de gerar filhos */
```

A Figura 30.14 mostra a função `meter`.

Utilizamos o mapeamento de memória anônimo, se for suportado (por exemplo, 4.4BSD), ou o mapeamento `/dev/zero` (por exemplo, SVR4). Como o array é criado por `mmap` antes que os filhos sejam gerados, ele é então compartilhado entre esse processo (o pai) e todos os seus filhos criados posteriormente por `fork`.

Então, modificamos nossa função `child_main` (Figura 30.12) de modo que cada filho incremente seu contador quando `accept` retornar, e nosso handler `SIGINT` imprime esse array depois que todos os filhos tiverem terminado.

server/meter.c

```
 1 #include   "unp.h"
 2 #include   <sys/mman.h>

 3 /*
 4  * Aloca um array de "nchildren" números longos na memória compartilhada que pode
 5  * ser utilizado por cada filho como um contador de quantos clientes ele atende.
 6  * Veja as páginas 467 a 470 de "Advanced Programming in the Unix Environment".
 7  */

 8 long *
 9 meter(int nchildren)
10 {
11     int    fd;
12     long  *ptr;

13 #ifdef MAP_ANON
14     ptr = Mmap(0, nchildren * sizeof(long), PROT_READ | PROT_WRITE,
15                MAP_ANON | MAP_SHARED, -1, 0);
16 #else
17     fd = Open("/dev/zero", O_RDWR, 0);

18     ptr = Mmap(0, nchildren * sizeof(long), PROT_READ | PROT_WRITE,
19                MAP_SHARED, fd, 0);
20     Close(fd);
21 #endif

22     return (ptr);
23 }
```

server/meter.c

Figura 30.14 A função `meter` para alocar um array na memória compartilhada.

A Figura 30.2 mostra a distribuição. Quando os filhos disponíveis são bloqueados na chamada a `accept`, o algoritmo de escalonamento do kernel distribui as conexões uniformemente para todos os filhos.

Colisões de `select`

Ao examinarmos esse exemplo sob o 4.4BSD, também podemos observar outro fenômeno mal entendido, mas raro. A seção 16.13 do TCPv2 fala sobre *colisões* com a função `select` e como o kernel trata dessa possibilidade. Uma colisão ocorre quando vários processos chamam `select` no mesmo descritor, porque espaço é alocado na estrutura `socket` para que apenas um ID de processo seja acordado quando o descritor estiver pronto. Se vários processos estão esperando pelo mesmo descritor, o kernel deve acordar *todos* os processos que estiverem bloqueados em uma chamada a `select`, pois ele não sabe quais processos são afetados pelo descritor que acaba de se tornar pronto.

Podemos forçar colisões de `select` com nosso exemplo, precedendo a chamada a `accept`, na Figura 30.12, com uma chamada a `select`, esperando por legibilidade no soquete ouvinte. Os filhos gastarão seu tempo bloqueados nessa chamada a `select`, em vez de na chamada a `accept`. A Figura 30.15 mostra a parte da função `child_main` que muda, usando sinais de adição para denotar as linhas que mudaram em relação à Figura 30.12.

```
      printf("child %ld starting\n", (long) getpid());
+     FD_ZERO(&rset);
      for ( ; ; ) {
+        FD_SET(listenfd, &rset);
+        Select(listenfd+1, &rset, NULL, NULL, NULL);
+        if(FD_ISSET(listenfd, &rset) == 0)
+           err_quit("listenfd readable");
+
         clilen = addrlen;
         connfd = Accept(listenfd, cliaddr, &clilen);

         web_child(connfd);        /* solicitação de processo */
         Close(connfd);
      }
```

Figura 30.15 Modificação da Figura 30.12 para bloquear em `select`, em vez de `accept`.

Se fizermos essa alteração e depois examinarmos o contador de colisão de `select` do kernel, antes e depois, veremos 1.814 colisões quando executarmos o servidor e 2.045 na próxima vez. Como os dois clientes criam um total de 5.000 conexões para cada execução do servidor, isso corresponde a cerca de 35 a 40% das chamadas a `select` ativando uma colisão.

Se compararmos o tempo de CPU do servidor para esse exemplo, o valor 1,8 na Figura 30.1 aumenta para 2,9 quando adicionamos a chamada a `select`. Parte desse aumento provavelmente se dá por causa da chamada de sistema adicional (pois estamos chamando `select` e `accept`, em vez de somente `accept`) e outra parte provavelmente por causa do overhead do kernel no tratamento das colisões.

A lição a ser aprendida com essa discussão é: quando vários processos estão sendo bloqueados no mesmo descritor, é melhor bloquear em uma função como `accept`, em vez de bloquear em `select`.

30.7 Servidor TCP pré-bifurcado, bloqueio de arquivo em torno de `accept`

A implementação que acabamos de descrever para 4.4BSD, que permite que vários processos chamem `accept` no mesmo descritor ouvinte, funciona somente com kernels derivados do Berkeley que implementam `accept` dentro do kernel. Os kernels do System V, que imple-

mentam `accept` como uma função de biblioteca, talvez não permitam isso. De fato, se executarmos o servidor da seção anterior em tal sistema, logo depois que os clientes iniciarem a conexão com o servidor, uma chamada a `accept` em um dos filhos retorna `EPROTO`, o que significa um erro de protocolo.

> As razões para esse problema com a versão de biblioteca SVR4 de `accept` provêm da implementação de STREAMS (Capítulo 31) e do fato de a função `accept` de biblioteca não ser uma operação atômica. O Solaris corrige isso, mas o problema ainda existe na maioria das outras implementações de SVR4.

A solução é a aplicação colocar um *bloqueio* de algum tipo em torno da chamada a `accept`, de modo que somente um processo por vez seja bloqueado nessa chamada. Os filhos restantes serão bloqueados ao tentar obter o acesso ao recurso bloqueado.

Há várias maneiras de fornecer esse bloqueio em torno da chamada a `accept`, conforme descreveremos no segundo volume desta série. Nesta seção, utilizaremos o bloqueio de arquivo POSIX com a função `fcntl`.

A única alteração na função `main` (Figura 30.9) é adicionar uma chamada a nossa função `my_lock_init`, antes do loop que cria os filhos.

```
+   my_lock_init("/tmp/lock.XXXXXX"); /* um arquivo de bloqueio para todos
                                         os filhos */
    for (i = 0; i < nchildren; i++)
       pids[i] = child_make(i, listenfd, addrlen); /* o pai retorna */
```

A função `child_make` permanece a mesma, como na Figura 30.11. A única alteração em nossa função `child_main` (Figura 30.12) é obter um bloqueio antes de chamar `accept` e liberá-lo depois que `accept` retornar.

```
    for ( ; ; ) {
       clilen = addrlen;
+      my_lock_wait();
       connfd = Accept(listenfd, cliaddr, &clilen);
+      my_lock_release();
       web_child(connfd);        /* solicitação de processo */
       Close(connfd);
    }
```

A Figura 30.16 mostra nossa função `my_lock_init`, que utiliza bloqueio de arquivo POSIX.

server/lock_fcntl.c

```
 1 #include "unp.h"
 2 static struct flock lock_it, unlock_it;
 3 static int lock_fd = -1;
 4                     /*fcntl() falhará se my_lock_init() não for chamada */
 5 void
 6 my_lock_init(char *pathname)
 7 {
 8  char        lock_file[1024];
 9       /*deve copiar a string do chamador, no caso de ser uma constante */
10 strncpy(lock_file, pathname, sizeof(lock_file));
11 lock_fd = Mkstemp(lock_file);
12 Unlink(lock_file);             /* mas lock_fd permanece aberta */
13 lock_it.l_type = F_WRLCK;
14 lock_it.l_whence = SEEK_SET;
15 lock_it.l_start = 0;
```

Figura 30.16 Função `my_lock_init` utilizando bloqueio de arquivo POSIX (*continua*).

```
16 lock_it.l_len = 0;

17 unlock_it.l_type = F_UNLCK;
18 unlock_it.l_whence = SEEK_SET;
19 unlock_it.l_start = 0;
20 unlock_it.l_len = 0;
21 }
```

server/lock_fcntl.c

Figura 30.16 Função `my_lock_init` utilizando bloqueio de arquivo POSIX (*continuação*).

9–12 O chamador especifica um modelo de nome de caminho como o argumento para `my_lock_init` e a função `mktemp` cria um nome de caminho único baseado nesse modelo. Então, é criado um arquivo com esse nome de caminho e imediatamente desvinculado (`unlink`). Com a remoção do nome de caminho do diretório, se o programa falhar, o arquivo desaparece completamente. Mas, contanto que um ou mais processos tenham o arquivo aberto (isto é, a contagem de referência do arquivo seja maior que 0), o arquivo em si não é removido. (Essa é a diferença fundamental entre remover um nome de caminho de um diretório e fechar um arquivo aberto.)

13–20 Duas estruturas `flock` são inicializadas: uma para bloquear o arquivo e outra para desbloqueá-lo. O intervalo do arquivo que é bloqueado inicia no deslocamento de byte 0 (um `l_whence` de `SEEK_SET` com `l_start` configurado como 0). Como `l_len` é configurado como 0, isso especifica que o arquivo inteiro é bloqueado. Nunca gravamos nada no arquivo (seu comprimento é sempre 0), mas isso está certo. O bloqueio consultivo ainda é tratado corretamente pelo kernel.

> Pode ser tentador inicializar essas estruturas utilizando
>
> ```
> static struct flock lock_it = { F_WRLCK, 0, 0, 0, 0 };
> static struct flock unlock_it = { F_UNLCK, 0, 0, 0, 0 };
> ```
>
> mas há dois problemas. Primeiro, não há nenhuma garantia de que a constante `SEEK_SET` seja 0. Mas o mais importante é que não há nenhuma garantia do POSIX quanto à ordem dos membros na estrutura. O membro `l_type` pode ser o primeiro na estrutura, mas não em todos os sistemas. Tudo que o POSIX garante é que os membros exigidos estão presentes na estrutura. O POSIX não garante a ordem dos membros e também permite que membros adicionais, não-POSIX, estejam na estrutura. Portanto, inicializar uma estrutura com algo que não seja zero sempre deve ser feito pelo código C real e não por um inicializador, quando a estrutura é alocada.
>
> Uma exceção a essa regra se dá quando o inicializador de estrutura é fornecido pela implementação. Por exemplo, ao inicializar um bloqueio mutex de Pthread, no Capítulo 26, escrevemos
>
> ```
> pthread_mutex_t mlock = PTHREAD_MUTEX_INITIALIZER;
> ```
>
> O tipo de dados `pthread_mutex_t` é freqüentemente uma estrutura, mas o inicializador é fornecido pela implementação e pode diferir de uma implementação para outra.

A Figura 30.17 mostra as duas funções que bloqueiam e desbloqueiam o arquivo. Elas são apenas chamadas para `fcntl`, utilizando as estruturas que foram inicializadas na Figura 30.16. Essa nova versão de nosso servidor pré-bifurcado funciona agora em sistemas SVR4, garantindo que somente um processo-filho por vez seja bloqueado na chamada a `accept`. Uma comparação das linhas 2 e 3, na Figura 30.1, mostra que esse tipo de bloqueio aumenta o tempo de CPU do controle de processos do servidor.

> O servidor da Web Apache, `http://www.apache.org`, pré-bifurca seus filhos e, então, utiliza uma das técnicas da seção anterior (todos os filhos bloqueados na chamada a `accept`), caso a implementação permita isso, ou o bloqueio de arquivo em torno de `accept`.

server/lock_fcntl.c

```
22 void
23 my_lock_wait()
24 {
25     int    rc;

26     while ( (rc = fcntl(lock_fd, F_SETLKW, &lock_it)) < 0) {
27         if (errno == EINTR)
28             continue;
29         else
30             err_sys("fcntl error for my_lock_wait");
31     }
32 }

33 void
34 my_lock_release()
35 {
36     if (fcntl(lock_fd, F_SETLKW, &unlock_it) < 0)
37         err_sys("fcntl error for my_lock_release");
38 }
```

server/lock_fcntl.c

Figura 30.17 Funções `my_lock_wait` e `my_lock_release` utilizando `fcntl`.

Efeito causado pela existência de filhos demais

Podemos verificar essa versão para ver se existe o mesmo problema de estouro da manada, o qual descrevemos na seção anterior. Verificamos isso aumentando o número de filhos (desnecessários) e observando que os resultados da medição de tempo pioram proporcionalmente.

Distribuição das conexões para os filhos

Podemos examinar a distribuição dos clientes no pool de filhos disponíveis, utilizando a função que descrevemos com a Figura 30.14. A Figura 30.2 mostra o resultado. O SO distribui os bloqueios de arquivo uniformemente para os processos que estão esperando (e esse comportamento foi uniforme nos vários sistemas operacionais que testamos).

30.8 Servidor TCP pré-bifurcado, bloqueio de thread em torno de `accept`

Conforme mencionamos, há várias maneiras de implementar bloqueio entre processos. O bloqueio de arquivo POSIX da seção anterior é portável para todos os sistemas compatíveis com POSIX, mas envolve operações de sistema de arquivos, que podem levar tempo. Nesta seção, utilizaremos o bloqueio de thread, tirando proveito do fato de que isso pode ser utilizado não apenas para bloqueio entre os threads dentro de determinado processo, mas também para bloqueio entre diferentes processos.

Nossa função `main` permanece a mesma da seção anterior, assim como nossas funções `child_make` e `child_main`. O que muda são apenas nossas três funções de bloqueio. Utilizar o bloqueio de thread entre diferentes processos exige que: (i) a variável de mutex deve ser armazenada na memória que é compartilhada entre todos os processos; e (ii) a biblioteca de thread deve ser informada de que o mutex é compartilhado entre diferentes processos.

Além disso, a biblioteca de thread deve suportar o atributo `PTHREAD_PROCESS_SHARED`.

Há várias maneiras de compartilhar memória entre processos diferentes, conforme descreveremos no segundo volume desta série. Em nosso exemplo, utilizaremos a função `mmap` com o dispositivo `/dev/zero`, que funciona sob Solaris e outros kernels SVR4. A Figura 30.18 mostra nossa função `my_lock_init`.

server/lock_pthread.c

```
 1 #include   "unpthread.h"
 2 #include   <sys/mman.h>
 3 static pthread_mutex_t *mptr;           /* o mutex real estará na memória
                                             compartilhada */
 4 void
 5 my_lock_init(char *pathname)
 6 {
 7     int    fd;
 8     pthread_mutexattr_t mattr;
 9     fd = Open("/dev/zero", O_RDWR, 0);
10     mptr = Mmap(0, sizeof(pthread_mutex_t), PROT_READ | PROT_WRITE,
11                 MAP_SHARED,fd, 0);
12     Close(fd);
13     Pthread_mutexattr_init(&mattr);
14     Pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
15     Pthread_mutex_init(mptr, &mattr);
16 }
```

server/lock_pthread.c

Figura 30.18 Função `my_lock_init` utilizando bloqueio de Pthread entre processos.

9–12 Abrimos `/dev/zero` e então chamamos `mmap`. O número de bytes que são mapeados é igual ao tamanho de uma variável `pthread_mutex_t`. O descritor é então fechado, o que está certo, pois a memória mapeada em `/dev/zero` permanecerá mapeada.

13–15 Em nossos exemplos de mutex Pthread anteriores, inicializamos a variável global ou estática mutex utilizando a constante `PTHREAD_MUTEX_INITIALIZER` (por exemplo, Figura 26.18). Mas, com um mutex na memória compartilhada, devemos chamar algumas funções de biblioteca Pthread para dizer à biblioteca que o mutex está na memória compartilhada e que será utilizado para bloqueio entre diferentes processos. Primeiro, inicializamos uma estrutura `pthread_mutexattr_t` com os atributos default para um mutex e, então, configuramos o atributo `PTHREAD_PROCESS_SHARED`. (O default para esse atributo é `PTHREAD_PROCESS_PRIVATE`, permitindo a utilização somente dentro de um único processo.) `pthread_mutex_init` inicializa então o mutex com esses atributos.

A Figura 30.19 mostra nossas funções `my_lock_wait` e `my_lock_release`. Cada uma é agora somente uma chamada a uma função Pthread para bloquear ou desbloquear o mutex.

Uma comparação das linhas 3 e 4, na Figura 30.1, para o servidor Solaris, mostra que o bloqueio de mutex de thread é mais rápido que o bloqueio de arquivo.

server/lock_pthread.c

```
17 void
18 my_lock_wait()
19 {
20     Pthread_mutex_lock(mptr);
21 }
22 void
23 my_lock_release()
24 {
25     Pthread_mutex_unlock(mptr);
26 }
```

server/lock_pthread.c

Figura 30.19 Funções `my_lock_wait` e `my_lock_release` utilizando bloqueio de Pthread.

30.9 Servidor TCP pré-bifurcado, passagem de descritor

A última modificação em nosso servidor pré-bifurcado é fazer com que somente o pai chame `accept` e, então, "passar" o soquete conectado para um filho. Isso evita a possível necessidade de bloqueio em torno da chamada a `accept` em todos os filhos, mas exige alguma forma de passagem de descritor do pai para os filhos. Essa técnica também complica bastante o código, pois o pai precisa monitorar quais filhos estão ocupados e quais estão livres, para passar um novo soquete para um filho livre.

Nos exemplos de bifurcação prévia anteriores, o processo nunca se preocupava com qual filho recebia uma conexão de cliente. O SO tratava desse detalhe, fornecendo a um dos filhos a primeira chamada a `accept` ou fornecendo a ele o bloqueio de arquivo ou de mutex. As duas primeiras colunas da Figura 30.2 também mostram que o SO que estamos medindo faz isso com um rodízio imparcial.

Nesse exemplo, precisamos manter uma estrutura de informações sobre cada filho. Mostramos nosso cabeçalho `child.h`, que define nossa estrutura `Child`, na Figura 30.20.

server/child.h

```
1 typedef struct {
2    pid_t  child_pid;     /* ID de processo */
3    int    child_pipefd;  /* pipe de fluxo do pai para/do filho */
4    int    child_status;  /* 0 = pronto */
5    long   child_count;   /* número de conexões tratadas */
6 } Child;
7 Child *cptr;             /* array de estruturas Child; com uso de calloc */
```

server/child.h

Figura 30.20 Estrutura `Child`.

Armazenamos a PID do filho, o descritor de pipe de fluxo do pai que está conectado ao filho, o *status* do filho e uma contagem do número de clientes que o filho tratou. Imprimiremos esse contador em nosso handler `SIGINT` para ver a distribuição das solicitações de cliente entre os filhos.

Vamos ver primeiro a função `child_make`, que mostramos na Figura 30.21. Criamos um pipe de fluxo, um soquete de fluxo de domínio Unix (Capítulo 15), antes de chamar `fork`. Depois que o filho é criado, o pai fecha um descritor (`sockfd[1]`) e o filho fecha o outro descritor (`sockfd[0]`). Além disso, o filho duplica sua extremidade do pipe de fluxo (`sockfd[1]`) no erro-padrão, de modo que cada filho lê e grava apenas no erro-padrão, para se comunicar com o pai. Isso nos dá a organização mostrada na Figura 30.22.

server/child05.c

```
1 #include   "unp.h"
2 #include   "child.h"
3 pid_t
4 child_make(int i, int listenfd, int addrlen)
5 {
6    int    sockfd[2];
7    pid_t  pid;
8    void   child_main(int, int, int);
9    Socketpair(AF_LOCAL, SOCK_STREAM, 0, sockfd);
10   if ( (pid = Fork()) > 0) {
```

Figura 30.21 Servidor pré-bifurcado com passagem de descritor da função `child_make` (*continua*).

```
11          Close(sockfd[1]);
12          cptr[i].child_pid = pid;
13          cptr[i].child_pipefd = sockfd[0];
14          cptr[i].child_status = 0;
15          return (pid);              /* pai */
16      }
17      Dup2(sockfd[1], STDERR_FILENO);/* pipe de fluxo do filho para o pai */
18      Close(sockfd[0]);
19      Close(sockfd[1]);
20      Close(listenfd);               /* o filho não precisa disso aberto */
21      child_main(i, listenfd, addrlen);    /* nunca retorna */
22  }
```

server/child05.c

Figura 30.21 Servidor pré-bifurcado com passagem de descritor da função `child_make` (*continuação*).

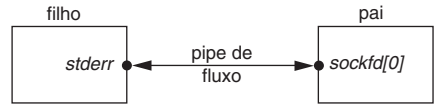


Figura 30.22 Pipe de fluxo após o pai e o filho fecharem uma extremidade.

Depois que todos os filhos são criados, temos a organização mostrada na Figura 30.23. Fechamos o soquete ouvinte em cada filho, pois somente o pai chama `accept`. Mostramos que o pai deve tratar do soquete ouvinte, junto com todos os soquetes de fluxo. Conforme você poderia supor, o pai utiliza `select` para multiplexar todos esses descritores.

A Figura 30.24 mostra a função `main`. As alterações em relação às versões anteriores dessa função são que as configurações do descritor são alocadas e que os bits correspondentes ao soquete ouvinte, junto com o pipe de fluxo para cada filho, são ativados na configuração. O valor máximo do descritor também é calculado. Alocamos memória para o array de estruturas `Child`. O loop principal é baseado em uma chamada a `select`.

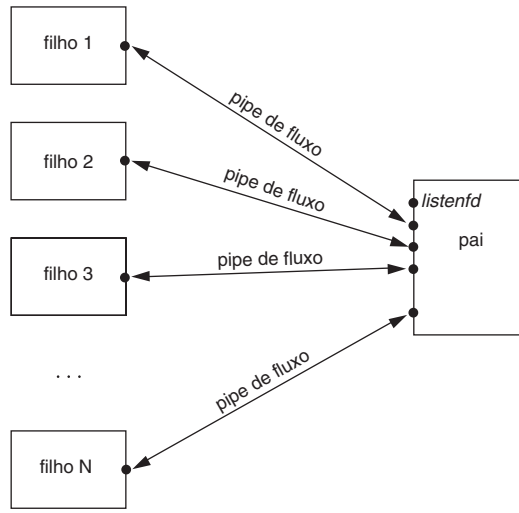


Figura 30.23 Pipes de fluxo após todos os filhos terem sido criados.

server/serv05.c

```
 1 #include   "unp.h"
 2 #include   "child.h"
 3 static int nchildren;
 4 int
 5 main(int argc, char **argv)
 6 {
 7     int    listenfd, i, navail, maxfd, nsel, connfd, rc;
 8     void   sig_int(int);
 9     pid_t  child_make(int, int, int);
10     ssize_ t n;
11     fd_set rset, masterset;
12     socklen_t addrlen, clilen;
13     struct sockaddr *cliaddr;
14     if (argc == 3)
15         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
16     else if (argc == 4)
17         listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
18     else
19         err_quit("usage: serv05 [ <host> ] <port#> <#children>");
20     FD_ZERO(&masterset);
21     FD_SET(listenfd, &masterset);
22     maxfd = listenfd;
23     cliaddr = Malloc(addrlen);
24     nchildren = atoi(argv[argc - 1]);
25     navail = nchildren;
26     cptr = Calloc(nchildren, sizeof(Child));
27         /* pré-bifurca todos os filhos */
28     for (i = 0; i < nchildren; i++) {
29         child_make(i, listenfd, addrlen); /* o pai retorna */
30         FD_SET(cptr[i].child_pipefd, &masterset);
31         maxfd = max(maxfd, cptr[i].child_pipefd);
32     }
33     Signal(SIGINT, sig_int);
34     for ( ; ; ) {
35         rset = masterset;
36         if (navail <= 0)
37             FD_CLR(listenfd, &rset);         /* desativa, caso nenhum filho
                                                    esteja disponível */
38         nsel = Select(maxfd + 1, &rset, NULL, NULL, NULL);
39             /* verifica novas conexões */
40         if (FD_ISSET(listenfd, &rset)) {
41             clilen = addrlen;
42             connfd = Accept(listenfd, cliaddr, &clilen);
43             for (i = 0; i < nchildren; i++)
44                 if (cptr[i].child_status == 0)
45                     break;  /* disponível */
46             if (i == nchildren)
47                 err_quit("no available children");
48             cptr[i].child_status = 1;     /* marca o filho como ocupado */
49             cptr[i].child_count++;
50             navail--;
51             n = Write_fd(cptr[i].child_pipefd, "", 1, connfd);
```

Figura 30.24 Função `main` que utiliza passagem de descritor (*continua*).

```
52              Close(connfd);
53              if (--nsel == 0)
54                  continue;       /* tudo feito com resultados de select() */
55          }
56              /* localiza todos os filhos recentemente disponíveis */
57          for (i = 0; i < nchildren; i++) {
58              if (FD_ISSET(cptr[i].child_pipefd, &rset)) {
59                  if ( (n = Read(cptr[i].child_pipefd, &rc, 1)) == 0)
60                      err_quit("child %d terminated unexpectedly", i);
61                  cptr[i].child_status = 0;
62                  navail++;
63                  if (--nsel == 0)
64                      break;      /* tudo feito com resultados de select() */
65              }
66          }
67      }
68  }
```

server/serv05.c

Figura 30.24 Função `main` que utiliza passagem de descritor (*continuação*).

Desativação do soquete ouvinte, caso nenhum filho esteja disponível

36–37 O contador `navail` monitora o número de filhos disponíveis. Se esse contador é 0, o soquete ouvinte é desativado no conjunto de descritores para `select`. Isso nos impede de aceitar uma nova conexão para a qual não haja filho disponível. O kernel ainda enfileira essas conexões entrantes, até o número máximo de acúmulo para `listen`, mas não queremos aceitá-las até que tenhamos um filho pronto para processar o cliente.

Aceitando `(accept)` novas conexões

39–55 Se o soquete ouvinte for legível, uma nova conexão está pronta para `accept`. Encontramos o primeiro filho disponível e passamos o soquete conectado a ele, utilizando nossa função `write_fd` da Figura 15.13. Gravamos um byte, junto com o descritor, mas o destinatário não vê o conteúdo desse byte. O pai fecha o soquete conectado.

Sempre começamos a procurar um filho disponível com a primeira entrada no array de estruturas `Child`. Isso significa que os primeiros filhos no array sempre recebem novas conexões para processar, antes dos elementos posteriores no array. Verificaremos isso quando discutirmos a Figura 30.2, e veremos os contadores `child_count` depois que o servidor terminar. Se não quiséssemos essa preferência em relação aos filhos anteriores, poderíamos lembrar qual filho recebeu a conexão mais recente e iniciar nossa pesquisa um elemento depois dele a cada vez, voltando ao primeiro elemento ao chegarmos no fim. Não há nenhuma vantagem em fazer isso (realmente não importa qual filho trata de uma solicitação de cliente, se vários filhos estão disponíveis), a menos que o algoritmo de escalonamento do SO penalize os processos com tempos de CPU totais mais longos. Dispersar a carga mais equilibradamente entre todos os filhos tenderia a equalizar seus tempos de CPU totais.

Tratando todos os filhos recentemente disponíveis

56–66 Veremos que nossa função `child_main` grava um único byte no pai, por meio do pipe de fluxo, quando o filho tiver terminado com um cliente. Isso torna a extremidade do pipe de fluxo do pai legível. Chamamos `read` para ler o único byte (ignorando seu valor) e então marcamos o filho como disponível. Se o filho terminar inesperadamente, sua extremidade do pipe de fluxo será fechada e `read` retornará 0. Capturamos isso e terminamos, mas uma estratégia melhor é registrar o erro e gerar um novo filho para substituir o que terminou.

Nossa função `child_main` é mostrada na Figura 30.25.

server/child05.c

```
23 void
24 child_main(int i, int listenfd, int addrlen)
25 {
26     char   c;
27     int    connfd;
28     ssize_t n;
29     void   web_child(int);

30     printf("child %ld starting\n", (long) getpid());
31     for ( ; ; ) {
32         if ( (n = Read_fd(STDERR_FILENO, &c, 1, &connfd)) == 0)
33             err_quit("read_fd returned 0");
34         if (connfd < 0)
35             err_quit("no descriptor from read_fd");

36         web_child(connfd);        /* solicitação de processo */
37         Close(connfd);

38         Write(STDERR_FILENO, "", 1);     /* diz ao pai que estamos prontos
                                               novamente */
39     }
40 }
```

server/child05.c

Figura 30.25 Função `child_main`: passagem de descritor, servidor pré-bifurcado.

Esperando o descritor do pai

32–33 Essa função difere daquelas das duas seções anteriores, porque nosso filho não chama mais `accept`. Em vez disso, ele bloqueia em uma chamada a `read_fd`, esperando que o pai passe um descritor de soquete conectado para ele processar.

Dizendo ao pai que estamos prontos

38 Quando terminamos com o cliente, gravamos (`write`) um byte por meio do pipe de fluxo, para dizer ao pai que estamos disponíveis.

Comparando as linhas 4 e 5, na Figura 30.1, vemos que esse servidor é mais lento que a versão da seção anterior, que utilizava bloqueio de thread entre os filhos. Passar um descritor por meio do pipe de fluxo para cada filho e gravar um byte de volta por meio do pipe de fluxo do filho leva mais tempo do que bloquear e desbloquear um mutex na memória compartilhada ou um bloqueio de arquivo.

A Figura 30.2 mostra a distribuição dos contadores `child_count` na estrutura `Child`, que imprimimos no handler `SIGINT` quando o servidor está terminado. Os filhos anteriores tratam de mais clientes, conforme discutimos na Figura 30.24.

30.10 Servidor TCP concorrente, um thread por cliente

As últimas cinco seções focalizaram um processo por cliente, um `fork` por cliente e bifurcaram previamente certo número de filhos. Se o servidor suporta threads, podemos utilizá-los, em vez dos processos-filhos.

Nossa primeira versão com threads aparece na Figura 30.26. Trata-se de uma modificação da Figura 30.4 que cria um thread por cliente, em vez de um processo por cliente. Essa versão é muito semelhante à Figura 26.3.

Loop de thread principal

19–23 O thread principal bloqueia em uma chamada a `accept` e, toda vez que uma conexão de cliente é retornada, um novo thread é criado por `pthread_create`. A função executada pelo novo thread é `doit` e seu argumento é o soquete conectado.

server/serv06.c

```
 1 #include    "unpthread.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     listenfd, connfd;
 6     void    sig_int(int);
 7     void    *doit(void *);
 8     pthread_t tid;
 9     socklen_t clilen, addrlen;
10     struct sockaddr *cliaddr;

11     if (argc == 2)
12         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
13     else if (argc == 3)
14         listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
15     else
16         err_quit("usage: serv06 [ <host> ] <port#>");
17     cliaddr = Malloc(addrlen);

18     Signal(SIGINT, sig_int);

19     for ( ; ; ) {
20         clilen = addrlen;
21         connfd = Accept(listenfd, cliaddr, &clilen);

22         Pthread_create(&tid, NULL, &doit, (void *) connfd);
23     }
24 }

25 void *
26 doit(void *arg)
27 {
28     void   web_child(int);

29     Pthread_detach(pthread_self());
30     web_child((int) arg);
31     Close((int) arg);
32     return (NULL);
33 }
```

server/serv06.c

Figura 30.26 Função `main` para servidor TCP com threads.

Função por thread

25–33 A função `doit` desprende-se para que o thread principal não tenha que esperar por ela e chama nossa função `web_client` (Figura 30.3). Quando essa função retorna, o soquete conectado é fechado.

Notamos, na Figura 30.1, que essa versão com threads simples é mais rápida até que a mais rápida das versões previamente bifurcadas. Essa versão de um thread por cliente é muitas vezes mais rápida que a versão de um filho por cliente (linha 1).

> Na Seção 26.5, observamos três alternativas para converter uma função que não é segura para thread em uma que é segura. Nossa função `web_child` chama nossa função `readline` e a versão mostrada na Figura 3.18 não é segura para thread. O tempo das alternativas 2 e 3 da Seção 26.5 foi comparado com o exemplo da Figura 30.26. O aumento de velocidade da alternativa 3 para a alternativa 2 foi menor que 1%, provavelmente porque `readline` é utilizado somente para ler a contagem de cinco caracteres do cliente. Portanto, por simplicidade, utilizamos a versão menos eficiente da Figura 3.17 para os exemplos de servidor com threads deste capítulo.

30.11 Servidor TCP pré-threaded, um `accept` por thread

Anteriormente neste capítulo, verificamos que é mais rápido pré-bifurcar um pool de filhos do que criar um filho para cada cliente. Em um sistema que suporta threads, é razoável esperar um aumento de velocidade semelhante, por meio da criação de um pool de threads quando o servidor inicia, em vez de criar um novo thread para cada cliente. O projeto básico desse servidor é criar um pool de threads e, então, deixar cada thread chamar `accept`. Em vez de ter cada thread bloqueado na chamada de `accept`, utilizaremos um bloqueio de mutex (semelhante à Seção 30.8) que permite que somente um thread por vez chame `accept`. Não há nenhuma razão para utilizar bloqueio de arquivo para proteger a chamada de `accept` de todos os threads, pois com múltiplos threads em um único processo sabemos que um bloqueio de mutex pode ser utilizado.

A Figura 30.27 mostra o cabeçalho `pthread07.h` que define uma estrutura `Thread` que mantém algumas informações sobre cada thread.

Também declaramos alguns itens globais, como o descritor de soquete ouvinte e uma variável mutex que todos os threads precisam compartilhar.

server/pthread07.h

```
1 typedef struct {
2     pthread_t thread_tid;  /* ID de thread */
3     long  thread_count;    /* número de conexões tratadas */
4 } Thread;
5 Thread *tptr;              /* array de estruturas Thread; usando calloc */

6 int     listenfd, nthreads;
7 socklen_t addrlen;
8 pthread_mutex_t mlock;
```

server/pthread07.h

Figura 30.27 Cabeçalho `pthread07.h`.

A Figura 30.28 mostra a função `main`.

server/serv07.c

```
 1 #include   "unpthread.h"
 2 #include   "pthread07.h"

 3 pthread_mutex_t mlock = PTHREAD_MUTEX_INITIALIZER;

 4 int
 5 main(int argc, char **argv)
 6 {
 7     int    i;
 8     void   sig_int(int), thread_make(int);

 9     if(argc == 3)
10         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
11     else if (argc == 4)
12         listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
13     else
14         err_quit("usage: serv07 [ <host> ] <port#> <#threads>");
15     nthreads = atoi(argv[argc - 1]);
16     tptr = Calloc(nthreads, sizeof(Thread));
17     for (i = 0; i < nthreads; i++)
18         thread_make(i);          /* somente o thread principal retorna */
19     Signal(SIGINT, sig_int);
20     for ( ; ; )
21         pause();                 /* tudo feito por threads */
22 }
```

server/serv07.c

Figura 30.28 Função `main` do servidor TCP com pré-threading.

As funções `thread_make` e `thread_main` são mostradas na Figura 30.29.

server/pthread07.c

```
 1 #include   "unpthread.h"
 2 #include   "pthread07.h"

 3 void
 4 thread_make(int i)
 5 {
 6     void   *thread_main(void *);

 7     Pthread_create(&tptr[i].thread_tid, NULL, &thread_main, (void *) i);
 8     return;                       /* o thread principal retorna */
 9 }

10 void *
11 thread_main(void *arg)
12 {
13     int    connfd;
14     void   web_child(int);
15     socklen_t clilen;
16     struct sockaddr *cliaddr;

17     cliaddr = Malloc(addrlen);

18     printf("thread %d starting\n", (int) arg);
19     for ( ; ; ) {
20         clilen = addrlen;
21         Pthread_mutex_lock(&mlock);
22         connfd = Accept(listenfd, cliaddr, &clilen);
23         Pthread_mutex_unlock(&mlock);
24         tptr[(int) arg].thread_count++;

25         web_child(connfd);        /* solicitação de processo */
26         Close(connfd);
27     }
28 }
```

server/pthread07.c

Figura 30.29 Funções `thread_make` e `thread_main`.

Criação do thread

7 Cada thread é criado e executa a função `thread_main`. O único argumento é o número de índice do thread.

21–23 A função `thread_main` chama as funções `pthread_mutex_lock` e `pthread_mutex_unlock` em torno da chamada a `accept`.

Comparando as linhas 6 e 7 da Figura 30.1, vemos que esta última versão de nosso servidor é mais rápida que a versão que cria um thread por cliente. Esperávamos isso, pois criamos o pool de threads somente uma vez, quando o servidor inicia, em vez de criar um thread por cliente. De fato, essa versão do nosso servidor é a mais rápida nesses dois hosts.

A Figura 30.2 mostra a distribuição dos contadores `thread_count` na estrutura `Thread`, que imprimimos no handler `SIGINT` quando o servidor está terminado. A uniformidade dessa distribuição é causada pelo algoritmo de escalonamento de thread, que parece circular por todos os threads em ordem, ao escolher qual thread recebe o bloqueio de mutex.

> Em um kernel derivado do Berkeley, não precisamos de nenhum bloqueio em torno da chamada a `accept` e podemos fazer uma versão da Figura 30.29 sem qualquer bloqueio e desbloqueio de mutex. Fazer isso, entretanto, aumenta o tempo de CPU para controle de processos. Se examinarmos os dois componentes do tempo de CPU, o tempo de usuário e o tempo de sistema, sem qualquer bloqueio, veremos que o tempo de usuário diminui (porque o bloqueio é feito na biblioteca de threads, que executa no espaço do usuário), mas que o tempo de sistema aumenta (o estouro da manada do kernel, pois todos os

threads bloqueados em `accept` são acordados quando uma conexão chega). Como alguma forma de exclusão mútua é exigida para retornar cada conexão para um único thread, é mais rápido os próprios threads fazerem isso do que o kernel.

30.12 Servidor TCP pré-threaded, thread principal chama `accept`

Nosso projeto de servidor utilizando threads final faz o thread principal criar um pool de threads ao iniciar e, então, somente o thread principal chama `accept` e passa cada conexão de cliente para um dos threads disponíveis no pool. Isso é semelhante à versão de passagem de descritor da Seção 30.9.

O problema do projeto é: como o thread principal "passa" o soquete conectado para um dos threads disponíveis no pool? Há várias maneiras de implementar isso. Poderíamos utilizar passagem de descritor, como fizemos anteriormente, mas não há necessidade de passarmos um descritor de um thread para outro, pois todos os threads e todos os descritores estão no mesmo processo. Tudo que o thread receptor precisa saber é o número de descritor. A Figura 30.30 mostra o cabeçalho `pthread08.h` que define uma estrutura `Thread`, que é idêntica à Figura 30.27.

server/pthread08.h

```
1 typedef struct {
2    pthread_t thread_tid; /* ID de thread */
3    long   thread_count;  /* número de conexões tratadas */
4 } Thread;
5 Thread *tptr;            /* array de estruturas Thread; usando calloc */

6 #define MAXNCLI 32
7 int    clifd[MAXNCLI], iget, iput;
8 pthread_mutex_t clifd_mutex;
9 pthread_cond_t clifd_cond;
```

server/pthread08.h

Figura 30.30 Cabeçalho `pthread08.h`.

Definição do array compartilhado para conter soquetes conectados

6–9 Também definimos um array `clifd`, no qual o thread principal armazenará os descritores de soquete conectados. Os threads disponíveis no pool pegam um desses soquetes conectados e atendem o cliente correspondente. `iput` é o índice nesse array da próxima entrada a ser armazenada pelo thread principal e `iget` é o índice da próxima entrada a ser buscada por um dos threads do pool. Naturalmente, essa estrutura de dados que é compartilhada entre todos os threads deve ser protegida e utilizamos um mutex, junto com uma variável condicional.

A Figura 30.31 é a função `main`.

Criação do pool de threads

23–25 `thread_make` cria cada um dos threads.

Espera por cada conexão de cliente

27–38 O thread principal bloqueia na chamada a `accept`, esperando que cada conexão de cliente chegue. Quando uma chega, o soquete conectado é armazenado na próxima entrada no array `clifd`, após a obtenção do bloqueio de mutex no array. Também verificamos que o índice `iput` não acompanhou o índice `iget`, o que indica que nosso array não é suficientemente grande. A variável condicional é sinalizada e o mutex é liberado, permitindo que um dos threads do pool atenda esse cliente.

As funções `thread_make` e `thread_main` são mostradas na Figura 30.32. A primeira é idêntica à versão da Figura 30.29.

server/serv08.c

```
 1 #include   "unpthread.h"
 2 #include   "pthread08.h"
 3 static int nthreads;
 4 pthread_mutex_t clifd_mutex = PTHREAD_MUTEX_INITIALIZER;
 5 pthread_cond_t clifd_cond = PTHREAD_COND_INITIALIZER;
 6 int
 7 main(int argc, char **argv)
 8 {
 9     int     i, listenfd, connfd;
10     void    sig_int(int), thread_make(int);
11     socklen_t addrlen, clilen;
12     struct sockaddr *cliaddr;
13     if (argc == 3)
14         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
15     else if (argc == 4)
16         listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
17     else
18         err_quit("usage: serv08 [ <host> ] <port#> <#threads>");
19     cliaddr = Malloc(addrlen);
20     nthreads = atoi(argv[argc - 1]);
21     tptr = Calloc(nthreads, sizeof(Thread));
22     iget = iput = 0;
23         /* cria todos os threads */
24     for (i = 0; i < nthreads; i++)
25         thread_make(i);           /* somente o thread principal retorna */
26     Signal(SIGINT, sig_int);
27     for ( ; ; ) {
28         clilen = addrlen;
29         connfd = Accept(listenfd, cliaddr, &clilen);
30         Pthread_mutex_lock(&clifd_mutex);
31         clifd[iput] = connfd;
32         if (++iput == MAXNCLI)
33             iput = 0;
34         if (iput == iget)
35             err_quit("iput = iget = %d", iput);
36         Pthread_cond_signal(&clifd_cond);
37         Pthread_mutex_unlock(&clifd_mutex);
38     }
39 }
```

server/serv08.c

Figura 30.31 Função `main` para o servidor com pré-threading.

Espera até que o descritor de cliente atenda

17–26 Cada thread do pool tenta obter um bloqueio no mutex que protege o array `clifd`. Quando o bloqueio é obtido, não há nada a fazer, caso os índices `iget` e `iput` sejam iguais. Nesse caso, o thread é posto para dormir pela chamada a `pthread_cond_wait`. Ele será acordado pela chamada a `pthread_cond_signal` no thread principal, depois que uma conexão é aceita. Quando o thread obtém uma conexão, ele chama `web_child`.

Os tempos da Figura 30.1 mostram que esse servidor é mais lento que o da seção anterior, no qual cada thread chamava `accept` após obter um bloqueio de mutex. A razão é que o exemplo desta seção exige tanto um mutex como uma variável condicional, comparado com somente um mutex da Figura 30.29.

Se examinarmos o histograma do número de clientes atendidos por cada thread do pool, veremos que ele é semelhante à última coluna da Figura 30.2. Isso significa que a biblioteca

de threads circula por todos os threads disponíveis ao realizar a ativação, baseada na variável condicional quando o thread principal chama `pthread_cond_signal`.

server/pthread08.c

```
 1 #include   "unpthread.h"
 2 #include   "pthread08.h"

 3 void
 4 thread_make(int i)
 5 {
 6     void   *thread_main(void *);

 7     Pthread_create(&tptr[i].thread_tid, NULL, &thread_main, (void *) i);
 8     return;                       /* o thread principal retorna */
 9 }

10 void *
11 thread_main(void *arg)
12 {
13     int    connfd;
14     void   web_child(int);

15     printf("thread %d starting\n", (int) arg);
16     for ( ; ; ) {
17         Pthread_mutex_lock(&clifd_mutex);
18         while (iget == iput)
19             Pthread_cond_wait(&clifd_cond, &clifd_mutex);
20         connfd = clifd[iget];     /* soquete conectado a atender */
21         if (++iget == MAXNCLI)
22             iget = 0;
23         Pthread_mutex_unlock(&clifd_mutex);
24         tptr[(int) arg].thread_count++;

25         web_child(connfd);        /* solicitação de processo */
26         Close(connfd);
27     }
28 }
```

server/pthread08.c

Figura 30.32 Funções `thread_make` e `thread_main`.

30.13 Resumo

Neste capítulo, vimos nove projetos de servidor diferentes e os executamos todos com o mesmo cliente no estilo Web, comparando a quantidade de tempo de CPU gasta realizando controle de processos:

0. Servidor iterativo (medida da linha de base; nenhum controle de processos)
1. Servidor concorrente, um `fork` por cliente
2. Pré-bifurcado, com cada filho chamando `accept`
3. Pré-bifurcado, com bloqueio de arquivo para proteger `accept`
4. Pré-bifurcado, com bloqueio mutex de thread para proteger `accept`
5. Pré-bifurcado, com o pai passando o descritor de soquete para o filho
6. Servidor concorrente, cria um thread por solicitação de cliente
7. Pré-threading com bloqueio mutex para proteger `accept`
8. Pré-threading com o thread principal chamando `accept`

Podemos fazer alguns comentários de resumo:

- Primeiro, se o servidor não é demasiadamente utilizado, o modelo de servidor concorrente tradicional, com um `fork` por cliente, serve bem. Isso pode até ser combinado com `inetd`, permitindo tratar da aceitação de cada conexão. O restante de nossos comentários se destina aos servidores pesadamente usados, como os servidores Web.
- Criar um pool de filhos ou um pool de threads reduz o tempo de CPU para controle de processos, comparado com o projeto tradicional de um `fork` por cliente, por um fator de 10 ou mais. A codificação não é complicada, mas o que é exigido, muito além dos exemplos que mostramos, é o monitoramento do número de filhos livres e aumentar ou diminuir esse número à medida que o número de clientes sendo atendidos mudar dinamicamente.
- Algumas implementações permitem que vários filhos ou threads sejam bloqueados em uma chamada a `accept`, enquanto em outras devemos colocar algum tipo de bloqueio em torno da chamada de `accept`. Um bloqueio de arquivo ou de mutex Pthread pode ser utilizado.
- Fazer com que todos os filhos ou threads chamem `accept` normalmente é mais simples e rápido do que fazer o thread principal chamar `accept` e, então, passar o descritor para o filho ou para o thread.
- Fazer com que todos os filhos ou threads bloqueiem em uma chamada a `accept` é preferível em relação ao bloqueio em uma chamada a `select`, por causa da possibilidade de colisões de `select`.
- Usar threads normalmente é mais rápido do que utilizar processos. Mas a escolha de um filho ou de um thread por cliente depende do que o SO fornece e também pode depender de quais outros programas, se houver, são ativados para atender cada cliente. Por exemplo, se o servidor que aceita a conexão do cliente chama `fork` e `exec`, pode ser mais rápido bifurcar um processo com um único thread do que com múltiplos threads.

Exercícios

30.1 Na Figura 30.13, por que o pai mantém o soquete ouvinte aberto, em vez de fechá-lo depois que todos os filhos são criados?

30.2 Você pode recodificar o servidor da Seção 30.9 para utilizar um soquete de datagrama de domínio Unix, em vez de um soquete de fluxo de domínio Unix? O que muda?

30.3 Execute o cliente e quantos servidores seu ambiente suportar e compare seus resultados com aqueles informados neste capítulo.

Capítulo 31

STREAMS

31.1 Introdução

Neste capítulo, forneceremos uma visão geral do sistema STREAMS e das funções utilizadas por uma aplicação para acessar um fluxo. Nosso objetivo é entender a implementação dos protocolos de rede dentro do framework STREAMS. Também desenvolveremos um cliente TCP simples, utilizando a Interface de Provedor de Transporte (TPI), a interface na camada de transporte que os soquetes normalmente utilizam em um sistema baseado no STREAMS. Informações adicionais sobre o STREAMS, inclusive sobre escrever rotinas de kernel que o utilizem, podem ser encontradas em (Rago, 1993).

> O STREAMS foi projetado por Dennis Ritchie (Ritchie, 1984) e se tornou amplamente disponível pela primeira vez com o SVR3, em 1986. A especificação POSIX define o STREAMS como um *grupo de opções*, o que significa que um sistema não pode implementar STREAMS, mas, se o fizer, a implementação deverá obedecer à especificação POSIX. Todo sistema derivado do System V deve oferecer suporte a POSIX, mas as várias versões do 4.xBSD não fazem isso.
>
> Deve-se ter cuidado ao distinguir entre o STREAMS, o sistema de E/S de fluxo que estamos descrevendo neste capítulo, e os "fluxos de E/S-padrão". Este último termo é utilizado ao se falar sobre a biblioteca de E/S-padrão (por exemplo, funções como `fopen`, `fgets`, `printf` e outras).

31.2 Visão geral

Os STREAMS fornecem uma conexão full-duplex entre um processo e um *driver*, conforme mostrado na Figura 31.1. Embora descrevamos a caixa da parte inferior como um driver, isso não precisa ser associado a um dispositivo de hardware; também pode ser um driver de pseudodispositivo (por exemplo, um driver de software).

O *topo do fluxo* (*stream head*) consiste nas rotinas do kernel que são ativadas quando a aplicação faz uma chamada de sistema para um descritor STREAMS (por exemplo, `read`, `putmsg`, `ioctl` e outros).

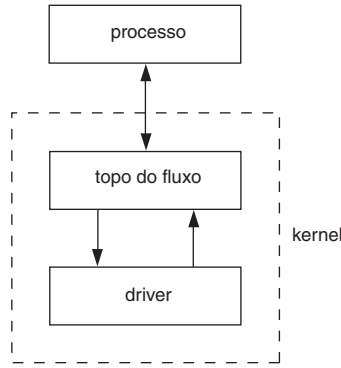


Figura 31.1 Um fluxo entre um processo e um driver.

Um processo pode adicionar e remover dinamicamente *módulos* de processamento intermediários entre o topo do fluxo e o driver. Um módulo realiza algum tipo de filtragem nas mensagens que passam em um fluxo. Mostramos isso na Figura 31.2.

Qualquer número de módulos pode ser empilhado em um fluxo. Quando dizemos "*empilhado*", queremos dizer que cada novo módulo é inserido imediatamente abaixo do topo do fluxo.

Um tipo especial de driver de pseudodispositivo é um *multiplexador*, que aceita dados de múltiplas fontes. Uma implementação baseada em STREAMS do conjunto de protocolos TCP/IP, como encontrado no SVR4, por exemplo, poderia ser configurada como mostra a Figura 31.3.

- Quando um soquete é criado, o módulo `sockmod` é empilhado no fluxo pela biblioteca de soquetes. É a combinação da biblioteca de soquetes e do módulo STREAMS `sockmod` que fornece a API de soquetes para o processo.
- Quando uma extremidade XTI é criada, o módulo `timod` é empilhado no fluxo pela biblioteca XTI. É a combinação da biblioteca XTI e do módulo STREAMS `timod` que fornece a API de Interface de Transporte X/Open (XTI) para o processo.

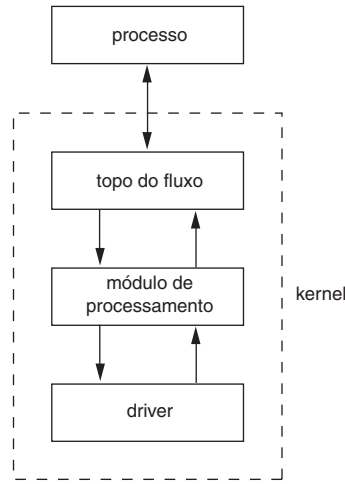


Figura 31.2 Um fluxo com um módulo de processamento.

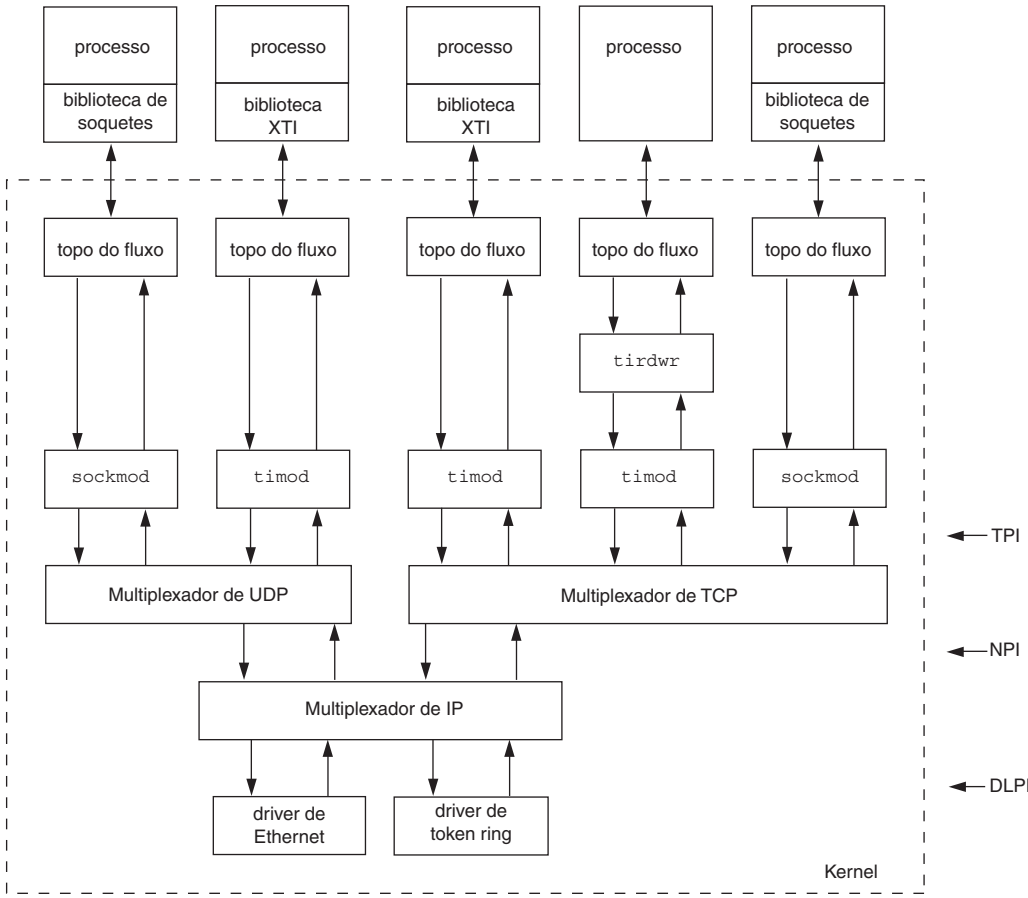


Figura 31.3 Uma possível implementação de TCP/IP utilizando STREAMS.

Estas é uma das poucas menções que fazemos do XTI. Uma edição anterior deste livro descreveu detalhadamente a API XTI, mas ela deixou de ser utilizada e até a especificação POSIX não a aborda mais; portanto, omitimos sua abordagem neste livro. A Figura 31.3 mostra onde normalmente fica a implementação XTI, mas pouco falaremos sobre ela, pois raramente há uma razão para utilizá-la.

- O módulo STREAMS `tirdwr` normalmente deve ser empilhado em um fluxo para utilizar `read` e `write` com uma extremidade XTI. O processo do meio que utiliza TCP, na Figura 31.3, fez isso. Fazendo isso, esse processo provavelmente abandonou o uso de XTI; portanto, não mostramos a biblioteca de XTI lá.
- Várias interfaces de serviço definem o formato das mensagens de rede trocadas em um fluxo. Descrevemos as três mais comuns. O TPI (Unix International, 1992b) define a interface fornecida por um provedor de camada de transporte (por exemplo, TCP e UDP) para os módulos acima dele. A *Network Provider Interface* (NPI) (Unix International, 1992a) define a interface fornecida por um provedor de camada de rede (por exemplo, IP). DLPI é a *Data Link Provider Interface* (Unix International, 1991). Uma referência alternativa para TPI e DLPI, que contém exemplo de código C, é Rago (1993).

Cada componente em um fluxo – o topo, os módulos de processamento e o driver – contém pelo menos um par de *filas*: uma fila de gravação e uma de leitura. Mostramos isso na Figura 31.4.

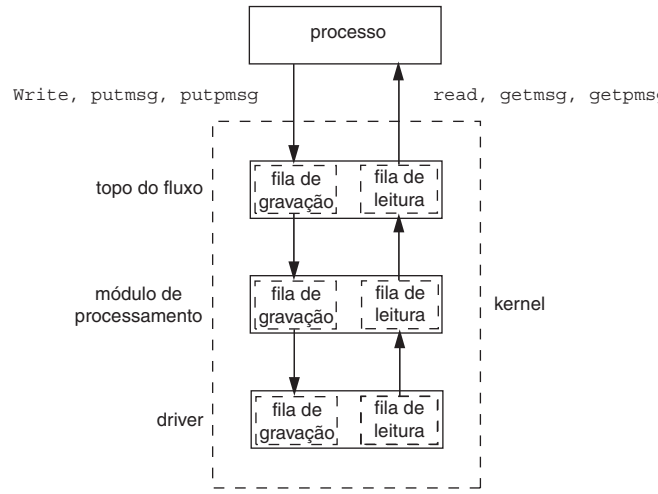


Figura 31.4 Cada componente em um fluxo tem pelo menos um par de filas.

Tipos de mensagem

As mensagens STREAMS podem ser classificadas como de *prioridade alta*, *banda de prioridade* ou *normais*. Há 256 bandas de prioridade diferentes, entre 0 e 255, com as mensagens normais na banda 0. A prioridade de uma mensagem STREAMS é utilizada para enfileiramento e controle de fluxo. Por convenção, as mensagens de alta prioridade não são afetadas pelo controle de fluxo.

A Figura 31.5 mostra a ordenação das mensagens em determinada fila.

Embora o sistema STREAMS suporte 256 bandas de prioridade diferentes, os protocolos de rede freqüentemente utilizam a banda 1 para dados urgentes e a banda 0 para dados normais.

> Os dados fora da banda do TCP não são considerados como verdadeiros dados urgentes pelo TPI. De fato, o TCP utiliza a banda 0 tanto para dados normais como para seus dados de fora da banda. O uso da banda 1 para dados urgentes serve para protocolos nos quais esses dados (não apenas o ponteiro urgente, como no TCP) são enviados adiante dos dados normais.
>
> Deve-se ter cuidado com o termo "*normal*". Nas versões anteriores ao SVR4, não havia bandas de prioridade; havia somente mensagens normais e mensagens prioritárias. O SVR4 implementava bandas de prioridade, exigindo as funções `getpmsg` e `putpmsg`, que descreveremos em breve. As antigas mensagens prioritárias foram denominadas de

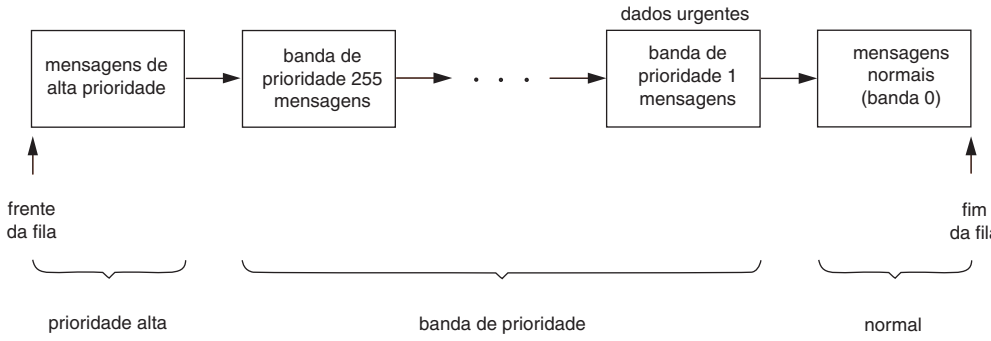


Figura 31.5 Ordenação de mensagens STREAMS em uma fila, com base na prioridade.

alta prioridade. A questão é como chamar as novas mensagens, com bandas de prioridade entre 1 e 255. A terminologia comum (Rago, 1993) refere-se a tudo que não seja mensagem de alta prioridade como mensagens de prioridade normal e, então, subdivide essas mensagens de prioridade normal em bandas de prioridade. O termo "*mensagem normal*" sempre deve se referir a uma mensagem com uma banda 0.

Embora falemos sobre mensagens de prioridade normal e mensagens de alta prioridade, existe cerca de uma dezena de tipos de mensagem de prioridade normal e aproximadamente 18 tipos de mensagem de alta prioridade. Da perspectiva da aplicação e das funções `getmsg` e `putmsg` que estamos para descrever, estamos interessados somente em três tipos de mensagens diferentes: `M_DATA`, `M_PROTO` e `M_PCPROTO` (`PC` significa "controle de prioridade" e implica uma mensagem de alta prioridade). A Figura 31.6 mostra como esses três tipos de mensagem diferentes são gerados pelas funções `write` e `putmsg`.

Função	Controle?	Dados?	Flag	Tipo de mensagem gerada
`write`		Sim		`M_DATA`
`putmsg`	Não	Sim	`0`	`M_DATA`
`putmsg`	Sim	Não importa	`0`	`M_PROTO`
`putmsg`	Sim	Não importa	`MSG_HIPRI`	`M_PCPROTO`

Figura 31.6 Tipos de mensagem STREAMS geradas por `write` e `putmsg`.

31.3 Funções `getmsg` e `putmsg`

Os dados transferidos para cima e para baixo em um fluxo consistem em mensagens e cada mensagem contém *controle*, *dados* ou ambos. Se utilizarmos `read` e `write` em um fluxo, eles transferirão somente dados. Para permitir que um processo leia e grave tanto dados como informações de controle, duas novas funções foram adicionadas.

```
#include <stropts.h>

int getmsg(int fd, struct strbuf *ctlptr, struct strbuf *dataptr, int *flagsp);

int putmsg(int fd, const struct strbuf *ctlptr,
           const struct strbuf *dataptr, int flags);
```

As duas retornam: valor não-negativo se OK (veja o texto), –1 em caso de erro

As partes referentes ao controle e aos dados da mensagem são descritas pela estrutura `strbuf` a seguir:

```
struct strbuf {
   int   maxlen;     /* tamanho máximo de buf */
   int   len;        /* volume real de dados em buf */
   char *buf;        /* dados */
};
```

Note a semelhança entre as estruturas `strbuf` e `netbuf`. Os nomes dos três elementos em cada uma delas são idênticos.

Mas os dois comprimentos em `netbuf` são inteiros sem sinal, enquanto em `strbuf` são inteiros com sinal. A razão disso é que algumas das funções STREAMS utilizam um valor de `len` ou `maxlen` igual a –1 para indicar algo especial.

Podemos enviar somente informações de controle, somente dados ou ambos, utilizando `putmsg`. Para indicar a ausência de informações de controle, podemos especificar *ctlptr* co-

mo um ponteiro nulo ou configurar *ctlptr->len* como –1. A mesma técnica é utilizada para indicar ausência de dados.

Se não há informações de controle, uma mensagem `M_DATA` é gerada por `putmsg` (Figura 31.6); caso contrário, uma mensagem `M_PROTO` ou `M_PCPROTO` é gerada, dependendo de *flags*. O argumento *flags* de `putmsg` é 0 para uma mensagem normal ou `RS_HIPRI` para uma mensagem de alta prioridade.

O último argumento de `getmsg` é de valor-resultado. Se o inteiro apontado por *flagsp* é 0, quando a função é chamada, a primeira mensagem no fluxo é retornada (pode ser normal ou de alta prioridade). Se o valor inteiro é `RS_HIPRI`, quando a função é chamada, a função espera que uma mensagem de alta prioridade chegue ao topo do fluxo. Em ambos os casos, o valor armazenado no inteiro apontado por *flagsp* será 0 ou `RS_HIPRI`, dependendo do tipo de mensagem retornado.

Supondo que passemos valores de *ctlptr* e *dataptr* não-nulos para `getmsg`, se não houver informações de controle para retornar (isto é, uma mensagem `M_DATA` está sendo retornada), isso é indicado pela configuração de *ctlptr->len* como –1 no retorno. De maneira semelhante, *dataptr->len* é configurado como –1, se não houver dados para retornar.

O valor de retorno de `putmsg` é 0, se tudo estiver correto, ou –1, em caso de erro. Mas `getmsg` retorna 0 somente se a mensagem inteira foi retornada para o chamador. Se o buffer de controle é pequeno demais para todas as informações de controle, o valor de retorno é `MORECTL` (que garantidamente é não-negativo). De maneira semelhante, se o buffer de dados é pequeno demais, `MOREDATA` pode ser retornado. Se ambos são pequenos demais, o valor da função lógica OU desses dois flags é retornado.

31.4 Funções `getpmsg` e `putpmsg`

Quando o suporte para bandas de prioridade diferentes foi adicionado aos STREAMS com o SVR4, as duas variantes de `getmsg` e `putmsg` a seguir também foram adicionadas:

```
#include <stropts.h>

int getpmsg(int fd, struct strbuf *ctlptr,
            struct strbuf *dataptr, int *bandp, int *flagsp);

int putpmsg(int fd, const struct strbuf *ctlptr,
            const struct strbuf *dataptr, int band, int flags);
```

As duas retornam: valor não-negativo se OK, –1 em erro

O argumento *band* de `putpmsg` deve estar entre 0 e 255, inclusive. Se o argumento *flags* é `MSG_BAND`, então uma mensagem é gerada na banda de prioridade especificada. Configurar *flags* como `MSG_BAND` e especificar uma banda de 0 é equivalente a chamar `putmsg`. Se *flags* é `MSG_HIPRI`, *band* deve ser 0, e uma mensagem de alta prioridade é gerada. (Observe que esse flag é nomeado diferentemente do flag `RS_HIPRI` de `putmsg`.)

Os dois inteiros apontados por *bandp* e *flagsp* são argumentos de valor-resultado de `getpmsg`. O inteiro apontado por *flagsp* de `getpmsg` pode ser `MSG_HIPRI` (para ler uma mensagem de alta prioridade), `MSG_BAND` (para ler uma mensagem cuja banda de prioridade é pelo menos igual ao inteiro apontado por *bandp*) ou `MSG_ANY` (para ler qualquer mensagem). No retorno, o inteiro apontado por *bandp* contém a banda da mensagem que foi lida e o inteiro apontado por *flagsp* contém `MSG_HIPRI` (se uma mensagem de alta prioridade foi lida) ou `MSG_BAND` (se alguma outra mensagem foi lida).

31.5 Função `ioctl`

Com STREAMS, encontramos novamente a função `ioctl` que foi descrita no Capítulo 17.

```
#include <stropts.h>

int ioctl(int fd, int request, ... /* void *arg */ );
```

Retorna: 0 se OK, –1 em erro

A única alteração em relação ao protótipo de função mostrado na Seção 17.2 são os cabeçalhos que devem ser incluídos ao se tratar com STREAMS.

Há cerca de 30 solicitações que afetam o topo do fluxo. Cada solicitação inicia com `I_` e normalmente elas são documentados na página man de `streamio`.

31.6 Transport Provider Interface (TPI)

Na Figura 31.3, mostramos que TPI é a interface de serviço para a camada de transporte a partir de cima. Tanto soquetes como XTI utilizam essa interface em um ambiente STREAMS. Na Figura 31.3, é uma combinação da biblioteca de soquetes e `sockmod`, junto com uma combinação da biblioteca de XTI e `timod`, que troca mensagens TPI com TCP e UDP.

A TPI é uma interface *baseada em mensagens*. Ela define as mensagens que são trocadas fluxo acima e fluxo abaixo entre a aplicação (por exemplo, a biblioteca de soquetes) e a camada de transporte: o formato dessas mensagens e qual operação cada mensagem realiza. Em muitos casos, a aplicação envia uma solicitação para o provedor (como "vincule esse endereço local") e este envia de volta uma resposta ("OK" ou "erro"). Alguns eventos ocorrem assincronamente no provedor (a chegada de uma solicitação de conexão para um servidor), fazendo uma mensagem ou um sinal ser enviado fluxo acima.

Podemos ignorar os soquetes e a XTI e usar a TPI diretamente. Nesta seção, reescreveremos nosso cliente de data/hora simples, mas com a TPI em vez de soquetes (Figura 1.5). Utilizando linguagens de programação como analogia, usar soquetes é como programar em uma linguagem de alto nível, como C ou Pascal, enquanto usar a TPI diretamente é como programar em linguagem assembly. Não estamos defendendo o uso da TPI diretamente em aplicações reais. Mas examinar como ela funciona e desenvolver esse exemplo nos dá um entendimento melhor de como a biblioteca de soquetes funciona em um ambiente STREAMS.

A Figura 31.7 é nosso cabeçalho `tpi_daytime.h`.

Precisamos incluir um cabeçalho STREAMS adicional, junto com `<sys/tihdr.h>`, que contém as definições das estruturas para todas as mensagens TPI.

A Figura 31.8 é a função `main` de nosso cliente de data/hora.

streams/tpi_daytime.h

```
1 #include   "unpxti.h"
2 #include   <sys/stream.h>
3 #include   <sys/tihdr.h>

4 void    tpi_bind(int, const void *, size_t);
5 void    tpi_connect(int, const void *, size_t);
6 ssize_t tpi_read(int, void *, size_t);
7 void    tpi_close(int);
```

streams/tpi_daytime.h

Figura 31.7 Nosso cabeçalho `tpi_daytime.h`.

streams/tpi_daytime.c

```
 1 #include   "tpi_daytime.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    fd, n;
 6     char   recvline[MAXLINE + 1];
 7     struct sockaddr_in myaddr, servaddr;

 8     if(argc != 2)
 9        err_quit("usage: tpi_daytime <IPaddress>");

10     fd = Open(XTI_TCP, O_RDWR, 0);

11         /* vincula qualquer endereço local */
12     bzero(&myaddr, sizeof(myaddr));
13     myaddr.sin_family = AF_INET;
14     myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
15     myaddr.sin_port = htons(0);

16     tpi_bind(fd, &myaddr, sizeof(struct sockaddr_in));

17         /* preenche o endereço do servidor */
18     bzero(&servaddr, sizeof(servaddr));
19     servaddr.sin_family = AF_INET;
20     servaddr.sin_port = htons(13); /* servidor de data/hora */
21     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

22     tpi_connect(fd, &servaddr, sizeof(struct sockaddr_in));

23     for ( ; ; ) {
24         if ( (n = tpi_read(fd, recvline, MAXLINE)) <= 0) {
25             if (n == 0)
26                 break;
27             else
28                 err_sys("tpi_read error");
29         }
30         recvline[n] = 0;          /* termina com nulo */
31         fputs(recvline, stdout);
32     }
33     tpi_close(fd);
34     exit(0);
35 }
```

streams/tpi_daytime.c

Figura 31.8 Função `main` de nosso cliente de data/hora escrita para TPI.

Abertura do provedor de transporte, vinculação do endereço local

10–16 Abrimos o dispositivo correspondente ao provedor de transporte (normalmente, `/dev/tcp`). Preenchemos uma estrutura de endereço de soquete de Internet com `INADDR_ANY` e uma porta 0, dizendo ao TCP para que vincule qualquer endereço local com nosso ponto final. Chamamos nossa própria função `tpi_bind` (mostrada em breve) para realizar o vínculo.

Preenchimento do endereço do servidor, estabelecimento da conexão

17–22 Preenchemos outra estrutura de endereço de soquete de Internet com o endereço IP do servidor (extraído da linha de comando) e a porta (13). Chamamos nossa função `tpi_connect` para estabelecer a conexão.

Leitura dos dados do servidor, cópia na saída-padrão

23–33 Como em nossos outros clientes de data/hora, simplesmente copiamos os dados da conexão na saída-padrão, parando quando recebemos o sinal de EOF do servidor (por exemplo, o FIN). Então, chamamos nossa função `tpi_close` para fechar nosso ponto final.

Nossa função `tpi_bind` é mostrada na Figura 31.9.

streams/tpi_bind.c

```
 1 #include   "tpi_daytime.h"

 2 void
 3 tpi_bind(int fd, const void *addr, size_t addrlen)
 4 {
 5     struct {
 6         struct T_bind_req msg_hdr;
 7         char   addr[128];
 8     } bind_req;
 9     struct {
10         struct T_bind_ack msg_hdr;
11         char     addr[128];
12     } bind_ack;
13     struct strbuf ctlbuf;
14     struct T_error_ack *error_ack;
15     int      flags;

16     bind_req.msg_hdr.PRIM_type = T_BIND_REQ;
17     bind_req.msg_hdr.ADDR_length = addrlen;
18     bind_req.msg_hdr.ADDR_offset = sizeof(struct T_bind_req);
19     bind_req.msg_hdr.CONIND_number = 0;
20     memcpy(bind_req.addr, addr, addrlen);       /* sockaddr_in{} */

21     ctlbuf.len = sizeof(struct T_bind_req) + addrlen;
22     ctlbuf.buf = (char *) &bind_req;
23     Putmsg(fd, &ctlbuf, NULL, 0);

24     ctlbuf.maxlen = sizeof(bind_ack);
25     ctlbuf.len = 0;
26     ctlbuf.buf = (char *) &bind_ack;
27     flags = RS_HIPRI;
28     Getmsg(fd, &ctlbuf, NULL, &flags);

29     if (ctlbuf.len < (int) sizeof(long))
30         err_quit("bad length from getmsg");

31     switch (bind_ack.msg_hdr.PRIM_type) {
32     case T_BIND_ACK:
33         return;

34     case T_ERROR_ACK:
35         if (ctlbuf.len < (int) sizeof(struct T_error_ack))
36             err_quit("bad length for T_ERROR_ACK");
37         error_ack = (struct T_error_ack *) &bind_ack.msg_hdr;
38         err_quit("T_ERROR_ACK from bind (%d, %d)",
39                  error_ack->TLI_error, error_ack->UNIX_error);

40     default:
41         err_quit("unexpected message type: %d", bind_ack.msg_hdr.PRIM_type);
42     }
43 }
```

streams/tpi_bind.c

Figura 31.9 Função `tpi_bind`: vincula um endereço local a uma extremidade.

Preenchimento da estrutura `T_bind_req`

16–20 O cabeçalho `<sys/tihdr.h>` define a estrutura `T_bind_req`.

```
struct T_bind_req {
   t_scalar_t    PRIM_type;       /* T_BIND_REQ */
```

```
    t_scalar_t    ADDR_length;     /* comprimento do endereço */
    t_scalar_t    ADDR_offset;     /* deslocamento do endereço */
    t_uscalar_t   CONIND_number;   /* indicações de conexão solicitadas */
          /* seguido pelo endereço de protocolo para vincular */
};
```

Todas as solicitações de TPI são definidas como uma estrutura que inicia com um campo de tipo inteiro longo. Definimos nossa própria estrutura `bind_req` que inicia com a estrutura `T_bind_req`, seguida de um buffer que contém o endereço local a ser vinculado. A TPI não diz nada sobre o conteúdo desse buffer; ele é definido pelo provedor. Os provedores de TCP esperam que esse buffer contenha uma estrutura `sockaddr_in`.

Preenchemos a estrutura `T_bind_req`, configurando o membro `ADDR_length` com o tamanho do endereço (16 bytes para uma estrutura de endereço de soquete de Internet) e `ADDR_offset` para o deslocamento de byte do endereço (ele vem imediatamente após a estrutura `T_bind_req`). Não temos certeza de que essa localização está adequadamente alinhada para a estrutura `sockaddr_in` que está armazenada lá; portanto, chamamos `memcpy` para copiar a estrutura do chamador em nossa estrutura `bind_req`. Configuramos `CONIND_number` como 0, porque somos um cliente e não um servidor.

Chamada a `putmsg`

21–23 A TPI exige que a estrutura que acabamos de construir seja passada para o provedor como uma mensagem `M_PROTO`. Portanto, chamamos `putmsg`, especificando nossa estrutura `bind_req` como as informações de controle, sem dados e com um flag igual a 0.

Chamada a `getmsg` para ler mensagens de alta prioridade

24–30 A resposta para nossa solicitação `T_BIND_REQ` será uma mensagem `T_BIND_ACK` ou `T_ERROR_ACK`. Essas mensagens de reconhecimento são enviadas como de alta prioridade (`M_PCPROTO`); portanto, as lemos utilizando `getmsg` com um flag igual a `RS_HIPRI`. Como a resposta é uma mensagem de alta prioridade, ela ultrapassará todas as mensagens de prioridade normal no fluxo.

Essas duas mensagens são as seguintes:

```
struct T_bind_ack {
    t_scalar_t     PRIM_type;      /* T_BIND_ACK */
    t_scalar_t     ADDR_length;    /* comprimento do endereço */
    t_scalar_t     ADDR_offset;    /* deslocamento do endereço */
    t_uscalar_t    CONIND_number; /* indica a conexão a ser enfileirada */
          /* seguida pelo endereço de vínculo */
};
struct T_error_ack {
    t_scalar_t     PRIM_type;      /* T_ERRO_ACK */
    t_scalar_t     ERROR_prim      /* primitiva com erro */
    t_scalar_t     TLI_error;      /* código de erro TLI */
    t_scalar_t     UNIX_error;     /* código de erro UNIX */
};
```

Todas essas mensagens iniciam com o tipo; portanto, podemos ler a resposta supondo que seja uma mensagem `T_BIND_ACK`, ver o tipo e processar a mensagem de acordo. Não esperamos quaisquer dados do provedor; portanto, especificamos um ponteiro nulo como terceiro argumento de `getmsg`.

> Quando verificamos se o volume das informações de controle retornado é pelo menos o tamanho de um inteiro longo, devemos ter o cuidado ao capturar o valor de `sizeof` em um inteiro. O operador `sizeof` retorna um valor inteiro sem sinal, mas é possível que o campo `len` retornado seja –1. Mas como a comparação menor-do-que está comparando um valor com sinal à esquerda com um valor sem sinal à direita, o compilador trata o valor com sinal como um valor sem sinal. Em uma arquitetura de complemento de dois, –1,

considerado como um valor sem sinal, é muito grande, o que faz com que seja maior do que 4 (se supormos que um inteiro longo ocupa 4 bytes).

Processamento da resposta

31–33 Se a resposta é `T_BIND_ACK`, o vínculo foi bem-sucedido e retornamos. O endereço real que foi vinculado ao ponto final é retornado no membro `addr` de nossa estrutura `bind_ack`, o qual ignoramos.

34–39 Se a resposta é `T_ERROR_ACK`, verificamos se a mensagem inteira foi recebida e, então, imprimimos os três valores de retorno na estrutura. Nesse programa simples, terminamos quando um erro ocorre; não retornamos para o chamador.

Podemos ver esses erros da solicitação de vínculo, alterando nossa função `main` para vincular alguma porta que não seja a 0. Por exemplo, se tentarmos vincular a porta 1 (que exige privilégios de superusuário, pois é menor que 1024), obteremos

```
solaris % tpi_daytime 127.0.0.1
T_ERROR_ACK from bind (3, 0)
```

O erro `TACCES` tem o valor 3 nesse sistema. Se alterarmos a porta para um valor maior que 1023, mas que esteja correntemente em uso por outro ponto final de TCP, obteremos

```
solaris % tpi_daytime 127.0.0.1
T_ERROR_ACK from bind (23, 0)
```

O erro `TADDRBUSY` tem o valor 23 nesse sistema.

A próxima função, mostrada na Figura 31.10, é `tpi_connect`, que estabelece a conexão com o servidor.

streams/tpi_connect.c

```
 1 #include   "tpi_daytime.h"

 2 void
 3 tpi_connect(int fd, const void *addr, size_t addrlen)
 4 {
 5     struct {
 6         struct T_conn_req msg_hdr;
 7         char   addr[128];
 8     } conn_req;
 9     struct {
10         struct T_conn_con msg_hdr;
11         char addr[128];
12     } conn_con;
13     struct strbuf ctlbuf;
14     union T_primitives rcvbuf;
15     struct T_error_ack *error_ack;
16     struct T_discon_ind *discon_ind;
17     int    flags;

18     conn_req.msg_hdr.PRIM_type = T_CONN_REQ;
19     conn_req.msg_hdr.DEST_length = addrlen;
20     conn_req.msg_hdr.DEST_offset = sizeof(struct T_conn_req);
21     conn_req.msg_hdr.OPT_length = 0;
22     conn_req.msg_hdr.OPT_offset = 0;
23     memcpy(conn_req.addr, addr, addrlen);       /* sockaddr_in{} */

24     ctlbuf.len = sizeof(struct T_conn_req) + addrlen;
25     ctlbuf.buf = (char *) &conn_req;
26     Putmsg(fd, &ctlbuf, NULL, 0);

27     ctlbuf.maxlen = sizeof(union T_primitives);
```

Figura 31.10 Função `tpi_connect`: estabelece uma conexão com o servidor (*continua*).

```
28      ctlbuf.len = 0;
29      ctlbuf.buf = (char *) &rcvbuf;
30      flags = RS_HIPRI;
31      Getmsg(fd, &ctlbuf, NULL, &flags);

32      if (ctlbuf.len < (int) sizeof(long))
33          err_quit("tpi_connect: bad length from getmsg");

34      switch (rcvbuf.type) {
35      case T_OK_ACK:
36          break;

37      case T_ERROR_ACK:
38          if (ctlbuf.len < (int) sizeof(struct T_error_ack))
39              err_quit("tpi_connect: bad length for T_ERROR_ACK");
40          error_ack = (struct T_error_ack *) &rcvbuf;
41          err_quit("tpi_connect: T_ERROR_ACK from conn (%d, %d)",
42                   error_ack->TLI_error, error_ack->UNIX_error);

43      default:
44          err_quit("tpi_connect: unexpected message type: %d", rcvbuf.type);
45      }

46      ctlbuf.maxlen = sizeof(conn_con);
47      ctlbuf.len = 0;
48      ctlbuf.buf = (char *) &conn_con;
49      flags = 0;
50      Getmsg(fd, &ctlbuf, NULL, &flags);

51      if (ctlbuf.len < (int) sizeof(long))
52          err_quit("tpi_connect2: bad length from getmsg");

53      switch (conn_con.msg_hdr.PRIM_type) {
54      case T_CONN_CON:
55          break;

56      case T_DISCON_IND:
57          if (ctlbuf.len < (int) sizeof(struct T_discon_ind))
58              err_quit("tpi_connect2: bad length for T_DISCON_IND");
59          discon_ind = (struct T_discon_ind *) &conn_con.msg_hdr;
60          err_quit("tpi_connect2: T_DISCON_IND from conn (%d)",
61                   discon_ind->DISCON_reason);

62      default:
63          err_quit("tpi_connect2: unexpected message type: %d",
64                   conn_con.msg_hdr.PRIM_type);
65      }
66  }
```

streams/tpi_connect.c

Figura 31.10 Função `tpi_connect`: estabelece uma conexão com o servidor (*continuação*).

Preenchimento da estrutura de solicitação e envio para o provedor

18–26 A TPI define uma estrutura `T_conn_req` que contém o endereço de protocolo e as opções da conexão.

```
struct T_conn_req {
    t_scalar_t     PRIM_type;      /* T_CONN_REQ */
    t_scalar_t     DEST_length;    /* comprimento do endereço de destino */
    t_scalar_t     DEST_offset;    /* deslocamento do endereço de destino */
    t_scalar_t     OPT_length;     /* comprimento das opções */
    t_scalar_t     OPT_offset;     /* deslocamento das opções */
          /* seguido pelo endereço de protocolo e pelas opções de conexão */
};
```

Como em nossa função `tpi_bind`, definimos nossa própria estrutura chamada `conn_req`, que inclui uma estrutura `T_conn_req`, junto com espaço para o endereço de protocolo. Preenchemos nossa estrutura `conn_req`, configurando como 0 os dois membros que tratam com as opções. Chamamos `putmsg` somente com as informações de controle e um flag igual a 0, para enviar uma mensagem `M_PROTO` fluxo abaixo.

Leitura da resposta

27-45 Chamamos `getmsg`, esperando receber uma mensagem `T_OK_ACK`, caso o estabelecimento da conexão tenha se iniciado, ou uma mensagem `T_ERROR_ACK` (que mostramos anteriormente).

```
struct T_ok_ack {
   t_scalar_t     PRIM_type;          /* T_OK_ACK */
   t_scalar_t     CORRECT_prim;       /* primitiva correta */
};
```

Em caso de erro, terminamos. Como não conhecemos o tipo de mensagem que receberemos, uma união (`union`) chamada `T_primitives` é definida como a união de todas as solicitações e respostas possíveis, e alocamos uma dessas que utilizamos como buffer de entrada para as informações de controle, quando chamamos `getmsg`.

Espera pelo estabelecimento da conexão

46-65 A mensagem de sucesso `T_OK_ACK` que acabou de ser recebida nos informa apenas que o estabelecimento de conexão foi iniciado. Agora, devemos esperar por uma mensagem `T_CONN_CON` para nos informar que a outra extremidade confirmou a solicitação de conexão.

```
struct T_conn_con {
   t_scalar_t     PRIM_type;   /* T_CONN_CON */
   t_scalar_t     RES_length;  /* comprimento do endereço de resposta */
   t_scalar_t     RES_offset;  /* deslocamento do endereço de resposta */
   t_scalar_t     OPT_length;  /* comprimento da opção */
   t_scalar_t     OPT_offset;  /* deslocamento da opção */
      /* seguido pelo endereço de protocolo do peer e pelas opções */
};
```

Chamamos `getmsg` novamente, mas a mensagem esperada é enviada como uma mensagem `M_PROTO` e não como `M_PCPROTO`; portanto, configuramos os flags como 0. Se recebemos a mensagem `T_CONN_CON`, a conexão é estabelecida e retornamos; mas, se a conexão não foi estabelecida (o processo peer não estava executando, um tempo-limite excedeu, ou o que for), uma mensagem `T_DISCON_IND` é enviada fluxo acima.

```
struct T_discon_ind {
   t_scalar_t     PRIM_type;     /* T_DISCON_IND */
   t_scalar_t     DISCON_reason; /* razão da desconexão */
   t_scalar_t     SEQ_number;    /* número de seqüência */
};
```

Podemos ver os diferentes erros que são retornados pelo provedor. Primeiro, especificamos o endereço IP de um host que não está executando o servidor de data/hora.

```
solaris % tpi_daytime 192.168.1.10
tpi_connect2: T_DISCON_IND from conn (146)
```

O erro 146 corresponde a `ECONNREFUSED`. Em seguida, especificamos um endereço IP que não está conectado à Internet.

```
solaris % tpi_daytime 192.3.4.5
tpi_connect2: T_DISCON_IND from conn (145)
```

O erro desta vez é `ETIMEDOUT`. Porém, se executarmos nosso programa novamente, especificando o mesmo endereço IP, obteremos um erro diferente.

```
solaris % tpi_daytime 192.3.4.5
tpi_connect2: T_DISCON_IND from conn (148)
```

O erro desta vez é EHOSTUNREACH. A diferença nos últimos dois resultados é que, na primeira vez, nenhum erro de "host inacessível" ICMP foi retornado, enquanto na segunda vez, esse erro foi retornado.

A próxima função é tpi_read, mostrada na Figura 31.11. Ela lê dados de um fluxo.

streams/tpi_read.c

```
 1 #include   "tpi_daytime.h"

 2 ssize_t
 3 tpi_read(int fd, void *buf, size_t len)
 4 {
 5     struct strbuf ctlbuf;
 6     struct strbuf datbuf;
 7     union T_primitives rcvbuf;
 8     int    flags;

 9     ctlbuf.maxlen = sizeof(union T_primitives);
10     ctlbuf.buf = (char *) &rcvbuf;

11     datbuf.maxlen = len;
12     datbuf.buf = buf;
13     datbuf.len = 0;

14     flags = 0;
15     Getmsg(fd, &ctlbuf, &datbuf, &flags);

16     if (ctlbuf.len >= (int) sizeof(long)) {
17         if (rcvbuf.type == T_DATA_IND)
18             return (datbuf.len);
19         else if (rcvbuf.type == T_ORDREL_IND)
20             return (0);
21         else
22             err_quit("tpi_read: unexpected type %d", rcvbuf.type);
23     } else if (ctlbuf.len == -1)
24         return (datbuf.len);
25     else
26         err_quit("tpi_read: bad length from getmsg");
27 }
```

streams/tpi_read.c

Figura 31.11 Função tpi_read: lê dados de um fluxo.

Leitura do controle e dos dados; processamento da resposta

9–26 Desta vez, chamamos getmsg para ler informações de controle e dados. A estrutura strbuf para os dados aponta para o buffer do chamador. Quatro cenários diferentes podem ocorrer no fluxo:

- Os dados podem chegar como uma mensagem M_DATA, o que é indicado pelo comprimento de controle retornado sendo configurado como –1. Os dados foram copiados no buffer do chamador por getmsg e retornamos apenas o comprimento desses dados como o valor de retorno da função.
- Os dados podem chegar como uma mensagem T_DATA_IND, neste caso as informações de controle serão uma estrutura T_data_ind.

```
struct T_data_ind {
  t_scalar_t   PRIM_type;   /* T_DATA_IND */
```

```
  t_scalar_t   MORE_flag;   /* mais dados */
};
```

Se essa mensagem é retornada, ignoramos o membro `MORE_flag` (ele nunca será configurado para um protocolo de fluxo, como o TCP) e retornamos apenas o comprimento dos dados que foram copiados no buffer do chamador por `getmsg`.

- Uma mensagem `T_ORDREL_IND` é retornada se todos os dados foram consumidos e o próximo item é um FIN.

```
struct T_ordrel_ind {
  t_scalar_t   PRIM_type;   /* T_ORDREL_IND */
};
```

Essa é a liberação ordenada. Apenas retornamos 0, indicando ao chamador que EOF foi encontrado na conexão.

- Uma mensagem `T_DISCON_IND` é retornada se uma desconexão foi recebida.

Nossa última função é `tpi_close`, mostrada na Figura 31.12.

streams/tpi_close.c

```
 1 #include   "tpi_daytime.h"
 2 void
 3 tpi_close(int fd)
 4 {
 5     struct T_ordrel_req ordrel_req;
 6     struct strbuf ctlbuf;
 7     ordrel_req.PRIM_type = T_ORDREL_REQ;
 8     ctlbuf.len = sizeof(struct T_ordrel_req);
 9     ctlbuf.buf = (char *) &ordrel_req;
10     Putmsg(fd, &ctlbuf, NULL, 0);
11     Close(fd);
12 }
```

streams/tpi_close.c

Figura 31.12 Função `tpi_close`: envia uma liberação ordenada para o peer.

Envio da liberação ordenada para o peer

7–10 Construímos uma estrutura `T_ordrel_req`

```
struct T_ordrel_req {
   long PRIM_type;   /* T_ORDREL_REQ */
};
```

e a enviamos como uma mensagem `M_PROTO` utilizando `putmsg`.

Esse exemplo nos deu uma idéia do TPI. A aplicação envia mensagens fluxo abaixo para o provedor (solicitações) e este envia mensagens fluxo acima (respostas). Algumas trocas seguem um cenário simples de solicitação-resposta (vinculando um endereço local), enquanto outras podem demorar um pouco (estabelecendo uma conexão), permitindo-nos fazer algo enquanto esperamos pela resposta. Nossa escolha de escrever um cliente TCP utilizando TPI foi feita por simplicidade; escrever um servidor TCP e tratar de conexões é muito mais difícil.

> Podemos comparar o número de chamadas de sistema exigidas para as operações de rede que vimos neste capítulo, ao utilizar TPI *versus* um kernel que implementa soquetes dentro dele. Vincular um endereço local exige duas chamadas de sistema com TPI, mas somente uma com soquetes de kernel (TCPv2, página 454). Estabelecer uma conexão em

um descritor com bloqueio exige três chamadas de sistema com TPI, mas somente uma com soquetes de kernel (TCPv2, página 466).

31.7 Resumo

Às vezes, os soquetes são implementados utilizando STREAMS. Quatro novas funções são fornecidas para acessar o subsistema STREAMS: `getmsg`, `getpmsg`, `putmsg` e `putpmsg`, mais a função `ioctl` existente, que também é bastante utilizada pelo subsistema STREAMS.

TPI é a interface STREAMS do SVR4, das camadas superiores para a camada de transporte. Ela é utilizada tanto por soquetes como pela XTI, conforme mostrado na Figura 31.3. Desenvolvemos uma versão de nosso cliente de data/hora usando a TPI diretamente, como um exemplo para mostrar a interface baseada em mensagens que a TPI utiliza.

Exercício

31.1 Na Figura 31.12, chamamos `putmsg` para enviar a solicitação de liberação ordenada fluxo abaixo e, então, fechamos (`close`) o fluxo imediatamente. O que acontece se nossa solicitação de liberação ordenada é perdida pelo subsistema STREAMS quando o fluxo é fechado?

Apêndice **A**

IPv4, IPv6, ICMPv4 e ICMPv6

A.1 Visão geral

Este apêndice é uma visão geral do IPv4, IPv6, ICMPv4 e ICMPv6. Este material fornece uma base adicional que pode ser útil no entendimento da discussão sobre TCP e UDP do Capítulo 2. Alguns recursos de IP e ICMP também foram utilizados em capítulos posteriores: as opções de IP (Capítulo 27) e os programas `ping` e `traceroute` (Capítulo 28), por exemplo.

A.2 Cabeçalho IPv4

A camada de IP fornece um serviço de entrega de datagrama de melhor esforço sem conexão (RFC 791 [Postel, 1981a]). O IP faz o máximo para entregar um datagrama de IP no destino especificado, mas não há nenhuma garantia de que ele chegue, de que chegará em ordem em relação aos outros pacotes ou de que chegará somente uma vez. Quaisquer confiabilidade, ordenação e supressão duplicada desejadas devem ser adicionadas pelas camadas superiores. No caso de uma aplicação TCP ou SCTP, isso é realizado pela camada de transporte. No caso de uma aplicação UDP, isso deve ser feito pela aplicação, pois o UDP não é confiável; mostramos um exemplo disso na Seção 22.5.

Uma das funções mais importantes da camada de IP é o *roteamento*. Cada datagrama de IP contém um endereço de origem e um de destino. A Figura A.1 mostra o formato de um cabeçalho Ipv4.

- O campo de *versão* de 4 bits é 4. Essa tem sido a versão de IP em utilização desde o início da década de 1980.
- O campo de *comprimento do cabeçalho* é o comprimento do cabeçalho IP inteiro, incluindo todas as opções, em palavras de 32 bits inteiras. O valor máximo desse campo de 4 bits é 15 (`0xf`), dando um comprimento máximo de cabeçalho IP de 60 bytes. Portanto, com a parte fixa do cabeçalho ocupando 20 bytes, isso permite até 40 bytes de opções.
- O campo *Differentiated Services Code Point* (DSCP) de 6 bits (RFC 2474 [Nichols *et al*., 1998]) e *Explicit Congestion Notification* (ECN) de 2 bits (RFC 3168 [Ramakrishnan,

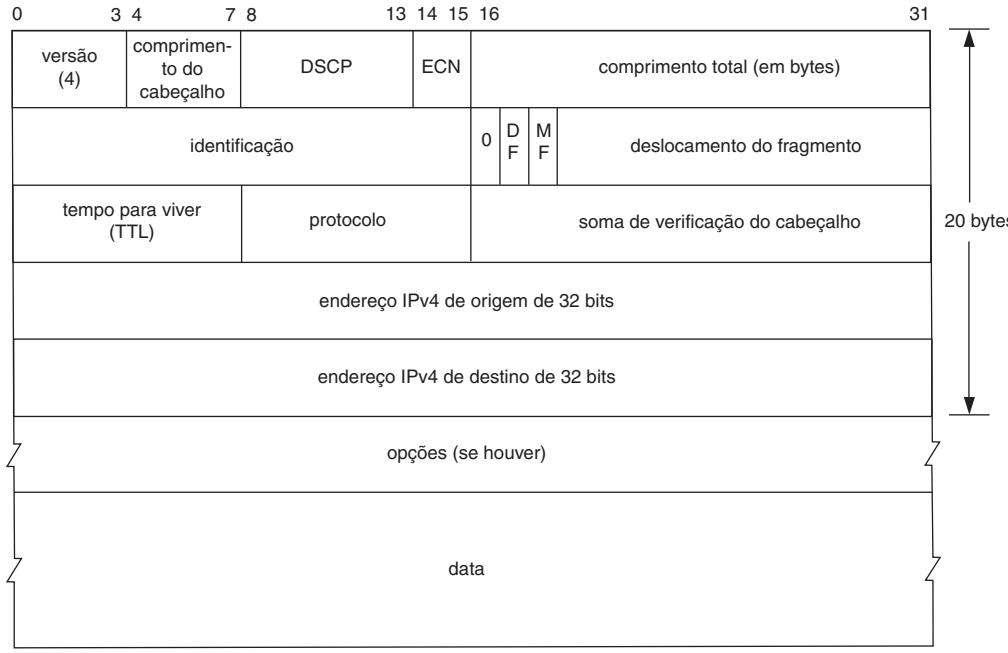


Figura A.1 Formato do cabeçalho IPv4.

Floyd e Black, 2001]) substituem o histórico campo de *tipo de serviço* (TOS) de 8 bits, que foi descrito na RFC 1349 (Almquist, 1992). Podemos configurar todos os 8 bits desse campo com a opção de soquete `IP_TOS` (Seção 7.6), embora o kernel possa sobrescrever qualquer valor que configuremos, para impor a política Diffserv ou implementar ECN.

- O campo de *comprimento total* de 16 bits é o comprimento total, em bytes, do datagrama de IP, incluindo o cabeçalho IPv4. O volume de dados no datagrama é esse campo menos 4 vezes o comprimento do cabeçalho (lembre-se de que o comprimento do cabeçalho está em unidades de palavras de 32 bits inteiras ou 4 bytes). Esse campo é exigido porque alguns enlaces de dados preenchem o quadro com algum comprimento mínimo (por exemplo, Ethernet) e é possível que o tamanho de um datagrama de IP válido seja menor que o enlace de dados mínimo.
- O campo de *identificação* de 16 bits é configurado com um valor diferente para cada datagrama de IP e permite fragmentação e montagem (Seção 2.11). O valor deve ser único para a origem, destino e protocolo do pacote, para o período de tempo durante o qual o datagrama poderia estar em trânsito. Se não há nenhuma chance de que o pacote seja fragmentado, por exemplo, o bit *DF* está configurado, não há necessidade de configurar esse campo.
- O bit *DF* (*don't fragment*), o bit *MF* (*more fragments*) e o campo de *deslocamento do fragmento* de 13 bits também são utilizados com fragmentação e remontagem. O bit *DF* também é utilizado com descoberta MTU de caminho (Seção 2.11).
- O campo de *tempo para viver* (*time to live* – TTL) de 8 bits é configurado pelo emissor e, então, decrementado por 1 sempre que um roteador encaminha o datagrama. O datagrama é descartado por qualquer roteador que decremente o valor para 0. Isso limita o tempo para viver de qualquer datagrama de IP para 255 hops. Um padrão comum para esse campo é 64, mas podemos consultar e alterar esse padrão com as opções de soquete `IP_TTL` e `IP_MULTICAST_TTL` (Seção 7.6).

- O campo de *protocolo* de 8 bits especifica o próximo protocolo de camada contido no datagrama de IP. Valores típicos são 1 (ICMPv4), 2 (IGMPv4), 6 (TCP) e 17 (UDP). Esses valores são especificados no registro "Números de Protocolo" do IANA [IANA].
- A *soma de verificação de cabeçalho* de 16 bits é calculada somente sobre o cabeçalho IP (incluindo todas as opções). O algoritmo é o de soma de verificação-padrão da Internet, uma simples adição de 16 bits em complemento de um, que mostramos na Figura 28.15.
- O *endereço IPv4 de origem* e o *endereço IPv4 de destino* são ambos campos de 32 bits.
- Descrevemos o campo de *opções* na Seção 27.2 e mostramos um exemplo da opção de rota de origem IPv4 na Seção 27.3.

A.3 Cabeçalho IPv6

A Figura A.2 mostra o formato de um cabeçalho IPv6 (RFC 2460 [Deering e Hinden, 1998]).

- O campo de *versão* de 4 bits é 6. Como esse campo ocupa os primeiros 4 bits do primeiro byte do cabeçalho (exatamente como a versão IPv4, Figura A.1), ele permite que uma pilha receptora de IP diferencie entre as duas versões. Essa diferenciação já é feita pela maioria das camadas de enlace, utilizando um encapsulamento diferente para IPv4 e IPv6.

  Durante o desenvolvimento do IPv6, no início da década de 1990, antes que o número de versão 6 fosse atribuído, o protocolo era chamado *IPng*, de "IP next generation" (IP de próxima geração). Referências ao IPng ainda podem ser encontradas.
- O campo de DSCP de 6 bits (RFC 2474 [Nichols *et al*., 1998]) e o campo de ECN de 2 bits (RFC 3168 [Ramakrishnan, Floyd e Black 2001]) substituem o histórico campo de

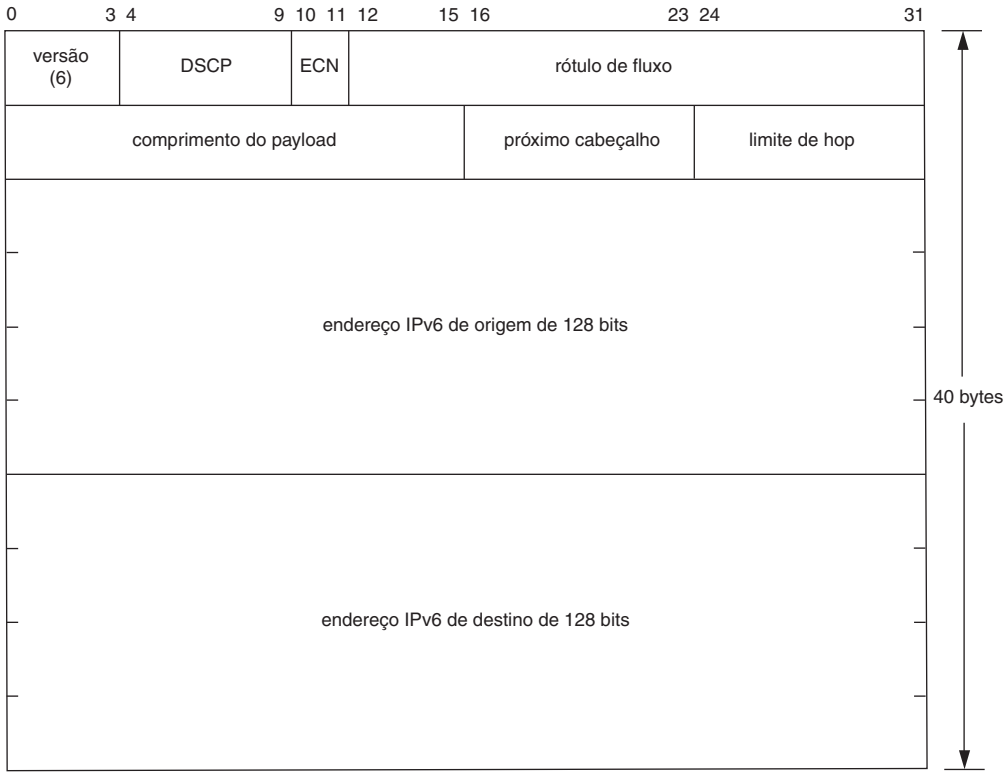


Figura A.2 Formato do cabeçalho IPv6.

classe de tráfego de 8 bits, que foi descrito na RFC 2460. Podemos configurar todos os 8 bits desse campo com a opção de soquete `IPV6_TCLASS` (Seção 22.8), embora o kernel possa sobrescrever qualquer valor que configurarmos para impor a diretiva Diffserv ou para implementar ECN.

- O campo de *rótulo de fluxo* de 20 bits pode ser escolhido pela aplicação ou pelo kernel para determinado soquete. Um *fluxo (flow)* é uma seqüência de pacotes de uma origem particular para um destino particular, para o qual a origem deseja tratamento especial dos roteadores intervenientes. Para determinado fluxo, uma vez que o rótulo de fluxo é escolhido pela origem, ele não muda. Um rótulo de fluxo igual a 0 (default) identifica pacotes que não pertencem a um fluxo. O rótulo de fluxo não muda enquanto fluir pela rede. Rajahalme *et al*. (2003) descrevem mais completamente o uso do rótulo de fluxo.

  A interface do rótulo de fluxo ainda está para ser definida completamente. O membro `sin6_flowinfo` da estrutura de endereço de soquete `sockaddr_in6` (Figura 3.4) está reservado para utilização futura. Alguns sistemas copiam os 28 bits inferiores do membro `sin6_flowinfo` diretamente no cabeçalho de pacote IPv6, sobrescrevendo os campos DSCP e ECN.

- O campo de *comprimento do payload* de 16 bits é o comprimento, em bytes, de tudo que vem após o cabeçalho IPv6 de 40 bytes. Observe que, ao contrário do IPv4, o campo de *comprimento do payload* não inclui o cabeçalho IPv6. Um valor igual a 0 significa que o comprimento exige mais de 16 bits para descrever e está contido em uma opção de payload jumbo (Figura 27.9). Isso é chamado de *jumbograma*.

- O campo *próximo cabeçalho* de 8 bits é semelhante ao campo de protocolo IPv4. De fato, quando o protocolo da camada superior fica basicamente inalterado do IPv4 para o IPv6, os mesmos valores são utilizados, como 6 para TCP e 17 para UDP. Houve tantas alterações do ICMPv4 para o ICMPv6 que este último recebeu o novo valor 58.

  Um datagrama IPv6 pode ter muitos cabeçalhos após o cabeçalho IPv6 de 40 bytes. Essa é a razão pela qual o campo é chamado de "próximo cabeçalho" e não de "protocolo".

- O campo *limite de hop* de 8 bits é semelhante ao campo TTL do IPv4. O limite de hop é decrementado por 1 sempre que um roteador encaminha o datagrama e este é descartado por qualquer roteador que decremente o valor para 0. O valor default para esse campo pode ser configurado e buscado com as opções de soquete `IPV6_UNICAST_HOPS` e `IPV6_MULTICAST_HOPS` (Seções 7.8 e 21.6). A opção de soquete `IPV6_HOPLIMIT` também nos permite configurar esse campo e a opção de soquete `IPV6_RECVHOPLIMIT` nos permite obter seu valor a partir de um datagrama recebido.

  > As especificações anteriores do IPv4 faziam os roteadores decrementarem o TTL por 1 ou pelo número de segundos durante os quais mantinham o datagrama, o que fosse maior. Daí o nome "tempo para viver". Na realidade, entretanto, o campo sempre era decrementado por 1. O IPv6 exige que seu campo de limite de hop seja sempre decrementado por 1, daí a mudança do nome em relação ao IPv4.

- O *endereço IPv6 de origem* e o *endereço IPv6 de destino* são ambos campos de 128 bits.

A alteração mais significativa do IPv4 para o IPv6 é, naturalmente, os campos de endereço IPv6 maiores. Outra alteração é a simplificação do cabeçalho IPv6, como segue, para facilitar o processamento mais rápido, enquanto um datagrama passa pela rede:

- Não há nenhum campo de comprimento do cabeçalho IPv6, pois o comprimento do cabeçalho IPv6 é fixo em 40 bytes. Cabeçalhos opcionais podem vir após o cabeçalho IPv6 de 40 bytes fixo, mas cada um deles tem seu próprio campo de comprimento.

- Os dois endereços IPv6 terminam alinhados em um limite de 64 bits, enquanto o cabeçalho em si é alinhado em 64 bits. Isso pode acelerar o processamento em arquiteturas de

64 bits. Os endereços IPv4 são somente alinhados em 32 bits em um cabeçalho IPv4 alinhado em 64 bits.

- Não existem campos de fragmentação no cabeçalho IPv6 porque há um cabeçalho de fragmentação separado para esse propósito. Essa decisão de projeto foi tomada porque a fragmentação é a exceção e as exceções não devem tornar lento o processamento normal.
- O cabeçalho IPv6 não inclui sua própria soma de verificação. Isso é porque todas as camadas superiores – TCP, UDP e ICMPv6 – têm sua própria soma de verificação que inclui o cabeçalho da camada superior, os dados da camada superior e os seguintes campos de cabeçalho IPv6: endereço de origem IPv6, endereço de destino IPv6, comprimento do payload e próximo cabeçalho (RFC 2460 [Deering e Hinden, 1998]). Omitindo a soma de verificação do cabeçalho, os roteadores que encaminham o datagrama não precisam recalcular uma soma de verificação de cabeçalho após modificarem o limite de hop. Novamente, a velocidade do encaminhamento dos roteadores é o ponto-chave.

No caso de este ser o seu primeiro contato com o IPv6, observamos também as seguintes diferenças importantes do IPv4 para o IPv6:

- Não há nenhum broadcasting com o IPv6 (Capítulo 20). O multicast (Capítulo 21), que é opcional no IPv4, é obrigatório no IPv6. O caso do envio para todos os sistemas em uma sub-rede é tratado com o grupo de multicast de todos os nós.
- Os roteadores IPv6 não fragmentam pacotes que encaminham. Se a fragmentação é exigida, o roteador omite o pacote e envia um erro ICMPv6 (Seção A.6). A fragmentação é realizada somente pelo host de origem, no IPv6.
- O IPv6 exige suporte para descoberta MTU de caminho (Seção 2.11). Tecnicamente, esse suporte é opcional e poderia ser omitido das implementações mínimas, como as rotinas de inicialização, mas, se um nó não implementa esse recurso, não deve enviar datagramas maiores que o MTU de enlace mínimo do IPv6 (1280 bytes). A Seção 22.9 descreve as opções de soquete para controlar o comportamento de descoberta MTU de caminho.
- O IPv6 exige suporte para autenticação e opções de segurança. Essas opções aparecem após o cabeçalho fixo.

A.4 Endereços IPv4

Os endereços IPv4 têm 32 bits de comprimento e normalmente são escritos como quatro números decimais separados por pontos (“.”). Isso é chamado de *notação decimal com pontos* e cada número decimal representa um dos 4 bytes do endereço de 32 bits. O primeiro dos quatro números decimais identifica o tipo de endereço, como mostrado na Figura A.3. Embora, historicamente, os endereços IPv4 fossem divididos em cinco classes, como mostrado na Figura A.3, as três classes utilizadas para endereços unicast são funcionalmente equivalentes; portanto, as mostramos como um intervalo.

Quando falamos sobre uma rede ou endereço de sub-rede IPv4, falamos sobre um endereço de rede de 32 bits e uma máscara de 32 bits correspondente. Os bits 1 na máscara cobrem o endereço de rede e os bits 0, o host. Como os bits 1 da máscara normalmente são contíguos a partir do bit mais à esquerda e os bits 0 são sempre contíguos a partir do bit mais à direita, essa máscara de endereço também pode ser especificada como um *comprimento de prefixo*,

Uso	Classe	Intervalo
Unicast	A, B, C	**0**.0.0.0 a **223**.255.255.255
Multicast	D	**224**.0.0.0 a **239**.255.255.255
Experimental	E	**240**.0.0.0 a **255**.255.255.255

Figura A.3 Intervalos para as cinco diferentes classes de endereços IPv4.

que denota o número de bits 1 contíguos, a partir da esquerda. Por exemplo, uma máscara 255.255.255.0 corresponde a um comprimento de prefixo 24. Esses são conhecidos como endereços *sem classe*, assim chamados porque a máscara é especificada explicitamente, em vez de ser sugerida pela classe do endereço. Os endereços de rede IPv4 normalmente são escritos como um número decimal com pontos, seguido por uma barra e pelo comprimento do prefixo. A Figura 1.16 mostrou exemplos disso.

> As máscaras de sub-rede descontínuas nunca foram eliminadas por qualquer RFC, mas são confusas e não podem ser representadas em notação de prefixo. O BGP4, o protocolo de roteamento entre domínios de Internet, não pode representar máscaras descontínuas. O IPv6 também exige que todas as máscaras de endereço sejam contíguas, iniciando no bit mais à esquerda.

A utilização de endereços sem classe exige roteamento sem classe, e isso é normalmente chamado de *roteamento entre domínios sem classe* (CIDR) (RFC 1519 [Fuller *et al*., 1993]). O uso de CIDR diminui o tamanho das tabelas de roteamento de backbone da Internet e reduz a taxa de esgotamento de endereços IPv4. Todas as rotas no CIDR devem ser acompanhadas por uma máscara ou por um comprimento de prefixo. A classe do endereço não implica mais a máscara. A seção 10.8 do TCPv1 fala mais sobre CIDR.

Endereços de sub-rede

Os endereços IPv4 são freqüentemente transformados em *sub-rede* (RFC 950 [Mogul e Postel, 1985]). Isso adiciona outro nível na hierarquia de endereços:

- ID de rede (atribuída ao site)
- ID de sub-rede (escolhida pelo site)
- ID de host (escolhida pelo site)

O limite entre os IDs de rede e de sub-rede é fixado pelo comprimento do prefixo do endereço de rede atribuído. Esse comprimento de prefixo normalmente é atribuído pelo provedor de serviços de Internet (ISP) da organização. Mas o limite entre os IDs de sub-rede e de host é escolhido pelo site. Todos os hosts em determinada sub-rede compartilham uma *máscara de sub-rede* comum que especifica o limite entre os IDs de sub-rede e de host. Os bits 1 na máscara de sub-rede cobrem os IDs de rede e de sub-rede e os bits 0 cobrem o ID de host.

Como exemplo, considere um site que receba o prefixo 192.168.42.0/24 de seu ISP (Internet Service Provider). Se ele optar por utilizar um ID de sub-rede de 3 bits, 5 bits serão deixados para o ID de host, como mostrado na Figura A.4.

Essa divisão resulta nas sub-redes mostradas na Figura A.5.

Isso nos dá de 6 a 8 sub-redes (IDs de sub-rede 1-6 ou 0-7), cada uma suportando 30 sistemas (IDs de host 1-30). A RFC 950 recomenda não utilizar as duas sub-redes com um ID de

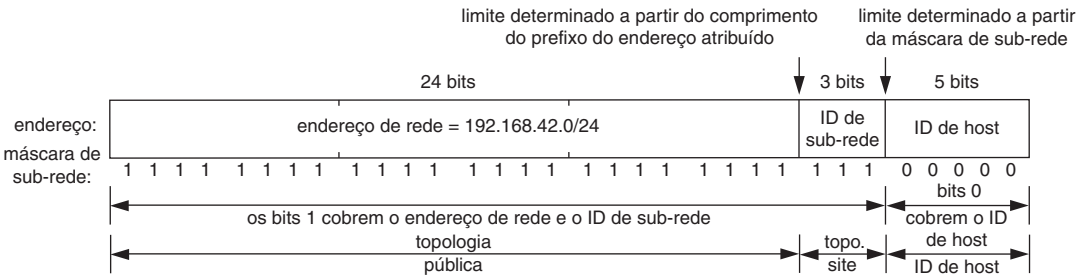


Figura A.4 Endereço de rede de 24 bits com ID de sub-rede de 3 bits e ID de host de 5 bits.

Sub-rede	Prefixo
0	192.168.42.0/27 †
1	192.168.42.32/27
2	192.168.42.64/27
3	192.168.42.96/27
4	192.168.42.128/27
5	192.168.42.160/27
6	192.168.42.192/27
7	192.168.42.224/27 †

Figura A.5 Lista de sub-redes para ID de sub-rede de 3 bits e ID de host de 5 bits.

sub-rede com todos os bits 0 ou com todos os bits 1 (aqueles marcados com uma cruz na Figura A.5). A maioria dos sistemas hoje suporta essas duas formas de IDs de sub-rede. O ID de host mais alto (31, nesse caso) é reservado para o endereço de broadcast. O ID de host 0 é reservado para identificar a rede e evitar problemas com sistemas antigos que usavam ID de host 0 como endereço de broadcast. Entretanto, em redes controladas sem tais sistemas, é possível utilizar ID de host 0. Em geral, os programas de rede não precisam se preocupar com IDs de sub-rede ou de host específicos e devem tratar de endereços IP como valores opacos.

Endereços de loopback

Por convenção, o endereço 127.0.0.1 é atribuído à interface de loopback. Qualquer coisa enviada para esse endereço IP faz um loop e torna-se entrada de IP sem jamais deixar a máquina. Freqüentemente, utilizamos esse endereço ao testar um cliente e um servidor no mesmo host. Esse endereço é normalmente conhecido pelo nome `INADDR_LOOPBACK`.

> Qualquer endereço na rede 127/8 pode ser atribuído à interface de loopback, mas 127.0.0.1 é comum e, freqüentemente, configurado de modo automático pela pilha de IP.

Endereço não-especificado

O endereço consistindo em 32 bits zero é o endereço IPv4 não-especificado. Em um pacote IPv4, ele só pode aparecer como o endereço de origem em pacotes enviados por um nó em inicialização, antes que esse nó conheça seu endereço IP. Na API de soquetes, esse endereço é chamado de endereço curinga e normalmente é conhecido pelo nome `INADDR_ANY`. Além disso, especificá-lo na API de soquetes, por exemplo, para vincular (`bind`) a um soquete TCP ouvinte, indica que o soquete aceitará conexões de cliente destinadas a qualquer um dos endereços IPv4 do nó.

Endereços privados

A RFC 1918 (Rekhter *et al*., 1996) separa três intervalos de endereço para "Internets privadas"; isto é, redes que não se conectam à Internet pública sem NAT ou proxies intermediários. Esses intervalos de endereço são mostrados na Figura A.6:

Número de endereços	Prefixo	Intervalo
16.777.216	10/8	10.0.0.0 a 10.255.255.255
1.048.576	172.16/12	172.16.0.0 a 172.31.255.255
65.536	192.168/16	192.168.0.0 a 192.168.255.255

Figura A.6 Intervalos para endereços IPv4 privados.

Esses endereços nunca devem aparecer na Internet; eles são reservados para utilização em redes privadas. Vários sites pequenos utilizam esses endereços privados e NAT para um único endereço IP público, visível para a Internet.

Múltiplos homes e alias de endereço

Tradicionalmente, a definição de um host *multihomed* tem sido a de um host com várias interfaces: duas Ethernets, por exemplo, ou uma Ethernet e um enlace ponto a ponto. Cada interface geralmente deve ter um endereço IPv4 único. Ao se contar interfaces para determinar se um host é multihomed, a interface de loopback não é levada em consideração.

Um roteador, por definição, é multihomed, pois encaminha pacotes que chegam a uma interface para outra interface. Mas um host multihomed não é um roteador, a menos que encaminhe pacotes. De fato, não se deve supor que um host multihomed seja um roteador apenas porque tem múltiplas interfaces; ele não deve agir como roteador, a menos que tenha sido configurado para isso (em geral, com o administrador ativando uma opção de configuração).

O conceito de "múltiplos homes", entretanto, é mais geral e cobre dois cenários diferentes (Seção 3.3.4 da RFC 1122 [Braden, 1989]):

- Um host com múltiplas interfaces é multihomed e, em geral, cada interface deve ter seu próprio endereço IP. (As interfaces "não-numeradas" não precisam ter endereços IP, mas encontramos isso principalmente em roteadores.) Essa é a definição tradicional.
- Os hosts mais recentes têm a capacidade de atribuir vários endereços IP a determinada interface física. Cada endereço IP adicional, após o primeiro (primário), é chamado de *alias* ou *interface lógica*. Freqüentemente, os endereços IP de alias compartilham o mesmo endereço de sub-rede do endereço primário, mas têm IDs de host diferentes. Mas também é possível que os alias tenham um endereço de rede ou endereços de sub-rede completamente diferentes do primário. Mostramos um exemplo de endereços de alias na Seção 17.6. Daí, a definição de host multihomed ser um com múltiplas interfaces visíveis para a camada de IP, independentemente de essas interfaces serem físicas ou lógicas.

  É comum dar a um servidor de alta utilização várias conexões com o mesmo comutador Ethernet e agregar essas conexões para que apareçam como uma única interface de largura de banda mais alta. Embora tal sistema tenha múltiplas interfaces físicas, não é considerado como multihomed, pois somente uma interface lógica é visível para o IP.

  O conceito de "múltiplos homes" também é utilizado em outro contexto. Uma rede que tenha várias conexões com a Internet também é chamada de multihomed. Por exemplo, alguns sites têm duas conexões com a Internet em vez de uma, fornecendo um recurso de backup. O protocolo de transporte SCTP pode tirar proveito dessas múltiplas conexões, comunicando que o site é multihomed para seu peer.

A.5 Endereços IPv6

Os endereços IPv6 têm 128 bits de comprimento e normalmente são escritos como oito números hexadecimais de 16 bits. Os bits de ordem superior do endereço de 128 bits indicam o tipo de endereço (RFC 3513 [Hinden e Deering, 2003]). A Figura A.7 mostra os diferentes valores dos bits de ordem superior e que tipo de endereço esses bits sugerem.

Esses bits de ordem superior são chamados de *prefixo de formato*. Por exemplo, se os três bits de ordem superior são `001`, o endereço é chamado de *endereço de unicast global*. Se os oito bits de ordem superior são `11111111` (`0xff`), trata-se de um endereço de multicast.

Endereços unicast globais

A arquitetura de endereçamento IPv6 evoluiu com base nas lições aprendidas na distribuição e do IPv4. A definição original de endereços unicast globais agregáveis, que na Figura A.7 co-

Alocação	Tamanho do ID da interface	Prefixo do formato	Referência
Não-especificado	n/d	`0000 0000 ... 0000 0000` (*128 bits*)	RFC 3513
Loopback	n/d	`0000 0000 ... 0000 0001` (*128 bits*)	RFC 3513
Endereço de unicast global	qualquer um	`000`	RFC 3513
Endereço baseado no NSAP global	qualquer um	`0000001`	RFC 1888
Endereço de unicast global agregável	64 bits	`001`	RFC 3587
Endereço de unicast global	64 bits	(qualquer um não mencionado de outra forma)	RFC 3513
Endereço de unicast de enlace local	64 bits	`1111 1110 10`	RFC 3513
Endereço de unicast de site local	64 bits	`1111 1110 11`	RFC 3513
Endereço de multicast	n/d	`1111 1111`	RFC 3513

Figura A.7 Significado dos bits de ordem superior de endereços IPv6.

meçam com o prefixo de três bits `001`, tinha uma estrutura fixa incorporada ao endereço. Essa estrutura foi removida pela RFC 3587 (Hinden, Deering e Nordmark, 2003) e, embora os endereços que iniciam com o prefixo `001` sejam os primeiros atribuídos, não há nenhuma diferença entre eles e qualquer outro endereço global. Esses endereços serão utilizados onde os endereços unicast IPv4 são usados hoje.

O formato dos endereços unicast baseados em agregação está definido na RFC 3513 (Hinden e Deering, 2003) e na RFC 3587 (Hinden, Deering e Nordmark, 2003), e contém os seguintes campos, começando no bit mais à esquerda e indo para a direita:

- Prefixo de roteamento global (*n* bits)
- ID de sub-rede (64-*n* bits)
- Identificador de interface (64 bits)

A Figura A.8 ilustra o formato de um endereço de unicast global.

O ID da interface deve ser construído no formato *EUI-64* modificado. Essa é uma variação do formato IEEE *EUI-64* (IEEE, 1997), que é um superconjunto dos endereços MAC IEEE 802 de 48 bits, que são atribuídos à maioria das placas de interface de rede local. Esse identificador deve ser atribuído automaticamente a uma interface, com base em seu endereço MAC de hardware, quando possível. Os detalhes para construir identificadores de interface modificados baseados no EUI-64 estão no Apêndice A da RFC 3513 (Hinden e Deering, 2003).

Como um EUI-64 modificado pode ser um identificador globalmente exclusivo para determinada interface e uma interface pode identificar um usuário, o formato EUI-64 modificado levanta certas preocupações com a privacidade. É possível monitorar as ações e os movimentos de determinado usuário, onde eles levam seu laptop, por exemplo, apenas a partir do valor de EUI-64 modificado em seu endereço IPv6. A RFC 3041 (Narten e Draves, 2001) descreve extensões de privacidade para gerar identificadores de interface que mudam várias vezes por dia, para evitar essa preocupação com a privacidade.

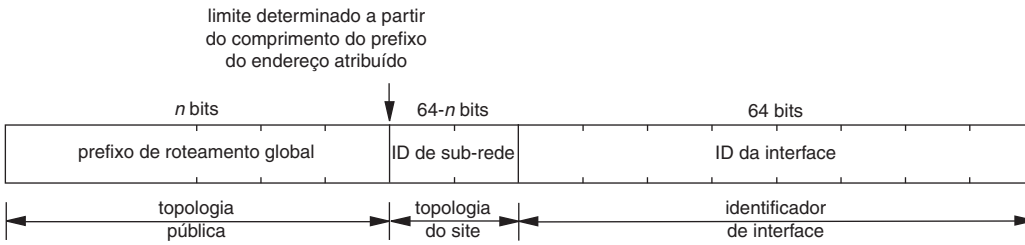


Figura A.8 Endereços unicast globais agregáveis do IPv6.

Endereços de teste 6bone

A 6bone é uma rede virtual utilizada para testes prévios dos protocolos IPv6 (Seção B.3). Embora os endereços unicast globais agregáveis estejam sendo atribuídos, os sites que não se qualificam para o espaço de endereçamento baseado nas regras utilizadas pelos registros regionais podem utilizar um formato especial desses endereços na 6bone (RFC 2471 [Hinden, Fink e Postel, 1998]), como mostrado na Figura A.9.

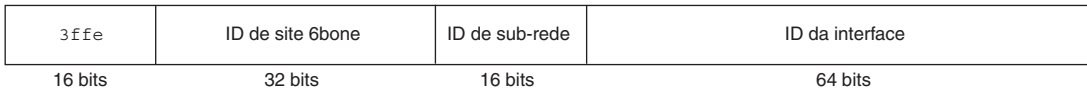


Figura A.9 Endereço de teste IPv6 para 6bone.

Esses endereços são considerados temporários e os nós que os utilizam terão de ser renumerados quando os endereços unicast globais agregáveis forem atribuídos.

Os dois bytes de ordem superior são `0x3ffe`. O *ID de site 6bone* é atribuído pelo responsável de atividade da 6bone. Essas atribuições se destinam a refletir como os endereços IPv6 seriam atribuídos em ambientes do mundo real. A atividade da 6bone está diminuindo (Fink e Hinden, 2003), agora que a distribuição de produção do IPv6 está bem adiantada (em 2002, foram feitas mais alocações de endereço de produção do que a 6bone alocou em oito anos). O *ID de sub-rede* e o *ID de interface* são utilizados como acima para sub-rede e identificação do nó.

Na Seção 11.2, mostramos o endereço IPv6 do host `freebsd`, da Figura 1.16, como `3ffe:b80:1f8d:1:a00:20ff:fea7:686b`. O ID de site da 6bone é `0x0b801f8d` e o ID de sub-rede é `0x1`. Os 64 bits de ordem inferior são o EUI-64 modificado, construído a partir do endereço MAC da placa Ethernet do host.

Endereços IPv6 mapeados de IPv4

Os endereços IPv6 mapeados de IPv4 permitem que as aplicações IPv6 em hosts que suportam tanto IPv4 como IPv6 se comuniquem com hosts que suportam somente IPv4 durante a transição da Internet para o IPv6. Esses endereços são criados automaticamente por resolvedores DNS (Figura 11.8), quando uma consulta é feita por uma aplicação IPv6 para os endereços IPv6 de um host que tem somente endereços IPv4.

Vimos, na Figura 12.4 que utilizar esse tipo de endereço com um soquete IPv6 faz com que um datagrama IPv4 seja enviado para o host IPv4. Esses endereços não são armazenados em qualquer arquivo de dados do DNS; eles são criados, quando necessário, por um resolvedor.

A Figura A.10 mostra o formato desses endereços. Os 32 bits de ordem inferior contêm um endereço IPv4.

Ao se escrever um endereço IPv6, uma string de zeros consecutiva pode ser abreviada com dois sinais de dois-pontos. Além disso, o endereço IPv4 incorporado é escrito utilizando-se notação decimal com pontos. Por exemplo, podemos abreviar o endereço IPv6 mapeado de IPv4 `0:0:0:0:0:FFFF:12.106.32.254` como `::FFFF:12.106.32.254`.

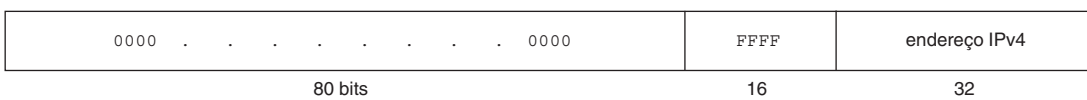


Figura A.10 Endereço IPv6 mapeado de IPv4.

Endereços IPv6 compatíveis com IPv4

Os endereços IPv6 compatíveis com IPv4 também foram planejados para serem utilizados durante a transição do IPv4 para o IPv6 (RFC 2893 [Gilligan e Nordmark, 2000]). O administrador de um host que suporta tanto IPv4 como IPv6, que não tenha um roteador de IPv6 vizinho, deve criar um registro de AAAA de DNS contendo um endereço IPv6 compatível com IPv4. Qualquer outro host IPv6 com um datagrama de IPv6 para enviar a um endereço IPv6 compatível com IPv4 *encapsulará* então o datagrama de IPv6 com um cabeçalho IPv4; isso é chamado de *túnel automático*. Entretanto, preocupações com a distribuição reduziram o uso desse recurso. Falaremos mais sobre tunelamento na Seção B.3 e mostraremos um exemplo desse tipo de datagrama de IPv6 encapsulado dentro de um cabeçalho IPv4, na Figura B.2.

A Figura A.11 mostra o formato de um endereço IPv6 compatível com IPv4.

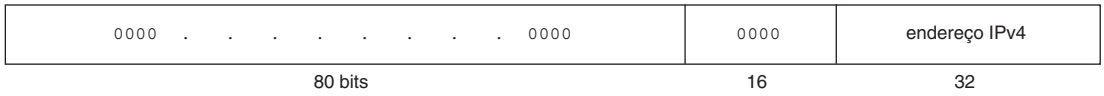


Figura A.11 Endereço IPv6 compatível com IPv4.

Um exemplo desse tipo de endereço é `::12.106.32.254`.

Os endereços IPv6 compatíveis com IPv4 também podem aparecer na origem ou no destino de pacotes IPv6 sem tunelamento, ao se utilizar o mecanismo de transição SIIT IPv4/IPv6 (RFC 2765 [Nordmark, 2000]).

Endereço de loopback

Um endereço IPv6 consistindo em 127 bits 0 e um único bit 1, escrito como `::1`, é o endereço IPv6 de loopback. Na API de soquetes, ele é referenciado como `in6addr_loopback` ou `IN6ADDR_LOOPBACK_INIT`.

Endereço não-especificado

Um endereço IPv6 consistindo em 128 bits 0, escrito como `0::0` ou somente `::`, é o endereço IPv6 não-especificado. Em um pacote IPv6, o endereço não-especificado pode aparecer somente como endereço de origem em pacotes enviados por um nó que é de inicialização, antes que o nó conheça seu endereço IPv6.

Na API de soquetes, esse endereço é chamado de endereço curinga. Especificá-lo, por exemplo, como `bind` para um soquete TCP ouvinte indica que o soquete aceitará conexões de cliente destinadas a qualquer um dos endereços do nó. Ele é referenciado como `in6addr_any` ou `IN6ADDR_ANY_INIT`.

Endereço de enlace local

Um endereço de enlace local é utilizado em um único enlace, quando é sabido que o datagrama não será encaminhado para fora da rede local. Exemplos de utilização são a configuração automática de endereço em tempo de inicialização e a descoberta de vizinho (semelhante ao ARP do IPv4). A Figura A.12 mostra o formato desses endereços.

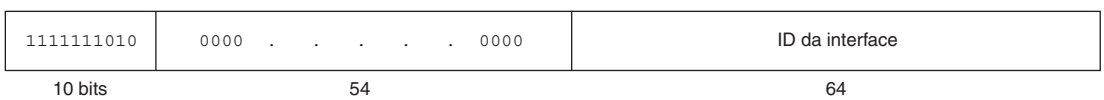


Figura A.12 Endereço de enlace local do IPv6.

Esses endereços sempre iniciam com `0xfe80`. Um roteador IPv6 não deve encaminhar um datagrama com um endereço de origem ou de destino de enlace local para outro enlace. Na Seção 11.2, mostramos o endereço de enlace local associado ao nome `aix-6ll`.

Endereço de site local

Quando este livro estava sendo escrito, o grupo de trabalho IETF IPv6 decidiu desaprovar os endereços de site locais em sua forma atual. O próximo substituto pode ou não finalmente utilizar o mesmo intervalo de endereços, conforme originalmente definido para endereços de site local (`fec0/10`). Os endereços de site local se destinavam a ser utilizados para endereçamento dentro de um site, sem a necessidade de um prefixo global. A Figura A.13 mostra o formato originalmente definido desses endereços.

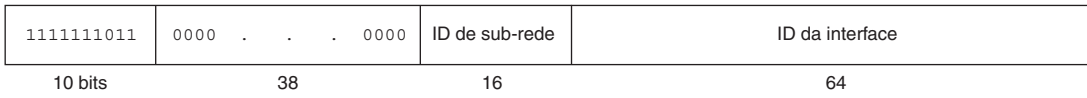


Figura A.13 Endereço de site local do IPv6.

Um roteador IPv6 não deve encaminhar um datagrama com um endereço de origem ou de destino de site local para fora desse site.

A.6 Internet Control Message Protocol (ICMPv4 e ICMPv6)

O ICMP é uma parte exigida e integrante de qualquer implementação de IPv4 ou IPv6. O ICMP é normalmente utilizado para comunicar erro ou mensagens informativas entre nós IP, tanto roteadores como hosts, mas ocasionalmente é utilizado por aplicações. As aplicações `ping` e `traceroute` (Capítulo 28), por exemplo, utilizam ICMP.

Os primeiros 32 bits das mensagens ICMPv4 e ICMPv6 são os mesmos e aparecem na Figura A.14. A RFC 792 (Postel, 1981b) documenta o ICMPv4 e a RFC 2463 (Conta e Deering, 1998) documenta o ICMPv6.

O *tipo* de 8 bits se refere ao tipo da mensagem ICMPv4 ou ICMPv6 e alguns tipos têm um *código* de 8 bits com informações adicionais. A *soma de verificação* é a soma de verificação de Internet padrão, embora no ICMPv4 cubra somente o payload ICMP iniciando com o campo de tipo, enquanto a soma de verificação ICMPv6 também inclui o pseudocabeçalho IPv6.

Da perspectiva da programação de rede, precisamos entender quais mensagens ICMP podem ser retornadas para uma aplicação, o que causa um erro e como um erro é retornado para a aplicação. A Figura A.15 lista todas as mensagens ICMPv4 e como elas são tratadas pelo

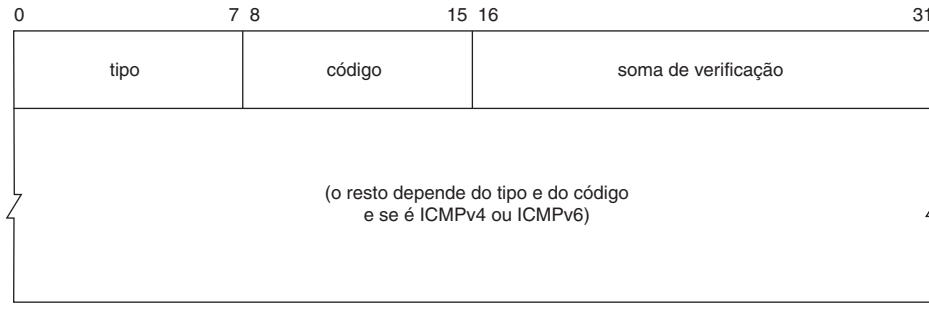


Figura A.14 Formato das mensagens ICMPv4 e ICMPv6.

FreeBSD. A Figura A.16 lista as mensagens ICMPv6. A terceira coluna indica o valor de `errno` retornado por essas mensagens que fazem um erro ser retornado para a aplicação. Ao se utilizar TCP, o erro é anotado, mas não é retornado imediatamente. Se, posteriormente, o TCP abandonar a conexão devido a um tempo-limite, qualquer indicação de erro ICMP será então retornada. Ao se utilizar UDP, a próxima operação de envio ou recebimento recebe o erro, mas somente ao utilizar um soquete conectado, como descrito na Seção 8.9.

tipo	*código*	Descrição	Tratado por ou `errno`	Veja também RFC
0	0	resposta do eco (Ping)	Processo de usuário (Ping)	792
3		destino inacessível:		
	0	rede inacessível	`EHOSTUNREACH`	792
	1	host inacessível	`EHOSTUNREACH`	792
	2	protocolo inacessível	`ECONNREFUSED`	792
	3	porta inacessível (*)	`ECONNREFUSED`	792
	4	fragmentação necessária, mas bit DF ligado	`EMSGSIZE`	792, 1191
	5	a rota de origem falhou	`EHOSTUNREACH`	792
	6	rede de destino desconhecida	`EHOSTUNREACH`	1122
	7	host de destino desconhecido	`EHOSTUNREACH`	1122
	8	host de origem isolado (obsoleto)	`EHOSTUNREACH`	1122
	9	rede de destino proibida administrativamente	`EHOSTUNREACH`	1108, 1122
	10	host de destino proibido administrativamente	`EHOSTUNREACH`	1108, 1122
	11	rede inacessível para TOS	`EHOSTUNREACH`	1122
	12	host inacessível para TOS	`EHOSTUNREACH`	1122
	13	comunicação proibida administrativamente	`ECONNREFUSED`	1812
	14	violação de precedência de host	`ECONNREFUSED`	1812
	15	interrupção de precedência em vigor	`ECONNREFUSED`	1812
4	0	extinção da fonte	Kernel para TCP, ignorado pelo UDP	792, 1812
5		redireciona:		
	0	redireciona para rede	O kernel atualiza tabela de roteamento (†)	792
	1	redireciona para host	O kernel atualiza tabela de roteamento (†)	792
	2	redireciona para TOS e rede	O kernel atualiza tabela de roteamento (†)	792
	3	redireciona para TOS e host	O kernel atualiza tabela de roteamento (†)	792
8	0	solicitação de eco (Ping)	O kernel gera resposta	792
9		anúncio de roteador:		
	0	roteador normal	Processo de usuário	1256
	16	roteador somente de IP móvel	Processo de usuário	2002
10		solicitação de roteador:		
	0	roteador normal	Processo de usuário	1256
	16	roteador somente de IP móvel	Processo de usuário	2002
11		tempo excedido:		
	0	TTL igual a 0 durante o trânsito	Processo de usuário	792
	1	tempo-limite durante a montagem	Processo de usuário	792
12		problema de parâmetro:		
	0	cabeçalho IP defeituoso (erro que abrange tudo)	`ENOPROTOOPT`	792
	1	opção exigida ausente	`ENOPROTOOPT`	1108, 1122
13	0	solicitação de indicador de tempo (timestamp)	O kernel gera resposta	792
14	0	resposta de indicador de tempo (timestamp)	Processo de usuário	792
15	0	solicitação de informações (obsoleto)	Processo de usuário	792
16	0	resposta de informações (obsoleto)	Processo de usuário	792
17	0	solicitação de máscara de endereço	O kernel gera resposta	950
18	0	resposta de máscara de endereço	Processo de usuário	950

* "Porta inacessível" é utilizado somente por protocolos de transporte que não têm seu próprio mecanismo para sinalizar que nenhum processo está ouvindo em uma porta. Por exemplo, o TCP envia uma mensagem de RST, então ele não precisa da mensagem de "porta inacessível".

† Os redirecionamentos são ignorados pelos sistemas que agem como roteadores por encaminhamento de pacotes.

Figura A.15 Tratamento dos tipos de mensagem ICMP pelo FreeBSD.

tipo	*código*	Descrição	Tratado por ou `errno`	Veja também RFC
1		destino inacessível:		
	0	nenhuma rota para o destino	`EHOSTUNREACH`	2463
	1	proibido administrativamente (filtro de firewall)	`EHOSTUNREACH`	2463
	2	fora do escopo do endereço de origem	`ENOPROTOOPT`	2463bis (**)
	3	endereço inacessível	`EHOSTDOWN`	2463
	4	porta inacessível (*)	`ECONNREFUSED`	2463
2	0	pacote grande demais	O kernel faz descoberta de PMTU	2463
3		tempo excedido:		
	0	limite de hop excedido em trânsito	Processo de usuário	2463
	1	tempo de montagem de fragmento excedido	Processo de usuário	2463
4		problema de parâmetro:		
	0	campo de cabeçalho errôneo	`ENOPROTOOPT`	2463
	1	próximo cabeçalho não reconhecido	`ENOPROTOOPT`	2463
	2	opção não reconhecida	`ENOPROTOOPT`	2463
128	0	solicitação de eco (Ping)	O kernel gera resposta	2463
129	0	resposta do eco (Ping)	Processo de usuário (Ping)	2463
130	0	consulta de receptor de multicast	Processo de usuário	2710
131	0	relatório de receptor de multicast	Processo de usuário	2710
132	0	receptor de multicast pronto	Processo de usuário	2710
133	0	solicitação de roteador	Processo de usuário	2461
134	0	anúncio de roteador	Processo de usuário	2461
135	0	solicitação de vizinho	Processo de usuário	2461
136	0	anúncio de vizinho	Processo de usuário	2461
137	0	redireciona	O kernel atualiza tabela de roteamento (†)	2461
141	0	solicitação de vizinho inversa	Processo de usuário	3122
142	0	anúncio de vizinho inverso	Processo de usuário	3122

** "RFC2463bis" designa a revisão em andamento da RFC 2463 (Conta e Deering, 2001).

Figura A.16 Mensagens ICMPv6.

A notação "processo de usuário" significa que o kernel não processa a mensagem e cabe a um processo de usuário com um soquete bruto tratar da mesma. Nenhum retorno de erro é desencadeado para essas mensagens. Também devemos observar que diferentes implementações podem tratar de certas mensagens de forma diferente. Por exemplo, embora os sistemas Unix normalmente tratem das solicitações e anúncios de roteador em um processo de usuário, outras implementações podem tratar dessas mensagens no kernel.

O ICMPv6 limpa o bit de ordem superior do campo *tipo* das mensagens de erro (*tipos* 1-4) e configura esse bit para as mensagens informativas (*tipos* 128-137).

Apêndice **B**

Redes Virtuais

B.1 Visão geral

Quando um novo recurso é adicionado ao TCP, como o suporte para *long fat pipes* (pipes de alta largura de banda chamados de pipes longos e gordos), definido na RFC 1323, o suporte é exigido somente nos hosts que utilizam TCP; nenhuma alteração é exigida nos roteadores. Essas alterações da RFC 1323, por exemplo, estão aparecendo lentamente nas implementações de host do TCP e, quando uma nova conexão TCP é estabelecida, cada extremidade pode determinar se a outra suporta o novo recurso. Se os dois hosts suportam o recurso, ele pode ser utilizado.

Isso difere das alterações que estão sendo feitas na camada de IP, como o multicast no final da década de 1980 e o IPv6 em meados da década de 1990, porque esses novos recursos exigem alterações em todos os hosts *e* em todos os roteadores. Mas, e se as pessoas quiserem começar a utilizar os novos recursos sem esperar que todos os sistemas estejam atualizados? Para fazer isso, uma *rede virtual* é estabelecida sobre a Internet IPv4 existente, utilizando *túneis*.

B.2 A MBone

Nosso primeiro exemplo de rede virtual construída usando túneis é a MBone, que iniciou por volta de 1992 (Eriksson, 1994). Se dois ou mais hosts em uma rede local suportam multicast, as aplicações multicast podem ser executadas em todos esses hosts e comunicar-se entre si. Para conectar essa rede local a alguma outra rede local que também tenha hosts capazes de usar multicast, um túnel é configurado entre dois hosts, um em cada uma das redes locais, como mostrado na Figura B.1. Os passos numerados nessa figura são:

1. Uma aplicação no host de origem, MH1, envia um datagrama multicast para um endereço de classe D.
2. Mostramos isso como um datagrama UDP, pois a maioria das aplicações de multicast utiliza UDP. Falamos mais sobre multicast e como enviar e receber datagramas multicast no Capítulo 21.

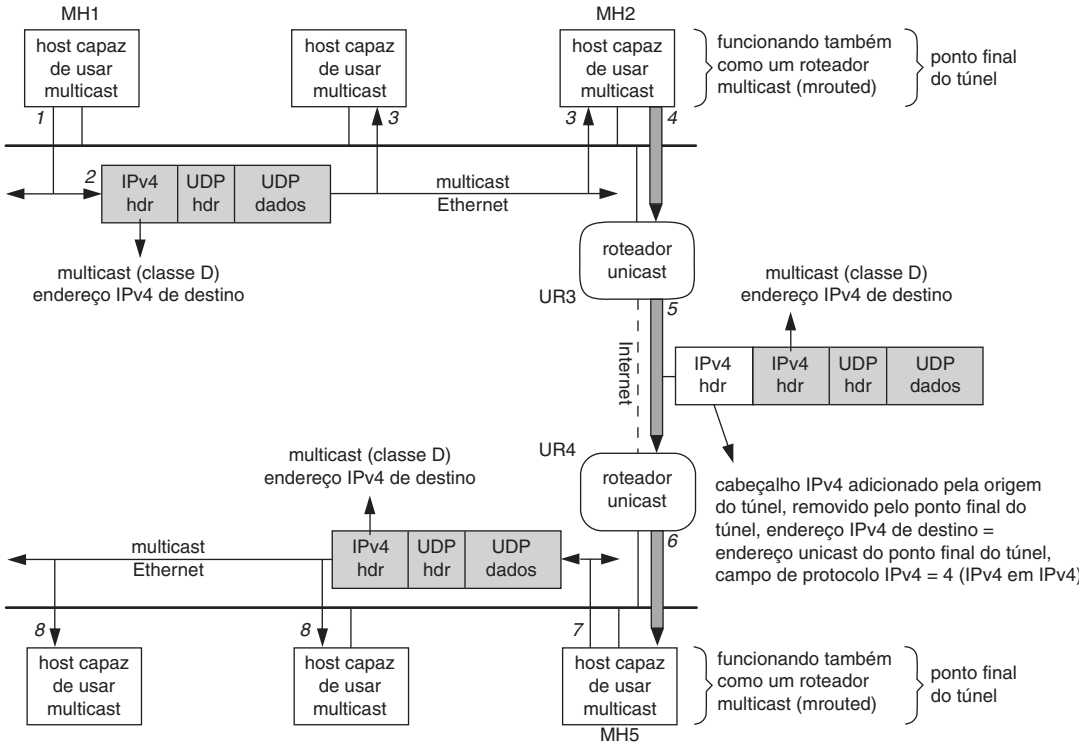


Figura B.1 Encapsulamento de IPv4 em IPv4 utilizado na MBone.

3. O datagrama é recebido por todos os hosts capazes de usar multicast na rede local, inclusive o MR2. Observamos que o MR2 também está funcionando como um roteador multicast, executando o programa `mrouted`, que realiza roteamento de multicast.
4. O MR2 prefixa outro cabeçalho IPv4 na frente do datagrama, com o endereço IPv4 de destino desse novo cabeçalho configurado com o endereço unicast do ponto final do túnel, MR5. Esse endereço unicast é configurado pelo administrador de MR2 e é lido pelo programa `mrouted` quando ele inicia. De maneira semelhante, o endereço unicast do MR2 é configurado para MR5, a outra extremidade do túnel. O campo de protocolo no novo cabeçalho IPv4 é configurado como 4, que é o valor para encapsulamento de IPv4 em IPv4. O datagrama é enviado para o roteador do próximo hop, UR3, que denotamos explicitamente como um roteador unicast. Isto é, o UR3 não entende multicast, que é a razão pela qual estamos utilizando um túnel. A parte sombreada do datagrama IPv4 não mudou em relação ao que foi enviado no Passo 1, a não ser pelo decremento do campo de TTL no cabeçalho IPv4 sombreado.
5. O UR3 vê o endereço IPv4 de destino no cabeçalho IPv4 mais externo e encaminha o datagrama para o roteador de próximo hop, UR4, outro roteador unicast.
6. O UR4 entrega o datagrama para seu destino, MR5, o ponto final do túnel.
7. O MR5 recebe o datagrama e, como o campo de protocolo indica encapsulamento de IPv4 em IPv4, ele remove o primeiro cabeçalho IPv4 e, então, gera saída do restante do datagrama (uma cópia do que foi difundido por multicast na rede local superior) como um datagrama multicast em sua rede local.
8. Todos os hosts capazes de multicast na rede local inferior recebem o datagrama multicast.

O resultado é que o datagrama multicast enviado na rede local superior também é transmitido como um datagrama multicast na rede local inferior. Isso ocorre mesmo que os dois roteadores que mostramos anexados a essas duas redes locais e todos os roteadores de Internet entre eles não sejam capazes de usar multicast.

Nesse exemplo, mostramos a função de roteamento multicast sendo realizada pelo programa `mrouted` executando em um host em cada rede local. Foi assim que a MBone começou. Mas, por volta de 1996, a funcionalidade de roteamento multicast começou a aparecer nos roteadores da maioria dos fornecedores de roteador importantes. Se os dois roteadores unicast UR3 e UR4, na Figura B.1, fossem capazes de usar multicast, então não precisaríamos executar `mrouted` e UR3 e UR4 funcionariam como roteadores multicast. Mas, se ainda há outros roteadores entre UR3 e UR4 que não são capazes de usar multicast, então um túnel é exigido. Os pontos finais do túnel seriam então MR3 (um substituto para UR3 capaz de usar multicast) e MR4 (um substituto de UR4 capaz de usar multicast), e não MR2 e MR5.

> No cenário que mostramos na Figura B.1, cada pacote multicast aparece duas vezes na rede local superior e duas vezes na rede local inferior: uma vez como um pacote multicast e novamente como um pacote unicast dentro do túnel, enquanto o pacote fica entre o host que executa `mrouted` e o roteador unicast do próximo hop (por exemplo, entre MR2 e UR3 e entre UR4 e MR5). Essa cópia extra é o custo do tunelamento. A vantagem de substituir os dois roteadores unicast UR3 e UR4, na Figura B.1, por roteadores capazes de usar multicast (que chamamos de MR3 e MR4) é evitar que essa cópia extra de cada pacote multicast apareça nas redes locais. Mesmo que MR3 e MR4 devam estabelecer um túnel entre eles próprios, porque alguns roteadores intermediários entre eles (que não mostramos) não são capazes de usar multicast, isso ainda é vantajoso, pois evita as cópias duplicadas em cada rede local.

De fato, a MBone é virtualmente inexistente agora, tendo sido dessa maneira substituída por multicast nativo. Provavelmente, ainda há túneis presentes na infra-estrutura de multicast da Internet, mas eles costumam estar entre os roteadores multicast nativos dentro da rede de um provedor de serviços e são invisíveis para o usuário final.

B.3 A 6bone

A 6bone é uma rede virtual que foi criada em 1996 por razões semelhantes às da MBone: os usuários com ilhas de hosts capazes de usar IPv6 queriam interligá-los utilizando uma rede virtual, sem esperar que todos os roteadores intermediários se tornassem capazes de usar IPv6. Quando este livro estava sendo escrito, ela estava sendo deixada de lado, em favor da distribuição de IPv6 nativo; é esperado que a 6bone deixe de operar em junho 2006 (Fink e Hinden, 2003). Abordamos a 6bone aqui porque os exemplos ainda demonstram túneis configurados. Expandiremos o exemplo para incluir túneis dinâmicos, na Seção B.4.

A Figura B.2 mostra um exemplo de duas redes locais capazes de usar IPv6 conectadas a um túnel somente por meio de roteadores IPv4. Os passos numerados nessa figura são:

1. O host H1, na rede local superior, envia um datagrama IPv6 contendo um segmento TCP para o host H4 na rede local inferior. Designamos esses dois hosts como "hosts IPv6", mas ambos provavelmente também executam IPv4. A tabela de roteamento de IPv6 em H1 especifica que o host HR2 é o roteador do próximo hop e um datagrama IPv6 é enviado para esse host.
2. O host HR2 tem um túnel configurado para o host HR3. Esse túnel configurado permite que datagramas IPv6 sejam enviados entre os dois pontos finais do túnel, através de uma Internet IPv4, pelo encapsulamento do datagrama IPv6 em um datagrama IPv4 (chamado de encapsulamento "IPv6 em IPv4"). O campo de protocolo IPv4 tem o valor 41. Observamos que os dois hosts IPv4/IPv6 nas extremidades do túnel, HR2 e HR3, estão ambos agindo como roteadores IPv6, pois estão encaminhando da-

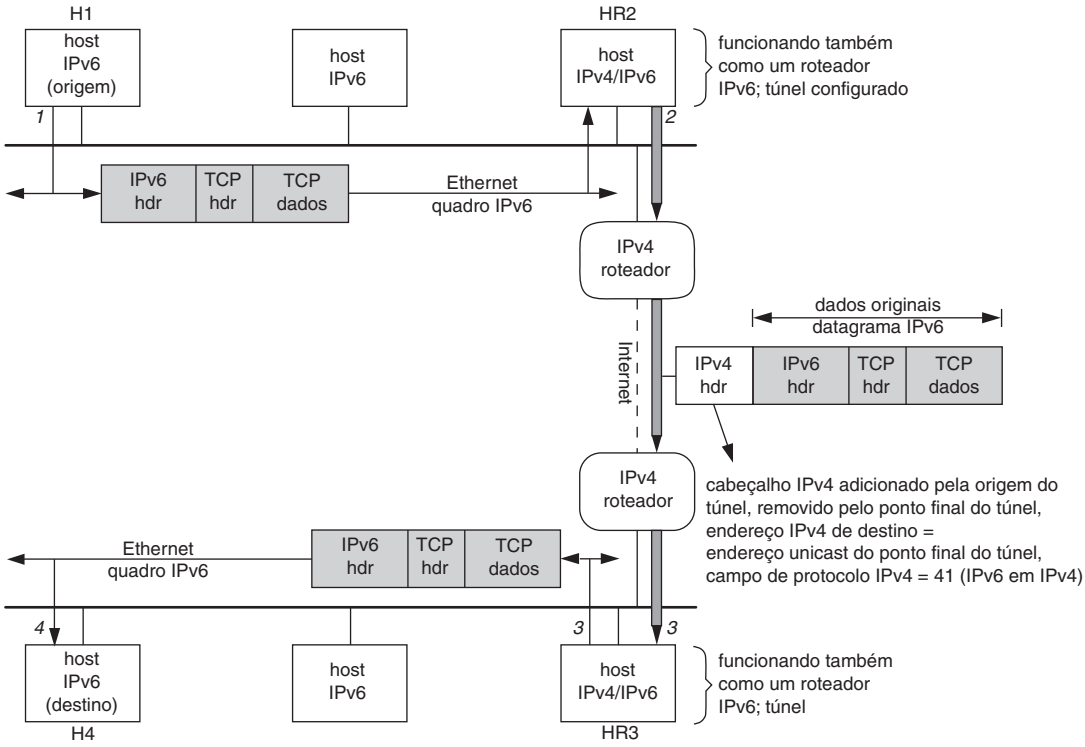


Figura B.2 Encapsulamento de IPv6 em IPv4 na 6bone.

tagramas IPv6 que recebem em uma interface fora da outra interface. O túnel configurado conta como uma interface, mesmo que seja uma interface virtual e não uma interface física.

3. O ponto final do túnel, HR3, recebe o datagrama encapsulado, retira o cabeçalho IPv4 e envia o datagrama IPv6 para sua rede local.
4. O destino, H4, recebe o datagrama IPv6.

B.4 Transição de IPv6: 6to4

O mecanismo de transição 6to4, descrito completamente em "Connection of IPv6 Domains via IPv4 Clouds" (RFC 3056 [Carpenter e Moore, 2001]), é um método de criar dinamicamente os túneis mostrados na Figura B.2. Ao contrário dos mecanismos de túnel dinâmicos projetados anteriormente, que exigiam que cada host envolvido tivesse um endereço IPv4 e estivesse ciente do mecanismo de tunelamento, a 6to4 envolve somente roteadores no processo de tunelamento. Isso permite uma configuração mais simples e uma localização central para impor a política de segurança. Isso também permite a colocação da funcionalidade 6to4 com a função de NAT/firewall comum, que está freqüentemente na margem de uma rede (por exemplo, um pequeno dispositivo de NAT/firewall na extremidade do cliente de uma conexão DSL ou de modem a cabo).

Os endereços 6to4 estão no intervalo `2002/16`. O endereço IPv4 vem após os próximos quatro bytes do endereço, como mostrado na Figura B.3; o prefixo `2002` de 16 bits e o endereço IPv4 de 32 bits criam um identificador de topologia público de 48 bits. Isso deixa dois bytes para o ID de sub-rede, antes do ID de interface de 64 bits. Por exemplo, o prefixo 6to4

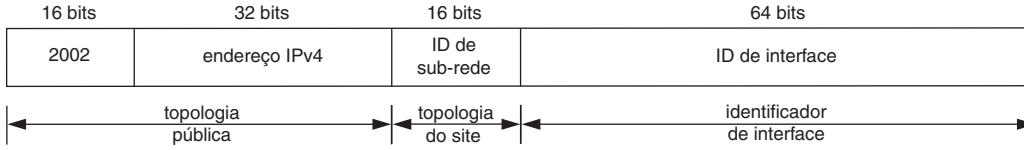


Figura B.3 Endereços 6to4.

correspondente ao nosso host `freebsd`, com endereço IPv4 `12.106.32.254`, é `2002:c6a:20fe/48`.

A vantagem da 6to4 sobre a 6bone é que os túneis que compõem a infra-estrutura 6to4 são construídos automaticamente; não há necessidade de nenhuma configuração previamente organizada. Um site utilizando 6to4 configura um roteador-padrão usando um endereço anycast IPv4 bem-conhecido, `192.88.99.1` (RFC 3068 [Huitema, 2001]). Isso corresponde ao endereço IPv6 `2002:c058:6301::`. Os roteadores na infra-estrutura IPv6 nativa que estejam dispostos a agir como gateways 6to4 anunciam uma rota para `2002/16` e encapsulam todo tráfego para o endereço IPv4 incorporado ao endereço 6to4. Tais roteadores podem ser locais para um site, regionais ou globais, dependendo do escopo de seus anúncios de rota.

O objetivo dessas redes virtuais é que, ao longo do tempo, à medida que os roteadores intermediários ganharem a funcionalidade exigida (por exemplo, roteamento de IPv6 em termos da 6bone e de outros mecanismos de transição de IPv6), as redes virtuais desapareçam.

Apêndice **C**

Técnicas de Depuração

Este apêndice contém algumas sugestões e técnicas para depurar aplicações de rede. Nenhuma técnica sozinha é a resposta para tudo; em vez disso, há várias ferramentas que devemos conhecer para, então, utilizar a que funcionar em nosso ambiente.

C.1 Rastreamento de chamadas de sistema

Muitas versões do Unix fornecem um recurso de rastreamento de chamadas de sistema. Isso freqüentemente pode fornecer uma valiosa técnica de depuração.

Trabalhando nesse nível, precisamos diferenciar entre uma *chamada de sistema* e uma *função*. A primeira é um ponto de entrada no kernel e é o que podemos controlar com as ferramentas que veremos nesta seção. O POSIX e a maioria dos outros padrões utilizam o termo "função" para descrever o que parece ser funções para o usuário, mesmo que, em algumas implementações, elas possam ser chamadas de sistema. Por exemplo, em um kernel derivado do Berkeley, `socket` é uma chamada de sistema, mesmo que pareça ser uma função C normal para o programador de aplicação. Porém, veremos em breve, no SVR4 trata-se de uma função na biblioteca de soquetes que faz chamadas a `putmsg` e `getmsg`, sendo estas duas chamadas de sistema reais.

Nesta seção, examinaremos as chamadas de sistema envolvidas na execução de nosso cliente de data/hora. Mostramos esse cliente na Figura 1.5.

Soquetes do kernel BSD

Iniciamos com o FreeBSD, um kernel derivado do Berkeley em que todas as funções de soquete são chamadas de sistema. O programa `ktrace` é fornecido pelo FreeBSD para executar um programa e rastrear as chamadas de sistema que são executadas. Ele grava as informações de rastreamento em um arquivo (cujo nome-padrão é `ktrace.out`), que imprimimos com `kdump`. Executamos nosso cliente de soquetes como

```
freebsd % ktrace daytimetcpcli 192.168.42.2
Tue Aug 19 23:35:10 2003
```

Então, executamos kdump para gerar saída das informações de rastreamento na saída-padrão.

```
3211 daytimetcpcli CALL  socket(0x2,0x1,0)
3211 daytimetcpcli RET   socket 3

3211 daytimetcpcli CALL  connect(0x3,0x7fdffffe820,0x10)
3211 daytimetcpcli RET   connect 0

3211 daytimetcpcli CALL  read(0x3,0x7fdffffe830,0x1000)
3211 daytimetcpcli GIO   fd 3 read 26 bytes
     "Tue Aug 19 23:35:10 2003
     "
3211 daytimetcpcli RET   read 26/0x1a
...

3211 daytimetcpcli CALL  write(0x1,0x204000,0x1a)
3211 daytimetcpcli GIO   fd 1 wrote 26 bytes
     "Tue Aug 19 23:35:10 2003\r
     "
3211 daytimetcpcli RET   write 26/0x1a

3211 daytimetcpcli CALL  read(0x3,0x7fdffffe830,0x1000)
3211 daytimetcpcli GIO   fd 3 read 0 bytes
     ""
3211 daytimetcpcli RET   read 0

3211 daytimetcpcli CALL  exit(0)
```

3211 é o PID. CALL identifica uma chamada de sistema, RET é o retorno e GIO significa E/S de processo genérica. Vemos as chamadas a socket e connect, seguidas da chamada a read que retorna 26 bytes. Nosso cliente grava esses bytes na saída-padrão e a próxima chamada de read retorna 0 (EOF).

Soquetes do kernel Solaris 9

O Solaris 2.x é baseado no SVR4 e todas as versões antes da 2.6 implementavam soquetes como mostrado na Figura 31.3. Um problema, entretanto, com todas as implementações de SVR4 que implementam soquetes dessa forma é que elas raramente fornecem 100% de compatibilidade com soquetes de kernel derivados do Berkeley. Para fornecer compatibilidade adicional, as versões a partir do Solaris 2.6 mudaram a técnica de implementação e implementaram soquetes utilizando um sistema de arquivos sockfs. Isso fornece soquetes de kernel, como podemos verificar utilizando truss em nosso cliente de soquetes.

```
solaris % truss -v connect daytimetcpcli 127.0.0.1
Mon Sep 8 12:16:42 2003
```

Após o vínculo de biblioteca normal, a primeira chamada de sistema que vemos é para so_socket, uma chamada de sistema ativada por nossa chamada a socket.

```
so_socket(PF_INET, SOCK_STREAM, IPPROTO_IP, "", 1) = 3
connect(3, 0xFFBFDEF0, 16, 1)                   =
        AF_INET name = 127.0.0.1 port = 13
read(3, " M o n  S e p       8   1".., 4096)    = 26
Mon Sep 8 12:48:06 2003
write(1, " M o n  S e p       8   1".., 26)     = 26
read(3, 0xFFBFDF03, 4096)                       = 0
_exit(0)
```

Os três primeiros argumentos de so_socket são nossos três argumentos para socket. Vemos que connect é uma chamada de sistema e truss, quando invocado com o flag -v connect, imprime o conteúdo da estrutura de endereço de soquete apontado pelo segundo

argumento (o endereço IP e o número da porta). As únicas chamadas de sistema que omitimos são algumas que lidam com entrada e saída-padrão.

C.2 Serviços-padrão da Internet

Conheça os serviços-padrão da Internet descritos na Figura 2.18. Utilizamos o serviço de data/hora muitas vezes para testar nossos clientes. O serviço de descarte é uma porta conveniente para a qual podemos enviar dados. O serviço de eco é semelhante ao servidor de eco que utilizamos por todo este livro.

> Muitos sites agora evitam o acesso a tais serviços por meio de firewalls, devido a alguns ataques de recusa de serviço que os utilizaram em 1996 (Exercício 13.3). Contudo, espera-se que você possa utilizar esses serviços dentro de sua própria rede.

C.3 Programa `sock`

O programa `sock` de Stevens apareceu pela primeira vez no TCPv1, onde era freqüentemente utilizado para gerar condições de casos especiais, a maioria das quais era então examinada no texto utilizando `tcpdump`. O interessante no programa é que ele gera muitos cenários diferentes, evitando que tenhamos de escrever programas de teste especiais.

Não mostraremos seu código-fonte (são mais de 2.000 linhas de código C), mas ele está livremente disponível (veja o Prefácio). O programa opera em um de quatro modos, cada um dos quais pode utilizar TCP ou UDP:

- Entrada-padrão, cliente de saída-padrão (Figura C.1):

  No modo cliente, tudo que é lido da entrada-padrão é escrito na rede e tudo que é recebido da rede é escrito na saída-padrão. O endereço IP do servidor e a porta devem ser especificados e, no caso do TCP, uma abertura ativa é realizada.
- Entrada-padrão, servidor de saída-padrão. Este modo é semelhante ao anterior, exceto que o programa vincula uma porta bem-conhecida ao seu soquete e, no caso do TCP, realiza uma abertura passiva.
- Cliente-fonte (Figura C.2):

  O programa realiza um número fixo de escritas em uma rede de algum tamanho especificado.
- Servidor-destino (Figura C.3):

O programa realiza um número fixo de leituras de uma rede.

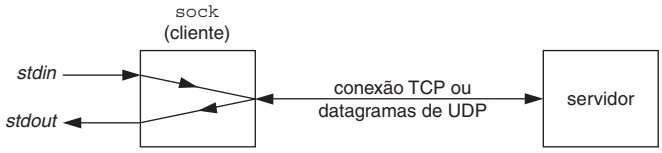


Figura C.1 Cliente `sock`, entrada-padrão, saída-padrão.

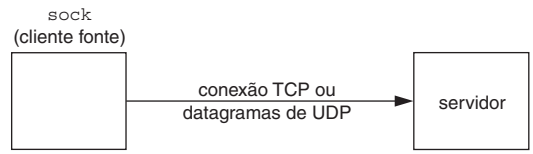


Figura C.2 Programa `sock` como cliente-fonte.

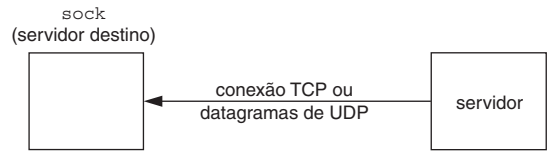


Figura C.3 Programa `sock` como servidor-destino.

Esses quatro modos de operação correspondem aos quatro comandos a seguir:

```
sock [opções] serviço nome-de-host
sock [opções] -s [nome-de-host] serviço
sock [opções] -i serviço nome-de-host
sock [opções] -is [nome-de-host] serviço
```

em que *nome-de-host* é um nome do host ou endereço IP e *serviço* é um nome do serviço ou número da porta. Nos dois modos de servidor, o endereço curinga é vinculado, a menos que o *nome-de-host* opcional seja especificado.

Aproximadamente 40 opções de linha de comando também podem ser especificadas e dirigem os recursos opcionais do programa. Não detalharemos essas opções aqui, mas muitas das opções de soquete descritas no Capítulo 7 podem ser configuradas. Executar o programa sem quaisquer argumentos imprime um resumo das opções.

```
-b n  bind n as client's local port number
-c    convert newline to CR/LF & vice versa
-f a.b.c.d.p foreign IP address = a.b.c.d, foreign port # = p
-g a.b.c.d loose source route
-h    issue TCP half-close on standard input EOF
-i    "source" data to socket, "sink" data from socket (w/-s)
-j a.b.c.d join multicast group
-k    write or writev in chunks
-l a.b.c.d.p client's local IP address = a.b.c.d, local port # = p
-n n  # buffers to write for "source" client (default 1024)
-o    do NOT connect UDP client
-p n  # ms to pause before each read or write (source/sink)
-q n  size of listen queue for TCP server (default 5)
-r n  # bytes per read() for "sink" server (default 1024)
-s    operate as server instead of client
-t n  set multicast ttl
-u    use UDP instead of TCP
-v    verbose
-w n  # bytes per write() for "source" client (default 1024)
-x n  # ms for SO_RCVTIMEO (receive timeout)
-y n  # ms for SO_SNDTIMEO (send timeout)
-A    SO_REUSEADDR option
-B    SO_BROADCAST option
-C    set terminal to cbreak mode
-D    SO_DEBUG option
-E    IP_RECVDSTADDR option
-F    fork after connection accepted (TCP concurrent server)
-G a.b.c.d strict source route
-H n  IP_TOS option (16=min del, 8=max thru, 4=max rel, 2=min cost)
-I    SIGIO signal
-J n  IP_TTL option
-K    SO_KEEPALIVE option
-L n  SO_LINGER option, n = linger time
-N    TCP_NODELAY option
-O n  # ms to pause after listen, but before first accept
-P n  # ms to pause before first read or write (source/sink)
-Q n  # ms to pause after receiving FIN, but before close
-R n  SO_RCVBUF option
```

```
-S n   SO_SNDBUF option
-T     SO_REUSEPORT option
-U n   enter urgent mode before write number n (source only)
-V     use writev() instead of write(); enables -k too
-W     ignore write errors for sink client
-X n   TCP_MAXSEG option (set MSS)
-Y     SO_DONTROUTE option
-Z     MSG_PEEK
```

C.4 Pequenos programas de teste

Outra técnica de depuração útil, que os autores sempre utilizam, é escrever pequenos programas de teste para ver como um recurso específico funciona em um caso de teste cuidadosamente construído. Escrever pequenos programas de teste ajuda a ter um conjunto de funções empacotadoras de biblioteca e algumas funções de erro simples, como aquelas que utilizamos ao longo deste livro. Isso reduz a quantidade de código que temos de escrever, mas ainda fornece o teste necessário para erros.

C.5 Programa `tcpdump`

Uma ferramenta inestimável ao lidar com programação de rede é o `tcpdump`. Esse programa lê pacotes de uma rede e imprime uma grande quantidade de informações sobre eles. Ele também tem a capacidade de imprimir somente os pacotes que correspondem a algum critério que especifiquemos. Por exemplo:

```
% tcpdump '(udp and port daytime) or icmp'
```

imprime somente os datagramas de UDP com a porta de origem ou de destino 13 (o servidor de data/hora) ou pacotes de ICMP. O comando a seguir:

```
% tcpdump 'tcp and port 80 and tcp[13:1] & 2 != 0'
```

imprime somente os segmentos TCP com a porta de origem ou de destino 80 (o servidor HTTP) que têm o flag SYN ligado. O flag SYN tem um valor igual a 2 no byte com um deslocamento de 13 a partir do início do cabeçalho TCP. O comando a seguir:

```
% tcpdump 'tcp and tcp[0:2] > 7000 and tcp[0:2] <= 7005'
```

imprime somente segmentos TCP com a porta de origem entre 7001 e 7005. A porta de origem inicia no deslocamento de byte 0 no cabeçalho TCP e ocupa 2 bytes.

O Apêndice A do TCPv1 oferece mais detalhes da operação desse programa.

> Esse programa está disponível no endereço `http://www.tcpdump.org/` e funciona sob muitas versões diferentes do Unix. Ele foi escrito originalmente por Van Jacobson, Craig Leres e Steven McCanne, na LBL, e agora é mantido por uma equipe no endereço `tcpdump.org`.
>
> Alguns vendedores fornecem um programa próprio com funcionalidade semelhante. Por exemplo, o Solaris 2.x fornece o programa `snoop`. A vantagem do `tcpdump` é que ele funciona sob muitas versões do Unix e utilizar uma única ferramenta em um ambiente heterogêneo, em vez de uma ferramenta diferente para cada ambiente, é uma grande vantagem.

C.6 Programa `netstat`

Utilizamos o programa `netstat` muitas vezes neste livro. Ele tem múltiplos propósitos:

- Mostrar o *status* dos pontos finais de rede. Mostramos isso na Seção 5.6, quando acompanhamos o *status* dos nosso ponto final quando iniciamos nossos cliente e servidor.

- Mostrar os grupos de multicast a que um host pertence, em cada interface. Os flags –`ia` são a maneira normal de mostrar isso ou o flag –`g` sob o Solaris 2.x.
- Mostrar a estatística por protocolo, com a opção –`s`. Mostramos isso na Seção 8.13, quando vimos a falta de controle de fluxo com UDP.
- Exibir a tabela de roteamento com a opção –`r` e as informações de interface com a opção –`i`. Mostramos isso na Seção 1.9, quando utilizamos `netstat` para descobrir a topologia da nossa rede.

Há outras utilizações de `netstat` e a maioria dos fornecedores adicionou seus próprios recursos. Verifique a página man em seu sistema.

C.7 Programa `lsof`

O nome `lsof` significa "listar arquivos abertos". Assim como o `tcpdump`, trata-se de uma ferramenta publicamente disponível que é útil para depurar e que foi portada para muitas versões do Unix.

Uma utilização comum do `lsof` com rede é localizar qual processo tem um soquete aberto em um endereço IP ou porta especificada. `netstat` nos diz quais endereços IP e portas estão em uso e o estado das conexões TCP, mas não identifica o processo. Por exemplo, para descobrirmos qual processo fornece o servidor de data/hora, executamos o seguinte:

```
freebsd % lsof -i TCP:daytime
COMMAND   PID USER  FD  TYPE        DEVICE SIZE/OFF NODE NAME
inetd     561 root  5u  IPv4 0xfffff8003027a260  0t0 TCP *:daytime (LISTEN)
inetd     561 root  7u  IPv6 0xfffff800302b6720  0t0 TCP *:daytime
```

Isso nos informa o comando (esse serviço é fornecido pelo servidor `inetd`), sua PID, o proprietário, o descritor (5 para IPv4 e 7 para IPv6, e o `u` significa que ele está aberto para leitura/gravação), o tipo de soquete, o endereço do bloco de controle do protocolo, o tamanho ou deslocamento do arquivo (não significativo para um soquete), o tipo de protocolo e o nome.

Uma utilização comum para esse programa é quando iniciamos um servidor que vincula sua porta bem-conhecida e obtém o erro de que o endereço já está em uso. Então, usamos `lsof` para localizar o processo que está utilizando a porta.

Como o `lsof` relata os arquivos abertos, ele não pode informar sobre pontos finais de rede que não estejam associados a um arquivo aberto: pontos finais de TCP no estado TIME_WAIT.

> `ftp://lsof.itap.purdue.edu/pub/tools/unix/lsof/` é o local desse programa. Ele foi escrito por Vic Abell.
>
> Alguns fornecedores distribuem seus próprios utilitários para fazer coisas semelhantes. Por exemplo, o FreeBSD fornece o programa `fstat`. A vantagem do `lsof` é que ele funciona sob muitas versões do Unix e utilizar uma única ferramenta em um ambiente heterogêneo, em vez de uma ferramenta diferente para cada ambiente, é uma grande vantagem.

Apêndice **D**

Miscelânea de Código-Fonte

D.1 Cabeçalho `unp.h`

Quase todos programas apresentados neste livro incluem nosso cabeçalho `unp.h`, mostrado na Figura D.1. Esse cabeçalho inclui todos os cabeçalhos de sistema-padrão de que a maioria dos programas de rede precisa, junto com alguns cabeçalhos de sistema gerais. Ele também define constantes, como `MAXLINE`, protótipos de função C ANSI para as funções que definimos no texto (por exemplo, `readline`) e todas as funções empacotadoras que utilizamos. No entanto, não mostramos esses protótipos.

lib/unp.h

```
 1 /* Nosso próprio cabeçalho. Tabulações configuradas c/ 4 espaços e não 8 */

 2 #ifndef __unp_h
 3 #define __unp_h

 4 #include   "../config.h"     /* opções de configuração para SO corrente */
 5                           /*"../config.h" é gerado pelo script configure */

 6 /* Se algo mudar na lista de #includes a seguir, deve mudar
 7    acsite.m4 também, para os testes de configure. */

 8 #include   <sys/types.h>     /* tipos de dados de sistema básicos */
 9 #include   <sys/socket.h>    /* definições de soquete básicas */
10 #include   <sys/time.h>      /* timeval{} para select() */
11 #include   <time.h>          /* timespec{} para pselect() */
12 #include   <netinet/in.h>    /* sockaddr_in{} e outras defs Internet */
13 #include   <arpa/inet.h>     /* funções inet(3) */
14 #include   <errno.h>
15 #include   <fcntl.h>         /* para não bloquear */
16 #include   <netdb.h>
17 #include   <signal.h>
18 #include   <stdio.h>
19 #include   <stdlib.h>
20 #include   <string.h>
```

Figura D.1 Nosso cabeçalho `unp.h` (*continua*).

```
21 #include   <sys/stat.h>        /* para constantes de modo de arquivo S_xxx */
22 #include   <sys/uio.h>        /* para iovec{} e readv/writev */
23 #include   <unistd.h>
24 #include   <sys/wait.h>
25 #include   <sys/un.h>         /* para soquetes de domínio Unix */

26 #ifdef HAVE_SYS_SELECT_H
27 # include  <sys/select.h>     /* por conveniência */
28 #endif

29 #ifdef HAVE_SYS_SYSCTL_H
30 # include  <sys/sysctl.h>
31 #endif

32 #ifdef HAVE_POLL_H
33 # include  <poll.h>           /* por conveniência */
34 #endif

35 #ifdef HAVE_SYS_EVENT_H
36 # include  <sys/event.h>      /* para kqueue */
37 #endif

38 #ifdef HAVE_STRINGS_H
39 # include  <strings.h>        /* por conveniência */
40 #endif

41 /* Três cabeçalhos normalmente são necessários para ioctl de soquete/arquivo:
42  * <sys/ioctl. h>, <sys/filio. h> e <sys/sockio. h>.
43  */
44 #ifdef HAVE_SYS_IOCTL_H
45 # include  <sys/ioctl.h>
46 #endif
47 #ifdef HAVE_SYS_FILIO_H
48 # include  <sys/filio.h>
49 #endif
50 #ifdef HAVE_SYS_SOCKIO_H
51 # include  <sys/sockio.h>
52 #endif

53 #ifdef HAVE_PTHREAD_H
54 # include  <pthread.h>
55 #endif

56 #ifdef HAVE_NET_IF_DL_H
57 # include  <net/if_dl.h>
58 #endif

59 #ifdef HAVE_NETINET_SCTP_H
60 #include   <netinet/sctp.h>
61 #endif

62 /* OSF/1 realmente desativa recv() e send() em <sys/socket. h> */
63 #ifdef __osf__
64 #undef recv
65 #undef send
66 #define recv(a,b,c,d)  recvfrom(a,b,c,d,0,0)
67 #define send(a,b,c,d)  sendto(a,b,c,d,0,0)
68 #endif

69 #ifndef INADDR_NONE
70 #define INADDR_NONE 0xffffffff  /* deve ter sido em <netinet/in. h> */
71 #endif

72 #ifndef SHUT_RD               /* esses três nomes POSIX são novos */
73 #define SHUT_RD     0         /* desligamento para leitura */
74 #define SHUT_WR     1         /* desligamento para gravação */
```

Figura D.1 Nosso cabeçalho `unp.h` (*continua*).

```
 75 #define SHUT_RDWR   2        /* desligamento para leitura e gravação */
 76 #endif

 77 #ifndef INET_ADDRSTRLEN
 78 #define INET_ADDRSTRLEN   16 /* "ddd.ddd.ddd.ddd\0"
 79                                 1234567890123456 */
 80 #endif

 81 /* Define o seguinte, mesmo que IPv6 não seja suportado, para que sempre
 82    possamos alocar um buffer de tamanho adequado sem #ifdefs no código. */
 83 #ifndef INET6_ADDRSTRLEN
 84 #define INET6_ADDRSTRLEN  46 /* tamanho máx da string de endereço IPv6:
 85                      "xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx" ou
 86                      "xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:ddd.ddd.ddd.ddd\0"
 87                       1234567890123456789012345678901234567890123456 */
 88 #endif

 89 /* Define bzero() como uma macro, se não estiver na biblioteca C-padrão. */
 90 #ifndef HAVE_BZERO
 91 #define bzero(ptr,n)         memset(ptr, 0, n)
 92 #endif

 93 /* Os resolvedores mais antigos não têm gethostbyname2() */
 94 #ifndef HAVE_GETHOSTBYNAME2
 95 #define gethostbyname2(host,family)     gethostbyname((host))
 96 #endif

 97 /* A estrutura retornada por recvfrom_flags() */
 98 struct unp_in_pktinfo {
 99    struct in_addr ipi_addr;  /* endereço IPv4 de dest */
100    int    ipi_ifindex;       /* índice de interface recebido */
101 };

102 /* Precisamos das macros CMSG_LEN() e CMSG_SPACE() mais recentes, mas poucas
103    implementações as suportam hoje. Essas duas macros realmente precisam
104     de uma macro ALIGN(), mas cada implementação faz isso diferentemente. */
105 #ifndef CMSG_LEN
106 #define CMSG_LEN(size)       (sizeof(struct cmsghdr) + (size))
107 #endif
108 #ifndef CMSG_SPACE
109 #define CMSG_SPACE(size)     (sizeof(struct cmsghdr) + (size))
110 #endif

111 /* O POSIX exige a macro SUN_LEN(), mas nem todas as implementações a definem
112    (ainda). Observe que essa macro 4.4BSD funciona independentemente de haver
113    um campo de comprimento ou não. */
114 #ifndef SUN_LEN
115 # define   SUN_LEN(su) \
116     (sizeof(*(su)) - sizeof((su)->sun_path) + strlen((su)->sun_path))
117 #endif

118 /* O POSIX renomeia "domínio Unix" como "IPC local".
119    Nem todos os sistemas definem AF_LOCAL e PF_LOCAL (ainda). */
120 #ifndef AF_LOCAL
121 #define AF_LOCAL    AF_UNIX
122 #endif
123 #ifndef PF_LOCAL
124 #define PF_LOCAL    PF_UNIX
125 #endif

126 /* O POSIX exige que um #include de <poll.h> defina INFTIM, mas muitos
127    sistemas ainda o definem em <sys/stropts.h>. Não queremos incluir todo
128    o material de STREAMS, se não for necessário, então, definimos apenas INFTIM
129    aqui. Esse é o valor-padrão, mas não há nenhuma garantia de que seja -1. */
130 #ifndef INFTIM
```

Figura D.1 Nosso cabeçalho `unp.h` (*continua*).

```
131 #define INFTIM          (-1)  /* tempo-limite de consulta seqüencial
                                     infinito */
132 #ifdef HAVE_POLL_H
133 #define INFTIM_UNPH           /* diz a unpxti.h que o definimos */
134 #endif
135 #endif

136 /* O seguinte poderia ser derivado de SOMAXCONN em <sys/socket.h>, mas muitos
137    kernels ainda o definem como 5, embora suportem muito mais */
138 #define LISTENQ     1024      /* segundo argumento de listen() */

139 /* Constantes diversas */
140 #define MAXLINE     4096      /* comprimento de linha de texto máx. */

141 #define BUFFSIZE    8192      /* tamanho de buffer para leituras e gravações */

142 /* Define algum número de porta que possa ser utilizado por nossos exemplos */
143 #define SERV_PORT        9877   /* TCP e UDP */
144 #define SERV_PORT_STR   "9877"  /* TCP e UDP */
145 #define UNIXSTR_PATH    "/tmp/unix.str"  /* Fluxo de domínio Unix */
146 #define UNIXDG_PATH     "/tmp/unix.dg"   /* Datagrama de domínio Unix */

147 /* O seguinte reduz todas as conversões de tipo de argumentos de ponteiro: */
148 #define SA struct sockaddr

149 #define HAVE_STRUCT_SOCKADDR_STORAGE
150 #ifndef HAVE_STRUCT_SOCKADDR_STORAGE
151 /*
152  * RFC 3493: marcador de lugar independente de protocolo para endereços de soquete
153  */
154 #define __SS_MAXSIZE    128
155 #define __SS_ALIGNSIZE  (sizeof(int64_t))
156 #ifdef HAVE_SOCKADDR_SA_LEN
157 #define __SS_PAD1SIZE     (__SS_ALIGNSIZE - sizeof(u_char) - sizeof(sa_family_t))
158 #else
159 #define __SS_PAD1SIZE     (__SS_ALIGNSIZE - sizeof(sa_family_t))
160 #endif
161 #define __SS_PAD2SIZE     (__SS_MAXSIZE - 2*__SS_ALIGNSIZE)

162 struct sockaddr_storage {
163 #ifdef HAVE_SOCKADDR_SA_LEN
164     u_char ss_len;
165 #endif
166     sa_family_t ss_family;
167     char     __ss_pad1[__SS_PAD1SIZE];
168     int64_t __ss_align;
169     char     __ss_pad2[__SS_PAD2SIZE];
170 };
171 #endif

172 #define FILE_MODE   (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
173                     /* permissões de acesso de arquivo default para novos
                        arquivos */
174 #define DIR_MODE    (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)
175                     /* permissões default para novos diretórios */

176 typedef void Sigfunc (int);      /* para handlers de sinal */

177 #define min(a,b)    ((a) < (b) ? (a) : (b))
178 #define max(a,b)    ((a) > (b) ? (a) : (b))

179 #ifndef HAVE_ADDRINFO_STRUCT
180 # include "../lib/addrinfo.h"
181 #endif

182 #ifndef HAVE_IF_NAMEINDEX_STRUCT
```

Figura D.1 Nosso cabeçalho `unp.h` (*continua*).

```
183 struct if_nameindex {
184     unsigned int if_index;    /* 1, 2, ... */
185     char *if_name;            /* nome terminado por caractere nulo: "le0", ... */
186 };
187 #endif

188 #ifndef HAVE_TIMESPEC_STRUCT
189 struct timespec {
190     time_t  tv_sec;           /* segundos */
191     long  tv_nsec;            /* e nanossegundos */
192 };
193 #endif
```

lib/unp.h

Figura D.1 Nosso cabeçalho `unp.h` (*continuação*).

D.2 Cabeçalho `config.h`

A ferramenta GNU `autoconf` foi utilizada para ajudar na portabilidade de todos os códigos-fonte neste livro. Ela está disponível no endereço `http://ftp.gnu.org/gnu/autoconf`. Essa ferramenta gera um script de shell chamado `configure`, que você deve executar depois de fazer download do software para seu sistema. Esse script determina os recursos fornecidos pelo seu sistema Unix: As estruturas de endereço de soquete têm um campo de comprimento? Multicast é suportado? As estruturas de endereço de soquete de enlace de dados são suportadas? e assim por diante, gerando um cabeçalho chamado `config.h`. Esse cabeçalho é o primeiro incluído por nosso cabeçalho `unp.h` na seção anterior. A Figura D.2 mostra o cabeçalho `config.h` para FreeBSD 5.1.

As linhas que iniciam com `#define` na coluna 1 correspondem aos recursos que o sistema fornece. As linhas que são desativadas e contêm `#undef` correspondem aos recursos que o sistema não fornece.

sparc64-unknown-freebsd5.1/config.h

```
 1 /* config.h. Gerado automaticamente pelo script configure. */
 2 /* config.h.in. Gerado automaticamente a partir de configure.in por
   autoheader. */

 3 /* CPU, fornecedor, e sistema operacional */
 4 #define CPU_VENDOR_OS "sparc64-unknown-freebsd5.1"

 5 /* Define se <netdb.h> define a estrutura addrinfo */
 6 #define HAVE_ADDRINFO_STRUCT 1

 7 /* Define se você tem o arquivo de cabeçalho <arpa/inet.h>. */
 8 #define HAVE_ARPA_INET_H 1

 9 /* Define se você tem a função bzero. */
10 #define HAVE_BZERO 1

11 /* Define se o dispositivo /dev/streams/xtiso/tcp existe */
12 /* #undef HAVE_DEV_STREAMS_XTISO_TCP */

13 /* Define se o dispositivo /dev/tcp existe */
14 /* #undef HAVE_DEV_TCP */

15 /* Define se o dispositivo /dev/xti/tcp existe */
16 /* #undef HAVE_DEV_XTI_TCP */

17 /* Define se você tem o arquivo de cabeçalho <errno.h>. */
18 #define HAVE_ERRNO_H 1
```

Figura D.2 Nosso cabeçalho `config.h` para FreeBSD 5.1 (*continua*).

```
19 /* Define se você tem o arquivo de cabeçalho <fcntl.h>. */
20 #define HAVE_FCNTL_H 1

21 /* Define se você tem a função getaddrinfo. */
22 #define HAVE_GETADDRINFO 1

23 /* define se o protótipo getaddrinfo está em <netdb.h> */
24 #define HAVE_GETADDRINFO_PROTO 1

25 /* Define se você tem a função gethostbyname2. */
26 #define HAVE_GETHOSTBYNAME2 1

27 /* Define se você tem a função gethostbyname_r function. */
28 /* #undef HAVE_GETHOSTBYNAME_R */

29 /* Define se você tem a função gethostname. */
30 #define HAVE_GETHOSTNAME 1

31 /* define se o protótipo gethostname está em <unistd.h> */
32 #define HAVE_GETHOSTNAME_PROTO 1

33 /* Define se você tem a função getnameinfo. */
34 #define HAVE_GETNAMEINFO 1

35 /* define se o protótipo getnameinfo está em <netdb.h> */
36 #define HAVE_GETNAMEINFO_PROTO 1

37 /* define se o protótipo getrusage está em <sys/resource.h> */
38 #define HAVE_GETRUSAGE_PROTO 1

39 /* Define se você tem a função hstrerror. */
40 #define HAVE_HSTRERROR 1

41 /* Define se o protótipo hstrerror está em <netdb.h> */
42 #define HAVE_HSTRERROR_PROTO 1

43 /* Define se <net/if.h> define struct if_nameindex */
44 #define HAVE_IF_NAMEINDEX_STRUCT 1

45 /* Define se você tem a função if_nametoindex. */
46 #define HAVE_IF_NAMETOINDEX 1

47 /* define se o protótipo if_nametoindex está em <net/if.h> */
48 #define HAVE_IF_NAMETOINDEX_PROTO 1

49 /* Define se você tem a função inet_aton. */
50 #define HAVE_INET_ATON 1

51 /* define se o protótipo inet_aton está em <arpa/inet.h> */
52 #define HAVE_INET_ATON_PROTO 1

53 /* Define se você tem a função inet_pton. */
54 #define HAVE_INET_PTON 1

55 /* define se o protótipo inet_pton está em <arpa/inet.h> */
56 #define HAVE_INET_PTON_PROTO 1

57 /* Define se você tem a função kevent. */
58 #define HAVE_KEVENT 1

59 /* Define se você tem a função kqueue. */
60 #define HAVE_KQUEUE 1

61 /* Define se você tem a biblioteca nsl (-lnsl). */
62 /* #undef HAVE_LIBNSL */

63 /* Define se você tem a biblioteca pthread (-lpthread). */
64 /* #undef HAVE_LIBPTHREAD */
```

Figura D.2 Nosso cabeçalho `config.h` para FreeBSD 5.1 (*continua*).

```
 65 /* Define se você tem a biblioteca pthreads (-lpthreads). */
 66 /* #undef HAVE_LIBPTHREADS */

 67 /* Define se você tem a biblioteca resolv (-lresolv). */
 68 /* #undef HAVE_LIBRESOLV */

 69 /* Define se você tem a biblioteca xti (-lxti). */
 70 /* #undef HAVE_LIBXTI */

 71 /* Define se você tem a função mkstemp. */
 72 #define HAVE_MKSTEMP 1

 73 /* Define se a estrutura msghdr contém o elemento msg_control */
 74 #define HAVE_MSGHDR_MSG_CONTROL 1

 75 /* Define se você tem o arquivo de cabeçalho <netconfig.h>. */
 76 #define HAVE_NETCONFIG_H 1

 77 /* Define se você tem o arquivo de cabeçalho <netdb.h>. */
 78 #define HAVE_NETDB_H 1

 79 /* Define se você tem o arquivo de cabeçalho <netdir.h>. */
 80 /* #undef HAVE_NETDIR_H */

 81 /* Define se você tem o arquivo de cabeçalho <netinet/in.h>. */
 82 #define HAVE_NETINET_IN_H 1

 83 /* Define se você tem o arquivo de cabeçalho <net/if_dl.h>. */
 84 #define HAVE_NET_IF_DL_H 1

 85 /* Define se você tem a função poll. */
 86 #define HAVE_POLL 1

 87 /* Define se você tem o arquivo de cabeçalho <poll.h>. */
 88 #define HAVE_POLL_H 1

 89 /* Define se você tem a função pselect. */
 90 #define HAVE_PSELECT 1

 91 /* Define se o protótipo pselect está em <sys/stat.h> */
 92 #define HAVE_PSELECT_PROTO 1

 93 /* Define se você tem o arquivo de cabeçalho <pthread.h>. */
 94 #define HAVE_PTHREAD_H 1

 95 /* Define se você tem o arquivo de cabeçalho <signal.h>. */
 96 #define HAVE_SIGNAL_H 1

 97 /* Define se você tem a função snprintf. */
 98 #define HAVE_SNPRINTF 1

 99 /* Define se o protótipo snprintf está em <stdio.h> */
100 #define HAVE_SNPRINTF_PROTO 1

101 /* Define se <net/if_dl.h> define a estrutura sockaddr_dl */
102 #define HAVE_SOCKADDR_DL_STRUCT 1

103 /* define se as estruturas de endereço de soquete têm campos de comprimento */
104 #define HAVE_SOCKADDR_SA_LEN 1

105 /* Define se você tem a função sockatmark. */
106 #define HAVE_SOCKATMARK 1

107 /* define se o protótipo sockatmark está em <sys/socket.h> */
108 #define HAVE_SOCKATMARK_PROTO 1

109 /* Define se você tem o arquivo de cabeçalho <stdio.h>. */
110 #define HAVE_STDIO_H 1
```

Figura D.2 Nosso cabeçalho `config.h` para FreeBSD 5.1 (*continua*).

```
111 /* Define se você tem o arquivo de cabeçalho <stdlib.h>. */
112 #define HAVE_STDLIB_H 1

113 /* Define se você tem o arquivo de cabeçalho <strings.h>. */
114 #define HAVE_STRINGS_H 1

115 /* Define se você tem o arquivo de cabeçalho <string.h>. */
116 #define HAVE_STRING_H 1

117 /* Define se você tem o arquivo de cabeçalho <stropts.h>. */
118 /* #undef HAVE_STROPTS_H */

119 /* Define se ifr_mtu é membro da estrutura ifreq. */
120 #define HAVE_STRUCT_IFREQ_IFR_MTU 1

121 /* Define se o sistema tem a estrutura de tipo sockaddr_storage. */
122 #define HAVE_STRUCT_SOCKADDR_STORAGE 1

123 /* Define se você tem o arquivo de cabeçalho <sys/event.h>. */
124 #define HAVE_SYS_EVENT_H 1

125 /* Define se você tem o arquivo de cabeçalho <sys/filio.h>. */
126 #define HAVE_SYS_FILIO_H 1

127 /* Define se você tem o arquivo de cabeçalho <sys/ioctl.h>. */
128 #define HAVE_SYS_IOCTL_H 1

129 /* Define se você tem o arquivo de cabeçalho <sys/select.h>. */
130 #define HAVE_SYS_SELECT_H 1

131 /* Define se você tem o arquivo de cabeçalho <sys/socket.h>. */
132 #define HAVE_SYS_SOCKET_H 1

133 /* Define se você tem o arquivo de cabeçalho <sys/sockio.h>. */
134 #define HAVE_SYS_SOCKIO_H 1

135 /* Define se você tem o arquivo de cabeçalho <sys/stat.h>. */
136 #define HAVE_SYS_STAT_H 1

137 /* Define se você tem o arquivo de cabeçalho <sys/sysctl.h>. */
138 #define HAVE_SYS_SYSCTL_H 1

139 /* Define se você tem o arquivo de cabeçalho <sys/time.h>. */
140 #define HAVE_SYS_TIME_H 1

141 /* Define se você tem o arquivo de cabeçalho <sys/types.h>. */
142 #define HAVE_SYS_TYPES_H 1

143 /* Define se você tem o arquivo de cabeçalho <sys/uio.h>. */
144 #define HAVE_SYS_UIO_H 1

145 /* Define se você tem o arquivo de cabeçalho <sys/un.h>. */
146 #define HAVE_SYS_UN_H 1

147 /* Define se você tem o arquivo de cabeçalho <sys/wait.h>. */
148 #define HAVE_SYS_WAIT_H 1

149 /* Define se <time.h> define a estrutura timespec */
150 #define HAVE_TIMESPEC_STRUCT 1

151 /* Define se você tem o arquivo de cabeçalho <time.h>. */
152 #define HAVE_TIME_H 1

153 /* Define se você tem o arquivo de cabeçalho <unistd.h>. */
154 #define HAVE_UNISTD_H 1

155 /* Define se você tem a função vsnprintf. */
156 #define HAVE_VSNPRINTF 1

157 /* Define se você tem o arquivo de cabeçalho <xti.h>. */
158 /* #undef HAVE_XTI_H */
```

Figura D.2 Nosso cabeçalho `config.h` para FreeBSD 5.1 (*continua*).

```
159 /* Define se você tem o arquivo de cabeçalho <xti_inet.h>. */
160 /* #undef HAVE_XTI_INET_H */

161 /* Define se o sistema suporta IPv4 */
162 #define IPV4 1

163 /* Define se o sistema suporta IPv6 */
164 #define IPV6 1

165 /* Define se o sistema suporta IPv4 */
166 #define IPv4 1

167 /* Define se o sistema suporta IPv6 */
168 #define IPv6 1

169 /* Define se o sistema suporta IP Multicast */
170 #define MCAST 1

171 /* o tamanho do campo sa_family em uma estrutura de endereço de soquete */
172 /* #undef SA_FAMILY_T */

173 /* Define se você tem os arquivos de cabeçalho ANSI C. */
174 #define STDC_HEADERS 1

175 /* Define se você pode incluir seguramente <sys/time.h> e <time.h>. */
176 #define TIME_WITH_SYS_TIME 1

177 /* Define se o sistema suporta soquetes de domínio UNIX */
178 #define UNIXDOMAIN 1

179 /* Define se o sistema suporta soquetes de domínio UNIX */
180 #define UNIXdomain 1

181 /* tipo de 16 bits com sinal */
182 /* #undef int16_t */

183 /* tipo de 32 bits com sinal */
184 /* #undef int32_t */

185 /* o tipo do elemento da estrutura sa_family */
186 /* #undef sa_family_t */

187 /* tipo inteiro sem sinal do resultado do operador sizeof */
188 /* #undef size_t */

189 /* um tipo apropriado para endereço */
190 /* #undef socklen_t */

191 /* Define para __ss_family se sockaddr_storage tem isso em vez de ss_family */
192 /* #undef ss_family */

193 /* um tipo com sinal apropriado para uma contagem de bytes ou uma
    indicação de erro */
194 /* #undef ssize_t */

195 /* tipo escalar */
196 #define t_scalar_t int32_t

197 /* tipo escalar sem sinal */
198 #define t_uscalar_t uint32_t

199 /* tipo de 16 bits sem sinal */
200 /* #undef uint16_t */

201 /* tipo de 32 bits sem sinal */
202 /* #undef uint32_t */

203         /* tipo de 8 bits sem sinal */
204 /* #undef uint8_t */
```

sparc64-unknown-freebsd5.1/config.h

Figura D.2 Nosso cabeçalho `config.h` para FreeBSD 5.1 (*continuação*).

D.3 Funções de erro-padrão

Definimos nosso próprio conjunto de funções de erro que são utilizadas por todo o livro para tratar de condições de erro. A razão para utilizar nossas próprias funções de erro é para nos permitir escrever nosso tratamento de erro com uma única linha de código C, como em

```
if (condição de erro)
    err_sys(formato de printf com qualquer número de argumentos);
```

em vez de

```
if (condição de erro) {
    char buff[200];
    snprintf(buff, sizeof(buff), (formato de printf com qualquer número de argumentos);
    perror(buff);
    exit(1);
}
```

Nossas funções de erro utilizam o recurso de lista de argumentos de comprimento variável do C ANSI. Para detalhes adicionais, veja a Seção 7.3 de Kernighan e Ritchie (1988).

A Figura D.3 lista as diferenças entre as várias funções de erro. Se o inteiro global `daemon_proc` é não-zero, a mensagem é passada para `syslog` com o nível indicado; caso contrário, o erro é gerado na saída do erro-padrão.

Função	`strerror(errno)`?	Termina?	Nível de `syslog`
`err_dump`	sim	`abort();`	`LOG_ERR`
`err_msg`	não	`return;`	`LOG_INFO`
`err_quit`	não	`exit(1);`	`LOG_ERR`
`err_ret`	sim	`return;`	`LOG_INFO`
`err_sys`	sim	`exit(1);`	`LOG_ERR`

Figura D.3 Resumo de nossas funções de erro-padrão.

A Figura D.4 mostra as primeiras cinco funções da Figura D.3.

lib/error.c

```
 1 #include   "unp.h"
 2 #include   <stdarg.h>     /* arquivo de cabeçalho C ANSI */
 3 #include   <syslog.h>     /* para syslog() */
 4 int    daemon_proc;       /* configurado como não-zero por daemon_init() */
 5 static void err_doit(int, int, const char *, va_list);
 6 /* Erro não-fatal relacionado à chamada de sistema
 7  * Imprime mensagem e retorna */
 8 void
 9 err_ret(const char *fmt, ...)
10 {
11     va_list ap;
12     va_start(ap, fmt);
13     err_doit(1, LOG_INFO, fmt, ap);
14     va_end(ap);
15     return;
16 }
17 /* Erro fatal relacionado à chamada de sistema
```

Figura D.4 Nossas funções de erro-padrão (*continua*).

```
18  * Imprime mensagem e termina */
19 void
20 err_sys(const char *fmt, ...)
21 {
22     va_list ap;
23     va_start(ap, fmt);
24     err_doit(1, LOG_ERR, fmt, ap);
25     va_end(ap);
26     exit(1);
27 }
28 /* Erro fatal relacionado à chamada de sistema
29  * Imprime mensagem, descarrega memória e termina */
30 void
31 err_dump(const char *fmt, ...)
32 {
33     va_list ap;
34     va_start(ap, fmt);
35     err_doit(1, LOG_ERR, fmt, ap);
36     va_end(ap);
37     abort();                    /* descarrega memória e termina */
38     exit(1);                    /* não deve chegar aqui */
39 }
40 /* Erro não-fatal não relacionado à chamada de sistema
41  * Imprime mensagem e retorna */
42 void
43 err_msg(const char *fmt, ...)
44 {
45     va_list ap;
46     va_start(ap, fmt);
47     err_doit(0, LOG_INFO, fmt, ap);
48     va_end(ap);
49     return;
50 }
51 /* Erro fatal não relacionado à chamada de sistema
52  * Imprime mensagem e termina */
53 void
54 err_quit(const char *fmt, ...)
55 {
56     va_list ap;
57     va_start(ap, fmt);
58     err_doit(0, LOG_ERR, fmt, ap);
59     va_end(ap);
60     exit(1);
61 }
62 /* Imprime mensagem e retorna para o chamador
63  * O chamador especifica "errnoflag" e "level" */
64 static void
65 err_doit(int errnoflag, int level, const char *fmt, va_list ap)
66 {
67     int    errno_save, n;
68     char   buf[MAXLINE + 1];
69     errno_save = errno;         /* valor que o chamador talvez queira
                                   impresso */
```

Figura D.4 Nossas funções de erro-padrão (*continua*).

```
70 #ifdef HAVE_VSNPRINTF
71     vsnprintf(buf, MAXLINE, fmt, ap);    /* seguro */
72 #else
73     vsprintf(buf, fmt, ap);      /* não seguro */
74 #endif
75     n = strlen(buf);
76     if (errnoflag)
77         snprintf(buf + n, MAXLINE - n, ": %s", strerror(errno_save));
78     strcat(buf, "\n");
79     if (daemon_proc) {
80         syslog(level, buf);
81     } else {
82         fflush(stdout);          /* no caso de stdout e stderr serem os
                                    mesmos */
83         fputs(buf, stderr);
84         fflush(stderr);
85     }
86     return;
87 }
```

lib/error.c

Figura D.4 Nossas funções de erro-padrão (*continuação*).

Apêndice **E**

Soluções dos Exercícios Selecionados

Capítulo 1

1.3 No Solaris, obtemos o seguinte:

```
solaris % daytimetcpcli 127.0.0.1
socket error: Protocol not supported
```

Para encontrar informações adicionais sobre esse erro, primeiro utilizamos `grep` para procurar a string `Protocol not supported` no cabeçalho `<sys/errno.h>`.

```
solaris % grep 'Protocol not supported' /usr/include/sys/errno.h
#define EPROTONOSUPPORT 120    /* Protocolo não suportado */
```

Esse é o `errno` retornado por `socket`. Então, vemos a página man:

```
aix % man socket
```

A maioria das páginas man fornece informações adicionais, embora breves, no final, sob um título da forma "Erros".

1.4 Alteramos a primeira declaração para o seguinte:

```
int     sockfd, n, counter = 0;
```

Adicionamos a instrução

```
counter++;
```

como a primeira do loop `while`. Por fim, executamos

```
printf("counter = %d\n", counter);
```

antes de terminar. O valor impresso é sempre 1.

1.5 Declaramos um valor `int` chamado `i` e alteramos a chamada a `write` para o seguinte:

```
for (i = 0; i < strlen(buff); i++)
    Write(connfd, &buff[i], 1);
```

Os resultados variam, dependendo do host do cliente e do host do servidor. Se o cliente e o servidor estão no mesmo host, o contador normalmente é 1, o que significa que, mesmo que o servidor execute 26 comandos `write`, os dados são retornados por um único comando `read`. Mas uma combinação de cliente e servidor pode produzir dois pacotes e outra combinação, 26 pacotes. (Nossa discussão sobre o algoritmo de Nagle, na Seção 7.9, explica uma razão para isso.)

O propósito desse exemplo é reiterar que diferentes TCPs fazem coisas diferentes com os dados e nossa aplicação deve estar preparada para ler os dados como um fluxo de bytes até que o fim do fluxo de dados seja encontrado.

Capítulo 2

2.1 Visite o endereço `http://www.iana.org/numbers.htm` e localize o registro chamado "IP Version Number". A versão 0 é reservada, as versões 1 a 3 não foram atribuídas e a versão 5 é o Protocolo Internet de Fluxo.

2.2 Todas as RFCs estão disponíveis gratuitamente por meio de correio eletrônico, de FTP anônimo ou da Web. Um ponto de partida é `http://www.ietf.org`. No diretório `ftp://ftp.rfc-editor.org/in-notes` são encontradas RFCs. Para começar, busque o índice atual de RFCs, normalmente, o arquivo `rfc-index.txt`, também disponível em uma versão HTML no endereço `http://www.rfc-editor.org/rfc-index.html`. Se pesquisarmos o índice de RFCs (veja a solução do exercício anterior) com um editor de algum tipo, procurando o termo "Stream", veremos que a RFC 1819 define a Versão 2 do Protocolo Internet de Fluxo. Ao se procurar informações que podem ser abordadas por uma RFC, o índice de RFCs deve ser pesquisado.

2.3 Com IPv4, isso gera um datagrama de IP de 576 bytes (20 bytes para o cabeçalho IPv4 e 20 bytes para o cabeçalho TCP), o tamanho mínimo de buffer de montagem com IPv4.

2.4 Nesse exemplo, o servidor realiza o fechamento ativo, não o cliente.

2.5 O host no token ring não pode enviar pacotes com mais de 1.460 bytes de dados, porque os MSS que recebeu eram 1.460. O host na Ethernet pode enviar pacotes com até 4.096 bytes de dados, mas não excederá o MTU da interface de saída (a Ethernet) para evitar fragmentação. O TCP não pode exceder o MSS anunciado pela outra extremidade, mas sempre pode enviar menos que essa quantidade.

2.6 A seção "Numbers Protocol" da página Assigned Numbers Web (`http://www.iana.org/numbers.htm`) mostra o valor 89 para OSPF.

2.7 Um reconhecimento seletivo indica somente que os dados cobertos pelos números da seqüência refletidos na mensagem de reconhecimento seletiva foram recebidos. Somente um reconhecimento cumulativo diz que os dados até (e incluindo) o número de seqüência na mensagem de reconhecimento cumulativa foram recebidos. Ao liberar dados do buffer de envio com base em um reconhecimento seletivo, o sistema somente pode liberar os dados exatos que foram reconhecidos e nada antes nem depois do reconhecimento seletivo.

Capítulo 3

3.1 Em C, uma função não pode alterar o valor de um argumento que é passado por valor. Para uma função chamada modificar um valor passado pelo chamador é necessário que este passe um ponteiro para o valor a ser modificado.

3.2 O ponteiro deve ser incrementado pelo número de bytes lidos ou gravados, mas a linguagem C não permite que um ponteiro `void` seja incrementado (pois o compilador não conhece o tipo de dados apontado).

Capítulo 4

4.1 Veja as definições das constantes que começam com `INADDR_`, exceto `INADDR_ANY` (que tem todos os bits em zero) e `INADDR_NONE` (que tem todos os bits em um). Por exemplo, o endereço de multicast de classe D `INADDR_MAX_LOCAL_GROUP` é definido como `0xe00000ff`, com o comentário "224.0.0.255", que está claramente na ordem de byte de host.

4.2 Eis as novas linhas adicionadas após a chamada a `connect`:

```
len = sizeof(cliaddr);
Getsockname(sockfd, (SA *) &cliaddr, &len);
printf("local addr: %s\n",
       Sock_ntop((SA *) &cliaddr, len));
```

Isso exige uma declaração de `len` como um `socklen_t` e uma declaração de `cliaddr` como um `struct sockaddr_in`. Observe que o argumento de valor-resultado para `getsockname` (`len`) deve ser inicializado antes da chamada, com o tamanho da variável apontada pelo segundo argumento. O erro de programação mais comum com argumentos de valor-resultado é esquecer essa inicialização.

4.3 Quando o filho chama `close`, a contagem de referência é decrementada de 2 para 1; portanto, um FIN não é enviado para o cliente. Posteriormente, quando o pai chama `close`, a contagem de referência é decrementada para 0 e o FIN é enviado.

4.4 `accept` retorna `EINVAL`, pois o primeiro argumento não é um soquete ouvinte.

4.5 Sem uma chamada a `bind`, a chamada a `listen` atribui uma porta efêmera ao soquete ouvinte.

Capítulo 5

5.1 A duração do estado TIME_WAIT deve ser entre 1 e 4 minutos, fornecendo um MSL entre 30 segundos e 2 minutos.

5.2 Nossos programas cliente/servidor não funcionam com um arquivo binário. Suponha que os primeiros 3 bytes no arquivo sejam 1 binário, 0 binário e um caractere de nova linha. A chamada a `fgets`, na Figura 5.5, lê até `MAXLINE-1` caracteres ou até que um caractere de nova linha seja encontrado, ou até o EOF. Nesse exemplo, lerá os três primeiros caracteres e, então, terminará a string com um byte nulo. Mas nossa chamada a `strlen`, na Figura 5.5, retorna 1, pois pára no primeiro byte nulo. Um byte é enviado para o servidor, mas este bloqueia em sua chamada a `readline`, esperando por um caractere de nova linha. O cliente bloqueia, esperando pela resposta do servidor. Isso é chamado de *deadlock*: os dois processos estão bloqueados, cada um esperando do outro algo que nunca chegará. O problema aqui é que `fgets` finaliza os dados que ele retorna com um byte nulo; portanto, os dados que ele lê não podem conter quaisquer bytes nulos.

5.3 Telnet converte as linhas de entrada em NVT ASCII (Seção 26.4 de TCPv1), que termina cada linha com a seqüência de quebra de linha de dois caracteres: um CR (retorno de carro) seguido de um LF (avanço de linha). Nosso cliente adiciona somente uma nova linha, que é na verdade um caractere LF. Contudo, podemos utilizar o cliente de

Telnet para nos comunicarmos com nosso servidor, pois este ecoa de volta cada caractere, incluindo o CR que precede cada nova linha.

5.4 Não, os dois segmentos finais da seqüência de terminação de conexão não são enviados. Quando o cliente envia os dados para o servidor, depois que eliminamos o servidor-filho (a "outra linha"), o TCP do servidor responde com um RST. O RST cancela a conexão e também impede que o lado do servidor desta (que fez o fechamento ativo) passe pelo estado TIME_WAIT.

5.5 Nada muda, porque o processo servidor que é iniciado no host servidor cria um soquete ouvinte e está esperando que novas solicitações de conexão cheguem. O que enviamos no Passo 3 é um segmento de dados destinado a uma conexão TCP ESTABLISHED. Nosso servidor com o soquete ouvinte nunca vê esse segmento de dados e o TCP do servidor ainda responde a ele com um RST.

5.6 A Figura E.1 mostra o programa. Executar esse programa no Solaris gera o seguinte:

```
solaris % tsigpipe 192.168.1.10
SIGPIPE received
write error: Broken pipe
```

tcpcliserv/tsigpipe.c

```
 1 #include   "unp.h"
 2 void
 3 sig_pipe(int signo)
 4 {
 5     printf("SIGPIPE received\n");
 6     return;
 7 }
 8 int
 9 main(int argc, char **argv)
10 {
11     int    sockfd;
12     struct sockaddr_in servaddr;
13     if (argc != 2)
14         err_quit("usage: tcpcli <IPaddress>");
15     sockfd = Socket(AF_INET, SOCK_STREAM, 0);
16     bzero(&servaddr, sizeof(servaddr));
17     servaddr.sin_family = AF_INET;
18     servaddr.sin_port = htons(13);       /* servidor de data/hora */
19     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
20     Signal(SIGPIPE, sig_pipe);
21     Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
22     sleep(2);
23     Write(sockfd, "hello", 5);
24     sleep(2);
25     Write(sockfd, "world", 5);
26     exit(0);
27 }
```

tcpcliserv/tsigpipe.c

Figura E.1 Gera `SIGPIPE`.

O `sleep` inicial de dois segundos serve para permitir que o servidor de data/hora envie sua resposta e feche sua extremidade da conexão. Nosso primeiro comando `write` envia um segmento de dados para o servidor, o qual responde com um RST (pois o servidor de data/hora fechou completamente seu soquete). Observe que nosso TCP nos permite gravar em um soquete que recebeu um FIN. O segundo `sleep` permite que o RST do servidor seja recebido e nosso segundo comando `write` gera `SIGPIPE`. Como nosso handler de sinal retorna, `write` retorna o erro `EPIPE`.

5.7 Supondo que o host servidor suporte o *modelo de sistema extremidade fraca* (que descrevemos na Seção 8.8), tudo funciona. Isto é, o host servidor aceitará um datagrama de IP entrando (que contém um segmento TCP, nesse caso) que chega ao enlace de dados mais à esquerda, mesmo que o endereço IP de destino seja o endereço do enlace de dados mais à direita. Podemos testar isso executando nosso servidor em nosso host `linux` (Figura 1.16) e, então, iniciando o cliente em nosso host `solaris`, mas especificando o outro endereço IP do servidor (206.168.112.96) para o cliente. Depois que a conexão é estabelecida, se executamos `netstat` no servidor, vemos que o endereço IP local é o endereço IP de destino do SYN do cliente e não o endereço IP do enlace de dados no qual o SYN chegou (conforme mencionamos na Seção 4.4).

5.8 Nosso cliente estava em um sistema Intel little-endian, no qual o inteiro de 32 bits com um valor 1 foi armazenado, como mostrado na Figura E.2.

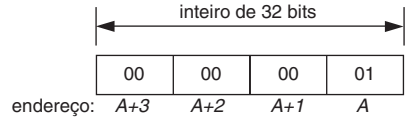


Figura E.2 Representação do inteiro de 32 bits 1 no formato little-endian.

Os 4 bytes são enviados através do soquete, na ordem *A*, *A + 1*, *A + 2* e *A + 3*, onde são armazenados no formato big-endian, como mostrado na Figura E.3.

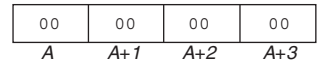


Figura E.3 Representação do inteiro de 32 bits da Figura E.2 no formato big-endian.

Esse valor de `0x01000000` é interpretado como 16.777.216. De maneira semelhante, o inteiro 2 enviado pelo cliente será interpretado no servidor como `0x02000000` ou 33.554.432. A soma desses dois inteiros é 50.331.648 ou `0x03000000`. Quando esse valor big-endian que está no servidor é enviado para o cliente, ele é interpretado neste como o valor inteiro 3.

O valor inteiro –22 de 32 bits é representado no sistema little-endian como mostrado na Figura E.4, supondo uma representação em complemento de dois para os números negativos.

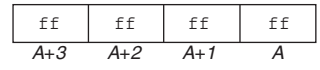


Figura E.4 Representação do inteiro de 32 bits –22 no formato little-endian.

Isso é interpretado no servidor big-endian como `0xeaffffff` ou –352.321.537. De maneira semelhante, a representação little-endian de –77 é `0xffffffb3`, mas isso é representado no servidor big-endian como `0xb3ffffff` ou –1.275.068.417. A adição no servidor gera o resultado binário `0x9effffe` ou –1.627.389.954. Esse valor big-endian é enviado através do soquete para o cliente, onde é interpretado como o valor little-endian `0xfeffff9e` ou –16.777.314, que é o valor impresso em nosso exemplo.

5.9 A técnica está correta (convertendo os valores binários para a ordem de byte de rede), mas as duas funções `htonl` e `ntohl` não podem ser utilizadas. Mesmo que nessas funções o valor `l` tenha significado "longo", elas operam sobre inteiros de 32 bits (Seção 3.4). Em um sistema de 64 bits, um valor `long` provavelmente ocupará 64 bits e essas duas funções não funcionarão corretamente. Alguém poderia definir duas novas funções, `hton64` e `ntoh64`, para resolver esse problema, mas isso não funcionará em sistemas que representam valores `long` utilizando 32 bits.

5.10 No primeiro cenário, o servidor bloqueia eternamente na chamada a `readn`, na Figura 5.20, porque o cliente envia dois valores de 32 bits, mas o servidor está esperando por dois valores de 64 bits.Trocar o cliente e o servidor entre os dois hosts faz com que o cliente envie dois valores de 64 bits, mas o servidor lê somente os primeiros 64 bits, interpretando-os como dois valores de 32 bits. O segundo valor de 64 bits permanece no buffer de recebimento do soquete do servidor. O servidor escreve de volta um valor de 32 bits e o cliente bloqueará eternamente em sua chamada a `readn`, na Figura 5.19, esperando para ler um valor de 64 bits.

5.11 A função de roteamento do IP vê o endereço IP de destino (o endereço IP do servidor) e pesquisa a tabela de roteamento para determinar a interface enviada e o próximo hop (Capítulo 9 do TCPv1). O endereço IP primário da interface enviada é utilizado como o endereço IP de origem, supondo que o soquete ainda não tenha vinculado um endereço IP local.

Capítulo 6

6.1 O array de inteiros está contido dentro de uma estrutura e a linguagem C permite que estruturas sejam atribuídas por meio de um sinal de igualdade.

6.2 Se `select` nos diz que o soquete é gravável, o buffer de envio do soquete tem espaço para 8.192 bytes, mas, quando chamamos `write` para esse soquete bloqueador, com um comprimento de buffer de 8.193 bytes, `write` pode bloquear, esperando por espaço para o byte final. As operações de leitura em um soquete bloqueador sempre retornarão uma contagem abreviada, se alguns dados estiverem disponíveis, mas as operações de gravação em um soquete bloqueador serão bloqueadas até que todos os dados possam ser aceitos pelo kernel. Portanto, ao utilizarmos `select` para testar a possibilidade de gravação, devemos configurar o soquete como não-bloqueador para evitar o bloqueio.

6.3 Se ambos os descritores são legíveis, somente o primeiro teste é realizado, o teste do soquete. Mas isso não interrompe o cliente; apenas o torna menos eficiente. Isto é, se `select` retorna com ambos os descritores legíveis, o primeiro `if` é verdadeiro, causando um `readline` do soquete, seguido de um `fputs`, na saída-padrão. O próximo `if` é pulado (por causa do `else` que prefixamos), mas `select` é então novamente chamado e imediatamente encontra a entrada-padrão legível e retorna imediatamente. O conceito-chave aqui é que o que esclarece a condição de "entrada-padrão estar legível" não é `select` retornando, mas a leitura do descritor.

6.4 Utilize a função `getrlimit` para buscar os valores para o recurso `RLIMIT_NOFILE` e, então, chame `setrlimit` para configurar o limite provisório corrente (`rlim_cur`) como o limite fixo (`rlim_max`). Por exemplo, no Solaris 2.5, o limite provisório é 64, mas qualquer processo pode aumentá-lo para o limite fixo default de 1.024.

`getrlimit` e `setrlimit` não fazem parte do POSIX.1, mas são exigidos pelo Unix 98.

6.5 A aplicação no servidor envia dados continuamente para o cliente, que o TCP deste reconhece e despreza.

6.6 `shutdown` com `SHUT_WR` ou `SHUT_RDWR` sempre envia um FIN, enquanto `close` envia um FIN somente se a contagem de referência do descritor for 1, quando `close` for chamado.

6.7 `read` retorna um erro e nossa função empacotadora `Read` termina o servidor. O servidor deve ser mais robusto que isso. Observe que tratamos dessa condição na Figura 6.26, embora mesmo esse código seja inadequado. Considere o que acontece se a conectividade é perdida entre o cliente e o servidor, e uma das respostas do servidor eventualmente tem seu tempo-limite esgotado. O erro retornado poderia ser `ETIMEDOUT`.

Em geral, um servidor não deve abortar para erros como esses. Ele deve registrar o erro, fechar o soquete e continuar a atender outros clientes. Entenda que tratar de um erro desse tipo, abortando, é inaceitável em um servidor como esse, no qual um processo está tratando de todos os clientes. Mas, se o servidor era um filho tratando somente de um cliente, então fazer um filho abortar não afetaria o pai (o qual supomos que trata de todas as novas conexões e gera os filhos), nem nenhum dos outros filhos que estão atendendo outros clientes.

Capítulo 7

7.2 A Figura E.5 mostra uma solução para este exercício. Removemos a impressão da string de dados retornada pelo servidor, pois esse valor não é necessário.

sockopt/rcvbuf.c

```
 1 #include   "unp.h"
 2 #include   <netinet/tcp.h>       /* para TCP_MAXSEG */

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int    sockfd, rcvbuf, mss;
 7     socklen_t len;
 8     struct sockaddr_in servaddr;

 9     if(argc != 2)
10        err_quit("usage: rcvbuf <IPaddress>");

11     sockfd = Socket(AF_INET, SOCK_STREAM, 0);

12     len = sizeof(rcvbuf);
13     Getsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &rcvbuf, &len);
14     len = sizeof(mss);
15     Getsockopt(sockfd, IPPROTO_TCP, TCP_MAXSEG, &mss, &len);
16     printf("defaults: SO_RCVBUF = %d, MSS = %d\n", rcvbuf, mss);

17     bzero(&servaddr, sizeof(servaddr));
18     servaddr.sin_family = AF_INET;
19     servaddr.sin_port = htons(13); /* servidor de data/hora */
20     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

21     Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

22     len = sizeof(rcvbuf);
```

Figura E.5 Imprime o tamanho do buffer de recebimento do soquete e do MSS, antes e depois do estabelecimento da conexão (*continua*).

```
23      Getsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &rcvbuf, &len);
24      len = sizeof(mss);
25      Getsockopt(sockfd, IPPROTO_TCP, TCP_MAXSEG, &mss, &len);
26      printf("after connect: SO_RCVBUF = %d, MSS = %d\n", rcvbuf, mss);
27      exit(0);
28  }
```

sockopt/rcvbuf.c

Figura E.5 Imprime o tamanho do buffer de recebimento do soquete e do MSS, antes e depois do estabelecimento da conexão (*continuação*).

Primeiro, não há uma saída "correta" desse programa. Os resultados variam de um sistema para outro. Alguns sistemas (notadamente o Solaris 2.5.1 e anteriores) sempre retornam 0 para os tamanhos de buffer de soquete, impedindo-nos de ver o que acontece com esse valor através da conexão.

Com relação ao MSS, o valor impresso antes de `connect` é o default da implementação (freqüentemente, 536 ou 512), enquanto o valor impresso depois de `connect` depender de uma possível opção de MSS do peer. Em uma Ethernet local, por exemplo, o valor depois de `connect` poderia ser 1.460. Após um `connect`, para um servidor em uma rede remota, entretanto, o MSS pode ser semelhante ao padrão, a menos que seu sistema suporte a descoberta do MTU do caminho. Se possível, execute uma ferramenta como `tcpdump` (Seção C.5), enquanto o programa estiver em execução, para ver a opção real de MSS no segmento SYN do peer.

Com relação ao tamanho do buffer de recebimento do soquete, muitas implementações arredondam esse valor para cima, depois que a conexão é estabelecida, para um múltiplo do MSS. Outra maneira de ver o tamanho do buffer de recebimento do soquete depois que a conexão é estabelecida é observar os pacotes utilizando uma ferramenta como `tcpdump` e examinar a janela anunciada do TCP.

7.3 Aloque uma estrutura `linger` chamada `ling` e inicialize-a como segue:

```
str_cli(stdin, sockfd);

ling.l_onoff = 1;
ling.l_linger = 0;
Setsockopt(sockfd, SOL_SOCKET, SO_LINGER, &ling, sizeof(ling));

exit(0);
```

Isso deve fazer o TCP do cliente terminar a conexão com um RST, em vez da troca de quatro segmentos normal. A chamada do servidor-filho a `readline` retorna um erro `ECONNRESET` e a mensagem impressa é a seguinte:

```
readline error: Connection reset by peer
```

O soquete cliente não deve passar pelo estado TIME_WAIT, mesmo que o cliente tenha realizado o fechamento ativo.

7.4 O primeiro cliente chama `setsockopt`, `bind` e `connect`. Mas, entre as primeiras chamadas do cliente a `bind` e `connect`, se o segundo cliente chama `bind`, `EADDRINUSE` é retornado. Porém, assim que o primeiro cliente se conecta ao peer, o `bind` do segundo cliente funcionará, pois o soquete do primeiro é então conectado. A única maneira de tratar disso é fazer o segundo cliente tentar chamar `bind` várias vezes, caso `EADDRINUSE` seja retornado, e não desistir na primeira vez que o erro for retornado.

7.5 Executamos o programa em um host com suporte a multicast (MacOS X 10.2.6).

```
macosx % sock -s 9999 &                    inicia o primeiro servidor com curinga
[1]    29697
macosx % sock -s 172.24.37.78 9999         tenta o segundo servidor, mas sem –A
can't bind local address: Address already in use
macosx % sock -s -A 172.24.37.78 9999 &    tenta de novo com –A; funciona
[2]    29699
macosx % sock -s -A 127.0.0.1 9999 &       terceiro servidor com –A; funciona
[3]    29700
macosx % netstat -na | grep 9999
tcp4       0      0  127.0.0.1.9999         *.*     LISTEN
tcp4       0      0  172.24.37.78.9999     *.*     LISTEN
tcp4       0      0  *.9999                 *.*     LISTEN
```

7.6 Primeiro, tentamos em um host que suporta multicast, mas não a opção SO_REUSEPORT (Solaris 9).

```
solaris % sock -s -u 8888 &                o primeiro inicia
[1]    24051
solaris % sock -s -u 8888
can't bind local address: Address already in use
solaris % sock -s -u -A 8888 &             tenta o segundo novamente, com –A;
                                           funciona
solaris % netstat -na | grep 8888          podemos ver as vinculações duplicadas
       *.8888                     Idle
       *.8888                     Idle
```

Nesse sistema, não precisamos especificar SO_REUSEADDR para a primeira vinculação, somente para a segunda.

Por fim, executamos esse cenário sob o MacOS X 10.2.6, que suporta multicast e a opção SO_REUSEPORT. Primeiro, tentamos SO_REUSEADDR para ambos os servidores, mas isso não funciona.

```
macosx % sock -u -s -A 7777 &
[1]    17610
macosx % sock -u -s -A 7777
can't bind local address: Address already in use
```

Em seguida, tentamos SO_REUSEPORT, mas somente para o segundo servidor, não para o primeiro. Isso não funciona, pois uma vinculação completamente duplicada exige a opção para todos os soquetes que a compartilham.

```
macosx % sock -u -s 8888 &
[1]    17612
macosx % sock -u -s -T 8888
can't bind local address: Address already in use
```

Por fim, especificamos SO_REUSEPORT para ambos os servidores, e isso funciona.

```
macosx % sock -u -s -T 9999 &
[1]    17614
macosx % sock -u -s -T 9999 &
[2]    17615
macosx % netstat -na | grep 9999
udp4       0      0  *.9999                 *.*
udp4       0      0  *.9999                 *.*
```

7.7 Isso não faz nada, porque ping utiliza um soquete ICMP e a opção de soquete SO_DEBUG afeta somente soquetes TCP. A descrição da opção de soquete SO_DEBUG sempre foi algo genérico, como "essa opção permite depurar na camada de protocolo respectiva", mas a única camada de protocolo a implementar a opção tem sido TCP.

7.8 A Figura E.6 mostra a linha do tempo.

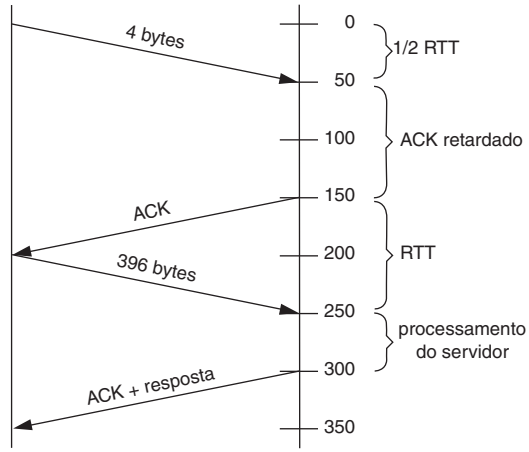


Figura E.6 Interação do algoritmo de Nagle com ACK retardado.

7.9 Configurar a opção de soquete `TCP_NODELAY` faz com que os dados do segundo comando `write` sejam enviados imediatamente, mesmo que a conexão tenha um pacote pequeno pendente. Mostramos isso na Figura E.7. O tempo total nesse exemplo é pouco mais de 150 ms.

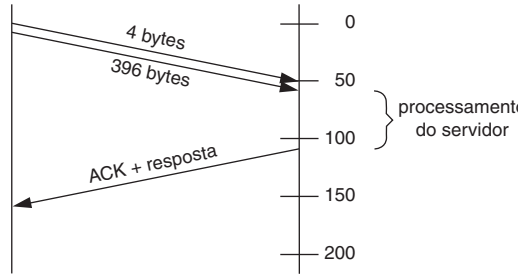


Figura E.7 Evitando o algoritmo de Nagle por meio da configuração da opção de soquete `TCP_NODELAY`.

7.10 A vantagem dessa solução é a redução do número de pacotes, como mostramos na Figura E.8.

7.11 A Seção 4.2.3.2 declara: "O retardo DEVE ser menor que 0,5 segundo e, em um fluxo de segmentos de tamanho completo, DEVE haver um ACK no mínimo para cada segundo segmento". As implementações derivadas do Berkeley retardam um ACK no máximo por 200 ms (página 821 do TCPv2).

7.12 O servidor-pai, na Figura 5.2, gasta a maior parte de seu tempo bloqueado na chamada a `accept` e o filho, na Figura 5.3, gasta a maior parte de seu tempo bloqueado na chamada a `read`, que é chamada por `readline`. A opção para manter ativo não tem nenhum efeito sobre um soquete ouvinte; portanto, o pai não é afetado caso o host cliente sofra colapso. A instrução `read` do filho retornará um erro `ETIMEDOUT`, às vezes cerca de duas horas após a última troca de dados através da conexão.

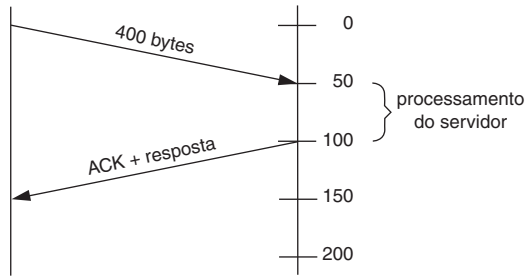


Figura E.8 Utilizando `writev` em vez de configurar a opção de soquete `TCP_NODELAY`.

7.13 O cliente da Figura 5.5 gasta a maior parte de seu tempo bloqueado na chamada a `fgets`, que por sua vez é bloqueada em algum tipo de operação de leitura na entrada-padrão, dentro da biblioteca de E/S-padrão. Quando o timer de manter-vivo (*keep-alive*) expira, cerca de duas horas após a última troca de dados através da conexão, e todas as sondagens de manter-vivo não conseguem extrair uma resposta do servidor, o erro pendente do soquete é configurado como `ETIMEDOUT`. Mas o cliente é bloqueado na chamada a `fgets` na entrada-padrão e não verá esse erro até realizar uma leitura ou uma gravação no soquete. Essa é uma razão pela qual modificamos a Figura 5.5 para utilizar select, no Capítulo 6.

7.14 Esse cliente gasta a maior parte de seu tempo bloqueado na chamada a `select`, que retornará o soquete como legível assim que o erro pendente for configurado como `ETIMEDOUT` (como descrevemos na solução anterior).

7.15 Somente dois segmentos são trocados, e não quatro. Há uma probabilidade muito baixa de que os dois sistemas tenham timers que estejam exatamente sincronizados; daí, o timer de manter-vivo de uma extremidade expirará pouco antes do outro. O primeiro a expirar envia a sondagem de manter-vivo, fazendo a outra extremidade reconhecer (ACK) essa prova. Mas o recebimento da sondagem de manter-vivo faz o timer de manter-vivo no host com o clock (ligeiramente) mais lento ser reinicializado para duas horas no futuro.

7.16 A API de soquetes original não tinha uma função `listen`. Em vez disso, o quarto argumento de `socket` continha opções de soquete e `SO_ACCEPTCON` era utilizado para especificar um soquete ouvinte. Quando `listen` foi adicionada, o flag permaneceu, mas agora ele é configurado somente pelo kernel (página 456 do TCPv2).

Capítulo 8

8.1 Sim, `read` retorna 4.096 bytes de dados, mas `recvfrom` retorna 2.048 (o primeiro dos dois datagramas). Uma `recvfrom` em um soquete de datagrama nunca retorna mais de um datagrama, independentemente de quantos a aplicação solicite.

8.2 Se o protocolo utiliza estruturas de endereço de soquete de comprimento variável, `clilen` poderia ser grande demais. Veremos, no Capítulo 15, que isso é aceitável no caso de estruturas de endereço de soquete do domínio Unix, mas a maneira correta de codificar a função é utilizar o comprimento real retornado por `recvfrom` como o comprimento de `sendto`.

8.4 Executar `ping` dessa forma é uma maneira fácil de ver mensagens de ICMP recebidas pelo host no qual `ping` está sendo executado. Reduzimos o número de pacotes enviados, do normal de um por segundo, apenas para reduzir a saída. Se executamos nosso

cliente UDP em nosso host `aix`, especificando o endereço IP do servidor como 192.168.42.1, e também executamos o programa `ping`, obtemos a seguinte saída:

```
aix % ping -v -i 60 127.0.0.1
PING 127.0.0.1: (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=255 time=0 ms
36 bytes from 192.168.42.1: Destination Port Unreachable
Vr HL TOS  Len   ID Flg  off TTL Pro  cks      Src      Dst Data
 4  5  00 0022 0007   0 0000  1e  11 c770 192.168.42.2  192.168.42.1
UDP: from port 40645, to port 9877 (decimal)
```

Observe que nem todos os clientes de `ping` imprimem os erros de ICMP recebidos, mesmo com o flag –v.

8.5 Ele provavelmente tem um tamanho de buffer de recebimento do soquete, mas os dados nunca são aceitos para um soquete TCP ouvinte. A maioria das implementações não aloca previamente memória para buffers de envio nem para buffers de recebimento do soquete. Os tamanhos de buffer de soquete especificados com as opções de soquete `SO_SNDBUF` e `SO_RCVBUF` são apenas limites superiores para esse soquete.

8.6 Executamos o programa `sock` no host multihomed `freebsd`, especificando as opções –`u` (use UDP) e –`l` (especificando o endereço IP local e a porta).

```
freebsd % sock -u -l 12.106.32.254.4444 192.168.42.2 8888
hello
```

O endereço IP local é a interface no lado da Internet na Figura 1.16, mas o datagrama deve sair para a outra interface para chegar ao destino. Observar a rede com `tcpdump` mostra que o endereço IP de origem é aquele que foi vinculado pelo cliente e não o endereço de interface de saída.

```
14:28:29.614846 12.106.32.254.4444 > 192.168.42.2.8888: udp 6
14:28:29.615225 192.168.42.2 > 12.106.32.254: icmp: 192.168.42.2
                                                udp port 8888 unreachable
```

8.7 Colocar um `printf` no cliente deve introduzir um retardo entre cada datagrama, permitindo que o servidor receba mais datagramas. Colocar um `printf` no servidor deve fazer com que ele perca mais datagramas.

8.8 O maior datagrama IPv4 tem 65.535 bytes, limitados pelo campo de comprimento total de 16 bits, na Figura A.1. O cabeçalho IP exige 20 bytes e o cabeçalho UDP exige 8 bytes, deixando um máximo de 65.507 bytes para dados de usuário. Com IPv6 sem suporte para jumbograma, o tamanho do cabeçalho IPv6 é de 40 bytes, deixando um máximo de 65.487 bytes para dados de usuário.

A Figura E.9 mostra a nova versão de `dg_cli`. Se você se esquecer de configurar o tamanho do buffer de envio, os kernels derivados do Berkeley retornarão o erro `EMSGSIZE` de `sendto`, pois o tamanho do buffer de envio do soquete normalmente é menor que o exigido para um datagrama de UDP de tamanho máximo (certifique-se de fazer o Exercício 7.1). Mas, se configurarmos os tamanhos de buffer de soquete do cliente como mostrado na Figura E.9 e executarmos o programa cliente, nada será retornado pelo servidor. Podemos verificar se o datagrama do cliente é enviado para o servidor, executando `tcpdump`, mas, se colocarmos um `printf` no servidor, sua chamada a `recvfrom` não retornará o datagrama. O problema é que o buffer de recebimento do soquete de UDP do servidor é menor que o datagrama que estamos enviando; portanto, o datagrama é descartado e não é entregue para o soquete. Em um sistema FreeBSD, podemos verificar isso executando `netstat -s` e examinando o contador "derrubado devido a buffers de soquete cheios", antes e depois que nosso datagrama grande é recebido. A solução final é modificar o servidor, configurando os tamanhos de buffer de envio e recebimento de seu soquete.

udpcliserv/dgclibig.c

```
 1 #include   "unp.h"
 2 #undef  MAXLINE
 3 #define MAXLINE 65507
 4 void
 5 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 6 {
 7     int    size;
 8     char   sendline[MAXLINE], recvline[MAXLINE + 1];
 9     ssize_t n;
10     size = 70000;
11     Setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, &size, sizeof(size));
12     Setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));
13     Sendto(sockfd, sendline, MAXLINE, 0, pservaddr, servlen);
14     n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
15     printf("received %d bytes\n", n);
16 }
```

udpcliserv/dgclibig.c

Figura E.9 Gravando o datagrama UDP/IPv4 de tamanho máximo.

Na maioria das redes, um datagrama de IP de 65.535 bytes é fragmentado. Lembre-se, da Seção 2.11, de que uma camada de IP deve suportar um tamanho de buffer de montagem de somente 576 bytes. Portanto, você pode encontrar hosts que não receberão os datagramas de tamanho máximo enviados neste exercício. Além disso, muitas implementações derivadas do Berkeley, incluindo a 4.4BSD-Lite2, têm um bug de sinal que impede que o UDP aceite um datagrama maior que 32.767 bytes (linha 95 da página 770 do TCPv2).

Capítulo 9

9.1 Há diversas situações em que `sctp_peeloff` pode desempenhar um papel importante. Um exemplo de aplicação que poderia utilizar essa função é um servidor tipo UDP tradicional que responde às solicitações como pequenas transações, mas ocasionalmente é solicitado a fazer uma auditoria de longo prazo. Na maioria dos casos, você só precisa enviar uma ou duas mensagens pequenas e nada mais; mas, quando chega um pedido de auditoria, uma conversação de longo prazo é ativada, enviando informações de auditoria. Nessa situação, você colocaria a auditoria em seu próprio thread ou processo para fazê-la.

Em resumo, qualquer aplicação que tenha principalmente solicitações pequenas, mas ocasionalmente uma conversação de longo prazo, pode tirar proveito de `sctp_peeloff`.

9.2 O lado do servidor é fechado automaticamente, quando o cliente fecha a associação. Isso porque o SCTP não suporta o estado meio fechado; portanto, quando o cliente chamar o fechamento, a seqüência de desligamento descarregará todos os dados pendentes que o servidor tiver enfileirado para o cliente e completará o desligamento, fechando a associação.

9.3 No estilo de um para um, uma chamada de conexão deve ser realizada primeiro e, então, quando o COOKIE for enviado para o peer, nenhum dado estará pendente. Para o estilo um para muitos, uma aplicação pode enviar dados para um peer, para configurar uma associação. Isso significa que, quando o COOKIE é enviado, os dados (DATA) estão disponíveis para serem enviados ao peer.

9.4 A única situação em que um peer, com o qual você estaria configurando uma associação, poderia enviar dados de volta, seria se ele tivesse dados (DATA) pendentes ANTES de a conexão ter sido estabelecida. Isso ocorreria se cada lado estivesse utilizando o estilo um para muitos e tivesse feito um envio para configurar implicitamente a associação. Esse tipo de configuração de associação é chamado de colisão de INIT e os detalhes sobre ele podem ser encontrados no Capítulo 4 de Stewart e Xie (2001).

9.5 Em alguns casos, nem todos os endereços que são vinculados podem ser passados para um ponto final de peer. Em particular, quando os endereços que uma aplicação tiver vinculado contiverem endereços IP privados e públicos, somente os endereços públicos poderão ser compartilhados com um ponto final de peer. Outro exemplo é encontrado no IPv6, onde os endereços de enlace locais não precisam necessariamente ser compartilhados com um peer.

Capítulo 10

10.1 Se a função `sctp_sendmsg` retornasse um erro, nenhuma mensagem seria enviada e a aplicação então executaria um `sctp_recvmsg` bloqueador, esperando por uma mensagem de resposta que nunca seria enviada a ela.

Uma maneira de corrigir isso é verificar os códigos de retorno de erro e, se um erro ocorrer no envio, o cliente NÃO deve realizar o recebimento, mas, em vez disso, deve informar um erro.

Se a função `sctp_recvmsg` retornar um erro, nenhuma mensagem chegará e o servidor ainda tentará enviar uma mensagem, possivelmente configurando uma associação. Para evitar isso, o código de erro deve ser verificado e, dependendo do erro, talvez você queira relatá-lo e fechar o soquete, permitindo então que o servidor também receba um erro; ou, se o erro for transitório, poderia tentar novamente a chamada de `sctp_recvmsg`.

10.2 Se o servidor receber uma solicitação e então sair, em sua forma atual, o cliente travará para sempre, esperando por uma mensagem que nunca virá. Um método que pode ser utilizado pelo cliente para detectar isso é ativar eventos de associação. Isso permitirá que a aplicação cliente receba uma mensagem quando o servidor sair, dizendo ao cliente que a associação agora está terminada. Isso permitiria ao cliente então executar uma ação de recuperação, como entrar em contato com um servidor diferente.

Um método alternativo que o cliente poderia utilizar é configurar um timer e abortar após algum período de tempo.

10.3 Escolhemos 800 bytes para tentar obter cada bloco em um único pacote. Uma maneira melhor seria obter ou configurar a opção de soquete `SCTP_MAXSEG` para determinar a quantidade de bytes que caberá em um bloco.

10.4 O algoritmo de Nagle (controlado pela opção de soquete `SCTP_NODELAY`; veja a Seção 7.10) só causará um problema se escolhermos um tamanho pequeno de transferência de dados. Assim, contanto que enviemos um tamanho que obrigue o SCTP a enviar imediatamente, nenhum dano ocorrerá. Entretanto, escolher um tamanho menor para `out_sz` distorceria os resultados, mantendo algumas transmissões esperando SACKs do ponto final remoto. Então, se um tamanho menor deve ser utilizado, desativar o algoritmo de Nagle (isto é, ativar a opção de soquete `SCTP_NODELAY`) seria uma boa idéia.

10.5 Se uma aplicação configurar uma associação e então alterar o número de fluxos, a associação não terá um novo número de fluxos, mas o número original, antes da alteração. Isso porque alterar o número de fluxos só afeta as novas associações, não as já existentes.

10.6 O estilo um para muitos permite a configuração implícita das associações. Para utilizar dados auxiliares a fim de alterar a configuração de uma associação, você precisa usar a chamada `sendmsg`, para fornecer os dados antes que a associação seja configurada. Portanto, você deve utilizar uma configuração de associação implícita.

Capítulo 11

11.1 A Figura E.10 mostra nosso programa que chama `gethostbyaddr`. Esse programa funciona bem para um host com um único endereço IP. Se executarmos o programa da Figura 11.3 em um host com quatro endereços IP, obteremos o seguinte:

```
freebsd % hostent cnn.com
official hostname: cnn.com
        address: 64.236.16.20
        address: 64.236.16.52
        address: 64.236.16.84
        address: 64.236.16.116
        address: 64.236.24.4
        address: 64.236.24.12
        address: 64.236.24.20
        address: 64.236.24.28
```

Mas, se executarmos o programa da Figura E.10 para o mesmo host, somente o primeiro endereço IP aparecerá na saída, como segue:

```
freebsd % hostent2 cnn.com
official hostname: cnn.com
          address: 64.236.24.4
          name = www1.cnn.com
```

O problema é que as duas funções, `gethostbyname` e `gethostbyaddr`, compartilham a mesma estrutura `hostent`, como mostramos no começo da Seção 11.18. Quando nosso novo programa chama `gethostbyaddr`, ele reutiliza essa estrutura, junto com o armazenamento para o qual ela aponta (isto é, o array de ponteiros `h_addr_list`), apagando os três endereços IP restantes retornados por `gethostbyname`.

names/hostent2.c

```
 1 #include   "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     char   *ptr, **pptr;
 6     char   str[INET6_ADDRSTRLEN];
 7     struct hostent *hptr;

 8     while (--argc > 0) {
 9         ptr = *++argv;
10         if ( (hptr = gethostbyname(ptr)) == NULL) {
11             err_msg("gethostbyname error for host: %s: %s",
12                     ptr, hstrerror(h_errno));
13             continue;
14         }
15         printf("official hostname: %s\n", hptr->h_name);

16         for (pptr = hptr->h_aliases; *pptr != NULL; pptr++)
17             printf("  alias: %s\n", *pptr);

18         switch (hptr->h_addrtype) {
19         case AF_INET:
20 #ifdef AF_INET6
```

Figura E.10 Modificação da Figura 11.3 para chamar `gethostbyaddr` (*continua*).

```
21        case AF_INET6:
22 #endif
23            pptr = hptr->h_addr_list;
24            for ( ; *pptr != NULL; pptr++) {
25                printf("\taddress: %s\n",
26                       Inet_ntop(hptr->h_addrtype, *pptr, str, sizeof(str)));
27                if ( (hptr = gethostbyaddr(*pptr, hptr->h_length,
28                                           hptr->h_addrtype))== NULL)
29                    printf("\t(gethostbyaddr failed)\n");
30                else if (hptr->h_name != NULL)
31                    printf("\tname = %s\n", hptr->h_name);
32                else
33                    printf("\t(no hostname returned by gethostbyaddr)\n");
34            }
35            break;
36        default:
37            err_ret("unknown address type");
38            break;
39        }
40    }
41    exit(0);
42 }
```

names/hostent2.c

Figura E.10 Modificação da Figura 11.3 para chamar `gethostbyaddr` (*continuação*).

11.2 Se seu sistema não fornece a versão reentrante de `gethostbyaddr` (que descrevemos na Seção 11.19), então você deve fazer uma cópia do array de ponteiros retornado por `gethostbyname`, junto com os dados apontados por esse array, antes de chamar `gethostbyaddr`.

11.3 O servidor `chargen` envia dados para o cliente até que este feche a conexão (isto é, até que você aborte o cliente).

11.4 Esse é um recurso de alguns resolvedores, mas você não pode contar com ele em um programa portável, porque o POSIX deixa o comportamento não especificado. A Figura E.11 mostra a versão modificada. A ordem dos testes na string de nome de host é importante. Chamamos `inet_pton` primeiro, pois se trata de um teste rápido na memória, para saber se a string é um endereço IP decimal com pontos válido ou não. Somente se isso falhar é que chamamos `gethostbyname`, que em geral envolve alguns recursos de rede e algum tempo.

names/daytimetcpcli2.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    sockfd, n;
 6     char   recvline[MAXLINE + 1];
 7     struct sockaddr_in servaddr;
 8     struct in_addr **pptr, *addrs[2];
 9     struct hostent *hp;
10     struct servent *sp;
11     if (argc != 3)
```

Figura E.11 Permite endereço IP decimal com pontos ou nome de host, número de porta ou nome de serviço (*continua*).

```
12          err_quit("usage: daytimetcpcli2 <hostname> <service>");

13      bzero(&servaddr, sizeof(servaddr));
14      servaddr.sin_family = AF_INET;

15      if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) == 1) {
16          addrs[0] = &servaddr.sin_addr;
17          addrs[1] = NULL;
18          pptr = &addrs[0];
19      } else if ((hp = gethostbyname(argv[1])) != NULL) {
20          pptr = (struct in_addr **) hp->h_addr_list;
21      } else
22          err_quit("hostname error for %s: %s", argv[1], hstrerror(h_errno));

23      if ( (n = atoi(argv[2])) > 0)
24          servaddr.sin_port = htons(n);
25      else if ((sp = getservbyname(argv[2], "tcp")) != NULL)
26          servaddr.sin_port = sp->s_port;
27      else
28          err_quit("getservbyname error for %s", argv[2]);

29      for ( ; *pptr != NULL; pptr++) {
30          sockfd = Socket(AF_INET, SOCK_STREAM, 0);

31          memmove(&servaddr.sin_addr, *pptr, sizeof(struct in_addr));
32          printf("trying %s\n", Sock_ntop((SA *) &servaddr, sizeof(servaddr)));

33          if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) == 0)
34              break;                  /* sucesso */
35          err_ret("connect error");
36          close(sockfd);
37      }
38      if (*pptr == NULL)
39          err_quit("unable to connect");

40      while ( (n = Read(sockfd, recvline, MAXLINE)) > 0) {
41          recvline[n] = 0;            /* termina com nulo */
42          Fputs(recvline, stdout);
43      }
44      exit(0);
45  }
```

names/daytimetcpcli2.c

Figura E.11 Permite endereço IP decimal com pontos ou nome de host, número de porta ou nome de serviço (*continuação*).

Se a string é um endereço IP decimal com pontos válido, fazemos nosso próprio array de ponteiros (`addrs`) para o único endereço IP, permitindo que o loop que utiliza `pptr` permaneça o mesmo.

Como o endereço já foi convertido para binário na estrutura de endereço de soquete, alteramos a chamada para `memcpy` na Figura 11.4, para chamar `memmove`, pois, quando um endereço IP decimal com pontos é introduzido, os campos de origem e destino são os mesmos nessa chamada.

11.5 A Figura E.12 mostra o programa.

Utilizamos o valor de `h_addrtype` retornado por `gethostbyname` para determinar o tipo de endereço. Também utilizamos nossas funções `sock_set_port` e `sock_set_addr` (Seção 3.8) para configurar esses dois campos na estrutura de endereço de soquete apropriada.

names/daytimetcpcli3.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int    sockfd, n;
 6     char   recvline[MAXLINE + 1];
 7     struct sockaddr_in servaddr;
 8     struct sockaddr_in6 servaddr6;
 9     struct sockaddr *sa;
10     socklen_t salen;
11     struct in_addr **pptr;
12     struct hostent *hp;
13     struct servent *sp;
14     if (argc != 3)
15         err_quit("usage: daytimetcpcli3 <hostname> <service>");
16     if ( (hp = gethostbyname(argv[1])) == NULL)
17         err_quit("hostname error for %s: %s", argv[1], hstrerror(h_errno));
18     if ( (sp = getservbyname(argv[2], "tcp")) == NULL)
19         err_quit("getservbyname error for %s", argv[2]);
20     pptr = (struct in_addr **) hp->h_addr_list;
21     for ( ; *pptr != NULL; pptr++) {
22         sockfd = Socket(hp->h_addrtype, SOCK_STREAM, 0);
23         if (hp->h_addrtype == AF_INET) {
24            sa = (SA *) &servaddr;
25            salen = sizeof(servaddr);
26         } else if (hp->h_addrtype == AF_INET6) {
27            sa = (SA *) &servaddr6;
28            salen = sizeof(servaddr6);
29         } else
30            err_quit("unknown addrtype %d", hp->h_addrtype);
31         bzero(sa, salen);
32         sa->sa_family = hp->h_addrtype;
33         sock_set_port(sa, salen, sp->s_port);
34         sock_set_addr(sa, salen, *pptr);
35         printf("trying %s\n", Sock_ntop(sa, salen));
36         if (connect(sockfd, sa, salen) == 0)
37            break;                /* sucesso */
38         err_ret("connect error");
39         close(sockfd);
40     }
41     if  (*pptr == NULL)
42         err_quit("unable to connect");
43     while ( (n = Read(sockfd, recvline, MAXLINE)) > 0) {
44         recvline[n] = 0;         /* termina com nulo */
45         Fputs(recvline, stdout);
46     }
47     exit(0);
48 }
```

names/daytimetcpcli3.c

Figura E.12 Modificação da Figura 11.4 para trabalhar com IPv4 e IPv6.

Embora esse programa funcione, ele tem duas limitações. Primeira: devemos tratar de todas as diferenças, examinando `h_addrtype` e, então, configurando `sa` e `salen` apropriadamente. Uma solução melhor é ter uma função de biblioteca que não apenas

pesquise o nome de host e o nome de serviço, mas também preencha a estrutura de endereço de soquete inteira (por exemplo, `getaddrinfo`, na Seção 11.6). Segunda: esse programa compila somente em hosts que suportam IPv6. Para fazê-lo compilar em um host somente IPv4, seria necessário adicionar numerosas instruções `#ifdef` no código, complicando-o.

Retornamos para o conceito de independência de protocolo do Capítulo 11 e vemos maneiras melhores de fazer isso.

11.7 Aloque um buffer grande (maior que qualquer estrutura de endereço de soquete) e chame `getsockname`. O terceiro argumento é um argumento de valor-resultado que retorna o tamanho real dos endereços do protocolo. Infelizmente, isso funciona somente para protocolos com estruturas de endereço de soquete de largura fixa (por exemplo, IPv4 e IPv6), mas não é garantido que funcione com protocolos que podem retornar estruturas de endereço de soquete de comprimento variável (por exemplo, soquetes de domínio Unix, Capítulo 15).

11.8 Primeiro, alocamos arrays para conter o nome de host e o nome de serviço, como segue:

```
char   host[NI_MAXHOST], serv[NI_MAXSERV];
```

Depois que `accept` retorna, chamamos `getnameinfo`, em vez de `sock_ntop`, como segue:

```
if (getnameinfo(cliaddr, len, host, NI_MAXHOST, serv, NI_MAXSERV,
                NI_NUMERICHOST | NI_NUMERICSERV) == 0)
   printf("connection from %s.%s\n", host, serv);
```

Como esse é um servidor, especificamos os flags `NI_NUMERICHOST` e `NI_NUMERICSERV` para evitar uma consulta de DNS e uma pesquisa de `/etc/services`.

11.9 O primeiro problema é que o segundo servidor não pode vincular (`bind`) a mesma porta que o primeiro, pois a opção de soquete `SO_REUSEADDR` não está configurada. A maneira mais fácil de tratar disso é fazer uma cópia da função `udp_server`, chamá-la de `udp_server_reuseaddr`, fazê-la configurar a opção de soquete e chamar essa nova função a partir do servidor.

11.10 Quando o cliente gera a saída "Tentando 206.62.226.35...", `gethostbyname` já retornou o endereço IP. Qualquer pausa de cliente antes disso é o tempo exigido pelo resolvedor para pesquisar o nome de host. A saída "Conectado a bsdi.kohala.com." significa que `connect` retornou. Qualquer pausa entre essas duas linhas de saída é o tempo exigido por `connect` para estabelecer a conexão.

Capítulo 12

12.1 Eis as exceções relevantes (por exemplo, com as listagens de login e diretório omitidas). Observe que o cliente FTP no sistema `freebsd` *sempre* tenta o comando `EPRT`, esteja utilizando IPv4 ou IPv6, e recorre ao comando `PORT` quando isso não funciona.

```
freebsd % ftp aix-4
Connected to aix-4.unpbook.com.
220 aix FTP server ...
...
230 Guest login ok, access restrictions apply.
ftp> debug
Debugging on (debug=1).
ftp> passive
Passive mode: off; fallback to active mode: off.
ftp> dir
---> EPRT |1|192.168.42.1|50484|
```

```
500 'EPRT |1|192.168.42.1|50484|': command not understood.
disabling epsv4 for this connection
---> PORT 192,168,42,1,197,52
200 PORT command successful.
---> LIST
150 Opening ASCII mode data connection for /bin/ls.
...
freebsd % ftp ftp.kame.net
Trying 2001:200:0:4819:203:47ff:fea5:3085...
Connected to orange.kame.net.
220 orange.kame.net FTP server ...
...
230 Guest login ok, access restrictions apply.
ftp> debug
Debugging on (debug=1).
ftp> passive
Passive mode: off; fallback to active mode: off.
ftp> dir
---> EPRT |2|3ffe:b80:3:9ad1::2|50480|
200 EPRT command successful.
---> LIST
150 Opening ASCII mode data connection for '/bin/ls'.
```

Capítulo 13

13.1 O erro de inicialização devido a uma contagem de argumento de linha de comando inválida seria registrado utilizando `syslog`.

13.2 As versões com TCP dos servidores `echo`, `discard` e `chargen` são todas executadas como um processo-filho, após serem bifurcadas (`fork`) por `inetd`, pois executam até que o cliente termine a conexão. Os outros dois servidores TCP, `time` e `daytime`, não exigem um `fork` porque seu serviço é simples de implementar (obtêm a data e hora atual, formatam, gravam e fecham a conexão); portanto, esses dois são tratados diretamente por `inetd`. Todos os cinco serviços de UDP são tratados sem `fork`, pois cada um gera no máximo um único datagrama em resposta ao datagrama de cliente que desencadeia o serviço. Esses cinco, portanto, são tratados diretamente por `inetd`.

13.3 Esse é um ataque de recusa de serviço bem-conhecido (CERT, 1996a). O primeiro datagrama da porta 7 faz o servidor `chargen` enviar um datagrama de volta para a porta 7. Isso é ecoado e envia outro datagrama para o servidor `chargen`. Esse loop continua. Uma solução, implementada no FreeBSD, é recusar datagramas para qualquer um dos servidores internos, caso a porta de origem do datagrama recebido pertença a qualquer um dos servidores internos. Outra solução é desativar esses serviços internos, ou por meio de `inetd` em cada host ou em um roteador da organização para a Internet.

13.4 O endereço IP e a porta do cliente são obtidos a partir da estrutura de endereço de soquete preenchida por `accept`.

A razão de `inetd` não fazer isso para um soquete UDP é porque o comando `recvfrom` para ler o datagrama é executado pelo servidor real que é executado (`exec`) e não pelo próprio `inetd`.

`inetd` poderia ler o datagrama especificando o flag `MSG_PEEK` (Seção 14.7), somente para obter o endereço IP e a porta do cliente, mas deixando o datagrama no lugar para o servidor real ler.

Capítulo 14

14.1 Se nenhum handler tivesse sido configurado, o retorno da primeira chamada a `signal` seria `SIG_DFL` e a chamada a `signal` para redefinir o handler apenas o configuraria de volta para seu default.

14.3 Eis somente o loop `for`:

```
for ( ; ; ) {
    if ( (n = Recv(sockfd, recvline, MAXLINE, MSG_PEEK)) == 0)
        break;       /* o servidor fechou a conexão */

    Ioctl(sockfd, FIONREAD, &npend);
    printf("%d bytes from PEEK, %d bytes pending\n", n, npend);

    n = Read(sockfd, recvline, MAXLINE);
    recvline[n] = 0;        /* termina com nulo */
    Fputs(recvline, stdout);
}
```

14.4 Os dados ainda aparecem na saída porque abandonar no fim da função `main` é o mesmo que retornar dessa função, e a função `main` é chamada pela rotina de inicialização C, como segue:

```
exit(main(argc, argv));
```

Daí, `exit` é chamada e, além disso, a rotina de limpeza de E/S-padrão também.

Capítulo 15

15.1 `unlink` remove o nome de caminho do sistema de arquivos e, quando o cliente chamar `connect` posteriormente, o comando `connect` falhará. O soquete ouvinte do servidor não é afetado, mas nenhum cliente poderá usar `connect` depois do comando `unlink`.

15.2 O cliente não pode se conectar ao servidor, mesmo que o nome de caminho ainda exista, porque, para o comando `connect` ser bem-sucedido, um soquete de domínio Unix deve estar correntemente aberto e vinculado a esse nome de caminho (Seção 15.4).

15.3 Quando o servidor imprime o endereço de protocolo do cliente chamando `sock_ntop`, a saída é "datagrama de (nenhum nome de caminho vinculado)", porque nenhum nome de caminho é vinculado ao soquete do cliente por default.

Uma solução é verificar especificamente um soquete de domínio Unix em `udp_client` e `udp_connect` e vincular (`bind`) um nome de caminho temporário ao soquete. Isso coloca a dependência de protocolo na função de biblioteca onde ele pertence e não em nossa aplicação.

15.4 Mesmo que obriguemos gravações (`write`) de 1 byte pelo servidor, para sua resposta de 26 bytes, colocar o comando `sleep` no cliente garante que todos os 26 segmentos sejam recebidos antes que `read` seja chamado, fazendo `read` retornar a resposta inteira. Isso serve apenas para confirmar (novamente) que o TCP é um fluxo de bytes sem marcadores de registro inerentes.

Para utilizar os protocolos de domínio Unix, iniciamos o cliente e o servidor com os dois argumentos de linha de comando `/local` (ou `/unix`) e `/tmp/daytime` (ou qualquer outro nome de caminho temporário que você queira utilizar). Nada muda: 26 bytes são retornados por `read` toda vez que o cliente executa.

Como o servidor especifica o flag MSG_EOR para cada send, cada byte é considerado um registro lógico e read retorna 1 byte toda vez que é chamado. O que está acontecendo aqui é que as implementações derivadas do Berkeley suportam o flag MSG_EOR por default. Entretanto, isso não está documentado e não deve ser utilizado em código de produção. Utilizamos aqui como um exemplo da diferença entre um fluxo de byte e um protocolo orientado para registro. Da perspectiva da implementação, cada operação de saída entra em um buffer de memória (mbuf) e o flag MSG_EOR é mantido pelo kernel com o mbuf, enquanto o mbuf vai do soquete de envio para o buffer de recebimento do soquete receptor. Quando read é chamado, o flag MSG_EOR ainda está anexado a cada mbuf; portanto, a rotina genérica read do kernel (que suporta o flag MSG_EOR, pois alguns protocolos o utilizam) retorna cada byte por si mesmo. Se tivermos utilizado recvmsg em vez de read, o flag MSG_EOR seria retornado no membro msg_flags sempre que recvmsg retornasse 1 byte. Isso não funciona com TCP porque o TCP de envio nunca vê o flag MSG_EOR no mbuf que está enviando e, mesmo que visse, não há nenhuma maneira de passá-lo para o TCP receptor no cabeçalho TCP. (Agradecemos a Matt Thomas por indicar esse "recurso" não documentado.)

15.5 A Figura E.13 mostra uma implementação desse programa.

debug/backlog.c

```
 1 #include   "unp.h"

 2 #define PORT        9999
 3 #define ADDR        "127.0.0.1"
 4 #define MAXBACKLOG  100

 5           /*globais */
 6 struct  sockaddr_in serv;
 7 pid_t   pid;                     /* do filho */

 8 int     pipefd[2];
 9 #define pfd pipefd[1]            /* extremidade do pai */
10 #define cfd pipefd[0]            /* extremidade do filho */

11           /* protótipos de função */
12 void   do_parent(void);
13 void   do_child(void);

14 int
15 main(int argc, char **argv)
16 {
17     if (argc != 1)
18         err_quit("usage: backlog");

19     Socketpair(AF_UNIX, SOCK_STREAM, 0, pipefd);

20     bzero(&serv, sizeof(serv));
21     serv.sin_family = AF_INET;
22     serv.sin_port = htons(PORT);
23     Inet_pton(AF_INET, ADDR, &serv.sin_addr);

24     if ( (pid = Fork()) == 0)
25         do_child();
26     else
27         do_parent();

28     exit(0);
```

Figura E.13 Determina o número real de conexões enfileiradas para diferentes valores de *backlog* (*continua*).

```
29 }
30 void
31 parent_alrm(int signo)
32 {
33     return;              /* apenas interrompe connect() bloqueado */
34 }
35 void
36 do_parent(void)
37 {
38     int    backlog, j, k, junk, fd[MAXBACKLOG + 1];
39     Close(cfd);
40     Signal(SIGALRM, parent_alrm);
41     for (backlog = 0; backlog <= 14; backlog++) {
42         printf("backlog = %d: ", backlog);
43         Write(pfd, &backlog, sizeof(int));  /* informa o valor do filho */
44         Read(pfd, &junk, sizeof(int));  /* espera pelo filho */
45         for (j = 1; j <= MAXBACKLOG; j++) {
46            fd[j] = Socket(AF_INET, SOCK_STREAM, 0);
47            alarm(2);
48            if (connect(fd[j], (SA *) &serv, sizeof(serv)) < 0) {
49                if (errno != EINTR)
50                    err_sys("connect error, j = %d", j);
51                printf("timeout, %d connections completed\n", j - 1);
52                for (k = 1; k <= j; k++)
53                     Close(fd[k]);
54                break;     /* próximo valor de acúmulo */
55            }
56            alarm(0);
57        }
58        if (j > MAXBACKLOG)
59            printf("%d connections?\n", MAXBACKLOG);
60     }
61     backlog = -1;                  /* diz ao filho que terminamos */
62     Write(pfd, &backlog, sizeof(int));
63 }
64 void
65 do_child(void)
66 {
67     int    listenfd, backlog, junk;
68     const int on = 1;
69     Close(pfd);
70     Read(cfd, &backlog, sizeof(int));    /* espera pelo pai */
71     while (backlog >= 0) {
72        listenfd = Socket(AF_INET, SOCK_STREAM, 0);
73        Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
74        Bind(listenfd, (SA *) &serv, sizeof(serv));
75        Listen(listenfd, backlog);  /* começa a ouvir */
76        Write(cfd, &junk, sizeof(int)); /* informa ao pai */
77        Read(cfd, &backlog, sizeof(int));  /* apenas espera pelo pai */
78        Close(listenfd);  /* fecha também todas as conexões enfileiradas */
79     }
80 }
```

debug/backlog.c

Figura E.13 Determina o número real de conexões enfileiradas para diferentes valores de *backlog* (*continuação*).

Capítulo 16

16.1 O descritor é compartilhado entre o pai e o filho; portanto, ele tem uma contagem de referência igual a 2. Se o pai chama `close`, isso apenas decrementa a contagem de referência de 2 para 1 e, como isso ainda é maior que 0, um FIN não é enviado. Essa é outra razão para a função `shutdown`: forçar um FIN a ser enviado, mesmo que a contagem de referência do descritor seja maior que 0.

16.2 O pai continuará gravando no soquete que recebeu um FIN e o primeiro segmento enviado para o servidor extrairá um RST em resposta. O próximo comando `write` depois desse enviará `SIGPIPE` para o pai, conforme discutimos na Seção 5.12.

16.3 Quando o filho chama `getppid` para enviar `SIGTERM` ao pai, a PID retornada será 1, o processo `init`, que herda todos os filhos cujos pais terminam enquanto seus filhos ainda estão executando. O filho tentará enviar o sinal para o processo `init`, mas não terá permissão adequada. Mas, se houver uma chance de que esse cliente possa executar com privilégios de superusuário, permitindo que envie esse sinal para `init`, então o valor de retorno de `getppid` deve ser testado antes de enviar o sinal.

16.4 Se essas duas linhas são removidas, `select` é chamada. Mas `select` retornará imediatamente, porque com a conexão estabelecida o soquete é gravável. Esse teste e `goto` servem para evitar a chamada desnecessária a `select`.

16.5 Isso pode acontecer quando o servidor envia dados imediatamente, quando sua função `accept` retorna, e quando o host cliente está ocupado no momento em que o segundo pacote do handshake de três vias chega para completar a conexão no lado do cliente (Figura 2.5). Os servidores de SMTP, por exemplo, gravam imediatamente em uma nova conexão, antes de ler dela, para enviar uma mensagem de saudação para o cliente.

Capítulo 17

17.1 Não, isso não importa, porque os três primeiros membros da união (`union`) na Figura 17.2 são estruturas de endereço de soquete.

Capítulo 18

18.1 O membro `sdl_nlen` será 5 e o membro `sdl_alen` será 8. Isso exige 21 bytes; portanto, o tamanho é arredondado para 24 bytes (página 89 do TCPv2), supondo uma arquitetura de 32 bits.

18.2 A resposta do kernel nunca é enviada para esse soquete. Essa opção de soquete determina se o kernel envia sua resposta para o processo emissor, como discutido nas páginas 649 e 650 do TCPv2. Ele tem ON como padrão, pois a maioria dos processos quer respostas. Mas desativar a opção impede que respostas sejam enviadas ao remetente.

Capítulo 20

20.1 Se você obtiver mais do que algumas respostas, elas não devem estar sempre na mesma ordem a cada vez. O host emissor, entretanto, é normalmente a primeira resposta, pois seus datagramas fazem loop internamente e não aparecem na rede real.

20.2 No FreeBSD, quando o handler de sinal grava o byte no pipe e então retorna, `select` retorna `EINTR`. Ele é chamado novamente e retorna legibilidade no pipe.

Capítulo 21

21.1 Quando executamos o programa, não há nenhuma saída. Para evitar a recepção acidental de datagramas de multicast que um servidor não está esperando, o kernel não entre-

ga grupos de multicast para um soquete que nunca realizou quaisquer operações de multicast (por exemplo, junção a um grupo). O que está acontecendo aqui é que o endereço de destino do datagrama de UDP é 224.0.0.1, o grupo de todos os hosts a que todos os nós capazes de usar multicast devem se juntar. O datagrama de UDP é enviado como um quadro Ethernet de multicast e todos os nós capazes de usar multicast recebem o datagrama, pois todos pertencem ao grupo. Entretanto, o kernel omite o datagrama recebido, pois o processo vinculado à porta de data/hora não configurou quaisquer opções de multicast.

21.2 A Figura E.14 mostra uma modificação simples da função `main` para vincular (`bind`) o endereço de multicast e a porta 0.

mcast/udpcli06.c

```
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     socklen_t salen;
 7     struct sockaddr *cli, *serv;
 8     if(argc != 2)
 9         err_quit("usage: udpcli06 <IPaddress>");
10     sockfd = Udp_client(argv[1], "daytime", (void **) &serv, &salen);
11     cli = Malloc(salen);
12     memcpy(cli, serv, salen);  /* copia a estrutura de endereço de soquete */
13     sock_set_port(cli, salen, 0);   /* e configura a porta como 0 */
14     Bind(sockfd, cli, salen);
15     dg_cli(stdin, sockfd, serv, salen);
16     exit(0);
17 }
```

mcast/udpcli06.c

Figura E.14 Função `main` do cliente UDP que vincula um endereço de multicast.

Infelizmente, nos três sistemas em que isso foi tentado – FreeBSD 4.8, MacOS X e Linux 2.4.7 –, todos permitiram a vinculação e, então, enviaram os datagramas de UDP com um endereço IP de origem de multicast.

21.3 Se fizermos isso a partir de nosso host `aix`, que é capaz de usar multicast, obteremos o seguinte:

```
aix % ping 224.0.0.1
PING 224.0.0.1: 56 data bytes
64 bytes from 192.168.42.2: icmp_seq=0 ttl=255 time=0 ms
64 bytes from 192.168.42.1: icmp_seq=0 ttl=64 time=1 ms (DUP!)
^C
----224.0.0.1 PING Statistics----
1 packets transmitted, 1 packets received, +1 duplicates, 0% packet loss
round-trip min/avg/max = 0/0/0 ms
```

Os dois sistemas na Ethernet do lado direito da Figura 1.16 respondem.

> Para evitar certos ataques de recusa de serviço, alguns sistemas não respondem a ping de broadcast ou multicast por default. Para fazer o `freebsd` responder, tivemos que configurá-lo com
>
> ```
> freebsd % sysctl net.inet.icmp.bmcastecho=1
> ```

21.5 O valor 1.073.741.824 é convertido para um número em ponto flutuante e dividido por 4.294.967.296, resultando em 0,250. Isso é multiplicado por 1.000.000, resultando em 250.000, que, em microssegundos, dá um quarto de segundo.

A maior fração é 4.294.967.295, que, dividida por 4.294.967.296, resulta em 0,99999999976716935634. Multiplicando isso por 1.000.000 e truncando para um inteiro, resulta em 999.999, o maior valor para o número de microssegundos.

Capítulo 21

22.1 Lembre-se de que `sock_ntop` utiliza seu próprio buffer estático para conter o resultado. Se o chamamos duas vezes como argumentos em uma chamada a `printf`, a segunda chamada sobrescreve o resultado da primeira.

22.2 Sim, se a resposta contém 0 bytes de dados de usuário (isto é, apenas uma estrutura `hdr`).

22.3 Como `select` não modifica a estrutura `timeval` que especifica seu limite de tempo, você precisa observar o momento em que o primeiro pacote é enviado (isso já é retornado em unidades de milissegundos por `rtt_ts`). Se `select` retorna com o soquete sendo legível, observe o tempo corrente e, se `recvmsg` é chamado novamente, calcule o novo tempo-limite para `select`.

22.4 A técnica comum é criar um soquete por endereço de interface, como fizemos na Seção 22.6, e enviar a resposta do mesmo soquete em que a solicitação chegou.

22.5 Chamar `getaddrinfo` sem argumento de nome de host e sem o flag `AI_PASSIVE` configurado faz com que ele assuma o endereço de host local: 0::1 (IPv6) e 127.0.0.1 (IPv4). Lembre-se de que uma estrutura de endereço de soquete IPv6 é retornada antes de uma estrutura de endereço de soquete IPv4 por `getaddrinfo`, supondo que IPv6 seja suportado. Se ambos os protocolos são suportados no host, a chamada a `socket` em `udp_client` será bem-sucedida com a família igual a `AF_INET6`.

A Figura E.15 é a versão independente de protocolo desse programa.

advio/udpserv04.c

```
 1 #include   "unpifi.h"
 2 void   mydg_echo(int, SA *, socklen_t);
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     sockfd, family, port;
 7     const int on = 1;
 8     pid_t   pid;
 9     socklen_t salen;
10     struct sockaddr *sa, *wild;
11     struct ifi_info *ifi, *ifihead;
12     if (argc == 2)
13         sockfd = Udp_client(NULL, argv[1], (void **) &sa, &salen);
14     else if (argc == 3)
15         sockfd = Udp_client(argv[1], argv[2], (void **) &sa, &salen);
16     else
17         err_quit("usage: udpserv04 [ <host> ] <service or port>");
18     family = sa->sa_family;
19     port = sock_get_port(sa, salen);
20     Close(sockfd);        /* queremos apenas a família, porta, salen */
```

Figura E.15 Versão independente de protocolo do programa da Seção 22.6 (*continua*).

```
21      for (ifihead = ifi = Get_ifi_info(family, 1);
22           ifi != NULL; ifi = ifi->ifi_next) {
23              /* vincula endereço de unicast */
24          sockfd = Socket(family, SOCK_DGRAM, 0);
25          Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
26          sock_set_port(ifi->ifi_addr, salen, port);
27          Bind(sockfd, ifi->ifi_addr, salen);
28          printf("bound %s\n", Sock_ntop(ifi->ifi_addr, salen));
29          if ( (pid = Fork()) == 0) { /* filho */
30              mydg_echo(sockfd, ifi->ifi_addr, salen);
31              exit(0);            /* nunca executado */
32          }
33          if (ifi->ifi_flags & IFF_BROADCAST) {
34                  /* tenta vincular endereço de broadcast */
35              sockfd = Socket(family, SOCK_DGRAM, 0);
36              Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
37              sock_set_port(ifi->ifi_brdaddr, salen, port);
38              if (bind(sockfd, ifi->ifi_brdaddr, salen) < 0) {
39                  if (errno == EADDRINUSE) {
40                      printf("EADDRINUSE: %s\n",
41                             Sock_ntop(ifi->ifi_brdaddr, salen));
42                      Close(sockfd);
43                      continue;
44                  } else
45                     err_sys("bind error for %s",
46                             Sock_ntop(ifi->ifi_brdaddr, salen));
47              }
48              printf("bound %s\n", Sock_ntop(ifi->ifi_brdaddr, salen));
49              if ( (pid = Fork()) == 0) { /* filho */
50                  mydg_echo(sockfd, ifi->ifi_brdaddr, salen);
51                  exit(0);        /* nunca executado */
52              }
53          }
54      }
55          /* vincula endereço curinga */
56      sockfd = Socket(family, SOCK_DGRAM, 0);
57      Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
58      wild = Malloc(salen);
59      memcpy(wild, sa, salen);     /* copia família e porta */
60      sock_set_wild(wild, salen);
61      Bind(sockfd, wild, salen);
62      printf("bound %s\n", Sock_ntop(wild, salen));
63      if ( (pid = Fork()) == 0) {   /* filho */
64          mydg_echo(sockfd, wild, salen);
65          exit(0);                /* nunca executado */
66      }
67      exit(0);
68  }
69  void
70  mydg_echo(int sockfd, SA *myaddr, socklen_t salen)
71  {
72      int     n;
73      char    mesg[MAXLINE];
74      socklen_t len;
75      struct sockaddr *cli;
```

Figura E.15 Versão independente de protocolo do programa da Seção 22.6 (*continua*).

```
76      cli = Malloc(salen);

77      for ( ; ; ) {
78          len = salen;
79          n = Recvfrom(sockfd, mesg, MAXLINE, 0, cli, &len);
80          printf("child %d, datagram from %s", getpid(), Sock_ntop(cli, len));
81          printf(", to %s\n", Sock_ntop(myaddr, salen));

82          Sendto(sockfd, mesg, n, 0, cli, len);
83      }
84  }
```

advio/udpserv04.c

Figura E.15 Versão independente de protocolo do programa da Seção 22.6 (*continuação*).

Capítulo 24

24.1 Sim, no primeiro exemplo, 2 bytes são enviados com um único ponteiro urgente que aponta para o byte após o `b`. Mas, no segundo exemplo (a duas chamadas de função), primeiro o `a` é enviado com um ponteiro urgente que aponta somente para além dele e isso é seguido por outro segmento TCP contendo o `b`, com um ponteiro urgente diferente, que aponta somente para além dele.

24.2 A Figura E.16 mostra a versão que utiliza `poll`.

oob/tcprecv03p.c

```
 1 #include   "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5      int    listenfd, connfd, n, justreadoob = 0;
 6      char   buff[100];
 7      struct pollfd pollfd[1];

 8      if(argc == 2)
 9          listenfd = Tcp_listen(NULL, argv[1], NULL);
10      else if (argc == 3)
11          listenfd = Tcp_listen(argv[1], argv[2], NULL);
12      else
13          err_quit("usage: tcprecv03p [ <host> ] <port#>");

14      connfd = Accept(listenfd, NULL, NULL);

15      pollfd[0].fd = connfd;
16      pollfd[0].events = POLLRDNORM;
17      for ( ; ; ) {
18          if (justreadoob == 0)
19              pollfd[0].events |= POLLRDBAND;
20          Poll(pollfd, 1, INFTIM);

21          if (pollfd[0].revents & POLLRDBAND) {
22              n = Recv(connfd, buff, sizeof(buff) - 1, MSG_OOB);
23              buff[n] = 0;          /* termina com nulo */
24              printf("read %d OOB byte: %s\n", n, buff);
25              justreadoob = 1;
26              pollfd[0].events &= ~POLLRDBAND;     /* desativa o bit */
27          }

28          if (pollfd[0].revents & POLLRDNORM) {
29              if ( (n = Read(connfd, buff, sizeof(buff) - 1)) == 0) {
30                  printf("received EOF\n");
```

Figura E.16 Versão da Figura 24.6 utilizando `poll` em vez de `select` (*continua*).

```
31                exit(0);
32            }
33            buff[n] = 0;          /* termina com nulo */
34            printf("read %d bytes: %s\n", n, buff);
35            justreadoob = 0;
36        }
37    }
38 }
```

oob/tcprecv03p.c

Figura E.16 Versão da Figura 24.6 utilizando `poll` em vez de `select` (*continuação*).

Capítulo 25

25.1 Não, a modificação introduz um erro. O problema é que `nqueue` é decrementada antes que a entrada de array `dg[iget]` seja processada, permitindo que o handler de sinal leia um novo datagrama nesse elemento do array.

Capítulo 26

26.1 No exemplo de `fork`, haverá 101 descritores em uso, um soquete ouvinte e 100 soquetes conectados. Mas cada um dos 101 processos (um pai, 100 filhos) tem somente um descritor aberto (ignorando todos os outros, como a entrada-padrão, caso o servidor não seja um daemon). No servidor com thread, entretanto, há 101 descritores em um único processo. Cada thread (incluindo o thread principal) está tratando de um descritor.

26.2 Os dois segmentos finais da terminação de conexão TCP – o FIN do servidor e o ACK do cliente desse FIN – não serão trocados. Isso deixa a extremidade do cliente da conexão no estado FIN_WAIT_2 (Figura 2.4). O tempo-limite excederá na extremidade do cliente, nas implementações derivadas do Berkeley, quando ele permanecer nesse estado por pouco mais de 11 minutos (páginas 825 a 827 do TCPv2). O servidor também ficará sem descritores (finalmente).

26.3 Essa mensagem deve ser impressa pelo thread principal quando ele ler um EOF do soquete *e* o outro thread ainda estiver executando. Uma maneira simples de fazer isso é declarar outra variável externa, chamada `done`, que é inicializada como 0. Antes que o thread `copyto` retorne, ele configura essa variável como 1. O thread principal verifica essa variável e, se for 0, imprime a mensagem de erro. Como somente um thread configura a variável, não há necessidade de qualquer sincronização.

Capítulo 27

27.1 Nada muda; todos os sistemas são vizinhos; portanto, uma rota de origem estrita é idêntica a uma rota de origem frouxa.

27.2 Colocaríamos um EOL (um byte igual a 0) no fim do buffer.

27.3 Como `ping` cria um soquete bruto (Capítulo 28), ele recebe o cabeçalho IP completo, incluindo todas as opções de IP, em cada datagrama que lê com `recvfrom`.

27.4 `rlogind` é ativado por `inetd` (Seção 13.5); portanto, o descritor 0 é o soquete para o cliente.

27.5 O problema é que o quinto argumento de `setsockopt` é o ponteiro para o comprimento, em vez do próprio comprimento. Esse bug provavelmente foi corrigido quando os protótipos C ANSI foram utilizados pela primeira vez.

Como se vê, o bug é inofensivo, porque, como mencionamos, para limpar a opção de soquete `IP_OPTIONS`, podemos especificar um ponteiro nulo como o quarto argumento ou um quinto argumento (o comprimento) igual a 0 (página 269 do TCPv2).

Capítulo 28

28.1 O campo de número de versão e o próximo campo de cabeçalho no cabeçalho IPv6 não estão disponíveis. O campo de comprimento de payload está disponível como um argumento de uma das funções de saída ou como o valor de retorno de uma das funções de entrada. Mas, se uma opção de payload jumbo é exigida, essa opção em si não está disponível para uma aplicação. O cabeçalho de fragmento também não está disponível para uma aplicação.

28.2 Por fim, o buffer de recebimento do soquete do cliente será preenchido, fazendo o comando `write` do daemon bloquear. Não queremos que isso aconteça, pois impede o daemon de tratar de quaisquer outros dados em qualquer um de seus soquetes. A solução mais fácil é o daemon configurar sua extremidade da conexão de domínio Unix para o cliente como não-bloqueadora. O daemon deve então chamar `write`, em vez da função empacotadora `Write`, e apenas ignorar um erro de `EWOULDBLOCK`.

28.3 Por default, os kernels derivados do Berkeley permitem broadcast em um soquete bruto (página 1057 do TCPv2). A opção de soquete `SO_BROADCAST` precisa ser especificada somente para soquetes UDP.

28.4 Nosso programa não verifica um endereço de multicast e não configura a opção de soquete `IP_MULTICAST_IF`. Portanto, o kernel escolhe a interface enviada, provavelmente pesquisando o endereço 224.0.0.1 na tabela de roteamento. Também não configuramos o campo `IP_MULTICAST_TTL`; portanto, ele usa o padrão 1, que é aceitável.

Capítulo 29

29.1 Esse flag indica que o buffer de desvio foi configurado por `sigsetjmp` (Figura 29.10). Embora o flag possa parecer supérfluo, há uma chance de que o sinal possa ser distribuído depois que o handler de sinal for estabelecido, mas antes da chamada a `sigsetjmp`. Mesmo que o programa não faça com que o sinal seja gerado, sinais podem ser gerados de outras maneiras, como com o comando `kill`.

Capítulo 30

30.1 O pai mantém o soquete ouvinte aberto caso precise bifurcar filhos adicionais posteriormente (o que seria um aprimoramento para nosso código).

30.2 Sim, um soquete de datagrama pode ser utilizado para passar um descritor, em vez de utilizar um soquete de fluxo. Com um soquete de datagrama, o pai não recebe um EOF em sua extremidade do pipe de fluxo, quando um filho termina prematuramente, mas poderia utilizar `SIGCHLD` para esse propósito. Uma diferença nesse cenário, onde `SIGCHLD` pode ser utilizado *versus* nosso daemon `icmpd` da Seção 28.7, é que, neste último, não havia nenhum relacionamento pai/filho entre o cliente e o servidor; portanto, o EOF no pipe de fluxo era a única maneira para o servidor detectar o desaparecimento de um cliente.

Capítulo 31

31.1 Assumimos aqui que o default para o protocolo é uma liberação ordenada quando o fluxo é fechado, o que é verdadeiro para o TCP.

Referências Bibliográficas

Todas as RFCs estão disponíveis gratuitamente por meio de correio eletrônico, de FTP anônimo ou da World Wide Web. Um ponto de partida é `http://www.ietf.org`. O diretório `ftp://ftp.rfc-editor.org/in-notes` é uma local de RFCs. URLs não são especificados para RFCs.

Os itens marcados como "Internet Draft" são trabalhos em andamento do IETF. Essas propostas (*drafts*) expiram seis meses após a publicação. A versão apropriada do rascunho pode ter mudado após a publicação deste livro ou o rascunho do mesmo pode ter sido publicado como uma RFC. Eles estão disponíveis gratuitamente via Internet, semelhante às RFCs. `http://www.ietf.org` é um repositório importante para Internet Drafts. Incluímos a parte referente ao nome do arquivo do URL para cada Internet Draft, pois o nome do arquivo contém o número de versão.

Toda cópia eletrônica encontrada de um artigo ou relatório referenciado nesta bibliografia teve seu URL incluído. Esteja ciente de que esses URLs podem mudar ao longo do tempo e os leitores são encorajados a verificar todas as Erratas deste livro na sua home page, para quaisquer alterações (`http://www.unpbook.com/`). Um banco de dados de artigos on-line impressionante pode ser encontrado no endereço `http://citeseer.nj.nec.com/cs`. Digitar o título de um artigo ou relatório não apenas localizará outros artigos que se referem ao que foi digitado, mas também apontará para versões on-line conhecidas.

Albitz, P. and Liu, C. 2001. *DNS and Bind, Fourth Edition.* O'Reilly & Associates, Sebastopol, CA.

Allman, M., Floyd, S., and Partridge, C. 2002. "Increasing TCP's Initial Window," RFC 3390.

Allman, M., Ostermann, S., and Metz, C. W. 1998. "FTP Extensions for IPv6 and NATs," RFC 2428.

Allman, M., Paxson, V., and Stevens, W. R. 1999. "TCP Congestion Control," RFC 2581.

Almquist, P. 1992. "Type of Service in the Internet Protocol Suite," RFC 1349 (obsoleta pela RFC 2474).

Definição original de como utilizar o campo de tipo de serviço no cabeçalho IPv4. Tornou-se obsoleta pela RFC 2474 (Nichols *et al*., 1998) e pela RFC 3168 (Ramakrishnan, Floyd e Black, 2001).

Baker, F. 1995. "Requirements for IP Version 4 Routers," RFC 1812.

Borman, D. A. 1997a. "Re: Frequency of RST Terminated Connections," end2end-interest mailing list (`http://www.unpbook.com/borman.97jan30.txt`).

Borman, D. A. 1997b. "Re: SYN/RST cookies," tcp-impl mailing list (`http://www.unpbook.com/borman.97jun06.txt`).

Borman, D. A., Deering, S. E., and Hinden, R. 1999. "IPv6 Jumbograms," RFC 2675.

Braden, R. T. 1989. "Requirements for Internet Hosts–Communication Layers," RFC 1122.

A primeira metade da RFC de requisitos de host. Essa metade cobre a camada de enlace, IPv4, ICMPv4, IGMPv4, ARP, TCP e UDP.

Braden, R. T. 1992. "TIME-WAIT Assassination Hazards in TCP," RFC 1337.

Braden, R. T., Borman, D. A., and Partridge, C. 1988. "Computing the Internet checksum," RFC 1071.

Bradner, S. 1996. "The Internet Standards Process – Revision 3," RFC 2026.

Bush, R. 2001. "Delegation of IP6.ARPA," RFC 3152.

Butenhof, D. R. 1997. *Programming with POSIX Threads.* Addison-Wesley, Reading, MA.

Cain, B., Deering, S. E., Kouvelas, I., Fenner, B., and Thyagarajan, A. 2002. "Internet Group Management Protocol, Version 3," RFC 3376.

Carpenter, B. and Moore, K. 2001. "Connection of IPv6 Domains via IPv4 Clouds," RFC 3056.

CERT, 1996a. "UDP Port Denial-of-Service Attack," Advisory CA-96.01, Computer Emergency Response Team, Pittsburgh, PA.

CERT, 1996b. "TCP SYN Flooding and IP Spoofing Attacks," Advisory CA-96.21, Computer Emergency Response Team, Pittsburgh, PA.

Cheswick, W. R., Bellovin, S. M., and Rubin, A. D. 2003. *Firewalls and Internet Security: Repelling the Wily Hacker, Second Edition.* Addison-Wesley, Reading, MA.

Conta, A. and Deering, S. E. 1998. "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification," RFC 2463.

Conta, A. and Deering, S. E. 2001. "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification," draft-ietf-ipngwg-icmp-v3-02.txt (Internet Draft).

Essa é uma revisão de Conta e Deering (1998) e espera-se que a substitua.

Crawford, M. 1998a. "Transmission of IPv6 Packets over Ethernet Networks," RFC 2464.

Crawford, M. 1998b. "Transmission of IPv6 Packets over FDDI Networks," RFC 2467.

Crawford, M., Narten, T., and Thomas, S. 1998. "Transmission of IPv6 Packets over Token Ring Networks," RFC 2470.

Deering, S. E. 1989. "Host extensions for IP multicasting," RFC 1112.

Deering, S. E. and Hinden, R. 1998. "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460.

Draves, R. 2003. "Default Address Selection for Internet Protocol version 6 (IPv6)," RFC 3484.

Eriksson, H. 1994. "MBONE: The Multicast Backbone," *Communications of the ACM*, vol. 37, no. 8, pp. 54-60.

Fink, R. and Hinden, R. 2003. "6bone (IPv6 Testing Address Allocation) Phaseout," draft-fink-6bone-phaseout-04.txt (Internet Draft).

Fuller, V., Li, T., Yu, J. Y., and Varadhan, K. 1993. "Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy," RFC 1519.

Garfinkel, S. L., Schwartz, A., and Spafford, E. H. 2003. *Practical UNIX & Internet Security, 3rd Edition.* O'Reilly & Associates, Sebastopol, CA.

Gettys, J. and Nielsen, H. F. 1998. *SMUX Protocol Specification* (`http://www.w3.org/TR/WD-mux`).

Gierth, A. 1996. *Private communication*.

Gilligan, R. E. and Nordmark, E. 2000. "Transition Mechanisms for IPv6 Hosts and Routers," RFC 2893.

Gilligan, R. E., Thomson, S., Bound, J., McCann, J., and Stevens, W. R. 2003. "Basic Socket Interface Extensions for IPv6," RFC 3493.

Gilligan, R. E., Thomson, S., Bound, J., and Stevens, W. R. 1997. "Basic Socket Interface Extensions for IPv6," RFC 2133 (obsoleted by RFC 2553).

Gilligan, R. E., Thomson, S., Bound, J., and Stevens, W. R. 1999. "Basic Socket Interface Extensions for IPv6," RFC 2553 (obsoleted by RFC 3493).

Haberman, B. 2002. "Allocation Guidelines for IPv6 Multicast Addresses," RFC 3307.

Haberman, B. and Thaler, D. 2002. "Unicast-Prefix-based IPv6 Multicast Addresses," RFC 3306.

Handley, M. and Jacobson, V. 1998. "SDP: Session Description Protocol," RFC 2327.

Handley, M., Perkins, C., and Whelan, E. 2000. "Session Announcement Protocol," RFC 2974.

Harkins, D. and Carrel, D. 1998. "The Internet Key Exchange (IKE)," RFC 2409.

Hinden, R. and Deering, S. E. 2003. "Internet Protocol Version 6 (IPv6) Addressing Architecture," RFC 3513.

Hinden, R., Deering, S. E., and Nordmark, E. 2003. "IPv6 Global Unicast Address Format," RFC 3587.

Hinden, R., Fink, R., and Postel, J. B. 1998. "IPv6 Testing Address Allocation," RFC 2471.

Holbrook, H. and Cheriton, D. 1999. "IP multicast channels: EXPRESS support for large-scale single-source applications," *Computer Communication Review*, vol. 29, no. 4, pp. 65-78.

Huitema, C. 2001. "An Anycast Prefix for 6to4 Relay Routers," RFC 3068.

IANA, 2003. *Protocol/Number Assignments Directory* (`http://www.iana.org/numbers.htm`).

IEEE, 1996. "Information Technology – Portable Operating System Interface (POSIX)–Part 1: System Application Program Interface (API) [C Language]," IEEE Std 1003.1, 1996 Edition, Institute of Electrical and Electronics Engineers, Piscataway, NJ.

Essa versão do POSIX.1 contém a API de base de 1990, as extensões 1003.1b de tempo real (1993), os pthreads 1003.1c (1995) e as correções técnicas 1003-1i (1995). Também é o International Standard ISO/IEC 9945-1: 1996 (E). O pedido de informações sobre padrões IEEE e padrões de rascunho está disponível no endereço `http://www.ieee.org`.

IEEE, 1997. *Guidelines for 64-bit Global Identifier (EUI-64) Registration Authority.* Institute of Electrical and Electronics Engineers, Piscataway, NJ (`http://standards.ieee.org/regauth/oui/tutorials/EUI64.html`).

Jacobson, V. 1988. "Congestion Avoidance and Control," *Computer Communication Review*, vol. 18, no. 4, pp. 314-329 (`ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z`).

Um artigo clássico que descreve algoritmos para evitar o início lento e o congestionamento para TCP.

Jacobson, V., Braden, R. T., and Borman, D. A. 1992. "TCP Extensions for High Performance," RFC 1323.

Descreve a opção de escala de janela, a opção de indicação de tempo (timestamp) e o algoritmo PAWS, junto com as razões pelas quais essas modificações foram necessárias.

Jacobson, V., Braden, R. T., and Zhang, L. 1990. "TCP Extension for High-Speed Paths," RFC 1185 (obsoleta pela RFC 1323).

Josey, A., ed. 1997. *Go Solo 2: The Authorized Guide to Version 2 of the Single UNIX Specification.* Prentice Hall, Uppser Saddle River, NJ.

Josey, A., ed. 2002. *The Single UNIX Specification–The Authorized Guide to Version 3.* The Open Group, Berkshire, UK.

Joy, W. N. 1994. *Private communication*.

Karn, P. and Partridge, C. 1991. "Improving Round-Trip Time Estimates in Reliable Transport Protocols," *ACM Transactions on Computer Systems*, vol. 9, no. 4, pp. 364-373.

Katz, D. 1993. "Transmission of IP and ARP over FDDI Networks," RFC 1390.

Katz, D. 1997. "IP Router Alert Option," RFC 2113.

Kent, S. T. 1991. "U. S. Department of Defense Security Options for the Internet Protocol," RFC 1108.

Kent, S. T. 2003a. "IP Authentication Header," draft-ietf-ipsec-rfc2402bis-04.txt (Internet Draft).

Kent, S. T. 2003b. "IP Encapsulating Security Payload (ESP)," draft-ietf-ipsec-esp-v3-06.txt (Internet Draft).

Kent, S. T. and Atkinson, R. J. 1998a. "Security Architecture for the Internet Protocol," RFC 2401.

Kent, S. T. and Atkinson, R. J. 1998b. "IP Authentication Header," RFC 2402.

Enquanto este livro estava sendo escrito, essa RFC estava sendo atualizada pelo IETF IPsec Working Group (veja Kent [2003a]).

Kent, S. T. and Atkinson, R. J. 1998c. "IP Encapsulating Security Payload (ESP)," RFC 2406.

Enquanto este livro estava sendo escrito, essa RFC estava sendo atualizada pelo IETF IPsec Working Group (veja Kent [2003b]).

Kernighan, B. W. and Pike, R. 1984. *The UNIX Programming Environment.* Prentice Hall, Englewood Cliffs, NJ.

Kernighan, B. W. and Ritchie, D. M. 1988. *The C Programming Language, Second Edition.* Prentice Hall, Englewood Cliffs, NJ.

Lanciani, D. 1996. "Re: sockets: AF_INET vs. PF_INET," Message-ID: <3561@news.IPSWITCH.COM>, USENET comp.protocols.tcp-ip Newsgroup (`http://www.unpbook.com/lanciani.96apr10.txt`).

Maslen, T. M. 1997. "Re: gethostbyXXXX() and Threads," Message-ID: <maslen.862463630@shellx>, USENET comp.programming.threads Newsgroup (`http://www.unpbook.com/maslen.97may01.txt`).

McCann, J., Deering, S. E., and Mogul, J. C. 1996. "Path MTU Discovery for IP version 6," RFC 1981.

McCanne, S. and Jacobson, V. 1993. "The BSD Packet Filter: A New Architecture for User-Level Packet Capture," *Proceedings of the 1993 Winter USENIX Conference*, San Diego, CA, pp. 259-269.

McDonald, D. L., Metz, C. W., and Phan, B. G. 1998. "PF_KEY Key Management API, Version 2," RFC 2367.

McKusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S. 1996. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley, Reading, MA.

Meyer, D. 1998. "Administratively Scoped IP Multicast," RFC 2365.

Mills, D. L. 1992. "Network Time Protocol (Version 3) Specification, Implementation," RFC 1305.

Mills, D. L. 1996. "Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI," RFC 2030.

Mogul, J. C. and Deering, S. E. 1990. "Path MTU discovery," RFC 1191.

Mogul, J. C. and Postel, J. B. 1985. "Internet Standard Subnetting Procedure," RFC 950.

Narten, T. and Draves, R. 2001. "Privacy Extensions for Stateless Address Autoconfiguration in IPv6," RFC 3041.

Nemeth, E. 1997. *Private communication*.

Nichols, K., Blake, S., Baker, F., and Black, D. 1998. "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers," RFC 2474.

Nordmark, E. 2000. "Stateless IP/ICMP Translation Algorithm (SIIT)," RFC 2765.

Ong, L., Rytina, I., Garcia, M., Schwarzbauer, H., Coene, L., Lin, H., Juhasz, I., Holdrege, M., and Sharp, C. 1999. "Framework Architecture for Signaling Transport," RFC 2719.

Ong, L. and Yoakum, J. 2002. "An Introduction to the Stream Control Transmission Protocol (SCTP)," RFC 3286.

The Open Group, 1997. *CAE Specification, Networking Services (XNS), Issue 5.* The Open Group, Berkshire, UK.

> Essa é a especificação para soquetes e XTI no Unix 98, agora superada pela *The Single UNIX Specification, Version 3*. Esse manual também tem apêndices que descrevem o uso de XTI com NetBIOS, os protocolos OSI, SNA e os protocolos NetWare IPX e SPX. Três apêndices abordam o uso de soquetes e XTI com ATM.

Partridge, C. and Jackson, A. 1999. "IPv6 Router Alert Option," RFC 2711.

Partridge, C., Mendez, T., and Milliken, W. 1993. "Host Anycasting Service," RFC 1546.

Partridge, C. and Pink, S. 1993. "A Faster UDP," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 429-440.

Paxson, V. 1996. "End-to-End Routing Behavior in the Internet," *Computer Communication Review*, vol. 6, no. 4, pp. 25-38 (`ftp://ftp.ee.lbl.gov/papers/routing.SIGCOMM.ps.Z`).

Paxson, V. and Allman, M. 2000. "Computing TCP's Retransmission Timer," RFC 2988.

Plauger, P. J. 1992. *The Standard C Library.* Prentice Hall, Englewood Cliffs, NJ.

Postel, J. B. 1980. "User Datagram Protocol," RFC 768.

Postel, J. B. 1981a. "Internet Protocol," RFC 791.

Postel, J. B. 1981b. "Internet Control Message Protocol," RFC 792.

Postel, J. B. 1981c. "Transmission Control Protocol," RFC 793.

Pusateri, T. 1993. "IP Multicast over Token-Ring Local Area Networks," RFC 1469.

Rago, S. A. 1993. *UNIX System V Network Programming.* Addison-Wesley, Reading, MA.

Rajahalme, J., Conta, A., Carpenter, B., and Deering, S. E. 2003. "IPv6 Flow Label Specification," draft-ietf-ipv6-flow-label-07.txt (Internet Draft).

Ramakrishnan, K., Floyd, S., and Black, D. 2001. "The Addition of Explicit Congestion Notification (ECN) to IP," RFC 3168.

Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G. J., and Lear, E. 1996. "Address Allocation for Private Internets," RFC 1918.

Reynolds, J. K. 2002. "Assigned Numbers: RFC 1700 is Replaced by an On-line Database," RFC 3232.

> O banco de dados referido nessa RFC é IANA (2003).

Reynolds, J. K. and Postel, J. B. 1994. "Assigned Numbers," RFC 1700 (obsoleta pela RFC 3232).

> Esse RFC é o último da série de RFCs "Assigned Numbers". Como as informações mudaram freqüentemente, foi decidido simplesmente manter o diretório on-line. Veja [Reynolds 2002] para mais explicações ou [IANA 2003] para o próprio banco de dados.

Ritchie, D. M. 1984. "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1897-1910.

Salus, P. H. 1994. *A Quarter Century of Unix.* Addison-Wesley, Reading, MA.

Salus, P. H. 1995. *Casting the Net: From ARPANET to Internet and Beyond.* Addison-Wesley, Reading, MA.

Schimmel, C. 1994. *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers.* Addison-Wesley, Reading, MA.

Spero, S. 1996. *Session Control Protocol (SCP)* (`http://www.w3.org/Protocols/HTTP-NG/http-ng-scp.html`).

Srinivasan, R. 1995. "XDR: External Data Representation Standard," RFC 1832.

Stevens, W. R. 1992. *Advanced Programming in the UNIX Environment.* Addison-Wesley, Reading, MA.

Todos os detalhes de programação de Unix. Referido neste livro como APUE.

Stevens, W. R. 1994. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley, Reading, MA.

Uma introdução completa aos protocolos de Internet. Referido neste livro como TCPv1.

Stevens, W. R. 1996. *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*. Addison-Wesley, Reading, MA.

Referido neste livro como TCPv3.

Stevens, W. R. and Thomas, M. 1998. "Advanced Sockets API for IPv6," RFC 2292 (obsoleta pela RFC 3542).

Stevens, W. R., Thomas, M., Nordmark, E., and Jinmei, T. 2003. "Advanced Sockets Application Program Interface (API) for IPv6," RFC 3542.

Stewart, R. R., Bestler, C., Jim, J., Ganguly, S., Shah, H., and Kashyap, V. 2003a. "Stream Control Transmission Protocol (SCTP) Remote Direct Memory Access (RDMA) Direct Data Placement (DDP) Adaptation," draft-stewart-rddp-sctp-02.txt (Internet Draft).

Stewart, R. R., Ramalho, M., Xie, Q., Tuexen, M., Rytina, I., Belinchon, M., and Conrad, P. 2003b. "Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration," draftietf-tsvwg-addip-sctp-07.txt (Internet Draft).

Stewart, R. R. and Xie, Q. 2001. *Stream Control Transmission Protocol (SCTP): A Reference Guide*. Addison-Wesley, Reading, MA.

Stewart, R. R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L., and Paxson, V. 2000. "Stream Control Transmission Protocol," RFC 2960.

Stone, J., Stewart, R. R., and Otis, D. 2002. "Stream Control Transmission Protocol (SCTP) Checksum Change," RFC 3309.

Tanenbaum, A. S. 1987. *Operating Systems Design and Implementation.* Prentice Hall, Englewood Cliffs, NJ.

Thomson, S. and Huitema, C. 1995. "DNS Extensions to support IP version 6," RFC 1886.

Torek, C. 1994. "Re: Delay in re-using TCP/IP port," Message-ID: <199501010028.QAA16863@elf.bsdi.com>, USENET comp.unix.wizards Newsgroup (`http://www.unpbook.com/torek.94dec31.txt`).

Touch, J. 1997. "TCP Control Block Interdependence," RFC 2140.

Unix International, 1991. *Data Link Provider Interface Specification.* Unix International, Parsippany, NJ, Revision 2.0.0 (`http://www.unpbook.com/dlpi.2.0.0.ps`).

Uma versão mais recente dessa especificação está disponível on-line em The Open Group, no endereço `http://www.rdg.opengroup.org/pubs/catalog/web.htm`.

Unix International, 1992a. *Network Provider Interface Specification.* Unix International, Parsippany, NJ, Revision 2.0.0 (`http://www.unpbook.com/npi.2.0.0.ps`).

Unix International, 1992b. *Transport Provider Interface Specification.* Unix International, Parsippany, NJ, Revision 1.5 (`http://www.unpbook.com/tpi.1.5.ps`).

Uma versão mais recente dessa especificação está disponível on-line no The Open Group, no endereço `http://www.rdg.opengroup.org/pubs/catalog/web.htm`.

Vixie, P. A. 1996. *Private communication*.

Wright, G. R. and Stevens, W. R. 1995. *TCP/IP Illustrated, Volume 2: The Implementation.* Addison-Wesley, Reading, MA.

A implementação dos protocolos de Internet no sistema operacional 4.4BSD-Lite. Referida neste texto como TCPv2.

Índice

B

C

E

F

G

J

K

M

N

Q

R

S

T

U

V

W

X

Y

Z