国外计算机科学教材系列

ANSI C版

用TCP/IP进行网际互联

第二卷：设计、实现与内核

（第三版）

Internetworking With TCP/IP Vol II:

Design, Implementation, and Internals

Third Edition

[美] Douglas E. Comer
David L. Stevens 著

张 娟 王 海 黄述真 译

谢希仁 审校

PH PTR

電子工業出版社
Publishing House of Electronics Industry
URL:http://www.phei.com.cn

国外计算机科学教材系列

用 TCP/IP 进行网际互联

第二卷：设计、实现与内核

（第三版）

Internetworking with TCP/IP Vol Ⅱ:
Design, Implementation, and Internals
Third Edition

[美] Douglas E. Comer
David L. Stevens 著

张 娟 王 海 黄述真 译

谢希仁 审校

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内容简介

本书是一部有关计算机网络的经典教科书。它是目前美国大多数大学里所开设的计算机网络课程的主要参考书。目前国内外能见到的各种有关TCP/IP的书籍，其主要内容均出自本书。本书的特点是：强调原理、概念准确、深入浅出、内容丰富且新颖。

全书共分为三卷。第一卷介绍了TCP/IP的基本概念，第二卷在第一卷的基础上，进一步详细讨论了TCP/IP的实现过程，这一卷的突出特点是非常注重实际。本书作者利用程序代码实现了TCP/IP的每一个具体细节，并且所有的代码在书中均可找到，这对于读者深入了解并掌握TCP/IP的细节内容大有帮助。各章之后附有很多习题，内容全面且结合实际。全书最后还有三个附录，分别给出了过程调用交叉参考表、程序代码中用到的C数据结构交叉参考表以及Xinu函数和常量。本书可供计算机和通信专业的研究生、高年级本科生作为教科书和学习参考书，也可供各种从事科研的人员参考。

Authorized translation from the English language edition published by Prentice-Hall, Inc. Copyright © 1999.
All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.
Simplified Chinese language edition published by Publishing House of Electronics Industry. Copyright © 2001.
本书中文简体专有翻译出版权由Pearson教育集团所属的Prentice-Hall, Inc.授予电子工业出版社。其原文版权及中文翻译出版权受法律保护。未经许可，不得以任何形式或手段复制或抄袭本书内容。

图书在版编目(CIP)数据

用TCP/IP进行网际互联第二卷：设计、实现与内核（第三版）/（美）科默（Comer, D. E.）史蒂文（Stevens, D. L.）著；张娟等译. —北京：电子工业出版社，2001.4
书名原文：Internetworking With TCP/IP Vol Ⅱ: Design, Implementation, and Internals Third Edition
国外计算机科学教材系列
ISBN 7-5053-6630-0

Ⅰ.用… Ⅱ.①科…②张… Ⅲ.计算机网络－通信协议－教材 Ⅳ.TN915.04

中国版本图书馆CIP数据核字（2001）第22857号

丛 书 名：国外计算机科学教材系列
书　　名：**用TCP/IP进行网际互联第二卷：设计、实现与内核（第三版）**
原 书 名：Internetworking With TCP/IP Vol Ⅱ: Design, Implementation, and Internals Third Edition
著　　者：［美］Douglas E. Comer David L. Stevens
译　　者：张 娟 王 海 黄述真
审 校 者：谢希仁
责任编辑：马 岚
排版制作：今日电子公司制作部
印 刷 者：北京东光印刷厂
出版发行：电子工业出版社 URL:http://www.phei.com.cn
北京市海淀区万寿路173信箱 邮编：100036 电话：68279077
经　　销：各地新华书店
开　　本：787 × 1092 1/16　　印张：34　字数：849千字
版　　次：2001年4月第1版　　2001年4月第1次印刷
书　　号：ISBN 7-5053-6630-0
TP · 3686
定　　价：52.00元
版权贸易合同登记号 图字：01-2000-3470

凡购买电子工业出版社的图书，如有缺页、倒页、脱页者，请向购买书店调换；若书店售缺，请与本社发行部联系调换。

出版说明

随着21世纪的到来，计算机技术的发展更加迅猛，在各行各业的应用更加广泛，越来越多的高等院校增设了有关计算机科学的课程内容，或对现有计算机课程设置进行了适当调整，以紧跟前沿技术。在这个教学体系和学科结构变革的大环境下，对适合不同院系、不同专业、不同层次的教材的需求量与日俱增。此时，如果能够借鉴、学习国外一流大学的先进教学体系，引进具有先进性、实用性和权威性的国外一流大学计算机教材，汲取其精华，必能更好地促进中国高等院校教学的全面改革。

美国 Prentice Hall 出版公司是享誉世界的高校教材出版商，自1913年成立以来，一直致力于教材的出版，所出版的计算机教材为美国众多大学采用，其中有不少是专业领域中的经典名著，已翻译成多种文字在世界各地的大学中使用，成为全人类的共同财富。许多蜚声世界的教授、学者都是该公司的资深作者，如道格拉斯·科默（Douglas E. Comer）、威廉·斯大林（William Stallings）等。早在1997年，电子工业出版社就从Prentice Hall引进了一套计算机英文版专业教材，并将其翻译出版，同时定名为《国外计算机科学教材系列》（下称：第一轮教材）。截至2000年12月，该系列教材已出版23种，深受读者欢迎，被许多大学选为高年级学生和研究生教材或参考书。

4年过去了，已出版的教材中多数已经有了后续版本。因此，我们开始设计新一轮教材（第二轮教材）的出版，成立了由我国计算机界著名专家和教授组成的“教材出版委员会”，并结合第一轮教材的使用情况和师生反馈意见，组织了第二轮《国外计算机科学教材系列》出版工作。

第二轮教材的出版原则为：

1. 引进 Prentice Hall 出版公司2000年和2001年推出的新版教材，作为替换版本。
2. 在著名高校教授的建议下，除了从 Prentice Hall 新选了一些教材之外，还从 McGraw-Hill 和 Addison Wesley Longman 等著名专业教材出版社、麻省理工学院出版社和剑桥大学出版社等著名大学出版社引进了一些经典教材，作为增补版本。
3. 对于第一轮中无新版本的优秀教材，我们将其作为延用版本，直接进入第二轮使用。
4. 对于第一轮中翻译质量较好且无新版本的教材，我们将其进行了修订，也作为延用版本，进入第二轮使用。

这次推出的教材覆盖学科范围广、领域宽、层次多，既有本科专业课程教材，也有研究生课程教材，以适应不同院系、不同专业、不同层次的师生对教材的需求。广大师生可自由选择和自由组合使用。

按照计划，本轮教材规划出版37种，其中替换版本8种，新增版本14种，延用版本15种。教材内容涉及的学科方向包括网络与通信、操作系统、计算机组织与结构、算法与数据结构、数据库与信息处理、编程语言、图形图像与多媒体、软件工程等。本轮教材计划于2001年7月前全部出版。教材的使用年限平均为3年。我们还将陆续推出一些教材的参考课件，希望能为授课老师提供帮助。

为了保证本轮教材的选题质量和翻译质量，我们约请了清华大学、北京大学、北京航空航天大学、复旦大学、上海交通大学、南京大学、浙江大学、哈尔滨工业大学、华中科技大学、西安交通

大学、国防科学技术大学、解放军理工大学等著名高校的教授和骨干教师参与了本轮教材的选题、翻译和审校工作。他们中既有讲授同类教材的骨干教师和博士，也有积累了几十年教学经验的教授和博士生导师。

在本轮教材的选题、翻译和编辑加工过程中，为提高教材质量，我们做了大量细致的工作，包括：

1. 对于新选题和新版本进行了全面论证。
2. 对于延用版本，认真审查了前一版本教材，修改了其中的印刷错误。
3. 对于译者和编辑的选择，达到了专业对口。
4. 对于从英文原书中发现的错误，我们通过与作者联络、从网上下载勘误表等方式，一一做了修改。
5. 对于翻译、审校、编辑、排版、印刷质量进行了严格的审查把关。

通过这些工作，保证了本轮教材的质量较前一轮有明显的提高。相信读者一定能够从字里行间体会到我们的这些努力。

今后，我们将继续加强与各高校教师的密切联系，为广大师生引进更多的国外优秀教材和参考书，为我国计算机科学教学体系与国际教学体系的接轨做出努力。

由于我们对国际计算机科学、我国高校计算机教育的发展存在认识上的不足，在选题、翻译、出版等方面的工作中还有许多有待提高之处，恳请广大师生和读者提出批评和建议。

电子工业出版社

2001 年春

教材出版委员会

主　任　　杨芙清　北京大学教授
中国科学院院士
北京大学信息与工程学部主任
北京大学软件工程研究所所长

委　员　　王　珊　中国人民大学信息学院院长、教授

胡道元　清华大学计算机科学与技术系教授
国际信息处理联合会通信系统中国代表

钟玉琢　清华大学计算机科学与技术系教授
中国计算机学会多媒体专业委员会主任

谢希仁　中国人民解放军理工大学教授
全军网络技术研究中心主任、博士生导师

尤晋元　上海交通大学计算机科学与工程系教授
上海分布计算技术中心主任

施伯乐　中国计算机学会常务理事、上海市计算机学会理事长
上海国际数据库研究中心主任、复旦大学教授

邹　鹏　国防科学技术大学计算机学院教授、博士生导师
教育部计算机基础教学课程指导委员会副主任委员

张昆藏　青岛大学信息工程学院教授

前　　言

很荣幸 Doug Comer 给我这个机会，让我能在他的这本书第三版付印时与诸位探讨一些个人的想法。在过去的十年期间，Internet 以惊人的速度发展着。Internet 上主机的数目从 1989 年的 100 000 台发展到 1998 年的 30 000 000 台。在本书第二版出版的时候，连接到 Internet 的网络大约有 26 000 个。到 1998 年，这个数字大约在 200 000 到 350 000 之间，这还不包括那些利用互联网技术但未连接到公共网络上的专用内联网。

除了规模上的变化，Internet 在应用上也发生了惊人的变化。Internet 技术与正在蓬勃发展的 WWW 作为一个完整的、不可缺少的系统已被人们所认可。而 WWW 在商业、学术和政府等部门都掀起了一场革命。在 WWW 上每天出现的“网页”可达 3.2 亿之多，并且还在不断涌现。有些学校大约四分之一的入学申请都是通过电子邮件或其他网页申请形式进行的。Dell 公司透露，他们通过网站每天销售价值大约 600 万美元的 PC 机。Amazon 网上书店以季度盈利达 6600 万美元而成为历史上发展最快的公司，这也是第一个在一年之内销售额超过 2.5 亿美元的公司。

目前，至少有两千多家无线电台将其音频服务推向了 Internet，许多网站也开始提供音频甚至低质量的视频服务。随着访问速度提高到兆比特的范围以及主干网容量的增加，视频服务质量也将得到提高。

随着 1996 年网络电视和 1997 年 Nokia 推出的支持网络功能的手机的问世，其他支持网络功能的产品也不断地涌现。更多的实用产品实现了相似的网络功能，例如水暖加热器既可以由家用电脑来控制，也可以由电力公司来控制、以满足适度的调峰需求。尽管智能代理还没有成为主流，但是 XML 近来的发展也表明 Internet 上有关事务处理的应用正在急剧增多。SML 的标准“文档”表示方式以及一致的常用解释格式构成了一种可传输对象，该对象已形成所有商业、金融事务、数据库事务及其他需要标准表示方式和解释格式的交易的基础。

再回头看看电话产品的发展，无论传统的服务商还是现代企业都在将支持网络的电话产品推向市场。“Soft PBX”系统利用 LAN 和 Internet 实现传统的专用交换机的功能，基于微处理器的电话机也正在改变远程通信的经济情况和可操作性。具有 IP 功能的传真机已经出现。网关把原有的模拟世界与现在的 Internet 世界联系在一起，起着非常重要的作用。

对 Internet 服务需求量的增长速度赶上甚至超过了网络本身的增长。主干网络的速度达到 622Mb/s 已是很平常的事。利用硬件 IP 交换机的新一代路由器可望以 10Gb/s（OC192）以上的速度处理 Internet 分组。下一个挑战是利用单模光纤处理每秒兆兆字节的信息量。

网络的安全性过去总是处于后台操作状态，随着更多领域依赖于Internet，安全性已逐渐走向前台。防火墙技术、端-端加密、密钥管理、证书系统和鉴别系统已成为成功管理Internet的关键因素。

再向未来展望，网络协议和结构已经朝着星际互联的方向发展，也许会形成和互联网一样的形式。域名系统如果还存在，还要考虑将不同的行星计入命名体系。“互联的 Internet”协议将能够处理更高的延迟，传统的 TCP 概念也将被更多的单向过程取代。未来的网际遨游者回看这十年，一定会觉得这是个充满挑战但技术相对落后的时代。在他们看来，这段前言中的观点可能已经过时又难以理解，因为他们所面对的是经历了另外二十年变革的 Internet。

Vinton Cerf
Camelot
Northern Virginia

序　　言

用TCP/IP进行网际互联第二卷提供了第一卷所没有包含的关于TCP/IP协议的一些细节问题。第二卷如同将TCP/IP置于放大镜下，考察每个协议的具体细节。它讨论了协议的实现方案，并着重于介绍协议软件的内部机制。第三版包含了对某些协议的修改和更新的内容。其中的代码改用C++语言的ANSI标准C子集，包含了函数原型和参数的声明。另外还纠正了一些错误。我们将SNMP更新为SNMPv2，其中包括替换了地址转换表，增加了UDP listener列表。在本书的最后，还增加了附录2，给出了书中代码所用到的主要数据结构的声明及变量的交叉参考表，并扩充了附录1中的过程调用交叉参考的内容。

书中的范例代码用Cnu C++编译器在Intel体系平台编译通过，也在Pentium™系统上经过测试。所有的代码均可在以下网址得到：

ftp://ftp.cs.purdue.edu/pub/comer/TCPIP-vol2.dist.tar.Z

虽然本书受到版权保护，但书中的代码可供读者使用，而且已经在许多商业产品中实现。这些代码使用的惟一限制是不得在公开出版物中出版。

我们鼓励读者利用计算机工具来查看、修改、编译和测试这些代码。事实上，尽管附录1和附录2中提供了定位代码条目的有效途径，但在查看大段代码时，UNIX grep程序的价值更是不可估量。

对于各种正式的协议规范，以及对协议的实现和使用的讨论，可参见请求评论文档（RFC）。尽管一些RFC文档对初学者来讲难以理解，但这些文档是信息详尽的权威性资源，没有哪个作者能够做到在自己编写的书中包含RFC文档中的所有内容。尽管RFC文档涉及了每一个协议，但有时它们对协议之间的交互问题并未加以说明。例如，选路信息协议（RIP或OSPF）之类的选路协议规定了网关如何将路由置入IP路由表中，以及如何将表中的路由通知其他网关。RIP还规定路由必须设立定时机制，一旦某条路由超时，就将其删除。但是，我们在RFC文档中并不容易看出RIP和其他协议之间是如何交互作用的，随之而来的问题是："路由超时机制将如何影响路由表中那些由ICMP设置的路由呢？"我们可能还会考虑这样一个问题："当RIP更新路由时，应不应该推翻那些由管理员直接输入的路由呢？"

为了有助于解释协议之间的交互作用，并确保我们的方案能协调工作，我们设计并构造了一个工作系统，作为全书的中心范例。该系统提供了TCP/IP协议族中的大部分协议，包括：TCP、IP、ICMP、IGMP、UDP、ARP、RIP、SNMPv2以及OSPF的主要部分。另外，该系统还有一个finger服务的客户和服务器范例。由于本书包括了每一个协议的程序代码，读者可以研究其实现方法并了解其内部结构。最重要的是，由于范例系统将所有协议软件集成为一个工作整体，读者可以清楚地了解协议之间的交互作用。

范例的程序代码试图做到一方面遵守协议标准，另一方面引入一些新的思想。例如，我们的TCP程序代码中包含了"糊涂窗口预防"技术、Jacobson-Karels的"慢启动"和"拥塞预防"等优化技术，诸如此类的性能可能在商业实现中被忽略。但同时我们也清楚地认识到商业领域并不总是遵从已经公布的标准，因此我们也努力将系统调整到能够在现实环境中使用。例如，程序代码中包含了

一个配置参数，使得它既可以采纳Internet标准，也可以采纳BSD UNIX中“TCP紧急数据指针”的实现方法。

我们并不认为书中所提供的程序代码都是准确无误的，甚至不能断言它肯定比其他实现方法要好。事实上，经过多年使用，我们仍然在不断完善这套软件，同时，也希望读者跟我们一起继续改进它。

本书可以作为网络工程人员的高级教程或者作为研究生教材使用。在作为本科教程使用时，应将重点放在前几章，而忽略有关OSPF、SNMP和RIP这几章内容。研究生可能会在有关TCP的章节中发现一些最为有趣和最难理解的概念。为保证其高性能而采用的自适应重发和相关的试探法尤为重要，应当仔细加以研究。纵观全书，绝大部分习题都向大家提示了其他可选择的实现方案及其大致情况，这些内容并不要求死记硬背，学生们可能需要阅读本书以外的其他资料，才能解答这些习题。

正如任何耗费甚巨的工程一样，本书中包含了许多人的心血，对此我们表示衷心的感谢。作者之一David Stevens完成了大部分软件的编制工作，其中包括一个完整的TCP版本。Shawn Ostermann为本书付出了许多努力。Shawn将TCP/IP代码集成到Xinu版本8中，并将其从最初的Sun 3平台移植到DECstation 3100上。在这一版本中，他还对上一版中由Vic Norman编写的SNMP软件进行了大量的修改，以符合第二版的标准并有助于代码的调试。John Lin对书中的技术细节进行了校对，纠正了一些错误。

珀杜（Purdue）大学的网际互联研究小组（Internetworking Research Group）的许多成员对程序代码的早期版本做过很多贡献。Andy Muckelbauer和Steve Chapin建立了一个UNIX兼容库，并与Shawn Ostermann和Scott Mark合作，使用TCP代码运行一个X window服务器。他们对TCP做了大量的测试工作，并指出其性能上存在的几个问题。Scott M. Ballew参与了一些软件的开发工作。

我的妻子Christine对本书手稿进行了编辑，并提出了许多建议。最后，我们感谢珀杜大学的计算机科学系和计算中心所提供的帮助。

Douglas E. Comer
David L. Stevens

目　　录

第 1 章　引言与概述 1
1.1　TCP/IP 协议 1
1.2　了解细节的必要性 1
1.3　协议间交互作用的复杂性 1
1.4　本书采用的方法 2
1.5　研究代码的重要性 2
1.6　Xinu 操作系统 2
1.7　本书其余部分的组织 3
1.8　小结 3
深入研究 4

第 2 章　操作系统中的 TCP/IP 软件结构 5
2.1　引言 5
2.2　进程的概念 5
2.3　进程的优先级 6
2.4　进程的同步通信 6
2.5　进程间通信 8
2.5.1　端口 8
2.5.2　消息传送 9
2.6　设备驱动程序和输入、输出程序 9
2.7　网络的输入和中断 10
2.8　向高层协议传递分组 11
2.9　IP 协议与传输协议之间的数据报传递 12
2.9.1　将传入的数据报发送给 TCP 12
2.9.2　将传入的数据报发送给 UDP 12
2.10　向应用程序的传递操作 13
2.11　输出时的信息流 13
2.12　从 TCP 经过 IP 到网络输出 14
2.13　UDP 输出处理 15
2.14　小结 15
深入研究 16
习题 16

第 3 章　网络接口层 18
3.1　引言 18

3.2 网络接口抽象模型 18
3.2.1 接口数据结构 18
3.3 以太网的基本定义 21
3.3.1 应用中的统计数据 24
3.4 接口的逻辑状态 24
3.5 本地主机接口 24
3.6 缓冲区管理 25
3.6.1 大缓冲区方案 25
3.6.2 链表方案（mbuf） 26
3.6.3 方案举例 26
3.6.4 有关缓冲区的其他议题 26
3.7 传入分组的多路分解 27
3.8 小结 28
深入研究 29
习题 29

第4章 地址的发现及绑定（ARP） 30
4.1 引言 30
4.2 ARP软件在理论上的结构 30
4.3 ARP设计方案举例 30
4.4 ARP高速缓存的数据结构 31
4.5 ARP输出处理 34
4.5.1 搜索ARP高速缓存 34
4.5.2 ARP请求分组的广播 35
4.5.3 输出过程 36
4.6 ARP输入处理 38
4.6.1 向表中增加已转换的表项 38
4.6.2 发送等待发送的分组 39
4.6.3 ARP输入过程 40
4.7 ARP高速缓存的管理 42
4.7.1 高速缓存表项的分配 42
4.7.2 高速缓存的定期维护管理 43
4.7.3 释放队列中的分组 44
4.8 ARP初始化 45
4.9 ARP参数配置 46
4.10 小结 46
深入研究 47
习题 47

第5章 IP：软件的总体结构 48
5.1 引言 48

5.2 中心环节 48
5.3 IP 软件设计思想 48
5.4 IP 软件结构和数据报流程 49
5.4.1 选择传入数据报的策略 49
5.4.2 允许 IP 进程被阻塞 51
5.4.3 IP 使用的常量的定义 54
5.4.4 校验和的计算 57
5.4.5 处理定向广播 57
5.4.6 识别一个广播地址 59
5.5 IP 首部中的字节顺序 60
5.6 向 IP 发送数据报 61
5.6.1 发送本地生成的数据报 61
5.6.2 发送传入数据报 63
5.7 表格的维护 63
5.8 小结 65
深入研究 65
习题 65

第 6 章 IP：选路表和选路算法 67
6.1 引言 67
6.2 路由维护和查找 67
6.3 选路表结构 67
6.4 选路表数据结构 68
6.5 路由的生成源及保持时间 70
6.6 为数据报选择路由 70
6.6.1 实用过程 70
6.6.2 获得一个路由 73
6.6.3 数据结构初始化 74
6.7 选路表的定期维护 75
6.7.1 增加路由 77
6.7.2 删除路由 80
6.8 IP 选项处理 82
6.9 小结 83
深入研究 83
习题 84

第 7 章 IP：分片与重组 85
7.1 引言 85
7.2 数据报的分片 85
7.2.1 为一个数据报片再次分片 85
7.3 分片的实现 85

7.3.1 发送一个数据报片 87
7.3.2 复制数据报首部 89
7.4 数据报的重组 90
7.4.1 数据结构 90
7.4.2 互斥操作 91
7.4.3 在链表中加入一个数据报片 91
7.4.4 溢出时的丢弃 93
7.4.5 测试一个完整的数据报 94
7.4.6 将数据报片组装成完整的数据报 96
7.5 数据报片链表的维护管理 97
7.6 初始化 99
7.7 小结 99
深入研究 100
习题 100

第8章 IP：差错处理（ICMP） 101
8.1 引言 101
8.2 ICMP报文格式 101
8.3 ICMP报文的实现 101
8.4 传入ICMP报文的处理 103
8.5 ICMP重定向报文的处理 105
8.6 设置子网掩码 107
8.7 为一个ICMP分组选择源地址 108
8.8 生成ICMP差错报文 109
8.9 避免出现关于差错报文的差错报文 111
8.10 为ICMP报文分配缓冲区 112
8.11 ICMP报文中的数据部分 114
8.12 ICMP重定向报文的生成 116
8.13 小结 117
深入研究 117
习题 117

第9章 IP：组播处理（IGMP） 119
9.1 引言 119
9.2 维护组播主机群的成员信息 119
9.3 主机群表 119
9.4 查找一个主机群 121
9.5 向主机群表中增加一个表项 122
9.6 为一个组播地址设置网络接口 124
9.7 IP组播地址和硬件组播地址之间的转换 125
9.8 从主机群表中删除一个组播地址 126

9.9 加入一个主机群 127
9.10 维持与一个组播路由器的联系 129
9.11 IGMP 成员关系报告的实现 130
9.12 计算随机时延 131
9.13 发送 IGMP 报告的进程 132
9.14 处理输入的 IGMP 报文 133
9.15 脱离主机群 134
9.16 IGMP 数据结构的初始化 136
9.17 小结 137
深入研究 137
习题 137

第 10 章　UDP：用户数据报 138
10.1 引言 138
10.2 UDP 端口和多路分解处理 138
10.2.1 成对通信使用的端口 138
10.2.2 多对一通信使用的端口 138
10.2.3 操作模式 139
10.2.4 多路分解处理中的细节问题 139
10.3 UDP 的输入处理 141
10.3.1 UDP 数据结构的说明 141
10.3.2 传入数据报队列的说明 142
10.3.3 UDP 端口号与队列的映射 144
10.3.4 分配空闲队列 144
10.3.5 网络字节顺序与本机字节顺序之间的相互转换 145
10.3.6 处理一个已到达的数据报 146
10.3.7 UDP 校验和的计算 148
10.4 UDP 输出的处理 149
10.4.1 一个 UDP 数据报的发送 150
10.5 小结 151
深入研究 152
习题 152

第 11 章　TCP：数据结构和输入处理 153
11.1 引言 153
11.2 TCP 软件概览 153
11.3 传输控制块 153
11.4 TCP 报文段格式 158
11.5 序列空间中的比较 159
11.6 TCP 有限状态机 160
11.7 状态变迁举例 160

11.8 有限状态机的说明 161
11.9 TCB的分配及初始化 163
11.9.1 分配一个TCB 163
11.9.2 释放一个TCB 164
11.10 有限状态机的实现 165
11.11 处理一个输入报文段 165
11.11.1 将TCP首部转换为本地字节顺序 167
11.11.2 计算TCP的校验和 168
11.11.3 为报文段查找对应的TCB 169
11.11.4 检查报文段的有效性 171
11.11.5 为当前状态选择一个过程 172
11.12 小结 173
深入研究 173
习题 173

第12章 TCP：有限状态机的实现 175
12.1 引言 175
12.2 CLOSED状态处理 175
12.3 从容关闭 175
12.4 关闭后的延迟计时 176
12.5 TIME-WAIT状态处理 177
12.6 CLOSING状态处理 178
12.7 FIN-WAIT-2状态处理 179
12.8 FIN-WAIT-1状态处理 180
12.9 CLOSE-WAIT状态处理 181
12.10 LAST-ACK状态处理 182
12.11 ESTABLISHED状态处理 183
12.12 处理报文段中的紧急数据 184
12.13 处理报文段中的其他数据 186
12.14 经常注意已接收的八位组 188
12.15 终止一个TCP连接 190
12.16 建立TCP连接 191
12.17 初始化TCB 191
12.18 SYN-SENT状态处理 193
12.19 SYN-RECEIVED状态处理 194
12.20 LISTEN状态处理 196
12.21 为一个新TCB初始化窗口变量 197
12.22 小结 199
深入研究 199
习题 199

第 13 章　TCP：输出处理 200
13.1　引言 200
13.2　TCP 输出控制的复杂性 200
13.3　TCP 输出的四种状态 200
13.4　作为一个进程的 TCP 输出 201
13.5　TCP 输出报文 201
13.6　对输出状态和 TCB 编号的编码 202
13.7　TCP 输出进程的实现 202
13.8　互斥操作 203
13.9　IDLE 状态的实现 203
13.10　PERSIST 状态的实现 204
13.11　TRANSMIT 状态的实现 205
13.12　RETRANSMIT（重发）状态的实现 206
13.13　发送一个报文段 206
13.14　计算 TCP 数据长度 210
13.15　序号计算 210
13.16　其他 TCP 过程 211
13.16.1　发送复位 211
13.16.2　转换成网络字节顺序 213
13.16.3　等待输出缓冲空间 213
13.16.4　唤醒等待 TCB 的进程 214
13.16.5　选择初始序号 216
13.17　小结 217
深入研究 217
习题 217

第 14 章　定时器管理 218
14.1　引言 218
14.2　定时事件的通用数据结构 218
14.3　TCP 事件使用的数据结构 219
14.4　定时器、事件和报文 220
14.5　TCP 定时器进程 220
14.6　删除 TCP 定时器事件 222
14.7　删除一个 TCB 的所有事件 223
14.8　确定出现一个事件的尚需时间 224
14.9　插入 TCP 定时器事件 225
14.10　启动无时延的 TCP 输出 227
14.11　小结 227
深入研究 228
习题 228

第15章 流量控制和自适应重发 229
15.1 引言 229
15.2 自适应重发中的难题 229
15.3 自适应重发的调整 229
15.4 重发定时器和退避 230
15.4.1 Karn算法 230
15.4.2 重发输出状态的处理 230
15.5 基于窗口的流量控制 231
15.5.1 糊涂窗口综合症 232
15.5.2 接收方预防糊涂窗口 232
15.5.3 零窗口之后的性能优化 233
15.5.4 调整发送方的窗口 233
15.6 最大报文段长度的计算 235
15.6.1 发送方的最大报文段长度 235
15.6.2 选项处理 236
15.6.3 通告一个最大输入报文段长度 238
15.7 网络拥塞预防与控制 239
15.7.1 成倍递减法 239
15.8 慢启动和拥塞预防 239
15.8.1 慢启动 239
15.8.2 超过上限后减速递增 240
15.8.3 递增拥塞窗口尺寸的实现 240
15.9 平均往返时间估值及超时 242
15.9.1 一种快速平均值更新算法 242
15.9.2 传入确认的处理 243
15.9.3 为窗口外的数据创建确认报文段 245
15.9.4 接收到一个确认后改变输出状态 246
15.10 技巧和注意事项 247
15.11 小结 247
深入研究 248
习题 248

第16章 紧急数据处理和推功能 250
16.1 引言 250
16.2 带外信令 250
16.3 紧急数据 250
16.4 标准的解释 250
16.4.1 带外数据的解释法 251
16.4.2 数据标记解释法 252
16.5 为Berkeley紧急指针解释法而进行的配置 252
16.6 通知应用程序 253

16.6.1 多个并发应用程序 253
16.7 从 TCP 中读取数据 254
16.8 发送紧急数据 255
16.9 TCP 的推功能 256
16.10 在失序交付时对推数据的解释 257
16.11 输入时推功能的实现 257
16.12 小结 258
深入研究 258
习题 259

第 17 章 套接层的接口 260
17.1 引言 260
17.2 通过设备形成的接口 260
17.2.1 单字节 I/O 261
17.2.2 其他一些非传送的函数 261
17.3 作为设备的 TCP 连接 262
17.4 TCP 客户程序举例 262
17.5 TCP 服务器程序举例 263
17.6 TCP 主设备的实现 265
17.6.1 TCP 主设备打开功能 265
17.6.2 被动 TCP 连接的形成 266
17.6.3 主动 TCP 连接的形成 267
17.6.4 分配一个未使用的本地端口 268
17.6.5 主动连接的完成 269
17.6.6 TCP 主设备的控制 271
17.7 TCP 从设备的实现 271
17.7.1 由 TCP 从设备输入数据 271
17.7.2 由 TCP 从设备输入单字节 273
17.7.3 通过 TCP 从设备的输出 274
17.7.4 TCP 连接的关闭 276
17.7.5 TCP 从设备的控制操作 278
17.7.6 接受来自被动设备的连接 279
17.7.7 改变 LISTEN 队列的长度 279
17.7.8 获取从设备中的统计数据 280
17.7.9 设置或清除 TCP 选项 282
17.8 从设备的初始化 283
17.9 小结 284
深入研究 284
习题 284

第18章 RIP：主动路由传播和被动获取 286
18.1 引言 286
18.2 主动和被动模式的参与者 286
18.3 基本的RIP算法和费用度量 286
18.4 不稳定性及解决方案 287
18.4.1 计数到无穷大 287
18.4.2 网关瘫痪和路由超时 288
18.4.3 水平划分 288
18.4.4 毒性逆转 288
18.4.5 具有毒性逆转的路由超时 289
18.4.6 触发更新 289
18.4.7 随机化以防止广播风暴 289
18.5 报文类型 290
18.6 协议特性 290
18.7 RIP的具体实现 291
18.7.1 实现的两种形式 291
18.7.2 定义 291
18.7.3 输出的理论结构 293
18.8 基本RIP进程 294
18.8.1 "必须为零"的字段内容必须为零 295
18.8.2 处理一个传入响应 296
18.8.3 在更新期间的锁定 298
18.8.4 验证一个地址 298
18.9 对输入请求的响应 299
18.10 生成更新报文 300
18.11 初始化一个更新报文的副本 301
18.11.1 向更新报文的副本中添加路由 302
18.11.2 计算一个待通告的费用值 304
18.11.3 为RIP报文分配数据报 305
18.12 生成定期的RIP输出 306
18.13 RIP的局限性 307
18.14 小结 307
深入研究 307
习题 307

第19章 OSPF：使用SPF算法的路由传播 308
19.1 引言 308
19.2 OSPF配置和选项 308
19.3 OSPF的图论模型 308
19.4 OSPF的说明 311
19.4.1 OSPF分组格式的说明 311

19.4.2 OSPF 接口说明 313
19.4.3 全局常量和数据结构的说明 314
19.5 邻接关系和链路状态信息的传播 316
19.6 用 Hello 发现相邻网关 317
19.7 Hello 分组的发送 319
19.7.1 Hello 分组的一个模板 320
19.7.2 Hello 分组输出进程 321
19.8 指定路由器 323
19.9 选出一个指定路由器 324
19.10 变动之后重建邻接关系 327
19.11 处理到达的 Hello 分组 329
19.12 在相邻网关表中增加一个网关 331
19.13 相邻网关状态的变迁 332
19.14 OSPF 定时器事件和重发 333
19.15 判断是否允许邻接关系 335
19.16 OSPF 输入的处理 336
19.17 链路状态处理中的说明和过程 339
19.18 数据库描述分组的生成 341
19.19 创建一个模板 343
19.20 传送数据库描述分组 344
19.21 处理到达的数据库描述分组 345
19.21.1 处理 EXSTART 状态下的分组 347
19.21.2 处理 EXCHNG 状态下的分组 348
19.21.3 处理 FULL 状态下的分组 349
19.22 处理链路状态请求分组 350
19.23 建立链路状态概要信息 352
19.24 OSPF 实用过程 354
19.25 小结 356
深入研究 357
习题 357

第 20 章 SNMP：MIB 变量、表示形式和绑定 358
20.1 引言 358
20.2 服务器的组织和名字的映射 358
20.3 MIB 变量 359
20.3.1 表格中的字段 359
20.4 MIB 变量名 360
20.4.1 变量名的数字表示形式 360
20.5 名字之间的字典顺序 360
20.6 除去前缀 361
20.7 在 MIB 变量上执行的操作 361

20.8 表格名 362
20.9 名字体系概念上的线索 362
20.10 MIB 变量的数据结构 363
20.10.1 使用独立的函数完成操作 365
20.11 用于快速查找的数据结构 365
20.12 散列表的实现 366
20.13 MIB 绑定的描述 366
20.14 绑定中使用的内部变量 373
20.15 散列表的查找 374
20.16 SNMP 的结构和常量 376
20.17 ASN.1 表示形式的处理 382
20.17.1 长度表示法 382
20.17.2 将整数转换为 ASN.1 格式 384
20.17.3 将对象标识符转换为 ASN.1 格式 386
20.17.4 用于转换对象值的例程 389
20.18 小结 391
深入研究 391
习题 392

第 21 章 SNMP：客户与服务器 393
21.1 引言 393
21.2 服务器中数据的表示形式 393
21.3 服务器的实现 393
21.4 对 SNMP 报文的分析 396
21.5 绑定链表中 ASN.1 名字的转换 400
21.6 解析一个请求 401
21.7 对 get-next 操作的解释 404
21.8 操作的间接执行 404
21.9 表格的间接寻址 407
21.10 应答报文的反向生成 408
21.11 将内部格式转换为 ASN.1 表示形式 410
21.12 服务器使用的实用函数 412
21.13 一个 SNMP 客户的实现 413
21.14 变量初始化 415
21.15 小结 417
深入研究 417
习题 417

第 22 章 SNMP：表格访问函数 419
22.1 引言 419
22.2 表格访问 419

22.3 表格的对象标识符 419
22.4 地址入口表函数 420
22.4.1 对地址入口表的 get 操作 421
22.4.2 对地址入口表的 get-first 操作 423
22.4.3 对地址入口表的 get-next 操作 424
22.4.4 地址入口表中的递增搜索 425
22.4.5 对地址入口表的 set 操作 426
22.5 地址转换表函数 426
22.5.1 对地址转换表的 get 操作 428
22.5.2 对地址转换表的 get-first 操作 429
22.5.3 对地址转换表的 get_next 操作 431
22.5.4 地址转换表中的递增搜索 432
22.5.5 乱中有序 433
22.5.6 对地址转换表的 set 操作 434
22.6 网络接口表的函数 435
22.6.1 接口表标识符的匹配 435
22.6.2 对网络接口表的 get 操作 436
22.6.3 对网络接口表的 get-first 操作 440
22.6.4 对网络接口表的 get-next 操作 441
22.6.5 对网络接口表的 set 操作 442
22.7 选路表函数 443
22.7.1 对选路表的 get 操作 444
22.7.2 对选路表的 get-first 操作 446
22.7.3 对选路表的 get-next 操作 447
22.7.4 选路表中的递增搜索 449
22.7.5 对选路表的 set 操作 450
22.8 TCP 连接表函数 452
22.8.1 对 TCP 连接表的 get 操作 453
22.8.2 对 TCP 连接表的 get-first 操作 455
22.8.3 对 TCP 连接表的 get-next 操作 456
22.8.4 TCP 连接表中的递增搜索 457
22.8.5 对 TCP 连接表的 set 操作 458
22.9 UDP Listener 表 460
22.9.1 对 UDP Listener 表的 get 操作 461
22.9.2 对 UDP Listener 表的 get-first 操作 462
22.9.3 对 UDP Listener 表的 get-next 操作 463
22.9.4 UDP Listener 表中的递增搜索 464
22.9.5 对 UDP Listener 表的 set 操作 465
22.10 IP 地址转换的实用例程 466
22.11 小结 467

深入研究 467
习题 468

第23章 实现的回顾 469
23.1 引言 469
23.2 程序代码统计分析 469
23.3 各个协议的程序代码行数 469
23.4 每个协议所需的函数和过程 470
23.5 小结 471
习题 471

附录1 过程调用交叉参考表 473

附录2 程序代码中使用到的C数据结构交叉参考表 493

附录3 程序代码中使用到的Xinu函数和常量 498

参考文献 513

第1章 引言与概述

1.1 TCP/IP 协议

TCP/IP互联网协议族在全世界范围内已经成为开放系统互联的标准。与其他任何协议相比，TCP/IP互联网协议能够提供更好的交互操作性能，可以在绝大多数销售商的系统中使用，并且它可兼容多种网络技术，在这一点上，它比其他任何一个协议族要强。Internet将全世界范围内的学校、政府部门、盈利性或非盈利性组织以及个人都联系在一起。

除了常规性地在专用工业网上使用TCP/IP外，许多组织也使用TCP/IP协议，通过与Internet连接，相互通信。内联网（intranet）是使用TCP/IP协议的专用互联网。尽管多数组织都没有公开他们的内联网的情况，但是许多已经进行了广泛深入的应用。例如，在航空工程、汽车制造、电子技术、宾馆服务、石油化工、出版印刷、药剂制造等众多领域中已经发展了具有多国站点的内联网。这些组织利用TCP/IP将广域网和局域网链接起来。

1.2 了解细节的必要性

尽管TCP/IP非常流行并得到了广泛应用，但对于大多数计算机专家来说，TCP/IP协议的细节，以及实现它们的软件的结构仍然是一个谜。从表面上看，是否理解TCP/IP的内部细节并不重要，但是，使用TCP/IP的程序员们都知道，如果能够完全了解协议是如何运作的，则能编写出更健壮的程序代码。譬如说，一个了解TCP紧急数据处理过程的程序员可以在自己的应用程序中增加更多的功能，而那些不了解此细节的程序员则无法做到。

即使了解一些简单的概念，诸如TCP是如何对数据进行缓冲的，也会对程序员在设计、实施和调试应用程序时有所帮助。举个例子，如果程序员不了解输出缓存、报文段传送、输入缓存和TCP推数据操作之间的关系，那么他就无法成功地编制出利用TCP协议的程序。研究TCP输入输出的细节，能够使程序员形成一个概念模型，用于解释各部分是如何交互作用的，并有助于理解如何使用底层机制。

1.3 协议间交互作用的复杂性

人们普遍认为，TCP/IP技术难以理解的一个主要原因是，有关的文章往往单独地讨论每一个协议，而没有去考虑多个协议是如何相互配合协同工作的。例如，各个协议标准文档往往只描述一个协议应当如何工作，它脱离系统的其他部分，孤立地讨论某个协议的行为以及对报文的响应。其实，这些协议最难理解的部分，正是它们之间的交互作用。当我们将所有协议的运作联系在一起考虑时，他们之间的交互会产生复杂的、有时甚至是不可预料的后果。那些原本看来并不重要的微小细节会突然变得非常重要。处理问题的启发性思考和协议设计上的细微差别将给整个操作或性能带来巨大的差异。

正如许多程序员已经发现的那样，协议之间的交互作用经常会决定它们的实现策略。选择数据结构时必须将所有的协议考虑在内。例如，IP通过选路表来决定如何继续向前发送数据报，但是，在选择路由表的数据结构时不能不考虑选路信息协议（RIP）、Internet控制报文协议（ICMP）和外部网关协议（EGP），因为所有这些协议都可能会修改选路表中的路由。更重要的是，选路表的更新原则必须认真选取，以保证这些协议或是协议之间的交互作用不会导致意外的结果。因此我们可以说：

> TCP/IP技术中包含了多个协议，这些协议之间相互影响。为了完全理解一个协议的细节及其实现方案，我们必须考虑到它与协议族中的其他协议之间的交互作用。

1.4 本书采用的方法

本书详细剖析TCP/IP协议，讲解有关的概念，并对每一协议的细节问题都做了阐释。我们在本书中将会对隐含在TCP/IP软件内的抽象概念进行讨论，并给出协议实现方案中用到的数据结构和过程。本书还分析了可选的设计方案，并讨论了不同设计方案的效果。

为了提供协议的实现范例，并帮助读者理解协议之间的关系，本书采用了整体考察法，我们将重点放在一个完整的工作系统上。书中给出了范例系统的数据结构和源代码，并且对其中隐含的一些原则加以说明。

范例系统的源代码不但有助于我们解答许多疑问，而且还可说明许多难以说明的细枝末节。源代码起到补充细节的作用，使读者了解协议中各部分在实现时的相对难易程度。这些源代码说明了选择合适的数据表达形式将会简化协议的实现（同时也从反面说明了不良的选择增加了实现的难度，使软件变得冗长不堪）。范例代码还有助于读者理解紧急数据的处理以及网络管理的概念；在程序代码中，有许多地方体现了这些概念。最重要的是，范例系统清楚地告诉了读者协议之间是如何交互影响的以及单个的协议实现是如何被集成起来的。总而言之：

> 为了解释TCP/IP的细节、内部组织形式及其实现，本书的重点是对一个范例工作系统加以研究。该范例系统的源代码使读者能够理解协议之间的交互作用，并能够理解这些软件是如何被集成为一个简单而有效的软件系统的。

1.5 研究代码的重要性

TCP/IP范例系统是本书的主体部分。为了理解数据结构、过程中的交互作用以及协议内部的细节，阅读和研究源代码[1]是十分必要的，因此：

> 范例系统应当被认为是本书内容的组成部分，而不仅仅是本书内容的辅助部分。

1.6 Xinu操作系统

在大多数机器上，TCP/IP协议软件位于操作系统的内核中。所有的应用程序共同使用一个单一

---

① 为便于实现范例工作系统，出版者允许我们将本书中的程序代码制作成机读的副本。本章末尾给出了一个URL。

的TCP/IP软件。本书中的TCP/IP软件是Xinu操作系统[1]的一部分。我们之所以选用Xinu操作系统有以下几个原因。第一，有两本介绍Xinu的教材已经出版，因此在学习过程中，完全可以利用Xinu操作系统的所有源代码。第二，由于Xinu没有过多的费用计算或其他诸如此类的管理模块，因而Xinu中的TCP/IP代码可以不涉及那些不必要的细节，也就更容易理解。第三，由于本书着重解释程序代码中隐含的抽象概念，其中大多数思想可以直接适用于其他一些实现。第四，使用我们自己编写的Xinu和TCP/IP软件可以完全避免商业许可的问题，使我们能够自由地销售本书。尽管我们提供的Xinu操作系统和本书的TCP/IP程序代码源自于一个研究课题，但读者将惊奇地发现这些软件已经非常完整，并且在大多数情况下，比商业应用中的同等软件功能更强。最后，由于我们一直试图严守RFC规范，读者可能会发现我们的TCP/IP的Xinu实现方案比许多流行的实现方案更严格地遵守了协议标准。

1.7 本书其余部分的组织

本书的内容是根据TCP/IP协议族中的各个模块组织编写的，顺序基本与第一卷相同。它首先回顾了TCP使用的操作系统函数，接下来介绍设备接口层。在此之后，其余章节叙述TCP/IP的各个协议，并给出范例代码以说明每一部分的具体实现过程。

某些章节描述整个协议族，而其他一些章节将重点放在设计的具体方面。例如，第15章探讨了平均往返时延的估算、重发以及指数形式缩小窗口的思想。有些程序代码可能会在不同的章节中被调用，而它们将出现在与内容联系最紧密的章节中。

附录1中包含了一张有关过程的交叉参考表，这些过程组成了我们在本书中讨论的TCP/IP协议软件。对于每一个过程、函数或是内联宏，交叉参考表说明了它们在哪些文件中定义，出现在本书的第几页，该文件调用的过程的列表，以及调用该文件的过程的列表。交叉参考表在查找某个过程的调用关系时特别有用，因为有时候仅从一段程序代码中并不能看出什么来。附录2提供了一个类似的主要数据结构的交叉参考表，这个交叉参考表对于代码中条目的定位非常有用。

附录3提供了一张函数和过程的列表，这些被调用的函数或过程代码并没有包括在本书中。大部分列出的过程来自于C运行库或底层的操作系统，其中包括在TCP/IP代码中出现的Xinu操作系统调用。对于每一个过程和函数，附录3给出了名字和参数，并对其执行做了简单描述。

1.8 小结

本书探讨了TCP/IP协议的细节、实现的具体过程以及实现时所使用的软件的内部结构。它将重点放在一个基于Xinu操作系统的实现范例上，包括组成该工作系统的源代码。尽管Xinu的实现方案并没有作为一种商业产品来设计，但它依然遵循协议标准。为了透彻理解协议，读者必须仔细研究范例程序。附录用来帮助读者理解程序代码。在附录中提供了TCP/IP实用程序的交叉参考表及一个涉及到的操作系统例程表。

---

① Xinu是一个小巧、精练的操作系统，它在很多性能上与UNIX相似。一些销售商已经将Xinu的几个版本作为其商业产品的内嵌系统提供给客户。

深入研究

第一卷[Comer 1995]描述了支持TCP/IP协议族的概念基础、每个协议的大致情况，并对Internet结构做了简单总结。这里我们假定读者已经对第一卷的大部分内容比较熟悉。Comer[1984]和Comer[1987] 描述了 Xinu 操作系统的结构，其中包括 ARP、 UDP 和 IP 程序代码的早期版本。Leffler、McKusick、Karels 和 Quarterman [1996] 描述了 Berkeley UNIX 系统。Stevens [1998] 提供了在多种操作系统环境下使用 TCP/IP 接口的样例。

本书中的代码已被应用到某些商业产品中。下面的 URL 提供了机读副本：

ftp://ftp.cs.purdue.edu/pub/comer/TCPIP-vol2.dist.tar.Z

第2章　操作系统中的TCP/IP软件结构

2.1　引言

在绝大多数运行TCP/IP软件的计算机中都利用一个操作系统来管理资源，比如管理某些外部设备。操作系统提供对并行处理的支持。即使在单处理器的计算机上，操作系统也能够通过迅速切换CPU的服务对象使用户感觉多个程序在同时执行。另外，操作系统管理着主内存，其中存放了正在执行的程序，同时它还管理着存放文件系统的二级（非易失）存储器。

TCP/IP软件一般位于操作系统内部，并且可以被在机器上运行的所有应用程序所共享。因此，虽然多个程序可以同时唤醒像TCP这样的程序代码段，但在操作系统中，该程序代码段是惟一的。当然，每个唤醒过程必须是独立的，这样一个程序的数据传输就不会影响到另一个程序。我们还会看到，可以被多个并发执行的程序使用的代码段无疑要比单程序中的代码段复杂得多。

本章简单介绍了操作系统中的一些概念，全书会经常用到这些概念。本章描述了协议软件的一般结构，并用简单的术语介绍了如何将协议软件嵌入操作系统中。在此后的章节中将分别探讨协议软件的各个组成部分，并给出大量的协议细节。

本章所举的例子都是基于Xinu操作系统的，事实上，我们自始至终都以Xinu操作系统为基础。尽管范例所涉及的系统调用和参数只在Xinu操作系统上提供，但是这些概念适用于大多数操作系统，包括非常流行的UNIX分时系统以及像Windows 95这样的商用系统。

2.2　进程的概念

在操作系统中有几个抽象概念，这些概念在理解TCP/IP协议的实现时要用到。其中最重要的应该是“进程”或者说“控制线程”[1]。从概念上讲，一个进程就是一次计算过程，并且它独立于其他计算过程。一个操作系统提供创建新进程和结束现有进程的机制。在我们将要使用的范例系统中，程序调用函数create创建一个新进程。create函数返回一个整型的进程标识号。在对进程操作时，利用进程标识号来引用该进程。

```
procid=create(arguments);     /* create a new process */
```

进程一旦被创建，就独立于它的创建者而工作。若要结束一个进程，则需要在程序中调用kill，并将create函数返回的进程标识号作为参数，传递给kill。

```
kill(procid);                 /* terminate a process */
```

常规（顺序执行）程序根据单一的控制线程，逐步执行属于某个“程序”的代码。与常规程序不同，进程并不局限于任何特定的代码段或数据。操作系统允许两个或更多的进程执行同一段代码。

---

① 尽管进程与控制线程在技术上是截然不同的，但在操作系统中运行并发进程时，这两个术语可以互换。

例如，虽然操作系统的代码只有一份，但两个进程可以并发地执行操作系统的代码。实际上，两个或更多的进程并发执行某个过程中的特定代码段是完全可能的。

由于进程是独立执行的，因此它们执行的速度有快有慢。在特定情况下，进程有时会因为执行某些操作而导致被“阻塞”或被“挂起”。例如，如果一个进程试图从键盘上读入一个字符，则它可能要等待用户按下键盘上的一个键。为了避免此类正在等待的进程占用CPU，操作系统阻塞该进程而允许其他进程继续执行。稍后，当操作系统接收到一个键盘敲击事件后，它将允许这个等待键盘输入的进程“恢复”执行。

在我们所要研究的TCP/IP软件的实现方案中使用了多个并发执行的进程。利用进程可将软件划分成更小的、更容易管理的模块，而不用编写一个能够处理全部可能事件序列的程序。我们将看到，使用进程可以简化设计，并使程序代码更容易理解和修改。

进程在处理存在于许多协议中的超时和重发机制时尤其有用。使用单一的程序解决多个协议中的超时问题将导致程序复杂化，因为超时有可能重叠。例如，假设我们试图编写一个能够管理所有TCP/IP协议中的定时器的程序。某个高层协议，例如TCP，可能会产生一个报文段，并将其封装在数据报中，然后发送该数据报，并启动定时器。同时，IP必须为此数据报选择路由，并将其传送到网络接口。最后，一个诸如ARP的低层协议将被唤醒，它可能会重复多次地循环执行传送请求、设置定时器、处理定时器溢出等动作，此时的重发请求与TCP的定时器无关。在一个单一的程序中，当某个协议的定时器溢出，而程序却正在执行另一个协议的代码时，其处理过程是很困难的。如果系统对每一个要求超时机制的协议使用单独的进程来处理，那么进程只需处理与该协议有关的超时事件，因而使进程的代码更容易理解，不易出错。

2.3 进程的优先级

我们曾说过，所有的进程并发执行，但这种说法过于简单化。事实上，设计软件的程序员给每一个进程都分配有一个“优先级”。操作系统在给各进程分配CPU使用权时参照各进程的优先级。我们采用的优先级分配策略简单易懂：CPU对没有被阻塞的进程赋予最高优先级。如果多个进程具有相同的优先级别，那么CPU将在这几个进程之间快速切换。

协议软件中的优先级策略是非常有用的，因为它允许程序员赋予某进程在执行时优先于其他进程的权力。例如，现有一个普通的应用程序和一个协议软件，该协议软件必须从硬件中接收抵达的分组。设计者可以给实现该协议的进程以高优先级，迫使它先于应用程序的进程执行。由于操作系统处理了所有的进程调度细节，进程本身可以不包含任何有关调度的代码。

2.4 进程的同步通信

如果每个进程都是一个独立的计算过程，那么数据是怎样从一个进程传递到另一个进程的呢？答案是：操作系统必须提供允许进程之间相互通信的机制。我们使用以下三种机制：计数信号量（semaphore）、端口和消息传递。

通常，计数信号量就是通用的进程同步机制。操作系统提供一个函数screate，如有必要，调用它以创建信号量。screate返回一个信号量标识号，在此之后的有关该信号量的操作中必须用到这个信号量标识号。

```
semid=screate(initcount);  /* create semaphore, specifying count */
```

每个信号量包含一个计数用的整数，调用者在创建信号量时赋予该整数一个初始值。一旦信号量产生，进程可以通过调用操作系统的 wait 和 signal 函数来管理信号量。当进程调用 wait 时，操作系统对信号量的计数值减 1，如果计数值变为负值，则进程被阻塞。当进程调用 signal 函数时，操作系统将信号量的计数值加 1，如果有任何一个进程因该信号量而被阻塞的话，此时这个进程将被重新激活。

尽管 wait 和 signal 从字面上看来可能使人产生误解，但实际上它们可以帮助解决几个重要的进程同步问题。最重要的是，它们可以用来提供“互斥”机制。互斥意味着在某段时间内，只允许一个进程执行一段特定代码。由于多个进程可以并发地执行同一段代码，因此互斥机制非常重要。要理解互斥机制的必要性，只需考虑一下如果两个向链表中增加新表项的进程同时执行将会带来什么后果。如果两个进程并发执行，它们可能在链表的同一个地方开始，并试图插入新表项。根据每个进程占用 CPU 时间长度的不同，第一个进程执行一段时间，然后切换到另一个进程继续执行，接着又一次切换到第一个进程，以此类推。其结果是，一个进程可能会推翻另一个进程的工作（使其中一个新表项完全丢失），或者可能会产生一个含有不正确指针的错误链表。

为了防止进程之间相互干扰，所有允许多个进程并发执行的协议软件必须使用信号量来实现互斥操作。要做到这一点，程序员必须为所有应受保护的代码段提供一个信号量，其初始值均为 1。

```
s = screate(1);              /* create mutual exclusion semaphore */
```

然后，程序员用 wait 和 signal 语句将关键代码段围起来，如下所示：

```
wait(s);                    /* before code to be protected */
...critical code...
signal(s);                  /* after code to be protected */
```

第一个进程执行到 wait(s)时，将信号量 s 的计数值减至 0，并继续执行（由于计数器此刻不为负值）。如果进程结束，则会执行 signal(s)，使 s 的计数值回到 1。然而，如果第一个进程正在使用关键代码时，第二个进程调用 wait(s)，计数器值变为负值，因此第二个进程将被阻塞。类似地，如果此时碰巧还有第三个进程执行 wait(s)，计数器仍然为负值，则第三个进程也将被阻塞。当第一个进程使用关键代码完毕后，它将调用 signal(s)，给计数器加 1，唤醒第二个进程。第二个进程执行关键代码，此时第三个进程仍然在等待。当第二个进程结束并执行了 signal(s)后，第三个进程才能够开始使用关键代码。其中心思想是任何时候只能有一个进程执行关键代码，所有其他试图执行相同关键代码的进程都将被信号量阻塞。

除了提供互斥机制以外，本书的范例中还利用信号量来保持队列访问的同步。由于队列的容量有限，因此队列的同步也是必要的。假设一个队列空间可以容纳 N 个数据项，有多个并发进程要将它们生成的数据置入该队列中。与此同时我们假设另外一组进程将从队列中提取数据并处理它们（一般来讲，是多个进程插入，一个进程提取）。通常往队列中插入数据的进程称为“生产者”，而从队列中提取数据的进程称为“消费者”。例如，这些数据可能是由一组用户应用程序产生的 IP 数据报，而一个 IP 进程提取这些数据报、选择路由并将它们送往相应的目的站。如果应用程序产生数据报的速度比 IP 进程处理这些数据报的速度快，那么队列最终将被装满。在队列全满的情况下，任何试图插入新元素的生产者必须被阻塞，以等待消费者从队列中取出一个数据项，使队列腾出一个空间。同样，如果消费者执行迅速，它可能会将队列中的所有数据全部取出，那么此时它必须被

阻塞，以等待另一个数据的到来。在访问一个长度为N的队列时，需要两个信号量来调度生产者和消费者。这两个信号量初始化如下：

```
s1 = screate(N);                /* counts space in queue */
s2 = screate(0);                /* counts items in queue */
```

信号量被初始化以后，生产者和消费者利用它们来协调同步。一个生产者的执行如下：

```
wait(s1);                       /* wait for space */
...insert item in next available slot...
signal(s2);                     /* signal item available */
```

而消费者执行：

```
wait(s2);                       /* wait for item in queue */
...extract oldest item from queue...
signal(s1);                     /* signal space available */
```

信号量保证了当队列全满时生产者进程被阻塞，而当队列全空时，消费者进程被阻塞。除此之外，生产者和消费者可以正常运行。

2.5 进程间通信

2.5.1 端口

在Xinu操作系统中，“端口”（port）这个抽象模型把进程的数据集中传送到一点处理。我们把端口看成是一个有限的消息队列，该消息队列有两个控制访问的信号量。程序调用pcreate函数生成一个端口，其入口参数指定队列的大小。pcreate返回一个端口标识号，用于引用该端口。

```
portid=pcreate(size);       /* create a port specifying size */
```

生成一个端口后，进程可以调用过程psend和preceive来置入和取出端口中的数据项。psend过程向端口发送消息。

```
psend(portid, message);     /* send a message to a port */
```

psend需要两个入口参数：一个端口标识号和一个要发送的单字消息（在TCP/IP代码中，消息经常是一个指针，指向某一分组）。

preceive从端口中提取一个消息。

```
message=preceive(port);     /* extract next message from port */
```

就像我们建议的那样，在我们的实现方案中使用了信号量机制，如果端口全满，则调用psend的进程将被阻塞，而如果端口全空，则调用preceive的进程将被阻塞。一旦某个进程由于调用psend过程而被阻塞，那么它将保持阻塞状态，直至另一个进程调用了preceive为止。反之亦然。因此，在设计用到端口的进程体系时，程序员必须保证该体系不会使进程永远被阻塞（这正如同要求程序

员在编制顺序执行软件时，必须小心地避免死循环一样）。

除了避免由于进程间的交互作用而使进程永远被阻塞之外，对某些设计的要求也极为严格。它们要求某些进程无论在什么情况下都不应当被阻塞，即使是很短暂的阻塞时间也不行。如果这些特定的进程被阻塞，系统就无法正常工作。例如，在一个网络系统中，可能会需要网络传入进程永远不被阻塞，以避免某些应用程序在停止接收传入分组的情况下会导致整个系统瘫痪。在这种情况下，进程需要判断调用 psend 过程会不会导致进程被阻塞。如果会，进程将采取变更方案（例如：丢弃该分组）。

为了使进程能够判断调用 psend 是否会导致阻塞，系统提供了一个函数：pcount，它允许进程检测一个端口是否已满。

```
n = pcount(portid);  /* find out whether a port is full */
```

进程调用pcount，将待检测的端口标识作为入口参数。pcount返回该端口内现有的数据项计数。如果返回值为 0，则端口为空。如果返回值与端口的大小相等，则端口全满。

2.5.2　消息传送

我们说过，进程还可以通过消息传送来实现相互通信以及传送同步。消息传送允许一个进程直接将消息发送给另外一个进程。进程通过调用 send 函数将消息发送给其他进程。send 函数以一个进程标识号和一个消息作为入口参数，并将给定的消息发送到指定的进程。

```
send(msg, pid);          /* send integer msg to process pid */
```

一个进程调用 receive 来等待某个消息的到达。

```
message = receive( );       /* wait for msg and return it */
```

在系统中，调用 receive 的进程将被阻塞，直至某个消息到达，而调用 send 函数的进程总是继续运行。如果在两次连续的 send 调用之间没有进程调用 receive 函数以接收消息，那么第二次调用 send 函数时将返回 SYSERR，而该消息将无法发送。构造一个良好的系统以保证消息不会丢失的工作由程序员来完成。为了协助消息交换的同步，程序可以调用函数 recvclr，该函数删除所有正在等待被接收的消息，但并不阻塞进程。

```
message = recvclr( );       /* clear message buffer */
```

由于大多数协议规定了等待确认的最长时间，因此在程序中经常会使用消息传送函数recvtim，该函数是 receive 的翻版，但它允许调用者指明等待的最长时间。如果消息在此上限时间内到达，recvtim 将其返回给调用者。否则，revtim 返回一个特定代码 TIMEOUT。

```
message = recvtim(50);      /* wait 5 seconds (50 tenths of a     */
                            /* second) for a mesg and return it   */
```

2.6　设备驱动程序和输入、输出程序

网络接口硬件将网络上传送来的分组送入计算机内存中，并通知操作系统一个分组已经到达。

通常，网络接口使用“中断”机制来完成这一任务。一个中断使CPU暂时将正常处理挂起，跳转到称为“设备驱动程序”（device driver）的代码段执行。此时由设备驱动程序管理操作的所有细节。例如，它要重置硬件中断，并且（可能）使网络接口硬件继续工作，准备接收下一个分组。设备驱动软件同时通知协议软件已有一个分组到达，并要求它马上处理。当设备驱动程序完成了这些繁琐的工作后，它将从中断中返回，CPU继续从中断发生处往下执行。因此，我们可以认为中断暂时“借用”了一下CPU来处理I/O事务。

与许多操作系统相同，当一个分组到达时，Xinu操作系统安排网络接口设备中断处理器的正常工作。由设备驱动程序代码处理该中断，并清理设备，使它可以继续接收下一个分组。

设备驱动程序同时还要为发送或接收该分组的程序提供一个方便的端口。实际上，它允许进程等待传入的分组（此时被阻塞）。从进程的观点来看，设备驱动程序隐藏在常规I/O接口背后，使传入分组的捕获更加容易。例如，要向以太网的一个接口发送一个帧（分组），程序调用如下所示：

```
write(device, buff, len);  /* write one Ethernet packet */
```

其中device是设备描述符，指定某个以太网接口设备，buffer给出将要传送的帧所处的缓冲区地址，len是以八位组（octet）计数的帧长度[1]。

2.7 网络的输入和中断

既然我们已经了解了由操作系统提供的各种设施，不妨来研究一下TCP/IP范例软件的整体结构。回想一下，在操作系统中包含有设备驱动软件，该软件用来与硬件I/O设备通信，并处理中断。这些程序代码隐含在称为“设备”的概念模型里，系统中包含了多个这种“设备”，其中每一个“设备”都对应于一个与机器直接相连的网络（许多主机只有一个网络接口，而网关或路由器[2]具有多个网络接口）。

为了适应分组到达的随机性，系统必须具有能够从任何网络接口中读取分组的能力。可以有多种方式解决等待随机接口的问题。一些操作系统使用计算机的“软件中断”（software interrupt）机制。当一个分组到达时，产生一个硬件中断（hardware interrupt），设备驱动程序接收分组，重置接口设备。在中断返回之前，设备驱动程序会通知硬件安排下一个较低优先级的中断。当此次硬件中断结束后，低优先级中断会继续执行，“就好像是另外一个硬件设备产生了中断一样”。这种“软件中断”将CPU的正常处理挂起，使CPU跳到处理软件中断的代码段执行。因此，在某些系统中，所有的输入处理过程均以一系列中断的形式完成。这个思想正式体现在UNIX系统V采用的机制中，称为“流水线”（STREAMS）机制。

软件中断是有效的，但它需要一些硬件，而这些硬件并不是每台计算机都具有的。为了使我们的协议软件具有普遍适用性，我们没有采纳软件中断机制，所有代码设计都只是依赖于常规的中断机制。

即使使用常规硬件中断的操作系统也有许多方法可以解决多接口问题。一些操作系统使用的机制是：允许单个进程在多个输入设备上阻塞，当其中某个输入设备上有分组到达时，该进程立即被

① 一个octet就是一个八比特的数据单位，在许多系统中称作字节。

② 网关与路由器在全文中都是同义的。当描述一个协议时，尽量使用该协议所用的术语。

唤醒。另外一些操作系统则为每个接口各提供一个进程，允许该进程在与其相对应的接口上阻塞，而当该接口有分组到达时，对应的进程被唤醒。为了使我们的设计具有高效率，我们使用了图 2.1 所示的组织形式。

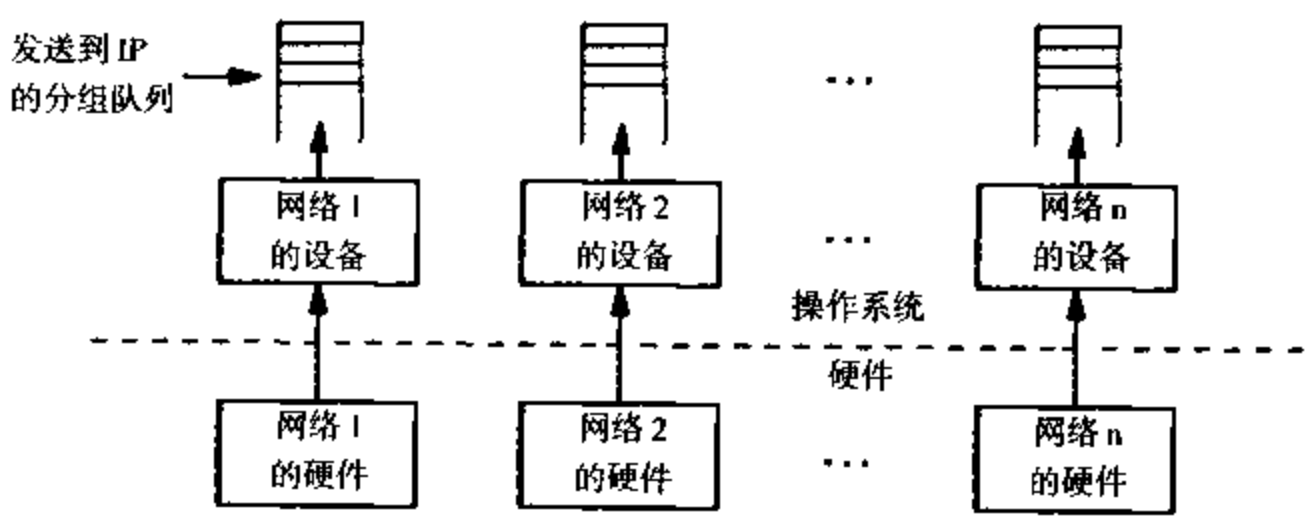


图 2.1　分组从网络接口硬件出发，经过操作系统中的设备驱动程序，到达与该设备相关的一个输入队列中

以太网的中断响应程序通过输入分组的分组类型字段，判断该分组应用的是什么协议。例如，如果一个以太网分组的分组类型值为 0800_{16}，则分组携带的是一个 IP 数据报。对于那些不能自识别帧类型的网络，系统设计者要么选用某个链路层的协议，以判别分组中的内容，要么认为分组类型为“预定类型”（priori）。前者的例子为 IEEE 802.2 链路层协议，后者的例子为 SLIP 协议。

2.8　向高层协议传递分组

由于输入操作发生在中断期间，此时设备驱动程序代码不能调用任何过程来进一步处理输入分组，它必须立即从中断中返回，因此中断过程并不直接调用 IP。再者，由于系统使用一个独立的进程实现 IP，因此设备驱动程序也不可能直接调用 IP。事实上，系统应用了前面所述的队列以及消息传送原语，以实现进程之间通信的同步。当一个携带 IP 数据报的分组到达后，中断过程必须将该分组置入队列中，并调用 send 函数通知 IP 进程已有一个数据报到达。如果此时 IP 进程手头没有分组可处理，则调用 receive 函数等待下一个数据报的到达。每一个网络设备都有一个相对应的输入队列，而惟一的 IP 进程从所有的队列中提取数据报进行处理。图 2.2 说明了这一概念。

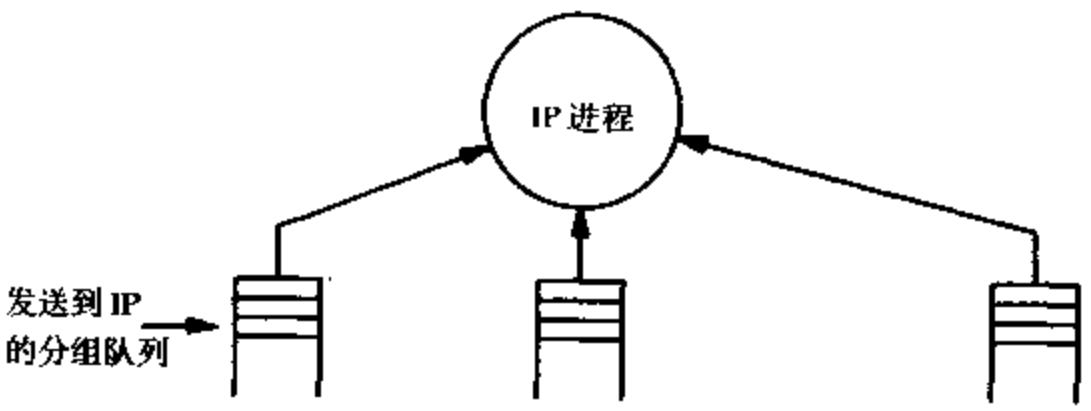


图 2.2　网络设备驱动程序和实现 IP 的进程之间的通信是利用一组队列完成的。当一个数据报到达时，网络输入进程将其放入队列中，并发送一个消息给 IP 进程

2.9 IP协议与传输协议之间的数据报传递

一旦IP进程接受了一个传入的数据报，它必须决定对将其发往何处做进一步处理。如果数据报中的内容是一个TCP报文段，则必须将其交付给TCP模块。如果它携带的是用户数据报（UDP），则必须将其交付给UDP模块，以此类推。我们将在后面详细介绍每一个模块的内部结构。而此刻，我们只关心这些进程的结构。

2.9.1 将传入的数据报发送给TCP

由于TCP比较复杂，因而在许多设计方案中，由一个独立的进程来处理传入的TCP报文段。由于IP和TCP作为各自独立的进程执行，因而IP和TCP必须利用进程间的通信机制来通信。它们应用前面介绍过的端口机制。IP调用psend将报文段存放在端口内，而TCP调用preceive来读取端口中的报文段。我们将会看到，其他一些向TCP发送报文的进程也要利用这个端口。

一旦TCP接收到一个报文段，就利用TCP协议端口号来寻找该报文段所属的连接。如果报文段中含有数据，TCP将把数据添加到与该连接相关的一个缓冲区中，并给发送方返回一个确认。如果输入的报文段中含有对发送出去的数据的确认，TCP输入进程还必须与TCP定时器管理进程通信，取消超时重发事件。

2.9.2 将传入的数据报发送给UDP

处理传入UDP数据报的进程结构与处理TCP进程采用的结构差别很大。由于UDP比TCP要简单得多，UDP模块不作为独立进程存在。事实上，它是由一些常规过程组成，IP进程通过调用来处理传入的UDP数据报。这些过程检查UDP目的站协议端口号，根据该端口号为UDP数据报选择一个操作系统队列（端口）。IP进程把UDP数据报存放在相应的端口中，使应用程序可从这些端口中提取数据报。图2.3描述了两者的不同之处。

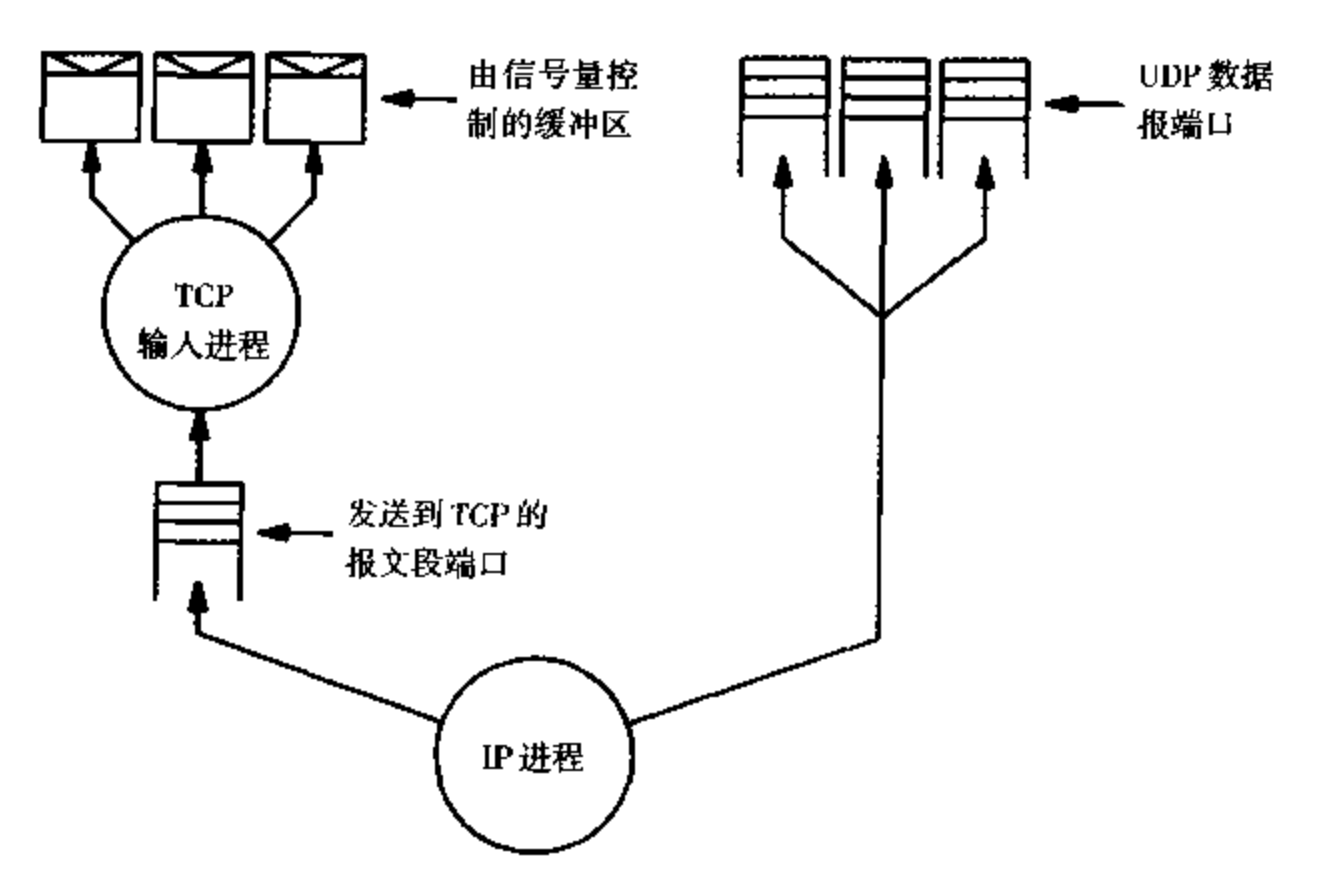


图2.3 数据报在高层协议软件间的流动。IP进程向TCP进程发送传入报文段，却直接将UDP数据报放入各个端口中，应用程序可以从这些端口中访问该数据报

2.10　向应用程序的传递操作

如图2.3所示，UDP根据协议端口号，将传入的用户数据报多路分解，将它们分别放入各自相应的操作系统队列中。同时，TCP分离出传入数据流，将这些数据存放在缓冲区。当一个应用程序希望接收UDP数据报或是TCP流的数据时，它必须访问UDP端口或TCP缓冲区。具体细节比较复杂，读者目前应该理解的一个简单概念是：

> 由于每个应用程序都作为一个独立的进程执行，它必须利用由系统提供的通信原语，才能与实现协议的进程协调。

例如，一个应用程序必须调用操作系统的preceive函数，来读取UDP数据报。当然，应用程序与操作系统内的进程之间的交互作用，要比操作系统内部的两个进程之间的交互作用复杂得多。

对于传入的TCP数据来说，应用程序并没有利用preceive函数。事实上，系统利用了信号量机制来控制对TCP缓冲区中数据的访问。希望从传入数据流中读取数据的应用程序将对控制缓冲区的信号量调用wait，而当TCP进程要将数据放入缓冲区时，它调用signal。

2.11　输出时的信息流

产生输出分组的原因不外乎以下两者之一。(1)某个应用程序将数据传递给一个高层协议软件，高层协议软件为此而向某个低层协议发送一个报文（或数据报），最终引起网络上的数据传输。(2)操作系统中的协议软件要传送一些信息（例如，一个确认或是一个对回送请求的响应）。在这两种情况下，到达底层硬件上的数据帧都必须经过某个网络接口发送出去。

为了有助于将分组的传输动作从那些实现了应用程序或协议的进程的运行中分离出来，系统为每一个网络接口设立了一个独立的输出队列。图2.4描述了这种设计思想。

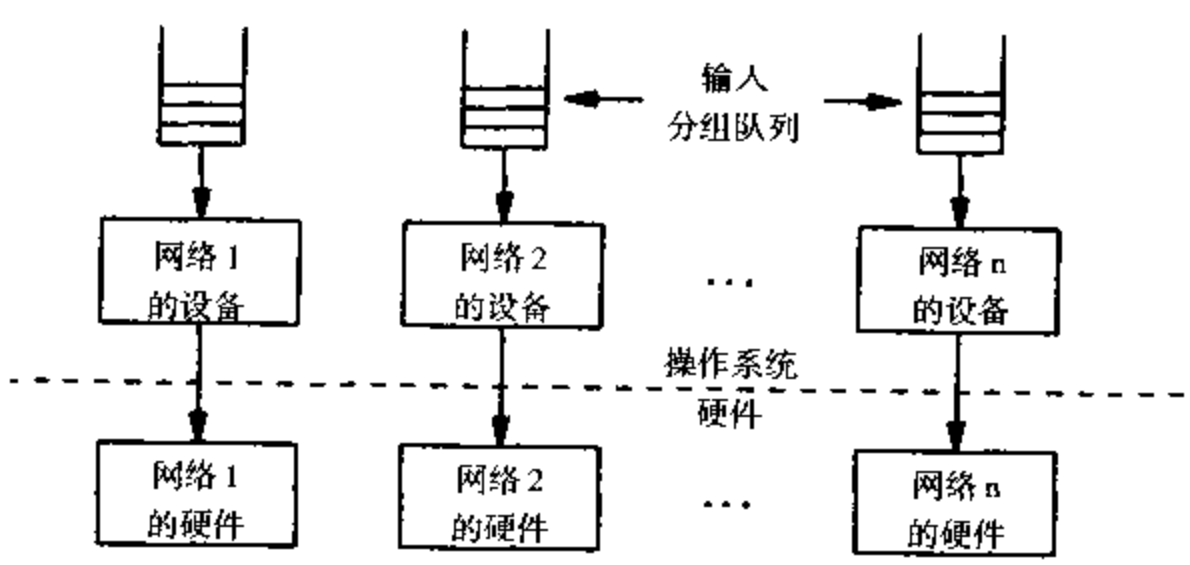


图 2.4　网络输出和存储输出分组队列。使用队列可将进程与网络传送分隔开

对应于输出设备的输出队列是整体设计思想的重要组成部分。有了这些队列，进程生成一个分组后，就可以将其存放在队列中等待输出，而进程继续往下执行，无须等待发送该分组。同时，硬件也可以并行地连续发送分组。当一个分组到达底层时，如果硬件正处于空闲状态（队列中没有分组存在），那么执行输出任务的进程将该分组放入队列中，同时调用一个设备驱动程序来启动硬件。当输出操作完成后，硬件向CPU发出中断请求。此时设备驱动程序从中断处理部分开始工作，将刚被发送出去的分组从队列中删除。如果队列中还有其他等待发送的分组，那么中断处理程序将重启

硬件，以继续发送下一个分组。最后中断处理程序从中断中返回，使其他常规处理得以继续执行。

因此，从IP进程的角度来看，分组传送是在后台自动进行的。只要队列里还有分组，硬件就会连续不断地传送它们。只有当IP进程将一个分组放入空队列中时才需要启动硬件。

当然，输出队列的容量是有限的，如果系统生成分组的速度比网络硬件传送分组的速度快，那么队列将会溢出。我们假设这种情况是罕见的。但如果确实出现了这种情况，生成分组的队列必须做出选择：要么丢弃分组，要么被阻塞，直到硬件完成传送一个分组的任务，队列腾出新的空间为止。

2.12 从TCP经过IP到网络输出

与TCP的输入一样，TCP的输出也是很复杂的。输出时必须先建立连接，所传送的数据必须放在报文段中，在对方发来确认之前，报文段必须不断重发。一旦某个报文段被装入一个数据报中后，就可交付给IP，由IP来处理选路和传送。系统软件利用了两个TCP进程来处理这一复杂过程。第一个进程被称作tcpout，由它来处理大部分的报文分段和数据传输细节。另一个进程是tcptimer，它管理着一个定时器，并为超时重发事件定时，当某个报文段不得不重发时，它将通知tcpout。

tcpout进程利用一个端口使来自多个进程的输入同步。由于TCP是面向数据流的，因而它允许应用程序一次发送几个字节的数据，端口内的数据项与个别的分组或报文段无关。实际上，一个生成数据的进程将其数据放入输出缓冲区中，并将一个消息放入端口中，通知TCP它已在输出缓冲区中写入了一些数据。定时器管理进程在定时器超时的时候，也会在端口中放入一个消息，通知TCP需要重发某个报文段。因此，我们可以认为端口是一个等待TCP处理的事件队列，每个事件都将导致一个报文段的传送或重发。另一方面，也有可能一个事件并不引起任何动作（例如，如果数据在接收窗口关闭时到达输出缓冲区）。我们将在今后的章节中详述事件的具体细节以及TCP的响应。

TCP生成一个数据报后，就将该数据报交付给IP，由IP来传送。虽然通信也有可能发生在位于同一台计算机上的两个应用程序之间，但在大多数情况下，数据报的目的站是另一台计算机。IP选取一个数据报输出时经过的网络接口，并将数据报传递给对应的网络输出进程。图2.5说明了TCP外发数据所经过的路径。

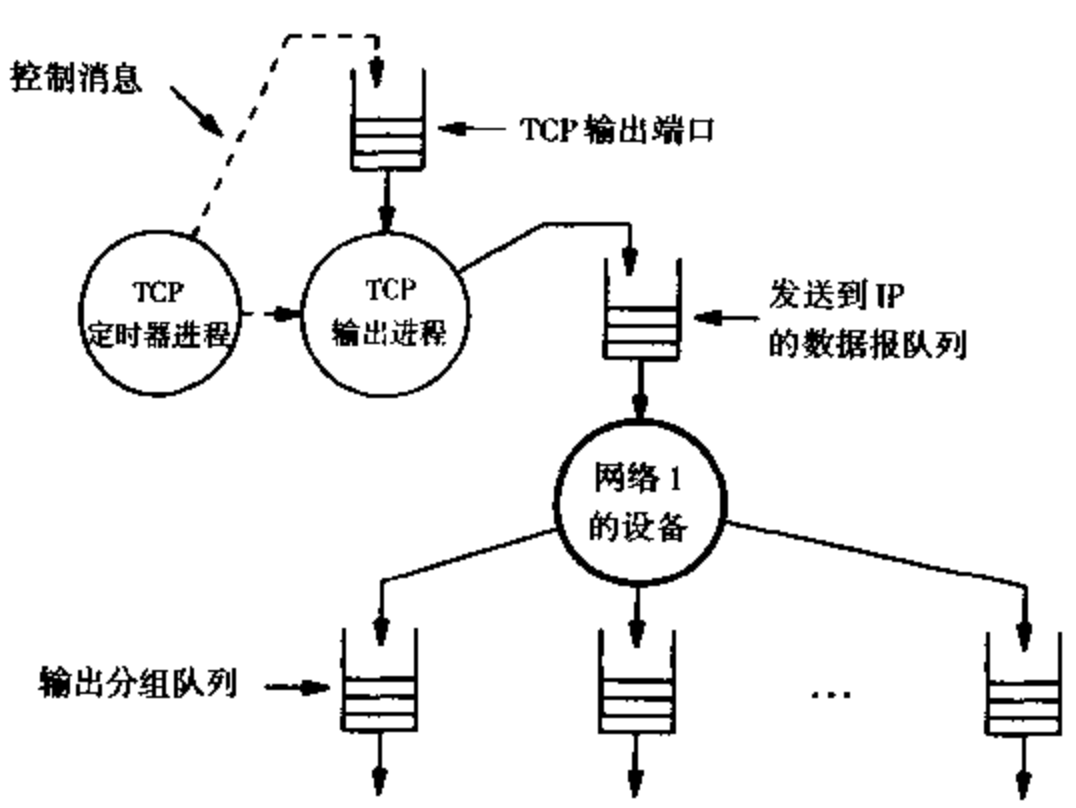


图2.5 TCP输出进程和定时器进程应用IP进程来传送数据

2.13　UDP 输出处理

UDP 外发数据时经历的路径比较简单。由于 UDP 不保证可靠的传送，因而发送方无需保留数据报的备份，也不用超时重发。只要数据报一生成，就可以被传送，而且此时发送方就可以丢弃该数据报的备份。

任何一个发送 UDP 数据报的进程都必须执行 UDP 过程来格式化数据，同时还要执行封装数据的过程，把生成的 IP 数据报传递给 IP 进程。

2.14　小结

TCP/IP协议软件是计算机操作系统中的一部分。它使用进程这一抽象概念使各个协议软件相互独立。使各自独立的协议软件模块更便于理解、设计和修改。每一个进程独立执行，并提供明确的并行机制。这种系统中有 IP 进程、TCP 输入进程、TCP 输出进程和 TCP 定时器管理进程，同时每一个应用程序还各有一个进程。

操作系统提供信号量机制，进程使用信号量来实现它们之间的操作同步。范例代码使用信号量来解决互斥问题（例如，保证在一段时间内只有一个进程访问某一段特定代码），以及协调生产者和消费者之间的关系（例如，一组进程产生数据项，而另一组进程消耗这些数据）。操作系统同时还提供了端口机制，允许进程通过一个有限队列互相发送消息。端口机制利用信号量来协调使用队列的多个进程。如果一个进程试图往一个全满的端口发送消息，那么该进程被阻塞，直到另一个进程从该端口中取出了一条消息。同样，如果一个进程试图从一个空的端口中读取消息，那么它也将被阻塞，直到其他进程往端口中放入一条消息时为止。

实现这些协议的进程同时使用常规队列和端口机制在它们之间传递分组。例如，IP输入进程将 TCP 报文段放入一个端口中，而 TCP 进程从该端口中提取这些报文段。同理，网络输入进程将到达的数据报放入一个队列中，然后由 IP 进程从该队列中提取它们。当数据经由常规队列传递时，系统必须使用消息传递机制或信号量机制来实现独立执行的进程同步。

图 2.6 总结了输出过程中应用程序和网络硬件之间的信息流动过程。一个应用程序，作为一个独立进程执行，调用系统例程向 TCP 传递数据流，或者向 UDP 传递数据报。对于 UDP 输出处理，运行某个应用程序的进程进入操作系统（通过一个系统调用），执行一段 UDP 过程，申请一个 IP 数据报，填写相应的目的地址，将 UDP 数据封装在 IP 数据报中，然后将 IP 数据报传递给 IP 进程，由 IP 进程发送出去。

对于 TCP 输出，运行应用程序的进程调用一个系统例程，将数据传入操作系统中，并保存在一个缓冲区内。应用进程随后通知 TCP 输出进程，有新的数据等待发送。当 TCP 输出进程执行时，它将数据流划分成报文段，然后，为了便于发送，它将这些报文段封装在 IP 数据报中。最后，TCP 输出进程将 IP 数据报放入某一端口中，IP 进程将从该端口中提取并发送这些数据报。

图 2.7 总结了输入信息的流程。网络设备驱动程序将所有携带 IP 数据报的传入分组置入 IP 进程的队列中。IP 从队列中提取这些分组，并对它们进行多路分解，将各个分组传递给适当的高层协议软件。当 IP 发现一个携带 UDP 的数据报时，它将调用 UDP 过程，将传入数据报置入适当的端口中，由应用程序读取它。当 IP 进程发现一个携带 TCP 报文段的数据报时，就将该数据报放入另一个端口中，由 TCP 输入进程从该端口中提取数据报。注意，IP 进程是设计思想的中心环节：一个 IP 进程既处理了输入，又处理了输出。

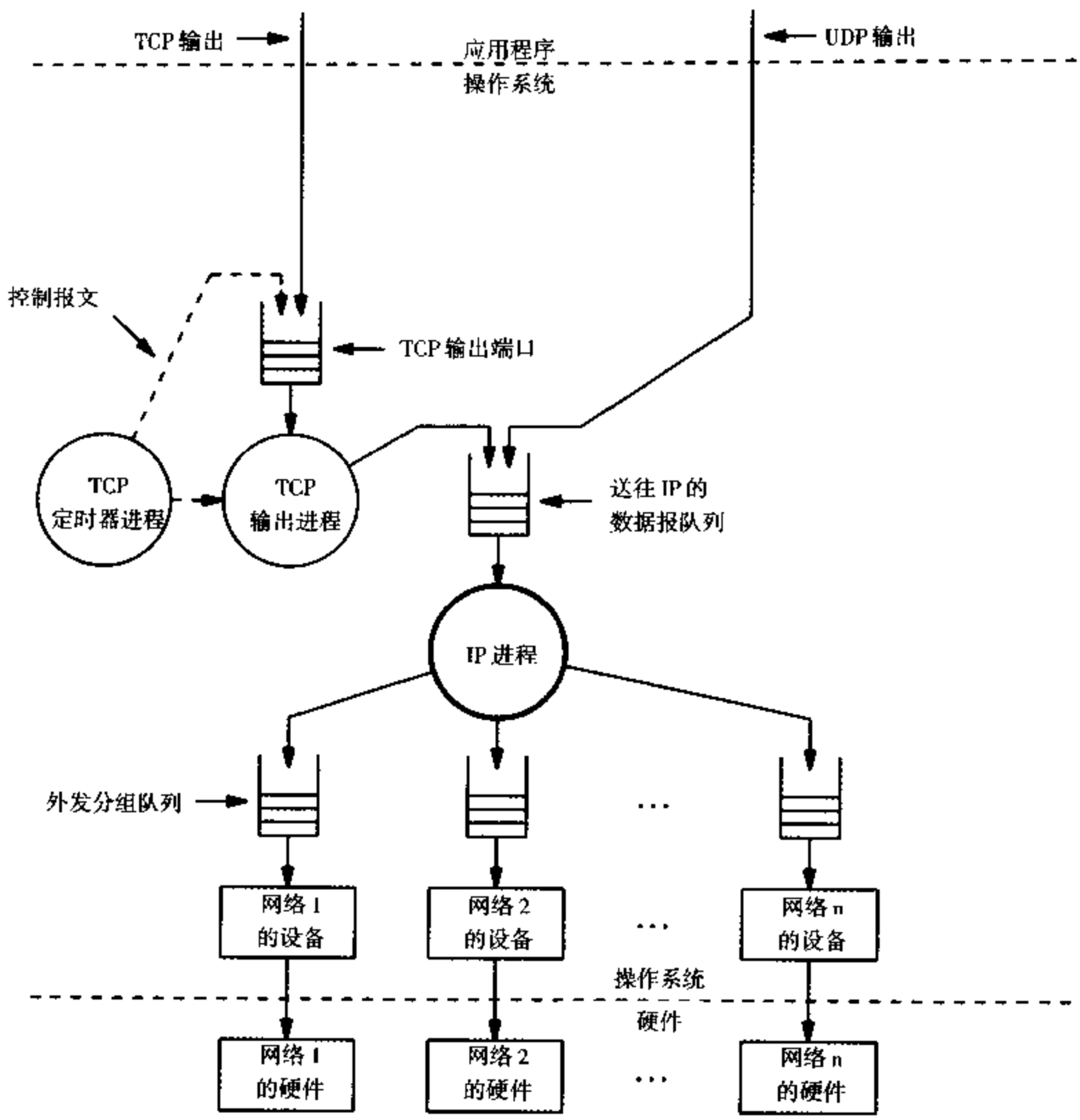


图2.6 输出进程结构，体现了应用程序和网络硬件之间的数据所经历的路径。设备队列中的输出操作在中断时开始。IP是设计的中心环节：输入输出的软件共用一个IP进程

深入研究

在我们的范例系统中使用了Xinu操作系统。Comer［1984］对该系统做了详细描述，包括进程和端口等抽象概念。Comer［1987］讲述了进程和端口如何在诸如UDP这样的简单协议中应用。Ritchie［1984］描述了UNIX系统V中的I/O流，Romkey［RFC 1055］中包含了SLIP规范。

习题

2.1 为什么协议实现者尽量减少协议使用的进程的数目？

2.2 如果本章描述的系统工作在一个CPU速度比网络硬件传送分组的速度慢的计算机上，会发生什么情况？

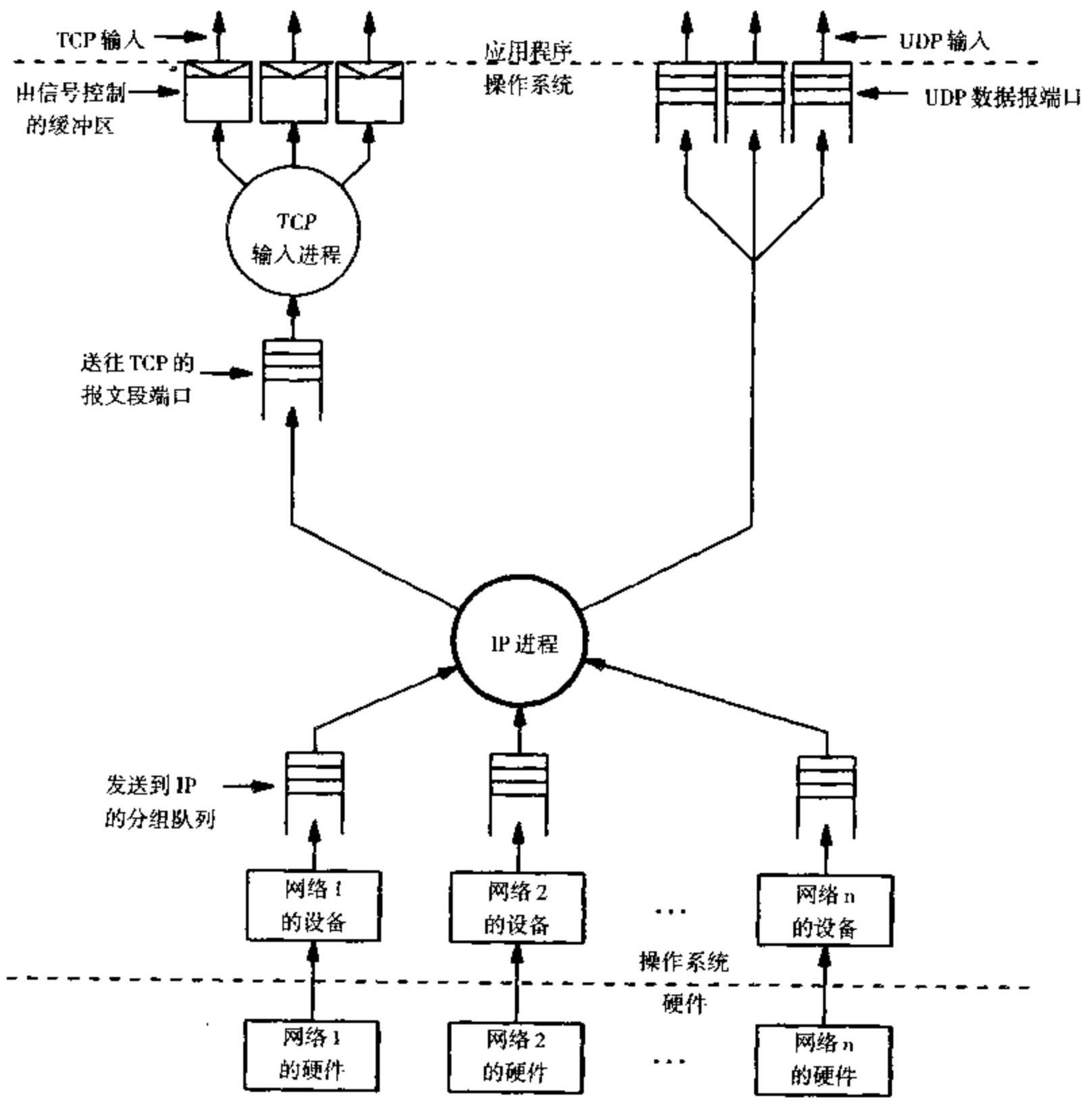


图 2.7　输入进程结构，体现了应用程序和网络硬件之间的数据所经历的路径。输入分组从接口传递到设备队列的过程与系统处理过程是异步的。IP 是设计的中心环节，输入输出的软件共用一个 IP 进程

2.3 大量阅读有关软件中断的资料，粗略设计一种使用软件中断、但不使用进程的协议实现方案。

2.4 阅读有关 UNIX STREAMS 的文章，将其与本章描述的基于进程的实现方案相比，两者的优缺点分别是什么？

2.5 比较下面两种设计方案：在第一个方案中，每一个发送 UDP 数据报的应用程序都要直接执行所有 UDP 和 IP 代码；在第二个方案中，一个独立的 UDP 进程接受来自所有应用程序的外发 UDP 数据报。两种方案的主要优缺点是什么？

2.6 考虑一个使用了大量进程来处理分组的协议软件设计。假设系统为到达的每一个数据报或是本地应用程序产生的每一个数据报分配一个进程。同时假设进程将跟随该数据报，直到它被发送或被传递给上层协议。该设计方案的主要优点是什么？主要缺点是什么？

第3章　网络接口层

3.1　引言

TCP/IP Internet 协议软件在概念上分为五层，如图 3.1 所示。

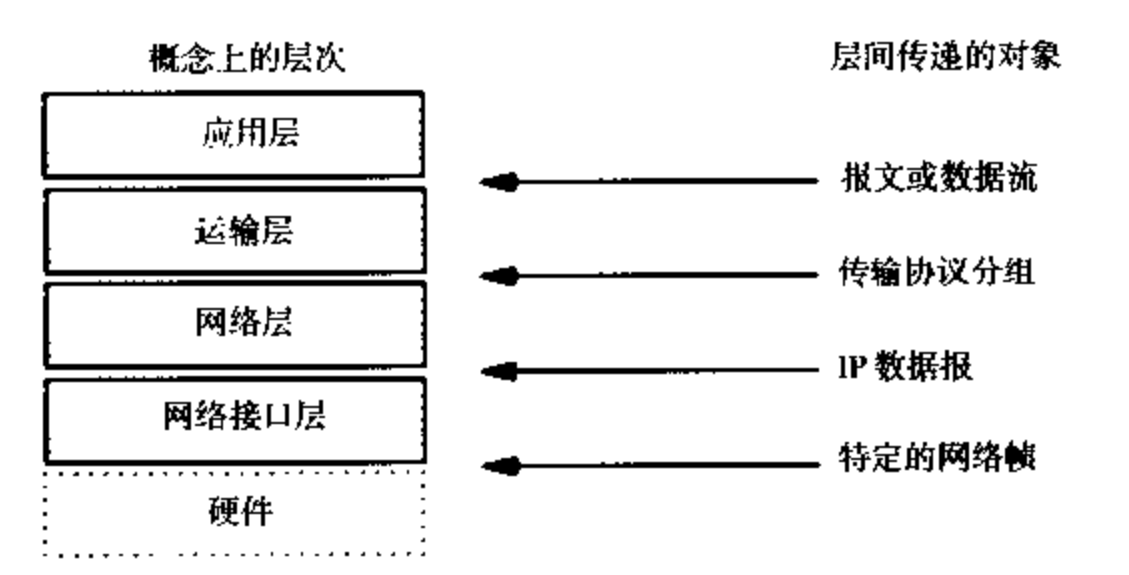


图 3.1　TCP/IP 协议软件概念上的分层结构

本章详细讨论最底层，也就是网络接口层。从概念上来看，网络接口层管理网络的硬件，完成从IP地址到硬件地址的映射过程，封装并发送外发分组，接收传入分组并将其发往对应的模块。在本章中，你将会了解到设备驱动程序和接口软件的结构，这种结构使高层协议软件能够区分并控制一台机器上的多个网络硬件接口。本章还涉及缓冲区管理及如何将分组发往各自对应的模块以进一步处理。第 4 章将讨论地址转换及封装处理。

在这里我们有意忽略了对网络设备驱动程序代码的介绍，因为其中包含了许多底层细节问题，这些细节只有在非常熟悉某种网络硬件设备的情况下，才能完全理解。而本章的重点是网络接口层的组成，理解它们是掌握高层协议软件的基础。

3.2　网络接口抽象模型

网络接口层软件形成了一个适用于整个协议体系的网络接口抽象模型，其思想很简单：

> 网络接口抽象模型定义了操作系统中的协议软件与底层硬件之间的接口。它隐蔽了硬件细节，使协议软件能够使用相同的数据结构作用于不同的网络硬件。

3.2.1　接口数据结构

为了实现网络接口层软件与硬件的相对独立，我们定义了一个数据结构，用以保存所有与硬件无关的接口信息（例如，硬件是在正常工作还是发生了故障），并协调各个协议软件，使它们主要

通过这个数据结构来完成与硬件的交互作用。在我们的范例程序中，网络接口由一个数组（nif）构成，其中每个数组元素对应于该机器上的一个硬件接口[1]。接口数组中的各个元素通过数组索引被整个系统使用。这样，我们就能够提及“0号网络接口”或是“第一个网络接口”了。在文件netif.h中对有关的数据结构做了说明。

```
/* netif.h - NIGET */

#define     NI_MAXHWA 14                /* max size of any hardware     */
                                        /*  (physical) net address      */
struct hwa  {                           /* a hardware address           */
            int  ha_len;                /* length of this address       */
            unsigned char   ha_addr[NI_MAXHWA];  /* the address         */
};

#define NI_INQSZ        30              /* interface input queue size   */
#define NETNLEN         30              /* length of network name       */

#define NI_LOCAL        0               /* index of local interface     */
#define NI_PRIMARY      1               /* index of primary interface   */
#define NI_MADD         0               /* add multicast (ni_mcast)     */
#define NI_MDEL         1               /* delete mnlticast(ni-mcast)   */

/* interface states */

#define NIS_UP          0x1
#define NIS_DOWN        0x2
#define NIS_TESTING     0x3

/* Definitions of network interface structure (one per interface)     */

struct netif {                          /* info about one net interface */
      char          ni_name[NETNLEN];   /* domain name of interface     */
      char          ni_state;           /* interface states: NIS_ above */
      IPaddr        ni_ip;              /* IP address for this interface */
      IPaddr        ni_net;             /* network IP address           */
      IPaddr        ni_subnet;          /* subnetwork IP address        */
      IPaddr        ni_mask;            /* IP subnet mask for interface */
      IPaddr        ni_brc;             /* IP broadcast address         */
      IPaddr        ni_nbrc;            /* IP net broadcast address     */
      unsigned int  ni_mtu;             /* max transfer unit (bytes)    */
      unsigned int  ni_hwtype;          /* hardware type (for ARP)      */
      struct hwa    ni_hwa;             /* hardware address of interface */
      struct hwa    ni_hwb;             /* hardware broadcast address   */
                                        /* multicast add/delete func.   */
int   (*ni_mcast)(int op, int dev, Eaddr hwa, IPaddr ipa);
```

① 注意，可以定义没有对应硬件设备的其他接口，例如用于测试的loopback接口。

```
        Bool        ni_ivalid;          /* is ni_ip valid?              */
        Bool        ni_nvalid;          /* is ni_name valid?            */
        Bool        ni_svalid;          /* is ni_subnet valid?          */
        int         ni_dev;             /* the Xinu device descriptor   */
        int         ni_ipinq;           /* IP input queue               */
        int         ni_outq;            /* (device) output queue        */
        /* Interface MIB */
        char        *ni_descr;          /* text description of hardware */
        int         ni_mtype;           /* MIB interface type           */
        long        ni_speed;           /* bits per second              */
        char        ni_admstate;        /* administrative status (NIS_*) */
        long        ni_lastchange;      /* last state change (1/100 sec) */
        long        ni_ioctets;         /* # of octets received         */
        long        ni_iucast;          /* # of unicast received        */
        long        ni_inucast;         /* # of non-unicast received    */
        long        ni_idiscard;        /* # dropped - output queue full */
        long        ni_ierrors;         /* # input packet errors        */
        long        ni_iunkproto;       /* # in packets for unk. protos */
        long        ni_ooctets;         /* # of octets sent             */
        long        ni_oucast;          /* # of unicast sent            */
        long        ni_onucast;         /* # of non-unicast sent        */
        long        ni_odiscard;        /* # output packets discarded   */
        long        ni_oerrors;         /* # output packet errors       */
        long        ni_oqlen;           /* output queue length          */
        long        ni_maxreasm;        /* max dategram can ressemble   */
};

#define     NIGET(ifn)   ((struct ep *)deq(nif[ifn].ni_ipinq))

#define     NIF  Neth+Noth+1            /* # of interfaces, +1 for Iocal*/

extern struct netif   nif[];

int netwrite (strnct netif*, struct ep*, unsigned);
```

数据结构netif定义了数组nif中每个数组元素的组成。netif中的字段不但定义了协议软件需要的所有数据项，而且定义了用于收集统计数据的变量。例如字段ni_IP的内容是分配给接口的IP地址，而字段ni_mtu的内容是最大传输单元（maximum transfer unit），即网络中一个分组的最大长度，以八位组为单位。名字以valid结尾的字段中包含的是布尔变量，这些布尔变量指出与其名字的前半部相对应的字段是否有效：一旦对应字段被赋予有效值，初始化软件就将它们设置为TRUE。例如，当字段ni_IP中包含了有效的IP地址时，字段ni_ivalid的值就为TRUE。

设备驱动软件将已经到达的，并且将由IP进程来处理的数据报放入一个队列中。字段ni_ipinq中包含了一个指向该队列的指针。程序利用宏NIGET从队列中依次取出这些数据报。宏NIGET以接口号为参数，从接口队列中取出下一个数据报，并返回指向该数据报的指针。

3.3　以太网的基本定义

尽管我们有意避免大多数底层设备驱动程序的细节问题，但是在本书中一些基本的定义还是会经常用到。例如，在netif.h的声明中引用了C数据结构ep，ep定义了以太网（Ethernet）数据报存储进入的缓冲区格式。因为以太网作为本书的网络例子在全书中都会出现，所以ep还是需要重视的。声明是在文件ether.h中。以太网数据报的说明跟随4个字段，这4个字段位于我们的软件所用的ep结构的起始部分。

```
/* ether.h */

/* Ethernet definitions and constants */

#define EP_MAXMULTI      10        /* multicast address table size      */

#define EP_MINLEN        60        /* minimum packet length             */
#define EP_DLEN        1500        /* length of data field ep           */
#define EP_HLEN          24        /* size of (extended) Ether header   */
#define EP_CRC            4        /* ether CRC (trailer)               */
#define EP_MAXLEN      EP_HLEN+EP_DLEN
#define EP_ALEN 6            /* number of octets in physical address  */
typedef unsigned char   Eaddr[EP_ALEN]; /* a physical Ethernet address  */
#define EP_RETRY       3     /* number of times to retry xmit errors  */
#define EP_BRC  "\377\377\377\377\377\377"/* Ethernet broadcast address */
#define EP_RTO 300           /* time out in seconds for reads         */
#define EP_NUMRCV 16         /* number LANCE recv buffers (power 2)   */
#define EP_NUMRCVL2 4        /* log2 of the number of buffers         */

#define EPT_LOOP       0x0060          /* type: Loopback              */
#define EPT_ECHO       0x0200          /* type: Echo                  */
#define EPT_PUP        0x0400          /* type: Xerox PUP             */
#define EPT_IP         0x0800          /* type: Internet Protocol     */
#define EPT_ARP        0x0806          /* type: ARP                   */
#define EPT_RARP       0x8035          /* type: Reverse ARP           */

struct  eh {                   /* ethernet header                     */
        Eaddr  eh_dst;         /* destination host address            */
        Eaddr  eh_src;         /* source host address                 */
        unsigned short  eh_type;/* Ethernet packet type (see below)   */
};

struct  ep     {               /* complete structure of Ethernet packet */
        u_long  ep_nexthop;    /* niput() uses this                   */
        short   ep_ifn;        /* originating interface number        */
        short   ep_len;        /* length of the packet                */
        short   ep_order;      /* byte order mask (for debugging)     */
```

```
        struct  eh ep_eh;    /* the ethernet header                    */
        char    ep_data[EP_DLEN];      /* data in the packet            */
};

/* Packet Byte Order Constants (set means host order) */
#define EPO_NET         1           /* network layer                 */
#define EPO_IP          2           /* level 1 protocols...          */
#define EPO_ARP         2
#define EPO_RARP        2
#define EPO_ICMP        4           /* level 2 protocols             */
#define EPO_IGMP        4
#define EPO_TCP         4
#define EPO_UDP         4
#define EPO_OSPF        4
#define EPO_DNS         5           /* application protocols         */

/* these allow us to pretend it's all one big happy structure    */

#define ep_dst  ep_eh.eh_dst
#define ep_src  ep_eh.eh_src
#define ep_type ep_eh.eh_type

#if     Noth > 0
/* this is for Othernet simulation */

struct  otblk {
        Bool    or_valid;         /* these entries are valid?            */
        Eaddr   ot_paddr;         /* the Othernet physical address       */
        Eaddr   ot_baddr;         /* the Othernet broadcast address      */
        int     ot_rpid;          /* id of process reading from othernet */
        int     ot_rsem;          /* mutex for reading from the othernet */
        int     ot_pdev;          /* Physical device devtab index        */
        int     ot_intf;          /* the associate interface             */
        char    *ot_descr;        /* text description of the device      */
};
#endif  Noth


#define ETOUTQSZ        30

/* Ethernet control block descriptions */

struct  etblk   {
        int       etxpending;     /* number of packets pending output     */
        int       etrpending;     /* 1 => a receive is pending already    */
        int       etwtry;         /* num. of times to retry xmit errors   */
        char      *etwbuf;        /* pointer to current transmit buffer   */
```

```
        short     etnextbuf;      /* for checking buffers round robin      */
#if     Noth > 0
        struct  otblk   *etoth[Noth];    /* eaddr to oaddr translations  */
#endif  Noth
};


/* ethernet function codes * /

#define EPC_PROMON     1                /* turn on promiscuous mode     */
#define EPC_PROMOFF    2                /* turn off promiscuous mode    */
#define EPC_MADD       3                /* add multicast address        */
#define EPC_MDEL       4                /* delete multicast address     */

struct utdev {
        unsigned int   ud_iomem;
        unsigned int   ud_iosize;
        struct devsw  *ud_pdev;
        Eaddr          ud_paddr;
        Eaddr          ud_bcast;
        char          *ud_descr;
        int            ud_outq;
        int            ud_ifnum;
        int            ud_nextbuf;
        int            ud_xpending;
        int            ud_xmaddr;      /* transmit base buf #          */
        int            ud_wretry;
        int            ud_rmin;        /* receive ring buffer base addr*/
        int            ud_rmax;        /* receive ring buffer max addr */
};

struct etdev {
        unsigned int   ed_iomem;
        unsigned int   ed_iosize;
        struct devsw  *ed_pdev;
        Eaddr          ed_paddr;
        Eaddr          ed_bcast;
        char          *ed_descr;
        int            ed_outq;
        int            ed_ifnum;
        int            ed_wretry;
        struct scb    *ed_scb;
        struct cbl    *ed_cbl;
        struct rfd    *ed_rfd;        /* first in RFD ring            */
        struct rfd    *ed_rfdend;     /* last in RFDring              */
```

```
        struct rbd    *ed_rbd;          /* first in RBD ring            */
        struct rbd    *ed_rbdend;       /* last in RBD ring             */
        struct tbd    *ed_tbd;          /* transmit buffer descriptor   */
        unsigned char *ed_xmbuf;        /* transmit buffer base         */
        unsigned char  ed_irq;          /* SIRQ reg value               */
        int            ed_mcc;          /* # valid mca's                */
        Eaddr          ed_mca[EP_MAXMULTI]; /* multicast addr. table   */
/*XXX*/
        int            ed_xpending;
};

extern struct utdev     ue[];
extern struct etdev     ee[];

extern  struct  etblk   eth[];
```

3.3.1 应用中的统计数据

保存接口的统计数据对程序调试与网络管理来说很重要。例如，文件netif.h中的ni_iucast字段包含对传入的单播（或非广播）数据分组数目的记录，而ni_idiscard和ni_odiscard字段记录差错时必须丢弃的输入和输出数据分组数目。

接口的数据结构分别在字段ni_hwa和ni_hwb中记录硬件的物理（硬件）地址和物理（硬件）广播地址。物理地址的长度取决于底层硬件，因此协议软件利用数据结构hwa来代表这种地址。每一个硬件的地址都起始于一个整数形式的长度字段，这样高层的协议软件不必知道硬件的细节问题就可以处理硬件的地址。

3.4 接口的逻辑状态

在故障分析过程中，管理人员经常要让某个机器上的一个或多个接口停止工作。ni_state字段提供了控制接口逻辑状态的机制，该机制与底层硬件无关。例如，网络管理员可以将ni_state赋值为NIS_DOWN，以完全终止输入输出。此后，管理员又可将ni_state赋值为NIS_UP来重新启动输入输出。

将接口的逻辑状态与硬件的物理状态区分开是很重要的，因为这样就给了管理员控制接口操作的自由。显然，当硬件出现故障时，管理员就声明此接口已停止工作（down）。但声明某个接口停止工作，并不是指物理硬件被拆除，也不表示硬件不能正常工作。事实上，管理员的这项声明仅仅导致软件停止接收传入分组，并阻止外发分组的发送。例如，当网络过载时，管理员就可以声明网络中的某个接口停止工作。

3.5 本地主机接口

IP协议除了要在各个网络接口之间给数据报选择路由外，还必须给那些往返于本机IP协议和上层协议软件之间的数据报进行选路。IP与本机之间的交互作用可由以下两种方法实现：

- 在 IP 程序段中对此种情况进行明确判别
- 附加一个本机网络接口

在我们的设计思想中采用了伪网络接口（pseudo-network interface）技术。伪网络接口既没有与之相关的设备驱动程序，也没有具体的硬件设施，如图 3.2 所示。事实上，发往伪网络接口的数据报将被转交给本机的协议软件。类似地，当协议软件生成一个需要发送出去的数据报时，它通过伪网络接口将此数据报传递给 IP。

为本机设立伪网络接口有以下几个优点。首先，它消除了特殊情况的存在，简化了 IP 程序。其次，它使得本机在路由表中的表示方法与其他目的站的表示方法完全一致。第三，它使网络管理员能够像查询其他接口一样方便地查询本地接口（例如，获取本地应用程序生成的分组的数目）。

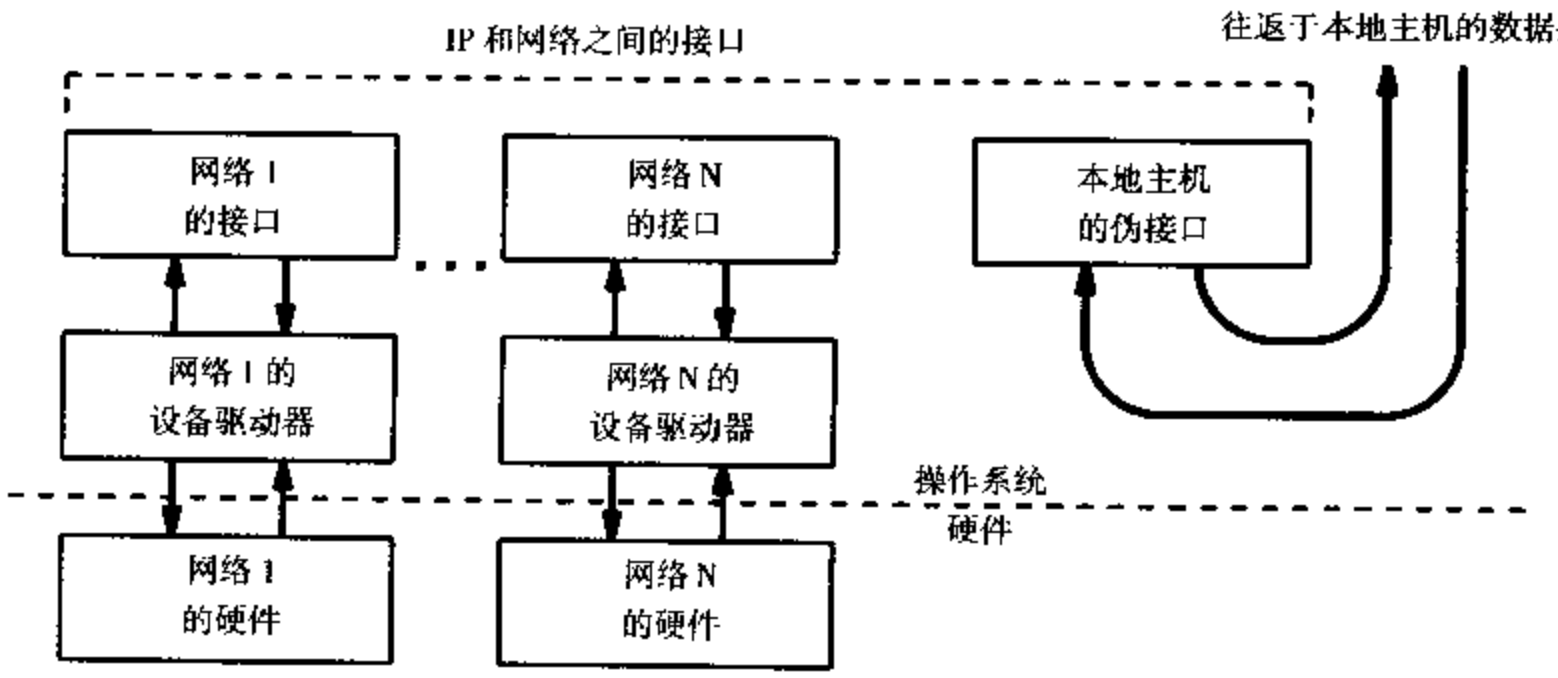


图 3.2　用于和本机通信的伪网络接口

3.6　缓冲区管理

传入的分组必须被保存在内存中，并将其传递给适当的协议软件做进一步处理。同时，应用程序产生的输出数据也必须以分组的形式存储在内存中，并将其传递给网络硬件设备传送出去。因此，网络接口层接受内存中准备外发的数据，并把将要传递给上层协议软件的传入数据放入内存中。协议软件的有效性最终取决于如何管理保存这些分组的存储器。一个良好的设计要做到快速分配存储空间，并且避免分组在各层协议软件之间移动时的数据复制。

理论上，系统可以将内存划分为大小固定的缓冲区，并使每个缓冲区都大得足以保存一个分组，以达到有效地分配内存空间的目的。然而在实际中，由于以下几个原因使得选择缓冲区的最佳大小变得复杂。第一，一台计算机可能与几个网络相连，而每个网络都有各自不同的最大分组长度限制。再者，它应该在不改变系统缓冲区大小的情况下，能够与其他新类型的网络建立连接。其次，IP 协议软件有可能要保存比底层网络分组更大的数据报（例如，装配一个很长的数据报）。第三，应用程序可能会发送或接收任意长度的信息。

3.6.1　大缓冲区方案

看起来，最理想的解决办法是将缓冲区划分得足够大，能够存储最长的有效信息或分组。但是，由于 IP 数据报可能有 64K 八位组长，倘若使缓冲区大得足以容纳任意长度的数据报，有效内

存空间根本划分不了几个缓冲区就会被耗尽。再者，短小的分组才是常规的，而长数据报并不多见。因此，使用大缓冲区将会导致内存空间的实际利用率很低，与此同时系统却没有足够多的缓冲区来满足通信量的需求。

事实上，采用大缓冲区的设计者们通常都会选择一个系统能够处理的数据报的最大长度限制D，然后使缓冲区足够大，可以容纳长度为D的数据报外加物理网络帧头。采用这种方法是在允许使用较长的数据报和能够给正常的通信量提供足够多的缓冲区之间的折衷。因此，D的大小不但取决于缓冲区大小的期望值，还取决于正常的系统利用率。典型地，分时系统给D选择了4K字节到8K字节之间的一个值。

3.6.2 链表方案（mbuf）

除了大缓冲区方案外，主要还可以采用将较小的缓冲区以链表的形式连接起来，以处理任意长度的数据报。在链表设计方案中，表中的缓冲区既可以是大小固定的，也可以是不固定的。大多数系统分配大小固定的缓冲区，因为这样做可以杜绝内存碎片，并保证内存的高利用率。通常，每个缓冲区都很小（如，在128字节到1K字节之间），一个完整的数据报需要许多个链接在一起的缓冲区来保存。例如BSD UNIX使用称为mbuf结构的链接，其中每个mbuf为128字节长。各个mbuf本身不必写满，它具有一个短小的首部，指出数据从何处开始，以及一共有多少字节。

允许链表中的每个缓冲区只包含部分数据还具有其他一些优点：它允许快速封装而无须复制操作。当某一层的软件接收到上一层协议发送来的信息后，就分配一个新的缓冲区，填写其中的首部信息，并将新缓冲区插入保存此信息的链接列表中。这样，附加的字节就可以很容易地插在某个信息的前面，而无须移动已存在的数据。

3.6.3 方案举例

在我们的范例系统中，采用能有效地存储任意长度数据报的大缓冲区方案与小缓冲区的链表方案之间的折衷方案：它既划分了许多大小能够容纳单个分组的缓冲区，同时又划分了少数用于保存大数据报的缓冲区。系统使用小缓冲区完成分组一级的输入输出，而仅当生成或装配大数据报时才利用大缓冲区。选择这样的设计思想，是因为我们预计大多数数据报都比约定的网络最大传输单元（MTU）短，但同时也希望能够装配大数据报。这样，在大多数情况下，当一个分组被读入缓冲区后，就可以直接将整个缓冲区传递给IP，系统仅在装配一个大数据报时才需要复制数据。

为了使缓冲区的处理过程统一，范例系统采用了由操作系统提供的缓冲器自识别策略。为了分配一个缓冲区，系统调用getbuf函数，并且指定所要求的是大缓冲区还是小缓冲区。但是，只要缓冲区分配完成，需要保存的仅仅是指向这个缓冲区的指针。为了将缓冲区归还空闲缓冲区列表，系统调用freebuf，将被释放的缓冲区的指针交还给它，freebuf自动计算缓冲区的大小。缓冲区自识别策略的优点是，协议软件可以不用记住它是来自大缓冲区池，还是来自小缓冲区池，而只需传递指向该缓冲区的指针。因而，外发分组就能被保存在一张简单的列表中，由地址来识别它们。一旦设备将一个分组发送出去，驱动程序软件就可以调用freebuf来释放此缓冲区，同样无须知道它的类型。

3.6.4 有关缓冲区的其他议题

直接存储器存取的存储器（DMA Memory）。硬件上的某些要求经常会使缓冲区管理复杂化。例

如，有些设备的输入输出只能在内存的直接存储器存取（DMA）区中执行。在这样的系统中，操作系统可能会选择分配两套缓冲区：协议软件使用的缓冲区和设备传输时使用的缓冲区。这种系统必须在发送数据前将常规缓冲区中的数据复制到DMA区中，而将传入的数据从DMA区复制到常规缓冲器中。

整写散读（gather-write，scatter-read）。有些设备能够在不连续的内存块中发送或接收分组。对输出来说，这种设备接受一张缓冲区地址和长度列表。它们从列表的缓冲区中，收集整理分组的所有部分，并将得到字节流顺序发送出去，系统不必在连续的内存中重新组装分组。这种技术称为整写（gather-write）。与此类似，这个硬件很可能还支持散读（scatter-read）。此时，硬件根据一张由设备驱动程序填写的缓冲区地址列表，将一个分组分置于不连续的内存块中。显然整写和散读功能使链接式缓冲区的分配变得容易且有效，因为硬件可以直接从一张链表的缓冲区中获得一个分组的所有块，而不需要处理程序在内存中装配完整的分组。这两种技术也可用于大小固定的缓冲区，因为它们允许驱动程序在不进行复制的情况下封装数据报。要做到这一点，驱动程序只要将帧首部存储在内存中的某个地方，然后把首部地址连同即将成为物理分组中数据部分的数据报地址一起传递给硬件。

页面分配（page assignment）。在一个支持虚拟分页内存的计算机系统中，协议软件可以试图在页面的分界线上划分缓冲区，以使进程之间传递缓冲区的过程能够通过交换分页表格中的表项来完成，这样就不需要数据复制。这种技术对分页尺寸小的计算机来说尤为有用（如Digital公司的VAX结构，它具有512字节的页面），但对分页尺寸大的计算机来说，效果就不那么理想（如Sun Microsystems的Sun3结构，它的页面为8K字节）。进一步说，当数据在操作系统和应用程序之间移动时，交换分页表格的表项将大大提高效率。但是，传入的分组中含有一系列的首部，这些首部使得在分组被解读之前，要确定用户数据的确切偏移量是十分困难的，或者说是不可能的。因此，几乎没有实例程序试图将分组与页面的分界线对齐。

3.7　传入分组的多路分解

当一个分组到达时，网络接口层中的设备驱动程序软件检查分组中的类型字段，以确定使用哪种协议软件来处理此分组。在编写接口软件时，设计者们通常采用以下两种基本手段中的一种：或者将多路分解程序编成代码放在一个过程中，或者利用一张表格，将分组类型与适当的过程一一对应。一般说来，使用代码的方式更为有效，但这意味着当又有新的协议加入时，软件必须重新编译，因此在试验阶段时，利用表格方式则更方便。在范例中，我们选择了利用一个过程对分组进行多路分解。过程ni_in中包含了完成多路分解操作的代码。

```
/* ni_in.c - ni_in */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#include <ospf.h>

int arp_in (struct netif*, struct ep*);
int rarp_int (struct netif*, struct ep*);
int ip_in (struct netif*, struct ep*);
```

```
/*-----------------------------------------------------------------------
 *  ni_in - network interface input function
 *-----------------------------------------------------------------------
 */
int
ni_in (struct netif *pni, struct ep *pep, unsigned len)
{
        int     rv;

        pep->ep_ifn = pni - &nif[0];        /* record originating intf # */

        pni->ni_ioctets += len;
        if (!memcmp (pni->ni_hwa.ha_addr, pep->ep_dst, EP_ALEN))
                pni->ni_iucast++;
        else
                pni->ni_inucast++;
        switch (pep->ep_type) {
        case EPT_ARP:  rv = arp_in(pni, pep);   break;
        case EPT_RARP: rv = rarp_in(pni, pep);  break;
        case EPT_IP:   rv = ip_in(pni, pep);    break;
        default:
                pni->ni_iunkproto++;
                freebuf(pep);
                rv = ok;
        }
        return rv;
}
```

在我们的范例中，一旦某个表示新的输入分组到达的中断发生时，设备驱动程序调用ni_in。ni_in对四种情况分别做出处理。如果分组携带一个ARP信息、RARP信息或IP数据报时，ni_in将此分组传递给适当的协议程序并返回结果。否则，它将相应的缓冲区释放回缓冲池，丢弃此分组。如果分组被接受，ni_in使相应的计数器值递增，以记录下这一广播分组或非广播分组的到达。我们将在以后详细描述ni_in中调用到的一些过程。

3.8 小结

网络接口层中的软件完成网络硬件设备与其他协议软件之间的通信。它包括缓冲区管理程序、下层设备驱动程序，并含有许多与硬件有关的细节。最为重要的是，它提供了一个称为网络接口的抽象模型，使得上层协议与硬件细节无关。

在netif结构中定义了为每个网络接口保存的信息结构。它包括所有与接口有关的信息，以便上层协议在不知道具体硬件接口细节的情况下，也能访问接口信息。在netif的各个字段中，有一些含有硬件信息（如硬件地址），而另一些字段中所含的信息供协议软件使用（如子网掩码）。

深入研究

Comer [1984] 提供了在范例程序中采用的缓冲池策略的更多细节。Comer [1987] 描述以太网的硬件接口，提供了设备驱动程序的细节，并解释了设备驱动程序是如何嵌入一个操作系统中的。Leffler、McKusick、Karels 和 Quarterman [1989] 描述了 UNIX 中 mbuf 的使用。

习题

3.1 查看一下 SNMP（RFC 1213 和 2013）使用的 MIB。它特别为每个网络接口保存的统计数据有哪些？是否每个统计数据在接口的数据结构中都有一个相应的字段？

3.2 阅读 BSD UNIX 源代码，看看 mbuf 是如何构造的。为什么在首部中要含有两个指向其他 mbuf 块的指针？

3.3 用 4BSD UNIX 中的 ping 程序做试验（即 ICMP 回送请求 / 应答）来判断在你的本地环境中，设备可发送和接收的最大分组长度。它与网络的 MTU 相比较又如何？

3.4 找一份描述 Lance 以太网接口设备的硬件手册。是否有可能将多个等待发送的分组放入队列中？如果能够，这样做对软件设计者来说有什么好处与坏处？

3.5 找一份描述具有 DMA 功能的 I/O 设备的硬件结构手册。DMA 的主要优势是什么？设备驱动程序是如何将 DMA 存储器用作缓冲区的？

第4章　地址的发现及绑定（ARP）

4.1　引言

在上一章中我们了解了网络接口层的组成，其中包括为网络硬件服务的设备驱动程序，以及与设备驱动程序相关的、用于发送外发分组和接受传入分组的软件。设备驱动程序直接与网络硬件通信，并只能使用网络物理地址发送和接收分组。

本章将详细讨论地址转换协议（ARP）软件，它也同样属于网络接口层。ARP将上层的IP地址与底层的物理地址进行绑定（binding）。地址绑定软件形成了只能使用IP地址的上层协议软件与只能使用物理地址的下层设备驱动程序软件之间的分界线。在稍后的章节中将讨论一些高层协议，那时将很清楚地看出ARP是如何做到使高层协议与物理地址完全独立的。

我们已经提到过地址绑定属于网络接口层，并且在我们的实现方案中也反映出这一点。虽然ARP软件维护管理了一张地址映射表，用以将IP地址映射为物理地址，但高层协议软件并不直接访问该表。事实上，ARP软件拥有地址映射表，并能够用来完成表格数据的查找和修改。

4.2　ARP软件在理论上的结构

从理论上讲，ARP软件可划分为三部分：输出模块、输入模块和一个高速缓存管理程序。在发送数据报时，网络接口软件调用输出模块中的一个过程，将高层协议地址（如IP地址）与相应的物理地址相绑定。输出过程返回一个绑定，网络接口程序利用它封装和发送分组。输入模块处理来自网络的ARP分组，并通过增加新的绑定来修改ARP高速缓存中的内容。高速缓存管理程序实现了高速缓存替换策略：它检查高速缓存中所有的表项，并删除已达到规定时限的表项。

在讲解实现ARP的过程之前，有必要先了解一下它的基本设计思想及用于ARP地址绑定高速缓存的数据结构，详细的讨论将在下一节进行。

4.3　ARP设计方案举例

ARP协议看似简单，但各种细节问题使软件复杂化。许多实现方案并不能完全正确地阐释ARP协议规范，而其他一些则忽略了高速缓存中的超时处理，虽然这样做的本意是试图提高效率，但也会生成错误的绑定。因此全面周密地考虑ARP软件的设计方案，并做到不漏掉协议的任何细节是十分重要的。

我们例举的ARP软件遵循以下几个简单的设计规则：

- **单一的ARP高速缓存**。在一个单一的高速缓存中为所有网络保管表项。高速缓存的每个表项中有一个字段，指出此绑定来自于哪一个网络。与此相反的是多个高速缓存方案，也就是为每个网络接口保留一个独立的ARP高速缓存。使用一个，还是使用多个高速缓存，仅对

网关或连接多个网络的多地址主机来讲是有区别的。

- **全局替换策略**。我们的高速缓存策略规定，如果高速缓存已满，而又必须向其中增加新的绑定时，此高速缓存中的某个表项将被删除，并且，这与新的绑定是否和被删除表项中的绑定来自同一网络无关。另一种办法是本地替换策略，此时新的绑定只能替换来自相同网络的旧的绑定。从本质上说，本地替换策略要求为每个网络接口预分配高速缓存空间，而其效果与使用多个独立的高速缓存是一样的。
- **高速缓存中的超时与删除**。当一个表项在高速缓存中保留一定时间后，重新判定其有效性是十分重要的。在我们的设计方案中，高速缓存的每个表项都有相关的寿命字段。高速缓存中增加新表项（或此表项变为有效）时，ARP软件初始化该表项中的寿命字段。随着时间的推移，高速缓存管理程序递减寿命字段中的值，并当其值达到零时，丢弃该表项。高速缓存中表项的删除与它的使用频率无关。超时表项的丢弃迫使ARP软件必须通过网络，从目的站中获取新的绑定，ARP不会主动使被删除的表项重新有效，要等到某个外发分组需要这个绑定时，ARP软件才会再次获取它。
- **分组以多队列的形式等待发送**。我们的设计允许将多个还未发送出去的分组放置在队列中，等待地址的转换。ARP高速缓存的每个表项都有一个外发分组队列，队列中的分组将被发往由该表项指出的目的站。当含有发送所需要的物理地址的ARP响应分组到达时，软件就从队列中取出这些分组，并把它们发送出去。
- **互斥访问**。我们的软件不允许中断，并避免高速缓存中的内容替换，以保证在任何时间仅有一个进程访问ARP高速缓存。对高速缓存操作的过程（如查找）要求互斥访问，但在该过程中并不包括完成互斥操作的程序代码，互斥机制由调用者完成。

一般说来，为每个接口分配一个独立的高速缓存，或采用本地替换策略，这或多或少地保持了网络接口之间的相对独立性。因为在最糟糕的情况下，如果某个网络接口上的通信量所涉及的目的站数目大大超过了其他网络接口，那么，从使用频繁的网络接口发送来的绑定，通过替换来自其他网络的绑定，占据了大部分高速缓存。其弊病与任何运行不良的高速缓存一样：高速缓存总是保持着100%的容量，但在其中查找到某个表项的概率却很小。我们的设计方案假定管理员会监督网络性能问题，并在出现这样的问题时，分配附加高速缓存空间。

尽管我们的设计方案在最糟糕的情况下可能表现恶劣，但在正常情况下，它可以提供更多的灵活性，因为高速缓存的分配可以动态地跟随网络负载的变化。譬如说，在一段时间内，大部分数据流量仅涉及少数几个网络，这些网络中的主机的绑定将在高速缓存中占主导地位。而如果稍后的通信量又偏向了另一组网络，那么新的这组网络中的主机的表项最终将占据高速缓存的大部分。

4.4　ARP 高速缓存的数据结构

文件arp.h中说明了ARP分组格式的数据结构、ARP高速缓冲存储器的内部数据结构以及在整个ARP程序中使用的符号常量。

```
/* arp.h - SHA, SPA, THA, TPA */

/* Internet Address Resolution Protocol  (see RFCs 826, 920)          */

#define     AR_HARDWARE     1     /* Ethernet hardware type code        */
```

```
/* Definitions of codes used in operation field of ARP packet */

#define AR_REQUEST     1          /* ARP request to resolve address     */
#define AR_REPLY       2          /* reply to a resolve request         */

#define RA_REQUEST     3          /* reverse ARP request (RARP packets) */
#define RA_REPLY       4          /* reply to a reverse request (RARP ")*/

struct  arp       {
        u_short   ar_hwtype;      /* hardware type                      */
        u_short   ar_prtype;      /* protocol type                      */
        u_char    ar_hwlen;       /* hardware address length            */
        u_char    ar_prlen;       /* protocol address length            */
        u_short   ar_op;          /* ARP operation (see list above)     */
        u_char    ar_addrs[1];    /* sender and target hw & proto addrs */
/*      char      ar_sha[???];    - sender's physical hardware address  */
/*      char      ar_spa[???];    - sender's protocol address (IP addr.) */
/*      char      ar_tha[???];    - target's physical hardware address  */
/*      char      ar_tpa[???];    - target's protocol address (IP)      */
};

#define SHA(p)    (&p->ar_addrs[0])
#define SPA(p)    (&p->ar_addrs[p->ar_hwlen])
#define THA(p)    (&p->ar_addrs[p->ar_hwlen + p->ar_prlen])
#define TPA(p)    (&p->ar_addrs[(p->ar_hwlen*2) + p->ar_prlen])

#define MAXHWALEN      EP_ALEN          /* Ethernet                     */
#define MAXPRALEN      IP_ALEN          /* IP                           */

#define ARP_HLEN       8     /* ARR header length                       */

#define ARP_TSIZE      50    /* ARP cache size                          */
#define ARP_QSIZE      10    /* ARP port queue size                     */

/* cache timeouts */

#define ARP_TIMEOUT    600   /* 10 minutes                              */
#define ARP_INF        0X7fffffff      /* infinite timeout value        */
#define ARP_RESEND     1     /* resend if no reply in 1 sec             */
#define ARP_MAXRETRY   4     /* give up after ~30 seconds               */

struct arpentry  {                /* format of entry in ARP cache      */
       short     ae_state;        /* state of this entry (see below)   */
       short     ae_hwtype;       /* hardware type                     */
       short     ae_prtype;       /* protocol type                     */
       char      ae_hwlen;        /* hardware address length           */
       char      ae_prlen;        /* protocol address length           */
```

```
        struct netif *ae_pni;     /* pointer to interface structure     */
        int       ae_queue;       /* queue of packets for this address  */
        int       ae_attempts;        /* number of retries so far       */
        int       ae_ttl;             /* time to live                   */
        u_char    ae_hwa[MAXHWALEN];        /* Hardware address         */
        u_char    ae_pra[MAXPRALEN];        /* Protocol address         */
};

#define AS_FREE       0          /* Entry is unused (initial value)    */
#define AS_PENDING    1          /* Entry is used but incomplete       */
#define AS_RESOLVED   2          /* Entry has been resolved            */

/* RARP variables */

extern int  rarppid;             /* id of process waiting for RARP reply */
extern int  rarpsem;             /* semaphore for access to RARP service */

/* ARP variables */

extern struct         arpentry   arptable[ARP_TSIZE];

/* ARP function declarations */

int               arp_in(struct netif*, struct ep*);
struct arpentry *arpfind(u_char*, u_short, struct netif*);
struct arpentry *arpadd(struct netif*, struct arp*);
```

数组arptable构成了全局ARP高速缓存。数组的每个元素对应于一个协议（IP）地址（ae_pra字段）和物理地址（ae_hwa字段）之间的惟一绑定。ae_state字段给出了表项的状态，这个状态值必定是AS_FREE（此表项当前空闲）、AS_PENDING（此表项正在使用中，但绑定尚未找到）或AS_RESOLVED（此表项正在使用中，绑定正确）三者之一。每个表项中还含有硬件和协议类型字段（ae_hwtype和ae_prtype），以及物理地址和协议地址长度字段（ae_hwlen和ae_prlen）。ae_pni字段指向对应于发送此绑定的网络的网络接口结构。在地址尚未转换的表项中，ae_queue字段指向一个分组队列，当相应的ARP响应分组到达后，队列中的分组就可被发送出去。在状态为AS_PENDING的表项中，ae_attempts字段指出该表项的请求分组已经被广播了多少次。最后，ae_ttl字段给出了在定时器超时，该表项不得不被删除之前，能在高速缓存中保留多长时间（以秒为单位）。

数据结构ARP定义ARP分组的格式。字段ar_hwtype和ar_prtype指出硬件和协议类型，字段ar_hwlen和ar_prlen中的内容分别是整型的物理地址长度和协议地址长度，字段ar_op指明此ARP分组的内容是一个请求还是一个响应。

由于ARP分组中的地址字段长度与硬件类型及被转换的协议地址的类型有关，因此ARP的数据结构不能确定所有字段的长度。事实上，它只是将大小固定的字段放在分组的前部，然后使用一个名为ar_addrs的字段，标志出分组中的剩余内容。就概念上看，以ar_addrs字段打头的一组字节中包括了四个字段：发送方和发送目标的物理地址和协议地址各一对，正如在定义文件中注释说明的那样。每个地址字段的长度可由分组首部大小固定的字段所提供的信息来决定，因而能够有效地计算出每个地址的正确位置。计算分别由内部函数SHA、SPA、THA和TPA完成。每个函数均以ARP分组的地址作为其惟一的入口参数，并返回分组中与函数名对应的地址字段所处的位置。

4.5 ARP输出处理

4.5.1 搜索ARP高速缓存

处理输出的网络接口程序采用地址转换协议（ARP）将IP地址转换为对应的物理地址。实际上，网络输出进程调用arpfind过程，搜索并找出在ARP高速缓存中与某个协议地址相匹配的表项。

```
/* arpfind.c - arpfind */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 * arpfind - find an ARP entry given a protocol address and interface
 *------------------------------------------------------------------------
 */
struct arpentry * arpfind(pra, prtype, pni)
u_char          *pra;
u_short         prtype;
struct netif    *pni;
{
        struct arpentry *pae;
        int         i;

        for (i=0; i<ARP_TSIZE; ++i) {
                pae = &arptable[i];
                if (pae->ae_state == AS_FREE)
                        continue;
                if (pae->ae_prtype == prtype &&
                    pae->ae_pni == pni &&
                    BLKEQU(pae->ae_pra, pra, pae->ae_prlen))
                    return pae;
        }
        return 0;
}
```

参数pra指向被转换的上层（协议）地址，参数prtype给出地址类型（采用ARP协议类型标准值），而参数pni指向一个网络接口结构。arpfind在高速缓存中线性查找，直至找到与指定的IP地址相匹配的表项。它返回指向该表项的指针。

回想一下，我们的设计方案是将所有的ARP绑定全部保存在惟一的一张表格中。对于类似以太网的技术，由于它们的物理地址是全球惟一的，使用单张表格不会出现问题。但有些网络技术允许在各自独立的多个物理网络中重复使用一个物理地址。因此，一个网关有可能会对它的高速缓存中的某个物理地址（如地址5）做出多种反应。参数pni可保证arpfind选择出对应于正确的网络接

口的绑定。因此从理论上来讲，我们在范例中其实采用了物理地址结合网络接口号的手段，来惟一地识别一个表项。

4.5.2　ARP 请求分组的广播

一旦在ARP高速缓存中为某个IP地址分配了一个表项后，网络接口软件调用过程arpsend，来构成并广播一个用来获取相应物理地址的ARP请求分组。

```
/* arpsend.c - arpsend */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 * arpsend - broadcast an ARP request
 *     N.B. Assumes interrupts disabled
 *------------------------------------------------------------------------
 */
int arpsend(struct arpentry *pae)
{
        struct netif *pni = pae->ae_pni;
        struct ep    *pep;
        struct arp   *parp;
        int                arplen;

        pep = (struct ep *) getbuf(Net.netpool);
        if ((int)pep == SYSERR)
             return SYSERR;
        memcpy(pep->ep_dst, pni->ni_hwb.ha_addr, pae->ae_hwlen);
        pep->ep_type = EPT_ARP;
        pep->ep_order = EPO_NET;
        parp = (struct arp *) pep->ep_data;
        parp->ar_hwtype = hs2net(pae->ae_hwtype);
        parp->ar_prtype = hs2net(pae->ae_prtype);
        parp->ar_hwlen = pae->ae_hwlen;
        parp->ar_prlen = pae->ae_prlen;
        parp->ar_op = hs2net(AR_REQUEST);
        memcpy(SHA(parp), pni->ni_hwa.ha_addr, pae->ae_hwlen);
        memcpy(SPA(parp), &pni->ni_ip, pae->ae_prlen);
        memset(THA(parp), 0, pae->ae_hwlen);
        memcpy(TPA(parp), pae->ae_pra, pae->ae_prlen);
        arplen = ARP_HLEN + 2*(parp->ar_hwlen + parp->ar_prlen);
        write(pni->ni_dev, pep, EP_HELEN + arplen);
        return OK;
}
```

arpsend的入口参数是一个指针，指向高速缓存中的一个表项。ARP为该表项中的IP地址生成一个ARP请求分组，并发送请求分组。这段程序代码比你认为的要简单得多。为这个请求分组分配到一个缓冲区之后，arpsend填写分组中的每个字段，其中大多数所需信息来自参数pae指向的高速缓存的表项中。它使用了物理广播地址作为其目的地址，并指明这是一个ARP请求分组（AR_REQUEST）。当物理地址和协议地址的长度字段被赋值后，arpsend就可以利用内部函数SHA、SPA、THA和TPA来计算ARP分组中各个变长地址字段的实际位置。

arpsend生成ARP请求分组后，唤醒系统调用write来发送分组。

4.5.3 输出过程

过程netwrite接收将要被发往某个指定网络接口的分组。

```
/* netwrite.c - netwrite */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#include <q.h>

struct  arpentry  *arpalloc();
struct  arpantry  *arpfind(u_char*, u_short, struct netif*)
int     arpsend (struct arpentry*);
int     local_out(struct ep*);

/* #define    DEBUG */

/*------------------------------------------------------------------------
 * netwrite - write a packet on an interface, using ARP if needed
 *------------------------------------------------------------------------
 */
int netwrite(pni, pep, len)
struct  netif     *pni;
struct  ep        *pep;
unsigned          len;
{
        struct   arpentry   *pae;
        STATWORD            ps;

        if (pni->ni_state != NIS_UP) {
                freebuf(pep);
                return SYSERR;
        }
        pep->ep_len = len;

# ifdef DEBUG
```

```
 if (pni != &nif[NI_LOCAL])
 {
 struct ip *pip = (struct ip*)pep->ep_data;
         if (pip->ip_proto == TPT_OSPF){
                 struct ospf *po = (struct espf*)pip->ip_data;
/*               if(po->ospt_type != T_HELLO) { */
{
                        kprintf("netwrite(pep %x, len %d)\n", pep, len);
                        pdump(pep);
                 }
         }
}
#endif  /* DEBUG */
        if (pni == &nif[NI_LOCAL])
                return local_out(pep);
        else if (isbrc(pep->ep_nexthop)) {
                memcpy(pep->ep_dst, pni->ni_hwb.ha_addr, EP_ALEN);
                write(pni->ni_dev, pep, len);
                return OK;
}
/* else, look up the protocol address... */

disable(ps);
pae = ((u_char*)&pep->ep_nexthop, pep->cp_type, pni);
if (pae && pae->ae_state == AS_RESOLVED) {
        memcpy(pep->ep_dst, pae->ae_hwa, pae->ae_hwlen);
        restore(ps);
        return write(pni->ni_dev, pep, len);
}
if (IP_CLASSD(pep->ep_nexthop)) {
        retore(ps);
        return SYSERR;
}
if (pae == 0) {
        pae = arpalloc();
        pae->ae_hwtype = AR_HARDWARE;
        pae->ae_prtype = EPT_IP;
        pae->ae_hwlen = EP_ALEN;
        pae->ae_prlen = IP_ALEN;
        pae->ae_pni = pni;
        pae->ae_queue = EMPTY;
        memcpy(pae->ae_pra, &pep->ep_nexthop, pae->ae_prlen);
        pae->ae_attempts = 0;
        pae->ae_ttl = ARP_RESEND;
        arpsend(pae);
```

```
        }
        if (pae->ae_queue == EMPTY)
                pae->ae_queue = newq(ARP_QSIZE, QF_NOWAIT);
        if (enq(pae->ae_queue, pep, 0) < 0)
                freebuf(pep);
        restore(ps);
        return OK;
}
```

netwrite调用arpfind，为分组中的目的地址在高速缓存中查找相应表项。如果此表项中的地址已被正确转换，netwrite将其中的物理地址复制到分组中，并调用write发送此分组。如果这个表项既没有被正确转换，也没有等待转换，那么netwrite调用arpalloc分配一个ARP请求，然后netwrite填写ARP表项中的各个字段，并调用arpsend来广播请求分组。

因为netwrite必须做到无延迟地返回主调程序，所以它让分组在一个与高速缓存中的某个表项相关联的队列中等待表项中地址的转换。netwrite首先检查这个队列是否存在。如有必要，它调用newq建立一个新队列。最后，netwrite调用enq将分组置入队列中，以便在地址正确转换后发送出去。每个输出队列都有大小限制。如果netwrite在向队列中添加一个分组时发现队列已满，则丢弃此分组。

4.6 ARP输入处理

4.6.1 向表中增加已转换的表项

ARP在输入处理中使用了两个实用过程arpadd和arpqsend。arpadd读取由网络传送来的ARP分组，为它在高速缓存中分配一个表项，并利用ARP分组中的信息填写该表项。由于它同时填写了物理地址字段和协议地址字段，所以arpadd将该表项的状态字段赋值为AS_RESOLVED。它还要为寿命字段赋予最大超时时间，ARP_TIMEOUT。

```
/* arpadd.c - arpadd */

#include <conf.h>
#include <kernel.h>
#include <network.h>

struct arpentry  *arpalloc(void);

/*------------------------------------------------------------------------
 * arpadd - Add a RESOLVED entry to the ARP cache
 *     N.B. Assumes interrupts disabled
 *------------------------------------------------------------------------
 */
struct arpentry *arpadd(pni, parp)
struct netif   *pni;
struct arp     *parp;
```

```
{
        struct    arpentry   *pae;

        pae = arpalloc();

        pae->ae_hwtype = parp->ar_hwtype;
        pae->ae_prtype = parp->ar_prtype;
        pae->ae_hwlen = parp->ar_hwlen;
        pae->ae_prlen = parp->ar_prlen;
        pae->ae_pni = pni;
        pae->ae_queue = EMPTY;
        memcpy(pae->ae_hwa, SHA(parp), parp->ar_hwlen);
        memcpy(pae->ae_pra, SPA(parp), parp->ar_prlen);
        pae->ae_ttl = ARP_TIMEOUT;
        pae->ae_state = AS_RESOLVED;
        return pae;
}
```

4.6.2　发送等待发送的分组

我们已经知道，ARP 的输出过程将正在等待地址转换的分组保存在队列中。当一个含有地址转换所需要的信息的 ARP 分组到达时，ARP 输入过程调用 arpqsend 发送正在等待发送的分组。

```
/* arpqsend.c - arpqsend */

#include <conf.h>
#include <kernel.h>
#include <network.h>

int netwrite(struct netif*, struct ep*, unsigned);

/*------------------------------------------------------------------------
 * arpqsend - write packets queued waiting for an ARP resolution
 *------------------------------------------------------------------------
 */
void arpqsend(pae)
struct arpentry   *pae;
{
        struct     ep    *pep;
        struct     netif*pni;

        if (pae->ae_queue == EMPTY)
             return;

        pni = pae->ae_pni;
        while (pep = (struct ep *)deq(pae->ae_queue))
                netwrite(pni, pep, pep->ep_len);
```

```
        freeq(pae->ae_queue);
        pae->ae_queue = EMPTY;
}
```

arpqsend并不直接传送等待发送的分组，而是遍历队列，取出其中的数据分组并调用netwrite将它们逐个放入网络输出队列中（网络设备将从其中取出并发送分组）。一旦队列中所有的分组都被取出，arpqsend调用freeq释放队列本身。

4.6.3 ARP输入过程

正如我们已经看到的，当一个ARP分组到达时，网络设备驱动程序将它传递给过程arp_in做进一步处理。

```
/* arp_in.c - arp_in */

#include <conf.h>
#include <kernel.h>
#include <network.h>

void arpqsend(struct arpentry*);

/*------------------------------------------------------------------------
 *  arp_in  -  handle ARP packet coming in from Ethernet network
 *     N.B. - Called by ni_in-- SHOULD NOT BLOCK
 *------------------------------------------------------------------------
 */
int arp_in(pni, pep)
struct netif   *pni;
struct ep      *pep;
{
    struct arp        *parp = (struct arp *)pep->ep_data;
    struct arpentry   *pae;
    int               arplen;

    parp->ar_hwtype = net2hs(parp->ar_hwtype);
    parp->ar_prtype = net2hs(parp->ar_prtype);
    parp->ar_op = net2hs(parp->ar_op);

    if (parp->ar_hwtype != pni->ni_hwtype ||
        parp->ar_prtype != EPT_IP) {
            freebuf(pep);
            return OK;
}

 if (pae = arpfind(SPA(parp), parp->ar_prtype, pni)) {
        memcpy(pae->ae_hwa, SHA(parp), pae->ae_hwlen);
        pae->ae_ttl = ARP_TIMEOUT;
```

```
    }
    if (memcmp(TPA(parp), &pni->ni_ip, IP_ALEN)) {
            freebuf(pep);
            return OK;
    }
    if (pae == 0)
            pae = arpadd(pni, parp);
    if (pae->ae_state == AS_PENDING) {
            pae->ae_state = AS_RESOLVED;
            arpqsend(pae);
    }
    if (parp->ar_op == AR_REQUEST) {
            parp->ar_op = AR_REPLY;
            memcpy(TPA(parp), SPA(parp), parp->ar_prlen);
            memcpy(THA(parp), SHA(parp), parp->ar_hwlen);
            memcpy(pep->ep_dst, THA(parp), EP_ALEN);
            memcpy(SHA(parp), pni->ni_hwa.ha_addr,
                    pni->ni_hwa.ha_len);
            memcpy(SPA(parp), &pni->ni_ip, IP_ALEN);

            parp->ar_hwtype = hs2net(parp->ar_hwtype);
            parp->ar_prtype = hs2net(parp->ar_prtype);
            parp->ar_op = hs2net(parp->ar_op);

            arplen = ARP_HLEN + 2*(parp->ar_prlen + parp->ar_hwlen);

            write(pni->ni_dev, pep, arplen);
        } else
            freebuf(pep);
        return OK;
}
```

协议标准规定，ARP应丢弃任何含有设备不承认的上层协议类型的信息。因此，在我们的arp_in范例中，只承认协议地址类型为IP且物理地址类型与发送该分组的网络接口类型相一致的ARP分组。如果接收到的分组中含有其他类型的地址，arp_in将其丢弃。

在处理一个有效的ARP分组时，arp_in调用arpfind，在ARP高速缓存中搜索与发送方的IP地址相匹配的表项。协议规定接收方应首先利用传入的请求分组来满足正在等待中的表项（即它应当利用发送方的地址修改自己的高速缓存）。因此，如果找到了一个匹配的表项，arp_in将利用分组中的发送方物理地址字段来修改表项中的物理地址，并将表项中的超时字段设置为ARP_TIMEOUT。

协议还规定，如果传入的ARP分组中含有一个指向接收方的请求，接收方也必须在其高速缓存中增加这个发送方的地址（即便接收方并没有等待此地址转换的表项）。因此，arp_in会查看分组中的IP目的地址是否与本机的IP地址相符。如果相符，arp_in调用arpadd将其插入高速缓存中。在高速缓存中插入一个表项后，arp_in将查看其中的地址是否在等待转换。如果是，它调用arpqsend

发送队列中等待发送的分组。

最后，arp_in查看分组中是否含有请求。如果有，arp_in通过交换分组中的目标地址和发送地址，向发送方提供被请求的物理地址，然后将操作由AR_REQEST改为AR_REPLY。arp_in直接发送响应分组。

4.7 ARP高速缓存的管理

到目前为止，我们已集中讨论了输入和输出的处理过程。然而，其中涉及到的ARP高速缓存的管理要求输入和输出软件之间的协调。此外它还需要定期地做一些与输入输出无关的计算。在下一节中，将讲解高速缓存的管理策略，并使你了解软件是如何实现这一策略的。

4.7.1 高速缓存表项的分配

如果一个进程（如IP进程）需要发送一份数据报，但其目的地址不在ARP高速缓存的任何一个表项中，则IP必须创建一个新表项，然后广播相应的请求分组，并将等待发送的分组置入队列中。过程arpalloc在ARP高速缓存中选取一个表项，用于存放新的绑定。

```
/* arpalloc.c - arpalloc */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

void arpdq(struct arpentry*);

/*------------------------------------------------------------------------
 * arpalloc - allocate an entry in the ARP table
 *     N.B. Assumes interrupts DISABLED
 *------------------------------------------------------------------------
 */
struct arpentry *arpalloc()
{
        static     int  aenext = 0;
        struct     arpentry *pae;
        int  i;

        for (i=0; i<ARP_TSIZE; ++i) {
             if (arptable[aenext].ae_state == AS_FREE)
             break;
             aenext = (aenext + 1) % ARP_TSIZE;
        }
        pae = & arptable[aenext];
        aenext = (aenext + 1) % ARP_TSIZE;
```

```
        if (pae->ae_state == AS_PENDING && pae->ae_queue >= 0)
            arpdq(pae);
        pae->ae_state = AS_PENDING;
        return pae;
}
```

arpalloc 实现了高速缓存替换策略，因为当它为新表项寻找空间时，必须决定在已满的高速缓存中删除哪一个旧表项。我们选择了一种简单的替换策略：

> 在为 ARP 高速缓存中的新成员分配空间时，如果存在一个空闲表项，就选择此空闲表项。否则，采用循环法（round-robin）删除旧表项。

也就是说，每当高速缓存管理程序选取了一个表项并将其删除后，就依次移向下一个表项。当它再次回到某一表项之前，必定已循环经历了高速缓存的所有表项。因此，一旦高速缓存管理程序删除了一个旧表项，并将其分配给一个新的绑定后，就不再理会这一表项，直到它不得不删除并替换了其他所有的绑定后。

在考虑 ARP 高速缓存替换策略时，必须注意到，高速缓存全满的情况总是不受欢迎的，因为这意味着系统操作处于饱和状态。这时，若为了发送一个数据报而需要在高速缓存中插入一个新的绑定时，系统必须删除表中的某个绑定。当再次要用到这个旧的、被删除的绑定时，ARP 还得删除表中的另一个绑定，并广播请求分组。在最糟糕的情况下，每发送一个数据报，ARP 都需要广播请求分组。我们假设系统管理员在监察到类似的情况后，将为系统重新配置一个更大的高速缓存。这样，需要不断抢占表中某一项目的情况将很少发生，因而我们所列举的循环法在实际中工作良好。

要完成抢占策略，arpalloc 中有一个静态整型变量 aenext。arpalloc 中的 for 循环语句，从以 aenext 为索引的表项开始，搜索整张表格，表格的头尾衔接，结束时回到出发点。如果找到了一个空闲表项，搜索立即终止。如果高速缓存中没有闲置空间，arpalloc 删除索引 aenext 对应的表项。最后，arpalloc 将 aenext 的值加 1，使下一轮查找的起点跨过新分配的表项。

4.7.2　高速缓存的定期维护管理

在我们的设计方案中，安排了一个独立的定时进程，以定期执行过程 arptimer。

```
/* arptimer.c - arptimer */

#include <conf.h>
#include <kernel.h>
#include <network.h>

void arpdq (struct arpentry*);
int arpsend (struct arpentry*)'

/*------------------------------------------------------------------------
 * arptimer - Iterate through ARP cache, aging (possibly removing) entries
 *------------------------------------------------------------------------
 * gran  - time since last iteration
 */
```

```
void arptimer(gran)
int     gran;
{
        struct arpentry *pae;
        STATWORD        ps;
        int             i;

        disable(ps);    /* mutex */

        for (i=0; i<ARP_TSIZE; ++i) {
                if ((pae = &arptable[i])->ae_state == AS_FREE)
                        continue;
                if ((pae->ae_ttl == gran) <= 0)
                if (pae->ae_state == AS_RESOLVED)
                        pae->ae_state = AS_FREE;
                else if (++pae->ae_attempts > ARP_MAXRETRY) {
                        pae->ae_state = AS_FREE;
                        arpdq(pae);
                } else {
                        pae->ae_ttl = ARP_RESEND;
                        arpsend(pae);
                }
        }
        restore(ps);
}
```

当定时进程调用arptimer时，将向其以参数gran的形式给出此次调用和上一次调用之间的时间间隔。arptimer利用这个间隔时间值，计算每个表项在高速缓存中的存在时间。这样，通过调用，arptimer遍历每个表项，并将表项的寿命字段中的值减去gran，其中gran指出从上次遍历到现在已经过了多少秒。如果寿命字段变为零或负数，arptimer从高速缓存中删除此表项。删除一个地址已经转换的表项，仅仅是指将其状态改为AS_FREE，这样就允许arpalloc在下次需要时能够使用该表项。如果一个正在等待地址转换的表项中的寿命字段超时，arptimer检查ae_attempts字段，看它是否已被广播了ARP_MAXRETRY次。如果没有，则arptimer调用arpsend再次广播此请求。如果请求已被广播了ARP_MAXRETRY次，arptimer释放相关的等待发送的分组队列，然后删除该表项。

4.7.3 释放队列中的分组

在ARP高速缓存全满的情况下，被arpalloc选中删除的表项可能有一个与之相关联的外发分组队列。如果是这样，arpalloc调用arpdq从队列中取出分组并丢弃。

```
/* arpdq.c - arpdq */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```

```
/*------------------------------------------------------------------------
 * arpdq - destroy an arp queue that has expired
 *------------------------------------------------------------------------
 */
void arpdq(pae)
struct arpentry *pae;
{
        struct     ep    *pep;
        struct     ip    *pip;

        if (pae->ae_queue < 0)          /* nothing to do */
             return;

        while (pep = (struct ep *)deq(pae->ae_queue)) {
             if (gateway && pae->ae_prtype == EPT_IP) {
                  pip = (struct ip *)pep->ep_data;
                  icmp(ICT_DESTUR, ICC_HOSTUR, pip->ip_src, pep, 0);
             } else
                  freebuf(pep);
        }
        freeq(pae->ae_queue);
        pae->ae_queue = EMPTY;
}
```

arpdq遍历与ARP高速缓存中的某个表项相关联的分组队列，并逐一丢弃它们。如果被丢弃的分组是一个IP数据报，且本机是一个网关，arpdq调用icmp过程为该数据报生成一个ICMP“目的站不可达”信息。最后，arpdq调用freeq释放队列本身。

4.8　ARP初始化

系统在启动时，调用一次arpinit过程。arpinit产生与RARP一起使用的互斥信号量rarpsem，并将ARP高速缓存中所有表项的状态设为AS_FREE。另外，arpinit为相应的RARP协议初始化几个数据项，这些与本章中的程序代码关系不大。注意arpinit并不初始化定时器进程或设立arptimer的调用。将这些细节分离出去，是因为在我们的范例中，众多的协议共同使用着惟一的一个定时器进程。

```
/* arpinit.c - arpinit */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  arpinit  -  initialize data structures for ARP processing
```

```
 *------------------------------------------------------------------------
 */
void arpinit()
{
        int   i;

        rarpsem = screate(1);
        rarppid = BADPID;

        for (i=0; i<ARP_TSIZE; ++i)
             arptable[i].ae_state = AS_FREE;
}

int     rarpsem;
int     rarppid;

struct arpentry   arptable[ARP_TSIZE];
```

4.9 ARP参数配置

在编写ARP软件时，程序员通过选取参数值来设置系统，如：

- ARP高速缓存的大小
- 发送方等待一个ARP响应的时间限制
- 允许发送方重发一个请求分组的次数
- 重发请求分组的时间间隔
- 高速缓存表项的（寿命字段）超时时间
- 等待发送的分组队列长度

在典型的设计中，为这些参数定义了一些符号常量，如高速缓存的大小，这样系统管理员就能够根据具体的安装变换其设置。对于要求管理员做更多控制操作的安装，可以编写一些实用程序，使管理员能够在运行中改变设置。例如，在一些软件中，管理员可以检查ARP高速缓存、删除表项以及修改字段值（如寿命字段）等。然而有一些参数不能被轻易改变。例如，许多程序员在固定的重发时延或按指数规律变化的退避时间之间进行选择，并将其选择嵌入到程序代码中，就如我们的实例程序一样。

4.10 小结

我们的ARP范例采用了惟一的全局高速缓存，用以保存来自所有网络的绑定。它允许多个分组以队列的形式等待地址的转换，并利用一个独立的定时器过程计算高速缓存中表项的生存时间。所有表项最终都会超时。当一个新表项必须被插入高速缓存时，如果高速缓存已满，则必须删除一个旧表项。在我们的设计方案中，采用了循环法的替换策略，并通过一个全局指针来实现。每当高速缓存中的一个表项被选中后，此指针依次移向下一个表项。范例代码提供了高速缓存数据结构的

定义，以及在这些数据结构上操作的各个过程。

深入研究

Plummer［RFC 826］定义了ARP标准，而Clark［RFC 814］一般性地讨论了地址及绑定。Parr［RFC 1029］的内容涉及容错地址转换。Malkin［RFC 1868］提出了UNARP。Laubach［RFC 1577］讨论了非广播网络中的ARP，Heinanen和Govindan［RFC 1735］对此进行了进一步的讨论。

习题

4.1　什么网络硬件使用ARP?

4.2　为不使用ARP的网络接口构思一份地址绑定软件设计草图。

4.3　在网关中采用单张表格来保存ARP高速缓存的主要缺点是什么？主要优点又是什么?

4.4　设想某网点决定对仅在给定的网络中惟一的地址使用ARP（虽然不用ARP也可能实现地址绑定）。我们的范例程序能在与两个这种网络相连接的一个网关上正确地运行吗？为什么?

4.5　以太网硬件规范规定最小分组的大小为60八位组。仔细阅读操作系统中以太网设备驱动软件，考虑驱动程序是怎样发送少于60八位组的分组的?

4.6　如果ARP软件不允许多个分组以队列的形式等待一个ARP的绑定过程，用户会察觉其性能有所不同吗?

4.7　如何为ARP_MAXRETRY、ARP_TIMEOUT以及时效期的粒度（granularity）选择合理的值?

4.8　因为任何设备都有可能响应一个ARP分组的广播，所以ARP容易上当受骗。通过增加判断以下情况的语句来修改范例：

（a）两个或更多个设备响应了某个给定IP地址的请求分组。

（b）一个设备接收到自己的IP地址的ARP绑定。

（c）一个设备响应了多个IP地址请求分组。

4.9　作为解决上一题中提到的电子欺骗问题的另一种方案，在范例软件中增加一个判断语句，以保证ARP分组中SHA字段的物理地址与物理帧中的物理地址一致。这两种方案的优、缺点各是什么?

4.10　阅读有关桥式令牌环网寻址的资料。ARP应该使用本地环（local ring）广播地址，还是全环（all ring）广播地址，为什么?

4.11　如果使用代理ARP，对我们的范例程序应做何改动?

第5章　IP：软件的总体结构

5.1　引言

本章所关心的是实现网际协议（IP）的软件的组织结构。IP提供的功能看似简单，但它的纷繁难懂使软件的实现过程变得复杂，而其精细微妙之处又使软件很难做到准确无误。在讲解IP的过程时，为了不至于因突然涉及所有部分而使人无法接受，我们将具体实现方案分3章讨论。本章给出了数据结构并描述软件的整体结构，还讨论了IP软件的理论性操作以及通过IP层的数据报流程。在稍后的章节中，还要详述选路和差错处理的细节，展示IP软件中的各段程序代码是如何使用这些数据结构的。

5.2　中心环节

从理论上说，IP是整个协议软件的中心环节。它接收来自网络接口软件的传入数据报，同时也接收由上层协议生成的外发数据报。IP在为数据报选择路由后，或者将它发送给一个网络接口，或者交给本机的上层协议。

单从主机的角度来看，人们会很自然地将IP软件分为两个不同的部分考虑：一部分处理输入，另一部分处理输出。输入部分使用IP首部中的PROTO字段来决定应该由上层协议中的哪个模块来接收传入数据报。输出部分使用一张本地选路表来选择外发数据报要送往的下一跳。

将IP分割成输入和输出两部分虽然有其直觉上的吸引力，但它使得IP和上层协议软件之间的沟通变得十分笨拙。另外，IP软件必须能够在网关中工作，而网关中的选路过程要比主机中的更复杂。实际上，由于一个网关必须将发送来的数据报继续转送到它的下一跳，所以网关中的软件都不能轻易地被划分成输入和输出部分。这样，在IP处理传入数据报时可能会产生输出数据。而当发送来的数据报引起错误时，网关还必须生成ICMP差错信息，这更进一步地使输入和输出的界限模糊。在以下的讨论中，将把重点放在网关上，而把主机作为特殊情况对待。

5.3　IP软件设计思想

为保持IP软件的简单性和统一性，在范例中采用的组织结构上的技巧有三种：

- **统一的输入队列及统一的选路过程**。IP过程对必须由它做处理的所有数据报采用相同的输入队列形式，与这些数据报是来自于网络的还是由本机产生的无关。IP不考虑数据报的来源，只是从队列中取出每个数据报并为它们选择路由。一个统一的输入结构使事情变得简单：IP不需要在程序代码中对本地生成的数据报个别对待。再者说，由于IP使用单一的路由算法来为所有的数据报选择路由，这样更便于人们理解一个数据报所采纳的路由。
- **独立的IP进程**。IP软件作为一个单一的、自包含的进程执行。为IP单独创建一个进程使软件容易理解和修改。它使编写出的IP软件能够不依赖于硬件中断或是应用程序调用的那些

过程。

- **本地主机接口**。为避免将发往本机的情况作为特例对待，在我们的实现策略中，为本地传送创建了一个伪网络接口。回想一下，本地接口具有与其他网络接口相同的结构，但它所对应的是本地协议软件，而不是一个物理网络。IP算法为每个数据报选择路由，并将其传递给某个网络接口，包括以本机为目标的数据报。当一个常规网络接口收到一份数据报时，它将该数据报发往某个物理网络。当本地接口接收到一份数据报时，它利用PROTO字段来决定应当由本机上的哪个协议软件模块来接收此数据报。这样，IP认为所有选择路由的过程是统一而且对称的：它从任一接口接收一份数据报，并为其选择路由，然后发送到另一个接口，而不必为本机产生的（或发往本机的）数据报做特殊处理。

虽然建立网关的必要性引发了许多的设计思想，但针对网关的设计在主机上同样有效，并允许在主机和网关上使用同样的程序。显然，统一的路由算法和本机接口的结合，将消除程序代码中对一些特殊情况的特殊处理。更重要的是，由于本机是选路表中可用表项来控制的有效目的站，因而就有可能为其增加访问保护措施，以允许管理员实施某些交付原则。例如，管理员能够轻易地做到允许或不允许指定设备上的两个应用程序之间互相交换信息，如同允许或不允许不同设备上的应用程序之间的通信一样。

5.4　IP 软件结构和数据报流程

第 2 章描述了 IP 软件理论上的组织结构，并给出了输入输出的数据流。这一节我们将展开描述，并补充细节。回想一下，IP 由一个单一的进程和一组网络接口队列组成，数据报必须经过这些队列发送给进程。IP 不断地从其中的一个队列中提取数据报，利用选路表为该数据报选择下一跳，并将数据报送往适当的网络输出进程进行发送。

5.4.1　选择传入数据报的策略

在第 3 章中曾提到过，每个网络接口，包括伪网络接口，都有属于自己的、送往 IP 进程的数据报队列。图 5.1 描画出该数据流。

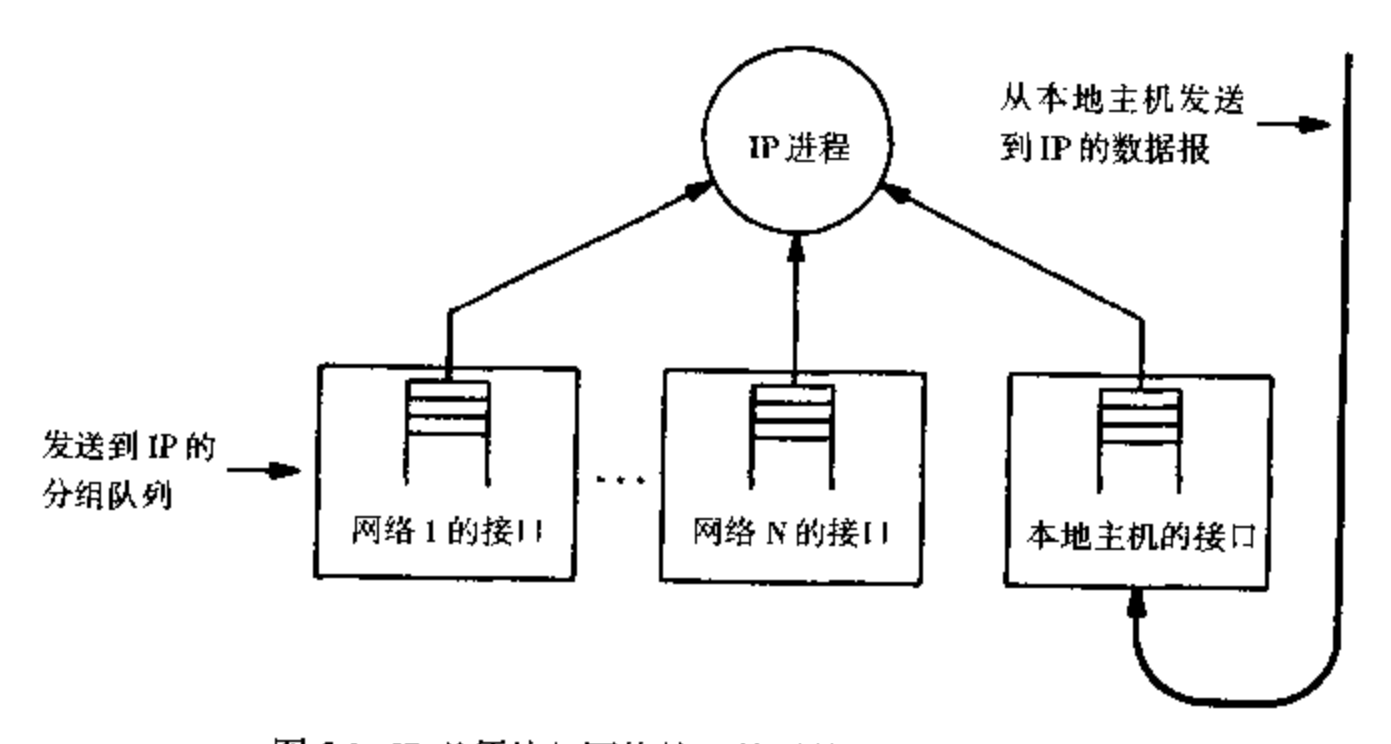


图 5.1　IP 必须从与网络接口关联的队列中选出一个数据报进行处理。伪网络接口提供了本地生成的数据报使用的队列

如果有多个数据报正在输入队列中等待，IP进程必须选取其中之一，并为其选择路由。IP选取哪一个数据报将决定系统的行为：

> 挑选数据报并为其选择路由的IP程序段实现了一个重要策略：它决定了数据报来源的相对优先级。

譬如说，如果IP总是最先选择伪网络接口队列，那么它就给予由本机生成的外发数据报最高优先级。如果IP仅当其他所有队列都空时，才选择伪网络队列，那么通过网络发送过来的数据报就拥有高优先级，而本地生成的数据报的优先级最低。

很显然，这两种极端都是不可取的。一方面，给外来的数据报赋予高优先级将意味着本地软件可能在等待IP为数据报选择路由时，被阻塞任意长的时间。对某个与工作繁忙的网络相连接的网关来说，这种延迟会使本地应用程序，包括网络管理应用程序无法进行通信。另一方面，给本地生成的数据报以高优先级将导致在本机上运行的任何应用程序都要优先于来自网络的IP数据流。如果因错误而导致一个本地应用程序不断地向外发送数据报，这些输出分组将阻止外来数据报到达网络管理软件。因此，管理员也就无法使用网络管理工具来排除故障。

正确的做法是公平分配优先权，使传入和外发数据流在选择路由时享有平等的优先权。我们的实现策略是以循环法（round-robin）来选择数据报，以期达到公平合理。也就是说，它从某个队列中选择出一个数据报，并为其做路由处理之后，继续向前移动并检查下一个队列。如果有K个含有数据报的队列正在等待着路由处理，那么IP在处理完所有K个队列中的第一个数据报之后，才有可能处理到任一队列中的第二个数据报。

过程*ipgetp*实现了循环法选择的策略。

```
/* ipgetp.c - ipgetp */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <proc.h>

static int  ifnext = NI_LOCAL;

/*------------------------------------------------------------------------
 * ipgetp  --  choose next IP input queue and extract a packet
 *------------------------------------------------------------------------
 */
struct ep *ipgetp(pifnum)
int    *pifnum;
{
        struct    ep    *pep;
        int             i;

        recvclr(); /* make sure no old messages are waiting */
        while (TRUE) {
             for (i=0; i < Net.nif; ++i, ++ifnext) {
                  if (ifnext >= Net.nif)
```

```
                ifnext = 0;
                if (nif[ifnext].ni_state == NIS_DOWN)
            continue;
                if (pep = NIGET(ifnext)) {
                *pifnum = ifnext;
                return pep;
            }
        }
        ifnext = receive();
    }
    /* can't reach here */
}
```

正如程序代码所示，静态变量ifnext用作接口数组的索引。它遍历整个网络接口结构。在每个接口中，它检查状态变量ni_state，以确定接口是否已被打开。当ipgetp发现一个打开的接口中有正在等待的数据报时，它就使用宏NIGET来提取并返回第一个数据报。再次调用ipgetp时，将跨过前一次处理的接口，继续查找。

5.4.2　允许 IP 进程被阻塞

过程 ipgetp 包含了一种微妙的优化思想：

> 如果所有输入队列都空，IP进程在调用过程ipgetp时被阻塞。一旦有一个数据报到达，IP进程就恢复执行并立刻检查已有数据报到达的那个接口。

要理解这个优化思想，有必要先了解两个事实。首先，当与某个特定接口相关联的设备驱动程序在它的输入队列中放入一个数据报时，就向IP进程发送一个报文。其次，ipgetp中的循环语句以调用receive结束。在ipgetp遍历了所有网络接口而没有发现任何数据报之后，它调用receive，receive在信息到达之前为阻塞状态。当receive返回时，它以函数值的形式向主调过程传递一个报文。报文中包含了一个指针，指向已有数据报到达的接口。ipgetp将该接口指针赋值给ifnext，并重新开始遍历。

既然已经知道了IP所使用的选择数据报的策略，那么来查看一下IP进程的结构。其基本算法非常简单。IP不断地调用ipgetp来选择一个数据报，接着调用一个过程来计算下一跳的地址，并将数据报置入与将要发送该数据报的网络接口相关联的队列中。

尽管从概念上看似简单，但许多细节问题使程序代码复杂化。例如，如果数据报来自于某个网络，IP必须验证此数据报的校验和是否正确。如果选路表中没有指定的目的站，IP必须生成一个ICMP“目的站不可达”报文。如果选路表指出该数据报应当被送往产生这个数据报的网络中的某个目的站，IP必须生成一个ICMP“重定向”报文。最后，IP必须处理定向广播这一特殊情况，此时IP向指定的网络发送数据报的副本，并向网关自身的上层协议软件也发送一份副本。IP进程从过程 ipproc 的执行开始。

```
/* ipproc.c - ipproc */

#include <conf.h>
#include <kernel.h>
```

```
#include <network.h>

struct ep      *ipgetp(int*);
struct route   *rtget(IPaddr, Bool);

/*-----------------------------------------------------------------------
 *  ipproc  -  handle an IP datagram coming in from the network
 *-----------------------------------------------------------------------
 */
PROCESS
ipproc(void)
{
        struct     ep     *pep;
        struct     ip     *pip;
        struct            route      *prt;
        Bool              nonlocal;
        int               ifnum;

        ippid = getpid();     /* so others can find us        */

        signal(Net.sema);     /* signal initialization done   */

        while (TRUE) {
             pep = ipgetp(&ifnum);
             pip = (struct ip *)pep->ep_data;

             if ((pip->ip_verlen>>4) != IP_VERSION) {
                     IpInHdrErrors++;
                     freebuf(pep);
                     continue;
             }
             if (IP_CLASSE(pip->ip_dst)) {
                     IpInAddrErrors++;
                     freebuf(pep);
                     continue;
             }
             if (ifnum != NI_LOCAL) {
                     if (cksum((WORD *)pip, IP_HLEN(pip))) {
                             IpInHdrErrors++;
                             freebuf(pep);
                             continue;
                     }
                     ipnet2h(pip);
                     pep->ep_order |= EPO_IP;
             }
             prt = rtget(pip->ip_dst, (ifnum == NI_LOCAL));
```

```
if (prt == NULL) {
        if (gateway) {
                iph2net(pip);
                pep->ep_order &= ~EPO_IP;
                icmp(ICT_DESTUR, ICC_NETUR,
                                pip->ip_src, pep, 0);
        } else {
                IpOutNoRoutes++;
                freebuf(pep);
        }
        continue;
}
nonlocal = ifnum != NI_LOCAL && prt->rt_ifnum != NI_LOCAL;
if (!gateway && nonlocal) {
        IpInAddrErrors++;
        freebuf(pep);
        rtfree(prt);
        continue;
}
if (nonlocal)
        IpForwDatagrams++;
/* fill in src IP, if we're the sender */

if (ifnum == NI_LOCAL) {
        if (pip->ip_src == ip_anyaddr)
                if (prt->rt_ifnum == NI_LOCAL)
                        pip->ip_src == pip->ip_dst;
                else
                        pip->ip_src =
                                nif[prt->rt_ifnum].ni_ip;
} else if (--(pip->ip_ttl) == 0 &&
                prt->rt_ifnum != NI_LOCAL) {
        IpInHdrErrors++;
        iph2net(pip);
        icmp(ICT_TIMEX, ICC_TIMEX, pip->ip_src, pep, 0);
        rtfree(prt);
        continue;
}
ipdbc(ifnum, pep, prt);   /* handle directed broadcasts   */
ipredirect(pep, ifnum, prt); /* do redirect, if needed    */
if (prt->rt_metric != 0)
        ipputp(prt->rt_ifnum, prt->rt_gw, pep);
else
        ipputp(prt->rt_ifnum, pip->ip_dst, pep);
```

```
            rtfree(prt);
        }
    }
    int     ippid, gateway, bsdbrc;
```

ipproc先将其进程ID保存在全局变量ippid中，并发出网络初始化信号，然后进入一个无限循环。在每一次循环执行时，ipproc处理一个数据报。它调用ipgetp来选取一个数据报，并将ifnum设置为与该数据报对应的接口索引。在检查了数据报的版本并验证其中没有E类地址后，ipproc调用cksum来验证校验和（除非此数据报是由本机产生的）。

一旦ipproc得到的是一个有效的数据报，它调用过程rtget为该数据报选择路由。在下一章中将详述rtget中的细节。目前，重要的是了解过程rtget计算一个路由并返回指向描述该路由结构的指针。如果路由不存在，ipproc调用过程icmp[1]，生成并发送一个ICMP“目的站不可达”报文。

ipproc必须为本地生成的数据报填写正确的源地址。为做到这一点，它首先检查数据报，确认上层协议是否已为它指定了一个固定的源地址。如果没有，ipproc填写源地址字段。根据标准规定，ipproc将数据报的源地址设置为发送该数据报的网络接口的IP地址。如果路由指向本机接口（例如，该数据报来自本机，选择路由后又回到本机），ipproc将数据报的目的地址复制到源地址字段中。

一旦选路完成，ipproc递减寿命计数器（ip_ttl）。如果寿命字段的值达到0，ipproc生成一个ICMP超时报文。

ipproc调用过程ipdbc来处理定向广播。如第5.4.5节中所示，ipdbc复制到达本机的定向广播数据报，并向本地软件发送一个副本。ipproc将副本原件发往指定网络。

必要时ipproc还要调用过程ipredirect[2]生成ICMP“重定向”报文。为了决定是否需要重新发送这个报文，ipproc比较两个网络接口，一个是将数据报发送来的网络接口，另一个是数据报在选择路由后将要发往的接口，如果它们是相同的，就需要重定向。ipredirect检查网络的子网掩码，决定它应该发送一个网络重定向路由还是一个主机重定向路由报文。

最后，ipproc检查路由度量值，决定该数据报应当被直接发往自己的目的站，还是先发送到下一跳。度量值为0的路由表示网关可以直接将数据报发送到它的目的站，其他任何大于0的值表示网关应当先将数据报发往下一跳。不管是选择了下一跳的地址，还是选择了数据报的目的地址，ipproc调用ipputp将数据报插入某个网络输出队列中。

5.4.3 IP使用的常量的定义

在文件ip.h中定义了IP软件使用的符号常量。另外，它还用结构IP定义了IP数据报的格式。

```
/* ip.h - IP_HLEN, IP_CLASS{A,B,C,D,E} */

/* Internet Protocol (IP)  Constants and Datagram Format                */

#define IP_ALEN  4                  /* IP address length in bytes (octets)*/
typedef unsigned long IPaddr;       /*  internet address                  */

#if    BYTE_ORDER == BIG_ENDIAN
```

① 第8章描述了icmp的实现过程。
② 过程ipredirect请参见第8章。

```
#defind IP_CLASSA(x) (((x) & 0x80000000) == 0)          /* IP Class A */
#defind IP_CLASSB(x) (((x) & 0xc0000000) == 0x80000000) /* IP Class B */
#defind IP_CLASSC(x) (((x) & 0xe0000000) == 0xc0000000) /* IP Class C */
#defind IP_CLASSD(x) (((x) & 0xf0000000) == 0xe0000000) /* IP Class D */
#defind IP_CLASSE(x) (((x) & 0xf0000000) == 0xf0000000) /* IP Class E */
#else  /* BYTE_ORDER */
#define IP_CLASSA(x)  (((x & 0x80) == 0x00)      /* IP Class A address */
#define IP_CLASSB(x)  (((x & 0xc0) == 0x80)      /* IP Class B address */
#define IP_CLASSC(x)  (((x & 0xe0) == 0xc0)      /* IP Class C address */
#define IP_CLASSD(x)  (((x & 0xf0) == 0xe0)      /* IP Class D address */
#define IP_CLASSE(x)  (((x & 0xf8) == 0xf0)      /* IP Class E address */
#endif  /* BYPE_ORDER */

/* Some Assigned Protocol Numbers */

#define IPT_ICMP      1    /* protocol type for ICMP packets        */
#define IPT_IGMP      2    /* protocol type for IGMP packets        */
#define IPT_TCP       6    /* protocol type for TCP packets         */
#define IPT_EGP       8    /* protocol type for EGP packets         */
#define IPT_UDP       17   /* protocol type for UDP packets         */
#define IPT_OSPF      89   /* protocol type for OSPF packets        */

struct ip   {
      u_char   ip_verlen;  /* IP version & header length (in longs)  */
      u_char   ip_tos;     /* type of service                        */
      u_short  ip_len;     /* total packet length (in octets)        */
      short    ip_id;      /* datagram id                            */
      short    ip_fragoff; /* fragment offset (in 8-octet's)         */
      u_char   ip_ttl;     /* time to live, in gateway hops          */
      u_char   ip_proto;   /* IP protocol (see IPT_* above)          */
      short    ip_cksum;   /* header checksum                        */
      IPaddr   ip_src;     /* IP address of source                   */
      IPaddr   ip_dst;     /* IP address of destination              */
      u_char   ip_data[1]; /* variable length data                   */
};

#define IP_VERSION    4          /* current version value            */
#define IP_MINHLEN    5          /* minimum IP header length (in longs) */
#define IP_TTL        255        /* Initial time-to-live value       */

#define IP_MF         0x2000     /* more fragments bit               */
#define IP_DF         0x4000     /* don't fragment bit               */
#define IP_FRAGOFF    0x1fff     /* fragment offset mask             */
#define IP_PREC       0xe0       /* precedence portion of TOS        */

/* IP Precedence values */
```

```
#define IPP_NETCTL      0xe0        /* network control                    */
#define IPP_INCTL       0xc0        /* internet control                   */
#define IPP_CRIT        0xa0        /* critical                           */
#define IPP_FLASHO      0x80        /* flash over-ride                    */
#define IPP_FLASH       0x60        /* flash                              */
#define IPP_IMMED       0x40        /* immediate                          */
#define IPP_PRIO        0x20        /* priority                           */
#define IPP_NORMAL      0x00        /* normal                             */

/* macro to compute a datagram's header length (in bytes)                */
#define IP_HLEN(pip)    ((pip->ip_verlen & 0xf)<<2)
#define IPMHLEN         20          /* minimum IP header length (in bytes) */

/* IP options */
#define IPO_COPY        0x80        /* copy on fragment mask              */
#define IPO_CLASS       0x60        /* option class                       */
#define IPO_NUM         0x17        /* option number                      */

#define IPO_EOOP        0x00        /* end of options                     */
#define IPO_NOP         0x01        /* no operation                       */
#define IPO_SEC         0x82        /* DoD security/compartmentalization */
#define IPO_LSRCRT      0x83        /* loose source routing               */
#define IPO_SSRCRT      0x89        /* strict source routing              */
#define IPO_RECRT       0x07        /* record route                       */
#define IPO_STRID       0x88        /* stream ID                          */
#define IPO_TIME        0x44        /* internet timestamp                 */

#define IP_MAXLEN       BPMAXB-EP_HLEN  /* Maximum IP datagram length     */

/* IP process info */

extern int              ipproc();
#define IPSTK           512         /* stack size for IP process          */
#define IPPRI           100         /* IP runs at high priority           */
#define IPNAM           "ip"        /* name of IP process                 */
#define IPARGC          0           /* count of args to IP                */

extern IPaddr     ip_maskall;       /* = 255.255.255.255                  */
extern IPaddr     ip_anyaddr;       /* = 0.0.0.0                          */
extern IPaddr     ip_loopback;      /* = 127.0.0.1                        */

extern int  ippid, gateway;

struct ip *iph2net(struct ip *), *ipnet2h(struct ip *);
unsigned short cksum(WORD *, unsigned);
int ipsend(Ipaddr, struct ep *, unsigned, u_char, u_char, u_char);
int ipputp(unsigned, IPaddr, struct ep  *);
Bool isbrc(IPaddr);
```

5.4.4　校验和的计算

ipproc利用过程cksum来计算或验证数据报的首部校验和。首部校验和将首部视为16位整数的序列，并把校验和定义为对首部中所有 16 位整数各求反码，并将结果相加，再对求得的和计算一次二进制反码。得到的和数及反码定义为使用二进制反码算法。

大多数设备使用二进制补码算法，因此仅累加得出一个16位的校验和并不能得到正确的结果。为了便于移植并避免以汇编语言编写程序，过程 cksum 使用了 C 语言编写。在范例中使用了 32 位（长整型）算法来累加得到一个和，然后通过向这个整数和中增加所有进位位的办法，将结果折合成一个 16 位的值。最后，cksum 返回结果的反码。

```
/* cksum.c - cksum */

/*------------------------------------------------------------------------
 *  cksum  -  Return 16-bit ones complement of 16-bit ones complement sum
 *------------------------------------------------------------------------
 */
short cksum(buf, nwords)
unsigned short   *buf;
int              nwords;
{
        unsigned long   sum;

        for (sum=0; nwords>0; nwords--)
                sum += *buf++;
        sum = (sum >> 16) + (sum & 0xffff);  /* add in carry   */
        sum += (sum >> 16);                  /* maybe one more */
        return ~sum;
}
```

5.4.5　处理定向广播

只要一个数据报发送到定向广播地址，则在指定的目的网络中的所有机器都必须接收到一份副本。关键是要记住：

> 定向广播包括了目的网络上的网关和主机，即便是负责在网络上继续向前发送该数据报的网关也不能例外。

然而，大多数网络硬件不会将广播分组的副本再次传送回发出这个广播的机器。如果一个网关需要一份广播数据报的副本，那么软件必须采取明确的措施，以保留一份副本。因此，如果一个网关接收到的数据报的目的地址是直接与这个网关相连的某个网络的定向广播地址，那么这个网关必须做两件事：（1）为本机的协议软件复制一份数据报。（2）在指定的网络上广播该数据报。过程 ipdbc 中包含了处理这种广播数据报的程序代码。

```
/* ipdbc.c - ipdbc */

#include <conf.h>
```

```
#include <kernel.h>
#include <network.h>

struct route *rtget(Ipaddr, Bool);
/*-----------------------------------------------------------------------
 * ipdbc - handle IP directed broadcast copying
 *-----------------------------------------------------------------------
 */
void ipdbc(ifnum, pep, prt)
unsigned        ifnum;
struct  ep      *pep;
struct  route   *prt;
{
        struct  ip      *pip = (struct ip *)pep->ep_data;
        struct          ep *pep2;
        struct  route   *prt2;
        int             len;

        if (prt->rt_ifnum != NI_LOCAL)
                return;                 /* not ours             */
        if (!isbrc(pip->ip_dst))
                return;                 /* not broadcast        */

        prt2 = rtget(pip->ip_dst, RTF_LOCAL);
        if (prt2 == NULL)
                return;
        if (prt2->rt_ifnum == ifnum) {  /* not directed         */
                rtfree(prt2);
                return;
        }

        /* directed broadcast; make a copy */

        /* len = ether header + IP packet */

        len = EP_HLEN + pip->ip_len;
        if (len > EP_MAXLEN)
                pep2 = (struct cp *)getbuf(Net.lrgpool);
        else
                pep2 = (struct ep *)getbuf(Net.netpool);
        if (pep2 == (struct ep *)SYSERR) {
                rtfree(prt2);
                return;
        }
        memcpy(pep2, pep, len);
        /* send a copy to the net */
```

```
        ipputp(prt2->rt_ifnum, pip->ip_dst, pep2);
        rtfree(prt2);

        return;         /* continue; "pep" goes locally in IP     */
}
```

ipproc为所有的数据报调用ipdbc，其中绝大多数并没有指定要定向广播。ipdbc一开始就检查数据报来源，因为如果该数据报是由本地机器生成的，则不需要复制。然后ipdbc调用isbrc，将数据报的目的地址与这个网关直接相连接的所有网络的定向广播地址相比较，如果均不相同，则说明这是一个非广播数据报，而非广播数据报也不需要复制。在不需要复制的情况下，ipdbc不采取任何行动，直接返回。ipproc将像平常那样选择一个路由并继续向前发送该数据报。

如果数据报的目的地址是某个直接与网关相连的网络上的定向广播地址，它必须被复制。其中一份副本送往本地机器软件，另一份以正常方式继续转发。为了复制数据报，ipdbc根据数据报的大小从标准网络缓冲池或大缓冲池中分配一个缓冲区。如果缓冲区分配成功，ipdbc将数据报复制到新缓冲区中，并把新缓冲区放入输出端口，此端口与发送数据报时经过的网络接口相关联。当ipdbc返回后，ipproc将副本原件通过伪网络接口传递给本地机器。

5.4.6 识别一个广播地址

IP协议标准规定了三种类型的广播地址：本地网络广播地址（全“1”）、定向网络广播地址（主机地址段为全“1”的A、B、C三级IP地址）以及一个子网广播地址（主机地址段为全“1”的子网IP地址）。遗憾的是，当Berkeley将TCP/IP合并入BSD UNIX时，他们决定使用非标准广播地址，有时称为Berkeley广播，这些广播的格式在应该是全“1”的地方使用了全“0”表示。

尽管Berkeley格式的广播地址肯定是非标准的，但许多从Berkeley程序代码中衍生出来的商用体系已经采纳了这种格式。为了与广泛使用的Berkeley规范兼容，我们的范例代码接收使用全“0”或全“1”两种格式的广播。此程序代码段包含在过程isbrc中。

```
/* isbrc.c - isbrc */

#include <conf.h>
#include <kernel.h>
#include <sleep.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  isbrc  -  Is "dest" a broadcast address?
 *------------------------------------------------------------------------
 */
Bool isbrc(dest)
IPaddr dest;
{
        int  inum;

        /* all 0's and all 1's are broadcast */

        if (dest == ip_anyaddr || dest == ip_maskall)
```

```
            return TRUE;

        /* check real broadcast address and BSD-style for net & subnet */

        for (inum=0; inum < Net.nif; ++inum)
             if (dest == nif[inum].ni_brc ||
                 dest == nif[inum].ni_nbrc ||
                 dest == nif[inum].ni_subnet ||
                 dest == nif[inum].ni_net)
                     return TRUE;

        return FALSE;
}
```

5.5 IP首部中的字节顺序

为了使网际协议独立于运行它的机器，协议标准为首部中所有整型值规定了网络字节顺序：

> 在发送一个数据报之前，主机必须将所有整型值从本机字节顺序转换成标准网络字节顺序。而当接收到一个数据报之后，主机必须将这些整型值从标准网络字节顺序转换成本机字节顺序。

过程iph2net和ipnet2h执行这一转换任务：ipnet2h在ipproc中被调用，而iph2net在ipfsend、ipproc和ipputp中被调用。实用程序使用函数net2hs（从网络到主机短整型）和hs2net（从主机短整型到网络）为每个字段进行转换。这里的术语出自C程序设计语言，“短整型”通常是指16位整数，而“长整型”指32位整数。

为了优化处理时间，在我们的程序代码中，所有IP地址以网络字节顺序保存，并且不转换协议首部中的地址字段。这样，这段程序代码只需转换不含IP地址的整型字段。

```
/* iph2net.c - iph2net */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  iph2net - convert an IP packet header from host to net byte order
 *------------------------------------------------------------------------
 */
struct ip *iph2net(pip)
struct ip   *pip;
{
 /* NOTE: does not include IP options */

 pip->ip_len = hs2net(pip->ip_len);
 pip->ip_id = hs2net(pip->ip_id);
```

```
 pip->ip_fragoff = hs2net(pip->ip_fragoff);
 return pip;
}

/* ipnet2h.c - ipnet2h */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  ipnet2h - convert an IP packet header from net to host byte order
 *------------------------------------------------------------------------
 */
struct ip *
ipnet2h(struct ip *pip)
{
        /* NOTE: does not include IP options */

        pip->ip_len = net2hs(pip->ip_len);
        pip->ip_id = net2hs(pip->ip_id);
        pip->ip_fragoff = net2hs(pip->ip_fragoff);
        return pip;
}
```

5.6　向 IP 发送数据报

5.6.1　发送本地生成的数据报

给出一个本地生成的数据报和一个 IP 目的地址，过程 ipsend 填写 IP 首部并将数据报放入本机接口的队列中，IP 进程将从这个队列中取出并发送数据报。

```
/* ipsend.c - ipsend */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <proc.h>

static ipackid = 1;

/*------------------------------------------------------------------------
 *  ipsend  -  send an IP datagram to the specified address
 *------------------------------------------------------------------------
 */
int
```

```
ipsend(IPaddr faddr, struct ep *pep, unsigned datalen,
        u_char proto, u_char ptos, u_char ttl)
{
        struct     ip *pip = (struct ip *) pep->ep_data;

        pep->ep_type = EPT_IP;
        pep->ep_order |= EPO_IP|EPO_NET;
        pip->ip_verlen = (IP_VERSION<<4) | IP_MINHLEN;
        pip->ip_tos = ptos;
        pip->ip_len = datalen+IP_HLEN(pip);
        pip->ip_id = ipackid++;
        pip->ip_fragoff = 0;
        pip->ip_ttl = ttl;
        pip->ip_proto = proto;
        pip->ip_dst = faddr;

        /*
         * special case for ICMP, so source matches destination
         * on multi-homed hosts.
         */
        if (pip->ip_proto != IPT_ICMP)
                pip->ip_src = ip_anyaddr;

        if (enq(nif[NI_LOCAL].ni_ipinq, pep, 0) < 0) {
             freebuf(pep);
             IpOutDiscards++;
        }
        send(ippid, NI_LOCAL);
        IpOutRequests++;
        return OK;
}
/* special IP addresses */

IPaddr  ip_anyaddr = 0;
#if     BYTE_ORDER == LITTLE_ENDIAN
IPaddr  ip_loopback = 0x0I00007F;       /* 127.0.0.1 */
#else   /* BYPE_ORDER */
IPaddr  ip_loopback = 0x7F000001;       /* 127.0.0.1 */
#endif  /* BYPE_ORDER */
```

主调进程利用参数指定IP首部中的某些值。参数proto中的值用于指定协议类型，ptos中的值用于指定优先级和服务类型，而参数ttl中的值用于寿命字段。

ipsend填写首部中的每个字段，包括指定的目的地址。为了保证每个外发数据报中标识字段都有惟一的值，ipsend将全局变量ipackid的值赋予该标识字段，然后递增变量ipackid。ipsend完成填写首部的任务之后，调用enq把数据报置入本地主机（伪网络）接口的队列中。

注意，虽然网络接口的ni_ipinq队列在正常情况下包含的是传入数据报（也就是来自其他网点

的数据报），但从应用软件的角度看，伪网络接口的队列中包含的是“外发”的数据报。最后，在IP进程因等待数据报的到达而处于阻塞状态的情况下，ipsend调用send向IP进程发送一个报文。

5.6.2　发送传入数据报

当一个经由网络发送来的数据报到达时，网络接口层的设备驱动程序必须将其放置在适当的IP队列中。它调用ip_in完成这项工作。

```
/* ip_in.c - ip_in */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  ip_in - IP input function
 *------------------------------------------------------------------------
 */
int ip_in(pni, pep)
struct netif  *pni;
struct        ep *pep;
{
        struct    ip    *pip = (struct ip *)pep->ep_data;

        IpInReceives++;
        if (enq(pni->ni_ipinq, pep, pip->ip_tos & IP_PREC) < 0) {
                IpInDiscards++;
                freebuf(pep);
        }
        send(ippid, (pni-&nif[0]));
        return OK;
}
```

给出一个指向分组缓冲区的指针，ip_in调用enq将分组置入接口的队列中。如果队列已满，ip_in递增变量IpInDiscards的值，以记录下这个队列的溢出差错，并丢弃分组。最后，在IP进程被阻塞并等待接收数据报的情况下，ip_in向IP进程发送一个报文。

5.7　表格的维护

IP软件需要一个时钟机制，用于维护网络数据结构，包括IP选路表和分片重装表。在我们的范例中，使用一个单一的定时器进程来完成这种周期性任务。事实上，定时器进程并不仅用于IP任务中，它还触发ARP高速缓存的超时，并且可被用于其他任何没有严格时延要求的长时间的周期性任务中。在过程slowtimer中的程序代码段，体现了如何轻易地将新任务加入列表中。

```
/* slowtimer.c - slowtimer */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <sleep.h>
#include <date.h>
#include <network.h>

#define STGRAN   1          /* Timer granularity (delay) in seconds   */

void arptimer(unsigned), ipftimer(unsigned), rttimer(unsigned);
#ifdef OSPF
void ospftimer(unsigned);
#endif /*OSPF*/

/*------------------------------------------------------------------------
 * slowtimer - handle long-term periodic maintenance of network tables
 *------------------------------------------------------------------------
 */
PROCESS
slowtimer(void)
{
        unsigned long   lasttime, now;  /* prev and current times (secs)*/
        int             delay;          /* actual delay in seconds      */

        signal(Net.sema);

        gettime(&lasttime);
        while (1) {
                sleep(STGRAN);
                gettime(&now);
                delay = now - lasttime;
                if (delay <= 0 || delay > 4*STGRAN)
                        delay = STGRAN; /* likely clock reset */
                lasttime = now;
                arptimer(delay);
                ipftimer(delay);
                rttimer(delay);
#ifdef OSPF
                ospftimer(delay);
#endif /* OSPF */
        }
}
```

正如程序代码所示，slowtimer由一个无限循环构成，它重复不断地唤醒一组维护过程。一个特定的维护过程可能花费相当长的时间来完成它的任务，而且一次唤醒与下一次唤醒之间的执行时间

也可能不一样长。因此slowtimer计算两次执行之间的实际时延，并将其作为参数传递给维护过程。

5.8　小结

为简化程序代码，IP作为惟一的、独立的进程执行，并通过一个伪网络接口与本机的高层协议软件交互作用。在没有数据报时，IP进程被阻塞。一旦从任何地方发来一个或多个数据报后，IP进程被唤醒，并处理这些数据报，直至为它们选择路由为止。为了使处理过程公平合理，并避免出现"饥饿状态"，我们在范例中对所有输入数据源，包括对应于本机的伪网络接口，采用了循环法策略。这样不论是本地产生的通信量，还是来自网络连接的传入通信量都没有优先权。

定向广播是指将数据报发送到特定网络的所有主机和网关上。协议标准允许设计者自己决定是否继续向前传送由其他网络产生的定向广播。如果网关选择了允许定向广播，它正常地为这些定向广播选择路由。如果目的地址给出了一个直接与网关相连的网络，IP必须保证本机的上层协议软件接收到一份数据报的副本。为了提高它的实用性，在实现范例中，既可使用TCP/IP标准（全"1"），也可使用BSD UNIX格式（全"0"）的广播地址。它复制广播数据报，并安排网络接口发送副本，而原数据报经过选路，被发往本机的协议软件。

IP校验和由16位二进制反码组成，其值可用32位二进制补码及进位传递来计算。

深入研究

在Postel［RFC 791］中可找到IP的早期标准。Mogul和Postel［RFC 950］给出了子网编址标准。在Braden［RFC 1122］给出了这二者的更新内容。Baker［RFC 1812］总结了对网关的一些要求。Mallory［RFC 1141］讨论IP校验和的增量修改。Braden、Borman和Partridge［RFC 1071］提供了早期的讨论。Padlipsky［RFC 875］及Hinden和Sheltzer［RFC 823］描述了有关网关的早期思想。

习题

5.1　"0"的反码有两个值。cksum将返回哪一个？

5.2　用汇编语言重写cksum。它的速度与C语言编写的版本相比较又如何？

5.3　考虑为所有发往IP的数据报使用一个单一的输入队列的实现方案。这种方案的主要缺点是什么？

5.4　仔细研究过程ipproc中的程序代码。辨别出其中所有将发往/来自本机的数据报作为特殊情况对待的地方。

5.5　如果在上层协议将输出数据报置入队列时，已完成了计算任务，上题中提到的特殊情况会被消除吗？

5.6　指出为什么在没有数据报等待被处理的情况下，ipproc也有可能做最后一次循环。提示：考虑IP进程和一个放置数据并向IP发送报文的驱动程序之间的定时处理。

5.7　考虑V UNIX用于建立设备驱动软件和协议软件的STREAMS机制。可以用它来实现IP吗？如何实现？

5.8　在单一的独立进程中实现IP的主要优点是什么？主要缺点又是什么？

5.9 过程ipsend为数据报首部中的寿命字段提供了一个固定值。这样做合理吗？

5.10 仔细观察数据报标识字段的初始值。讨论在设备启动、发送一个数据报、出现故障、快速重启动以及向同一目的站发送了一个不同的数据报时，分片是否会引起严重错误。

5.11 尝试对代码进行修改来消除上面一个练习中出现的错误。

5.12 当过程ip_in发现一个接口队列已满时，会丢弃传入数据报。阅读RFC 792和1122，了解当这种情况发生时，IP应不应当产生一个差错报文。

5.13 设计一个程序，只对slowtimer做很小的改动就可在调用维护过程中生成更精确的值。这两种实现方法的优点和缺点是什么？

第6章　IP：选路表和选路算法

6.1　引言

前一章描述了网际协议（IP）软件的总体结构，并提供了其关键过程ipproc的程序代码。本章将提供选择路由的细节，继续讨论IP。本章详细描述IP选路表的结构，以及实现选路表的数据结构，还讨论了路由算法，并叙述了IP在选择路由时如何使用子网掩码。最后，它叙述了IP是如何区分网络路由、子网路由以及主机路由的。

6.2　路由维护和查找

选路软件在理论上可划分为两组。其中一组由那些用来为数据报选择正确路由的过程组成，另一组由增加、改变或删除路由的过程组成。因为网关必须为自己所处理的每个数据报判定路由，所以路由查找程序代码决定了网关的整体性能。因此，通常将查找程序代码优化，以期达到最高速度。

路由的插入、改变或删除的速率通常比为数据报选择路由的速率要低得多。计算新路由的程序需要与其他设备通信以建立其可达性。它们可以在重定向之前花任意长的时间。因此，路由更新过程不需要像查找操作那样被优化。其基本观点是：

应当选择能使查找路由花费最小的IP数据结构和算法，维护路由的花费则并不那么重要。

虽然在早期的TCP/IP软件中，选路表的查找经常采用线性查找，但现在大多数系统利用一张散列表，使之能够快速查找任意大小的选路表。

我们的软件采用了桶散列（bucket hashing）结构。它将选路表的入口划分成许多“散列表元”，并利用一个散列函数迅速找到正确的散列表元。

6.3　选路表结构

图6.1所示为选路表使用的数据结构。

保存路由的主要数据结构是数组。数组的每个元素对应一个散列表元，并包含一个指针，指向被装入这个散列表元中的通往目的站的路由记录链表。表中每个记录包含一个IP目的地址、子网掩码、下一跳地址、用于向下一跳地址发送数据的网络接口，以及其他在路由管理中使用的信息。由于IP预先并不知道子网掩码，因此它仅使用IP目的地址的网络部分计算散列函数。然而当IP在一个链表中查找表项时，使用的是完整的目的地址来进行比较。在后面的章节中将详述其细节。

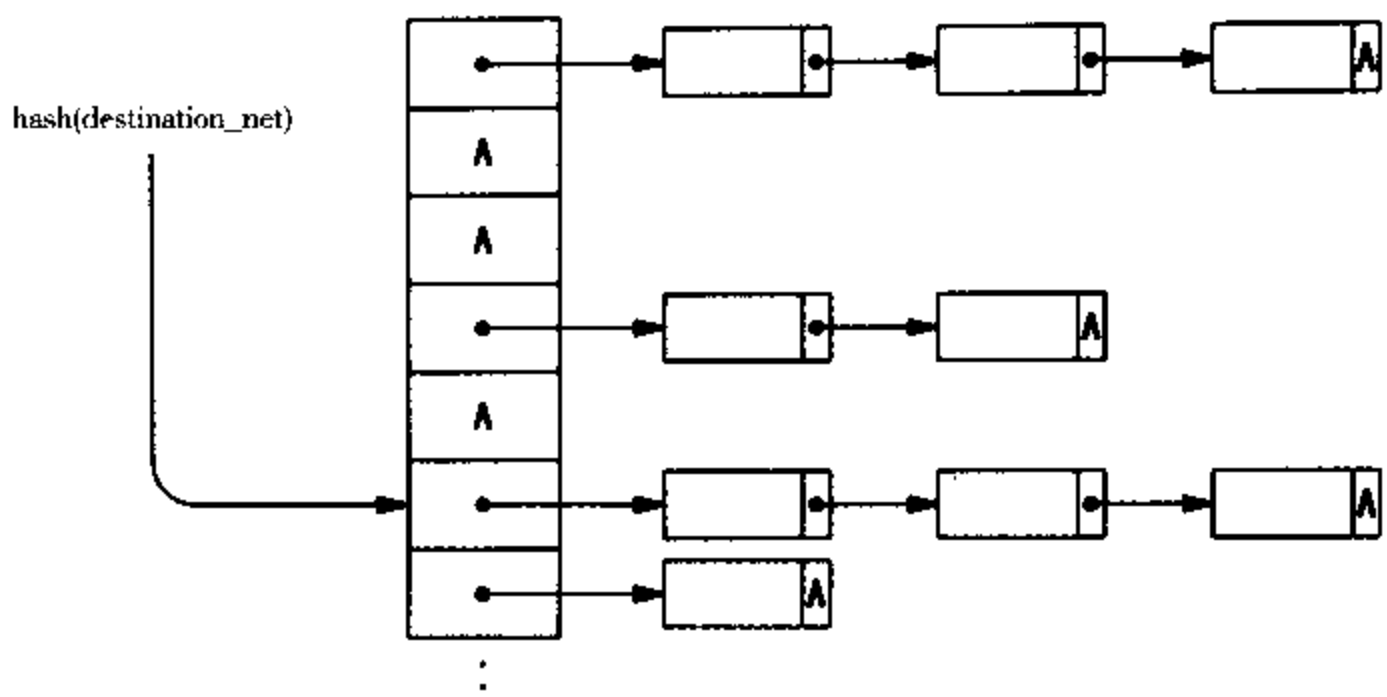


图 6.1 利用一个数组实现散列选路表。数组中的每个入口指向一个记录链表，每个记录链表包含了一个目的地址和到达此目的站的路由

6.4 选路表数据结构

文件 route.h 中说明了选路表的数据结构。

```
/* route.h - RTFREE */

/* Routing Table Entries: */
struct route {
        IPaddr  rt_net;             /* network address for this route    */
        IPaddr  rt_mask;            /* mask for this route               */
        IPaddr  rt_gw;              /* next IP hop                       */
        u_short rt_metric;          /* distance metric                   */
        u_short rt_ifnum;           /* interface number                  */
        short   rt_key;             /* sort key                          */
        short   rt_ttl;             /* time to live (seconds)            */
        struct  route *rt_next;     /* next entry for this hash value    */
/* stats */
        int  rt_refcnt;             /* current reference count           */
        int  rt_usecnt;             /* total use count so far            */
};

/* Routing Table Global Data: */
struct rtinfo {
        struct  route   *ri_default;
        int             ri_bpool;
        Bool            ri_valid;
        int             ri_mutex;
};
```

```
#define RT_DEFAULT ip_anyaddr    /* the default net                    */
#define RT_LOOPBACK ip_loopback  /* the loopback net                   */
#define RT_TSIZE      512        /* these are pointers; it's cheap     */
#define RT_INF        999        /* no timeout for this route          */

#define RTM_INF       16         /* an infinite metric                 */

/* rtget()'s second argument... */

#define RTF_REMOTE    0          /* traffic is from a remote host      */
#define RTF_LOCAL     1          /* traffic is locally generated       */

#define RT_BPSIZE     100        /* max number of routes               */

/* RTFREE - remove a route reference (assumes ri_mutex HELD)           */

#define RTFREE(prt)                                   \
        if (--prt->rt_refcnt <= 0) {                  \
            freebuf(prt);                             \
        }

extern struct rtinfo  Route;
extern struct route   *rttable[];

int rtadd(IPaddr, IPaddr, IPaddr, unsigned, unsigned, unsigned);
int rtdel(IPaddr, IPaddr), rthash(IPaddr), rtfree(struct route *);
struct route *rtget(IPaddr, Bool);
void rtinit(void);
void ipredirect(struct ep *, unsigned, struct route *);
void ipdbc(unsigned, struct ep *, struct route *);
```

route结构定义了链表中一个结点的内容，并包含一个可能的目的站的选路信息。字段rt_net指出其目的地址（可能是一个网络、子网，也可能是完整的主机地址）。字段rt_mask指定了与此目的站一起使用的32位掩码。掩码项可能覆盖地址中的网络部分、网络加子网部分，或是完整的32位（即它们有可能包括主机部分）。

字段rt_gw指出该路由中下一跳网关的IP地址，而字段rt_metric给出网关距离，以跳（hop）为单位。字段rt_ifnum给出这条路由使用的网络接口的内部编号（即网络用来到达下一跳网关的接口）。

剩余的几个字段供IP软件使用。字段rt_key包含的是一个排序关键字，在向链表中插入结点时使用。字段rt_refcnt包含一个进程的引用计数器，用于保存指向路由的指针，而字段rt_usecnt记录该路由被使用的次数。最后，字段rt_next包含了一个指针，指向链表的下一个结点（链表的最后结点中此值为NULL）。

除route结构外，文件route.h中还定义了选路表rttable。如图6.1所示，rttable是指向route结构的指针数组。

除选路表之外，IP还需要其他几个数据项，它们保存在全局结构rtinfo中。例如，系统提供惟一的“默认路由”，任何不在选路表中的目的站都可使用。字段ri_default指向一个route结构，在这

个route结构中包含的是默认路由的下一跳地址。在字段ri_valid中是一个布尔变量，如果此路由的数据结构已被初始化，则它的值为TRUE。

6.5 路由的生成源及保持时间

选路表中的信息有几个来源。通常，当系统启动时，初始化程序从备份存储器中获得一组初始路由，并将它们安装到选路表中。在执行过程中，传入的报文会导致ICMP或选路协议软件改变现有的路由或建立新路由。最后，网络管理员也可以增加路由或重定向。

选路表项的易失性与它的来源有关。例如，初始路由被认为是简单估计的，只要有其他地方发来的路由信息，它就应该被替换。然而，网络管理员必须能够推翻任何路由并建立不可变的永久性路由，以使他们能够排除网络中的路由问题，而无需路由协议的干涉。

为了适应路由的灵活性，每个选路表项中的rt_ttl（寿命）字段指出此表项还能保持多长时间有效，以秒为单位计算。当rt_ttl值达到0时，路由失效并被丢弃。由选路协议建立的路由，其寿命根据协议规则计算得到，而管理员能够建立寿命无限长的路由，以保证它们不会被删除。

6.6 为数据报选择路由

6.6.1 实用过程

一些实用过程提供了选择路由时用到的功能。过程netnum利用地址类别确定在地址字段中哪些八位组包含的是网络部分，哪些八位组是主机部分，从而提取出IP地址中的网络部分。它返回给出的地址，但这时其主机地址字节为全0。

```
/* netnum.c - netnum */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  netnum  -  compute the network portion of a given IP address
 *------------------------------------------------------------------------
 */
IPaddr
netnum(IPaddr ipa)
{
        IPaddr mask = ~0;

        if (IP_CLASSA(ipa)) mask = hl2net(0xff000000);
        if (IP_CLASSB(ipa)) mask = hl2net(0xffff0000);
        if (IP_CLASSC(ipa)) mask = hl2net(0xffffff00);
        return ipa & mask;
}
```

在为数据报选择路由时，IP使用过程netmatch，将目的（主机）地址与选路表项中的地址进行比较，选路表项中包含了子网掩码和给定网络的IP地址。netmatch使用子网掩码来屏蔽目的地址中的主机位，并将结果与表项中的网络地址相比较，如果它们相符，netmatch返回TURE，否则返回FALSE。

广播是一种特殊情况，因为要采取何种动作取决于数据报的来源。来自网络接口的广播数据报必须经由伪网络接口发送给本地机器，而本地生成的广播数据报必须被发送到适当的网络接口。为了区分这两种情况，软件使用了一个特定主机路由（全“1”的掩码）为来自网络的广播数据报选择路由，并用特定网络路由（掩码仅屏蔽网络部分）为向外发送的广播选择路由。这样netmatch明确地判定一个广播数据报的来源，并使用IP源地址来决定此广播是否与给定路由相符。

```
/* netmatch.c - netmatch */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  netmatch  -  Is "dst" on "net"?
 *------------------------------------------------------------------------
 */
Bool netmatch(dst, net, mask, islocal)
IPaddr dst, net, mask;
Bool   islocal;
{
        if ((dst & mask) != (net & mask))
                return FALSE;
        /*
         * local srcs should only match broadcast addresses (host routes)
         */
        if (islocal)
                if (isbrc(dst) || IP_CLASSD(dst))
                        return mask != ip_maskall;
        return TRUE;
}
```

要为一个数据报选择路由，IP必须先查看它是否认识该数据报目的地址的有效子网掩码。要做到这一点，IP调用过程netmask。

```
/* netmask.c - netmask */

#include <conf.h>
#include <kernel.h>
#include <network.h>

IPaddr          netnum(IPaddr);
```

```
/*------------------------------------------------------------------------
 *  netmask  -  set the default mask for the given net
 *------------------------------------------------------------------------
 */
IPaddr
netmask(IPaddr net)
{
        IPaddr netpart;
        int    i;

        if (net == 0)
                return net;
        /* check for net match (for subnets) */

        netpart = netnum(net);
        for (i=0; i<Net.nif; ++)
                if (nif[i].ni_svalid && nif[i].ni_ivalid &&
                    nif[i].ni_net == netpart)
                        return nif[i].ni_mask;
        if (IP_CLASSA(net)) renturn hl2net(0xff000000);
        if (IP_CLASSB(net)) renturn hl2net(0xffff0000);
        if (IP_CLASSC(net)) renturn hl2net(0xffffff00);
        return ~0;
}
```

netmask 把IP目的地址当作参数，然后检查几种情况。根据约定，如果目的地址为全“0”，则意味着使用默认路由，因此netmask返回全“0”的子网掩码。对其他类型的目地站，netmask调用netnum从目的地址中提取出网络部分，然后检查每个与本机直接连接的网络，如果有任何与本地直接连接的网络地址与目的地址的网络部分相符，netmask从和此网络相关的网络接口结构中提取出子网掩码，然后将其返回调用者。最后，如果IP找不到有关这个目的地址的子网掩码的信息，它就根据地址的类别是A、B还是C来设置屏蔽地址网络部分的子网掩码。

选路函数调用实用过程rthash计算目的地网络地址的散列值。

```
/* rthash.c - rthash */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  rthash  -  compute the hash for "net"
 *------------------------------------------------------------------------
 */
int rthash(net)
IPaddr net;
{
```

```
        int         bc = IP_ALEN;   /* # bytes to count  */
        unsigned int    hv = 0;         /* hash value        */

        if (IP_CLASSA(net)) bc = I;
        else if (IP_CLASSB(net)) bc = 2;
        else if (IP_CLASSC(net)) bc = 3;
        else if (IP_CLASSD(net))
                return ((net>>24) & 0xf0) % RT_TSIZE;
        while (bc--)
                hv += (char *) & net[bc] & 0xff;
        return hv % RT_TSIZE;
}
```

散列函数是一个简单有效的计算函数。对于A、B或C类的地址，rthash将网络地址的每个八位组相加，所得的和除以散列表长度，最后返回余数。对于D类的地址，rthash将网络地址的前4位比特数乘以16，所得的值除以散列表长度，最后返回余数。

6.6.2 获得一个路由

给定一个目的地址，过程rtget在选路表中查找，并返回一个指针，指向找到的路由的表项。

```
/* rtget.c - rtget */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  rtget  -  get the route for a given IP destination
 *------------------------------------------------------------------------
 */
struct route *rtget(dest, local)
IPaddr dest;
Bool   local;
{
        struct      route*prt;
        int         hv;

        if (!Route.ri_valid)
                rtinit();
        wait(Route.ri_mutex);
        hv = rthash(dest);
        for (prt=rttable[hv]; prt; prt=prt->rt_next) {
                if (prt->rt_ttl <= 0)
                        continue;       /* route has expired */
                if (netmatch(dest, prt->rt_net, prt->rt_mask, local))
                        if (prt->rt_metric < RTM_INF)
```

```
                        break;
        }
        if (prt == 0)
             prt = Route.ri_default;    /* may be NULL too... */
        if (prt != 0 && prt->rt_metric >= RTM_INF)
             prt = 0;
        if (prt) {
             prt->rt_refcnt++;
             prt->rt_usecnt++;
        }
        signal(Route.ri_mutex);
        return prt;
}
```

全局变量Route.ri_valid指出选路表是否已被初始化，如果没有，rtget调用rtinit。一旦选路表及其相关的数据结构已被初始化，rtget等待互斥信号量，以保证任何时间只有一个过程有权访问选路表。然后它计算目的地址的散列值，以此作为表内查找的索引，并在选路表表项指向的链表内查找。

对每一个路由表项，rtget调用netmatch，检查其入口参数给出的目的地址是否能与表项中的地址匹配。如果在查找中没有发现精确的匹配，rtget使用Route.ri_default中默认的路由。

当然，也可能出现既没有精确匹配，也没有默认路由的情况，因此，当完成路由查找工作后，rtget还必须继续检查它找到的是否是一个有效指针。如果是有效指针，rtget在返回调用者之前，要给路由表项中的引用计数值和使用计数值字段加1。维护软件利用引用计数字段来决定如果删除与该路由相关的存储区是否安全。只要调用rtget的过程还需要这个路由表项，该表项的引用计数值就不为0。使用计数字段提供了管理网络的一种手段，用以掌握每个表项在数据报选择路由时使用的频繁程度。

6.6.3 数据结构初始化

过程rtinit初始化选路表和默认路由，生成互斥信号量，为路由链表中的结点分配存储区并将存储区链接成一个空闲列表。其实现方案很简单。

```
/* rtinit.c - rtinit */

#include <conf.h>
#include <kernel.h>
#include <sleep.h>
#include <network.h>

struct rtinfo     Route;
struct route *rttable[RT_TSIZE];

/*------------------------------------------------------------------------
 *  rtinit  -  initialize the routing table
 *------------------------------------------------------------------------
 */
```

```
void
rtinit(void)
{
        int i;

        for (i=0; i<RT_TSIZE; ++i)
                rttable[i] = 0;
        Route.ri_bpool = mkpool(sizeof(struct route), RT_BPSIZE);
        Route.ri_valid = TRUE;
        Route.ri_mutex = screate(1);
        Route.ri_default = NULL;
}
```

6.7　选路表的定期维护

系统定期发起清理选路表的行动，它递减选路表中所有表项的寿命字段值，如果某个寿命字段超时，则丢弃对应的路由。过程 rttimer 用来实现这项定期性的表项更新任务。

```
/* rttimer.c - rttimer */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

extern Bool  dorip;          /* TRUE if we're running RIP output  */
extern int   rippid;         /* RIP output pid, if running        */

/*------------------------------------------------------------------------
 * rttimer - update ttls and delete expired routes
 *------------------------------------------------------------------------
 */
void
rttimer(unsigned int delta)
{
        struct route    *prt, *prev;
        Bool            ripnotify;
        int             i;

        if (!Route.ri_valid)
                return;
        wait(Route.ri_mutex);

        ripnotify = FALSE;
        for (i=0; i<RT_TSIZE; ++i) {
                if (rttable[i] == 0)
                        continue;
```

```
                for (prev = NULL, prt = rttable[i]; prt != NULL;) {
                    if (prt->rt_ttl != RT_INF)
                            prt->rt_ttl -= delta;
                    if (prt->rt_ttl <= 0) {
#ifdef RIP
                            if (dorip && prt->rt_metric < RTM_INF) {
                                    prt->rt_metric = RTM_INF;
                                    prt >rt_ttl = RIPZTIME;
                                    ripnotify = TRUE;
                                    continue;
                            }
#endif /* RIP */
                            if (prev) {
                                    prev->rt_next = prt->rt_next;
                                    RTFREE(prt);
                                    prt = prev->rt_next;
                            } else {
                                    rttable[i] = prt->rt_next;
                                    RTFREE(prt);
                                    prt = rttable[i];
                            }
                            continue;
                    }
                    prev = prt;
                    prt = prt->rt_next;
            }
        }
        prt = Route.ri_default;
        if (prt && (prt->rt_ttl<RT_INF) && (prt->rt_ttl -= delta) <= 0) {
#ifdef RIP
                if (dorip && prt->rt_metric < RTM_INF) {
                        prt->rt_metric = RTM_INF;
                        prt->rt_ttl = RIPZTIME;
                } else
#endif /* RIP */
                {
                        RTFREE(Route.ri_default);
                        Route.ri_default = 0;
                }
        }
        signal(Route.ri_mutex);
#ifdef RIP
        if (dorip && ripnotify)
                send(rippid, 0);  /* send anything but TIMEOUT    */
#endif /* RIP */
```

```
        return;
}
```

定时进程（执行slowtimer[1]的进程）大约每秒调用一次rttimer，利用入口参数delta，提供上次调用与这次调用之间的间隔时间。在等待互斥信号量之后，rttimer循环遍历选路表。对每个表项来说，它遍历并查看各个路由链表。对正常的路由，rttimer递减寿命字段计数值，如果计时值为0，从链表中删除该结点。然而，如果网关运行RIP，则rttimer将超时路由标记为费用无穷大，这样它就无法将其作为数据报的有效路由，同时又使这些超时路由在表中短暂地保留一段时间[2]。最后，rttimer递减默认路由中的寿命计数值。

6.7.1　增加路由

网络管理软件和选路信息协议都可调用那些用来增加、删除或重定向的函数。例如，过程rtadd向表中增加一个新路由。

```
/* rtadd.c - rtadd */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

struct route *rtnew(IPaddr, IPaddr, IPaddr, unsigned,unsigned,unsigned);
void rtinit(void);
int rthash(IPaddr);

/*------------------------------------------------------------------------
 *  rtadd  -  add a route to the routing table
 *------------------------------------------------------------------------
 */
int rtadd(net, mask, gw, metric, intf, ttl)
IPaddr net, mask, gw;
unsigned    metric, intf, ttl;
{
        struct route    *prt, *srt, *prev;
        Bool            isdup;
        int             hv, i;

        if (!Route.ri_valid)
             rtinit();

        prt = rtnew(net, mask, gw, metric, intf, ttl);
        if (prt == (struct route *)SYSERR)
                return SYSERR;
```

① 参见5.7节slowtimer.c程序清单。
② 第18章叙述RIP怎样使用选路表。

```
        /* compute the queue sort key for this route */
        prt->rt_key = 0;
        for (i=0; i<32; ++i)
                prt->rt_key += (mask >> i) & 1;
        wait(Route.ri_mutex);

        /* special case for default routes */
        if (net == RT_DEFAULT) {
                if (Route.ri_default)
                        RTFREE(Route.ri_default);
                Route.ri_default = prt;
                signal(Route.ri_mutex);
                return OK;
        }
        prev = NULL;
        hv = rthash(net);
        isdup = FALSE;
        for (srt=rttable[hv]; srt; srt = srt->rt_next) {
                if (prt->rt_key > srt->rt_key)
                        break;
                if (srt->rt_net == prt->rt_net &&
                    srt->rt_mask == prt->rt_mask) {
                        isdup = TRUE;
                        break;
                }
                prev = srt;
        }
        if (isdup) {
                struct    route*tmprt;

                if (srt->rt_gw == prt->rt_gw) {
                        /* just update the existing route */
#ifdef RIP
                        if (dorip) {
                                srt->rt_ttl = ttl;
                                if (srt->rt_metric != metric) {
                                        if (metric == RTM_INF)
                                                srt->rt_ttl = RIPZTIME;
                                        send(rippid, 0);
                                }
                        }
#endif /* RIP */
                        srt->rt_metric = metric;
                        RTFREE(prt);
                        signal(Route.ri_mutex);
```

```
                return OK;
        }
            /* else, someone else has a route there... */
            if (srt->rt_metric <= prt->rt_metric) {
                    /* no better off to change; drop the new one */

                    RTFREE(prt);
                    signal(Route.ri_mutex);
                    return OK;
            }
#ifdef RIP
            else if (dorip)
                send(rippid, 0);
#endif /* RIP */
            tmprt = srt;
            srt = srt->rt_next;
            RTFREE(tmprt);
            }
#ifdef RIP
        else if (dorip)
                send(rippid, 0);
#endif /* RIP */
        prt->rt_next = srt;
        if (prev)
                prev->rt_next = prt;
        else
                rttable[hv] = prt;
        signal(Route.ri_mutex);
        return OK;
}
```

rtadd 调用过程 rtnew 来分配一个新结点，并初始化它的各个字段。然后 rtadd 将默认路由当作特殊情况对待。对于非默认的路由，rtadd 利用 rthash 计算新路由在路由表中的索引值，并遍历该表项中的路由链表。一旦它在链表中找到新路由应该被插入的地方，就查看链表是否包含一个具有相同目的站的已有路由。如果有，rtadd 比较旧路由和新路由的度量，是否新路由更好。如果不是，就丢弃新路由。最后，rtadd 或者在链表中插入一个新结点，或者将信息复制到具有同样地址的已有结点中。

过程 rtnew 分配并初始化一个新路由表项。它调用 getbuf 来为新结点分配存储区，然后填写首部信息。

```
/* rtnew.c - rtnew */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```

```
/*------------------------------------------------------------------------
 *  rtnew  -  create a route structure
 *------------------------------------------------------------------------
 */
struct route *rtnew(net, mask, gw, metric, ifnum, ttl)
IPaddr net, mask, gw;
unsigned    metric, ifnum, ttl;
{
        struct     route *prt;

        prt = (struct route *)getbuf(Route.ri_bpool);
        if (prt == (struct route *)SYSERR) {
                IpRoutingDiscards++;
                return (struct route *)SYSERR;
        }

        prt->rt_net = net;
        prt->rt_mask = mask;
        prt->rt_gw = gw;
        prt->rt_metric = metric;
        prt->rt_ifnum = ifnum;
        prt->rt_ttl = ttl;
        prt->rt_refcnt = 1;  /* our caller */
        prt->rt_usecnt = 0;
        prt->rt_next = NULL;
        return prt;
}
```

6.7.2 删除路由

过程rtdel以一个目的地址作为入口参数，并通过从选路表中移走结点来删除路由。

```
/* rtdel.c - rtdel */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  rtdel  -  delete the route with the given net, mask
 *------------------------------------------------------------------------
 */
int rtdel(net, mask)
IPaddr net, mask;
{
        struct     route      *prt, *prev;
        int                   hv, i;
```

```
        if (!Route.ri_valid)
                return SYSERR;
        wait(Route.ri_mutex);
        if (Route.ri_default &&
            net == Route.ri_default->rt_net {
                RTFREE(Route.ri_default);
                Route.ri_default = 0;
                signal(Route.ri_mutex);
                return OK;
        }
        hv = rthash(net);

        prev = NULL;
        for (prt = rttable[hv]; prt; prt = prt->rt_next) {
                if (net == prt->rt_net &&
                    mask == prt->rt_mask)
                        break;
                prev = prt;
        }
        if (prt == NULL) {
                signal(Route.ri_mutex);
                return SYSERR;
        }
        if (prev)
                prev->rt_next = prt->rt_next;
        else
                rttable[hv] = prt->rt_next;
        RTFREE(prt);
        signal(Route.ri_mutex);
        return OK;
}
```

像通常那样，程序代码查看并以特殊情况对待默认路由。如果没有找到匹配表项，rtdel计算目的地址的散列值，并在路由链表中搜索。一旦它找到正确的路由，rtdel将该结点从链表中删除，并利用宏 RTFREE 来递减引用计数值。回想一下，如果引用计数值达到0，RTFREE 就将该结点返回空闲结点表。如果引用计数值仍为正值，表示肯定还有其他一个或几个进程正在使用该结点。只有当最后一个进程将引用计数值减至0时，此结点才被释放回空闲结点表。

宏RTFREE假设正在执行中的进程已经获得了单独访问选路表的权力，因此它可在诸如rtdel之类的过程中被使用。任意一个需要递减选路表中引用计数值的过程都可以调用过程 rtfree。当 rtfree 被唤醒后，首先它等待互斥信号量，然后唤醒宏 RTFREE，最后释放信号量。

```
/* rtfree.c - rtfree */

#include <conf.h>
#include <kernel.h>
```

```
#include <network.h>

/*------------------------------------------------------------------------
 *  rtfree  -  remove one reference to a route
 *------------------------------------------------------------------------
 */
int rtfree(prt)
struct route*prt;
{
        if (!Route.ri_valid)
                return SYSERR;
        wait(Route.ri_mutex);
        RTFREE(prt);
        signal(Route.ri_mutex);
        return OK;
}
```

6.8 IP选项处理

IP支持一组选项，用来控制IP如何处理主机和网关中的数据报。为了使例程代码简洁易懂，我们决定忽略对选项的处理。虽然如此，我们还是在程序代码中包括了两个实例程序的框架，用于搜寻IP首部中的选项。网关调用过程ipdoopts，它仅仅返回调用者，使得网关在前送数据报时，并没有对选项做任何处理。

```
/* ipdoopts.c - ipdoopts */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  ipdoopts - do gateway handling of IP options
 *------------------------------------------------------------------------
 */
int ipdoopts(pni, pep)
struct netif*pni;
struct ep    *pep;
{
        return OK;      /* not implemented yet */
}
```

主机调用过程ipdstopts来处理到达主机的数据报的选项。虽然我们在过程中并没有真正实现选项的处理，但它分析了IP首部中选项长度八位组的信息，并将选项字段从IP首部中删除。

```
/* ipdstopts.c - ipdstopts */
```

```
#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  ipdstopts - do host handling of IP options
 *------------------------------------------------------------------------
 */
int ipdstopts(pni, pep)
struct netif   *pni;
struct ep      *pep;
{
        struct ip  *pip = (struct ip *)pep->ep_data;
        u_char     *popt, *popend;
        int        len;

        if (IP_HLEN(pip) == IPMHLEN)
             return OK;
        popt = pip->ip_data;
        popend = (u_char*)&pep->ep_data[IP_HLEN(pip)];

        /* NOTE: options not implemented yet */

        /* delete the options */
        len = pip->ip_len - IP_HLEN(pip);    /* data length  */
        if (len)
        memcpy(pip->ip_data, &pep->ep_data[IP_HLEN(pip)], len);
        pip->ip_len = IPMHLEN + len;
        pip->ip_verlen = (pip->ip_verlen&0xf0) | IP_MINHLEN;
        return OK;
}
```

6.9 小结

IP 选路表是一个至关重要的数据结构。当为数据报选择路由时，IP 进程利用该选路表，在表中查找指向数据报目的站的下一跳路由。由于路由查找的频繁使用，因此选路表在其组织结构上尽量考虑提高查找的效率。同时，掌握新的路由信息的高层协议软件还要在选路表中插入、删除或更改路由。

本章详述了负责选路表的查找和维护的过程。从中可了解到选路表怎样利用桶散列结构提高效率的，还可以了解到当进程正在删除路由时引用计数器怎样允许另一进程继续使用该路由。

深入研究

Postel［RFC 791］给出了网际协议的标准，Horning［RFC 894］规定了通过以太网传输 IP 数

据报的标准，同时Mogul和Postel等［RFC 950和940］讨论子网的处理。Broden［RFC 1122］有更新内容。在IP中使用的具体常量可从Reynolds和Postel［RFC 1700］中查找。

Baker［RFC 1812］对网关如何处理IP数据报做了总结。Postel［RFC 791］描述了IP选项的处理，而Su［RFC 781］对时间戳选项做了评述。Mills［RFC 981］的内容有关多路径选路，而Braun［RFC 1104］讨论了基于各种策略的选路方法。

习题

6.1 考虑系统启动时由两个进程来自动初始化选路表的方案。这两个进程之间有可能互相干扰吗？为什么？

6.2 散列表元的数目决定了桶散列方案的有效性，因为它决定了链表的平均长度。如果希望平均每个链表不多于3个表项，那么要保存1000条路由需要多大的内存？

6.3 如果过程rtdel调用rtfree而不是使用宏RTFREE，结果怎样？

6.4 ICMP重定向报文仅允许网关将目的站指定为主机重定向或网络重定向。本章中的程序代码是如何帮助网关判断一个地址是否是子网地址？

6.5 假设在IP的下一个版本中，所有地址都是自识别的（例如，每个地址都有一个正确的子网掩码）。你将如何重新设计选路表的数据结构，以使它们更有效？

6.6 阅读关于CIDR选路的内容，重新设计本章的选路表的数据结构以适应CIDR寻址（例如，各网络前缀的范围应该同样对待）。

6.7 考虑为一个广播数据报选择路由（参看netmatch）。为什么程序代码要谨慎地区别本地生成的广播数据报和传入的广播数据报？

6.8 为广播数据报选择路由时，可能会引起一些特殊情况，因此我们可以为每个路由表项增加一个字段，规定此表项应被输入通信量还是输出通信量使用，还是两者都可以？增加一个这样的字段会给网络管理带来什么困难？

6.9 我们曾讲过，将本地主机接口看作伪网络接口来实现可以消除本地主机接口的特殊性。在本章的程序中，有多少次是明确地将主机作为特别情况对待的？

6.10 设计一个具有备份路由的选路表（即为每个目的站保持几条路由）有必要吗？

6.11 在本章中，为IP选路表增加一个可根据其服务类型选择路由的功能。这样就使选路不仅与目的地址有关，而且与数据报的服务类型有关。

6.12 在本章中，为IP选路表增加安全性选路功能，这样就使选路不仅与目的地址有关，而且与数据报的源地址及协议类型有关。

第7章 IP：分片与重组

7.1 引言

本章详述将外发数据报进行分片以及将传入数据报进行重组的软件。由于最终目的站需要把分片的数据报重新组装起来，因此每个应用TCP/IP的计算机必须具有负责重组的程序代码，否则，它就不能与互联网上所有的计算机通信。

协议标准规定，任何实现IP的程序都必须能够将数据报分片及重组。事实上，任何网关若连接两个以上且具有不同MTU大小的网络，则经常需要将数据报分片传输。由于设计良好的应用软件会主动生成足够小的、能够直接通过网络传输的数据报，因此主机并不需要经常执行分片任务。

7.2 数据报的分片

数据报的分片发生在IP为数据报选择路由之后，以及将其置入一个与指定的网络接口相关联的队列之前的这段时间里。IP把数据报长度与网络的MTU相比较，以确定是否有必要将其分片。在最简单的情况下，整个数据报可以被封装成一个网络分组或帧，因而无需分片。

对需要分片的情况，IP首先生成多个数据报，并将每个数据报中的分片比特置为1，然后将源数据报中的数据按顺序分片，并将它们装入这些数据报中。它还要在同一源数据报产生的所有数据报片中将MF（more fragment）比特置为1，末尾的分片除外。IP一边为数据报分片，一边将它们传递给网络接口发送出去。

7.2.1 为一个数据报片再次分片

如果被分片的数据报本身就是一个数据报片，分片处理会变得复杂一些。当一个数据报通过两个或更多的网关时，可能会出现这种情况。如果某一个网关为源数据报分片，分片后的数据报可能还是太大，以致无法通过沿途经过的下一个网络。因此，一个网关可能会接收到需要再次分片的数据报片。

为数据报分片和为数据报片再次分片之间的细微差别就在于网关处理MF比特的不同。当一个网关为原来未分片的数据报分片时，除了末尾的数据报片，它将其余所有分片上的MF比特都置为1。类似地，如果一个数据报片上的MF比特并未置为1，则网关会和处理原来未分片的数据报完全一样地处理这个数据报片，除末尾的分片外，将每个子分片的MF比特置为1。然而，当网关为一个非末尾的数据报片再次分片时，它会把生成的所有（子）分片中的MF比特全部置为1，因为所有这些子分片都不可能是整个数据报的末尾数据报片。

7.3 分片的实现

在示例的代码中，过程ipputp决定如何分片。

```
/* ipputp.c - ipputp */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 * ipputp  -  send a packet to an interface's output queue
 *------------------------------------------------------------------------
 */
int ipputp(ifn, nh, pep)
unsigned    ifn;
IPaddr      nh;
struct ep   *pep;
{
        struct     netif*pni = &nif[ifn];
        struct     ip    *pip;
        int        hlen, maxdlen, tosend, offset, offindg;

        if (pni->ni_state == NIS_DOWN) {
             freebuf(pep);
             return SYSERR;
        }
        pip = (struct ip *)pep->ep_data;
        if (pip->ip_len <= pni->ni_mtu) {
             pep->ep_nexthop = nh;
             pip->ip_cksum = 0;
             iph2net(pip);
             pep->ep_order & =~ EPO_IP;
             pip->ip_cksum = cksum((WORD*)pip, IP_HLEN(pip));
             return netwrite(pni, pep, EP_HLEN+net2hs(pip->ip_len));
        }
        /* else, we need to fragment it */

        if (pip->ip_fragoff & IP_DF) {
             IpFragFails++;
             icmp(ICT_DESTUR, ICC_FNADF, pip->ip_src, pep, 0);
             return OK;
        }
        maxdlen = (pni->ni_mtu - IP_HLEN(pip)) &~ 7;
        offset = 0;
        offindg = (pip->ip_fragoff & IP_FRAGOFF)<<3;
        tosend = pip->ip_len - IP_HLEN(pip);

        while (tosend > maxdlen) {
             if (ipfsend(pni,nh,pep,offset,maxdlen,offindg) != OK) {
```

```
            IpOutDiscards++;
            freebuf(pep);
            return SYSERR;
        }
        IpFragCreates++;
        tosend -= maxdlen;
        offset += maxdlen;
        offindg += maxdlen;
    }
    IpFragOKs++;
    IpFragCreates++;
    hlen = ipfhcopy(pep, pep, offindg);
    pip = (struct ip *)pep->ep_data;
    /* slide the residual down */
    memcpy(&pep->ep_data[hlen], &pep->ep_data[IP_HLEN(pip)+offset],
        tosend);
    /* keep MF, if this was a frag to start with */
    pip->ip_fragoff = (pip->ip_fragoff & IP_MF)|(offindg>>3);
    pip->ip_len = tosend + hlen;
    pip->ip_cksum = 0;
    iph2net(pip);
    pep->ep_order &= ~EPO_IP;
    pip->ip_cksum = cksum((WORD*)pip, hlen);
    pep->ep_nexthop = nh;
    return netwrite(pni, pep, EP_HLEN+net2hs(pip->ip_len));
}
```

ipputp的入口参数给出了路由对应的接口号、下一跳地址以及一个分组。如果分组的长度小于网络的MTU，ipputp就调用netwrite来发送数据报并返回调用者。如果仅用一个分组无法发送该数据报，那么ipputp将数据报顺序划分为能够装入分组中的数据报片。为了做到这一点，ipputp计算分片的最大可能长度，它必须是8的倍数，然后将数据报顺序划分为一连串最大长度的数据报片，最后剩余的数据（不管还剩余多长）被装入末尾数据报片。一旦ipputp计算出了最大分片长度，它遍历整个数据报，并调用ipfsend，发送每个数据报片。

在程序代码中包含了几个微妙的地方。首先，每个数据报片必须含有一个IP首部，因此一次可发送的最大数据量等于MTU减去IP首部的长度，然后截短成为最接近的8的倍数。其次，仅当数据报中剩余的数据大于一次可发送的最大数据量时，发送动作才会继续进行。因此，在发送末尾数据报片之前，如果末尾数据报片的长度与其他数据报片的长度相同，则不会发送最后这个数据报片。第三，发送末尾数据报片时，ipputp仅对源数据报做适当更改，而不再把生成的数据报片复制到一个新缓冲区中。第四，通常末尾数据报片中的MF比特为0。然而，当网关为一个非末尾的数据报片再次分片时，它必须设置所有子分片的MF比特为1。

7.3.1 发送一个数据报片

过程ipfsend生成并发送一个数据报片。它首先为副本数据分配一个新缓冲区，调用ipfhcopy将

IP首部和选项复制到新缓冲区中，形成新数据报，然后将作为此数据报片的数据复制到新数据报中，并将结果传递给netwrite。

```
/* ipfsend.c - ipfsend */

#include <conf.h>
#include <kernel.h>
#include <network.h>

int ipfhcopy(struct ep*, struct ep*, unsigned);

/*------------------------------------------------------------------------
 *  ipfsend -  send one fragment of an IP datagram
 *------------------------------------------------------------------------
 */
int ipfsend(pni, nexthop, pep, offset, maxdlen, offindg)
struct netif   *pni;
IPaddr         nexthop;
struct ep      *pep;
unsigned       offset, maxdlen, offindg;
{
        struct     ep    *pepnew;
        struct     ip    *pip, *pipnew;
        int        hlen, len;

        pepnew = (struct ep *)getbuf(Net.netpool);
        if (pepnew == (struct ep *)SYSERR)
             return SYSERR;
        pepnew->ep_order =~0;
        hlen = ipfhcopy(pepnew, pep, offindg);     /* copy the headers */

        pip = (struct ip *)pep->ep_data;
        pipnew = (struct ip *)pepnew->ep_data;
        pipnew->ip_fragoff = IP_MF | (offindg>>3);
        pipnew->ip_len = len = maxdlen + hlen;
        pipnew->ip_cksum = 0;

        iph2net(pipnew);
        pepnew->ep_order &=~EPO_IP;
        pipnew->ip_cksum = cksum((WORD*)pipnew, hlen);

        memcpy(&pepnew->ep_data[hlen],
             &pep->ep_data[IP_HLEN(pip)+offset], maxdlen);
        pepnew->ep_nexthop = nexthop;

        return netwrite(pni, pepnew, EP_HLEN+len);
}
```

7.3.2 复制数据报首部

过程ipfhcopy用于复制数据报首部。大多数程序代码要考虑到IP选项的细节。根据协议标准，某些选项只应当出现在第一个数据报片中，而其他一些则必须出现在所有的数据报片中。ipfhcopy遍历选项字段，并查看是否应该将某个选项复制到所有分片中。最后，当ipfhcopy返回后，ipfsend调用netwrite来发送这个数据报片。

```
/* ipfhcopy.c - ipfhcopy */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  ipfhcopy -  copy the hardware, IP header, and options for a fragment
 *------------------------------------------------------------------------
 */
int ipfhcopy(pepto, pepfrom, offindg)
struct ep   *pepto, *pepfrom;
unsigned    offindg;
{
        struct    ip    *pipfrom = (struct ip *)pepfrom->ep_data;
        unsigned        i, maxhlen, olen, otype;
        unsigned        hlen = (IP_MINHLEN<<2);

        if (offindg == 0) {
                memcpy(pepto, pepfrom, EP_HLEN+IP_HLEN(pipfrom));
                return IP_HLEN(pipfrom);
        }
        memcpy(pepto, pepfrom, EP_HLEN+hlen);

        /* copy options */

        maxhlen = IP_HLEN(pipfrom);
        i = hlen;
        while (i < maxhlen) {
                otype = pepfrom->ep_data[i];
                olen = pepfrom->ep_data[++i];
                if (otype & IPO_COPY) {
                        memcpy(&pepto->ep_data[hlen],
                                &pepfrom->ep_data[i-1], olen);
                        hlen += olen;
                } else if (otype == IPO_NOP || otype == IPO_EOOP) {
                        pepto->ep_data[hlen++] = otype;
                        olen = 1;
                }
```

```
            i += olen-1;

            if (otype == IPO_EOOP)
                break;
        }
        /* pad to a multiple of 4 octets */
        while (hlen % 4)
            pepto->ep_data[hlen++] = IPO_NOP;
        return hlen;
}
```

7.4 数据报的重组

重组数据报时，要求接收方的IP在能够重组完整的数据报之前，不断累积传入的数据报片。一旦重组成功，IP为数据报选择路由，继续送往目的站。因为IP并不保证顺序交付，协议要求IP能够接受顺序混乱的或延迟的数据报片。此外，在某个数据报的各个分片陆续到达期间，也可能混杂着其他数据报的分片。

7.4.1 数据结构

为了使数据报的重组效率更高，用于保存数据报片的数据结构必须能够做到：为构成某个特定数据报的一组数据报片快速定位；在一组数据报片中快速插入新的数据报片；有效地判断一个完整的数据报是否已经全部抵达；具有数据报片超时机制，如果在重组完成之前定时器溢出，则删除数据报片。

本例中的代码使用一个列表数组来保存数据报片。数组中的每个元素对应于一个已有数据报片到达的数据报，并包含了一个指针，指向此数据报的分片表，文件ipreass.h说明了数据结构。

```
/* ipreass.h */

/* Internet Protocol (IP)  reassembly support */

#define IP_FQSIZE      10    /* max number of frag queues              */
#define IP_MAXNF       10    /* max number of frags/datagram           */
#define IP_FTTL        60    /* time to live (secs)                    */

/* ipf_state flags */

#define IPFF_VALID     1     /* contents are valid                     */
#define IPFF_BOGUS     2     /* drop frags that match                  */
#define IPFF_FREE      3     /* this queue is free to be allocated     */

struct ipfq    {
      char     ipf_state;        /* VALID, FREE or BOGUS               */
      lPaddr   ipf_src;          /* IP address of the source           */
      short    ipf_id;           /* datagram id                        */
```

```
        int     ipf_ttl;            /* countdown to disposal          */
        int     ipf_q;              /* the queue of fragments         */
};

extern int      ipfmutex;           /* mutex for ipfqt[]              */
extern struct   ipfq    ipfqt[];    /* IP frag queue table            */

int ipfsend(struct netif*, IPaddr, struct ep*, unsigned,unsigned,unsigned);
int ipfhcopy(struct ep*, struct ep*, unsigned);
```

数组ipfqt构成了数据报片的主要数据结构，数组的每个元素对应一个数据报。结构ipfq定义了需要保存的信息，除数据报的源地址和ID字段（ipf_src和ipf_id）之外，每个元素还包含了一个寿命计数字段（ipf_ttl），用来指出该数组元素在超时之前还能保留多长时间。ipf_q字段指向一个链表，链表的表项是所有已经到达的数据报片。

重组软件必须判断某个数据报的数据报片是否已经全部抵达。为了提高判断的效率，分片链表按顺序存放。链表中的数据报片根据它们在原来的数据报中的偏移量大小排序。协议设计思想使序号的选取变得简单，因为即使是为一个数据报片再次分片时，它的偏移量还是以原来的数据报为基准的，因而能够做到在链表中插入任何数据报而不需要知道它是一次分片还是多次分片的结果。

7.4.2 互斥操作

为了保证进程在访问数据报片的链表时不会互相干扰，重组程序代码使用了一个互斥信号量ipfmutex。在文件ipreass.h中说明了它是一个外部的整数值，可被所有程序代码访问。正如我们将要看到的，由于互斥操作使系统能够使用各自独立的超时进程和重组进程，而使它显得尤其重要。

7.4.3 在链表中加入一个数据报片

IP使用传入数据报片首部中的信息来确定出对应的数据报片链表。如果几个数据报片的源地址字段和IP标识符字段中的值相同，那么这些数据报片就属于同一个数据报。过程ipreass接收数据报片，找出相应的链表，并将数据报片插入该链表中。给定一个数据报片，ipreass在分片表中搜索，看表中是否有与该数据报片所属的数据报相对应的表项。对每一个表项，它比较两者的源地址和标识符字段，如果找到匹配的表项，就调用ipfadd向链表中加入该数据报片，然后调用ipfjoin来查看是否所有的数据报片已全部到齐，可以重组数据报。如果没有发现匹配表项，ipreass分配数组中的第一个空闲表项，复制源地址和标识符字段，并将数据报片置入一个新分配的队列中。

在我们的实现过程中，使用了线性查找来为一个传入数据报片找出适当的链表。对于性能要求高的应用，这种方法似乎效率很低。当然，一些计算机的确同时接收来自许多数据报的数据报片，因而要求更快的查找方式。然而，由于大多数计算机更多地与本地环境中的计算机通信，它们很少会收到分片的数据报。再者，因为重组只能在数据报到达本机时才发生，而途经的数据流不需要重组，所以网关重组数据报的速度不需要像为数据报选择路由时那么快。因而，对典型的计算机系统来说，线性查找已经足够了。

```
/* ipreass.c - ipreass */

#include <conf.h>
#include <kernel.h>
```

```
#include <network.h>

int ipfadd(struct ipfq*, struct ep*);
struct ep   *ipfjoin (struct iptq*);

/*-----------------------------------------------------------------------
 *  ipreass  -  reassemble an IP datagram, if necessary
 *     returns packet, if complete; 0 otherwise
 *-----------------------------------------------------------------------
 */
struct ep *ipreass(pep)
struct ep   *pep;
{
        struct     ep    *pep2;
        struct     ip    *pip;
        int        firstfree;
        int        i;

        pip = (struct ip *)pep->ep_data;

        wait(ipfmutex);

        if ((pip->ip_fragoff & (IP_FRAGOFF|IP_MF)) == 0) {
             signal(ipfmutex);
             return pep;
        }
        IpReasmReqds++;
        firstfree = -1;
        for (i=0; i<IP_FQSIZE; ++i) {
             struct     ipfq *piq = &ipfqt[i];

             if (piq->ipf_state == IPFF_FREE) {
                  if (firstfree == -1)
                       firstfree = i;
                  continue;
             }
             if (piq->ipf_id != pip->ip_id)
                  continue;
             if (piq->ipf_src != pip->ip_src)
                  continue;
             /* found a match */
             if (ipfadd(piq, pep) == 0) {
                  signal(ipfmutex);
                  return 0;
             }
             pep2 = ipfjoin(piq);
             signal(ipfmutex);
```

```
        return pep2;

    }
    /* no match */

    if (firstfree < 0) {
        /* no room-- drop */
        freebuf(pep);
        signal(ipfmutex);
        return 0;
    }
    ipfqt[firstfree].ipf_q = newq(IP_FQSIZE, QF_WAIT);
    if (ipfqt[firstfree].ipf_q < 0) {
        freebuf(pep);
        signal(ipfmutex);
        return 0;
    }
    ipfqt[firstfree].ipf_src = pip->ip_src;
    ipfqt[firstfree].ipf_id = pip->ip_id;
    ipfqt[firstfree].ipf_ttl = IP_FTTL;
    ipfqt[firstfree].ipf_state = IPFF_VALID;
    ipfadd(&ipfqt[firstfree], pep);
    signal(ipfmutex);
    return 0;
}

int     ipfmutex;
struct ipfq ipfqt[IP_FQSIZE];
```

7.4.4 溢出时的丢弃

过程ipfadd将数据报片插入给定的链表。在正常情况下，此过程显得微不足道。ipfadd只是调用enq将数据报片放入队列中，然后将数据报的寿命字段复位。

但是，在分片列表空间已全满的情况下，新的数据报片无法加入链表中。当这种情况发生时，ipfadd丢弃对应数据报的所有数据报片，并释放ipfqt数组中对应的表项。表面上看，这种做法似乎有些奇怪，然而丢弃整个链表的原因其实很简单：只要丢失一个数据报片，IP就永远无法重组并继续处理这个数据报，因此，释放它所对应的数据报片占用的内存空间，可以使其他数据报有可能进行重组。更进一步讲，一旦链表的容量达到极限值，它就无法再扩展。此时若保留该链表，只会徒然耗费内存资源，而不会对成功重组数据报起什么作用。

```
/* ipfadd.c - ipfadd */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>
```

```
/*------------------------------------------------------------------------
 *  ipfadd  -  add a fragment to an IP fragment queue
 *------------------------------------------------------------------------
 */
Bool ipfadd(iq, pep)
struct ipfq *iq;
struct ep   *pep;
{
        struct ip  *pip;
        int        fragoff;

        if (iq->ipf_state != IPFF_VALID) {
                freebuf(pep);
                return FALSE;
        }
        pip = (struct ip *)pep->ep_data;
        fragoff = pip->ip_fragoff & IP_FRAGOFF;

        if (enq(iq->ipf_q, pep, -fragoff) < 0) {
                /* overflow-- free all frags and drop */
                freebuf(pep);
                IpReasmFails++;
                while (pep = (struct ep *)deq(iq->ipf_q)) {
                        freebuf(pep);
                        IpReasmFails++;
                }
                freeq(iq->ipf_q);
                iq->ipf_state = IPFF_BOGUS;
                return FALSE;
        }
        iq->ipf_ttl = IP_FTTL;          /* restart timer */
        return TRUE;
}
```

7.4.5 测试一个完整的数据报

当IP向链表中增加一个新数据报片后，必须检查它是否已拥有了组成一个完整数据报的全部数据报片。过程ipfjoin检查数据报片链表，看它们是否能形成一个完整数据报。

```
/* ipfjoin.c - ipfjoin */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>
```

```
struct ep   *ipfcons(struct ipfq*);

/*-----------------------------------------------------------------------
 *  ipfjoin  -  join fragments, if all collected
 *-----------------------------------------------------------------------
 */
struct ep *ipfjoin(iq)
struct ipfq *iq;
{
        struct ep  *pep;
        struct ip  *pip = 0;
        int        off, packoff;

        if (iq->ipf_state == IPFF_BOGUS)
                return 0;
        /* see if we have the whole datagram */

        off = 0;
        while (pep=(struct ep *)seeq(iq->ipf_q)) {
                pip = (struct ip *)pep->ep_data;
                packoff =  (pip->ip_fragoff & IP_FRAGOFF)<<3;
                if (off < packoff) {
                        while(seeq(iq->ipf_q))
                                /*empty*/;
                        return 0;
                }
                off = packoff + pip->ip_len - IP_HLEN(pip);
        }
        if (off > MAXLRGBUF) {          /* too big for us to handle */
                while (pep = (struct ep *)deq(iq->ipf_q))
                        freebuf(pep);
                freeq(iq->ipf_q);
                iq->ipf_state = IPFF_FREE;
                return 0;
        }
        if (pip == 0 || (pip->ip_fragoff & IP_MF) == 0)
                return ipfcons(iq);

        return 0;
}
```

在证实了指定的某个数据报片链表正处于使用状态后，ipfjoin进入一个循环语句，遍历链表中的每个数据报片。它先设置变量off的值为0，然后利用变量off来查看它当前指向的数据报片所处的位置是否正确，是否就是它在原来的数据报中所处的位置。首先，ipfjoin查看当前数据报片的偏移量是否与off值相符。如果当前数据报片的偏移量超过了off的值，那么必定还有尚未到达的数据

报片，因此ipfjoin返回0（这意味着数据报片还不能被重组）。如果偏移量与off值一致，那么ipfjoin通过将off值加上当前数据报片的长度，计算出下一个数据报片的偏移量。

一旦ipfjoin证实所有的数据报片已被全部收集齐，则通过判断来保证数据报可以被装入一个大缓冲区中。软件只能处理装入大缓冲区中的数据报，因为数据报在被传送给应用程序之前，必须在连续的存储空间中重组。这样，如果数据报无法装入到一个缓冲区中，ipfjoin就将这些数据报片丢弃。最后，对于可装入一个缓冲区的数据报来说，ipfjoin调用ipfcons来收集数据报片并重建完整的数据报。

7.4.6 将数据报片组装成完整的数据报

过程ipfcons将数据报片重组成完整的数据报。除了将每个数据报片中的数据复制到合适的位置外，它还要建立一个有效的数据报首部。所需要的信息来自第一个数据报片的首部中，但数据报长度字段中的值要修改为整个数据报的长度。ipfcons将MF比特复位为0，以表明这个重建的数据报不是数据报片，并将段偏移字段置为0。在数据报的重组过程中，ipfcons逐个释放保存每个数据报片的缓冲区。当重组完成后，ipfcons释放分片表ipfqt中对应的表项。

```
/* ipfcons.c - ipfcons */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  ipfcons  -  construct a single packet from an IP fragment queue
 *------------------------------------------------------------------------
 */
struct ep *
ipfcons (struct ipfq *iq)
{
        struct ep     *pep, *peptmp;
        struct ip     *pip;
        int           off, seq;

        pep = (struct ep *) getbuf(Net.lrgpool);
        if (pep == (struct ep *)SYSERR) {
                while (peptmp = (struct ep *)deq(iq->ipf_q) {
                        IpReasmFails++;
                        freebuf (peptmp);
        }
        freeq(iq->ipf_q);
        iq->ipf_state = IPFF_FREE;
        return 0;
        }
        /* copy the Ether and IP headers */
```

```
peptmp = (struct ep *)deq(iq->ipf_q);
pip = (struct ip *)peptmp->ep_data;
off = IP_HLEN (pip);
seq = 0;
memcpy(pep, peptmp, EP_HLEN+off);

/* copy the data */
while (peptmp !=0) {
        int dlen, doff;

        pip = (struct ip  *)peptmp->ep_data;
        doff = IP_HLEN(pip) + seq
                - ((pip->ip_fragoff&IP_FRAGOFF)<<3);
                dlen = pip->ip_len - doff;
                memcpy (pep->ep_data+off, peptmp >ep_data+doff, dlen);
                off += dlen;
                seq += dlen;
                freebuf(peptmp);
                peptmp = (struct ep *)deq(iq->ipf_q);
        }
                /* fix the large packet header */
                pip = (struct ip *)pep->ep_data;
                pip->ip_len = off;
                pip->ip_fragoff = 0;

                /* release resources */
                freeq(iq->ipf_q);
                iq->ipf_state = IPFF_FREE;
                IpReasmOKs++;
                return pep;
}
```

7.5 数据报片链表的维护管理

因为IP是不可靠交付机制，数据报在互联网中传输时有可能会丢失。如果一个数据报片丢失，接收端的IP软件就不能重组原来的数据报。更进一步说，因为IP不提供确认机制，所以无法重传数据报片。因此一旦某个数据报片丢失，IP将永远无法恢复它所对应的数据报。事实上，如TCP的高层协议，会使用一个新的数据报进行重传[1]。

为了使丢失数据报片的数据报不再浪费存储资源，并防止因为标识符字段的重新使用而给IP带来混乱，当已经不可能再收到剩余数据报片时，IP必须定期检查数据报片链表并丢弃旧链表，过程

① TCP报文段的每一次重传，要使用一个具有惟一IP标识符的数据报，因此在组装时，IP不会把两次传输的数据报片弄混。

ipftimer执行定期的清理工作。

```
/* ipftimer.c - ipftimer */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 * ipftimer - update time-to-live fields and delete expired fragments
 *------------------------------------------------------------------------
 */
void ipftimer(gran)
int    gran;
{
        struct     ep    *pep;
        struct     ip    *pip;
        int              i;

        wait(ipfmutex);
        for (i=0; i<IP_FQSIZE; ++i) {
             struct ipfq *iq = &ipfqt[i];

             if (iq->ipf_state == IPFF_FREE)
                  continue;
             iq->ipf_ttl -= gran;
             if (iq->ipf_ttl <= 0) {
                  if (iq->ipf_state == IPFF_BOGUS) {
                       /* resources already gone */
                       iq->ipf_state = IPFF_FREE;
                       continue;
                  }
                  if (pep = (struct ep *)deq(iq->ipf_q)) {
                       IpReasmFails++;
                       pip = (struct ip *)pep->ep_data;
                       icmp(ICT_TIMEX, ICC_FTIMEX,
                            pip->ip_src, pep, 0);
                  }
                  while (pep = (struct ep *)deq(iq->ipf_q)) {
                       IpReasmFails++;
                       freebuf(pep);
                  }
                  freeq(iq->ipf_q);
                  iq->ipf_state = IPFF_FREE;
             }
        }
```

```
        signal(ipfmutex);
}
```

ipftimer在每次被调用时（通常每秒一次）遍历数据报片链表，它递减每个表项中的寿命字段，并丢弃其值达到0的链表。在丢弃一个链表时，ipftimer从链表中提取出第一个结点，并利用分组缓冲区向源地址发送ICMP超时报文。在发送了ICMP报文之后，ipftimer释放链表，并将ipfqt中的表项设置为空闲状态，可以被再次使用。

7.6 初始化

重组数据报片时所使用的数据结构的初始化过程很简单。过程ipfinit生成互斥信号量并将ipfqt数组中的每个表项置为空闲状态，使之可用。

```
/* ipfinit.c - ipfinit */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 * ipfinit  -  initialize IP fragment queue data structures
 *------------------------------------------------------------------------
 */
void ipfinit(void)
{
 int  i;

 ipfmutex = screate(1);
 for (i=0; i<IP_FQSIZE; ++i)
      ipfqt[i].ipf_state = IPFF_FREE;
}
```

7.7 小结

所有实现IP协议的机器必须能够将外发的数据报分片。并且当被分片的数据报到达后，将其重组。

事实上，网关通常在遇到一个太大的数据报而无法将其作为网络MTU传送到它必须经过的网络上时，就需要将这个数据报分片。数据报的分片过程包括为每个数据报片复制数据报的首部，设置段偏移字段和MF比特，复制部分数据，并每次发送一个所得到的数据报片。软件在IP为数据报选择路由之后，以及IP将其置入一个与特定网络接口相关联的输出队列之前，将一个数据报分片。与重组相比，分片要更直截了当。

为了执行重组，IP采用了一个数据结构，用于把来自某一特定数据报的所有数据报片收集起来。一旦所有的数据报片都已聚齐，就可以重组（重建）数据报并继续处理。

重组与一个维护进程并行工作。每当某个数据报的一个新数据报片到达时，IP在分片表中将此

数据报的寿命字段复位。维护进程定期检查数据报片链表，并递减每个表项中的寿命字段。如果在所有数据报片到达之前，寿命已达到0，则维护进程丢弃整个数据报。

深入研究

许多教科书中描述了链表存储区采用的算法和数据结构。有关分片管理的更多信息可在IP规范［RFC 791］以及主机需求文档［RFC 1122］中查找到。

习题

7.1 细心阅读IP规范，是否会存在来自不同数据报的两个数据报片的IP源地址和标识字段的值都相同的情况？为什么？提示：考虑机器的重启动。

7.2 仔细查看ipputp和ipfhcopy过程。会不会出现ipputp将分片的最大有效长度估计得过小的情况？为什么？

7.3 例子中的代码选择了最大的有效分片长度，并以这个长度为单位将数据报划分为许多数据块，最后将剩余的数据划为一块。如果将数据报尽可能地划分成同等长度的块，这样做有什么优点？为什么？

7.4 过程ipreass为每个新创建的数据报片链表的寿命字段赋予一个固定的值。有没有更好的选择初始寿命值的办法？为什么？

7.5 修改数据报片的数据结构，使之由线性查找变为计算散列值，并测试其性能的改进程度。会得到什么样的结论？在什么情况下，采用散列表能够节省时间？

7.6 使用ping命令生成不同长度的数据报，并将它们发往一个远程的计算机。看你是否能从它们远程环回时不同的延迟时间中推测出发生分片的长度阈值。从中能得到关于分片费用的什么结论？

7.7 仔细阅读IP规范说明书。例子中的代码是否正确地处理了“不分片”（do not fragment）比特？为什么？

7.8 考虑一个每秒接受1000个数据报片的网络容量。这样的网络在为数据报片选择寿命字段时有什么样的限制？假设IP为所有数据报片使用一个超时时间常量。

7.9 每当一个数据报片到达时，就为这个数据报的寿命字段重置初始值，与在第一个数据报片到达时给整个数据报设置一个寿命字段相比，有什么优点及缺点？

第8章 IP：差错处理（ICMP）

8.1 引言

Internet 控制报文协议（Internet Control Message Protocol，ICMP）是IP的一个组成部分，用于提供差错报告。ICMP处理几种类型的差错情况，并总是向发出数据报的源站报告。任何使用IP的计算机都必须能够接受ICMP报文，并对差错报告进行响应，改变自己的行为。网关还必须随时做好准备，当传入的数据报引起异常问题时，能够生成ICMP差错报文。

本章详述了ICMP处理过程的细节，也给出了生成差错报文的程序代码，以及当这样的差错报文抵达时，处理这些差错报文的程序代码。

8.2 ICMP报文格式

和那些具有固定报文格式的协议不同，ICMP报文是类型相关（type-dependent）的。报文中字段的数目，每个字段的解释，以及报文携带的数据量都与其类型有关。

8.3 ICMP报文的实现

如下所示，文件icmp.h中含有关于ICMP差错报文的说明。类型相关的报文使ICMP报文格式的说明比其他协议的类似说明要复杂得多。结构icmp定义了报文格式。所有ICMP报文都由一个固定的首部开始，这个首部由字段ic_type（报文类型）、ic_code（报文子类型）和ic_cksum（报文校验和）定义。在ICMP报文中，紧接着首部的32位与报文类型有关，并采用C语言的union结构定义。在ICMP回送请求（echo request）和回送应答（echo reply）报文中，它包含的是16位标识符和16位序号。在ICMP的重定向（redirect）报文中，这32位是某个网关的IP地址。在参数有问题（parameter problem）的报文中，这32位包含一个8位的指针和3个八位组填充字段。在其他类型的报文中，该32位全为0。最后，字段ic_data定义了ICMP报文的数据区。就如我们以前见到过的许多协议一样，尽管一个报文中会包含多个八位组的数据，但结构ic_data仅仅说明了数据的第一个八位组。

除了所有ICMP报文需要用到的符号常量外，icmp.h还定义了一些缩写名，可以用这些较短的名字来代表union中的长字段名称。例如，程序员可以利用缩写名something.ic_gw来指定网关地址字段，而不用它的全名something.icu.ic2_gw。

```
/* icmp.h */

/* Internet Control Message Protocol Constants and Packet Format */

/* ic_type field */
```

```
#define ICT_ECHORP      0     /* Echo reply                          */
#define ICT_DESTUR      3     /* Destination unreachable             */
#define ICT_SRCQ        4     /* Source quench                       */
#define ICT_REDIRECT    5     /* Redirect message type               */
#define ICT_ECHORQ      8     /* Echo request                        */
#define ICT_TIMEX       11    /* Time exceeded                       */
#define ICT_PARAMP      12    /* Parameter Problem                   */
#define ICT_TIMERQ      13    /* Timestamp request                   */
#define ICT_TIMERP      14    /* Timestamp reply                     */
#define ICT_INFORQ      15    /* Information request                 */
#define ICT_INFORP      16    /* Information reply                   */
#define ICT_MASKRQ      17    /* Mask request                        */
#define ICT_MASKRP      18    /* Mask reply                          */

/* ic_code field */
#define ICC_NETUR       0     /* dest unreachable, net unreachable   */
#define ICC_HOSTUR      1     /* dest unreachable, host unreachable  */
#define ICC_PROTOUR     2     /* dest unreachable, proto unreachable */
#define ICC_PORTUR      3     /* dest unreachable, port unreachable  */
#define ICC_FNADF       4     /* dest unr, frag needed & don't frag  */
#define ICC_SRCRT       5     /* dest unreachable, src route failed  */

#define ICC_NETRD       0     /* redirect: net                       */
#define ICC_HOSTRD      1     /* redirect: host                      */
#define IC_TOSNRD       2     /* redirect: type of service, net      */
#define IC_TOSHRD       3     /* redirect: type of service, host     */
#define ICC_TIMEX       0     /* time exceeded, ttl                  */
#define ICC_FTIMEX      1     /* time exceeded, frag                 */

#define IC_HLEN         8     /* octets                              */
#define IC_PADLEN       3     /* pad length (octets)                 */

#define IC_RDTTL        300   /* ttl for redirect routes             */

/* ICMP packet format (following the IP header)                      */

struct icmp     {               /* ICMP packet                       */
        char    ic_type;        /* type of message (ICT_* above)     */
        char    ic_code;        /* code (ICC_* above)                */
        short   ic_cksum;       /* checksum of ICMP header+data      */

        union   {
                struct {
                        int ic1_id:16;      /* echo type, a message id */
                        int ic1_seq:16;     /* echo type, a seq. number*/
        } ic1;
        IPaddr  ic2_gw;         /* for redirect, gateway             */
```

```
        struct {
                char ic3_ptr;   /* pointer, for ICT_PARAMP          */
                char ic3_pad[IC_PADLEN];
        } ic3;
        int     ic4_mbz;        /* must be zero                     */
        } icu;
        char ic_data[1];        /* data area of ICMP message        */
};

/* format 1 */
#define     ic_id       icu.ic1.ic1_id
#define     ic_seq      icu.ic1.ic1_seq

/* format 2 */
#define     ic_gw       icu.ic2_gw

/* format 3 */
#define     ic_ptr      icu.ic3.ic3_ptr
#define     ic_pad      icu.ic3.ic3_pad

/* format 4 */
#define     ic_mbz      icu.ic4_mbz
int icmp (u_short, u_short, IPaddr, void*, void*);
```

8.4 传入ICMP报文的处理

当以本地机器为目的站的一个IP数据报携带一个ICMP报文到达时，IP进程将其传递给过程icmp_in。

```
/* icmp_in.c - icmp_in */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <proc.h>

int icredirect (struct ep*);
void icsetsrc (struct ip*);
/*------------------------------------------------------------------------
 *  icmp_in  -  handle ICMP packet coming in from the network
 *------------------------------------------------------------------------
 */
int icmp_in(pni, pep)
struct netif   *pni;
struct ep      *pep;
{
```

```
        struct     ip    *pip;
        struct     icmp  *pic;
        int        i, len;

        pip = (struct ip *)pep->ep_data;
        pic = (struct icmp *) pip->ip_data;

        len = pip->ip_len - IP_HLEN(pip);
        if (cksum((WORD*)pic,len)) {
             IcmpInErrors++;
             freebuf(pep);
             return SYSERR;
        }
        IcmpInMsgs++;
        switch(pic->ic_type) {
        case ICT_ECHORQ:
             IcmpInEchos++;
             return icmp(ICT_ECHORP, 0, pip->ip_src, pep, 0);
        case ICT_MASKRQ:
             IcmpInAddrMasks++;
             if (!gateway) {
                  freebuf(pep);
                  return OK;
             }
             pic->ic_type = (char) ICT_MASKRP;
             *(IPaddr*)(pic->ic_data = netmask(pip->ip_dst);
             break;
        case ICT_MASKRP:
             IcmpInAddrMaskReps++;
             for (i=0; i<Net.nif; ++i)
                  if (nif[i].ni_ip == pip->ip_dst)
                       break;
             if (i != Net.nif) {
                  setmask(i, *(IPaddr*)pic->ic_data);
                  send(pic->ic_id, ICT_MASKRP);
             }
             freebuf(pep);
             return OK;
        case ICT_ECHORP:
             IcmpInEchoReps++;
             if (send(pic->ic_id, (int)pep) != OK)
                  freebuf(pep);
             return OK;
        case ICT_REDIRECT:
             IcmpInRedirects++;
             icredirect(pep);
```

```
        return OK;
    case ICT_DESTUR: IcmpInDestUnreachs++;  freebuf(pep);     return OK;
    case ICT_SRCQ:   IcmpInSrcQuenchs++;    freebuf(pep);     return OK;
    case ICT_TIMEX:  IcmpInTimeExcds++;     freebuf(pep);     return OK;
    case ICT_PARAMP: IcmpInParmProbs++;     freebuf(pep);     return OK;
    case ICT_TIMERQ: IcmpInTimestamps++;    freebuf(pep);     return OK;
    case ICT_TIMERP: IcmpInTimestampReps++; freebuf(pep);     return OK;
    default:
        IcmpInErrors++;
        freebuf(pep);
        return OK;
    }
    icsetsrc(pip);

    len = pip->ip_len - IP_HLEN(pip);

    pic->ic_cksum = 0;
    pic->ic_cksum = cksum((WORD*)pic,len);

    IcmpOutMsgs++;
    ipsend(pip->ip_dst, pep, len, IPT_ICMP, IPP_INCTL, IP_TTL);
    return OK;
}
```

icmp_in的第二个入口参数是一个指针，指向一个含有IP数据报的缓冲区。icmp_in找出该数据报中的ICMP报文，并利用ICMP的类型字段，为其选择六种ICMP报文类型中一种。程序代码分别处理每种类型。

为了处理ICMP回送请求报文，icmp_in调用icmp（在以后讨论）来生成一个ICMP回送应答报文。与此不同的是，为了处理ICMP回送应答报文，ICMP从报文中取出标识字段的值，并认为它就是发送了回送请求的进程的标识号，并将应答分组发送给这个进程。

为了响应ICMP地址掩码请求（address mask request），icmp_in首先将该报文改变为地址掩码应答（address mask reply），然后使用netmask来找出适当的子网掩码，并从switch语句中跳出，以发送这个应答分组。

对于一个ICMP地址掩码应答报文，icmp_in首先遍历各个接口，直至找到一个与应答分组中的网络地址相匹配的接口，然后调用setmask（如源程序所示）为该接口设置子网掩码。它向setmask传递从应答分组中得到的子网掩码。最后，icmp_in调用send向已等待应答的过程发送报文。

icmp_in调用过程icredirect来处理传入的ICMP重定向报文。下一节中将详述icredirect是如何修改选路表的。

对于所有这些情况，甚至包括未加处理的ICMP报文，icmp_in都要将其累加到传入报文计数值中。在稍后的章节中将解释SNMP是如何使用这些计数值的。

8.5 ICMP重定向报文的处理

过程icredirect处理一个重定向的请求。

```
/* icredirect.c - icredirect */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  icredirect -  handle an incoming ICMP redirect
 *------------------------------------------------------------------------
 */
int icredirect(pep)
struct ep   *pep;
{
        struct     route*prt;
        struct     ip    *pip, *pip2;
        struct     icmp *pic;
        IPaddr           mask;

        pip = (struct ip *)pep->ep_data;
        pic = (struct icmp *)pip->ip_data;
        pip2 = (struct ip *)pic->ic_data;

        if (pic->ic_code == ICC_HOSTRD)
             mask = ip_maskall;
        else
             mask = pip2->ip_dst;
        prt = rtget(pip2->ip_dst, RTF_LOCAL);
        if (prt == 0) {
             freebuf(pep);
             return OK;
        }
        if (pip->ip_src == prt->rt_gw) {
             rtdel(pip2->ip_dst, mask);
             rtadd(pip2->ip_dst, mask, pic->ic_gw, prt->rt_metric,
                        prt->rt_ifnum, IC_RDTTL);
        }
        rtfree(prt);
        freebuf(pep);
        return OK;
}
```

icrediect从重定向报文中提取指定的目的地址，调用netmask来计算其对应的子网掩码，并使用rtget查找现有路由。如果当前路由指向发送该重定向报文的网关，icredirect删除这条已经存在的路由，并利用重定向报文中指定的新网关，在选路表中增加一个新路由。

8.6 设置子网掩码

当 icmp_in 接收到一个“子网掩码应答”后，就调用过程 setmask 记录该网络接口结构的子网掩码。

```
/* setmask.c - setmask */

#include <conf.h>
#include <kernel.h>
#include <network.h>

extern int bsdbrc;     /* use Berkeley (all 0's) broadcast */

/*------------------------------------------------------------------------
 *  setmask - set the net mask for an interface
 *------------------------------------------------------------------------
 */
int
setmask(unsigned ifn, IPaddr mask)
{
        IPaddr defmask;

        if (nif[ifn].ni_svalid) {
                /* one set already-- fix things */

                rtdel(nif[ifn].ni_subnet, nif[ifn].ni_mask);
                rtdel(nif[ifn].ni_brc, ip_maskall);
                rtdel(nif[ifn].ni_subnet, ip_maskall);
        }
        nif[ifn].ni_mask = mask;
        nif[ifn].ni_svalid = TRUE;
        defmask = netmask(nif[ifn].ni_ip);

        nif[ifn].ni_subent = nif[ifn].ni_ip & nif[ifn].ni_mask;
        if (bsdbrc)
                nif[ifn].ni_brc = nif[ifn].ni_subnet;
        else
                nif[ifn].ni_brc = nif[ifn].ni_subent |
                                ~nif[ifn].ni_mask;
        /* set network (not subnet) broadcast */
        nif[ifn].ni_nbrc = nif[ifn].ni_ip | ~defmask;

        /* install routes */
        /* net */
        rtadd(nif[ifn].ni_subnet, nif[ifn].ni_mask, nif[ifn].ni_ip,
```

```
        0, ifn, RT_INF);
if (bsdbrc) {
        IPaddr aobrc;           /* all 1's broadcast */

        aobrc = nif[ifn].ni_subnet | ~nif[ifn].ni_mask;
        rtadd(aobrc, ip_maskall,nif[ifn].ni_ip, 0,
                NI_LOCAL, RT_INF);
} else /* broadcast (all 1's) */
        rtadd(nif[ifn].ni_brc, ip_maskall, nif[ifn].ni_ip, 0,
                NI_LOCAL, RT_INF);
/* broadcast (all 0's) */
rtadd(nif[ifn].ni_subnet, ip_maskall, nif[ifn].ni_ip, 0,
                NI_LOCAL, RT_INF);
return OK;
}

IPaddr ip_maskall = 0xffffffff;
```

由于改变子网掩码的同时还要改变这个网络地址对应的路由，setmask首先调用rtdel来删除当前接口地址、广播地址和子网广播地址对应的路由，然后它将新的子网掩码复制到字段ni_mask中，并将ni_svalid字段置为TRUE。

当一个新的掩码被记录下之后，setmask为该接口计算一个新的子网地址以及子网广播地址。最后，它调用rtadd为子网和子网广播地址安装新路由。

8.7 为一个ICMP分组选择源地址

对那些需要进行应答的情况（例如，ICMP回送请求），ICMP必须掉换数据的源地址和目的地址。要做到这一点，过程icmp调用icsetsrc，如下所示。

```
/* icsetsrc.c - icsetsrc */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  icsetsrc -  set the source address on an ICMP packet
 *------------------------------------------------------------------------
 */
void icsetsrc(pip)
struct ip   *pip;
{
        int  i;

        for (i=0; i<Net.nif; ++i) {
```

```
            if (i == NI_LOCAL)
                continue;
            if (netmatch(pip->ip_dst,nif[i].ni_ip,nif[i].ni_mask,0))
                break;
        }
        if (i == Net.nif)
            pip->ip_src = ip_anyaddr;
        else
            pip->ip_src = nif[i].ni_ip;
}
```

icsetsrc 遍历所有网络接口，并寻找与 ICMP 报文中 IP 目的地址相匹配并且与该接口相关联的网络或子网 IP 地址。如果它找到一个匹配的地址，icsetsrc 将该网络的本地机器地址复制到数据报的源地址字段中。在找不到匹配地址的情况下，icsetsrc 用 ip_anyaddr（全“0”）填充数据报的源地址字段，使得选路程序能用一个接口的地址替换它，该接口是所选择的路由将要经过的接口。

8.8 生成 ICMP 差错报文

网关在发生网络拥塞、寿命字段超时或其他差错情况时生成 ICMP 差错报文。它是调用过程 icmp 产生并发送报文的。

```
/* icmp.c - icmp */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*
 * ICT_REDIRECT   - pa2 == gateway address
 * ICT_PARAMP     - pa2 == (packet) pointer to parameter error
 * ICT_MASKRP     - pa2 == mask address
 * ICT_ECHORQ     - pa1 == seq, pa2 == data size
 */

struct ep *icsetbuf(int, void *, Bool *, Bool *);
Bool icerrok(struct ep *);
void icsetsrc(struct ip *);
int icsetdata(int, struct ip *, void *);

/*------------------------------------------------------------------------
 *  icmp -  send an ICMP message
 *------------------------------------------------------------------------
 */
int
icmp(type, code, dst, pa1, pa2)
u_short     type, code;
```

```
IPaddr dst;
void   *pa1, *pa2;
{
        struct ep       *pep;
        struct ip       *pip;
        struct icmp     *pic;
        Bool            isresp, iserr;
        int             datalen;

        IcmpOutMsgs++;

        pep = icsetbuf(type, pa1, &isresp, &iserr);
        if (pep == 0) {
                IcmpOutErrors++;
                return SYSERR;
        }
        pip = (struct ip *)pep->ep_data;
        pic = (struct icmp *) pip->ip_data;

        datalen = IC_HLEN;

        /* we fill in the source here, so routing won't break it */

        if (isresp) {
                if (iserr) {
                        if (!icerrok(pep)) {
                                freebuf(pep);
                                return OK;
                        }
                        memcpy(pic->ic_data, pip, IP_HLEN(pip)+8);
                        datalen += IP_HLEN(pip)+8;
                }
                icsetsrc(pip);
        } else
                pip->ip_src = ip_anyaddr;
        pip->ip_dst = dst;

        pic->ic_type = (char) type;
        pic->ic_code = (char) code;
        if (!isresp) {
                if (type == ICT_ECHORQ)
                        pic->ic_seq = (int) pa1;
                else
                        pic->ic_seq = 0;
                pic->ic_id = getpid();
        }
```

```
        datalen += icsetdata(type, pip, pa2);

        pic->ic_cksum = 0;
        pic->ic_cksum = cksum((WORD *)pic, datalen);

        ipsend(dst, pep, datalen, IPT_ICMP, IPP_INCTL, IP_TTL);
        return OK;
}
```

icmp 将 ICMP 报文中的 type 和 code 字段作为入口参数，同时还有一个参数是 IP 目的地址，而最后两个参数的内容通常是指针。有关最后两个参数的确切含义和类型与 ICMP 报文类型有关。例如，对 ICMP 回送请求报文来说，参数 pa1 中包含了一个（整型）序号，而参数 pa2 中包含的是（整型）数据长度值。对于 ICMP 回送应答，参数 pa1 中含有一个指针，指向引发该应答的 ICMP 回送请求分组，参数 pa2 未用（其值为 0）。

为了生成一个 ICMP 报文，过程 icmp 调用 icsetbuf 为其分配一个缓冲区。为了保证遵守协议规定，它在向 IP 发送报文之前，先填入数据报的源地址。对于应答分组，icmp 利用接收到的与之相对应的请求分组中的目的地址填写该应答分组的源地址。否则，它调用 ip_anyaddr 来填写源地址，并允许 IP 选路过程为其选择一个外发地址。对于应答分组，icmp 还要调用 icerrok，以证实它没有生成一个差错报文的差错报文。

接着 icmp 填写首部中剩余的字段，包括 type 和 code 字段。对一个回送请求，它将标识字段置为发送该请求的进程的标识号。最后，它调用 icsetdata 来填写数据区，计算 ICMP 校验和，并调用 ipsend 发送数据报。

8.9　避免出现关于差错报文的差错报文

过程 icerrok 检查引发问题的数据报，以确定网关可以为其发送一个差错报文。其规则很简单：网关永远不应该为一个差错报文产生一个差错报文，也不应该为除第一个数据报片之外的其他任何一个数据报片，或广播数据报片产生差错报文。程序代码检查每种情况，如果不允许产生差错报文，则返回 FALSE，而在允许发送差错报文的情况下，返回 TRUE。

```
/* icerrok.c - icerrok */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  icerrok -  is it ok to send an error response?
 *------------------------------------------------------------------------
 */
Bool icerrok(pep)
struct ep   *pep;
{
        struct    ip   *pip = (struct ip *)pep->ep_data;
```

```
        struct     icmp *pic = (struct icmp *)pip->ip_data;

        /* don't send errors about error packets... */

        if (pip->ip_proto == IPT_ICMP)
             switch(pic->ic_type) {
             case ICT_DESTUR:
             case ICT_REDIRECT:
             case ICT_SRCQ:
             case ICT_TIMEX:
             case ICT_PARAMP:
                  return FALSE;
             default:
                  break;
             }
        /* ...or other than the first of a fragment */

        if (pip->ip_fragoff & IP_FRAGOFF)
             return FALSE;
        /* ...or broadcast or multicast packets */

        if (isbrc(pip->ip_dst) || IP_CLASSD(pip->ip_dst))
             return FALSE;
        return TRUE;
}
```

8.10 为ICMP报文分配缓冲区

过程icsetbuf为一个ICMP差错报文分配缓冲区，并设置两个布尔变量，其中一个变量指出该报文是否是一个差错报文（或是一个信息请求），而另一个指出这个报文的类型是否是对上一个请求做出的应答。

```
/* icsetbuf.c - icsetbuf */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  icsetbuf -  set up a buffer for an ICMP message
 *------------------------------------------------------------------------
 */
struct ep *icsetbuf(type, pal, pisresp, piserr)
int    type;
void   *pal;
Bool   *pisresp, *piserr;
```

```
{
        struct     ep    *pep;

        *pisresp = *piserr = FALSE;

        switch (type) {
        case ICT_REDIRECT:
             pep = (struct ep *)getbuf(Net.netpool);
             if ((int)pep == SYSERR)
                  return 0;
             memcpy(pep, pal, MAXNETBUF);
             pal = (void *)pep;
             *piserr = TRUE;
             break;
        case ICT_DESTUR:
        case ICT_SRCQ:
        case ICT_TIMEX:
        case ICT_PARAMP:
             pep = (struct ep *)pal;
             *piserr = TRUE;
             break;
        case ICT_ECHORP:
        case ICT_INFORP:
        case ICT_MASKRP:
             pep = (struct ep *)pal;
             *pisresp = TRUE;
             break;
        case ICT_ECHORQ:
        case ICT_TIMERQ:
        case ICT_INFORQ:
        case ICT_MASKRQ:
             pep = (struct ep *)getbuf(Net.lrgpool);
             if ((int)pep == SYSERR)
                  return 0;
             break;
        case ICT_TIMERP:          /* Not Implemented */
             /* IcmpOutTimestampsReps++; */
             IcmpOutErrors--;     /* Kludge: we increment above */
             freebuf(pal);
             return 0;
        default:
             kprintf("icsetbuf: unknown ICMP type (%d)\n", type);
             return 0;
        }
        if (*piserr)
             *pisresp = TRUE;
```

```
switch (type) {         /* Update MIB Statistics */
case ICT_ECHORP:        IcmpOutEchos++;             break;
case ICT_ECHORQ:        IcmpOutEchoReps++;          break;
case ICT_DESTUR:        IcmpOutDestUnreachs++;      break;
case ICT_SRCQ:          IcmpOutSrcQuenchs++;        break;
case ICT_REDIRECT:      IcmpOutRedirects++;         break;
case ICT_TIMEX:         IcmpOutTimeExceds++;        break;
case ICT_PARAMP:        IcmpOutParmProbs++;         break;
case ICT_TIMERQ:        IcmpOutTimestamps++;        break;
case ICT_TIMERP:        IcmpOutTimestampReps++;     break;
case ICT_MASKRQ:        IcmpOutAddrMasks++;         break;
case ICT_MASKRP:        IcmpOutAddrMaskReps++;      break;
}
return pep;
}
```

程序代码很直观，分为四种基本情况。对绝大多数的应答，icsetbuf重新利用抵达的请求分组占用的缓冲区（即返回由入口参数pa1提供的地址）。对于没有具体实现的报文类型，icsetbuf释放引起差错的数据报，并返回SYSERR。对含有大量数据的ICMP报文（如一个“回送应答”），icsetbuf为其分配一个大缓冲区。对其他不能利用原缓冲区的报文，icsetbuf为它们分配一个标准缓冲区。

8.11 ICMP报文中的数据部分

过程icsetdata生成ICMP报文中的数据部分。它所采取的行动与报文的类型有关，icsetdata将该类型值作为一个入口参数。

```
/* icsetdata.c - icsetdata */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/* ECHOMAX must be an even number */
#define     ECHOMAX(pip)    (MAXLRGBUF-IC_HLEN-IP_HLEN(pip)-EP_HLEN-EP_CRC)

/*------------------------------------------------------------------------
 *  icsetdata -  set the data section. Return value is data length
 *------------------------------------------------------------------------
 */
int icsetdata(type, pip, pa2)
int         type;
struct ip   *pip;
void        *pa2;
{
      struct    icmp *pic = (struct icmp *)pip->ip_data;
```

```
int         i, len;

switch (type) {
case ICT_ECHORP:
     len = pip->ip_len - IP_HLEN(pip) - IC_HLEN;
     if (isodd(len))
          pic->ic_data[len] = 0;     /* so cksum works */
     return len;
case ICT_DESTUR:
case ICT_SRCQ:
case ICT_TIMEX:
     pic->ic_mbz = 0;           /* must be 0 */
     break;
case ICT_REDIRECT:
     pic->ic_gw = (IPaddr)pa2;
     break;
case ICT_PARAMP:
     pic->ic_ptr = (u_char) ((unsigned)pa2 & 0xff);
     for (i=0; i<IC_PADLEN; ++i)
          pic->ic_pad[i] = 0;
     break;
case ICT_MASKRP:
     memcpy(pic->ic_data, &pa2, IP_ALEN);
     break;
case ICT_ECHORQ:
     if ((int)pa2 > ECHOMAX(pip))
          pa2 = (void*)ECHOMAX(pip);
     for (i=0; i<(int)pa2; ++i)
          pic->ic_data[i] = i;
     return (int)pa2;
case ICT_MASKRQ:
     memcpy(pic->ic_data, &ip_anyaddr, IP_ALEN);
     return IP_ALEN;
}
return 0;
}
```

对于应答报文，icmp已经在传入请求分组的基础上生成了外发报文，因而此时无需复制数据，然而icsetdata必须计算并返回正确的数据长度。对大多数报文来说，首部已经包含了所有必要的信息，因此数据长度为0，icsetdata填写适当的字段。例如，在ICMP重定向报文中，调用者利用入口参数pa2提供一个指针，指向新网关地址，而icsetdata将其复制到自己的报文中。

对ICMP回送应答报文，icsetdata从传入的请求报文中计算其长度。要做到这一点，它用传入数据报的长度减去IP首部长度和ICMP首部长度。另外，对奇数长度的回送应答报文，icsetdata必须在报文后附加全“0”的一个八位组，以便16位的校验和算法能正确工作。对ICMP回送请求报文，参数pa2指明了数据的长度。

8.12 ICMP重定向报文的生成

有了以上我们所介绍的几个ICMP过程，生成一个ICMP差错报文就变得很简单。例如，过程ipredirect生成ICMP重定向报文。

```
/* ipredirect.c - ipredirect */

#include <conf.h>
#include <kernel.h>
#include <network.h>

struct route *rtget(IPaddr, Bool);

/*------------------------------------------------------------------------
 *  ipredirect  -  send redirects, if needed
 *------------------------------------------------------------------------
 */
void ipredirect(pep, ifn, prt)
struct ep      *pep;
unsigned       ifn;
struct route   *prt;
{
        struct ip *pip = (struct ip *)pep->ep_data;
        struct     route*tprt;
        int        rdtype, isonehop;
        IPaddr     nmask;     /* network part's mask                */

        if (ifn == NI_LOCAL || ifn != prt->rt_ifnum)
             return;
        tprt = rtget(pip->ip_src, RTF_LOCAL);
        if (!tprt)
             return;
        isonehop = tprt->rt_metric == 0;
        rtfree(tprt);
        if (!isonehop)
             return;
        /* got one... */

        nmask(netmask, prt->rt_net);    /* get the default net mask     */
        if (prt->rt_mask == nmask)
             rdtype = ICC_NETRD;
        else
             rdtype = ICC_HOSTRD;
        icmp(ICT_REDIRECT, rdtype, pip->ip_src, pep, (void*)prt->rt_gw);
}
```

ipredirect的三个入口参数分别给出了一个指针，指向含有报文分组的缓冲区；一个接口号，分组经由此接口到达；一个指向新路由的指针。在ipredirect检查并确定该接口并非与本地机器相关的接口，并且新路由指向的接口也并非该传入分组经过的接口之后，它调用rtget来计算路由，路由的目的站就是发送该数据报的机器。

由于协议规定，网关只允许向直接相连的网络中的主机发送 ICMP 重定向（redirect）报文，因此，ipredirect检查通往目的站的路由的长度。如果路由长度值大于0，就意味着该主机不在与网关直接连接的网络中，因而导致 ipredirect 直接返回，不发送任何报文。一旦 ipredirect 找到对方主机在一个直接相连的网络上，它必须检查新路由，以决定这是一条指定主机的路由还是一条指定网络的路由。为了做到这一点，它检查与此路由相关联的子网掩码。如果掩码覆盖掉的不仅仅是网络部分，则 ipredirect 说明这个报文是一个“主机重定向”，否则，它说明此报文为“网络重定向”。

8.13　小结

从概念上来看，ICMP 可被划分为两部分：一部分处理传入的 ICMP 报文，而另一部分生成外发的 ICMP 报文。尽管主机和网关都必须处理输入的报文，但大部分外发的报文仅由网关生成。因此网关上的 ICMP 程序代码通常要比主机上的复杂。

在实际情况中，传入和外发报文之间的许多细节问题和交互作用使ICMP程序代码复杂化。在设计方案中采用了两个主要过程：icmp_in 用来处理传入报文，而 icmp 用来生成外发报文。这两个过程都要调用一些子过程来处理建立缓冲区、设置子网掩码、填写首部及数据字段、计算正确的源地址等细节问题。

深入研究

Postel［RFC 792］描述了 ICMP 协议。Mogul 和 Postel［RFC 950］补充了子网掩码请求和应答报文，而 Braden［RFC 1122］给出了进一步说明。网关需求文档、Baker［RFC 1812］讨论了网关应如何生成和处理 ICMP 报文。

习题

8.1 考虑过程 icsetsrc，在什么情况下循环语句遍历了所有接口后，仍然找不到一个匹配接口？

8.2 在形成一个应答时，ICMP仅仅是将请求分组中的源地址字段和目的地址字段互换吗？为什么？提示：阅读协议规范说明书。

8.3 当主机接收到一个 ICMP 超时（time exceeded）报文后，它应该做哪些工作？

8.4 当主机接收到一个 ICMP 源站抑制（source queneh）报文后，它应该做哪些工作？

8.5 假设一个网关为某个含有子网地址的目的站生成了一个 ICMP“重定向”报文（即，子网掩码延伸超过了地址中的网络部分）。它应该将此“重定向”报文作为“主机重定向”，还是“网络重定向”？为什么？提示：参考［RFC 1812］。

8.6 在示例的代码中，对 ICMP 源站抑制报文是如何响应的？还有其他哪些报文与它的处理方

法一样？

8.7 仔细查看setmask。它处理两种类型的广播地址（全“0”和全“1”）。在协议标准中找出有关必须使用、可以使用或是禁止使用这类广播的相关条文。

第9章 IP：组播处理（IGMP）

9.1 引言

主机和网关使用Internet组管理协议（Internet Group Management Protocal，IGMP）来管理参与组播数据报传送的计算机群。本章详述了组播数据报的选路和IGMP处理过程，从中可看出主机是如何管理有关组播主机群的信息、识别输入组播数据报并发送输出数据报的。本章还描述了主机如何加入或脱离一个组播主机群，响应来自网关的查询并将IP组播地址映射为对应的物理地址。

9.2 维护组播主机群的成员信息

IP使用D类地址传送组播数据报。从理论上讲，一个D类地址定义了一组主机，它们同时接收任何发向该地址的数据报的副本，在协议标准中规定使用主机群（host group）来定义一组与某个特定的组播地址相关联的主机。主机群的成员是动态的，某个主机可以选择任何时候加入或脱离一个主机群。在任何时候，某个主机可以不属于任何主机群，也可以属于多个主机群，而一个主机群中可以没有任何成员，也可以有多个成员。

每个参与IP组播的主机维护着自己的主机群成员记录，没有任何主机或网关知道某个主机群的成员情况。因为所有的记录都由本地保存，主机可以不用经过其他任何主机或网关的允许，自由加入或脱离主机群[①]。由本地保存主机群的成员关系，意味着组播数据报的传送像常规IP数据报的传送那样尽最大努力交付。因为计算机不知道成员关系，所以它在发送组播数据报时无法判定主机群中的每一个成员是否都接收到一份副本。

9.3 主机群表

绝大多数实现IP组播的软件都利用了一张表来保存本机当前所属的主机群的情况。如果一个多地址（multi-homed）主机参与了IP组播，它必须选择以下两种方案之一：该主机可以在多个接口上支持组播，在这种情况下，主机必须为每个接口单独地保存主机群信息。另一种办法是，多地址主机也可以设置一个接口用于组播。如果选择了后者，多地址主机只能在设置好的接口上实现组播。

实现方案中仅用一张全局表保存所有接口的主机群信息。虽然程序代码将数据报的组播限制在一个单独的接口上，但全局表的每个表项中有一个字段，指向此表项对应的接口。文件igmp.h中是对它的说明。

---

① 当主机加入一个主机群时，会通知组播网关，但它不是必须得到允许的，而且主机也不会收到来自组播网关的确认。

```
/* igmp.h - IG-VER, IG-TYP */

#define HG_TSIZE        15              /* host group table size        */

#define IG_VERSION      1               /* RFC 1112 version number      */

#define IG_HLEN         8               /* IGMP header length           */
#define IGT_1HQVERY     1               /* host membership query        */
#define IGT_HREPORT     2               /* host membership report       */

struct igmp {
        unsigned char   ig_vertyp;      /* version and type field       */
        char            ig_unused;      /* not used by IGMP             */
        unsigned short  ig_cksum;       /* compl. of 1's compl.sum      */
        IPaddr          ig_gaddr;       /* host group IP address        */
}

#define IG_VER(pig)   (((pig)->ig_vertyp>>4) & 0xf)
#define IG_TYP(pig)   ((pig)->ig_vertyp & 0xf)

#define IG_NSEND        2     /* # IGMP join messages to send           */
#define IG_DELAY        5     /* delay for resends (1/10 secs)          */

/* Host Group Membership States */

#define HGS_FREE        0     /* unallocated host group table entry     */
#define HGS_DELAYING    1     /* delay timer running for this group     */
#define HGS_1DLE        2     /* in the group but no report pending     */
#define HGS_STATIC      3     /* for 224.0.0.1;no state changes         */

struct hg {
        unsigned char hg_state;       /* HGS-* above                    */
        unsigned char hg_ifnum;       /* interface index for group      */
        IPaddr        hg_ipa;         /* IP multicast address           */
        unsigned long hg_refs;        /* reference count                */
        Bool          hg_ttl;         /* max IP ttl for this group      */
};

/* Host Group Update Process Info. */

extern PROCESS          igmp_update( );
#define IGUSTK          512             /* stack size for update proc.  */
#define IGUPRI          50              /* update process priority      */
#define IGUNAM          "igmp-update"   /* name of update process       */
#define IGUARGC         0               /* count of args to hgupdate    */

struct hginfo {
        Bool     hi_valid;  /* TRUE if hginit( ) has been called       */
        int      hi_mutex;  /* table mutual exclusion                  */
```

```
        int     hi_uport;  /* listen port for delay timer expiress   */
};

extern struct hginfo  HostGroup;

extern IPaddr ig_allhosts;  /* "all hosts" group address (224.0.0.1 ) */
extern IPaddr ig_allDmask;  /* net mask to match all class D addrs.   */
extern struct hg hgtable[];

int hgjoin(unsigned, IPaddr, Bool);
int hgadd(unsigned, IPaddr, Bool);
int hgleave(unsigned, IPaddr);
int hgarpdel(unsigned, IPaddr);
struct hg *hglookup(unsigned,  IPaddr);
int igmp(int, unsigned, IPaddr);
int igmp_settimers(unsigned);
int hgrand(void);
```

数组hgtable实现了主机群表。hgtable中每个表项对应于一个主机群，并包含了由结构hg定义的五个字段。字段hg_state记录该表项的当前状态。当hg_state的值为HGS_FREE，表示该表项当前未使用，表项中的任何其他字段都是无效的。字段hg_ifnum指出表项对应的接口，在字段hg_ipa中的是主机群的IP组播地址，在字段hg_refs中有一个引用计数器，用于指出当前有多少进程正在使用该表项。而字段hg_ttl则定义了表项的最大寿命值。

文件igmp.h还定义了字符常量、报文格式和其他一些组播程序代码中使用的数据结构。例如，为了保证任何时候只有一个进程搜索或修改hgtable中的表项，在程序代码中使用了一个互斥信号量。hginfo结构的hi_mutex字段中包含了信号量标识符。每个过程在使用主机群表之前，必须等待信号量被释放，并且在对该表的操作结束之后释放信号量。

9.4 查找一个主机群

当某个主机改变了它在主机群中的成员身份，或当一个组播数据报到达时，IGMP软件调用过程hglookup，在主机群表中查找该组播地址。

```
/* hglookup.c - hglookup */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

/* ------------------------------------
 * hglookup - get host group entry (if any) for a group
 *     N.B - Assumes HostGroup.hi-mutex *held*
 * ------------------------------------
 */
```

```
struct hg *hglookup(ifnum, ipa)
unsigned    ifnum;
IPaddr  ipa;
{
        struct hg       *phg;
        int         i;

        phg = &hgtable[0];
        for (i=0; i <HG_TSIZE;++i, ++phg) {
                if (phg->hg_stage == HGS_FREE)
                        continue;
                if (ifnum == phg->hg_ifnum && ipa == phg->hg_ipa)
                        return phg;
        }
        return 0;
}
```

hglookup搜索hgtable，直到它找到与给出的组播地址和接口号相匹配的表项，组播地址由参数ipa给出，接口号由参数ifnum给出。hglookup若发现匹配表项，则返回该表项地址，否则返回0。

在我们的hglookup实现方案中采用了线性查找，因为我们假设主机群表中表项的数量很少。然而，这段程序代码被独立地包含在一个单独的过程中，因而能够随时替换成其他查找方案，以便有效地处理较大的主机群表。

9.5 向主机群表中增加一个表项

当某个应用程序第一次参加一个主机群时，必须在hgtable中插入一个新表项。另外，必须设置其网络硬件，使之能识别出主机群使用的硬件组播地址。过程hgadd完成这两项任务。

```
/* hgadd.c - hgadd */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

int hgarpadd (int, IPaddr);

/* ------------------------------------------------------------------------
 * hgadd - add a host group entry for a group
 * ------------------------------------------------------------------------
 */
int hgadd(ifnum, ipa, islocal)
unsigned    ifnum;
IPaddr  ipa;
Bool  islocal;
{
```

```
    struct hg       *phg;
    struct int      start;
    int             i;

    wait(HostGroup.hi_mutex);
    for (i=0; i <HG_TSIZE;++i) {
                if (++strart>=HG_TSIZE)
                        start = 0;
                if (hgtable[start].hg_state == HGS_FREE)
                        break;
}
    phg = &hgtable[start];
    if (phg->hg_state!=HGS_FREE) {
          signal(HostGroup.hi_mutex);
          return SYSERR;        /* table full */
    }
    if (hgarpadd(ifnum, ipa) == SYSERR) {
                signal(HostGroup.hi_mutex);
                return SYSERR;
    }
    phg->hg_ifnum = ifnum;
    phg->hg_refs = 1;
    if (islocal)
                phg->hg_ttl = I;
    else
                phg->hg_ttl = IP_TTL;
    phg->hg-ipa = ipa;
    if (ipa == ig_allhosts)
            phg->hg_state = HGS_STATIC;
    else
            phg->hg_state=HGS_1DLEm;
    signal(HostGroup.hi_mutex);
    return OK;
}
```

过程hgadd首先等待信号量HostGroup.hi_mutex的释放，以保证互斥地访问主机群表。然后，从由静态变量start指出的位置开始，hgadd搜索整个hgtable表。当hgadd找到一个空闲表项或完全检查了整张表之后，搜索结束。如果有一个空闲表项存在，其hg_start字段中的内容是HGS_FREE。如果没有空闲表项，hgadd释放互斥信号量，并向调用者返回SYSERR。在主机群表中存在空闲表项的情况下，hgadd设置网络接口和硬件，初始化hgtable中该表项的各个字段，释放互斥信号量，并向其调用者返回OK。

9.6 为一个组播地址设置网络接口

当主机第一次参加主机群时，必须设置其软件和硬件，以便能处理这个主机群中数据报的发送和接收。为了能够发送组播数据报，必须做两个层次上的修改：在IP选路表中建立一个路由，以及配置网络接口软件，使之能将IP组播地址与适当的硬件地址相绑定。为了能够接收组播数据报，必须设置系统，使之能够识别以该主机群为目的站的数据报。

如果底层硬件支持组播，IP可利用硬件组播设施，发送和接收IP组播数据报。每个主机群被赋予一个惟一的组播硬件地址，用于在该主机群中传送数据报。如果硬件不支持组播，则IP利用硬件广播的形式传送所有组播数据报。为了区分这两种情况，范例中的接口结构包含了ni_mcast字段。如果网络支持硬件组播，ni_mcast中的内容是一个驱动程序的地址，它用来初始化硬件，以接收正确的组播地址。如果网络不支持硬件组播，ni_mcast的内容为0。

过程hgarpadd设置硬件，以便接收以该硬件组播地址为目的站的分组；它还设置地址绑定机制，以便将外发数据报中的IP组播地址映射为适当的硬件地址。这段程序代码在文件hgarpadd.c中。

```
/* hgarpadd.c - hgarpadd */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

/* ------------------------------------
 * hgarpadd - add an ARP table entry for a multicast address
 * ------------------------------------
 */
int hgarpadd(ifnum, ipa)
unsigned    ifnum;
IPaddr  ipa;
{
        struct netif        *pni = &nif[ifnum];
        struct arpentry     *pae, *arpalloc();
        int                 ifdev = nif[ifnum].ni_dev;
        STATWORD            ps;

        disable(ps);
        pae = arpalloc();
        if (pae == 0) {
                restore(ps);
                resurn SYSERR;
        }
        pae->ae_hwtype = pni->ni_hwtype;
        pae->ae_prtype = EPT_IP;
        pae->ae_pni = pni;
        pae->ae_hwlen = pni->ni_hwa.ha_len;
```

```
        pae->ae_prlen = IP_ALEN;
        pae->ae_queue = EMPTY;
        memcpy(pae->ae_pra, &ipa, IP_ALEN);
        if (pni->ni_mcast)
                (pni->ni_mcast)(NI_MADD, ifdev, pae->ae_hwa, ipa);
        else
                memcpy(pae->ae_hwa, pni->ni_hwb.ha_addr, pae->ae_hwlen);
        pae->ae_ttl = ARP_INF;
        pae->ae_state = AS_RESOLVED;
        restore(ps);
        return OK;
}
```

程序代码使用ARP高速缓存中的一个表项来保存一个IP组播地址及其对应的硬件地址的地址绑定。hgarpadd调用arpalloc在ARP高速缓存中分配一个表项，并填写表项中的各个字段。它将由参数ipa传递的IP组播地址复制到字段ae_pra中，并询问网络接口，以得知硬件地址的类型和长度值。为了保证ARP软件不做超时处理，并且不会删除该表项，hgarpadd把字段ae_ttl赋值为ARP_INF，这意味着无限长的生存时间。

hgarpadd测试网络接口结构中的ni_mcast字段，以决定如何计算其硬件地址。如果ni_mcast的值为0，网络不支持硬件组播，hgarpadd将硬件广播地址复制到ARP表项中。如果ni_mcast的值非0，它给出的是一个函数的地址，此函数将IP组播地址转换成对应的硬件组播地址。无论在哪种情况下，hgarpadd都要填写ARP表项中的硬件地址字段，这样ARP程序代码才能为外发的组播数据报找到一个有效的硬件地址。

9.7 IP组播地址和硬件组播地址之间的转换

为硬件组播分配地址的细节取决于采用的网络技术。在以太网中，IGMP标准规定，一个主机群的硬件组播地址的计算方法是将0x01005E000000与其D类地址的低23位相加。过程ethmcast执行将IP组播地址转换为以太网组播地址的操作。

```
/* ethmcast.c - ethmcast */

#include <conf.h>
#include <kernel.h>
#include <network.h>

Eaddr multitemplate = { 0x01, 0x00, 0x5E, 0x00, 0x00, 0x0C};

/* ------------------------------------------------------------------------
 * ethmcast - generate & set an IP multicast hardware address
 * ------------------------------------------------------------------------
 */
int ethmcast (op, dev, hwa, ipa)
int  op;
```

```
int  dev;
Eaddr  hwa;
IPaddr  ipa;
{
        ipa = net2hl(ipa);
        memcpy(hwa, multitemplate, EP_ALEN);

        /* add in low-order 23 bits of IP multicast address */
        hwa[3] = (ipa>>16) & 0x7f;
        hwa[4] = (ipa>>8);
        hwa[5] = ipa;

        switch (op) {
        case NI_MADD:
                return control(dev, EPC_MADD, hwa);
                break;
        case NI_MDEL:
                return control(dev, EPC_MDEL, hwa);
                break;
        }
        return OK;
}
```

ethmcast有四个入口参数，分别定义了一个操作（是增加还是删除此地址）、一个硬件设备地址、一个保存了硬件组播地址的存储区地址以及IP组播地址所处的位置。因为用于数据报组播的基本以太网地址的低位比特的值为0，加法运算显然是不必要的。事实上，ethmcast将变量multitemplate中的基地址复制到参数hwa给出的存储区中，然后从参数ipa中移入IP组播地址的低23位。

当ethmcast形成了一个硬件组播地址之后，调用Xinu系统的control函数，请求设备驱动程序通知以太网硬件。一旦硬件得知新的地址，它将开始接收以该地址为目的地址的以太网分组。

9.8 从主机群表中删除一个组播地址

当主机脱离一个主机群时，它调用过程hgarpdel，从ARP高速缓存中删除一个表项，并通知硬件。

```
/* hgarpdel.c - hgarpdel */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

struct arpentry *arpfind(u_char*, u_short, struct netif*);
```

```
/* ------------------------------------------------------------------------
 * hgarpdel - remove an ARP table entry for a multicast address
 * ------------------------------------------------------------------------
 */
int hgarpdel(ifnum, ipa)
unsigned    ifnum;
IPaddr      ipa;
{
        struct netif        *pni = &nif[ifnum];
        int                 ifdev = nif[ifnum].ni_dev;
        struct arpentry     *pae,
        STATWORD            ps;

         disable(ps);
         if (pae = arpfind((u_char *)&ipa, EPT_IP, pni))
                    pae->ac_state = AS_FREE;
         if (pni->ni_mcast)
                    (pni->ni_mcast)(NI_MDEL, ifdev, pae->ae_hwa, ipa);
         restore(ps);
         return OK;
}
```

hgarpdel执行起来直截了当。它调用arpfind在ARP高速缓存中查找给定的IP地址所对应的表项的位置，并将该表项的状态变为AS_FREE。hgarpdel还要检查接口结构中的ni_mcast字段，以得知网络硬件是否支持数据报的组播。如果支持，那么hgarpdel调用设备驱动软件功能，通知硬件，使它不再接收以该主机群硬件组播地址为目的地址的传入分组。

9.9 加入一个主机群

应用程序通过调用函数hgjoin发出请求，使它的主机加入一个主机群。hgjoin设置主机，使之能够发送和接收以该主机群地址为地址的组播数据报，并通知网络中的其他机器，它已加入了这个主机群，这段程序代码在文件hgjoin.c中。

```
/* hgjoin.c - hgjoin */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <sleep.h>
#include <igmp.h>

/* ------------------------------------------------------------------------
 * hgjoin - handle application request to join a host group
 * ------------------------------------------------------------------------
 */
```

```
int hgjoin(ifnum, ipa, islocal)
unsigned    ifnum;
IPaddr  ipa;
Bool    islocal;
{
        struct hg  *phg;
        int        i;

        if (!IP_CLASSD(ipa))
                return SYSERR;
        /* restrict multicast in multi_homed host to primary interface */
        if (ifnum != NI_PRIMARY)
                return SYSERR;
        wait(HostGroup.hi_mutex);
        if (phg = hglookup(ifnum, ipa)) {
                phg->hg_refs++;
                signal(HostGroup.hi_mutex);
                return OK;     /* already in it */
        }
        signal(HostGroup.hi_mutex);
        /* add to host group and routing tables */
        if (hgadd(ifnum, ipa, islocal) == SYSERR)
                return SYSERR;
        rtadd(ipa, ip_maskall, ipa, 0, NI_LOCAL, RT_INF);
        /*
         * advertise membership to multicast router(s); don't advertise
         * 224.0.0.1 (all multicast hosts) membership.
         */
         if (ipa != ig_allhosts)
                for (i=0; i<IG_NSEND; ++i){
                        igmp(IGT_HREPORT, ifnum, ipa);
                        sleep10(IG_DELAY);
                }
        return OK;
}
```

hgjoin首先检查参数ipa，以证实它包含的是一个D类地址，如果不是，返回SYSERR。然后hgjoin证实主机群表中并没有这个地址。要做到这一点，它必须先取得单独使用主机群表的权利，并调用hglookup来搜索表中是否有给定的IP地址。如果hglookup在表中找到了该地址，hgjoin将此表项的引用计数值加1，并释放互斥信号量，返回调用者。

如果地址ipa有效，而主机群表中却没有这个地址，hgjoin对主机进行设置，使之加入该主机群。要做到这一点，hgjoin首先调用hgadd向主机群表中增加这个新地址，并在ARP高速缓存中为该地址插入一个永久性表项。然后通知硬件，它已可以接受以相应的硬件组播地址为目的地址的分组。如果hgadd成功返回，hgjoin调用rtadd向IP路由表中增加一个永久性路由。IP路由表在处理该路由的传入组播数据报时与处理广播数据报一样，任何以组播地址ipa为目的站的传入数据报都将

被送到本地接口上。

9.10　维持与一个组播路由器的联系

为了消除网络中不必要的组播通信量，主机和组播路由器应用Internet组管理协议（IGMP）。其基本工作原理是，组播路由器定期发出IGMP询问，请求所有参与IP组播的主机向其汇报它们各自所属的一组主机群的情况。主机为它所属的每个主机群发送一份IGMP报告报文。如果几个循环之后，某个主机群中还没有任何主机成员向路由器报告，组播路由器就停止向这个主机群发送数据报。

协议标准规定，当主机第一次加入某个主机群时，它应当在该主机群中发送一份通告报文。hgjoin 的最后几行程序代码实现了这份报文的发送。为了保证报文不会丢失，程序发送该报文 IG_NSEND 次，每次时间间隔为 IG_DELAY 个 0.1 秒。

过程 igmp 通知并发送一个 IGMP 报文。

```
/* igmp.c - igmp */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

/* ------------------------------------
 * igmp - send IGMP request/responses
 * ------------------------------------
 */
int
igmp(int typ, unsigned ifnum, IPaddr hga)
{
        struct   ep     *pep;
        struct   ip     *pip;
        struct   igmp   *pig;

        pep = (struct ep *)getbuf(Net.netpool);
        if (pep == (struct ep *)SYSERR)
                return SYSERR;
        pip = (struct ip *)pep->ep_data;
        pig = (struct igmp *) pip->ip_data;
        pig->ig_vertyp = (IG_VERSION<<4) | (typ & 0xf);
        pig->ig_unused = 0;
        pig->ig_cksum = 0;
        pig->ig_gaddr = hga;
        pig->ig_cksum = cksum((WORD *)pig, IG_HLEN>>1);

        ipsend(hga, pep, IG_HLEN, IPT_IGMP, IPP_INCTL, 1);
```

① 对结构 igmp 的说明见第 9.3 节中文件 igmp.h。

```
        return OK;
}

/* special IGMP-relevant address & mask */

#if     BYTE_ORDER == LITTLE_ENDIAN
IPaddr  ig_allhosts = 0x010000E0;           /* 224.0.0.1 */
IPaddr  ig_allDmask = 0xF0;                 /* 240.0.0.0 */
#else   /* BYTE_ORDER */
IPaddr  ig_allhosts = 0xE0000001;           /* 224.0.0.1 */
IPaddr  ig_allDmask = 0xF0000000;           /* 240.0.0.0 */
#endif  /* BYTE_ORDER */
```

过程igmp为一个分组分配网络缓冲区，然后填入IGMP报文。结构igmp定义了IGMP报文格式[1]。字段ig_vertyp的内容是协议版本号和报文类型，调用者通过参数typ指定IGMP报文类型。在报告报文中，字段ig_gaddr中包含的是主机群地址，它的值由调用者通过参数hga传递给igmp。

组播路由器向地址224.0.0.1发送查询报文，224.0.0.1代表了所有主机群。因为所有组播路由器接收所有组播分组，所以主机既不必知道路由器的地址，也不必向每个路由器直接发送响应。事实上，主机使用主机群的组播地址为该主机群发送响应。这样，每个参与该主机群的主机都会接收到所有成员的报告。

为了避免在IGMP查询之后，所有的主机一起发送响应，而导致响应分组的突增现象，协议规定，主机在发送报告时，每两个报告之间必须间隔1～10秒的一个随机时间量。另外，当有一个主机为某个主机群发送了一个报告之后，其他所有主机取消它们在该主机群上的计时，直至下一次查询到达。

9.11 IGMP成员关系报告的实现

当一个IGMP查询到达时，主机必须为每个主机群设置一个随机定时器。过程igmp_settimers使用通用定时机制完成这项任务，通用定时机制在第14章中描述。

```
/* igmp_settimers.c - igmp_settimers */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

/* ------------------------------------------------------------------------
 * igmp_settimers - generate timer events to send IGMP reports
 * ------------------------------------------------------------------------
 */
int igmp_settimers(ifnum)
unsigned        ifnum;
```

① 对结构igmp的说明见第9.3节中文件igmp.h。

```
{
        int             i;

        wait(HostGroup.hi_mutex);
        for (i=0;i<HG_TSIZE;++i) {
                struct hg       *phg = &hgtable[i];
                if (phg->hg_state ! = HGS_IDLE || phg->hg_ifnum != ifnum)
                        continue;
                phg->hg_state = HGS_DELAYING;
                tmset(HostGroup.hi_uport, HG_TSIZE, phg, hgrand());
                }
                signal(HostGroup.hi_mutex);
                return OK;
}
```

igmp_settimers 遍历主机群表并检查每个表项。如果字段 hg_state 的内容为 HGS_IDLE，该表项代表一个没有安排定时事件的有效主机群。对每个这样的表项来说，igmp_settimers 将其状态改为 HGS_DELAYING，并调用 tmset 为该表项创建定时事件。

tmset 的第一个入口参数给出了一个 Xinu 端口，当定时器超时后，报文将通过该端口发送出去。第二个参数给出了该端口的最大尺寸。第三个参数的内容是将要发送出去的报文，而第四个参数给出了时延，以 0.01 秒为单位。igmp_settimers 向主机群表项传递一个指针，作为将要被发送的报文。

9.12　计算随机时延

因为协议标准规定，为每个主机群发送的报告必须延迟一段随机时间，所以 igmp_settimers 调用函数 hgrand 计算这个时延，这段程序代码包含在 hgrand.c 文件中。

```
/* hgrand.c - hgrand */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

int         modulus = I009;             /* 0~10 secs in 1/100'th secs   */
int          offset = 523;              /* additive constant            */
int          hgseed;                    /* initialized in hginit ()     */

/* ------------------------------------------------------------------------
 * hgrand - return "random" delay between 0 & 10 secs (in 1/100 secs)
 * ------------------------------------------------------------------------
 */
int hgrand (void)
{
```

```
        int rv;

        rv = ((modulus+1) * hgseed + offset) % modulus;
        if (rv<0)
                rv += modulus; /* return only positive values */
        hgseed = rv;
        return rv;
}
```

因为定时机制要求使用以0.01秒为单位的整数值来给出时延值，所以hgrand使用整数运算。像大多数伪随机数生成程序一样，hgrand每次调用时返回到一个0到模数（modulus）之间的整数值，这样产生的数值序列看上去是随机的。我们选择的模数为1009，也就是大约等于10秒的时延。

如果网络上的所有主机使用相同的伪随机数生成算法，那么它们会生成相同的时延序列。由于相同的时延将导致冲突并生成不必要的通信量，IGMP标准规定，使用伪随机数生成程序的主机必须使用一个初始种值（seed value），以保证生成惟一的时延序列。其结果是，两个主机不再使用完全相同的时延序列，哪怕它们具有相同的硬件并运行相同的程序代码。范例代码根据主机群的IP地址初始化一个种值，保存在全局变量hgseed中，以保证时延序列的惟一性。

9.13 发送IGMP报告的进程

如果定时器超时，定时机制向某个操作系统端口发送一个报文，在该端口上必须有一个进程正在等待着接收报文：进程igmp_update接收IGMP定时事件。

```
/* igmp_update.c - igmp_update */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ports.h>
#include <igmp.h>

/* ------------------------------------------------------------------------
 * igmp_update - send (delayed) IGMP host group updates
 * ------------------------------------------------------------------------
 */
PROCESS igmp_update(void)
{
        struct hg       *phg;

        HostGroup.hi_uport = pcreate(HG_TSIZE);
        while (1) {
                phg = (struct hg *)preceive(HostGroup.hi_uport);
                wait(HostGroup.hi_mutex);
```

```
                if (phg->hg_state == HGS_DELAYING) {
                        phg->hg_state = HGS_IDLE;
                        igmp(IGT_HREPORT, phg->hg_ifnum, phg->hg_ipa);
                }
                signal(HostGroup.hi_mutex);
        }
}
```

igmp_update 先创建一个端口，然后进入无限循环。在每次循环中，它调用 preceive 阻塞该端口，直到有一个报文到达。一旦有一个报文到达，igmp_update 等待互斥信号量的释放，以获得单独访问主机群表的权力，然后可以发送报告报文，最后释放互斥信号量。

对 preceive 的调用返回一个指针，指向一个在主机群表中定时器已超时的表项，igmp_update 应当为该表项发送一份 IGMP 报告。因为 igmp_update 总是作为进程运行的，所以时间设定和内容交换可能会延长它的执行时间。实际上，在这段延迟期间，可能有数据报到达，也可能其他进程运行。因此，该表项的状态可能会在定时器超时和 igmp_update 执行的瞬间被改变。为了保证确实只发送一个报告，igmp_update 检查字段 hg_state。如果该表项的状态为 HGS_DELAYING，igmp_update 调用 igmp 发送一个报告，并将其状态改为 HGS_IDLE。如果状态已经被改变，igmp_update 不再发送报告。

9.14　处理输入的 IGMP 报文

当 IGMP 报文到达时，IP 调用过程 igmp_in 来处理它。

```
/* igmp_in.c - igmp_in */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

/* ------------------------------------------------------------------------
 * igmp_in - handle IGMP packet coming in from the network
 * ------------------------------------------------------------------------
 */
int igmp_in(pni, pep)
struct netif    *pni;
struct ep       *pep;
{
        struct ip       *pip;
        struct igmp     *pig;
        struct hg       *phg;
        int             ifnum = pni - &nif[0];
        int             i, len;

        pip = (struct ip *)pep->ep_data;
```

```
        pip = (struct igmp *) pip->ip_data;

        len = pip->ip_len - IP_HLEN(pip);
        if (len !=IG_HLEN||IG_VER(pig) !=IG_VERSION||
            cksum((WORD *)pig, len>>1)) {
                freebuf(pep);
                return SYSERR;
        }
        switch(IG_TYP(pig)) {
        case IGT_HQUERY:
                igmp_settimers(NI-PRIMARY);
                break;
        case IGT_HREPORT:
                wait (HostGroup.hi_mutex);
                if ((phg = hglookup(NI_PRIMARY, pig->ig_gaddr)) &&
                    phg->hg_state == HGS_DELAYING) {
                        tmclear(HostGroup.hi_uport, phg);
                        phg->hg_state = HGS_IDLE;
                }
                signal(HostGroup.hi_mutex);
                break;
        default:
                break;
        }
        freebuf(pep);
        return OK;
}
```

igmp_in首先检查传入报文的首部，计算报文的实际长度，并将该长度值与保存在首部中的长度值相比较。然后igmp_in查看IGMP首部中的版本号，以保证它与软件的版本号相符，并验证首部中的校验和。如果有任何一项检查通不过，igmp_in就丢弃该报文。

一旦igmp_in接受了一个报文，它应用宏IG_TYP提取报文的类型。如果该报文是一个查询，则igmp_in调用igmp_settimers，启动主机群表中每个表项的定时器。

如果传入的报文是一个报告，则意味着另一个主机已经对某次查询做了应答。igmp_in调用hglookup来决定在它的主机群表中有没有任何对应于该主机群的表项。如果有，igmp_in调用tmclear清除正在等待的定时事件。不管是什么情况，在igmp_in处理完一个报文后，要调用freebuf释放缓冲区。

9.15 脱离主机群

从理论上讲，脱离一个主机群包括了从主机群表中删除该表项，从IP路由表中删除组播路由并设置网络硬件，使之忽略以该主机群硬件组播地址为地址的分组。然而在实际情况下，有几个细节使脱离主机群显得稍有些复杂。例如，在所有使用主机群表中某一表项的进程全部结束之前，不能删除该表项。

过程 hgleave 处理这些细节，当一个应用程序决定脱离某个主机群时，它调用过程 hgleave。该程序代码段在 hgleave.c 文件中。

```
/* hgleave.c - hgleave */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

/* ------------------------------------------------------------------------
 * hgleave - handle application request to leave a host group
 * ------------------------------------------------------------------------
 */
int hgleave(ifnum, ipa)
unsigned ifnum;
IPaddr  ipa;
{
        struct  hg          *phg,

        if (!IP_CLASSD(ipa))
                return SYSERR;
        wait (HostGroup.hi_mutex);
        if (!(phg = hglookup(ifnum, ipa)) || --(phg->hg_refs)) {
                signal (HostGroup.hi_mutex);
                return OK;
        }
        /* else, it exists & last reference */
        rtdel(ipa, ip_maskall);
        hgarpdel(ifnum, ipa);
        if (phg->hg_state == HGS_DELAYING)
                tmclear(HostGroup.hi_uport, phg);
        phg->hg_state = HGS_FREE;
        signal(HostGroup.hi_mutex);
        return OK;
}
```

过程 hgleave 首先检查它的参数，以保证调用者传递的是一个有效的 D 类地址。它还调用 hglookup 来验证当前在主机群表中存在这个给出的地址。如果一切正常，它递减该表项的引用计数值 hg_refs。如果引用计数值仍为正数，那么 hgleave 返回调用者，因为显然还有其他进程正在使用该表项。

当最后一个使用该表项的进程将引用计数值减 1 后，计数值为 0，此时该表项可以被删除。要删除一个表项，hgleave 调用 rtdel 从选路表中删除该路由，然后调用 hgarpdel 删除 ARP 高速缓存中的对应表项，并使网络硬件停止接收向该主机群发送的分组。

在返回调用者之前，hgleave 检查字段 hg_state，以确定该表项是否还有定时事件存在。如果有，hgleave 在将该表项置为空闲状态之前，调用 tmclear 删除定时事件。

9.16 IGMP数据结构的初始化

系统启动时调用过程hginit。hginit创建一个进程，处理组播更新，初始化主机群表，并加入allhosts组播群。

```
/* hginit.c - hginit */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <sleep.h>
#include <network.h>
#include <igmp.h>

extern int      hgseed;
struct hginfo   HostGroup;
struct hg       hgtable[HG_TSIZE];

/* ------------------------------------------------------------------------
 * hginit - initialize the host group table
 * ------------------------------------------------------------------------
 */
void hginit(void)
{
        int i;

        HostGroup.hi_mutex = screate(0);
        HostGroup.hi_valid = TRUE;
        resume(create(igmp_update, IGUSTK, IGUPRI, IGUNAM, IGUARGC));
        for (i=0; i<HG_TSIZE; ++i)
                hgtable[i].hg_state = HGS_FREE;
        hgseed = nif[NI_PRIMARY].ni_ip;
        signal (HostGroup.hi_mutex);
        rtadd(ig_allhosts, ig_allDmask, ig_allhosts, 0, NI_PRIMARY,
                RT_INF);
        hgjoin(NI_PRIMARY, ig_allhosts, TRUE);
}
```

hginit生成互斥信号量，在它启动更新进程之前，互斥信号量的初始值为0，以保证在hginit把每个表项的hg_state字段赋值为HGS_FREE之前，没有任何其他进程能够访问该主机群表。hginit还要将全局变量hgseed赋值为该主机主接口的IP地址值。一旦数据结构初始化完成，hginit释放互斥信号量，以允许其他进程访问主机群表。

hginit调用rtadd在IP选路表中为全主机组播群增加一个路由。这个路由使所有发往该地址的外发数据报经过主接口（primary interface）。调用时指定了其寿命为RT_INF，这样该表项就是永久

性的。

作为初始化的最后一步，hginit 调用 hgjoin 将该主机加入这个全主机群。一旦主机加入了全主机群，它就开始接收 IGMP 查询。

9.17　小结

主机和网关使用IP组播向全体主机中的一个子网发送数据报。这组主机通过给定的IP组播地址进行通信，它们被称为主机群。IGMP 协议允许主机在任何时候加入或脱离主机群。

为了避免不必要的通信量，组播路由器定期发送一个IGMP查询报文，以判定主机群是否在每个网络中还有成员存在。当一个查询报文到达时，主机为它所属的每个主机群设置一个随机定时器。当这个定时器超时，主机发送IGMP报告，通知网关在本地网络中至少还有一个属于该主机群的主机成员。某个主机群中的全体主机都会接收到该主机群中的报告报文的副本，如果该主机群中已有一个主机发送了报告，那么其他主机取消各自的定时事件。

深入研究

Deering［RFC 1112］描述了 IGMP 协议并给出了报文格式，另外，它提供了实现要求及设计的合理性。Braden［RFC 1122］进行了进一步的描述。

习题

9.1 如果应用程序在主机加入主机群之前就生成了一个发给该主机群的外发数据报，会发生什么情况？

9.2 假设数据丢失的概率为 p_l，且 $0 \leq p_l \leq 1$。试推导一个公式，给出在声明所有主机已完全脱离某个主机群之前，一个组播网关至少应循环等待的次数。

9.3 考虑使用常规（单播）IP 地址处理方法作为两个组播网关之间通信的机制。当一个网关需要向另一个网关发送组播数据报时，它将组播数据封装在一个单播数据报中，然后传送出去。这种方法的优点是什么？缺点是什么？

9.4 当主机连接到一个利用 ATM 技术的网络上时，IGMP 还有用处吗？为什么？

9.5 查阅 RFC 1122，主机怎样检测失效的网关，如果检测到，如何处置？

9.6 范例代码有一个条件是，应用程序必须首先知道加入哪个组播主机群。IGMP协议是否包含了允许应用程序获得关于主机群信息的报文？

第10章 UDP：用户数据报

10.1 引言

用户数据报协议（User Datagram Protocol）提供应用程序之间的无连接通信。它允许某个机器上的一个程序向其他机器上的一个或多个程序发送数据报并接收响应信息。本章讨论UDP的实现过程，主要集中讲述UDP如何利用协议端口号来识别通信的端点。它讨论了两种解决协议端口号绑定（binding）问题的方法，并给出其中一种方法的实现细节。最后，本章描述了UDP伪首部（pseudo-header），并详述了计算UDP校验和的过程是如何使用伪首部的。

10.2 UDP端口和多路分解处理

理论上，使用UDP的通信非常简单。协议标准规定应用程序使用一个称为协议端口号（protocol port number）的抽象模型来识别通信的端点。当机器A中的应用程序与机器B中的一个应用程序通信时，这两个应用程序都必须从自己的本地操作系统中获取一个UDP端口号。两端在通信时都必须使用协议端口号。使用协议端口号（而不是系统提供的标识符，如进程、任务或作业标识符）的优点是保持了协议与特定系统之间的独立性，并允许不同类型的计算机系统应用程序之间互相通信。

虽然UDP协议端口号的设计思想看起来直截了当，但在实现时有两种基本的方法。两种方法都与协议标准一致，但它们为应用程序提供的接口稍有不同。下一节将描述客户和服务器如何使用UDP，并说明了两种方法是如何互相兼容的。

10.2.1 成对通信使用的端口

如图10.1（a）所示，有些应用程序使用UDP来进行成对通信。要做到这一点，两个应用程序各自从本地操作系统中获取UDP端口号，并且他们利用这对端口号交换UDP报文。在这种情况下，应用程序和协议软件之间的理想的接口，是将具体的地址操作从发送和接收数据报的操作中分离出来。这样，接口允许一个应用程序一次性指定本地的和远程的协议端口号，然后多次发送和接收数据报。当然，在指定另一个机器上的协议端口号时，应用程序必须同时指定该机器的IP地址。一旦协议端口号被指定，应用程序就能发送和接收任意多的数据报。

10.2.2 多对一通信使用的端口

绝大多数应用程序使用客户-服务器模式的交互形式，如图10.1（b）所示。单一的服务器应用程序接收来自多个客户的UDP报文。当服务器启动时，它不会指定另一个机器上的一个IP地址或UDP端口，因为它必须允许任意一个机器向其发送报文。事实上，它仅仅指定一个本地UDP端口号。由客户送往服务器的每个报文必须给出客户的UDP端口号及服务器的UDP端口号。服务器从输入的UDP数据报中提取源端口号，并在它发送应答报文时将该端口号作为目的端口号。当然服务器还必须在UDP数据报到达时获取客户的IP地址，这样在发送应答报文时，它也能够指定IP地址。

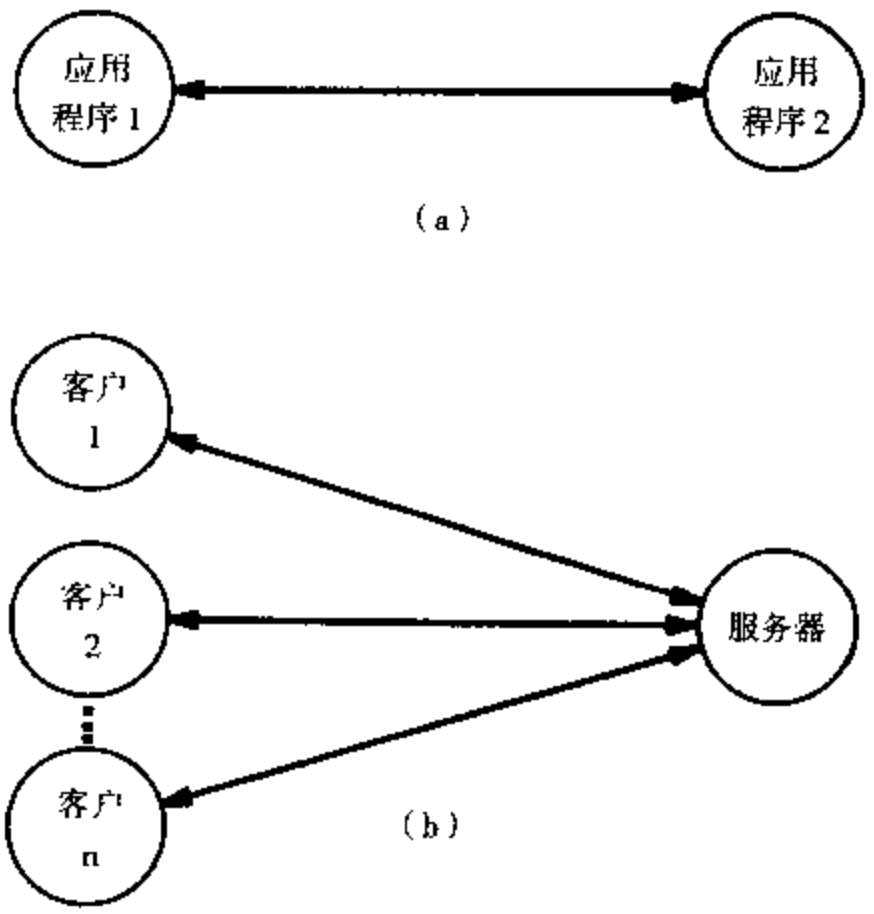


图 10.1　应用 UDP 的程序之间交互作用的两种形式。(a) 客户及其他程序之间使用成对交互形式；(b) 服务器使用多对一交互形式，此时一个应用程序可能向多个目的站发送数据报

因为服务器与多个客户通信，所以不能设置一个永久的IP目的地址或永久的UDP协议端口号。事实上，用于多对一通信的接口必须允许服务器在每次发送数据报时给出有关目的站的信息。这样，与成对通信使用的理想型接口不同，服务器使用的理想型接口并没有将地址标识和数据报的传送分开。

10.2.3　操作模式

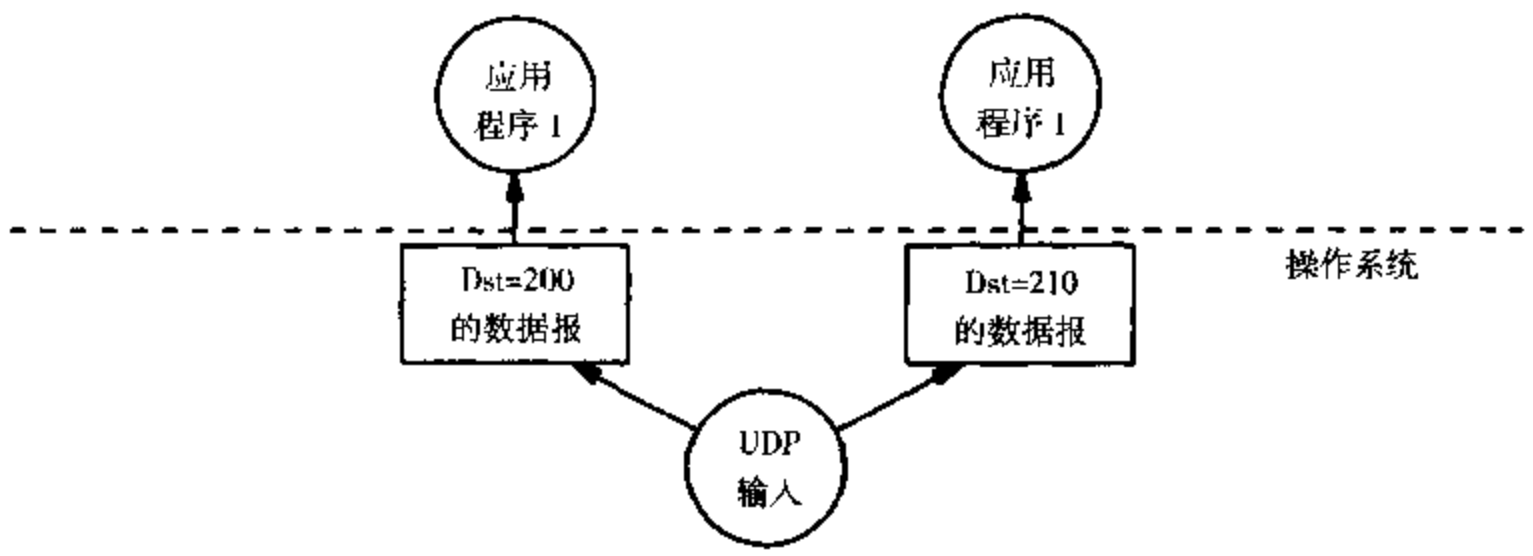
为了与成对通信和多对一通信方式都兼容，大多数UDP接口利用参数来控制交互的模式（mode）。一种典型的采用成对交互作用的模式是客户。它允许应用程序一次性指定本地和对方的协议端口号，然后发送和接收UDP数据报，并且无需每次都给出端口号。另一种模式包含了服务器。它允许服务器仅仅指定本地端口，然后就可接收来自任意客户的数据报。这种系统可能要求一个应用程序明确地说明交互模式，也可能从应用程序给出的端口绑定中推断出其模式。

10.2.4　多路分解处理中的细节问题

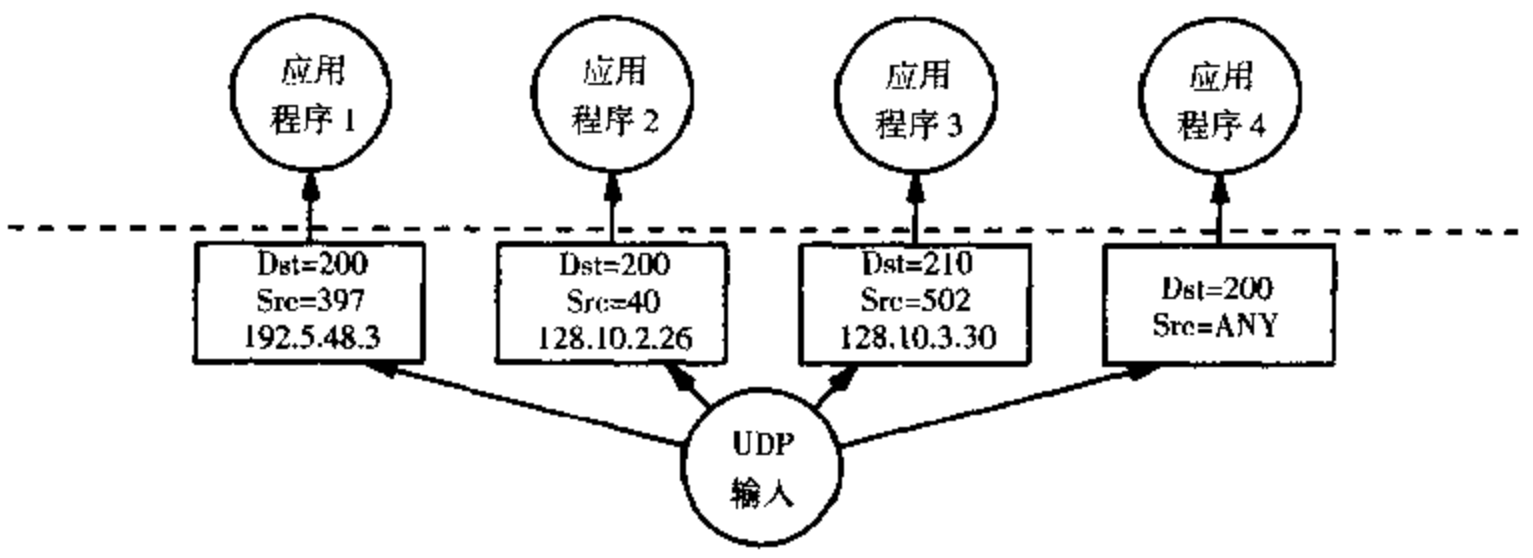
除了要考虑交互模式之外，UDP实现方案还要提供对协议端口多路分解处理的解释。这里有两种可能性：

- 多路分解时仅使用目的站的协议端口号。
- 多路分解时同时使用源站和目的站的协议端口号。

不同的选择会对应用程序与协议软件之间的交互作用产生微妙的影响。要理解其中详情，考虑如图 10.2 所示的两种多路分解形式。



(a)



(b)

图10.2 两种方式的UDP多路分解处理：(a)仅使用目的站端口；(b)使用端口对(源站和目的站)。在(a)方式下，应用程序接收指向特定目的端口的所有数据报。而在(b)方式下，它仅接收来自特定源站的数据报

在第一种多路分解方式下，系统将发向特定目的站协议端口的所有数据报都送到同一队列中。在第二种多路分解方式下，系统在为数据报多路分解时要利用源地址(源站协议端口号和源IP地址)。因此在第二种方式下，每个队列中的所有数据报都来自特定的某个源站。

这两种方式各有利弊。例如，在第一种方式下，服务器发挥的作用很小，因为应用程序接收了所有发往给定协议端口号的数据报，与它们的生成源无关。然而，由于系统对多个源站不加区分，所以系统不能过滤掉有地址错误的数据报。只要是一个以该端口为目的站的数据报到达，程序就会接收，哪怕它被错误地发送了。在第二种方式下，客户发挥的作用小，因为一个给定的应用程序有权选择接收另一应用程序发送来的数据报。然而，如果一个应用程序需要与两个远程应用程序同时通信，它就必须分配两个队列，每个远程应用程序对应一个队列。再者，系统可能需要提供附加的机制，以允许一个程序在两个队列上等待I/O端口的事件[①]。

虽然看起来困难，但还是有可能同时使用两种方式的多路分解处理，以同时适用于客户和服务器。在第一种方式下，仅与一个远程应用程序通信的客户必须选择一个未被其他任何本地程序使用的本地协议端口号。在第二种方式下，服务器必须使用一个通配符[②]，如图10.2所示。被标志为ANY

① 有的系统使用select允许一个应用程序等待一组I/O端口描述符中任何一个I/O活动。

② 增加一个通配符机制使第二种方案与第一种方案在功能上等效。

的源站代表一个与任意源匹配的通配符（任意源IP地址和源协议端口号）。在某一特定时刻，系统最多允许为某个目的端口使用一个通配符。当数据报到达时，该实现方案在检查通配符之前，先查看其源站和目的站是否与给出的源站-目的站对相匹配。因此，在例子中，如果一个目的端口号为200，源端口号为397，而源IP地址为192.5.48.3的数据报到达，系统将其放入应用程序1对应的队列中。类似地，系统将目的端口号为200，源端口号为40，而源IP地址为128.10.2.26的数据报放入应用程序2对应的队列中。系统使用通配符机制与其他目的端口号为200的数据报相匹配，并将它们放入应用程序4对应的队列中。

10.3　UDP的输入处理

我们的范例程序使用的多路分解方式是为仅有目的协议端口号的输入数据报选择一个队列。我们选用这种方式是因为它既能有效地进行多路分解处理，又允许应用程序与多个远程站点同时通信。在介绍了UDP使用的数据结构的定义之后，我们将叙述软件如何处理输入的数据报，以及如何发送外发数据报。

10.3.1　UDP数据结构的说明

在文件udp.h中说明的结构udp定义了UDP数据报的格式。除了16位的源站和目的站协议端口号外，UDP首部还包含了16位的数据报长度字段和16位的校验和。

```
/* udp.h */

/* User Datagram Protocol (UDP) constants and formats */

#define     U_HLEN     8          /* UDP header length in bytes        */

/* maximum data in UDP packet    */
#define     U_MAXLEN   (IP_MAXLEN-(IP_MINHLEN<<2)-U_HLEN)

struct udp {                      /* message format of DARPA UDP  */
      unsigned short  u_src;      /* source UDP port number       */
      unsigned short  u_dst;      /* destination UDP port number  */
      unsigned short  u_len;      /* length of UDP data           */
      unsigned short  u_cksum;    /* UDP checksum (0 => none)     */
      char u_data[U_MAXLEN];      /* data in UDP message          */
};

/* UDP constants */

#define     ULPORT          2050 /* initial UDP local "port" number   */

/* assigned UDP port numbers */

#define UP_ECHO        7     /* echo server                          */
#define UP_DISCARD     9     /* discard packet                       */
#define UP_USERS       11    /* users server                         */
```

```
#define UP_DAYTIME      13    /* day and time server                    */
#define UP_QOTD         17    /* quote of the day server                */
#define UP_CHARGEN      19    /* character generator                    */
#define UP_TIME         37    /* time server                            */
#define UP_WHOIS        43    /* who is server (user information)       */
#define UP_DNAME        53    /* domain name server                     */
#define UP_TFTP         69    /* trivial file transfer protocol server  */
#define UP_RWHO         513   /* remote who server (ruptime)            */
#define UP_RIP          520   /* route information exchange (RIP)       */

#ifndef Ndg
#define UPPS            1     /* number of xinu ports used to           */
#else                         /*  demultiplex udp datagrams             */
#define UPPS            Ndg
#endif
#define UPPLEN          50    /* size of a demux queue                  */

/* mapping of external network UDP "port" to internal Xinu port */

struct upq  {                           /* UDP demultiplexing info      */
      Bool            up_valid;         /* is this entry in use?        */
      unsigned short  up_port;          /* local UDP port number        */
      int             up_pid;           /* port for waiting reader      */
      int             up_xport;         /* corresponding Xinu port on   */
};                                      /*  which incoming pac. queued  */

extern struct  upq    upqs[];
extern int            udpmutex;         /* for UDP port searching mutex */

int udpsend(IPaddr, u_short, u_short, struct ep *, unsigned, Bool);
unsigned short udpcksum(struct ep *, int);
void udpnet2h(struct udp *), udph2net(struct udp *);
```

除UDP数据报格式的说明之外，udp.h中还包含了一些符号常量，用于为最常用的UDP协议端口号赋值。例如，TFTP服务器总是在第69号端口上操作，而RIP使用的端口号为520。

10.3.2 传入数据报队列的说明

UDP软件将存储传入数据报的数据结构划分为两部分：第一部分由传入数据报的队列组成，而第二部分包含了UDP用来选择队列的映像信息。前者是UDP和用于提取传入数据报的应用程序之间接口的一部分。后者是操作系统的一部分——UDP软件利用它来选择一个队列，但应用程序不能访问此队列。文件dgram.h中包含了对应用程序使用的队列的定义。

```
/* dgram.h */

/* datagram pseudo-device control block */
```

```
struct dgblk    {                   /* datagram device control block     */
        int     dg_dnum;            /* device number of this device      */
        int     dg_state;           /* whether this device allocated     */
        IPaddr  df_lip;             /* whether this device allocated     */
        u_short dg_lport;           /* local datagram port number        */
        IPaddr  dg_fip;             /* foreign machine IP address        */
        u_short dg_fport;           /* foreign datagram port number      */
        int     dg_xport;           /* incoming packet queue             */
        int     dg_upq;             /* index of our upq entry            */
        int     dg_mode;            /* mode of this interface            */
};

/* datagram psuedo-device state constants */

#define DGS_FREE        0           /* this device is available          */
#define DGS_INUSE       1           /* this device is in use             */

#define DG_TIME         30          /* read timeout (tenths of sec)      */

/* constants for dg pseudo-device control functions */

#define DG_SETMODE      1           /* set mode of device                */
#define DG_CLEAR        2           /* clear all waiting datagrams       */

/* constants for dg pseudo-device mode bits */

#define DG_NMODE        001         /* normal (datagram) mode            */
#define DG_DMODE        002         /* data-only mode                    */
#define DG_TMODE        004         /* timeout all reads                 */
#define DG_CMODE        010         /* generate checksums (default)      */

/* structure of xinugram as dg interface delivers it to user */

struct xgram    {                   /* Xinu datagram (not UDP)           */
        IPaddr  xg_fip;             /* foreign host IP address           */
        unsigned short  xg_fport;   /* foreign UDP port number           */
        unsigned short  xg_lport;   /* local UDP port number             */
        u_char  xg_data[U_MAXLEN];  /* maximum data to/from UDP          */
};

#define XGHLEN   8     /* error in ( (sizeof(struct xgram)) - U_MAXLEN)*/

/* constants for port specifications on UDP open call */

#define ANYFPORT 0                  /* accept any foreign UDP port       */
#define ANYLPORT 0                  /* assign a fresh local port num     */

extern struct   dgblk   dgtab[Ndg];
extern int      dgmutex;
```

```
int     dgalloc(void);
```

虽然这个文件中包含的许多细节超过了本章讲述的范围，但有两个定义与我们此刻探讨的内容密切相关。用来存储传入数据报的基本数据结构由一个数组dgtab组成，数组中的元素的类型是dgblk。可以把dgtab看作是一组队列，在dgtab中每个正在使用中的本地UDP协议端口对应一个活动的表项。字段dg_lport指定本地UDP协议端口号，而字段dg_xport定义了以该端口为目的站的数据报所对应的队列。字段dg_state说明了该表项是正在使用中（DGS_INUSE）还是未分配（DGS_FREE）的。

除了为多路分解处理定义的结构之外，dgram.h还定义了在应用程序和UDP协议软件之间传输的数据报的格式。范例软件没有将UDP数据报传送给应用程序，而是在结构xgram里定义了一个新格式。回想一下使用的多路分解处理方式，应用程序打开一个给定的协议端口号，并接收发往该端口的所有数据报。系统以xgram格式向应用程序传送数据报，这样应用程序就能够同时确定发送者的IP地址和协议端口号。

10.3.3 UDP端口号与队列的映射

UDP利用传入数据报中的目的端口号来选取dgtab中正确的表项。它在数组upqs中查找映射地址。数组upqs在文件udp.h中说明。过程udp_in（在10.3.6节给出）将目的站协议端口号与upqs数组的每个表项中的up_port字段相比较，直到它找到一个匹配表项为止。然后它利用字段up_xport来决定是否与用来将数据报置入队列的Xinu端口一致。

将upqs中的端口映射从dgtab中的队列中分离出去似乎有些多余，因为当前的实现方案在端口映射时应用的是线性查找。然而线性查找对只有少量活动UDP端口的系统来说是有效的。具有大量端口的系统应使用更有效的查找策略，如散列表（hashing）。将端口映射时使用的数据结构与数据报队列使用的数据结构分开，就有可能在改变映射算法时不用改变应用程序接口的数据结构。它们两者的分离还使操作系统有可能直接使用UDP，而无需依赖于应用程序这样的接口。

10.3.4 分配空闲队列

因为我们的范例代码对upqs数组采用了线性查找法，所以分配一个表项是十分简单明了的。

```
/* upalloc.c - upalloc */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>
#include <ports.h>

/*------------------------------------------------------------------------
 *  upalloc  -  allocate a UDP port demultiplexing queue
 *------------------------------------------------------------------------
 */
int upalloc(void)
{
```

```
struct upq      *pup;
int             i;

wait(udpmutex);
for (i=0 ; i<UPPS ; i++) {
      pup = &upqs[i];
      if (!pup->up_valid) {
            pup->up_valid = TRUE;
            pup->up_port = 0;
            pup->up_pid = BADPID;
            pup->up_xport = pcreate(UPPLEN);
            signal(udpmutex);
            return i;
      }
  }
  signal(udpmutex);
  return SYSERR;
}

struct upq  upqs[UPPS];
```

过程 upalloc 在数组中搜索，直到它找到一个当前未被使用的表项，然后它填写表项中的各个字段，创建一个用来传入数据报队列的 Xinu 端口，并向调用者返回此表项在数组中的索引值。

10.3.5　网络字节顺序与本机字节顺序之间的相互转换

有两个实用程序用于处理 UDP 首部字段在网络字节顺序和本机字节顺序之间的转换。过程 udpnet2h 为输入数据报处理向本机字节顺序的转换，程序代码自身有清楚的说明。

```
/* udpnet2h.c - udpnet2h */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  udpnet2h -  convert UDP header fields from net to host byte order
 *------------------------------------------------------------------------
 */
void
udpnet2h(pudp)
struct udp  *pudp;
{
      pudp->u_src = net2hs(pudp->u_src);
      pudp->u_dst = net2hs(pudp->u_dst);
      pudp->u_len = net2hs(pudp->u_len);
}
```

另一个与此有关的过程是udph2net，它用于将首部字段从本机字节顺序转换为标准网络字节顺序。

```
/* udph2net.c - udph2net */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  udph2net -  convert UDP header fields from host to net byte order
 *------------------------------------------------------------------------
 */
void
udph2net(pudp)
struct udp   *pudp;
{
        pudp->u_src = hs2net(pudp->u_src);
        pudp->u_dst = hs2net(pudp->u_dst);
        pudp->u_len = hs2net(pudp->u_len);
}
```

10.3.6 处理一个已到达的数据报

当有一个以本机为目的站的UDP数据报到达后，伪网络接口中的一个过程就调用过程udp_in。它以参数的形式传递含有输入分组的缓冲区地址以及网络接口索引，分组通过该网络接口到达。

```
/* udp_in.c - udp_in */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>
#include <ports.h>

/*------------------------------------------------------------------------
 *  udp_in -  handle an inbound UDP datagram
 *------------------------------------------------------------------------
 */
int udp_in(pni, pep)
struct netif *pni;
struct ep    *pep;
{
        struct     ip    *pip = (struct ip *)pep->ep_data;
        struct     udp   *pudp = (struct udp *)pip->ip_data;
        struct     upq   *pup;
        unsigned short   dst;
        int        i;
```

```
if (pudp->u_cksum && udpcksum(pep, net2ns(pudp->u_len))) {
        freebuf(pep);
        return SYSERR;              /* checksum error */
}
udpnet2h(pudp);        /* convert UDP header to host order */
dst = pudp->u_dst;
wait(udpmutex);
for (i=0 ; i<UPPS ; i++) {
        pup = &upqs[i];
        if (pup->up_port == dst) {
                /* drop instead of blocking on psend */
                if (pcount(pup->up_xport) >= UPPLEN) {
                        signal(udpmutex);
                        freebuf(pep);
                        UdpInErrors++;
                        return SYSERR;
                }
                psend(pup->up_xport, (worp)pep);
                UdpInDatagrams++;
                if (!isbadpid(pup->up_pid)) {
                        send(pup->up_pid, OK);
                        pup->up_pid = BADPID;
                }
                signal(udpmutex);
                return OK;
        }
}
signal(udpmutex);
UdpNoPorts++;
icmp(ICT_DESTUR, ICC_PORTUR, pip->ip_src, pep, 0);
return OK;
}

int     udpmutex;
```

udp_in 首先查看发送方是否支持校验和选项（判断其校验和字段是否非0）。如果有校验和存在，udp_in调用udpcksum来验证该校验和。如果分组包含的校验和正确，调用返回0。如果校验和项不为0，但又不正确，udp_in不再做进一步处理，直接丢弃该UDP数据报。udp_in还调用udpnet2h将所有首部字段转换为本机字节顺序。

在转换了首部之后，udp_in将数据报多路分解，并搜索一组数据报队列（数组upqs），直到它为传入数据报的目的UDP端口找到一个对应的队列。如果端口未满，udp_in调用psend将数据报置入队列，然后调用send向任何正在等待该数据报的进程发送一个报文。如果队列已满，udp_in记录溢出错误并丢弃该数据报。

如果udp_in搜索了全部数据报队列，却没有找到一个为传入数据报中的目的端口而保留的队

列，则意味着没有任何应用程序同意接收此端口上的数据报。在这种情况下，udp_in必须调用icmp向生成数据报的源站返回一个ICMP“目的站不可达”报文。

10.3.7 UDP校验和的计算

过程udpcksum为一个UDP数据报计算其校验和。与以前描述的过程cksum类似，该过程可用来生成一个校验和（通过将校验和的首部字段置0），或验证一个现有的校验和。然而，UDP校验和在一个重要的方面与以前提到的校验和不同：

> UDP校验和要包括整个UDP数据报及其伪首部，伪首部中包括了IP源地址和目的地址、UDP长度以及UDP协议标识符等。

在为一个外发数据报计算校验和的时候，协议软件必须找出在将UDP报文封装到一个IP数据报时所需要用到的数值。在验证一个已到达的报文的校验和时，UDP从携带该报文的IP数据报中提取各个值。在计算校验和时，将IP源地址和目的地址都包含在内的做法提供了某种保护措施，以免有路由错误的数据报也能顺利通过。

过程udpcksum不在内存中组装伪首部。事实上，它从IP首部中挑选出各个字段，然后在计算校验和时将它们都包括在其中。例如，udpcksum给数据报中的IP源地址字段的地址sptr赋值，并将以该地址打头的4个16比特存储区中的二进制数相加，这样就包括了IP源地址和目的地址。

```
/* udpcksum.c - udpcksum */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  udpcksum -  compute a UDP pseudo-header checksum
 *------------------------------------------------------------------------
 */
unsigned short udpcksum(pep, len)
struct ep    *pep;
int    len;
{
        struct  ip      *pip = (struct ip *)pep->ep_data;
        struct  udp     *pudp = (struct udp *)pip->ip_data;
        unsigned        short  *sptr;
        unsigned        long ucksum;
        int             i;

        ucksum = 0;

        sptr = (unsigned short *) &pip->ip-src;
        /* 2*IP_ALEN octets = IP_ALEN shorts...   */
        /* they are in net order.                 */
        for (i=0; i<UDP_ALEN; ++i)
```

```
        ucksum += *sptr++;
sptr = (unsigned short *)pudp;
ucksum += hs2net(IPT_UDP + len);
if (len % 2) {
          ((char *)pudp)[len] = 0;        /* pad */
          len += 1;       /* for the following division */
}
len >>= 1;      /* convert to length in shorts */

for (i=0; i<len; ++i)
        ucksum += *sptr++;
ucksum = (ucksum >> 16) + (ucksum & 0xffff);
ucksum += (ucksum >> 16);

return (short)(~ucksum & 0xffff);
}
```

10.4 UDP 输出的处理

在应用程序能够使用UDP通信之前，首先需要一个本地UDP端口号。使用熟知端口的服务器要求操作系统指定其中的一个端口。通常，客户不需要指定端口，它们可使用任意端口号。然而，由于我们的系统多路分解处理使用了仅有目的端口号的方式，所以必须为客户指定一个由它独自使用的端口号。过程udpnxtp生成一个当前空闲的UDP端口号。

```
/* udpnxtp.c - udpnxtp */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  udpnxtp  -  return the next available UDP local "port" number
 *     N.B.: assumes udpmutex HELD
 *------------------------------------------------------------------------
 */
unsigned short udpnxtp()
{
 static  unsigned short     lastport = ULPORT;
 Bool                       inuse = TRUE;
 struct  upq                *pup;
 int                        i;

 while (inuse) {
      lastport++;
      if (lastport == 0)
           lastport = ULPORT;
```

```
        inuse = FALSE;
        for (i=0; !inuse && i<UPPS ; i++) {
             pup = &upqs[i];
             inuse = pup->up_valid && pup->up_port == lastport;
        }
    }
    return lastport;
}
```

为了产生一个空闲的端口号，udpnxtp首先对全局计数器lastport的值加1。然后它循环经过所有UDP输入队列，查看是否有任何一个应用程序对应的端口号与当前lastport的值相同。通常，在循环过程中不会发现相同的值，这时udpnxtp向调用者返回lastport的值。如果它真的发现了相同的值，则udpnxtp将lastport的值再次加1，并重新开始循环查找。

10.4.1 一个UDP数据报的发送

当应用程序生成UDP输出之后，它将控制权移交给操作系统，并最终调用过程udpsend来发送UDP数据报。

```
/* udpsend.c - udpsend */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <igmp.h>

/*------------------------------------------------------------------------
 *  udpsend  -  send one UDP datagram to a given IP address
 *------------------------------------------------------------------------
 */
int
udpsend(fip, fport, lport, pep, datalen, docksum)
IPaddr      fip;
u_short     fport, lport;
struct ep   *pep;
unsigned    datalen;
Bool        docksum;
{
        struct  ip      *pip = (struct ip *) pep->ep_data;
        struct  udp     *pudp = (struct udp *) pip->ip_data;
        struct  hg      *phg;
        struct  route   *prt,
        int              ttl = IP_TTL;

        pudp->u_src = lport;
        pudp->u_dst = fport;
```

```
        pudp->u_len = U_HLEN+datalen;
        pudp->u_cksum = 0;
        udph2net(pudp);
        pep->ep_order &= ~EPO_UDP;
        if (docksum) {
             prt = rtget(fip, RTF_LOCAL);
             if (prt == NULL) {
                  IpOutNoRoutes++;
                  freebuf(pep);
                  return SYSERR;
             }
             pip >ip_dst = fip;
             if (prt->rt_ifnum == NI_LOCAL)
                  pip->ip_src = pip->ip_dst;
             else
                  pip->ip_src = nif[prt->rt_ifnum].ni_ip;
             rtfree(prt);
             pudp->u_cksum = udpcksum(pep, U_HLEN+datalen);
             if (pudp->u_cksum == 0)
                  pudp->u_cksum = ~0;
        }
        UdpOutDatagrams++;
        if (IP_CLASSD(fip)) {
                wait(HostGroup.hi_mutex);
                phg = hglookup(NI_PRIMARY, fip);
                if (phg)
                        ttl = phg->hg_ttl;
                else
                        ttl = 1;
                signal(HostGroup.hi_mutex);
         }
         return ipsend(fip, pep, U_HLEN+datalen,IPT_UDP,IPP_NORMAL,ttl);
}
```

由于网关与多个网络连接，因而它们有多个 IP 地址。在 udpsend 计算 UDP 校验和之前，它需要知道携带报文的 IP 数据报的源 IP 地址到底是其中的哪一个 IP 地址。为了找到正确的源 IP 地址，udpsend 调用过程 rtget，将 IP 目的地址作为参数传递给 rtget。一旦它选择好了一条路由，udpsend 提取网络接口号，从该接口上可获取该数据报的 IP 源地址。

rtget为IP数据报计算出源地址之后，udpsend填写UDP首部中剩余未填写的字段，调用udpcksum来计算校验和，并调用 ipsend 将最后生成的 IP 数据报传送给 IP 进程，以便为其选择路由和传输。

10.5　小结

UDP不但提供了对等程序之间的成对通信，还提供了客户与服务器之间的多对一通信方式。两

种多路分解处理的基本方式都可支持客户和服务器，且每种方式各有利弊。用于多路分解的范例程序代码仅仅使用了目的站协议端口号，并且服务器在其中起的作用很小。为了有助于支持客户，系统包括了一个用于生成惟一的（空闲的）协议端口号的过程。

UDP的输入和输出都很简单。IP进程执行UDP输入过程，该过程将数据报多路分解，且将各个数据报置入与它们目的协议端口对应的队列中。应用程序申请用来传输的端口，然后调用输出过程生成并发送UDP数据报。

UDP的校验和在计算时包括了伪首部中的字段，这样就可验证在携带UDP的IP数据报中包含的IP源地址和目的地址都是准确无误的。对输入来说，UDP可从携带UDP报文的IP输入数据报中得到它的伪首部各个字段的值。对输出来说，伪首部的处理使UDP校验和的计算复杂化，因为UDP不得不先确定IP将使用什么地址作为该数据报的源地址。

深入研究

Postel［RFC 768］定义了UDP协议，并规定了报文格式。Braden［RFC 1122］提供更进一步的说明。Leffler、Mckusick、Karels和Quarterman［1989］提供了BSD UNIX实现方案的细节。

习题

10.1 仔细阅读RFC，并判断是否所有用于验证UDP校验和的伪首部字段都必须从携带UDP数据报的IP数据报中获取？可以使用常量吗？为什么？

10.2 阅读BSD UNIX文档，它使用了哪种多路分解方式？

10.3 查看多数客户-服务器所使用的API套接字，它是否允许服务器明确指定客户终端？为什么？

10.4 你的本地系统是否允许你指定UDP输入队列的长度？如果可以，如何选择一个合理的长度值？

10.5 仔细阅读udpcksum。如果计算出的校验和的值正好是0，将发生什么？为什么？

10.6 请解释为什么udpnxtp生成的本地端口号的值在0到ULPORT（2050）之间滑动？

10.7 一个服务器系统能同时使用多少个UDP端口？一个典型的客户工作站又使用多少个？

10.8 在我们的例程代码中，多路分解时使用了线性查找法来查找一个UDP端口。设计一个可以减少查找时间的散列查找方案。你的方案是不是有时反而会增加查找时间？提示：可参考上题。

第11章　TCP：数据结构和输入处理

11.1　引言

TCP在互联网协议族的所有协议中是最复杂的。它为两个任意处理速率的、使用不可靠IP连接机制的机器之间的通信提供了可靠的、具有流量控制的、端到端的数据流服务。像大多数可靠的传输协议一样，TCP使用“重传超时机制”（timeout with retransmission）以实现其可靠性。然而，与大多数传输协议不同，TCP在构造时必须非常仔细，以保证其在数据报被延迟、复制、丢失、不按顺序发送或在发送损坏、截断的数据时仍然能够正确工作。更进一步说，TCP允许正在通信中的机器在任何时刻都能够重新启动或重新建立连接，而不会引起混乱，它能够分清哪个连接正在使用，哪个连接是新的。

本章详述了TCP软件的总体组成结构，并描述了TCP用来管理连接信息的数据结构。第12章具体描述了连接管理和用于输入的TCP有限状态机的实现方案。第13章中讨论了输出以及用来控制输出的有限状态机。第14章到第16章具体讨论定时器的管理、平均往返延迟时间的估算、重发以及其他细节问题，如紧急数据的处理。

11.2　TCP软件概览

在第2章中，我们在TCP实现方案中使用了三个进程。其中一个进程处理传入数据，另一个管理外发数据，第三个是管理诸如重发超时等延迟事件的定时器。从理论上讲，使用各自独立的进程可以使输入、输出以及TCP的事件定时部分互不影响，并允许独立地设计每个进程部分。但事实上，这几个进程之间关系密切，相互作用。例如，在将传入的确认与外发数据相比较以看它们是否匹配时，输入和输出进程必须通力合作，在取消某个重发事件对应的定时器时也是如此。类似地，当输出进程为某个重发事件定时或定时器触发重发事件时，输出进程又必须与定时器进程合作。

11.3　传输控制块

TCP通过一个数据结构为每个TCP连接协调传送、接收和重发动作，这个数据结构被所有进程共享。该数据结构称作传输控制块（transmission control block）或TCB。TCP为每个活动的连接保留一个TCB。TCB中包含了有关TCP连接的所有信息，包括连接端点的地址和端口号，当前平均往返时间的估计值，发送或接收的数据，是否需要确认或重发，以及一组有关该连接的使用情况的统计数据。

虽然协议标准定义了TCB的构思，并对其中的一些内容做了建议，但它没有硬性规定TCB的所有细节。因此，设计者必须为TCB选择明确的内容。在范例实现中，信息保存在结构tcb中。在绝大多数情况下，其中的字段名与协议标准中使用的名称一致。

```
/* tcb.h */

/* TCP endpoint types */

#define TCPT_SERVER         1
#define TCPT_CONNECTION     2
#define TCPT_MASTER         3

/* TCP process info */
extern PROCESS          tcpinp();
#define TCPISTK         1024        /* stack size for TCP input          */
#define TCPIPRI         100         /* TCP runs at high priority         */
#define TCPINAM         "tcpinp"    /* name of TCP input process         */
#define TCPIARGC        0           /* count of args to tcpin            */

extern PROCESS          tcpout();
#define TCPOSTK         1024        /* stack size for TCP output         */
#define TCPOPRI         100         /* TCP runs at high priority         */
#define TCPONAM         "tcpout"    /* name of TCP output process        */
#define TCPOARGC        0           /* count of args to tcpout           */

#define TCPQLEN         20          /* TCP process port queue length     */

/* TCP exceptional conditions */

#define TCPE_RESET          -1
#define TCPE_REFUSED        -2
#define TCPE_TOOBIG         -3
#define TCPE_TIMEDOUT       -4
#define TCPE_URGENTMODE     -5
#define TCPE_NORMALMODE     -6

/* string equivalents of TCPE_*, in "tcpswitch.c" */
extern char *tcperror[];

#define READERS         1
#define WRITERS         2

/* tcb_flags */

#define TCBF_NEEDOUT   0x01 /* we need output                            */
#define TCBF_FIRSTSEND 0x02 /* no data to ACK                            */
#define TCBF_GOTFIN    0x04 /* no more to receive                        */
#define TCBF_RDONE     0x08 /* no more receive data to process           */
#define TCBF_SDONE     0x10 /* no more send data allowed                 */
#define TCBF_DELACK    0x20 /* do delayed ACK's                          */
#define TCBF_BUFFER    0x40 /* do TCP buffering (default no)             */
#define TCBF_PUSH      0x80 /* got a push; deliver what we have          */
#define TCBF_SNDFIN    0x100/* user process has closed; send a FIN       */
```

```
#define TCBF_RUPOK     0x200/* receive urgent pointer is valid        */
#define TCBF_SUPOK     0x400/* send urgent pointer is valid           */

/* aliases, for user programs */

#define TCP_BUFFER     TCBF_BUFFER
#define TCP_DELACK     TCBF_DELACK

/* receive segment reassembly data */

#define NTCPFRAG       10

struct tcb {
        short   tcb_state;          /* TCP state                      */
        short   tcb_ostate;         /* output state                   */
        short   tcb_type;           /* TCP type (SERVER, CLIENT)      */
        int     tcb_mutex           /* tcb mutual exclusion           */
        short   tcb_code;           /* TCP code for next packet       */
        short   tcb_flags;          /* various TCB state flags        */
        short   tcb_error;          /* return error for user side     */

        IPaddr  tcb_rip;            /* remote IP address              */
        u_short tcb_rport;          /* remote TCP port                */
        IPaddr  tcb_lip;            /* local IP address               */
        u_short tcb_lport;          /* local TCP port                 */
        struct netif    *tcb_pni;   /* pointer to our interface       */

        tcpseq  tcb_suna;           /* send unacked                   */
        tcpseq  tcb_snext;          /* send next                      */
        tcpseq  tcb_slast;          /* sequence of FIN, if TCBF_SNDFIN */
        u_long  tcb_swindow;        /* send window size (octets)      */
        tcpseq  tcb_lwseq;          /* sequence of last window update */
        tcpseq  tcb_lwack;          /* ack seq of last window update  */
        u_int   tcb_cwnd;           /* congestion window size (octets) */
        u_int   tcb_ssthresh;       /* slow start threshold (octets)  */
        u_int   tcb_smss;           /* send max segment size (octets) */
        tcpseq  tcb_iss;            /* initial send sequence          */

        int     tcb_srt;            /* smoothed Round Trip Time       */
        int     tcb_rtde;           /* Round Trip deviation estimator */
        int     tcb_persist;        /* persist timeout value          */
        int     tcb_keep;           /* keepalive timeout value        */
        int     tcb_rexmt;          /* retransmit timeout value       */
        int     tcb_rexmtcount;     /* number of rexmts sent          */

        tcpseq  tcb_rnext;          /* receive next                   */
        tcpseq  tcb_rupseq;         /* receive urgent pointer         */
        tcpseq  tcb_supseq;         /* send urgent pointer            */
```

```
        int     tcb_lqsize;         /* listen queue size (SERVERs)        */
        int     tcb_listenq;        /* listen queue port (SERVERs)        */
        struct  tcb *tcb_pptcb;     /* pointer to parent TCB (for ACCEPT)*/
        int     tcb_ocsem;          /* open/close semaphore               */
        int     tcb_dvnum;          /* TCP slave pseudo device number     */

        int     tcb_ssema;          /* send semaphore                     */
        u_char  *tcb_sndbuf;        /* send buffer                        */
        u_int   tcb_sbstart;        /* start of valid data                */
        u_int   tcb_sbcount;        /* data character count               */
        u_int   tcb_sbsize;         /* send buffer size (bytes)           */

        int     tcb_rsema;          /* receive semaphore                  */
        u_char  *tcb_rcvbuf;        /* receive buffer (circular)          */
        u_int   tcb_rbstart;        /* start of valid data                */
        u_int   tcb_rbcount;        /* data character count               */
        u_int   tcb_rbsize;         /* receive buffer size (bytes)        */
        u_int   tcb_rmss;           /* receive max segment size           */
        tcpseq  tcb_cwin;           /* seq of currently advertised window*/
        int     tcb_rseqq;          /* segment fragment queue             */
        tcpseq  tcb_finseq;         /* FIN sequence number, or 0          */
        tcpseq  tcb_pushseq;        /* PUSH sequence number, or 0         */
};
/* TCP fragment structure */

struct tcpfrag {
        tcpseq  tf_seq;
        int     tf_len;
};
/* TCP control() functions */

#define TCPC_LISTENQ   0x01 /* set the listen queue length              */
#define TCPC_ACCEPT    0x02 /* wait for connect after passive open      */
#define TCPC_STATUS    0x03 /* return status info (all, for master)     */
#define TCPC_SOPT      0x04 /* set user-selectable options              */
#define TCPC_COPT      0x05 /* clear user-selectable options            */
#define TCPC_SENDURG   0x06 /* write urgent data                        */

/* global state information */

extern int tcps_oport;      /* Xinu port to start TCP output            */
extern int tcps_iport;      /* Xinu port to send TCP input packets      */
extern int tcps_lqsize;     /* default SERVER queue size                */
extern int tcps_tmutex;     /* tcb table mutex                          */

extern int  (*tcpswitch[])(struct tcb*, struct ep*),
            (*tcposwitch[])(int, int);
```

```
#ifdef Ntcp
extern struct    tcb  tcbtab[];
#endif

struct tcb *tcpdemux(struct ep *), *tcballoc(void);
int tfcoalesce(struct tcb *, unsigned, struct tcp *);
int tfinsert(struct tcb *, tcpseq, unsigned*);
int tcbdealloc(struct tcb *);
Bool tcpok(struct tcb *, struct ep *);
int tcpackit(struct tcb *, struct ep *);
int tcpwait(struct tcb *);
int tcpdata(struct tcb *, struct ep *);
int tcpacked(struct tcb *, struct ep *);
int tcpabort(struct tcb *, int);
int tcpopts(struct tcb *, struct ep *);
int tcpswindow(struct tcb *, struct ep *);
int tcpdodat(struct tcb *, struct tcp *, tcpseq, unsigned);
int tcpkick(struct tcb *);
int tcpwakeup(int, struct tcb *);
int tcpostate(struct tcb *, int);
int tcprwindow(struct tcb *);
int tcprtt(struct tcb *);
int tcpsync(struct tcb *);
int tcpwinit(struct tcb *, struct tcb *, struct ep *);
int tosmss(struct tcb *, struct tcp *, u_char *);
int tosmss(struct tcb *, struct ip *);
int tcpsndlen(struct tcb *, Bool, unsigned int *);
int tcpowmuch(struct tcb *);
int tcplq(struct tcb *, unsigned);
```

尽管在讲解使用TCB各字段的过程之前，很难完全理解TCB中所有字段的含义，但其中有些字段的含义还是显而易见的。例如，除了指出该连接当前输入、输出状态的字段（tcb_state 和 tcb_ostate）之外，tcb 结构中还包括了如下字段：一个互斥信号量（tcb_mutex）、本地和远程 IP 地址（tcb_lip 和 tcb_rip）、本地和远程端口号（tcb_lport 和 tcb_rport）以及使用的网络接口（tcb_pni）。

当然在tcb结构中还包含了发送报文段时需要的信息：接收方当前窗口大小（tcb_swindow），准备发送的下一个报文段的序号（tcb-snext），在序列中还未收到确认的报文的最低字节（tcb_suna），拥塞窗口大小（tcb_cwnd），慢启动阈值（tcb_ssthresh）以及最大报文段长度（tcb_smss）。

对重发来说，在 tcb 结构中保存已平滑过的平均往返时延估值（tcb_srt），往返时延偏差估值（tcb_rtde），重发超时时间（tcb_rexmt），以及连续重发次数的计数器（tcb_rexmtcount）。

另外，tcb 中还有用于接收报文段的字段。除接收缓冲区的地址（tcb_rcvbuf）外，tcb 还包含了以下字段：用于指出有效数据的起点的字段（tcb_rbstart），计算接收缓冲区中字符数的计数器（rbcount），最大允许报文段长度（tcb_rmss）和前一次通告窗口的序号（tcb_cwin）。我们将在今后的章节中讨论其余的字段。

由于报文段可能不按顺序到达，当这些报文段抵达时，TCP还必须保存有关数据块的信息，直

至这些数据块能被重组成连续的数据流。TCP将该信息保存在一个使用tcpfrag结构的链表中，在文件tcb.h中定义，用于保存起始序号以及每个数据块的长度。

11.4 TCP报文段格式

结构tcp定义了TCP报文段的格式。文件tcp.h中不但有对tcp的说明，还有定义了字段tcp_code中各个比特含义的符号常量说明。

```
/* tcp.h - TCP_HLEN, SEQCMP */

typedef     long tcpseq;

/*
 * SEQCMP - sequence space comparator
 *     This handles sequence space wrap-around. Overlow/Underflow makes
 * the result below correct ( -, 0, + ) for any a, b in the sequence
 * space. Results:     result     implies
 *                     -          a < b
 *                     0          a = b
 *                     +          a > b
 */
#define     SEQCMP(a, b)    ((a) - (b))

/* tcp packet format */

struct tcp {
      unsigned  short  tcp_sport;    /* source port                    */
      unsigned  short  tcp_dport;    /* destination port               */
                tcpseq tcp_seq;      /* sequence                       */
                tcpseq tcp_ack;      /* acknowledged sequence          */
      unsigned  char   tcp_offset;
      unsigned  char   tcp_code;     /* control flags                  */
      unsigned  short  tcp_window;   /* window advertisement           */
      unsigned  short  tcp_cksum;    /* check sum                      */
      unsigned  short  tcp_urgptr;   /* urgent pointer                 */
      unsigned  char   tcp_data[1];
};

/* TCP Control Bits */

#define TCPF_URG      0x20      /* urgent pointer is valid            */
#define TCPF_ACK      0x10      /* acknowledgement field is valid     */
#define TCPF_PSH      0x08 /* this segment requests a push            */
#define TCPF_RST      0x04 /* reset the connection                    */
#define TCPF_SYN      0x02 /* synchronize sequence numbers            */
#define TCPF_FIN      0x01 /* sender has reached end of its stream    */
```

```
#define TCPMHLEN        20    /* minimum TCP header length                */
#define TCPHOFFSET      0x50  /* tcp_offset value for TCPMHLEN            */
#define TCP_HLEN(ptcp)        (((ptcp)->tcp_offset & 0xf0)>>2)

/* TCP Options */

#define TPO_EOOL        0     /* end Of Option List                       */
#define TPO_NOOP        1     /* no Operation                             */
#define TPO_MSS         2     /* maximum Segment Size                     */

int tcpxmit (int, int);
unsigned short tcpcksum(struct ep*, unsigned);
strct tcp *tcpnet2h(struct tcp *), *tcph2net (struct tcp *);
int tcpreset(struct ep *);
int tcpsend(int, Bool);
int tcpiss(void);
int tcprexmt(int, int);
int tcpwr(struct devsw *, u_char *, unsigned, Bool);
```

在文件tcp.h中还定义了宏函数TCP_HLEN，它用于计算TCP报文段首部的长度，以八位组为单位。以32比特字为单位的首部长度被保存在8比特偏移字段的高位4比特中。为了计算以字节为单位的首部长度，宏TCP_HLEN必须把以32比特字为单位的长度值乘以4。要做到这一点，它引用整个偏移字段，用逻辑“与”计算并提取其中表示长度的4比特，然后将它们转换成需要的值。

11.5 序列空间中的比较

TCP为数据流中的每个八位组设定一个整数，称作“序号”（或“序列值”）。当TCP以报文段的形式发送数据时，它在每个报文段的首部填写该序号。TCP在接收数据时用序号来判断抵达的报文段的顺序是否混乱，并将它们重新按序排列成正确的线性顺序数据流。TCP为每个连接随机地选取起始的第一个序号，以防止由于分组的延迟或重复而在其三次握手时导致混乱。

这一组所有可能的整型序号被称为TCP“序列空间”（sequence space）。由于TCP报文段首部的序号字段具有固定长度（32比特），因而允许序号在达到序列空间的尽头时，绕回到0，重新开始。TCP软件经常需要比较两个序号，以决定一个报文段中的序号是大于还是小于另一个报文段的序号。如果使用常规的比较法，则像“0”这样的小数值永远小于其他大数值，虽然当序号从序列空间的尽头绕回到“0”时，“0”是跟在最大的序号后面的。

出乎意料的是，常规的计算机算法可用来正确地建立两个序号之间的关系，只要序列空间长度等于该机器上整数值的范围，并且要求两个被比较的值之间的差距不大于整数空间的一半。在目前的计算机和网络上，TCP永远也不需要比较差距大于最大整数值一半的两个数值，因为计算机无法做到极快速地生成输出数据，而导致在数据报超时之前绕经了整个序列空间。

如果两个相差不大的序号相比较，简单的整数减法就可得到正确的结果。当一个很小的数减去一个很大的数值时，整数的下溢处理将正确对待这种情况。也就是说，如果a和b是差距不超过最大有效整数值一半的两个序号，则其比较结果如下：

a-b的结果	在序列空间中的关系
-	a先于b
0	a等于b
+	a后于b

图11.1 差距不超过最大有效整数值一半的两个序列值相减所得到的结果

因此我们可以总结如下：

TCP利用整数减法来比较两个序列值的大小，因为它可以假设两个序列值之间的差距永远也不会大于序列空间值的一半。在这种情况下，当一个很大的值与一个很小的值相比较时，整数计算时的下溢仍将产生正确结果。

文件tcp.h[1]中的宏SEQCMP实现了序列空间中的比较。为了有助于读者清楚地区分常规性减法操作还是序列空间中的比较操作，TCP程序代码在做比较操作时使用SEQCMP。

11.6 TCP有限状态机

从理论上讲，TCP使用有限状态机制来控制所有进程的交互作用。每个TCP连接的端点上都具有一份状态机的副本，并利用它来控制报文段到达时发生的交互作用。图11.2所示为TCP有限状态机以及各状态之间的变迁关系。

从理论上看，有限状态机已经完全体现了一台机器上的TCP是如何与另一台机器上的TCP交互作用的。但实际上，有限状态机并没有完整地描述各种交互作用。事实上，状态机仅描述了TCP的宏观状态，而另外一些变量更进一步地说明了微观状态的细节。更重要的是，因为由状态机规定的宏观状态之间的变迁并不能控制输出或重发动作，所以这一类的事件必须另行处理。我们总结如下：

有限状态机规定了TCP在响应一个输入或用户命令时，如何在各个宏观状态之间互相变迁。一个具体的实现中还要包括另一个独立机制，用来完成各个微观状态之间的相互变迁，以控制输出和重发动作。

11.7 状态变迁举例

为了理解TCP有限状态机制，让我们来看一个例子，它是用于在客户和服务器之间建立连接的三次握手操作。不论是客户还是服务器，它们都要为通信创建一个端点，并且都具有一份有限状态机的副本。服务器首先发起一个“被动打开”（passive open）的操作，这将导致服务器的有限状态机进入“听”（LISTEN）状态。服务器在“听”状态中等待，直到有一个客户与它联络。当有一个客户发起“主动打开”（active open）操作时，它导致该机器上的TCP软件发送一个SYN报文段给服务器，并进入“SYN已发出”（SYN-SENT）状态。

当正在“听”状态中等待的服务器接收到SYN报文段后，它以一个SYN，外加一个ACK报文段作为应答，并创建一个新的TCB，将新的TCB置于“SYN收到”（SYN-RECEIVED）状态。当SYN

① 文件tcp.h可在11.4节中找到。

加上 ACK 的报文段到达客户后，客户的 TCP 以一个 ACK 作为应答，并从“SYN 已发出”状态中变迁到“已经建立”（ESTABLISHED）状态。最后，当客户的 ACK 报文段到达新创建的 TCB 后，该 TCB 也进入“已经建立”状态，这时就可以进行数据传输了。

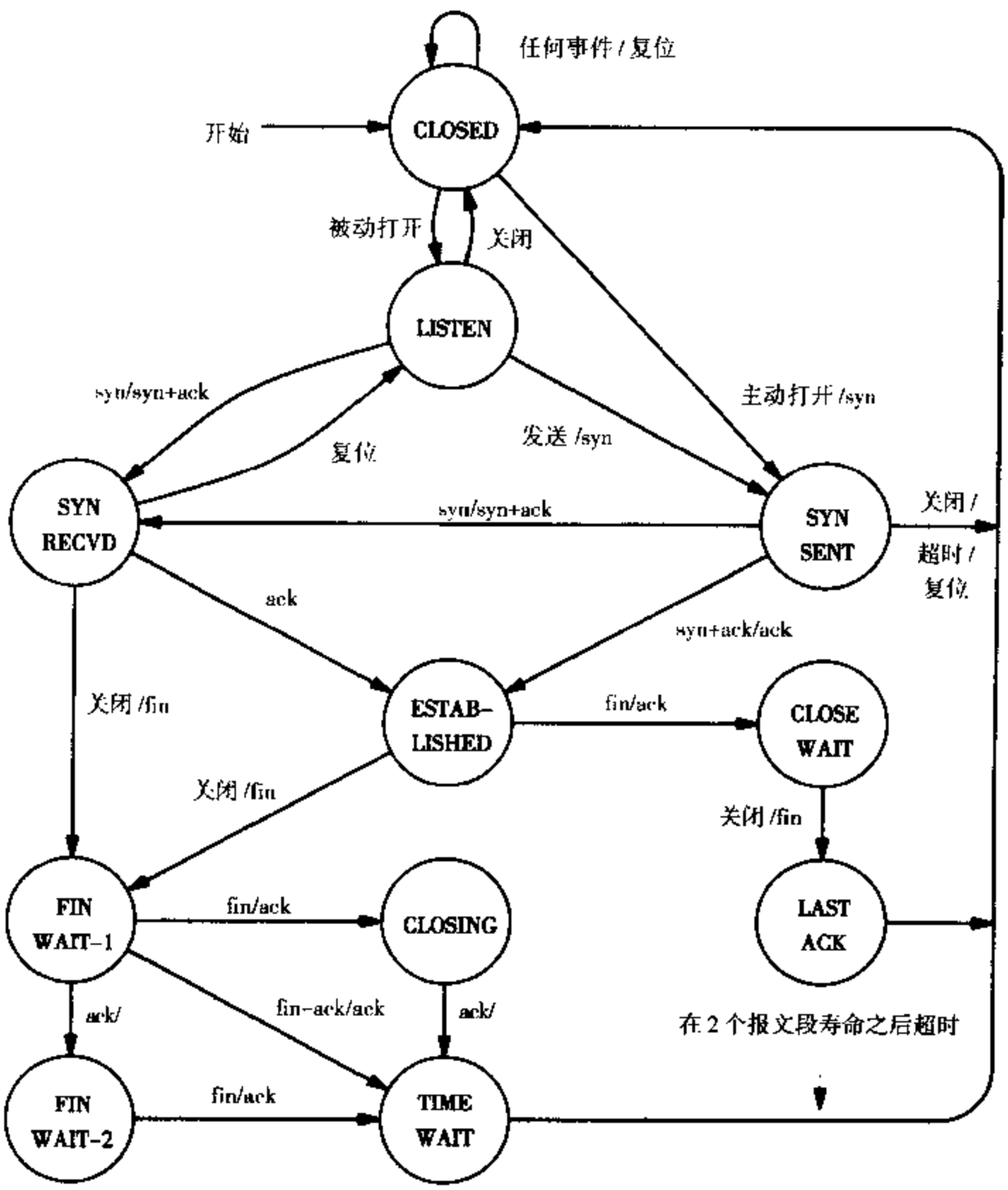


图 11.2　TCP 用于控制进程动作的有限状态机

11.8　有限状态机的说明

文件 tcpfsm.h 中的内容是对 TCP 有限状态机中各状态的符号常量的说明。

```
/* tcpfsm.h - TCB, EVENT, MKEVENT */

/* TCP states */

#define TCPS_FREE           0
```

```
#define TCPS_CLOSED         1
#define TCPS_LISTEN         2
#define TCPS_SYNSENT        3
#define TCPS_SYNRCVD        4
#define TCPS_ESTABLISHED    5
#define TCPS_FINWAIT1       6
#define TCPS_FINWAIT2       7
#define TCPS_CLOSEWAIT      8
#define TCPS_LASTACK        9
#define TCPS_CLOSING        10
#define TCPS_TIMEWAIT       11

#define NTCPSTATES          12

/* Output States */

#define TCPO_IDLE           0
#define TCPO_PERSIST        1
#define TCPO_XMIT           2
#define TCPO_REXMT          3

#define NTCPOSTATES         4

/* event processing */

#define SEND            0x1
#define PERSIST         0x2
#define RETRANSMIT      0x3
#define DELETE          0x4
#define TMASK           0x7

#define EVENT(x)        ((int)(x) & TMASK)
#define TCB(x)          ((int)(x) >> 3)
#define MKEVENT(timer, tcb) (void *)((tcb<<3) | ((timer) & TMASK))

/* implementation parameters */

#define TCP_MAXRETRIES   12      /* max retransmissions before giving up*/
#define TCP_TWOMSL    12000      /* 2 minutes (2 * Max Segment Lifetime)*/
#define TCP_MAXRXT     2000      /* 20 seconds max rexmt time           */
#define TCP_MINRXT       50      /* 1/2 second min rexmt time           */
#define TCP_ACKDELAY     20      /* 1/5 sec ACK delay, if TCBF_DELACK   */

#define TCP_MAXPRS     6000      /* 1 minute max persist time           */

/* second argument to tcpsend(): */

#define TSF_NEWDATA    0         /* send all new data                   */
#define TSF_REXMT      1         /* retransmit the first pending segment */
```

```
/* third argument to tcpwr():     */

#define TWF_NORMAL     0          /* normal data write                */
#define TWF_URGENT     1          /* urgent data write                */
```

11.9 TCB 的分配及初始化

11.9.1 分配一个 TCB

在 TCP 建立一个连接时，用于实现 TCP 有限状态机的过程必须分配并初始化一个 TCB。要做到这一点，它调用过程 tcballoc。

```
/* tcballoc.c - tcballoc */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 * tcballoc - allocate a Transmission Control Block
 *------------------------------------------------------------------------
 */
struct tcb *tcballoc(void)
{
        struct  tcb     *ptcb;
        int             slot;

        wait(tcps_tmutex);
        /* look for a free TCB */

        for (ptcb=&tcbtab[0], slot=0; slot<Ntcp; ++slot, ++ptcb)
                if (ptcb->tcb_state == TCPS_FREE)
                        break;
        if (slot < Ntcp) {
                ptcb->tcb_state = TCPS_CLOSED;
                ptcb->tcb_mutex = screate(0);
        } else
                ptcb = (struct tcb *)SYSERR;
        signal(tcps_tmutex);
        return ptcb;
}
```

tcballoc 搜索数组 tcbtab，直到它找到一个空闲表项（即，一个状态为 TCPS_FREE 的表项）。如果有空闲表项存在，tcballoc 将状态变为“关闭”（CLOSED，连接的初始状态），同时为 TCB 生成一个互斥信号量，并向调用者返回新分配表项的地址。调用返回时持有 tcb_mutex。也就是说，调用

返回时，已获取了单独访问这个新TCB的权力。如果没有空闲TCB存在，tcballoc返回一个SYSERR，指出有差错发生。

11.9.2 释放一个TCB

当取消一个连接时，TCP软件调用过程tcbdealloc来释放该连接的TCB，并允许它被其他连接再次使用。

```
/* tcbdealloc.c - tcbdealloc */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <sem.h>
#include <ports.h>
#include <mem.h>

/*------------------------------------------------------------------------
 *  tcbdealloc - deallocate a TCB and free its resources
 *     ASSUMES ptcb->tcb_mutex HELD
 *  ------------------------------------------------------------------------
 */
int tcbdealloc(ptcb)
struct tcb  *ptcb;
{
        if (ptcb->tcb_state == TCPS_FREE)
                return OK;
        switch (ptcb->tcb_type) {
        case TCPT_CONNECTION:
                tcpkilltimers(ptcb);
                sdelete(ptcb->tcb_ocsem);
                sdelete(ptcb->tcb_ssema);
                sdelete(ptcb->tcb_rsema);
                freemem(ptcb->tcb_sndbuf, ptcb->tcb_sbsize);
                freemem(ptcb->tcb_rcvbuf, ptcb->tcb_rbsize);
                if (ptcb->tcb_rsegq >= 0)
                        freeq(ptcb->tcb_rsegq);
                break;
        case TCPT_SERVER:
                pdelete(ptcb->tcb_listenq, 0);
                break;
        default:
                signal(ptcb->tcb_mutex);
                return SYSERR;
        }
        ptcb->tcb_state = TCPS_FREE;
```

```
        sdelete(ptcb->tcb_mutex);
        return OK;
}
```

因为有两种特殊情况的存在，使 TCB 的释放稍有些复杂。如果一个连接已经被使用，那么 tcbdealloc 必须首先调用 tcpkilltimers 删除任何未完成的定时事件。然后，它还要删除发送和接收信号量，以及用于存储传入和外发数据的缓冲存储区。对一个服务器来说，tcbdealloc 必须删除传入的连接请求的队列。最后，无论在哪种情况下，tcbalealloc 都必须删除互斥信号量。我们将会在今后的章节中看到 TCP 软件是如何分配并使用信号量和缓冲区的。

11.10　有限状态机的实现

设计者在实现 TCP 有限状态机时必须选取以下两种基本方案中的一种：

- 表驱动
- 过程驱动

单纯的表驱动方案利用一个二维数组，其中每一行对应一个状态，每一列对应一种可能的输入事件，或引起状态变迁的操作。因此每个表项对应一个处于某种状态下的输入事件。该表项中包含了被调用来处理该事件的过程的地址，同时还有状态变迁后的整型状态值。在TCB中的一个状态字段指明当前状态。当某个输入事件发生后，TCP将其转换到对应的列表中，并利用引起变迁的输入事件和当前状态来选取表中的一个表项。TCP利用该表项选择并调用一个过程，然后更新当前状态变量值。

过程驱动方案为每个输入状态启动一个过程。当某一事件发生时，TCP利用当前状态来选取一个适当的过程。该过程处理输入事件并更新当前状态变量值。

在实现一个结构整齐、语义简单且具有相对来说较复杂的变迁图的有限状态机时，表驱动方案工作得更出色一些。而在实现一个变迁状态少、但语义复杂的有限状态机时，过程驱动方案工作得更出色。我们选择了后者。

> 因为TCP状态机包含少量的状态，规定的状态间的变迁也不多，同时又提供了复杂的操作，并包括许多处理差错的特殊情况，所以我们的实现范例中使用过程驱动方案。

因此在我们的实现方案中，每个状态对应一个过程，如图 11.2 所示，并在 TCB 中有一个字段指出当前状态。当一个输入报文段到达时，TCP 调用当前状态对应的过程。另外，我们的实现方案还为每个本地操作提供一个对应过程（例如，服务器利用一个独立的过程发出“被动打开”的报文）。

11.11　处理一个输入报文段

当 IP 接收到以本机为目的站的 TCP 报文段时，它最终会调用 tcp_in，向 TCP 传递此报文段。

```
/* tcp_in.c - tcp_in */

#include <conf.h>
```

```
#include <kernel.h>
#include <network.h>
#include <ports.h>

/*------------------------------------------------------------------------
 *  tcp_in - deliver an inbound TCP packet to the TCP process
 *------------------------------------------------------------------------
 */
int tcp_in(pni, pep)
struct netif*pni;
struct ep    *pep;
{
        /* drop instead of blocking on psend */

        TcpInSegs++;
        if (pcount(tcps_iport) >= TCPQLEN) {
                freebuf(pep);
                return SYSERR;
        }
        psend(tcps_iport, (int)pep);
        return OK;
}
```

如程序代码所示，tcp_in向TCP输入端口发送传入的报文段，TCP输入进程从该端口中提取报文段。TCP输入进程执行过程tcpinp。

```
/* tcpinp.c - tcpinp */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ports.h>

/*------------------------------------------------------------------------
 *  tcpinp  -  handle TCP segment coming in from IP
 *------------------------------------------------------------------------
 */
PROCESS tcpinp(void)
{
        struct     ep    *pep;
        struct     ip    *pip;
        struct     tcp   *ptcp;
        struct     tcb   *ptcb;

        tcps_iport = pcreate(TCPQLEN);
        signal(Net.sema);
        while (TRUE) {
```

```
        pep = (struct ep *)preceive(tcps_iport);
        if ((int)pep == SYSERR)
                break;
        pip = (struct ip *)pep->ep_data;
        if (tcpcksum(pep, pip->ip_len-IP_HELN(pip)) {
                ++TcpInErrs;
                freebuf(pep);
                continue;
        }
        ptcp = (struct tcp *)pip->ip_data;
        tcpnet2h(ptcp); /* convert all fields to host order */
        pep >ep_order |= EPO_TCP;
        ptcb = tcpdemux(pep);
        if (ptcb == 0) {
                ++TcpInerrs;
                tcpreset(pep);
                freebuf(pep);
                continue;
        }
        if (!tcpok(ptcb, pep))
                tcpackit(ptcb, pep);
        else {
                tcpopts(ptcb, pep);
                tcpswitch[ptcb->tcb_state](ptcb, pep);
        }
        if (ptcb->tcb_state != TCPS_FREE)
                signal(ptcb->tcb_mutex);
        freebuf(pep);
    }
}

int tcps_oport, tcps_iport, tcps_lqsize, tcps_tmutex;
```

tcpinp 不断地从输入端口中提取一个个报文段，然后调用 tcpcksum 来验证其校验和，并调用 tcpnet2h将其首部字段的网络字节顺序转换成本地字节顺序。它利用tcpdemux为该报文段找出正确的TCB（如果不存在这样的TCB，则调用tcpreset来发送一个RESET），然后它调用tcpok验证对于当前窗口该报文段是否可以接受，如果不能，则调用tcpackit来发送一个确认报文[①]。最后，tcpinp 利用 tcpopts 处理报文段中的选项，并利用数组 tcpswitch 为当前输入状态选择一个对应的过程。从下一小节开始，我们将看到 tcpinp 调用的各个过程。

11.11.1　将 TCP 首部转换为本地字节顺序

过程 tcpnet2h 将 TCP 首部中的整数字段从网络标准字节顺序转换为本地机器字节顺序。

① 根据协议标准，该确认并不证实是否收到了不可接受的报文段，而只证实收到了正确的报文序列及当前窗口大小的信息。

```
/* tcpnet2h.c - tcpnet2h */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpnet2h -  convert TCP header fields from net to host byte order
 *------------------------------------------------------------------------
 */
struct tcp *tcpnet2h(ptcp)
struct tcp  *ptcp;
{
        /* NOTE: does not include TCP options */

        ptcp->tcp_sport = net2hs(ptcp->tcp_sport);
        ptcp->tcp_dport = net2hs(ptcp->tcp_dport);
        ptcp->tcp_seq = net2hl(ptcp->tcp_seq);
        ptcp->tcp_ack = net2hl(ptcp->tcp_ack);
        ptcp->tcp_window = net2hs(ptcp->tcp_window);
        ptcp->tcp_urgptr = net2hs(ptcp->tcp_urgptr);
        return ptcp;
}
```

11.11.2 计算TCP的校验和

TCP与UDP的计算校验和的方法相同。开始时，tcpcksum计算伪首部的校验和，该伪首部中包括了IP源地址和目的地址、报文段长度以及IP使用的协议类型值（在字段ip_proto中的值）。然后它将报文段看作一组16位的整数，并将它们相加以得到校验和。最后，tcpcksum处理溢出并向调用者返回校验和的补码。

```
/* tcpcksum.c - tcpcksum */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpcksum -  compute a TCP pseudo-header checksum
 *------------------------------------------------------------------------
 */
unsigned short tcpcksum(pep, len)
struct ep   *pep;
unsigned    len;
{
        struct ip       *pip=(struct ip*)pep->ep_data;
        struct tcp      *ptcp = (struct tcp *)pip->ip_data;
```

```
        unsigned        short *sptr;
        unsigned        long tcksum;
        unsigned        i;

        tcksum = 0;

        sptr = (unsigned short *) &pip->ip_src;
        /* 2*IP_ALEN octets = IP_ALEN shorts... */
        /* they are in net order.       */
        for (i=0; i<IP_ALEN; ++i)
             tcksum += *sptr++;
        sptr = (unsigned short *)ptcp;
        tcksum += hs2net(IPT_TCP + len);
        if (len % 2) {
             ((char *)ptcp)[len] = 0;   /* pad */
             len += 1;  /* for the following division */
        }
        len >>= 1; /* convert to length in shorts */

        for (i=0; i<len; ++i)
             tcksum += *sptr++;
        tcksum = (tcksum >> 16) + (tcksum & 0xffff);
        tcksum += (tcksum >> 16);

        return (short)(~tcksum & 0xffff);
}
```

11.11.3　为报文段查找对应的 TCB

过程tcpdemux为一个输入的报文段查找正确的TCB。程序代码顺序地搜索数组tcbtab。对于那些对应于已建立的连接的TCB来说，tcpdemux要进行四次比较，以检查连接的两个端点。除了将报文段中的源站和目的站协议端口号与表项中的两个端口号相比较外，它还要将IP数据报中的源站和目的站IP地址与表项中的两个IP地址相比较。然而，由于服务器仅指定一个本地IP地址或协议端口号，因此tcpdemux对这些表项中的源地址不进行比较。这样，对处于“听”状态中的TCB来说，tcpdemux仅比较目的站协议端口号。

如果某个输入报文段的连接已经存在，tcpdemux在获得互斥信号量之后为该报文段返回一个指针，指向对应的表项。如果连接不存在，tcpdemux检查报文段的类型。对大多数报文段类型来说，tcpdemux返回差错码（“0”）。然而，如果输入报文段中含有同步（SYN）请求，并且已经有一个服务器发出了“被动打开”的报文，tcpdemux为该服务器返回一个指针，指向TCB表项。当然，如果并没有服务器为这个特定的目的地址创建TCB，则tcpdemux为该SYN请求返回差错报文。

为了使搜索的效率更高，tcpdemux一次性搜索所有可能的连接。在搜索过程中，它查找一个精确的匹配项（即TCB中的两端点与输入报文段中两端点都匹配的连接），而且还要记录下非完全匹配的项（只需目的站匹配的服务器连接）。在搜索完成后，它查看该报文段是否包含SYN请求。如果是，它返回任何可能找到的非完全匹配项。

```
/* tcpdemux.c - tcpdemux */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpdemux -  do TCP port demultiplexing
 *------------------------------------------------------------------------
 */
struct tcb *tcpdemux(pep)
struct ep    *pep;
{
        struct     ip    *pip = (struct ip *)pep->ep_data;
        struct     tcp   *ptcp = (struct tcp *)pip->ip_data;
        struct     tcb   *ptcb;
        int              tcbn, lstcbn;

        wait(tcps_tmutex);
        for (tcbn=0, lstcbn = -1; tcbn<Ntcp; ++tcbn) {
             if (tcbtab[tcbn].tcb_state == TCPS_FREE)
                  continue;
             if (ptcp->tcp_dport == tcbtab[tcbn].tcb_lport &&
                 ptcp->tcp_sport == tcbtab[tcbn].tcb_rport &&
                 pip->ip_src == tcbtab[tcbn].tcb_rip &&
                 pip->ip_dst == tcbtab[tcbn].tcb_lip {
                  break;
             }
             if (tcbtab[tcbn].tcb_state == TCPS_LISTEN &&
                 ptcp->tcp_dport == tcbtab[tcbn].tcb_lport)
                  lstcbn = tcbn;
        }
        if (tcbn >= Ntcp)
             if (ptcp->tcp_code & TCPF_SYN)
                  tcbn =  lstcbn;
             else
                  tcbn = -1;
        signal(tcps_tmutex);
        if (tcbn < 0)
             return 0;
        wait(tcbtab[tcbn].tcb_mutex);
        if (tcbtab[tcbn].tcb_state == TCPS_FREE)
             return 0;              /* OOPS! Lost it... */
        return &tcbtab[tcbn];
}
```

11.11.4　检查报文段的有效性

我们已经知道 tcpinp 在为有限状态机变迁状态之前，先要调用函数 tcpok 来检查报文段的有效性。tcpok 比较输入的报文段中的信息和 TCB 中的信息，以判断该报文段中的数据是否在接收窗口之内。

```
/* tcpok.c - tcpok */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpok -  determine if a received segment is acceptable
 *------------------------------------------------------------------------
 */
Bool tcpok(ptcb, pep)
struct tcb  *ptcb;
struct ep   *pep;
{
        struct  ip      *pip = (struct ip *)pep->ep_data;
        struct  tcp     *ptcp = (struct tcp *) pip->ip_data;
        int             seglen, rwindow;
        tcpseq          wlast, slast, sup;
        Bool            rv;

        if (ptcb->tcb_state < TCPS_SYNRCVD)
                return TRUE;
        seglen = pip->ip_len - IP_HLEN(pip) - TCP_HLEN(ptcp);

        /* add SYN and FIN */
        if (ptcp->tcp_code & TCPF_SYN)
                ++seglen;
        if (ptcp->tcp_code & TCPF_FIN)
                ++seglen;
        rwindow = ptcb->tcb_rbsize - ptcb->tcb_rbcount;
        if (rwindow == 0 && seglen == 0)
                return ptcp->tcp_seq == ptcb->tcb_rnext;
        wlast = ptcb->tcb_rnext + rwindow - 1;
        rv = (ptcp->tcp_seq - ptcb->tcb_rnext) >= 0 &&
                (ptcp->tcp_seq - wlast) <= 0;
        if (seglen == 0)
                return rv;
        slast = ptcp->tcp_seq + seglen - 1;
        rv |= (slast - ptcb->tcb_rnext) >= 0 && (slast - wlast) <= 0;
```

```
        /* If no window, strip data but keep ACK, RST and URG          */
        if (rwindow == 0)
             pip->ip_len = IP_HLEN(pip) + TCP_HLEN(ptcp);
        return rv;
}
```

在“未同步状态”（CLOSED、LISTEN和SYN-SENT）时，tcpok不拒绝任何报文段。但对其他的状态，它计算报文段的长度。从概念上讲，SYN和FIN在序列空间中要占一个位置，因此无论是SYN还是FIN比特的值为1时，tcpok要将该报文段的长度值加1。一旦tcpok决定了报文段的长度，它计算接收窗口大小（rwindow）以及窗口内可接收的最大序号（wlast）。如果报文段中的数据位于可接受范围内（即在窗口以下或以内），tcpok返回TRUE。即使窗口大小为0，仍会有一些报文段被处理。因此，当窗口大小为0时，tcpok改变IP首部的长度值，使该报文段看上去根本未携带数据。由于tcpinp已经验证了校验和，因此不必再次计算。

11.11.5 为当前状态选择一个过程

一旦TCB为输入报文段找到了TCB，并验证其中的数据在已通告的窗口之内，它利用当前连接状态（可在ptcb->tcb_state中找到）来选取处理该数据段的过程。数组tcpswitch中仅包含了各个过程的地址，而且这些过程与状态一一对应。每个过程有两个入口参数：一个指向TCB表项的指针以及一个指向传入分组的指针。文件tcpswitch.c中包含了对tcpswitch数组的定义。

```
/* tcpswitch.c */

#include <conf.h>
#include <kernel.h>
#include <network.h>

char   *tcperror[] = {
       "no error",
       "connection reset",              /* TCPE_RESET        */
       "connection refused",            /* TCPE_REFUSED      */
       "not enough buffer space",       /* TCPE_TOOBIG       */
       "connection timed out",          /* TCPE_TIMEDOUT     */
       "urgent data pending",           /* TCPE_URGENTMODE   */
       "end of urgent data",            /* TCPE_NORMALMODE   */
 };
/* SEGMENT ARRIVES state processing */

typedef int (tcpin_t) (struct tcp *, struct ep *);

tcpin_t
       tcpclosed, tcplisten,tcpsynsent, tcpsynrcvd,
       tcpestablished, tcpfin1, tcpfin2, tcpclosewait,
       tcpclosing, tcplastack, tcptimewait;

tcpin_t *tcpswitch[NTCPSTATES] = {
```

```
 0,                          /* TCPS_FREE        */
 tcpclosed,                  /* TCPS_CLOSED      */
 tcplisten,                  /* TCPS_LISTEN      */
 tcpsynsent,                 /* TCPS_SYNSENT     */
 tcpsynrcvd,                 /* TCPS_SYNRCVD     */
 tcpestablished,             /* TCPS_ESTABLISHED */
 tcpfin1,                    /* TCPS_FINWAIT1    */
 tcpfin2,                    /* TCPS_FINWAIT2    */
 tcpclosewait,               /* TCPS_CLOSEWAIT   */
 tcplastack,                 /* TCPS_LASTACK     */
 tcpclosing,                 /* TCPS_CLOSING     */
 tcptimewait,                /* TCPS_TIMEWAIT    */
};
/* Output event processing */

typedef int (tcpout_t) (int tcbnum, int event);

tcpout_t tcpidle, tcppersist, tcpxmit, tcprexmt;

tcpout_t *tcposwitch[NTCPOSTATES] = {
 tcpidle,                    /* TCPO_IDLE        */
 tcppersist,                 /* TCPO_PERSIST     */
 tcpxmit,                    /* TCPO_XMIT        */
 tcprexmt,                   /* TCPO_REXMT       */
};
```

11.12　小结

TCP使用三个独立的过程来处理输入、输出及定时器功能。这些过程通过一个称为“传输控制块”（TCB）的数据结构来相互协作。TCP 为每个活动的连接保留一个独立的 TCB。

实现范例中利用过程驱动方案实现了有限状态机，其中每个状态对应一个过程。本章讲述了TCP输入过程如何处理一个输入的报文段，它利用连接的端点将该报文段在许多活动的TCB中“多路分解”（demultiplex），并利用一张表来将其切换到适当的状态过程。

深入研究

Postel［RFC 793］重点讲述了建立TCB结构的基本思想，并具体描述了许多字段。Braden［RFC 1122］更进一步地精炼了其思想。TCB 中其他许多字段来自后面章节中讨论到的 RFC。

习题

11.1 过程 tcballoc 利用顺序搜索法查找一个空闲的TCB，这意味着搜索工作量与当前 TCP 活动

的连接数目成正比。请描述一种总是以同样的时间分配TCB的实现方案。

11.2 考虑范例中TCB的字段顺序。它们是否按各自的功能分成几组？为什么？

11.3 文件tcpswitch.c中代码认为数组tcpswitch是一个指针数组，指针指向返回一个整数的函数。在Pascal中并不提供指向过程的指针，如何用类似Pascal的语言来实现tcpswitch？两种实现方案各自的利弊是什么？

11.4 文件tcp.h代码定义TCP首部中某些字段为短整型，另一些为长整型。对所有的机器来说，这样的定义是否都适用？为什么？

11.5 文件tcpfsm.h定义了TCP在最终放弃之前的最大重发次数为12。讨论它是不是一个合理的限制？

11.6 文件tcpfsm.h定义报文段的寿命为2分钟。你认为数据报能够生存2分钟以上的互联网会是什么样子？为什么？

11.7 如果应用程序使用TCP携带一串16位整数从一个机器传输到另一个机器，当它们到达时是否还是原来的16位整数？为什么？

11.8 数组tcperror（在文件tcpswitch.c中说明）中的内容是一组指针，指向每个可能的TCP差错报文的字串。将所有这些报文集中在一个数组中有什么好处？

第 12 章　TCP：有限状态机的实现

12.1　引言

第 11 章讨论了 TCP 软件的总体构成，其中每个过程对应 TCP 有限状态机中的一个状态。我们也看到 TCP 输入进程如何将输入报文段分配给各个 TCB，如何使用 TCB 中的状态变量来选取一个对应于状态机状态的过程。本章将详述每个状态过程。

12.2　CLOSED 状态处理

CLOSED 状态（关闭状态）表示 TCB 已经被分配，但还并没有开始使用。实际上，分配该 TCB 的应用程序既没有完成“主动打开”（active open）操作，也没有完成“被动打开”（passive open）操作。因此，其结果是任何输入报文段都会生成一个 TCP 复位（RESET）报文。过程 tcpclosed 实现 CLOSED 状态。它调用输出过程 tcpreset 来生成并发送复位报文。

```
/* tcpclosed.c - tcpclosed */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpclosed -  do CLOSED state processing
 *------------------------------------------------------------------------
 */
int tcpclosed(ptcb, pep)
struct tcb  *ptcb;
struct ep   *pep;
{
        tcpreset(pep);
        return     SYSERR;
}
```

12.3　从容关闭

TCP 使用修改的三次握手操作来释放连接。譬如，连接的一方称作 A 方，它通过发起“关闭”（close）操作来启动拆链。A 方的 TCP 发送一个 FIN 报文段，并转换到 FIN-WAIT-1 状态。另一方称作 B 方，当 B 方接收到该 FIN 之后，回送一个 ACK，并转换到 CLOSE-WAIT 状态，等待应用程

序释放连接。在A这一方，当它接收到ACK后，就导致TCP转换到FIN-WAIT-2状态。

当B方应用程序执行“关闭”操作后，TCP发送FIN，并转换到LAST-ACK状态。A方接收到B方发送的FIN之后，转移到TIME-WAIT状态，并发送最后一个ACK，同时释放连接。当最后的ACK抵达B方后，B方也会释放连接。后面几节将讲解处理从容关闭（graceful shutdown）的过程。

12.4 关闭后的延迟计时

因为网际协议是尽最大努力交付的传输系统，所以它的数据报可能会重复、延迟或失序传送。由于TCP允许应用程序重复使用协议端口号，因此数据报的重复和延迟，会给像TCP这一类使用IP来传送数据报的协议带来一个潜在的问题。特别是，一个丢失的确认会导致发送RESET，并使发送方误以为它的最后分组（包括FIN和数据）未曾发送。另外，如果TCP允许在连接结束后立刻重新使用这些端口号，那么在上次连接中生成的一个FIN请求的副本，可能会导致使用相同端口号的下一次连接也被终止。

为了防止复制的报文段干扰下一个连接，TCP并不会在关闭连接后立刻删除TCB。事实上，它在短时间内对该TCB不管不问。协议标准规定TCP在删除一个连接的记录之前，应该等待最大报文段寿命[1]的两倍时间长度。在我们的实现方案中，过程tcpwait安排删除一个TCB的延迟时间。与本章中大多数过程不同，tcpwait不对应于任何输入状态。事实上，其他输入状态的过程会调用它来为删除事件定时。

```
/* tcpwait.c - tcpwait */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpwait - (re)schedule a DELETE event for 2MSL from now
 *------------------------------------------------------------------------
 */
int tcpwait(ptcb)
struct tcb  *ptcb;
{
        int  tcbnum = ptcb - &tcbtab[0];

        tcpkilltimers(ptcb);
        tmset(tcps_oport, TCPQLEN, MKEVENT(DELETE, tcbnum), TCP_TWOMSL);
        return OK;
}
```

tcpwait利用第14章中描述的定时器进程。它调用tcpkilltimers删除任何与该TCB相关的正在进行的事件（例如，重传事件），同时tmset会创建一个“删除事件”（deletion event），并且在TCP_TWOMSL个单位时间后，该事件发生。当删除事件发生后，它将触发定时进程删除TCB。

① 最大报文段寿命是指报文段在底层交付系统中被丢弃之前存在的最长时间。

12.5　TIME-WAIT 状态处理

TCP 成功地完成了从容关闭后，将导致该连接进入 TIME-WAIT 状态。过程 tcptimewait 实现 TIME-WAIT 状态的处理。

```
/* tcptimewait.c - tcptimewait */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------  ------------------------------------------------------------------
 *  tcptimewait -  do TIME_WAIT state input processing
 *-  ------------------------------------------------------------------
 */
int tcptimewait(ptcb, pep)
struct tcb  *ptcb;
struct ep   *pep;
{
        struct    ip   *pip = (struct ip *)pep->ep_data;
        struct    tcp  *ptcp = (struct tcp *)pip->ip_data;

        if (ptcp->tcp_code & TCPF_RST)
             return tcbdealloc(ptcb);
        if (ptcp->tcp_code & TCPF_SYN) {
             tcpreset(pep);
             return tcbdealloc(ptcb);
        }
        tcpacked(ptcb, pep);
        tcpdata(ptcb, pep);        /* just ACK any packets */
        tcpwait(ptcb);
        return OK;
}
```

如果连接的一方收到一个 RESET 报文，则另一方必定已重新初始化了，因此 tcptimewait 释放该TCB。为防止延迟了的SYN请求再次建立连接，tcptimewait在SYN报文段到达时也发送 RESET报文。此时，如果有任何其他的报文段到达，则可能意味着某个确认被丢失，因此TCP以正常方式响应该报文段。它调用 tcpacked 处理确认报文段，且调用 tcpdata 来处理该报文段中携带的数据。最后，它调用tcpwait取消旧的删除事件，并安排新的删除事件。为每一个新报文段都重新启动定时器的结果可能会出乎意料之外：

> 由于 TCP 在每个非 SYN 报文段之后都会重新启动 TCB 删除超时定时器，因此，只要对方不断地发送报文段，TCB 将永远不会超时。

在这种情况下，使 TCB 保持原状的好处是使 TCP 能够正确地处理延迟的报文。而这种实现方

案也有缺点，如果连接的一方总是不断地发送报文段，将导致对方机器上的资源被无限期地占有。

12.6 CLOSING状态处理

TCP在接收到对方对它发出的FIN所响应的FIN报文之后进入CLOSING状态。这就是说双方都同意关闭，且TCP已经进入CLOSING状态，等待对方返送对其FIN的确认。过程tcpclosing实现CLOSING状态。

```
/* tcpclosing.c - tcpclosing */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpclosing -  do CLOSING state input processing
 *------------------------------------------------------------------------
 */
int tcpclosing(ptcb, pep)
struct tcb  *ptcb;
struct ep   *pep;
{
        struct     ip    *pip = (struct ip *)pep->ep_data;
        struct     tcp   *ptcp = (struct tcp *)pip->ip_data;

        if (ptcp->tcp_code & TCPF_RST)
             return tcbdealloc(ptcb);
        if (ptcp->tcp_code & TCPF_SYN) {
             tcpreset(pep);
             return tcbdealloc(ptcb);
        }
        tcpacked(ptcb, pep);
        if ((ptcb->tcb_code & TCPF_FIN) == 0) {
             ptcb->tcb_state = TCPS_TIMEWAIT;
             signal(ptcb->tcb_ocsem);        /* wake closer  */
             tcpwait(ptcb);
        }
        return OK;
}
```

如果tcpclosing收到一个RESET报文，它将释放TCB。如果收到的是一个SYN请求，tcpclosing会通过发送一个RESET报文并释放TCB来做出响应。对其他的报文段，tcpclosing会调用tcpacked来处理确认。在TCB的代码字段中，TCPF_FIN比特记录了对一个已被发送的FIN是否有确认到达。当用于确认该FIN的报文段到达后，tcpacked会将该比特置0。因此，tcpclosing检查该比特，如果它为0，则导致向TIME-WAIT状态的变迁。当tcpclosing变迁状态时，它将调用tcpwait删除任何正

在进行的事件，并启动删除 TCB 的定时器。

12.7　FIN-WAIT-2 状态处理

通常，当连接的一方发送 FIN 之后，另一方会立即对其加以确认，并延迟发送第二个 FIN 的时间。状态机处理状态 FIN-WAIT-2 中的时延，它由过程 tcpfin2 实现。

```
/* tcpfin2.c - tcpfin2 */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpfin2 -  do FIN_WAIT_2 state input processing
 * ------------------------------------------------------------------------
 */
int tcpfin2(ptcb, pep)
struct tcb  *ptcb;
struct ep   *pep;
{
        struct     ip    *pip = (struct ip *)pep->ep_data;
        struct     tcp   *ptcp = (struct tcp *)pip->ip_data;

        if (ptcp->tcp_code & TCPF_RST)
             return tcpabort(ptcb, TCPE_RESET);
        if (ptcp->tcp_code & TCPF_SYN) {
             tcpreset(pep);
             return tcpabort(ptcb, TCPE_RESET);
        }
        if (tcpacked(ptcb, pep) == SYSERR)
             return OK;
        tcpdata(ptcb, pep);  /* for data + FIN ACKing */

        if (ptcb->tcb_flags & TCBF_RDONE) {
             ptcb->tcb_state = TCPS_TIMEWAIT;
             tcpwait(ptcb);
        }
  return OK;
}
```

如果 tcpfin2 收到一个 RESET 报文，则调用 tcpabort 来终止连接并释放 TCB。对于到达的 SYN 报文，tcpfin2 响应一个 RESET 报文。如果对方在报文段中所携带的是数据或是一个 FIN，tcpfin2 则调用 tcpacked 对此输入加以确认，并调用 tcpdata 处理数据。

有限状态机规定，TCP 应该在收到一个 FIN 时，变迁到 TIME-WAIT 状态。然而，应该注意到，

TCP并不仅仅因为收到一个FIN比特为1的数据段，就会做出如上所述的状态变迁。事实上，为了能正确地处理不按顺序到达的数据报，它要等到整个数据序列（包括FIN）全部接收之后，才能做出决定。也就是：

> 由于TCP必须处理失序的数据传送，因而它并不立刻执行所有的状态变迁。实际上，它会延迟由FIN报文段而引起的状态变迁，直到所有的数据都被接收，并全部加以确认之后。

从实现的角度来看，如果TCP接收到了所有数据以及FIN，对tcpdata的调用将导致TCB中的TCBF_RDONE比特置1。因此，在检查它是否应该变迁到TIME-WAIT状态时，tcpfin2会检查TCB中的TCBF_RDONE比特，而不是报文段中的FIN比特。在将状态变迁到TIME-WAIT时，tcpfin2调用tcpwait删除定时器事件，并创立一个TCB删除事件。

12.8 FIN-WAIT-1 状态处理

当用户发起一个“关闭”操作之后，TCP就进入FIN-WAIT-1状态，并导致TCP发送FIN。对方可对该FIN响应一个ACK，或发送自己的FIN，或两者兼备。如果只有一个FIN到达，则说明对方已经开始释放连接，因此TCP用ACK响应，并进入CLOSING状态。如果只有一个ACK到达，则TCP进入FIN-WAIT-2状态，以等待FIN的到达。最后，如果既有FIN又有ACK到达，则TCP进入TIME-WAIT状态。过程tcpfin1实现这些状态变迁。

```
/* tcpfin1.c - tcpfin1 */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpfin1 -  do FIN_WAIT_1 state input processing
 *------------------------------------------------------------------------
 */
int tcpfin1(ptcb, pep)
struct tcb  *ptcb;
struct ep   *pep;
{
        struct    ip   *pip = (struct ip *)pep->ep_data;
        struct    tcp  *ptcp= (struct tcp *)pip->ip_data;

        if (ptcp->tcp_code & TCPF_RST)
             return tcpabort(ptcb, TCPE_RESET);
        if (ptcp->tcp_code & TCPF_SYN) {
             tcpreset(pep);
             return tcpabort(ptcb, TCPE_RESET);
        }
        if (tcpacked(ptcb, pep) == SYSERR)
             return OK;
```

```
        tcpdata(ptcb, pep);
        tcpswindow(ptcb, pep);

        if (ptcb->tcb_flags & TCBF_RDONE) {
             if (ptcb->tcb_code & TCPF_FIN)        /* FIN not ACKed*/
                  ptcb->tcb_state = TCPS_CLOSING;
             else {
                  ptcb->tcb_state = TCPS_TIMEWAIT;
                  signal(ptcb->tcb_ocsem);   /* wake closer  */
                  tcpwait(ptcb);
             }
        } else if ((ptcb->tcb_code & TCPF_FIN) == 0) {
             signal(ptcb->tcb_ocsem);        /* wake closer  */
             ptcb->tcb_state = TCPS_FINWAIT2;
        }
        return OK;
}
```

如果tcpfin1接收到一个RESET报文，则立即终止连接，并且也发送一个RESET报文来响应接收到的SYN。一般说来，TCP此时仍然应当处理正在输入的数据，因而它调用tcpacked处理接收到的确认，调用tcpdata处理报文段中的数据，并且调用tcpswindow来调整自己的发送窗口大小。一旦输入处理完毕，tcpfin1查看是否有必要变迁状态。如果TCBF_RDONE比特的值为1，且FIN也已到达，那么表示所有发送序列中，FIN以前的数据已经全部到达。如果TCPF_FIN比特为0，则表示对刚才发送的FIN，对方响应的ACK已到达。tcpfin1利用这两个比特来决定是否应该变迁到CLOSING状态，还是进入FIN-WAIT-2或TIME-WAIT状态。在进入TIME-WAIT状态时，它必须调用tcpwait为一个TCB删除事件定时。当外发的FIN已被确认后，tcpfin1释放信号量ocsem，使得释放该连接的应用程序能够完成关闭操作。如果已有多个应用程序访问该TCB，则必须在最后一个应用程序发出“关闭”操作后，才能删除该TCB。

12.9　CLOSE-WAIT 状态处理

到目前为止，我们已经看到了当一个应用程序通过发起“关闭”操作来启动关闭动作时，那些与释放连接有关的状态是如何变迁的。但是，如果在应用程序发起“关闭”操作之前，有一个FIN到达，则TCP进入CLOSE-WAIT状态。它利用文件结尾标志来通知应用程序，说明对方已经释放连接，并在进入LAST-ACK状态之前，等待应用程序发起“关闭”操作。

当TCP在CLOSE-WAIT状态中等待时，利用过程tcpclosewait来处理输入的报文段。

```
/* tcpclosewait.c - tcpclosewait */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
```

```
 *  tcpclosewait -  do CLOSE_WAIT state input processing
 *------------------------------------------------------------------------
 */
int tcpclosewait(ptcb, pep)
struct tcb   *ptcb;
struct ep    *pep;
{
        struct     ip    *pip = (struct ip *)pep->ep_data;
        struct     tcp   *ptcp = (struct tcp *)pip->ip_data;

        if (ptcp->tcp_code & TCPF_RST) {
             TcpEstabResets++;
             TcpCurrEstab--;
             return tcpabort(ptcb, TCPE_RESET);
        }
        if (ptcp->tcp_code & TCPF_SYN) {
             TcpEstabResets++;
             TcpCurrEstab--;
             tcpreset(pep);
             return tcpabort(ptcb, TCPE_RESET);
        }
        tcpacked(ptcb, pep);
        tcpswindow(ptcb, pep);
        return OK;
}
```

如果到达的是一个RESET报文，tcpclosewait调用tcpabort终止这个连接，并删除该TCB。如果是一个SYN到达，tcpclosewait生成RESET报文，并终止该连接。最后，它调用tcpacked来处理确认报文，并调用tcpswindow来更新发送窗口大小。

12.10 LAST-ACK状态处理

当应用程序发起“关闭”操作后，TCP从CLOSE-WAIT状态变迁到LAST-ACK状态。在此期间，TCP会安排一个待发送的FIN报文，并进入LAST-ACK状态以等待确认。FIN将在发送完所有剩余数据后被发送，如果接收方已经关闭了自己的窗口，这些数据可能被延迟。一旦TCP发送了FIN，它将继续为重发事件定时。如果在正常的重发超时时间内没有确认报文到达，则TCP将重发该FIN。过程tcplastack实现LAST-ACK状态的处理。

```
/* tcplastack.c - tcplastack */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
```

```
 *  tcplastack -  do LAST_ACK state input processing
 *------------------------------------------------------------------------
 */
int tcplastack(ptcb, pep)
struct tcb   *ptcb;
struct ep    *pep;
{
        struct     ip    *pip = (struct ip *)pep->ep_data;
        struct     tcp   *ptcp = (struct tcp *)pip->ip_data;

        if (ptcp->tcp_code & TCPF_RST)
             return tcpabort(ptcb, TCPE_RESET);
        if (ptcp->tcp_code & TCPF_SYN) {
             tcpreset(pep);
             return tcpabort(ptcb, TCPE_RESET);
        }
        tcpacked(ptcb, pep);
        if ((ptcb->tcb_code & TCPF_FIN) == 0)
             signal(ptcb->tcb_ocsem);   /* close() deallocs  */
        return OK;
}
```

如果一个RESET到达，tcplastack调用tcpabort终止该连接。如果一个SYN到达，它发送RESET并终止连接。对于其他情况，tcplastack调用tcpacked来处理输入的确认，并释放打开/关闭信号量，以便一旦发出的FIN被确认后，应用程序能够完成它们的“关闭”操作。

12.11　ESTABLISHED 状态处理

一旦某个连接被建立，双方在交换数据和确认报文时都处于ESTABLISHED状态。TCP调用过程tcpestablished来处理在该状态下到达的任何报文段。

```
/* tcpestablished.c - tcpestablished */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpestablished -  do ESTABLISHED state input processing
 *------------------------------------------------------------------------
 */
int tcpestablished(ptcb, pep)
struct tcb   *ptcb;
struct ep    *pep;
{
```

```
        struct    ip    *pip = (struct ip *)pep->ep_data;
        struct    tcp   *ptcp = (struct tcp *)pip->ip_data;

        if (ptcp->tcp_code & TCPF_RST) {
             TcpEstabResets++;
             TcpCurrEstab--;
             return tcpabort(ptcb, TCPE_RESET);
        }
        if (ptcp->tcp_code & TCPF_SYN) {
             TcpEstabResets++;
             TcpCurrEstab--;
             tcpreset(pep);
             return tcpabort(ptcb, TCPE_RESET);
        }
        if (tcpacked(ptcb, pep) == SYSERR)
             return OK;
        tcpdata(ptcb, pep);
        tcpswindow(ptcb, pep);
        if (ptcb->tcb_flags & TCBF_RDONE)
             ptcb->tcb_state = TCPS_CLOSEWAIT;
        return OK;
}
```

如果接收到的是一个RESET，则意味着对方必定已经重启动，并且丢失了有关这条连接的信息。因此，tcpestablisehed调用tcpabort立即终止连接。如果一个SYN到达，tcpestablished发送一个RESET，并终止该连接，否则，它调用tcpacked处理输入的确认，调用tcpdata来检查FIN比特，从报文段中提取数据，并且，如果该报文段中含有一个新窗口通告，则调用tcpswindow来更新发送窗口大小。如果一个FIN到达，且FIN之前的所有数据也已被接收，过程tcpdata的调用将把TCB标志字段中的TCBF_RDONE比特置为1。tcpestablished利用该比特决定是否应该进入CLOSE-WAIT状态。

12.12 处理报文段中的紧急数据

在ESTABLISHED状态下，TCP必须接受来自输入报文段中的数据，并利用这些数据来填写接收缓冲区，计算新窗口大小，并发送一个确认。过程tcpdata处理数据接收的细节。

```
/* tcpdata.c - tcpdata */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpdata -  process an input segment's data section
 *------------------------------------------------------------------------
```

```
 */
int
tcpdata(struct tcb *ptcb, struct ep *pep)
{
        struct  ip      *pip = (struct ip *)pep->ep_data;
        struct  tcp     *ptcp = (struct tcp *)pip->ip_data;
        tcpseq          first, last, wlast;
        unsigned        pb;
        int             datalen, rwindow, i, pp;

        if (ptcp->tcp_code & TCPF_URG) {
                int     rup = ptcp->tcb_seq + ptcp->tcp_urgptr;
#ifdef BSDURG
                rup--;
#endif
                if (! (ptcb->tcb_flags & TCBF_RUPOK) ||
                    SEQCMP(rup, ptcb->tcb_rupseq) > 0) {
                        ptcb->tcb_rupseq = rup;
                        ptcb->tcb_flags |= TCBF_RUPOK;
                }
        }
        if (ptcp->tcp_code & TCPF_SYN) {
                ptcb->tcb_rnext++
                ptcb->tcb_flags |= TCBF_NEEDOUT;
                ++ptcp->tcp_seq;        /* so we start with data */
        }
        datalen = pip->ip_len - IP_HLEN(pip) - TCP_HLEN(ptcp);
        rwindow =ptcb->tcb_rbsize - ptcb->tcb_rbcount;
        wlast = ptcb->tcb_rnext + rwindow-1;
        first = ptcp->tcp_seq;
        last = first + datalen -1;
        if (SEQCMP(ptcb->tcb_rnext, first) > 0) {
                datalen -= ptcb->tcb_rnext - first;
                first = ptcb->tcb_rnext;
        }
        if (SEQCMP(last, wlast) > 0) {
                datalen -= last - wlast;
                ptcp->tcp_code &= ~TCPF_FIN; /* cutting it off          */
        }
        pb = ptcb->tcb_rbstart + ptcb->tcb_rbcount; /* == rnext, in buf */
        pb += first - ptcb->tcb_rnext;          /* distance in buf         */
        pb %= ptcb->tcb_rbsize;                 /* may wrap                */
        pp = first - ptcp->tcp_seq;             /* distance in packet      */
        for (i=0; i<datalen; ++i) {
                ptcb->tcb_rcvbuf[pb] = ptcp->tcp_data[pp++];
```

```
                if (++pb >= ptcb->tcb_rbsize)
                        pb = 0;
        }
        tcpdodat(ptcb, ptcp, first, datalen);      /* deal with it      */
        if (ptcb->tcb_flags & TCBF_NEEDOUT)
                tcpkick(ptcb);
        return OK;
}
```

tcpdata首先要查看报文段中是否有紧急数据。它利用掩码TCPF_URG来查看代码字段中的紧急比特。如果该比特为1，则该报文段的字段tcp_urgptr中含有有效的紧急数据指针。tcpdata提取该指针并计算指针指向的一列数据所处的位置。虽然协议标准规定，紧急数据指针给出的是紧急数据的结束位置，但从BSD UNIX派生出的实现方案，却把该指针解释为指向该紧急数据外的一个字节位置。为了保证两类实现的兼容性，代码中包括了一个配置常量BSDURG。当使用BSD去解释紧急数据指针时，tcpdata将接收到的紧急数据指针值rup减去1。

在计算出紧急数据的序列值后，tcpdata将这个信息记录在TCB中。它利用掩码TCBF_RUPOK来查看TCB的“接收紧急数据指针”（receive urgent pointer）是否已经被置位。协议标准规定，如果有多个携带紧急数据的报文段到达，应用程序必须立刻接收所有紧急数据。因此如果TCB中已没有未处理完的紧急数据，或者新的紧急数据指针提供的序列值比现有的指针更大，则tcpdata将该紧急数据指针的序列空间值记录下来，并将标志位TCBF_RUPOK置1，表示该指针是有效的。

12.13 处理报文段中的其他数据

在处理完紧急数据之后，tcpdata检查输入报文段中的SYN位。从概念上看，一个SYN在输入数据序列空间中占据一个位置，因此如果该报文段含有一个SYN，则tcpdata将TCB中的序号加1。

为了处理输入报文段中的数据，tcpdata既要计算它的长度（datalen），还要计算缓冲区中剩余空间量（rwindow）。然后tcpdata计算接收缓冲区中数据起始位置的索引值（pb），以及报文段的数据区索引值（pp）。它认为接收缓冲区是一个环形数组，并从报文段复制datalen个八位组到缓冲区中，如果缓冲区索引值超过缓冲区大小，则绕回缓冲区的头部继续。在向接收缓冲区中复制数据之后，tcpdata调用过程tcpdodat来完成处理，并且如果需要输出，则调用过程tcpkick来启动输出（例如返回一个确认）。

过程tcpdodat处理以下几个细节。

```
/* tcpdodat.c - tcpdodat */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpdodat -  do input data processing
 *------------------------------------------------------------------------
 */
```

```
int tcpdodat(ptcb, ptcp, first, datalen)
struct tcb  *ptcb;
struct tcp  *ptcp;
tcpseq      first;
unsigned    datalen;
{
        int  wakeup = 0;

        if (ptcb->tcb_rnext == first) {
                if (datalen > 0) {
                        tfcoalesce(ptcb, datalen, ptcp);
                        ptcb->tcb_flags |= TCBF_NEEDOUT;
                        wakeup++;
                }
                if (ptcp->tcp_code & TCPF_FIN) {
                        ptcb->tcb_flags |= TCBF_RDONE|TCBF_NEEDOUT;
                        ptcb->tcb_rnext++;
                        wakeup++;
                }
                if (ptcp->tcp_code & (TCPF_PSH | TCPF_URG)) {
                        ptcb->tcb_flags |= TCBF_PUSH;
                        wakeup++;
                }
                if (wakeup)
                        tcpwakeup(READERS, ptcb);
        } else {
                /* process delayed controls */
                if (ptcp->tcp_code & TCPF_FIN)
                        ptcb->tcb_finseq = ptcp->tcp_seq + datalen;
                if (ptcp->tcp_code & (TCPF_PSH | TCPF_URG))
                        ptcb->tcb_pushseq = ptcp->tcp_seq + datalen;
                ptcp->tcp_code &= ~(TCPF_FIN|TCPF_PSH);
                tfinsert(ptcb, first, datalen);
        }
        return OK;
}
```

如果一个输入报文段中的数据是紧接在已成功地接收到的连续数据序列之后，tcpdodat必须立刻处理控制标志。首先，它将TCB中的输出标志字段（tcb_flags）置位，以便能生成一个确认。它调用过程tfcoalesce来决定该数据是否应该被用来填补那些由于报文段失序到达而形成的序列空间中的空缺部分。其次，如果报文段中除了数据之外，还含有一个FIN，则tcpdodat将该FIN也作为序列空间中的一项来计数，并将输出标志置位，以表示有一个FIN已到达，且需要发出一个ACK。第三，如果该输入报文段的push位为1，则tcpdodat要设置一个标志，以表明该报文段请求推处理功能。无论在哪种情况下，如果tcpdodat认为新数据是有效的，则它调用tcpwakeup来唤醒任何因等待该数据到达而被阻塞的应用程序进程。

如果一个报文段没有按顺序到达，则tcpdodat必须处理对延迟的控制。例如，一个携带FIN的报文段有可能在最后一个携带数据的报文段之前到达。在诸如此类的情况下，TCP已经在TCB中保存了有关FIN的信息，因此如果差缺的数据到达，tcpdodat能够执行被延迟的处理过程。tcpdodat调用tfinsert记录下接收到的八位组。

12.14 经常注意已接收的八位组

TCP必须能够正确处理失序传送。由于窗口的通告是根据已分配的缓冲区限制了输入数据量，因而TCP总是能够把到达的数据直接复制到缓冲区中。虽然如此，TCP还是必须保存一份记录，指出已接收的数据序列中的哪些八位组已被接收。要做到这一点，TCP保留一份由（序号，长度）这一序偶组成的链表，表中每一对值对应一个活动的TCB。

借用IP中的术语，实现方案称表中的项目为分片（fragment）。在TCP分片表中的每一个表项，代表一个已接收到的报文段中的数据，每个表项中包含了第一个八位组的序号和一个长度。该结构由文件tcb.h中的结构tcpfrag定义。每当有数据没有按顺序到达时，tcp就调用过程tfinsert向分片表中插入一个表项。

```
/* tfinsert.c - tfinsert */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <mem.h>
#include <q.h>

/*------------------------------------------------------------------------
 *  tfinsert - add a new TCP segment fragment to a TCB sequence queue
 *------------------------------------------------------------------------
 */
int tfinsert(ptcb, seq, datalen)
struct tcb   *ptcb;
tcpseq       seq;
unsigned     datalen;
{
        struct    tcpfrag    *tf;

        if (datalen == 0)
             return OK;
        tf = (struct tcpfrag *)getmem(sizeof(struct tcpfrag));
        tf->tf_seq = seq;
        tf->tf_len = datalen;
        if (ptcb->tcb_rseqq < 0)
             ptcb->tcb_rseqq = newq(NTCPFRAG, QF_WAIT);
        if (enq(ptcb->tcb_rseqq, tf, -tf->tf_seq) < 0)
```

```
            freemem(tf, sizeof(struct tcpfrag));
        return OK;
}
```

tfinsert有三个入口参数，分别用来描述已到达的数据在整体数据中的位置。首先它为该数据分配一个新结点，然后将该结点链接到分片表上。这三个参数由一个指向TCB的指针、一个初始序号和数据长度组成。如果第一个到达的数据就没有按顺序，此时该连接还没有对应的队列存在，因此tfinsert调用newq来创建一个队列。无论在哪种情况下，它都要调用enq向队列中插入一个新结点。

除了为接收到的数据增加表项，用以记录其序号之外，TCP还必须递增计数器的值，以指明该序列空间中已有多少连续的八位组被成功接收。其方法是，TCP沿着序列空间移动一个指针，直到它发现下一个“空穴”（hole）为止。过程tfcoalesce实现该操作。

```
/* tfcoalesce.c - tfcoalesce */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <mem.h>

/*------------------------------------------------------------------------
 *  tfcoalesce -  join TCP fragments
 *------------------------------------------------------------------------
 */
int tfcoalesce(ptcb, datalen, ptcp)
struct tcb   *ptcb;
unsigned     datalen;
struct tcp   *ptcp;
{
        struct tcpfrag  *tf = 0;
        int             newcount;

        ptcb->tcb_rnext += datalen;
        ptcb->tcb_rbcount += datalen;
        if (ptcb->tcb_rnext == ptcb->tcb_finseq)
            goto alldone;
        if ((ptcb->tcb_rnext - ptcb->tcb_pushseq) >= 0) {
            ptcp->tcp_code |= TCPF_PSH;
            ptcb->tcb_pushseq = 0;
        }
        if (ptcb->tcb_rseqq < 0)   /* see if this closed a hole */
            return OK;
        tf = (struct tcpfrag *)deq(ptcb->tcb_rseqq);
        if(tf == 0)
            return OK;
        while ((tf->tf_seq - ptcb->tcb_rnext) <= 0) {
            newcount = tf->tf_len - (ptcb->tcb_rnext - tf->tf_seq);
```

```
        if (newcount > 0) {
            ptcb->tcb_rnext += newcount;
            ptcb->tcb_rbcount += newcount;
        }
        if (ptcb->tcb_rnext == ptcb->tcb_finseq)
            goto alldone;
        if ((ptcb->tcb_rnext - ptcb->tcb_pushseq) >= 0) {
            ptcp->tcp_code |= TCPF_PSH;
            ptcb->tcb_pushseq = 0;
        }
        freemem(tf, sizeof(struct tcpfrag));
        tf = (struct tcpfrag *)deq(ptcb->tcb_rseqq);
        if (tf == 0) {
            freeq(ptcb->tcb_rseqq);
            ptcb->tcb_rseqq = EMPTY;
            return OK;
        }
    }
    enq(ptcb->tcb_rseqq, tf, -tf->tf_seq); /* got one too many     */
    return OK;
alldone:
    if (tf)
        freemem(tf, sizeof(struct tcpfrag));
    while (tf = (struct tcpfrag *)deq(ptcb->tcb_rseqq));
        freemem(tf, sizeof(struct tcpfrag));
    freeq(ptcb->tcb_rseqq);
    ptcb->tcb_rseqq = EMPTY;
    ptcp->tcp_code |= TCPF_FIN;
    return OK;
}
```

tfcoalesce的主循环遍历整个TCP分片表。在循环开始之前，它删除了第一个表项，然后每循环遍历一次就删除另外一个表项。在每次遍历时，tfcoalesce查看被删除的表项是否扩展了当前接收到的完整的序列空间。该项检查很简单：只有当一个新表项的初始序号在当前序列之内或紧邻当前序列时，它才会扩展现有的序列空间（字段tcb_rnext）。

在遍历过程中，如果字段tcb_rnext的值等于FIN的序号，则tfcoalesce声明该输入已完成，并跳转到标号alldone处运行，以删除该链表。如果循环完成，而链表中仍有表项存在，则tfcoalesce必须将上一次从链表上取下的表项重新插入链表中。

12.15 终止一个TCP连接

正如我们已经看到的，在几个状态过程中需要立即终止一个TCP连接。要做到这一点，它们调用过程tcpabort，向其传递一个指针参数，指向将被释放的TCB，以及一个整数参数，用于解释

终止的原因。

```
/* tcpabort.c - tcpabort */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpabort -  abort an active TCP connection
 *------------------------------------------------------------------------
 */
int tcpabort(ptcb, error)
struct tcb  *ptcb;
int         error;
{
        tcpkilltimers(ptcb);
        ptcb->tcb_flags |= TCBF_RDONE|TCBF_SDONE;
        ptcb->tcb_error = error;
        tcpwakeup(READERS|WRITERS, ptcb);
        return OK;
}
```

tcpabort 利用tcpkilltimers 删除所有与该连接相关的正在进行的事件。此外，它将标志字段中的所有位均置 1，表明接收和传送都已完成。tcpabort 将参数 error 保存在 TCB 中，以指出是何种差错导致连接被终止。最后，tcpabort 调用tcpwakeup来唤醒任何因等待 I/O 而被阻塞的“读”进程或“写”进程。每个等待读或写操作的应用程序都将被唤醒，并查找出保存在 TCB 中的差错类型。

12.16　建立 TCP 连接

回想一下，TCP 使用了三次握手操作来建立一个连接。当服务器发起“被动打开”操作，并在 LISTEN 状态中等待时，客户则发起“主动打开”操作，并进入 SYN-SENT 状态。于是服务器进入 SYN-RECEIVED状态。最终，客户和服务器都进入ESTABLISHED状态。后面几节中讲述在建立连接过程中用到的状态过程。

12.17　初始化 TCB

当应用程序发起“主动打开”或“被动打开”操作时，TCP 调用过程tcpsync来初始化一个TCB。

```
/* tcpsync.c - tcpsync */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```

```
#include <mem.h>

/*------------------------------------------------------------------------
 *  tcpsync - initialize TCB for a new connection request
 *------------------------------------------------------------------------
 */
int tcpsync(ptcb)
struct tcb  *ptcb;
{
        ptcb->tcb_state = TCPS_CLOSED;
        ptcb->tcb_type = TCPT_CONNECTION;

        ptcb->tcb_iss = ptcb->tcb_suna = ptcb->tcb_snext = tcpiss();
        ptcb->tcb_lwack = ptcb->tcb_iss;

        ptcb->tcb_sndbuf = (u_char *)getmem(TCPSBS);
        ptcb->tcb_sbsize = TCPSBS;
        ptcb->tcb_sbstart = ptcb->tcb_sbcount = 0;
        ptcb->tcb_ssema = screate(1);

        ptcb->tcb_rcvbuf = (u_char *)getmem(TCPRBS);
        ptcb->tcb_rbsize = TCPRBS;
        ptcb->tcb_rbstart = ptcb->tcb_rbcount = 0;
        ptcb->tcb_rsegq = EMPTY;
        ptcb->tcb_rsema = screate(0);
        ptcb->tcb_ocsem = screate (0);

        /* timer stuff */

        ptcb->tcb_srt = 0;              /* in sec/100   */
        ptcb->tcb_rtde = 0;             /* in sec/100   */
        ptcb->tcb_rexmt = 50;           /* in sec/100   */
        ptcb->tcb_rexmtcount = 0;
        ptcb->tcb_keep = 12000;         /* in sec/100   */

        ptcb->tcb_code = TCPF_SYN;
        ptcb->tcb_flags = 0;
        return OK;
}
```

tcpsync初始化连接，使之进入CLOSED状态，然后分配发送和接收缓冲区，创建发送和接收信号量，并初始化各类计数器及重发时延估值。最后，它将tcb_code字段中的TCPF_SYN位置1，这表示应该发送一个SYN。

12.18　SYN-SENT 状态处理

一旦TCP发出SYN请求，就进入SYN-SENT状态，过程tcpsynsent实现SYN-SENT状态的处理。

```
/* tcpsynsent.c - tcpsynsent */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpsynsent -  do SYN_SENT state processing
 *------------------------------------------------------------------------
 */
int tcpsynsent(ptcb, pep)
struct tcb  *ptcb;
struct ep   *pep;
{
        struct     ip    *pip = (struct ip *)pep->ep_data;
        struct     tcp   *ptcp = (struct tcp *)pip->ip_data;

        if ((ptcp->tcp_code & TCPF_ACK) &&
            ((ptcp->tcp_ack - ptcb->tcb_iss <= 0) ||
            (ptcp->tcp_ack - ptcb->tcb_snext) > 0))
             return tcpreset(pep);
        if (ptcp->tcp_code & TCPF_RST) {
             ptcb->tcb_state = TCPS_CLOSED;
             ptcb->tcb_error = TCPE_RESET;
             TcpAttemptFails++;
             tcpkilltimers(ptcb);
             signal(ptcb->tcb_ocsem);
             return OK;
        }
        if ((ptcp->tcp_code & TCPF_SYN) == 0)
             return OK;
        ptcb->tcb_swindow = ptcp->tcp_window;
        ptcb->tcb_lwseq = ptcp->tcp_seq;
        ptcb->tcb_rnext = ptcp->tcp_seq;
        ptcb->tcb_cwin = ptcb->tcb_rnext + ptcb->tcb_rbsize;
        tcpacked(ptcb, pep);
        tcpdata(ptcb, pep);
        ptcp->tcp_code &= ~TCPF_FIN;
        if (ptcb->tcb_code & TCPF_SYN)        /* our SYN not ACKed */
             ptcb->tcb_state = TCPS_SYNRCVD;
        else {
```

```
            TcpCurrEstab++;
            ptcb->tcb_state = TCPS_ESTABLISHED;
            signal(ptcb->tcb_ocsem);        /* return in open    */
        }
        return OK;
    }
```

如果有一个ACK到达，tcpsynsent检查该ACK，以保证其指明的是正确的序号。如果不是，则发送一个RESET。如果到达的是RESET，tcpsynsent进入CLOSED状态并调用tcpkilltimers删除所有正在进行的事件。如果该传入报文段中含有一个SYN，其中可能还含有其他数据或对上一次发出的SYN的确认，因此tcpsynsent调用tcpacked和tcpdata来处理该报文段。最后，tcpsynsent检查TCB中的TCPF_SYN比特，看是否该连接的SYN已被确认。如果该SYN已被确认，则tcpsynsent将连接状态变迁为ESTABLISHED状态。否则，进入“SYN-RECEIVED”状态。

12.19 SYN-RECEIVED状态处理

无论是由于对方发送的SYN的到达而启动了一个三次握手操作，还是由于在SYN-SENT状态下的连接收到一个没有被确认的SYN，TCP都将连接状态变迁为SYN-RECEIVED状态。过程tcpsynrcvd处理处于SYN-RECEIVED状态的连接上的输入报文段。

```
/* tcpsynrcvd.c - tcpsynrcvd */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ports.h>

/*------------------------------------------------------------------------
 *  tcpsynrcvd -  do SYN_RCVD state input processing
 *------------------------------------------------------------------------
 */
int tcpsynrcvd(ptcb, pep)
struct tcb   *ptcb;
struct ep    *pep;
{
        struct     ip    *pip = (struct ip *)pep->ep_data;
        struct     tcp   *ptcp = (struct tcp *)pip->ip_data;
        struct     tcb   *pptcb;

        if (ptcp->tcp_code & TCPF_RST) {
             TcpAttemptFails++;
             if (ptcb->tcb_pptcb != 0)
                  return tcbdealloc(ptcb);
             else
                  return tcpabort(ptcb, TCPE_REFUSED);
```

```
        }
        if (ptcp->tcp_code & TCPF_SYN) {
            TcpAttemptFails++;
            tcpreset(pep);
            return tcpabort(ptcb, TCPE_RESET);
        }
        if (tcpacked(ptcb, pep) == SYSERR)
            return OK;
        if (ptcb->tcb_pptcb != 0) {          /* from a passive open    */
            pptcb = ptcb->tcb_pptcb;
            if (wait(pptcb->tcb_mutex) != OK) {
                TcpAttemptFails++;
                tcpreset(pep);
                return tcbdealloc(ptcb);
            }
            if (pptcb->tcb_state != TCPS_LISTEN) {
                TcpAttemptFails++;
                tcpreset(pep);
                signal(pptcb->tcb_mutex);
                return tcbdealloc(ptcb);
            }
            if (pcount(pptcb->tcb_listenq) >= pptcb->tcb_lqsize) {
                TcpAttemptFails++;
                tcpreset(pep);
                signal(pptcb->tcb_mutex);
                return tcbdealloc(ptcb);
            }
            psend(pptcb->tcb_listenq, ptcb->tcb_dvnum);
            signal(pptcb->tcb_mutex);
        } else                               /* from an active open    */
            signal(ptcb->tcb_ocsem);
        TcpCurrEstab++;
        ptcb->tcb_state = TCPS_ESTABLISHED;
        tcpdata(ptcb, pep);
        if (ptcb->tcb_flags & TCBF_RDONE)
            ptcb->tcb_state = TCPS_CLOSEWAIT;
        return OK;
    }
```

如果一个 RESET 到达，tcpsynrcvd 终止连接并释放 TCB。对于被动打开的连接，TCB 是其“父 TCB”（parentTCB）的一个独立的副本，因此只需调用 tcpdealloc 删除该“子 TCB”（orphan TCB）。然而对主动打开的连接，tcpsynrcvd 需调用 tcpabort，它将记录下 TCB 中的差错。然后，如果有 SYN 到达，它还要终止该连接。

由于 TCP 仅在响应了一个 SYN 之后才进入 SYN-RECEIVED 状态，任何不是 RESET 或 SYN 的报文段都意味着对方认为该连接已建立。因此，当一个报文段到达时，tcpsynrcvd 调用 tcpacked 来

进行确认处理，并进入ESTABLISHED状态，然后调用tcpdata从报文段中提取数据。

tcpsynrcvd还要处理服务器与为特定连接而运行的进程之间的部分转换。我们将会看到，当服务器发生被动打开操作后，就会创建一个监听队列（listen queue）。然后服务器进入循环，从监听队列中提取下一个连接，并创建一个进程来处理该连接，总结如下：

> 服务器使用被动打开为某个TCP端口创建一个连接队列。TCP为每个新连接分配一个新TCB，将连接标识符放在对应于该端口的监听队列中，并唤醒服务器进程，使之能够处理新连接。

对于被动打开，tcpsynrcvd将tcb_dvnum置入该服务器的监听队列，tcb_dvnum是服务器要用到的连接描述符。为了避免阻塞输入进程，它使用pcount来确保队列中还有空间，并调用psend将连接标识符置入监听队列中。对于主动的打开，tcpsynrcvd释放打开/关闭信号量，以允许进一步处理主动打开。不论在哪种情况下，tcpsynrcvd将该连接的状态变迁为ESTABLISHED状态，并在有数据存在的情况下调用tcpdata从报文段中提取数据。最后它检查tcb_flag字段，看是否有FIN到达（可能未按顺序，也可能与SYN在同一报文段中），如果有，则进入CLOSE-WAIT状态。

12.20 LISTEN状态处理

服务器在LISTEN状态下等待来自客户的连接，它是最复杂的状态之一，因为它要为每个输入的连接创建一个新TCB。过程tcplisten提供其实现方案。

```
/* tcplisten.c - tcplisten */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcplisten -  do LISTEN state processing
 *------------------------------------------------------------------------
 */
int tcplisten(ptcb, pep)
struct tcb   *ptcb;
struct ep    *pep;
{
        struct     tcb   *newptcb, * tcballoc();
        struct     ip    *pip = (struct ip *)pep->ep_data;
        struct     tcp   *ptcp = (struct tcp *)pip->ip_data;

        if (ptcp->tcp_code & TCPF_RST)
             return OK;            /* "parent" TCB still in LISTEN */
        if ((ptcp->tcp_code & TCPF_ACK) ||
            (ptcp->tcp_code & TCPF_SYN) == 0)
             return tcpreset(pep);
        newptcb - tcballoc();
        if ((int)newptcb == SYSERR || tcpsync(newptcb) == SYSERR)
```

```
        return SYSERR;
    newptcb->tcb_state = TCPS_SYNRCVD;
    newptcb->tcb_ostate = TCPO_IDLE;
    newptcb->tcb_error = 0;
    newptcb->tcb_pptcb = ptcb;          /* for ACCEPT              */

    newptcb->tcb_rip = pip->ip_src;
    newptcb->tcb_rport = ptcp->tcp_sport;
    newptcb->tcb_lip = pip->ip_dst;
    newptcb->tcb_lport = ptcp->tcp_dport;

    tcpwinit(ptcb, newptcb, pep);   /* initialize window data       */

    newptcb->tcb_finseq = newptcb->tcb_pushseq = 0;
    newptcb->tcb_flags = TCBF_NEEDOUT;
    TcpPassiveOpens++;
    tcpdata(newptcb, pep);
    signal(newptcb->tcb_mutex);
    return OK;
}
```

tcplisten开始时的情况正如我们所料。它忽略RESET请求，因为这时还没有连接存在。tcplisten为所有不是SYN的传入报文段发送RESET。

一旦接收到一个SYN后，tcplisten必须调用tcpalloc为这条新连接创建一个TCB。我们称原TCB为父，新TCB为子。tcplisten调用tcpsync来初始化子TCB中的字段，然后将其变迁到SYN-RECEIVED状态。同时，父TCB仍处在LISTEN状态中。在初始化新TCB时的一个重要步骤是从接收到的报文段中将发送方的IP地址和协议端口号复制到新创建的TCB中。此后，tcplisten调用tcpwinit来完成新TCB的窗口信息初始化工作。

tcplisten不会为新连接明确地生成SYN或ACK响应。事实上，它将tcb_flags字段中的TCBF_NEEDOUT位置1，表示需要输出，并调用tcpdata处理报文段中的任何数据，然后开始输出。最后，tcplisten为新的TCB释放互斥信号量，以允许TCP的其他组成部分可以使用该信号量（如，接受输入或生成确认报文）。

12.21　为一个新TCB初始化窗口变量

过程tcpwinit初始化用于控制窗口大小和报文段长度的变量。

```
/* tcpwinit.c - tcpwinit*/

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpwinit - initialize window data for a new connection
```

```
 *------------------------------------------------------------------------
 */
int
tcpwinit(struct tcb *ptcb, struct tcb *newptcb, struct ep *pep)
{
        struct  ip      *pip = (struct ip *)pep->ep_data;
        struct  tcb     *ptcp = (struct tcp *)pip->ip_data;
        struct  route   *prt;
        unsigned        mss;
        Bool            local;

        newptcb->tcb_swindow = ptcp->tcp_window;
        newptcb->tcb_lwseq = ptcp->tcp_seq;
        newptcb->tcb_lwack = newptcb->tcp_iss;    /* set in tcpsync()  */

        prt = (struct route *)rtget(pip->ip_src, RTF_REMOTE);
        local = prt && prt->rt_metric == 0
        newptcb->tcb_pni = &nif[prt->rt_ifnum];
        rtfree(prt);
        if (local)
                mss = newptcb->tcb_pni->ni_mtu-IPMHLEN-TCPMHLEN;
        else
                mss = 536;    /* RFC 1122 */
        if (ptcb->tcb_smss) {
                newptcb->tcb_smss = min(ptcb->tcb_smss, mss);
                ptcb->tcb_smss = 0;                /* reset server smss */
        } else
                newptcb->tcb_smss = mss;
        newptcb->tcb_rmss = mss;                   /* receive mss       */
        newptcb->tcb_cwnd = newptcb->tcb_smss;     /* 1 segment         */
        newptcb->tcb_ssthresh = 65535;             /* IP max window     */
        newptcb->tcb_rnext = ptcp->tcp_seq;
        newptcb->tcb_cwin = newptcb->tcb_rnext + newptcb->tcb_rbsize;
}
```

因为第15章将讨论拥塞控制以及用于实现拥塞控制的窗口大小限制，所以在阅读第15章之前，大多数变量的初始化都不容易理解。不过有关最大报文段长度的选择还是很有趣的，而且也不难理解。

TCP协议标准规定，当TCP与那些没有直接相连的网络中的目的站通信时，应当使用536八位组这个默认的最大报文段长度（Maximum Segment Size，MSS）[①]。然而当它与直接相连的网络上的目的站通信时，TCP可使用网络MTU来计算最佳MSS值。要做到这一点，tcpwinit调用rtget来找出

① 最大报文段长度的计算是将默认IP报文段长度（576八位组）减去一个最小IP首部的八位组数（20八位组），及一个最小TCP首部的八位组数（20八位组）。

到达远程终点的路由。如果路由度量为0，则目的站就在直接相连的网络内，因此TCP通过从网络MTU中减去TCP和IP首部的长度来计算MSS的值。如果远程机器上的TCP规定了MSS值，则tcpwinit使用规定的MSS和由网络MTU计算出的MSS之中的较小值。无论在哪种情况下，tcpwinit处理输入时都要使用由网络MTU计算出的MSS。

12.22　小结

在我们所讨论的实现方案中，一个过程就代表TCP有限状态机中的一个状态。本章讲述了11个状态过程以及这些过程使用的实用过程。每个状态过程都要处理传入的报文段。它必须能正确处理终止请求（如RESET）和特殊的启动请求（如SYN）以及关闭请求（FIN）。

由于TCP必须能够接受失序到达的报文段，这使大多数过程都更加复杂化。例如，TCP可能在所有数据到达之前接收到一个关闭请求（FIN），或有可能在用于完成三次握手操作的报文段全部到达而成功建立连接之前，就会接收到一个携带数据的报文段。实现方案通过在TCB中记录下启动和关闭事件，并且在每个报文段到达时，通过检查这些记录来正确处理失序传送。

深入研究

TCP协议标准［RFC 793］规范了有限状态机并给出了状态变迁的细节。主机需求文档［RFC 1122］讨论了一些变动与说明。

习题

12.1 图11.2的状态图指出，当用户发起一个“关闭”操作时，状态从SYN-RECEIVED直接转换到FIN-WAIT-1。请解释在我们的TCP实现方案中为什么不能做如此转换？

12.2 假设底层网络表现出的性能非常不好，且需要为数据报完全重新排序。有没有可能在一个连接的SYN还未到达之前，该连接就已经有一个FIN到达了？为什么？

12.3 仔细阅读协议标准。如果某个数据还没有被发送出去，却接收到对它的确认，此时TCP应当如何响应？

12.4 如果TCP进入状态FIN-WAIT-2，然而此时远程站点崩溃，请描述将会发生什么现象。提示：此时本地有哪些定时器在运行？请解释应当如何解决此问题。

第13章 TCP：输出处理

13.1 引言

第11章和第12章讨论了如何利用有限状态机控制TCP输入处理。虽然标准规定在输入处理时使用有限状态机的状态变迁，但具体的实现方案要比一张简单的状态图解所隐含的要复杂得多。本章将讨论TCP的输出部分，从中可得知TCP又是如何利用有限状态机来控制输出处理的。另外，本章还将讨论本地机器的应用程序发送信息时的报文段输出，为到达的报文段发送响应时的确认输出，以及因重发定时器超时而触发的重发数据输出。

13.2 TCP输出控制的复杂性

由于TCP输出本身就与TCP的输入及定时事件有着密切的交互影响，而且它们三者总是并发动作的，因此TCP输出较为复杂。例如，当输出进程发送一个报文段的同时，它还必须为重发事件定时。在此之后，如果重发定时器超时，必须由定时进程重发该报文段。同时，应用程序可能会生成新数据，引起TCP发送更多的报文段，或由于对方确认的到达，致使TCP取消预定的重发事件。然而由于底层IP协议可能会丢失、延迟或不按顺序发送数据报，因此所有这些事件可能并不按预计的顺序发生。有可能即使数据的确到达了远程机器，但确认却丢失了。因为远程机器不按顺序接收数据，所以一个ACK也可能确认了多个报文段的接收。更进一步，一台机器可能在接收到所有报文段之前，先接收到了该连接的FIN，因此即使应用程序已关闭了该连接，也可能需要重新传输一些数据。所以对输入和输出的正确响应依赖于先前事件发生的历史背景，而不能孤立地做出简单的规定。

为了有助于控制TCP输入、输出以及定时器进程之间交互作用的复杂性，实现方案采用了一个简单的有限状态机来控制输出操作。与输入时的有限状态机不同，输出状态机不是TCP标准中的一部分。事实上，它只是设计中的一部分，而其他实现方案可能会使用略微不同的策略来控制输出处理。一般说来，因为输入状态机不会区分输出操作，所以任何实现方案都必须有某种处理输出细节的技术。

我们把输出状态机看作是在输入状态机的某个状态中发生的、可定义的、微观的状态变迁。

因此，一旦输入状态机进入ESTABLISHED状态后，输出状态机就在控制传送、重发以及没有任何东西发送的空闲状态之间转换。

13.3 TCP输出的四种状态

从原理上讲，输出状态机比输入状态机简单。输出状态机在概念上只包含四种可能的状态，而且它们之间的相互变迁也相当简单，如图13.1所示。例如，当一个应用程序产生了新数据，需要TCP

来格式化并发送报文段时，它先将数据置入缓冲区中，然后进入 TRANSMIT（发送）输出状态，并向输出进程发出信号，使之能够执行。输出进程调用适当的过程来生成并发送报文段，在此之后输出状态机回到 IDLE（空闲）状态。

状态图解仅提供了一个设计者用来构造软件的模型。就如输入一样，在输出中例外和特殊情况的存在使实现方案变得更复杂，而且也不是一个简单的状态转换图就能够解释所有这些细枝末节的。

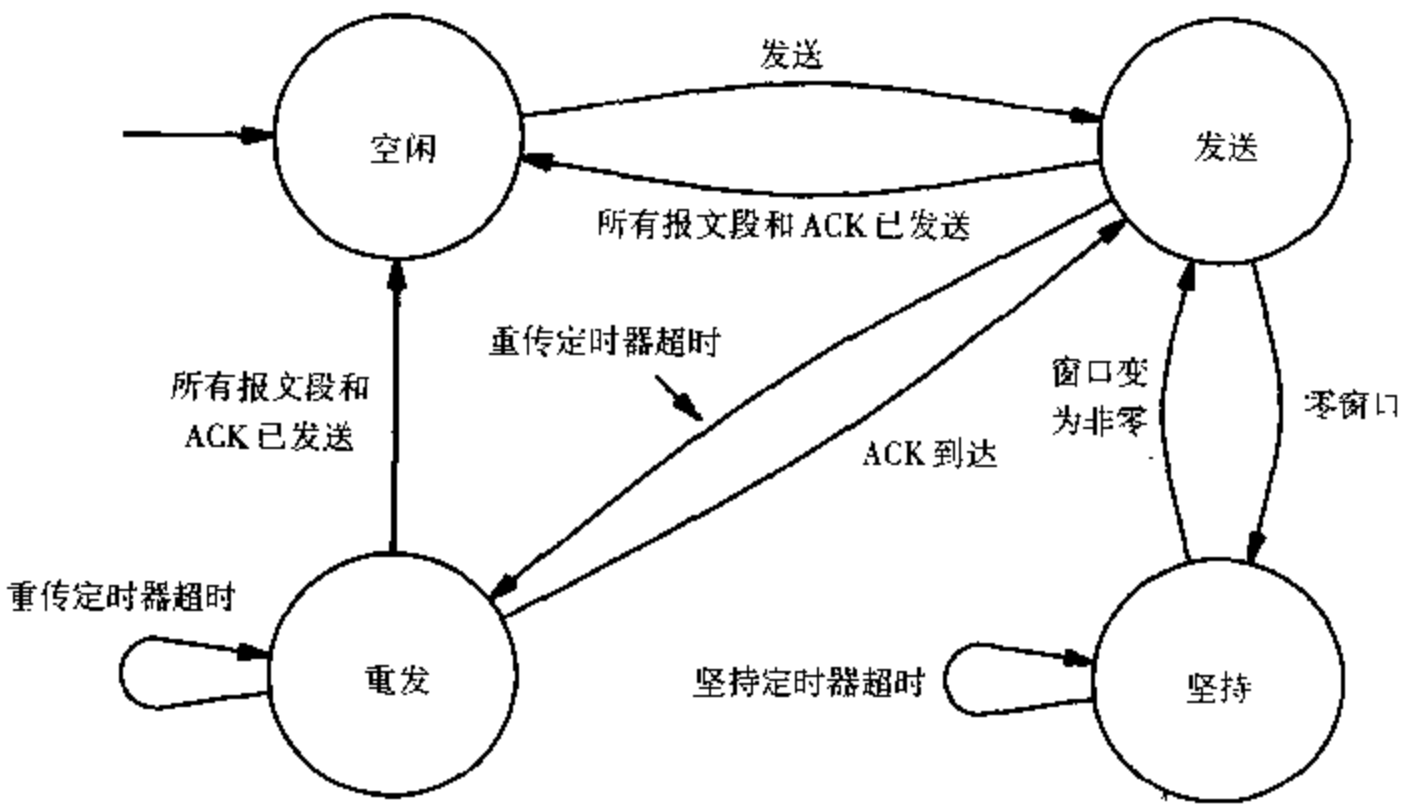


图 13.1　TCP 的四个输出状态之间理论上的状态变迁。与输入时使用的有限状态机不同，输出状态机不是由 TCP 协议标准规定的，它是实现范例定义的

13.4　作为一个进程的 TCP 输出

使用一个独立的 TCP 输出进程有助于区分输入、定时器以及输出进程的执行，并使它们能够并行运行。例如，当输入进程正在为一个传入报文段发送确认响应时，重发定时器可能会超时，并触发重发事件。因为 TCP 数据段可以在携带数据的同时捎带上确认报文，所以这些交互作用可能非常复杂。如果每个需要输出操作的过程都独立地动作，TCP 会生成许多不必要的数据址。为了协调所有的输出操作，范例实现中使用了一个独立的进程来处理输出，并使所有交互作用都是“报文驱动”的。当一个进程需要生成输出数据时，它先将有关的信息保存在 TCB 中，然后向 TCP 输出进程发送一个报文。因此，在生成输出数据的进程之间几乎没有什么交互作用，也几乎不需要互斥操作。

13.5　TCP 输出报文

图 13.2 列出了 TCP 输出进程可接受的报文类型。

编号	报文	含义
1	发送（SEND）	发送数据和/或 ACK
2	坚持（PERSIST）	发送探询以测试接收方的零窗口
3	重发（RETRANSMIT）	重发数据报文段
4	删除（DELETE）	删除已超时的 TCB

图 13.2　TCP 输出进程可接受的四种消息类型

虽然图中的两种报文类型与输出时使用的有限状态机中的两种状态具有相同的名称，但千万不要将它们混淆。一个报文指定一个需要做的动作，而一个状态指出该连接的当前状态。例如，指定了RETRANSMIT（重发）动作的报文可能发生在连接的TRANSMIT（发送）状态中。

13.6 对输出状态和TCB编号的编码

从理论上讲，一旦某个进程向TCP输出进程发送一个报文后，它还必须发送以下两项内容：该报文适用的TCB编号以及一个识别报文类型的值。在例程代码中，操作系统仅提供传递一个整数值的报文传递机制。为了适应报文传递上的限制，例程中的TCP将TCB编号和报文类型值编码成一个可被传递的单一整数。

除输出状态中的符号常量外，文件tcpfsm.h（参见第11章）包含了三个内联（in-line）宏函数的定义，它们用于报文的编码和译码。函数MKEVENT的入口参数是一个TCB编号和报文类型（称为定时器事件）并将它们编码成一个整数，其中低位3位表示事件，高位各位存储TCB索引值。函数TCB的入口参数是一个编码后的整数值，它从中取出TCB索引，而函数EVENT的入口参数也是编码后的整数值，它提取事件类型值。

13.7 TCP输出进程的实现

输出有限状态机的实现方案基本与处理输入时的方式相同。每个状态由一个独立的过程处理，输出进程利用从TCB中得到的当前输出状态来选择适当的过程。程序代码在过程tcpout中。

```
/* tcpout.c - tcpout */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ports.h>

/*------------------------------------------------------------------------
 *  tcpout - handle events affecting TCP output processing
 *------------------------------------------------------------------------
 */
PROCESS tcpout(void)
{
        struct      tcb  *ptcb;
        int         i;

        tcps_oport = pcreate(TCPQLEN);
        signal(Net.sema);           /* synchronize on startup */

        while (TRUE) {
             i = preceive(tcps_oport);
             ptcb = &tcbtab[TCB(i)];
             if (ptcb->tcb_state <= TCPS_CLOSED)
```

```
                continue;        /* a rogue; ignore it                    */
            wait(ptcb->tcb_mutex);
            if (ptcb->tcb_state <= TCPS_CLOSED)
                continue;        /* TCB deallocated                       */
            if (EVENT(i) == DELETE)          /* same for all states      */
                tcbdealloc(ptcb);
            else
                tcposwitch[ptcb->tcb_ostate](TCB(i), EVENT(i));
            if (ptcb->tcb_state != TCPS_FREE)
                signal(ptcb->tcb_mutex);
        }
}
```

tcpout 首先调用 pcreate 来创建一个端口，使报文能够以队列的形式保存。它将端口标识号记录在全局变量 tcp_oport 中，这样使得其他进程知道应该向何处发送报文。然后 tcpout 进入一个无限循环语句，等待到达该端口的下一个报文，取出报文，然后处理。

在从端口接收到一个报文后，tcpout 利用函数 TCB 和 EVENT 译码得出 TCB 索引值与事件类型值。然后它利用 TCB 中的输出状态变量（tcb_ostate）以及 tcposwitch 数组来为当前输出选择一个过程。

tcpout 中有两处做了优化处理。首先，如果 TCB 是关闭的，状态处理过程不起任何作用，因此 tcpout 要明确判断一个 TCB 是否关闭。如果关闭，tcpout 不调用任何状态过程，继续执行。第二，由于任何状态在响应 DELETE 事件时都会释放该 TCB，因此 tcpout 也直接判断是否是 DELETE 事件，如果有 DELETE 报文到达，直接调用 tcbdealloc。

13.8　互斥操作

为了保证 tcpout 能够单独使用 TCB，它必须等待互斥信号量 tcb-mutex。因此，每个状态过程被调用时，都已能够完全单独地访问 TCB。其结果是，各个状态过程不必再等待互斥信号量。总结如下：

> TCP 输出进程在调用一个状态过程之前先获取单独使用 TCB 的权力。因而，该状态过程不必再等待互斥信号量的释放，否则会发生死锁。

13.9　IDLE 状态的实现

过程 tcpidle 实现了 IDLE 状态的处理。每当空闲的连接中有一个事件发生时，输出进程调用 tcpidle。

```
/* tcpidle.c - tcpidle */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```

```
/*------------------------------------------------------------------------
 *  tcpidle - handle events while a connection is idle
 *------------------------------------------------------------------------
 */
int tcpidle(tcbnum, event)
int     tcbnum;
int     event;
{
        if (event == SEND)
                tcpxmit(tcbnum, event);
        return OK;
}
```

前面曾经提到，tcpout会直接判断发生的事件是不是一个DELETE事件。对其他几种事件来说，PERSIST和RETRANSMIT报文不可能在空闲的连接中发生。因此在IDLE状态只有SEND报文才有意义。tcpidle调用tcpxmit来发送数据或确认。

13.10 PERSIST状态的实现

当远程的接收方通告了一个零窗口时，就由PERSIST状态来处理各种事件。为了避免由于更新窗口大小的数据在传输时被丢失，从而导致TCP永远不能发送数据，协议标准要求发送方能够定期向接收方发送探询报文段。接收方将在其确认中返回它的最近窗口大小。

过程tcppersist实现了PERSIST状态。

```
/* tcppersist.c - tcppersist */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcppersist - handle events while the send window is closed
 *------------------------------------------------------------------------
 */
int tcppersist(tcbnum, event)
int     tcbnum, event;
{
        struct     tcb  *ptcb = &tcbtab[tcbnum];

        if (event != PERSIST && event != SEND)
                return OK; /* ignore everything else */
        tcpsend(tcbnum, TSF_REXMT);
        ptcb->tcb_persist = min(ptcb->tcb_persist<<1, TCP_MAXPRS);
        tmset(tcps_oport, TCPQLEN, MKEVENT(PERSIST, tcbnum),
                ptcb->tcb_persist);
```

```
        return OK;
}
```

在 PERSIST 状态中，只允许定期测试窗口大小的事件发生，因此，tcppersist 丢弃其他所有事件。它调用 tcpsend 来发送一个报文段，并调用 tmset 为其后的另一个 PERSIST 事件重新定时。

13.11　TRANSMIT 状态的实现

过程 tcpxmit 处理传输的所有细节。

```
/* tcpxmit.c - tcpxmit */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpxmit - handle TCP output events while we are transmitting
 *------------------------------------------------------------------------
 */
int tcpxmit(int tcbnum, int event)
{
        struct  tcb     *ptcb = &tcbtab[tcbnum];
        void            *tv;
        int             tosend, pending, window;

        if (event == RETRANSMIT) {
                tmclear(tcps_oport, MKEVENT(SEND, tcbnum));
                tcprexmt(tcbnum, event);
                ptcb->tcb_ostate = TCPO_REXMT;
                return OK;
        } /* else SEND */
        tosend = tcphowmuch(ptcb);
        if (tosend == 0) {
                if (ptcb->tcb_flags & TCBF_NEEDOUT)
                        tcpsend(tcbnum, TSF_NEWDATA);   /* just an ACK */
                if (ptcb->tcb_snext == ptcb->tcb_suna)
                        return OK;
                /* still unacked data; restart transmit timer  */
                tv = MKEVENT(RETRANSMIT, tcbnum);
                if (!tmleft(tcps_oport, tv))
                        tmset(tcps_oport, TCPQLEN, tv, ptcb->tcb_rexmt);
                return OK;
        } else if (ptcb->tcb_swindow == 0) {
                ptcb->tcb_ostate = TCPO_PERSIST;
                ptcb->tcb_persist = ptcb->tcb_rexmt;
```

```
        tcpsend(tcbnum, TSF_NEWDATA);
        tmset(tcps_oport, TCPQLEN, MKEVENT(PERSIST,tcbnum),
             ptcb->tcb_persist);
        return OK;
    }   /* else, we have data and window */
    ptcb->tcb_ostate = TCPO_XMIT;
    window = min(ptcb->tcb_swindow, ptcb->tcb_cwnd);
    pending = ptcb->tcb_snext - ptcb->tcb_suna;
    while (tcphowmuch(ptcb) > 0 && pending < window) {
        tcpsend(tcbnum, TSF_NEWDATA);
        pending = ptcb->tcb_snext - ptcb->tcb_suna;
    }
    tv = MKEVENT(RETRANSMIT, tcbnum);
    if (!tmleft(tcps_oport, tv))
            tmset(tcps_oport, TCPQLEN, tv, ptcb->tcb_rexmt);
    return OK;
}
```

如果对tcpxmit的调用是由重发事件引起的，那么连接就进入RETRANSMIT状态，并调用tcprexmit来发送该报文段。

对于正常的传输，tcpxmit检查以下几种可能性。首先它调用tcphowmuch计算将被发送的数据有多少八位组。如果没有需要发送的数据，则tcpxmit查看字段tcb_flags中的TCBF_NEEDOUT位，以得知是否还需要输出操作（如一个确认、推数据或窗口更新数据），如果有正在等待执行的输出操作，则调用tcpsend。

如果输出缓冲区中的数据已准备好并随时可以被传输，则tcpxmit检查发送窗口（tcb_swindow）。如果接收方给出的窗口大小为零，则tcpxmit使连接进入PERSIST状态，它将当前的重发定时器时间（tcb_rexmt）作为坚持时间，并根据它来安排PERSIST事件。

最后，tcpxmit对数据已准备好且接收方公布的窗口大小非零的情况进行处理。它首先进入TRANSMIT状态，并调用tcphowmuch来计算要发送多少数据。然后，它重复不断地调用tcpsend来发送报文段。为了处理一个或多个被发送的报文段丢失的情况，tcpxmit为窗口中的第一个报文段安排一个重发事件。

13.12 RETRANSMIT（重发）状态的实现

因为“重发”状态的实现方案中涉及了对往返时延的估算，以及延迟传送策略（backoff heuristics），所以其程序代码放在第15章中讲解。

13.13 发送一个报文段

当tcpxmit需要发送一个报文段时，它调用tcpsend来完成这项任务。tcpsend分配一个缓冲区，将数据组合成报文段，并将报文段封装在一个IP数据报中发送出去。

```
/* tcpsend.c - tcpsend */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpsend -  compute and send a TCP segment for the given TCB
 *------------------------------------------------------------------------
 */
int tcpsend(tcbnum, rexmt)
int    tcbnum;
Bool   rexmt;
{
       struct  tcb     *ptcb = &tcbtab[tcbnum];
       struct  ep      *pep;
       struct  ip      *pip;
       struct  tcp     *ptcp;
       u_char          *pch;
       unsigned        i, datalen, tocopy, off;
       int             newdata;

       pep = (struct ep *)getbuf(Net.netpool);
       if ((int)pep == SYSERR)
               return SYSERR;
       pep->ep_order = ~0;
       pip = (struct ip *)pep->ep_data;
       pip->ip_src, ptcb->tcb_lip;
       pip->ip_dst, ptcb->tcb_rip;
       datalen = tcpsndlen(ptcb, rexmt, &off);   /* get length & offset*/
       ptcp = (struct tcp *)pip->ip_data;
       ptcp->tcp_sport = ptcb->tcb_lport;
       ptcp->tcp_dport = ptcb->tcb_rport;
       if (!rexmt)
               ptcp->tcp_seq = ptcb->tcb_snext;
       else
               ptcp->tcp_seq = ptcb->tcb_suna;
               ptcp->tcp_seq = ptcb->tcb_suna;
       ptcp->tcp_ack = ptcb->tcb_rnext;

       if ((ptcb->tcb_flags & TCBF_SNDFIN) &&
           SEQCMP(ptcp->tcp_seq+datalen, ptcb->tcb_slast) == 0)
            ptcb->tcb_code |= TCPF_FIN;
       ptcp->tcp_code = ptcb->tcb_code;
       ptcp->tcp_offset - TCPHOFFSET;
       if ((ptcb->tcb_flags & TCBF_FIRSTSEND) == 0)
```

```
        ptcp->tcp_code |= TCPF_ACK;
    if (ptcp->tcp_code & TCPF_SYN)
        tcprmss(ptcb, pip);
    if (datalen > 0)
        ptcp->tcp_code |= TCPF_PSH;
    ptcp->tcp_window = tcprwindow(ptcb);
    if (ptcb->tcb_flags & TCBF_SUPOK) {
            short up = ptcb->tcb_supseq - ptcp->tcp_seq;

            if (up >= 0) {
#ifdef      BSDURG
                    ptcp->tcp_urgptr = up + 1;      /* 1 past end   */
#else /* BSDURG */
                    ptcp->tcp_urgptr = up;
#endif      /* BSDURG */
                    ptcp->tcp_code | = TCPF_URG;
            } else
                    ptcp->tcp_urgptr = 0;
        } else
                    ptcp->tcp_urgptr = 0;
        pch = &pip->ip_data[TCP_HLEN(ptcp)];
        i = (ptcb->tcb_sbstart+off) % ptcb->tcb_sbsize;
        for (tocopy=datalen; tocopy > 0; --tocopy) {
                *pch++ = ptcb->tcb_sndbuf[i];
                if (++i >= ptcb->tcb_sbsize)
                        i = 0;
        }
        ptcb->tcb_flags &= ~TCBF_NEEDOUT;    /* we're doing it    */
        if (rexmt) {
                newdata = ptcb->tcb_suna + datalen - ptcb->tcb_snext;
                if (newdata < 0)
                        newdata = 0;
                TcpRetransSegs++;
        } else {
                newdata = datalen;
                if (ptcb->tcb_code & TCPF_SYN)
                        newdata++; /* SYN is part of the sequence */
                if (ptcb->tcb_code & TCPF_FIN)
                        newdata++; /* FIN is part of the sequence */
        }
        ptcb->tcb_snext += newdata;
        if (newdata >= 0)
                TcpOutSegs++;
        if (ptcb->tcb_state == TCPS_TIMEWAIT)     /* final ACK    */
                tcpwait(ptcb);
```

```
        datalen += TCP_HLEN(ptcp);
        tcph2net(ptcp);
        pep->ep_order &= ~FPO_TCP;
        ptcp->tcp_cksum = 0;
        ptcp->tcp_cksum = tcpcksum(pep, datalen);
        return ipsend(ptcp->tcp_rip, pep, datalen, IPT_TCP, IPP_NORMAL,
                IP_TTL);
}
```

虽然发送一个报文段的原理很简单，但细节问题使具体的程序代码复杂化。从理论上讲，TCP 如图 13.3 所示那样维护管理序列空间。

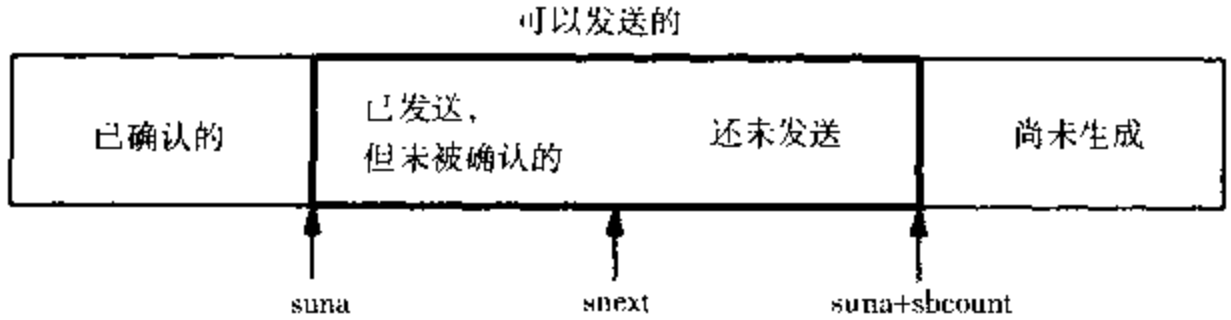


图 13.3　序列空间概念模型，TCB 中含有一个指针字段，指向该序列空间。序号自左至右递增

由于TCP使用了一个环形输出缓冲区来保存数据，所以tcpsend必须在访问数据时，把序列空间中的计算结果转换成对应的缓冲区地址。图 13.4解释了有效数据是如何映射到环形缓冲区中的。

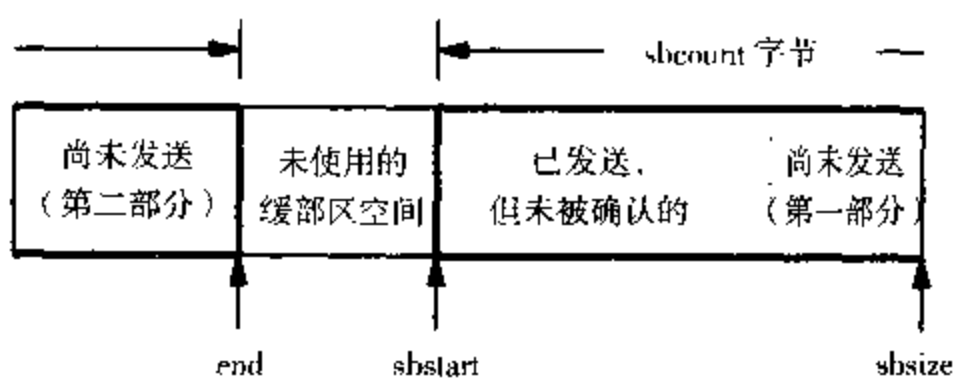


图 13.4　有效数据在环形 TCP 输出缓冲区中首尾衔接地存放。变量 sbcount 给出有效数据字节数，而变量 end 给出数据的最后一个字节位置。end 可通过计算（sbstart + sbcount），再除以 sbsize 取模得到

tcpsend 首先分配用来保存 IP 数据报和完整的 TCP 报文段的缓冲区。它将本地和远程 IP 地址复制到 IP 数据报中，并调用 tcpsndlen 来计算被发送的数据长度，以及第一个八位组的序号。如果该调用是由重发引起的，则 tcpsend 利用第一个未被确认的数据八位组的序号来代替第一个八位组序号。

tcpsend 把希望接收到的下一个传入数据八位组的序号（tcb_rnext）保存在报文段的 acknowledgement 字段中。在填写完首部中其他字段值之后，tcpsend 调用 tcprwindow 来计算窗口大小通告值，并检查紧急数据。最后，它将输出缓冲区中的数据部分复制到报文段中，填写剩余的首部字段，调用 tcpb2net 将这些字段转换为网络字节顺序，调整用 tcpcksum 来计算报文段的校验和，并将最终得到的 IP 数据报传递给 ipsend 发送出去。

13.14 计算TCP数据长度

过程tcpsndlen计算被发送的数据量。

```
/* tcpsndlen.c - tcpsndlen */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpsndlen - compute the packet length and offset in sndbuf
 *------------------------------------------------------------------------
 */
int tcpsndlen(ptcb, rexmt, poff)
struct tcb  *ptcb;
Bool        rexmt;
unsigned    *poff;
{
        struct  uqe     *puqe, *puqe2;
        unsigned        datalen;

        if (rexmt || (ptcb->tcb_code & TCPF_SYN))
                *poff = 0;
        else
                *poff = ptcb->tcb_snext - ptcb->tcb_suna;
        datalen = ptcb->tcb_sbcount - *poff;
        datalen = min(datalen, ptcb->tcb_swindow);
        return min(datalen, ptcb->tcb_smss);
}
```

它对正常情况的处理很简单。如果该长度值是报文段重发时所要求的，或是为数据流中的第一个报文段计算的，则tcpsndlen将偏移量初始值置为0（即，与第一个未确认数据字节之间的偏移量），然后，它将数据长度值设置为输出缓冲区中的八位组数。对于其他一些情况，tcpsndlen计算该数据相对于第一个未发送数据字节的偏移量，并通过该字节的序号与输出缓冲区中最高八位组的序号之间的差来计算将要发送的数据长度。当然，对于非重发数据，tcpsndlen必须遵守接收方的窗口通告值，因此，如果窗口比较小，它必须限制数据长度。

13.15 序号计算

过程tcpxmit调用函数tcphowmuch来决定它是否需要产生一个报文段。tcphowmuch计算了还有多少数据在等待发送。

```
/* tcphowmuch.c.c - tcphowmuch */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcphowmuch.c - compute how much data is available to send
 *------------------------------------------------------------------------
 */
int tcphowmuch(ptcb)
struct tcb  *ptcb;
{
        int  tosend;

        tosend = ptcb->tcb_suna + ptcb->tcb_sbcount - ptcb->tcb_snext;
        if (ptcb->tcb_code & TCPF_SYN)
             ++tosend;
        if (ptcb->tcb_flags & TCBF_SNDFIN)
             ++tosend;
        return tosend;
}
```

tcphowmuch 与 tcpsndlen 之类的过程不同，因为它要处理一个重要的细节：它一方面要计算数据量，另一方面还要计算SYN和FIN码的数量。因此，其他计算数据长度的过程返回一个0时，它可能还会报告一个正值。

13.16　其他 TCP 过程

13.16.1　发送复位

当错误的报文段到达时（例如，当连接并不存在时），输入过程调用 tcpreset 生成并发送一个 RESET 报文段。该过程的入口参数是一个指针，指向引起差错的输入分组。

```
/* tcpreset.c - tcpreset */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpreset - generate a reset in response to a bad packet
 *------------------------------------------------------------------------
 */
```

```
int
tcpreset(struct ep *pepin)
{
        struct ep  *pepout;
        struct ip  *pipin = (struct ip *)pepin->ep_data, *pipout;
        struct tcp *ptcpin = (struct tcp *)pipin->ip_data, *ptcpout;
        int             datalen;

        if (ptcpin->tcp_code & TCPF_RST)
                return OK;          /* no RESETs on RESETs     */
        pepout = (struct ep *)getbuf(Net.netpool);
        if ((int)pepout == SYSERR)
                return SYSERR;
        pipout = (struct ip *)pepout->ep_data;
        pipout->ip_src = pipin->ip_dst;
        pipout->ip_dst = pipin >ip_src;

        ptcpout = (struct tcp *)pipout->ip_data;
        ptcpout->tcp_sport = ptcpin->tcp_dport;
        ptcpout->tcp_dport = ptcpin >tcp_sport;
        if (ptcpin->tcp_code & TCPF_ACK) {
                ptcpout->tcp_seq = ptcpin->tcp_ack;
                ptcpout->tcp_code = TCPF_RST;
        }else {
                ptcpout->tcp_seq = 0;
                ptcpout->tcp_code = TCPF_RST|TCPF_ACK;
                }
                datalen = pipin->ip_len - IP_HLEN(pipin)- TCP_HLEN(ptcpin);
                if (ptcpin->tcp_code & TCPF_SYN)
                        datalen++;
                if (ptcpin->tcp_code & TCPF_FIN)
                        datalen++;
                ptcpout->tcp_ack = ptcpin->tcp_seq + datalen;
                ptcpout->tcp_offset = TCPHOFFSET;
                ptcpout->tcp_window = ptcpout->tcp_urgptr = 0;
                tcph2net(ptcpout);
                ptcpout->tcp_cksum = 0;
                ptcpout->tcp_cksum = tcpcksum(pepout, TCPMHLEN);
                TcpOutSegs++;
                TcpOutRsts++;
                return ipsend(pipin->ips_src, pepout, TCPMHLEN, IPT_TCP,
                        IPP_NORMAL, IP_TTL);
}
```

tcpreset 检测引发问题的报文段中的 TCPF_RST 位，以避免为输入的 RESET 再生成一个 RESET 响应。然后，它继续分配用于保存一个 IP 数据报以及一个 RESET 报文段的缓冲区，并填写其 IP 首部。

在填写 TCP 首部时，tcpreset 查看引发问题的报文段中是否含有一个 ACK。如果有，则 tcpreset 从输入的确认字段中提取序号，作为 RESET 序号，否则序号为 0。

在填写完所有首部字段后，tcpreset 调用 tcph2net 将 TCP 首部中的整数转换为网络字节顺序。然后，它调用 tcpcksum 来计算校验和，并调用 ipsend 来发送最终得到的数据报。

13.16.2　转换成网络字节顺序

过程 tcph2net 将 TCP 首部中的字段转换为网络字节顺序。其程序代码非常简单。

```
/* tcph2net.c - tcph2net */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcph2net -  convert TCP header fields from host to net byte order
 *------------------------------------------------------------------------
 */
struct tcp *tcph2net(ptcp)
struct tcp  *ptcp;
{
        /* NOTE: does not include TCP options */

        ptcp->tcp_sport = hs2net(ptcp->tcp_sport);
        ptcp->tcp_dport = hs2net(ptcp->tcp_dport);
        ptcp->tcp_seq = hl2net(ptcp->tcp_seq);
        ptcp->tcp_ack = hl2net(ptcp->tcp_ack);
        ptcp->tcp_window = hs2net(ptcp->tcp_window);
        ptcp->tcp_urgptr = hs2net(ptcp->tcp_urgptr);
        return ptcp;
}
```

13.16.3　等待输出缓冲空间

产生输出的应用程序有可能在某个 TCB 相关联的缓冲区中没有足够空间的情况下，不得不被阻塞。它们调用过程 tcpgetspace 来分配空间。

```
/* tcpgetspace.c - tcpgetspace */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>
```

```
/*------------------------------------------------------------------------
 *  tcpgetspace  -  wait for space in the send buffer
 *     N.B. - returns with tcb_mutex HELD
 *------------------------------------------------------------------------
 */
int tcpgetspace(ptcb, len)
struct tcb  *ptcb;
unsigned    len;
{
        if (len > ptcb->tcb_sbsize)
                return TCPE_TOOBIG;  /* we'll never have this much   */
        while (1) {
                wait(ptcb->tcb_ssema);
                wait(ptcb->tcb_mutex);
                if (ptcb->tcb_state == TCPS_FREE)
                        return SYSERR;              /* gone         */
                if (ptcb->tcb_error) {
                        tcpwakeup(WRITERS, ptcb); /* propagate it */
                        signal(ptcb->tcb_mutex);
                        return ptcb->tcb_error;
                }
                if (len <= ptcb->tcb_sbsize - ptcb->tcb_sbcount)
                        return len;
                signal(ptcb->tcb_mutex);
        }
}
```

如果某个应用程序所需的空间比整个缓冲区的容量还大，那么tcpgetspace返回一个差错代码。否则，它释放互斥信号量，并再次等待“发送”信号量。tcpgetspace查看字段tcb_error，以确定是否已有差错发生(如，一个RESET引起了TCP终止该连接)。如果有差错，则tcpgetspace调用tcpwakeup来唤醒其他正在等待写操作的进程，释放互斥信号量，并向调用者返回差错代码。

如果没有发生差错，则tcpgetspace通过从缓冲区尺寸中减去已被使用的字节数来计算可用空间。如果可用空间足够大，能满足需要，则tcpgetspace返回调用者。否则它释放互斥信号量，并再次等待“发送”信号量。注意当tcpgetspace找到足够的空间并返回调用者时，它依旧持有该互斥信号量。因此其他过程都不能从缓冲区中取走空间，除非该调用者取走了它所需的空间并释放该信号量。

13.16.4 唤醒等待TCB的进程

当应用程序正在等待一个TCP连接上的数据传输时会被阻塞。如果有异常情况发生，导致了TCP清除该连接，那么它必须在释放该TCB之前唤醒所有正在等待中的进程。每个进程都会重新执行，通常这些进程是在“读”或“写”过程中，它们将找出在TCB中记录下的差错情况，并向其调用者报告。过程tcpwakeup唤醒正在等待的进程。它有两个入口参数：第一个指出应当被唤醒的进程的类型(或者是“读进程”(READERS)，或者是“写进程”(WRITERS))，而第二个参数给出一

个指针，指向适当的TCB。

```
/* tcpwakeup.c - tcpwakeup */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpwakeup -  wake up processes sleeping for TCP, if necessary
 *     NB: Called with tcb_mutex HELD
 *------------------------------------------------------------------------
 */
int tcpwakeup(type, ptcb)
int         type;
struct tcb  *ptcb;
{

        int        freelen;
        STATWORD   ps;
        disable(ps);
        if (type & READERS) {
             if (((ptcb->tcb_flags & TCBF_RDONE) ||
                 ptcb->tcb_rbcount > 0 ||
                 (ptcb->tcb_flag &TCBF_PUPOK)) &&
                 scount(ptcb->tcb_rsema) <= 0)
                  signal(ptcb->tcb_rsema);
        }
        if (type & WRITERS) {
             freelen = ptcb->tcb_sbsize - ptcb->tcb_sbcount;
             if (((ptcb->tcb_flags & TCBF_SDONE) || freelen > 0) &&
                 scount(ptcb->tcb_ssema) <= 0)
                  signal(ptcb->tcb_ssema);
             /* special for abort */
             if (ptcb->tcb_error && ptcb->tcb_ocsem > 0)
                  signal(ptcb->tcb_ocsem);
        }
        restore(ps);
        return OK;
}
```

有两个信号量用于控制读和写操作。当所有数据都已被进程读完，且没有正在等待读的进程时，“读”信号量计数值为0。因此任何试图进行“读”操作的新进程都会被阻塞。tcpwakeup查看输入缓冲区，看是否有数据可读，如果有，就释放“读”信号量。如果多个进程被阻塞，只有其中的一个可以继续执行。如果没有被阻塞的进程，它调用signal，将信号量计数值加1，这样下一个发出“读”操作的进程就能够继续执行。因此“唤醒”（wakeup）这个名称有些名不符实，因为此时

调用它并不能唤醒任何进程。

tcpwakeup查看该TCB，以决定允许继续执行的是读进程还是写进程。如果对方已经发送了所有数据，tcpwakeup将会看到TCBF_RDONE比特的值为1。它还要检查接收缓冲区的字节计数器，以判断数据是否已到达，并检查TCBF_RUPOK比特看其中是否有紧急数据。在这些情况下，tcpwakeup都要查看当前的信号量，以判断信号量是否允许其他进程访问，如果不能，则调用signal。

tcpwakeup还参与差错传播工作。如果一个差错发生时，进程仍因等待信号量而被阻塞，它调用signal以允许第一个进程继续执行。当该进程发现差错代码后，再次释放信号量，以允许下一个进程继续读入。进程一个个地执行，然后释放信号量以允许下一个进程继续执行，直到所有正在等待的进程都恢复执行并发现了差错代码为止。

唤醒正在等待写的进程时，tcpwakeup的行为是类似的。它释放“写”信号量，允许第一个进程继续执行，该进程在执行时唤醒下一个进程，以此类推。tcpwakeup还要检查几种特殊情况：连接被中止，以及有一个打开或关闭的操作正在等待执行。在这些情况下，tcpwakeup释放“打开/关闭”信号量。对打开或关闭操作的调用，将返回一个差错代码并删除该TCB。

13.16.5 选择初始序号

为了使TCP在报文段被丢失、复制或不按顺序地发送的情况下能正常工作，程序代码必须在应用程序试图创建一个新连接时，为其选择一个惟一的初始序号。过程tcpiss利用当前时钟的时间值生成初始序号。

```
/* tcpiss.c - tcpiss */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#define     TCPINCR         904

/*------------------------------------------------------------------------
 *  tcpiss -  set the ISS for a new connection
 *------------------------------------------------------------------------
 */
int tcpiss(void)
{
        static     int   seq = 0;
        extern     long clktime;          /* the system ticker */

        if (seq == 0)
             seq = clktime;
        seq += TCPINCR;
        return seq;
}
```

tcpiss管理一个静态变量，并利用时钟一次性初始化该变量。在初始化之后，tcpiss为每个新连

接递增起始序号，每次增加一个不太大的数值（904）。

13.17　小结

TCP的输出处理利用了一个非常简单的有限状态机，通常我们认为它是输入有限状态机的一个状态中的可控微状态变迁。输出状态机有四个状态，分别对应于一个空闲的连接、一个正在传输数据的连接、一个数据正在等待发送但接收窗口已关闭的连接以及一个正在重发数据的连接。

为了有助于区分输入、输出和定时器函数之间的交互作用，范例实现中为每个动作独立使用一个进程。所有正常的TCP输出动作都在输出进程中发生，输出进程在任何时候都只执行一个操作。因此不存在控制传输、重发及确认这些操作之间并行交互的问题。其他进程需要输出时利用报文传递通知输出进程，输出进程使用一个独立的报文队列，队列中的每个报文包含了一个TCB编号以及对该TCB的请求。这些请求可分为数据或确认的传输、重发、删除TCB或探询窗口是否仍然关闭。

在范例实现方案中为输出有限状态机的每种状态使用一个独立的过程。同时我们也看到虽然设计思想很简单，但细枝末节、例外和特殊情况的存在使得程序代码复杂化。

深入研究

Postel［RFC 793］规范了TCP协议，而［RFC 1122］中包含了更进一步的说明。

习题

13.1 如果在一个TCP连接已经打开后，路由发生了变化，则TCP校验和会发生什么变化？请解释。提示：考虑tcpsend使用的伪首部及IP地址。

13.2 假设一个低级别的进程正在等待一个TCP连接上的写操作，一个中级别的进程正在执行，而一个高级别的进程又需要创建新连接。那么tcpwakeup过程和TCB释放策略如何才能做到允许中级别进程防止高级别进程创建新连接？请解释原因。

13.3 请教David Stevens（dls@sequent.com）为什么在范例代码中，为起始序号每次增加904这个数字？

13.4 阅读标准规范，找出TCP在探询窗口是否关闭时应保持多长时间。而我们的范例实现中又保持了多久？

第 14 章　定时器管理

14.1　引言

实时时延处理是 TCP 的一个基本组成部分。除了在处理重发事件时显然要用到一些定时器之外，TCP 在连接关闭后，还要设置两个 MSL 的时延定时器；在接收到一个窗口大小为零的通告值后，要为探询事件设置定时器；并且在一些实现方案中，还要为确认到达的时延设置定时器。

本章所关心的是处理实时时延的软件实现方案。它体现了只用一个数据结构是如何有效地保存多种不同的时延请求的，并且只用一个定时器进程又是如何管理所有这些TCP时延事件请求的。在前几章中已经提到输入和输出软件是如何调用定时器例程来为时延事件定时，以及输出进程在超时的事件发生时是如何管理这些事件的。在今后的章节中，我们将继续讨论TCP的平均往返时延的估算，以及如何利用平均往返时延的估值来计算重发时延，从中可完全了解有关定时器的管理。

14.2　定时事件的通用数据结构

要有效地管理定时事件，其关键在于一个被称为delta表的数据结构。delta表中的每个表项都对应于一个被安排在将来发生的事件。因为每个定时事件可能发生在不同的时间，所以delta表中的每项都有一个字段，该字段给出事件将要发生的时间。为了提高 delta 表的更新效率，在表中的事件按照它们将要发生的时间顺序保存，并且被保存的是相对时间，而非绝对时间。例如，图 14.1 所示的 delta 表中包含了四个表项，它们分别被安排在将来的第 16、20、21 和 30 个单位时间时发生，但在表中，这些表项的时间值分别是16、4、1和9，因为第一个表项将在从现在算起的第 16 个单位时间发生，而第二个事件将在第一个事件发生之后的第 4 个单位时间发生，而第三个事件将在第二个事件发生之后，再过1个单位时间时发生，至于第四个事件将在第三个事件之后的第 9 个单位时间发生。

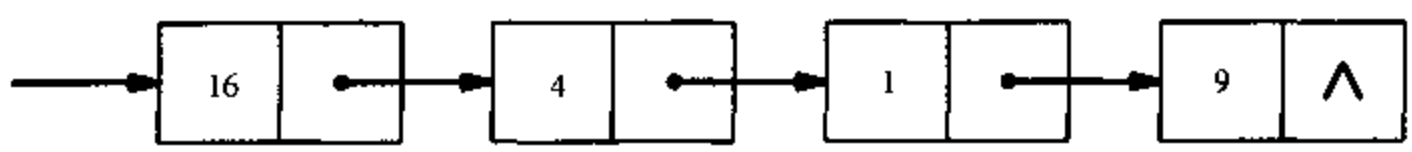


图 14.1　delta 表中的事件将分别在第 16、20、21 和 30 个单位时间发生，而被保存的时间值却给出了相对于前一事件发生的时间

使用delta表的主要优点在于它的高效率。因为所有的时间都是相对的，所以定时器进程只需要定期地递减表中第一项的时间值。当第一项的时间值为 0 时，事件发生，定时进程将该表项从 delta 表中删除。然后定时进程又定期递减表中下一个表项的时间值。当我们考虑 delta 表的管理软件时会更清楚地看到这种方法的简洁性和高效性。

14.3　TCP 事件使用的数据结构

在TCP的delta表中，每个表项的内容除了包含一个时间字段和一个用以指向下一个表项的指针之外，还必须说明一旦定时器超时，系统将要采取何种行动。结构tqent给出了TCP delta表中每个表项的精确结构。这些定义在文件tcptimer.h中给出。

```
/* tcptimer.h */

#ifndef_TCPTIMER_H_
#define_TCPTIMER_H_

/* A timer delta list entry */

struct tqent {
        int     tq_timeleft;        /* time to expire (1/100 secs)      */
        long    tq_time;            /* time this entry was queued       */
        int     tq_port;            /* port to send the event           */
        int     tq_portlen;         /* length of "tq_port"              */
        void    tq_msg;             /* data to send when expired        */
        struct tqent *tq_next;      /* next in the list                 */
};
/* timer process declarations and definitions */

extern  PROCESS  tcptimer();
#define TMSTK    512                /* stack size for fast timer        */
#define TMPRI    100                /* timer process priority           */
#define TMNAM    "tcptimer"         /* name of fast timer process       */
#define TMARGC   0                  /* count of args to TCP timer       */

extern long   ctr100;               /* 1/100th of a second clock        */

extern int    tqmutex;
extern int    tqpid;
extern struct tqent    *tqhead;

int tmclear(int, void *), tmleft(int, void *)
int tmset(int, int, void *, int);
int tcpkilltimers(struct tcb *);

#endif /* _TCPTIMER_H_ */
```

字段tq_next中包含的是指向表中下一个表项的指针。字段tq_timeleft指出何时该表项中的事件应当发生。在tq_timeleft中的值是相对的，以0.01秒为单位，并遵从以下delta表的原则：

> delta表中第一个表项的时间是相对于当前时间计时的，而其他表项中的时间值是相对于该表中前一个表项的时间计时的。

14.4 定时器、事件和报文

TCP定时器管理软件可根据以下两种基本设计原则之一进行设计：在delta表中的表项可用来保存命令，这些命令在事件发生时，由定时器进程做出解释。或者，表中的项目可用来保存一些报文，这些报文在事件发生时由定时器进程传送。前一种设计思想的主要优点在于，它允许每个定时器事件触发一个任意复杂的操作。而后一种设计思想的主要优点在于它的简洁性。定时器进程不需要知道每个报文的含义。当一个事件发生时，它可以采取同样的动作。

为了保持定时进程的简单与高效，我们选取了后一种设计方案。在delta表中的每个事件都有一个报文字段（字段tq_msg）以及一个端口标识符（字段tq_port），当事件发生后，事件中的报文应当被发送到该端口中。定时器机制将tq_msg中的报文发送到由tq_port指定的端口中。总结如下：

> TCP定时器机制不需要了解或翻译保存在事件中的报文的含义。当事件发生后，定时器仅将给出的报文送往指定的端口。

14.5 TCP定时器进程

当系统启动完成后，协议初始化软件创建一个TCP定时器进程，该进程执行过程tcptimer。

```
/* tcptimer.c - tcptimer */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <tcptimer.h>
#include <sleep.h>
#include <ports.h>
#include <men.h>
#include <proc.h>

int     tqmutex;
int     tqpid;
struct tqent *tqhead;

/*------------------------------------------------------------------------
 *  tcptimer -  TCP timer process
 *------------------------------------------------------------------------
 */
PROCESS tcptimer(void)
{
        long        now, lastrun;   /* times from system clock*/
        int         delta;          /* time since last iteration    */
        struct      tqent *tq;      /* temporary delta list ptr     */

        lastrun = ctr100;           /* initialize to "now"          */
        tqmutex = screate(1);       /* mutual exclusion semaphore   */
```

```
        tqpid = getpid();                   /* record timer process id      */
        signal(Net.sema);                   /* start other network processes*/

        while (TRUE) {
             sleep10(TIMERGRAN);            /* real-time delay              */
             if (tqhead == 0)               /* block timer process if delta */
                  suspend(tqpid);           /*   list is empty              */

             wait(tqmutex);
             now = ctr100;
             delta = now - lastrun;         /* compute elapsed time         */

             /* Note: check for possible clock reset (time moved          */
             /* backward or delay was over an order of magnitude too      */
             /* long)                                                     */

             if (delta < 0 || delta > TIMERGRAN*100)
                  delta = TIMERGRAN*10;     /* estimate the delay           */
             lastrun = now;
             while (tqhead != 0  &&  tqhead->tq_timeleft <= delta) {
                  delta -= tqhead->tq_timeleft;
                  if (pcount(tqhead->tq_port) <= tqhead->tq_portlen)
                       psend(tqhead->tq_port);
                                (int)tqhead->tq_msg);
                  tq = tqhead;
                  tqhead = tqhead->tq_next;
                  freemem(tq, sizeof(struct tqent));
             }
             if (tqhead)
                  tqhead->tq_timeleft -=delta;
             signal(tqmutex);
        }
}
```

定时器进程首先创建一个互斥信号量，并将其标识符保存在变量tqmutex中。tcptimer还要在变量tqpid中保存自己的进程标识号，并且释放网络信号量，以允许分组的传输。最后，tcptimer进入一个无限循环中。

在每次遍历主循环时，tcptimer调用sleep10来延迟TIMERGRAN0.1秒。然后，它检查delta表的第一个表项。如果表中已没有表项，定时器进程调用suspend使自己被阻塞，直至另外一些进程向表中放置了表项。虽然调用suspend不是必须的，但这样做的好处是当定时器进程无事可做时，就不必继续周期地执行了。

只要delta表不为空，tcptimer就继续循环执行。在每次循环时，进程等待互斥信号量（tqmutex），以获取单独使用delta表的权力。接着它处理表中的项目，然后释放互斥信号量，以允许其他进程能够再次访问该表。应注意：当tcptimer在调用sleep10而被阻塞时，该delta表一直是可用的。

在Xinu操作系统中，全局变量ctr100中包含的值是以0.01秒计数的、距离新纪元日期（1900

年1月1日）的值。tcptimer引用变量ctr100来获得当前的实际时间值，并利用变量lastrun记录上一次循环的时间。这样，tcptimer就能通过从当前时间值中减去lastrun中的值来计算出相邻两次循环之间的间隔时间。程序代码查看系统时钟是否已被重新设置（例如，时间值回退，或超出正常时延的10倍）。如果是，则tcptimer用一个合理的估值替换时延，并继续处理。

为了处理delta表中的项目，tcptimer将表项的剩余时间与两次循环之间的间隔时间相比较。如果该事件应当在上次循环和此次循环的间隔中发生，则tcptimer将事件中包含的报文（tq_msg）发送到该报文指定的端口（tq_port）。然后，它从delta表中删除该事件。

在删除一个事件时，tcptimer通过递减事件的时间值来更新变量delta的值。因此与表中所有项目一样，delta的值也总是一个相对时间值，这样就有可能直接将其与保存在各个表项中的时间值相比较。

当tcptimer删除了已发生事件的表项之后，存在两种可能性：delta表可能是空的，也可能是非空的。如果delta表为空，则不需要再继续执行下去。然而如果该表非空，那么在下一个表项中的事件发生之前，还剩下的时间必定大于变量delta的值。在这种情况下，tcptimer在下一次时延循环开始之前，要从表项的剩余时间中减去变量delta的值。

14.6 删除TCP定时器事件

TCP软件有可能在事件超时之前，就需要取消该事件。例如，当它接收到了一个确认时，TCP有可能取消一个对应的重传事件。为了取消一个事件，TCP必须从定时器delta表中删除相应的表项。过程tmclear提供必要的功能。

```
/* tmclear.c - tmclear */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <tcptimer.h>
#include <mem.h>

/*------------------------------------------------------------------------
 *  tmclear -  clear the indicated timer
 *------------------------------------------------------------------------
 */
int tmclear(port, msg)
int    port;
void *msg;
{
        struct  tqent   *prev, *ptq;
        int             timespent;

        wait(tqmutex);
        prev = 0;
        for (ptq = tqhead; ptq != NULL; ptq = ptq->tq_next) {
                if (ptq->tq_port == port && ptq->tq_msg == msg) {
```

```
                timespent = ctr100 - ptq->tq_time;
                if (prev)
                        prev->tq_next = ptq->tq_next;
                else
                        tqhead = ptq->tq_next;
                if (ptq->tq_next)
                        ptq->tq_next->tq_timeleft +=
                                        ptq->tq_timeleft;
                signal(tqmutex);
                freemem(ptq, sizeof(struct tqent));
                return timespent;
        }
        prev = ptq;
    }
    signal(tqmutex);
    return SYSERR;
}
```

tmclear 的两个入口参数分别是一个报文（msg）和一个端口标识符（port），它删除与这一对报文和端口号所对应的定时器事件。程序代码很简单。tmclear 搜索 delta 表，直至找到与两个入口参数相匹配的表项。在执行中，它总是保存着一个指向表中某表项的指针（ptq）和一个指向其前一个表项的指针（prev）。当 tmclear 找到匹配表项后，就将该表项从链表中拆除，以删除该表项，并调用 freemem 将存储区释放回系统的空闲存储区池。

曾经说过 delta 表的表项中保存的时间都是相对时间，因此，一旦 tmclear 删除了一个事件后，它还必须仔细地调整其后紧跟的事件的时延。为了完成这项任务，tmclear 检查字段 tq_next 以判断被删除的表项后面还有没有其他表项。如果有，tmclear 在下一个紧跟表项的时延上增加一个被删除表项的时延。

14.7　删除一个 TCB 的所有事件

在第 11 章和第 12 章中曾经提到，TCP 在删除一个 TCB 之前必须删除所有与该 TCB 相关的事件。过程 tcpkilltimers 完成这项任务。由于我们的 TCP 软件仅允许存在三种类型的报文，并且只需要安排将报文发送到 TCP 输出端口，因此 tcpkilltimers 可通过三次调用 tmclear 来删除某个 TCB 的所有 TCP 事件。

```
/* tcpkilltimers.c - tcpkilltimers */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpkilltimers    kill all outstanding timers for a TCB
 *------------------------------------------------------------------------
```

```
 */
int tcpkilltimers(ptcb)
struct tcb  *ptcb;
{
        int  tcbnum = ptcb - &tcbtab[0];

        /* clear all possible pending timers */

        tmclear(tcps_oport, MKEVENT(SEND, tcbnum));
        tmclear(tcps_oport, MKEVENT(RETRANSMIT, tcbnum));
        tmclear(tcps_oport, MKEVENT(PERSIST, tcbnum));
        return OK;
}
```

大多数软件在调用tcpkilltimers时都利用一个指针指向某个TCB，而不是利用一个数组索引。为了与此类软件兼容，将参数ptcb声明为一个指向该TCB的指针。然而tcpkilltimers在调用MKEVENT时需要用到的是索引值。为了计算该索引值，tcpkilltimers应用了指针算法，从指向给出的TCB的指针值中减去TCB数组起始地址值（tcbtab）。这样，变量tcbnum中的内容就是该TCB在数组中的索引值。

14.8 确定出现一个事件的尚需时间

过程tmleft确定等待一个事件的发生尚需多长的时间。如果该事件不存在，则返回“0”。

```
/* tmleft.c - tmleft */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <tcptimer.h>

/*------------------------------------------------------------------------
 *  tmleft -  how much time left for this timer?
 *------------------------------------------------------------------------
 */
int
tmleft (int port, void *msg)
{
        struct  tqent   *tq;
        int             timeleft = 0;

        if (tqhead == NULL)
                return 0;
        wait(tqmutex);
        for (tq = tqhead; tq != NULL; tq = tq->tq_next) {
                timeleft += tq->tq_timeleft;
```

```
            if (tq->tq_port == port && tq->tq_msg == msg) {
                signal(tqmutex);
                return timeleft;
            }
        }
        signal(tqmutex);
        return 0;
}
```

为了判断事件的尚需时间，tmleft必须累加从第一个事件一直到指定事件（包括该指定的事件）的所有事件的相对尚需时间。它利用指针tq遍历该链表，从tqhead开始，沿着指针tq_next经过每一个表项。在遍历链表的同时，tmleft在本地变量timeleft中累加这些表项的相对时间值。当它到达了指定的事件后，释放互斥信号量，并向调用者返回总计时间值。如果指定的事件不存在，tmleft返回“0”。

14.9 插入TCP定时器事件

TCP软件调用过程tmset来创建一个事件，并将其插入TCP的delta表中。tmset的入口参数给出了一个要求的时延（time）、一个在事件发生时被发送的报文（msg）、该报文被送往的端口（port）以及端口长度（portlen）。

```
/* tmset.c - tmset */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <tcptimer.h>
#include <mem.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  tmset -  set a fast timer
 *------------------------------------------------------------------------
 */
int tmset(port, portlen, msg, time)
int    port, portlen;
void   *msg;
int    time;
{
       struct     tqent *ptq, *newtq, *tq;

       newtq = (struct tqent *)getmem(sizeof(struct tqent));
       newtq->tq_timeleft = time;
       newtq->tq_time = ctr100;
       newtq->tq_port = port;
```

```
        newtq->tq_portlen = portlen;
        newtq->tq_msg = msg;
        newtq->tq_next = NULL;

        /* clear duplicates */
        (void) tmclear(port, msg);

        wait(tqmutex);
        if (tqhead == NULL) {
                tqhead = newtq;
                resume(tqpid);
                signal(tqmutex);
                return OK;
        }
        /* search the list for our spot */

        for (ptq=0, tq=tqhead; tq; tq=tq->tq_next) {
                if (newtq->tq_timeleft < tq->tq_timeleft)
                        break;
                newtq->tq_timeleft -= tq->tq_timeleft;
                ptq = tq;
        }
        newtq->tq_next = tq;
        if (ptq)
                ptq->tq_next = newtq;
        else
                tqhead = newtq;
        if (tq)
                tq->tq_timeleft -= newtq->tq_timeleft;
        signal(tqmutex);
        return OK;
}
```

tmset调用getmem为一个事件的表项分配空闲存储区，然后根据入口参数填写该表项中的字段。如果该表项已经在表中存在，它调用tmclear从表中删除该报文。最后，tmset等待互斥信号量，将新表项插入链表中，并在返回前释放互斥信号量。

虽然tmset中的插入程序代码段很简单，但还是有几个细节问题使它看起来有些复杂：只要没有正在等待的事件，定时进程就保持阻塞状态；当tmset向一个空表中插入表项后，它要调用resume重新启动定时进程；当它向一个非空链表中插入表项时，tmset必须搜索该链表，以寻找正确的插入位置。

在搜索过程中，tmset利用两个变量，其中一个指向表中的一个结点（tq），另一个指向该结点的前一个结点（ptq）。当tmset经过表中各项时，它从新表项的时延中不断减去这些表项的时延，使它的时延保持相对于表中当前的位置上，即新表项的相对时延。当while循环结束时，新表项应当位于ptq和tq指向的两个表项之间。tmset将新表项链接入表中，并从下一个表项中的时延减去新表项中的时延。

14.10　启动无时延的 TCP 输出

即使调用者指定的时延为 0，范例定时器软件也能正确地工作。tmset 会正确地向 delta 表的开头加入新请求。当 tcptimer 唤醒后，它将从 delta 表中删除该表项，并把其中的报文放置在 TCP 输出端口上。当 TCP 输出进程接收到该报文后，就会继续处理它。

虽然这种机制能正确地工作，但为一个时延为 0 的事件定时是浪费时间，因为它迫使操作系统在主调进程、TCP 定时器进程以及 TCP 输出进程之间不断地快速切换。然而，经常需要为一个时延为 0 的 SEND 事件定时（只要输入进程需要发送一个 ACK，或一个应用程序生成输出数据时都需要）。为了消除这种不必要的内部切换，范例代码提供了过程 tcpkick，它能够为一个无时延的 SEND 事件定时。

```
/* tcpkick.c - tcpkick */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ports.h>

/*------------------------------------------------------------------------
 *  tcpkick -  make sure we send a packet soon
 *------------------------------------------------------------------------
 */
int tcpkick(ptcb)
struct tcb  *ptcb;
{
        int  tcbnum = ptcb - &tcbtab[0];     /* for MKEVENT() */
        void tv;

        tv = MKEVENT(SEND, tcbnum);
        if (ptcb->tcb_flags & TCBF_DELACK && !tmleft(tcps_oport, tv))
                tmset(tcps_oport, TCPQLEN, tv, TCP_ACKDELAY);
        else if (pcount(tcps_oport) < TCPQLEN)
                psend(tcps_oport, (int)tv);     /* send now */
        return OK;
}
```

在生成一个必要的事件之后，tcpkick 检查 TCP，看它是否使用了延迟的 ACK（TCBF_DELACK 位）。只要 ACK 没有延迟，tcpkick 就直接向 TCP 输出进程发送该报文。如果它发现 ACK 应当被延迟，tcpkick 调用 tmset 为该事件安排一小段时延。

14.11　小结

TCP 要求以实时进程处理诸如重发之类的肯定在将来某个时间发生的事件。范例实现方案把这些延迟事件保存在一张 delta 表中，这样做使定期的数据更新工作效率更高。在 delta 表中的每个表

项对应一个事件。delta表根据事件发生的时间顺序组织这些表项，并且保存的时间是相对于表中前一个事件发生的时间。

一个独立的TCP进程管理着这张delta表。它定期递减表中第一个项目的时延，并在该时间值变为0时，触发该事件。当事件发生时，TCP定时器进程从事件中提取一个整型的报文和一个端口标识符，并将该报文发送到这个端口中。因此，定时器进程并不了解也不过问保存在事件中的报文的含义。

深入研究

Comer[1987]更详细地描述了delta表的处理过程，并给出在搜索时管理相对时间的多种方案。

习题

14.1 设计一个略有不同的数据结构，以消除tmset中某些或所有的特殊情况。

14.2 逐行阅读将一个新表项插入delta表中的程序代码，看看你是否能完全理解在搜索过程中如何维护管理相对时间。

14.3 重写tcpkilltimers，使之搜索并删除给定TCB对应的所有表项。与本书中的实现方案相比，它的效率又提高了多少？

14.4 上一题中建议的修改方案如何降低定时机制的通用性？

14.5 如果修改tcpkick使之能够处理SEND以外的报文，有没有必要？为什么？

第15章　流量控制和自适应重发

15.1　引言

TCP能兼容极其多样化的底层物理网络，它可容忍各种时延、吞吐量和分组丢失率。TCP独立地处理各条连接，允许一台计算机有多条连接，且每条连接的各个路径可以具有不同的底层特性。更重要的是，TCP自适应地修改特定连接的往返时延，即使底层分组交换系统发生拥塞或暂时性瘫痪，TCP传输依然是可靠的。

自适应重发机制是TCP的一个核心组成部分，并决定了TCP是否成功。从本质上讲，自适应重发是利用了刚刚发生的动作来预测机器将来的行为。它要求TCP为每次重发计算往返时延，并利用统计学原理，将一次次单独的样本值组合成合理的往返时延估值。更进一步讲，TCP不断地获取新的往返时延，同时也不断地更新往返时延估值。

本章所关心的是提供自适应重发的软件的实现方案。在本章中，将讨论往返时延的测量值、统计平滑处理过程、重发定时、生成和处理确认，以及基于窗口的流量控制。其中包括了拥塞控制、慢启动技巧以及定时器退避和其他优化方案。

虽然本章中所讨论的技巧在实现时仅需要不多的几行程序代码，但它们会对TCP性能产生极其巨大的影响。更重要的，它们是经过大量实验和精细分析得到的结果，并非普通的程序员可独立创作出来的。最后，值得注意的是，这些技巧的大部分现已成为TCP标准的一部分，因而不能视之为可有可无的。

15.2　自适应重发中的难题

从理论上讲，估计一个往返时延并不复杂，但实际互联网中的许多问题为其带来了一些困难。报文段或确认可能会丢失或延迟，这就使每次往返时延的测量值不准确。来自多个源站的拥挤的通信量会导致上下波动幅度很大的时延。再者，可能只要再增加一个连接，其负载就会导致网络或网关的拥塞。最后，报文段丢失后的重发可能会导致拥塞，或者使拥塞状况恶化。

虽然最初的TCP规范中含有许多隐含的缺陷及疏漏，但通过对统计平滑处理方法的改善，以及实践探试法的使用，大多数自适应重发问题已经得以解决。

15.3　自适应重发的调整

要使自适应重发具有高效性及健壮性，它必须在以下五个理论方面仔细研究：

- 重发定时器及退避
- 基于窗口的流量控制
- 最大报文段长度计算

- 避免拥塞及拥塞控制
- 往返时延的估计

在以后的章节中将仔细研究其中的每个方面，并给出用于解决这些微妙问题的技巧的实现方案。

15.4 重发定时器和退避

TCP应用了累计确认（cumulative acknowledgement）策略，也就是每个确认中都携带了一个序号。该序号指出远程站目前已接到数据流中多少个连续八位组。由于确认并不针对单个报文段，并且确认也可能丢失，所以发送方无法区分某个确认是由报文段的第一次传送引起的，还是由于重发而产生的。因此，发送方不能为重发报文段准确地度量往返时延。

15.4.1 Karn算法

协议标准规定TCP应当采用称作“Karn算法”（Karn's algorithm）的技术来控制重发定时器的值。正常的数据传输阶段中，在重发定时器超时之前，每个报文段都会有一个确认到达。这种情况下，Karn的算法不干扰正常的测量往返时延的过程，并为下一个将要发送的报文段计算重发超时时间。但是由于TCP无法准确地将发出去的报文段与接收到的确认一一对应，因此Karn算法规定TCP应当忽略所有重发报文段的往返时延测量值。另外，一旦重发开始，Karn算法将重发超时时间的计算和上一次传输的往返时延估值区分开，并使每个重发的超时时间翻倍。

为了实现Karn算法，软件需要保存三种信息。第一，它要保存重发超时的时限，该值可从当前往返时延的估值计算得出。第二，它必须保存一个指出TCP是否已经开始重发的标志。第三，它应当保存一个重发次数计数器。范例代码在TCB的字段中保存了这些值。字段tcb_rexmt保存了重发定时器的当前值。如果重发已经开始，字段tcb_ostate中的内容为TCPO_REXMT。最后，字段tcb_rexmtcount记录当前重发次数。

15.4.2 重发输出状态的处理

过程tcprexmt实现了由Karn算法规定的重发的计算。

```
/* tcprexmt.c - tcprexmt */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcprexmt - handle TCP output events while we are retransmitting
 *------------------------------------------------------------------------
 */
int tcprexmt(tcbnum, event)
int tcbnum, event;
{
```

```
struct     tcb  *ptcb = &tcbtab[tcbnum];

if (event != RETRANSMIT)
     return OK; /* ignore others while retransmitting     */
if (++ptcb->tcb_rexmtcount > TCP_MAXRETRIES) {
     tcpabort(ptcb, TCPE_TIMEDOUT);
     return OK;
}
tcpsend(tcbnum, TSF_REXMT);
tmset(tcps_oport, TCPQLEN, MKEVENT(RETRANSMIT, tcbnum),
     min(ptcb->tcb_rexmt<<ptcb->tcb_rexmtcount, TCP_MAXRXT));
if (ptcb->tcb_ostate != TCPO_REXMT)
     ptcb->tcb_ssthresh = ptcb->tcb_cwnd; /* first drop   */
ptcb->tcb_ssthresh = min(ptcb->tcb_swindow,ptcb->tcb_ssthresh)/2;
if (ptcb->tcb_ssthresh < ptcb->tcb_smss)
     ptcb->tcb_ssthresh = ptcb->tcb_smss;
ptcb->tcb_cwnd = ptcb->tcb_smss;
return OK;
}
```

tcprexmt对应于RETRANSMIT输出状态，一旦在“重发”状态时有某个定时器事件发生，TCP输出进程调用tcprexmt。由于此时该连接开始重发操作，因此类似SEND这样的事件就不能被处理，所以tcprexmt忽略除RETRANSMIT事件以外的任何事件。

tcprexmt将字段tcb_rexmtcount中的重发计数值加1，并通过将重发次数与常量TCP_MAXRETRIES相比较，来得知是否已达最大重发次数。当该值达到最大允许值时，tcprexmt调用tcpabort终止该连接，并向其传递差错代码TCPE_TIMEDOUT。在tcprexmt检查过差错之后，它调用tcpsend重发输出缓冲区中剩余的未被确认的数据。tcpsend的第二个入口参数指出本次调用是为了重发数据。

一旦tcprexmt重新传输该数据，它必须为另一个重发定时器事件定时。对tmset的调用实现了Karn算法中的定时器控制。它将tcb_rexmt中的值向左移动tcb_rexmtcount个比特，这样，每发生一次重发，其时延就乘以2。然后将计算出的时延作为参数传递给tmset，引起它为一个新的RETRANSMIT事件定时。

对于一个较小的TCP_MAXRETRIES值，这种将每次重发超时时限乘以2的方式工作良好。但如果系统允许一个较大的重发次数，那么这种方式将导致TCP在决定终止连接之前，其重发时延将大得极为可观。为了防止超时时限变得任意长，tcprexmt为最大超时时限选取一个值，这个值就是计算出的超时时限和常量TCP_MAXRXT值两者中较小的值。

15.7.1节讨论tcprexmt中最后几行语句，它们用以处理拥塞控制。

15.5 基于窗口的流量控制

当接收方机器上的TCP发送一个确认时，该报文段中包括了一个“窗口通告”（window advertisement），使发送方得知接收方还有多大的缓冲空间用以存放新增的数据。窗口通告总是指出接收方还可以接受多少未被确认的数据。TCP还规定，一旦接收方通告一个特定的窗口值，它可以

不再通告比该窗口更小的值（即，窗口永远不收缩）。显然，当发送方不断向通告的窗口中填入报文段时，确认字段的值也不断增加，而窗口字段的值可能越来越小，直至为0。然而，如果接收方已经同意接收序列空间中某一位置上的数据，它可以永远不降低该位置的值。因此，在发送方提供数据且确认序号不断增加的情况下，窗口通告才有可能减小。也就是说，不是仅仅因为接收方决定减小其缓冲区大小才减小窗口通告。

TCP利用窗口通告来控制一个连接上的数据流量。接收方可通过通告一个较小的窗口大小来限制发送方生成的数据量。在极端的情况下，通告一个大小为零的窗口可完全终止数据传输[1]。

15.5.1 糊涂窗口综合症

如果接收方只要缓冲区有剩余空间，就马上通告该缓冲空间，则可能导致被称为"糊涂窗口综合症"（silly windows syndrome）的行为。糊涂窗口行为被认为是当接收方窗口在0到一个很小的正值之间摆动不定时产生的一种情况，此时发送方在窗口打开的同时，传送几个小报文段就又填满窗口。这种行为导致很低的网络利用率，因为与位于报文段前部的TCP和IP首部相比较，报文段中包含的数据量非常少。

为了防止在传输中某对端TCP发生糊涂窗口综合症，TCP应用一种称为"接收方预防糊涂窗口"（receiver-side silly window avoidance）技术。预防糊涂窗口的规则就是说一旦接收方通告了一个零窗口，则它必须延迟对下一个非零窗口的通告，直至它具有"非平凡"（nontrival）的缓冲区空间时。一个"非平凡"的缓冲区空间是指存放最大报文段的空间，或四分之一个缓冲区的空间这两者中较大的一个值。

15.5.2 接收方预防糊涂窗口

过程tcprwindow在计算一个窗口通告值时实现了接收方预防糊涂窗口技术。

```
/* tcprwindow.c - tcprwindow */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcprwindow - do receive window processing for a TCB
 *------------------------------------------------------------------------
 */
int tcprwindow(ptcb)
struct tcb  *ptcb;
{
        int  window;

        window = ptcb->tcb_rbsize - ptcb->tcb_rbcount;
        if (ptcb->tcb_state < TCPS_ESTABLISHED)
             return window;
```

① 即接收方"关闭"窗口。

```
        /*
         *   Receiver-Side Silly Window Syndrome Avoidance:
         *  Never shrink an already-advertised window, but wait for at
         *  least 1/4 receiver buffer and 1 max-sized segment before
         *  opening a zero window.
         */
        if (window*4 < (int)ptcb->tcb_rbsize ||
            window < (int)ptcb->tcb_rmss)
                window = 0;
        window = max(window, ptcb->tcb_cwin - ptcb->tcb_rnext);
        ptcb->tcb_cwin = ptcb->tcb_rnext + window;
        return window;
}
```

tcprwindow 首先计算等于有效缓冲空间的窗口大小（即，接收缓冲区的大小减去当前缓冲区中已有数据量）。如果TCP刚刚开始三次握手动作，但还没有建立连接（状态号比TCPS_ESTABLISHED状态号小），此时接收方的最大报文段长度还没有初始化。因此 tcprwindow 不能应用接收方预防糊涂窗口技术，它只是保存TCB的tcb_window字段中的值，该值用于确定窗口大小，并将其返回调用者。一旦连接建立，tcprwindow 应用接收方预防糊涂窗口的规则，也就是除非有效空间是一个“不算小”的空间，否则 tcprwindow 将窗口值减至零。

tcprwindow 中最后几行语句是在窗口通告值中应用了拥塞预防技术，将在下一节中讨论。

15.5.3　零窗口之后的性能优化

一旦接收方通告一个零窗口，则发送方进入PERSIST输出状态，并开始探询接收方窗口[①]。接收方为每次的探询响应一个确认。只要窗口还是关闭的，就继续不断地探询，而确认中包含的窗口通告值一直为0。最后，当有足够的空间可用时，确认中将携带一个非零窗口值，此时发送方开始发送新的数据。

虽然探询零窗口的责任主要是由发送方来担负，但在接收方的一个小小的优化措施可改善其性能。该优化措施包括安排接收方在不等待下一个探询的情况下，生成一个“无偿确认”（gratuitous acknowledgement），其中包含了新窗口大小。因此，在范例中，只要一个应用程序从 TCP 输入缓冲区中提取了数据，它就检查因此而增加的空间是否可导致窗口打开，如果可以，就发送一个无偿确认。在发送方处理该确认时，它会发现该窗口通告值非零，从而导致它回到 TRANSMIT 状态，并恢复数据传输。

15.5.4　调整发送方的窗口

过程tcpswindow计算发送方的窗口大小。它处理输入报文段中的窗口通告值，并跟踪掌握对方 TCP 能够接收的数据量。

```
/* tcpswindow.c - tcpswindow */

#include <conf.h>
```

① 第 13 章已讨论了输出处理和输出状态机。

```
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpswindow -  handle send window updates from remote
 *------------------------------------------------------------------------
 */
int tcpswindow(ptcb, pep)
struct tcb   *ptcb;
struct ep    *pep;
{
        struct ip   *pip = (struct ip *)pep->ep_data;
        struct tcp  *ptcp = (struct tcp *)pip->ip_data;
        tcpseq          wlast, owlast;

        if (SEQCMP(ptcp->tcp_seq, ptcb->tcb_lwseq) < 0)
             return OK;
        if (SEQCMP(ptcp->tcp_seq, ptcb->tcb_lwseq) == 0 &&
            SEQCMP(ptcp->tcp_ack, ptcb->tcb_lwack) < 0)
             return OK;
        /* else, we have a send window update */

        /* compute the last sequences of the new and old windows */

        owlast = ptcb->tcb_lwack + ptcb->tcb_swindow;
        wlast = ptcp->tcp_ack + ptcp->tcp_window;

        ptcb->tcb_swindow = ptcp->tcp_window;
        ptcb->tcb_lwseq = ptcp->tcp_seq;
        ptcb->tcb_lwack = ptcp->tcp_ack;
        if (SEQCMP(wlast, owlast) <= 0)
             return OK;
        /* else,  window increased */
        if (ptcb->tcb_ostate == TCPO_PERSIST) {
             tmclear(tcps_oport, MKEVENT(PERSIST, ptcb-&tcbtab[0]));
             ptcb->tcb_ostate = TCPO_XMIT;
        }
        tcpkick(ptcb);              /* do something with it */
        return OK;
}
```

在TCB中，字段tcb_swindow随时指出在当前被确认的序号之后TCP还能发送多少字节。也就是说，它的值来自接收到的最新窗口通告。然而，由于报文段可能是不按顺序到达的，因此，TCP必须在更新tcb_swindow的值时十分小心。它必须证实输入的报文段是在上一个用来更新窗口值的报文段之后生成的。要做到这一点，当它更新窗口值时，会保存该报文段中的序号字段（tcb_lwseq）和确认字段（tcb_lwack）。

当一个报文段到达时，tcpswindow 将该报文段中的序号和确认字段中的值与保存在 TCB 中的值相比较。如果序号字段中的值比保存的序号值小，则该报文段是未按顺序到达的，并且它的窗口通告被否认。另外，如果报文段中的序号与保存的序号值相符，但报文段中的确认序号小于保存的确认序号，则该确认报文是未按顺序到达的，因此窗口通告仍然无效。当 tcpswindow 断定该报文段含有一个有效的通告时，它在字段 tcb_swindow 中保存新的窗口大小，并更新其保存的序号和确认序号值。

15.6　最大报文段长度的计算

当 tcprwindow 应用接收方预防糊涂窗口技术时，它必须知道可能到达的最大有效报文段的长度。另外，当TCP为携带数据而生成报文段时，它必须将这些报文段的长度限制在该连接允许发送的最大报文段长度（MSS）之内。TCP 在三次握手操作的互相交换请求阶段，同时为传入和外发报文段协商其 MSS。一旦 TCP 为这两个方向各自确定一个 MSS 之后，就不再改变它们的值。

15.6.1　发送方的最大报文段长度

为了理解范例TCP软件是如何为输出选取一个最大报文段长度的，让我们回顾第 12 章中的过程 tcpwinit[1]。tcpwinit 为最大报文段长度（MSS）计算其初始值，并将该值保存在TCB的 tcb_smss 字段中。为了更好地避免 IP 数据报分片，主机需求文档规定，如果该连接路过一个网关，则 TCP 必须使用536八位组作为最大报文段长度初始值。对位于直接相连的网络中的连接，TCP选取一个初始值，使网络分组尽可能充满（即，它从本地网络的、用来到达远程机器的 MTU 中，减去 TCP 和 IP首部的长度来计算最大数据长度初始值）。tcpwinit通过查找到达目的站的路由的度量是否比0大，来判断该连接是否要经过一个网关。

在选取了 MSS的值之后，TCP处理传入SYN 报文段中的最大报文段长度选项。过程 tcpsmss 处理有关 MSS 选项的细节。

```
/* tcpsmss.c - tcpsmss */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpsmss - set sender MSS from option in incoming segment
 *------------------------------------------------------------------------
 */
int tcpsmss(ptcb, ptcp, popt)
struct tcb   *ptcb;
struct tcp   *ptcp;
u_char       *popt;
```

① 当一个远程客户和一个本地服务器建立连接时，由 tcpwinit 初始化 MSS，当一个远程服务器和一个本地客户建立连接时，另有类似的一段代码初始化 MSS。

```
{
        unsigned  mss, len;

        len = *++popt;
        ++popt;         /* skip length field */
        if ((ptcp->tcp_code & TCPF_SYN) == 0)
             return len;
        switch (len-2) {      /* subtract kind & len     */
        case sizecf(char):
             mss = *popt;
             break;
        case sizeof(short):
             mss = net2hs(*(unsigned short *)popt);
             break;
        case sizeof(long):
             mss = net2hl(*(unsigned long *)popt);
             break;
        default:
             mss = ptcb->tcb_smss;
             break;
        }
        mss -= TCPMHLEN;      /* save just the data buffer size */
        if (ptcb->tcb_smss)
             ptcb->tcb_smss = min(mss, ptcb->tcb_smss);
        else
             ptcb->tcb_smss = mss;
        return len;
}
```

最大报文段长度只能在三次握手阶段商定。因此，除非在携带该选项的报文段中，SNY比特为1，否则tcpsmss忽视该选项。这时，它根据选项值占用的八位组数来选择四种情况中的一种。实现方案支持8、16或32位的MSS选项值[①]。tcpsmss提取选项值，并将其转换为本机字节顺序。在其他情况下，tcpsmss用TCB中的MSS初始值代替。最后，在从选项中提取了MSS值之后，tcpsmss将其与TCB中的MSS初始值相比较，并使用两者中较小的一个。因此，tcpsmss永远不会允许利用传入报文段中的MSS选项来增加MSS初始值。

15.6.2 选项处理

过程tcpopts处理有关选项的问题，并调用tcpsmss来提取MSS选项。

```
/* tcpopts.c - tcpopts */

#include <conf.h>
#include <kernel.h>
```

① 许多实现方案只支持16位选项值。

```
#include <network.h>

/*-----------------------------------------------------------------------
 *  tcpopts - handle TCP options for an inbound segment
 *-----------------------------------------------------------------------
 */
int tcpopts(ptcb, pep)
struct tcb   *ptcb;
struct ep    *pep;
{
        struct  ip      *pip = (struct ip *)pep->ep_data;
        struct  tcp     *ptcp = (struct tcp *)pip->ip_data;
        u_char          *popt, *popend;
        int             len;

        if (TCP_HLEN(ptcp) == TCPMHLEN)
                return OK;
        popt = ptcp->tcp_data;
        popend = &pip->ip_data[TCP_HLEN(ptcp)];
        do {
                switch (*popt) {
                case TPO_NOOP:  popt++;
                                /* fall through */
                case TPO_EOOL:  break;
                case TPO_MSS:
                        popt += tcpsmss(ptcb, ptcp, popt);
                        break;
                default:
                        poptt++; /* skip option code */
                        if (*popt > 0 && *popt <= popend - popt -1)
                                popt += *popt -1;
                        else
                                popt = popend; /* bogus option length */
                        break;
                }
        } while (*popt != TPO_EOOL && popt<popend);

        /* delete the options */
        len = pip->ip_len-IP_HLEN(pip)-TCP_HLEN(ptcp);
        if (len)
                memcpy(ptcp->tcp_data,&pip->ip_data[TCP_HLEN(ptcp)],len);
        pip->ip_len = IP_HLEN(pip) + TCPMHLEN + len;
        ptcp->tcp_offset = TCPHOFFSET;
        return OK;
}
```

由于目前的TCP标准规定仅有一个实用的选项MSS，因此程序代码非常简单。除了MSS选项之外，tcpopts还必须处理标志着无操作（no-operation）和选项结束（end of options）的代码。

一旦tcpopts遇到选项结束代码时，它就把数据部分从报文段中移走，以删除整个选项字段，并修正报文段首部中的长度字段。删除选项字段后，TCP软件中的过程才有可能为数据部分设定一个固定的偏移量。

15.6.3 通告一个最大输入报文段长度

过程tcprmss在SYN报文段中创建最大报文段长度选项。它假设最大报文段长度已经计算出并保存在TCB中。

```
/* tcprmss.c - tcprmss */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcprmss - set receive MSS option
 *------------------------------------------------------------------------
 */
int tcprmss(ptcb, pip)
struct tcb   *ptcb;
struct ip    *pip;
{
        struct  tcp     *ptcp = (struct tcp *)pip->ip_data;
        int             mss, hlen, olen, i;

        hlen = TCP_HLEN(ptcp);
        olen = 2 + sizeof(short);
        pip->ip_data[hlen] = TPO_MSS;           /* option kind       */
        pip->ip_data[hlen+1] = olen;            /* option length     */
        mss = ptcb->tcb_smss;
        for (i=olen-1; i>1; i--) {
                pip->ip_data[hlen+i] = mss & LOWBYTE;
                mss >>= 8;
        }
        hlen += olen + 3;     /* +3 for proper rounding below */
        /* header length is high 4 bits of tcp_offset, in longs   */
        ptcp->tcp_offset = ((hlen<<2) & 0xf0) | ptcp->tcp_offset & 0xf;
}
```

选项的创建非常简单。该选项由一个含有TPO_MSS的八位组、一个给出选项长度的八位组以及任意一个包含最大报文段长度的并以网络字节顺序存放的二进制整数组成。tcprmss修正报文段中TCP首部的长度，使之包括选项八位组的长度。

15.7　网络拥塞预防与控制

当网络发生拥塞时，时延增加，导致 TCP 不得不重发报文段。在最糟糕的情况下，重发将使拥塞更加恶化，并导致称为拥塞崩溃（congestion collapse）的结果。为了避免拥塞的恶化，目前协议标准规定TCP应当采用一些策略，在分组延迟或丢失发生时减少分组的重发量。其中首选策略称作“成倍递减”（multiplicative decrease）。

15.7.1　成倍递减法

成倍递减法的基本思想很简单：发送方的TCP维护管理一个称为拥塞窗口（congestion window）的内部变量，该变量用于限制发送的数据量。在传输时，TCP选用接收方通告的窗口值和内部拥塞窗口值中较小的一个来决定发送的数据量。

为了计算拥塞窗口的大小，先假设重发的次数提供了互联网中对拥塞的一种度量方式。在没有拥塞或丢失情况发生时，拥塞窗口的值被设置为接收方通告的窗口值。也就是说，利用接收方通告的窗口大小来决定发送多少数据。当拥塞开始时（即当发生重发事件后），就将拥塞窗口减小。在实际中，每次重发事件的发生，都会将拥塞窗口减小一半，但永远也不能将其减小到小于一个报文段所要求的大小。

如 15.4.2 节中所示，过程 tcprexmt 实现了成倍递减策略，在程序代码中，变量 tcb_ssthresh 的值或者设置为窗口大小通告值（tcb_swindow）的一半，或者设置为当前拥塞窗口（tcb_cwnd）的一半，取决于这两个值中哪一个更小。然后它查看并证实计算出的值不小 1 个 MSS。

虽然这种技术的名称中有“成倍”一词，但当以丢失报文段的数目作为对拥塞状况的一种度量方式时，拥塞窗口阈值将以指数形式减小。第一次报文段丢失，使拥塞窗口减小为原来窗口的一半，第二次报文段丢失时，为原来窗口的四分之一，第三次时为八分之一，以此类推。

15.8　慢启动和拥塞预防

15.8.1　慢启动

我们曾说过，当传输 TCP 报文段的互联网变得拥挤时，随之而来的重发报文段将使情况更加恶化。为了有助于从拥塞中恢复，目前协议标准要求 TCP 降低其传输速率。在实际中，TCP 假设分组丢失是因为网络拥塞造成的，并在网络恢复阶段应用称为慢启动（slow-start）的技术。为了进一步改善性能，避免在拥塞的情况下继续建立新连接，一旦TCP开始在一条新建立的连接上发送数据时，就必须应用慢启动技术。

慢启动恰好与成倍递减技术相反，它提供了成倍递增技术[①]。其原理一样很简单：拥塞窗口的初始值是一个最大报文段的长度（MSS），并发送一个报文段。如果此次通信成功，且在重发定时器超时之前接收到相应的确认报文，该拥塞窗口尺寸就变为两个报文段长度（即原窗口尺寸乘以 2）。每当有一个确认按时抵达，则拥塞窗口尺寸就增大一个报文段，如此继续下去，所以如果在第

① 慢启动这个名字起得不好，因为如果没有丢失的情况发生时，它迅速增加数据流量；它只是在丢失持续不断时才启动速度很慢。

二次传输时，两个报文段都成功地传送至对方，则拥塞窗口大小将增加到4个报文段，并且它将继续成倍增长，直至达到由成倍递减设置的上限值。

15.8.2 超过上限后减速递增

一旦拥塞窗口大小达到上限值之后，TCP减慢其增长速度。这时TCP在每经过一个往返时延后，才将拥塞窗口尺寸增大一个报文段长度，而不是每次接收到一个确认就增大一个报文段长度。为了估计往返时间，程序代码利用了一个窗口中发送所有数据，且接收到所有相应的确认所需要的时间。当然，TCP不会等待整个窗口中的数据全部发送和确认之后，再增大拥塞窗口大小。事实上，每次接收到一个确认，拥塞窗口就稍微增大一些。这个增量大小的选取应能做到，当整个窗口中的数据都被发送和确认之后，增量之和大约为一个报文段。要理解这种思想是如何转换成程序代码的，不妨考虑一个TCP最大发送报文段长度，刚才所说希望拥塞窗口的每次增量为：

增量 = 报文段长度 / 窗口尺寸

由于系统已发生了拥塞，因而目前窗口被限制为拥塞窗口大小，这意味着TCP递增窗口的次数取决于拥塞窗口中容纳的报文段数目。

窗口中的报文段数目 = 拥塞窗口大小 / 最大报文段长度

我们用N来表示窗口中报文段数目。为了使整个窗口中的数据发送并确认后，窗口的总增量正好是一个报文段长度，TCP在每接收到一个确认后就将窗口增大1/N个报文段，一共有N个确认。因此，当一个确认到达时，TCP就将窗口大小递增：

增量 = （一个报文段长度 /N）
= （MSS 个字节 /N）
= MSS /（拥塞窗口大小 /MSS）

或

增量 = （MSS * MSS）/ 拥塞窗口大小

15.8.3 递增拥塞窗口尺寸的实现

过程tcprtt的最后几行语句实现了当确认到达时拥塞窗口大小的递增[1]。

```
/* tcprtt.c - tcprtt */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcprtt - do round trip time estimates & cong. window processing
 *------------------------------------------------------------------------
 */
```

① tcprtt余下的代码完成重发定时器时间的估计工作，在后面会详细讨论。

```
int tcprtt(ptcb)
struct tcb  *ptcb;
{
        int  rrt,       /* raw round trip          */
             delta;          /* deviation from smoothed*/

        rrt = tmclear(tcps_oport, MKEVENT(RETRANSMIT, ptcb-&tcbtab[0]));
        if (rrt != SYSERR && ptcb->tcb_ostate != TCPO_REXMT) {
             if (ptcb->tcb_srt == 0)
                  ptcb->tcb_srt = rrt<<3;    /* prime the pump */
             /*
              * "srt" is scaled X 8 here, so this is really:
              *   delta = rrt - srt
              */
             delta = rrt - (ptcb->tcb_srt>>3);
             ptcb->tcb_srt += delta;         /* srt = srt + delta/8     */
             if (delta < 0)
                  delta = -delta;
             /*
              * "rtde" is scaled X 4 here, so this is really:
              *   rtde = rtde + (|delta| - rtde)/4
              */
             ptcb->tcb_rtde += delta - (ptcb->tcb_rtde>>2);
             /*
              * "srt" is scaled X 8, rtde scaled X 4, so this is:
              *   rto = srt + 2 * rtde
              */
             ptcb->tcb_rexmt = ((ptcb->tcb_srt>>2)+ptcb->tcb_rtde)>>1;
             if (ptcb->tcb_rexmt < TCP_MINRXT)
                  ptcb->tcb_rexmt = TCP_MINRXT;
        }
        if (ptcb->tcb_cwnd < ptcb->tcb_ssthresh)
             ptcb->tcb_cwnd += ptcb->tcb_smss;
        else
             ptcb->tcb_cwnd += (ptcb->tcb_smss * ptcb->tcb_smss) /
                  ptcb->tcb_cwnd;
        return OK;
}
```

tcprtt 中最后的 if 语句处理了拥塞窗口大小的递增。如果变量 tcb_cwnd 比 tcb_ssthresh 小，则过程 tcprtt 提供慢启动技术，它每次为拥塞窗口增加一个最大报文段长度。如果拥塞窗口已超过了上限值，则 tcprtt 应用累加增量逐渐增加拥塞窗口大小，增量值为（MSS * MSS / tcb_cwnd）。

15.9 平均往返时间估值及超时

从一开始，研究者就已意识到TCP的性能取决于它能否有效地估算一个连接上数据的平均往返时延。理解这个问题的最好办法是，想像经过一段时间后，我们获得了往返时间样本值的一个序列。TCP利用过去的样本值来估计现在的往返时延，并由该往返时延计算并选择重发超时时间。因为经过一段时间之后，所得到的一组往返时延样本值各不相同，所以TCP对近期的样本值要比对先前的样本值更为重视。然而当网络发生拥塞时，各别往返时延的样本值可能与正常值相去甚远，因而TCP也不能完全忽视先前的样本值。

对性能的研究表明，如果TCP不仅估算往返时延的平均值，同时也估算其方差，则TCP将表现出极高的数据吞吐量。掌握往返时延的方差，使TCP能够计算出可适应预期波动的超时时间，而无需做不必要的重发。目前的协议标准规定使用改进后的往返时延估算技术，我们将在本节中对此加以描述。

15.9.1 一种快速平均值更新算法

如果为了计算往返时延平均值和方差而保存了许多旧的样本值，这种做法显得很笨拙，因为有更好的增量算法存在。那就是，TCP保存一个“滚动平均值”（running average），每当TCP获得一个新的测量值（measurement）后，就更新该“滚动平均值”。例如，它可以用以下公式来计算并更新该平均值（average）：

$$\text{error} = \text{measurement} - \text{average}$$

并且

$$\text{average} = \text{average} + \delta * \text{error}$$

其中δ是一个小于1的分数。事实上，TCP可以保存一个“滚动平均偏差”（running mean deviation），并利用前面提到的error项来更新该偏差：

$$\text{deviation} = \text{deviation} + \delta(|\text{error}| - \text{deviation})$$

在实际中，实现方案利用2的n次方来使计算的结果按比例减小，并利用整数而非浮点数的算法以提高效率。例如，可以选取：

$$\delta = 1/2^n$$

使程序代码可通过移位操作完成除法运算。n的常规值为3。如果被保存的“平均值”是真正平均值按比例缩小的结果，则代码变为：

```
error = measurement - (average >> 3);
average = average + error;
if (error < 0)
  error =-error;
error = error - (deviation >> 3);
deviation = deviation + error;
```

为了更进一步改善其性能，TCP 选取一个稍微大一些的 δ 值，最后得到的结果是：

```
error = measurement - (average >> 3);
average = average + error;
if (error < 0)
  error =-error;
error = error - (deviation >> 2);
deviation = deviation + error;
retransmission_timer = ((average>>2) + deviation) >> 1;
```

过程 tcprtt 实现了对平均往返时间的估计。当有确认到达时，TCP 调用该过程。因此对 tcprtt 的调用表示重发结束，且输出状态回到 TRANSMIT 状态。

tcprtt 首先调用 tmclear。它完成两项任务：删除该连接上正在等待的重发事件，并计算出从该事件被排定后经过了多少时间。

回想我们曾说过由于确认的含义并不明确，无法为重发的报文段测量其往返时间，并且 Karn 算法因此而忽略了为重发的报文段发送来的确认。tcprrt 使用 Karn 算法，通过判断来确保该往返时间的样本值是有效的，与对应于重发报文段的确认无关（即，TCP 不在 RETRANSMIT 状态）。

如果间隔时间 rtt 是有效的，则 tcprtt 假设它提供了原始的往返时延的样本，并利用它来计算一个新的、经平滑处理后的往返时间值（tcb_srt），同时它还生成新的平均偏差估值（tcb_rtde）。最后，根据新的往返时延来估计当前的平均往返时延，并将其保存在 tcb_rexmt 中。注意，在我们的实现方案中，使用了常量 TCP_MINRXT 作为固定的重发时延最小值，以确保计算得到的值不接近于 0。

15.9.2　传入确认的处理

过程 tcpacked 处理一个传入报文段中的确认。它计算在序列空间中，位于那些由以前的报文段确认的八位组之后的八位组数，并将结果返回调用者。

```
/* tcpacked.c - tcpacked */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpacked - handle in-bound ACKs and do round trip estimates
 *------------------------------------------------------------------------
 */
int tcpacked(ptcb, pep)
struct tcb   *ptcb;
struct ep    *pep;
{
        struct  ip      *pip = (struct ip *)pep->ep_data;
        struct  tcp     *ptcp = (struct tcp *)pip->ip_data;
        int             acked, tcbnum, cacked;
        STATWORD        ps;
```

```
        if (!(ptcp->tcp_code & TCPF_ACK))
             return SYSERR;
        acked = ptcp->tcp_ack - ptcb->tcb_suna;
        cacked = 0;
        if (acked <= 0)
             return 0;  /* duplicate ACK */
        if (SEQCMP(ptcp->tcp_ack, ptcb->tcb_snext) > 0)
             if (ptcb->tcb_state == TCPS_SYNRCVD)
                  return tcpreset(pep);
             else
                  return tcpackit(ptcb, pep);
        tcprtt(ptcb);
        ptcb->tcb_suna = ptcp->tcp_ack;
        if (acked && ptcb->tcb_code & TCPF_SYN) {
             acked--;
             cacked++;
             ptcb->tcb_code &= ~TCPF_SYN;
             ptcb->tcb_flags &= ~TCBF_FIRSTSEND;
        }
        if ((ptcb->tcb_code & TCPF_FIN) &&
            SEQCMP(ptcp->tcp_ack, ptcb->tcb_snext) == 0) {
             acked--;
             cacked++;
             ptcb->tcb_code &= ~TCPF_FIN;
             ptcb->tcb_flags &= ~TCBF_SNDFIN;
        }
        if ((ptcb->tcb_flags & TCBF_SUPOK) &&
             SEQCMP(ptcp->tcp_ack, ptcb->tcb_supseq) >= 0)
                ptcb->tcp_flags &= ~TCBF_SUPOK;
        ptcb->tcb_sbstart = (ptcb->tcb_sbstart+acked) % ptcb->tcb_sbsize;
        ptcb->tcb_sbcount -= acked;
        if (acked) {
             disable(ps);
             if (scount(ptcb->tcb_ssema) <= 0)
                  signal(ptcb->tcb_ssema);
             restore(ps);
        }
        tcpostate(ptcb, acked+cacked);
        return acked;
}
```

如果调用tcpacked后，发现报文段中没有包含确认位，则tcpacked返回一个差错代码。如果其中包含有确认，tcpacked用报文段的确认序号减去TCB中保存的第一个未被确认的数据序号，计算得到此次被确认的八位组数目。如果确认提供的序号比当前记录的第一个未被确认数据的序号还要小，则该报文段必定是被复制了的，或没有按顺序到达，因此tcpacked返回0，表示此次并没有八

位组被确认。

tcpacked中还包括了对一种特殊情况的检查，那就是当确认提供的序号超出了已发送数据的序号时的情况。对大多数状态来说，协议规定TCP必须确认这些报文段。因此，tcpacked调用tcpackit来生成一个确认，然而在SYN-RECEIVED状态中，一个超过当前序号的确认报文意味着一个不正确的三次握手操作，而且必须用RESET响应。

一旦tcpacked查看并肯定了该确认在正常的范围内，则它调用tcprtt来更新往返时延估值，并计算一个新的超时重发时间。同时，它还要更新字段tcb_suna，其中包含的是第一个未被确认的数据的序号。

tcpacked处理以下两种特殊情况：FIN和SYN处理。理论上讲，它们都位于序号空间内。tcpacked通过清除TCB中的TCBF_FIRSTSFND位来记录下SYN的存在，同时它还要将返回调用者的确认计数器的值减1，因为该计数器仅指出数据量而不包括SYN。类似地，如果报文段确认了该连接的FIN，则tcpacked清除TCB代码标志段中的TCPF_FIN位。

tcpacked的最后一段更新TCB中的变量，以反映由于该确认到达而引起的变化。它要控制计数器和缓冲区指针，以删除发送缓冲区中已被确认的外发数据。首先，它将缓冲区指针（tcb_sbstart）向前移动acked个位置，如果遇到缓冲区尾部，则绕回缓冲区的头部。其次它将缓冲区中数据的计数器（tcb_sbcount）减去已被确认的八位组数。第三，如果被确认的数据已从缓冲区中删除，且有一个或多个应用程序正因等待缓冲区空间而被阻塞，tcpacked释放发送缓冲区信号量，以允许下一个程序向缓冲区中写入数据。

15.9.3　为窗口外的数据创建确认报文段

曾经提到，协议要求TCP为不正确的输入确认报文段创建一个响应确认。特别是，如果远程确认的数据超出了当前的输出窗口时，TCP必须发送一个确认。过程tcpackit为此类情况创建确认报文段。

```
/* tcpackit.c - tcpackit */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpackit -  generate an ACK for a received TCP packet
 *------------------------------------------------------------------------
 */
int tcpackit(ptcb, pepin)
struct tcb  *ptcb;
struct ep   *pepin;
{
        struct ep  *pepout;
        struct ip  *pipin = (struct ip *)pepin->ep_data, *pipout;
        struct tcp *ptcpin = (struct tcp *)pipin->ip_data, *ptcpout;

        if (ptcpin->tcp_code & TCPF_RST)
```

```
        return OK;
    if (pipin->ip_len <= IP_HLEN(pipin) + TCP_HLEN(ptcpin) &&
            !(ptcpin->tcp_code & (TCPF_SYN|TCPF_FIN)))
        return OK; /* duplicate ACK */
    pepout = (struct ep *)getbuf(Net.netpool);
    if ((int)pepout == SYSERR)
        return SYSERR;
    pepout->ep_order =~0;
    pipout = (struct ip *)pepout->ep_data;
    pipout->ip_src = pipin->ip_dst;
    pipout->ip_dst = pipin->ip_src;
    ptcpout = (struct tcp *)pipout->ip_data;
    ptcpout->tcp_sport = ptcpin->tcp_dport;
    ptcpout->tcp_dport = ptcpin->tcp_sport;
    ptcpout->tcp_seq = ptcb->tcb_snext;
    ptcpout->tcp_ack = ptcb->tcb_rnext;
    ptcpout->tcp_code = TCPF_ACK;
    ptcpout->tcp_offset = TCPHOFFSET;
    ptcpout->tcp_window = tcprwindow(ptcb);
    ptcpout->tcp_urgptr = 0;
    ptcpout->tcp_cksum = 0;
    tcph2net(ptcpout);
    pepout->ep_order &= ~EPO_TCP;
    ptcpout->tcp_cksum = tcpcksum(pepout, TCPMHLEN);
    TcpOutSegs++;
    return ipsend(pipout->ip_src, pepout, TCPMHLEN IPT_TCP,
            IPP_NORMAL, IP_TTL);
}
```

为了防止进入无限循环，tcpackit对RESET报文段不做响应。对其余所有报文段，它分配一个网络缓冲区，填写TCP和IP首部，将ACK位置为1（即字段tcp_code的TCPF_ACK位）。然后调用ipsend将该数据报发往它的目的站。

15.9.4 接收到一个确认后改变输出状态

当一个确认到达后，TCP必须查看TCB，以决定其输出状态是否应当改变。如果所有正在等待的数据都被确认了，则输出状态变迁到“闲置”状态。如果有某些正在等待的数据被确认，则输出从“重发”状态变迁到“传输”状态。tcpacked调用过程tcpostate来完成必要的判断。

```
/* tcpostate.c - tcpostate */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```

```
/*------------------------------------------------------------------------
 *  tcpostate - do TCP output state processing after an ACK
 *------------------------------------------------------------------------
 */
int tcpostate(ptcb, acked)
struct tcb  *ptcb;
int         acked;
{
        if (acked <= 0)
                return OK; /* no state change */
        if (ptcb->tcb_ostate == TCPO_REXMT) {
                ptcb->tcb_rexmtcount = 0;
                ptcb->tcb_ostate = TCPO_XMIT;
        }
        if (ptcb->tcb_sbcount == 0) {
                ptcb->tcb_ostate = TCPO_IDLE;
                return OK;
        }
        tcpkick(ptcb);
        return OK;
}
```

tcpostate 有两个入口参数，一个是指向 TCB 的指针，另一个是最近一次接收到的 ACK 确认的数据的数量。如果没有数据被新近确认，则状态不变。tcpostate 查看参数 acked，如果其值为 0，则立刻返回。

然后，tcpostate 检查由字段 tcb_ostate 给出的当前输出状态。如果连接原来在“重发”状态，则它变迁到“传输”状态。最后，如果还有悬而未决的数据正在等待，则 tcpostate 调用 tcpkick 初始化一个“传输”事件。

15.10　技巧和注意事项

过程 tcprtt 还要利用“聚丛”技术[1]协助发送方预防“糊涂窗口”，其具体实现是：在为另一个 RETRANSMIT 事件定时之前，先检查是否还有正在等待中的 RETRANSMIT 事件。其思想是避免不必要的传输，因为这些传输会触发一系列的确认。如果对方没有实现“预防糊涂窗口”技术，则确认可能会报告一个很小的窗口尺寸增量，反过来，这个值可能会导致发送方传输小报文段。因此，避免不必要的确认有助于保留网络带宽资源并防止糊涂窗口行为。

15.11　小结

自适应重发是 TCP 的关键部分，它使 TCP 能够在各种不同的底层网络上运行。为了使 TCP 更

① 发送方“糊涂窗口”预防技术自从作者在 RFC 中提出后也称为 Nagle 算法。

健壮并有效，自适应重发算法在以下几个方面进行调整：重发定时器退避、基于窗口的流量控制、最大报文段长度计算、拥塞的防止和控制以及平均往返时延的估计。

经过不断的实验和分析，已得出一些技术和方法，它们目前被TCP协议标准建议使用或要求使用。尽管每种技术仅仅需要几行程序代码就能实现，但当它们结合在一起时，就会为提高吞吐量和降低时延作出巨大贡献。

深入研究

Postel [RFC 793] 中含有TCP标准原型，其中包括对自适应重发的描述。本章中出现的许多技术在最近的RFC中已做了补充。Braden[RFC 1122]在标准中加入了很大的变化。Clark[RFC 813和816]描述了窗口管理和故障恢复。Postel[RFC 879]评述了最大报文段长度。由Nagle[RFC 896] 提供了预防糊涂窗口的技术。Karn和Partridge [1987] 详述TCP性能方面的改善。包括往返时延的估计和Karn的算法。Jacobson [1988] 给出了拥塞控制算法，它已成为标准中要求的一部分。Mills[RFC 889]讨论了Internet往返时延的测量值。Borman [April 1989] 对在Cray计算机上所做的高速TCP的试验做了总结。

习题

15.1 本章曾提到使用发送方预防糊涂窗口和接收方预防糊涂窗口两种技术。如果发送方没有采用预防糊涂窗口的技术，那么接收方的预防糊涂窗口技术工作是否良好？为什么？

15.2 对一个以太网连接（MTU = 1500八位组）来说，假设平均往返时延为3毫秒，试计算慢启动到底限制了多少吞吐量。再计算最初发送的32个分组的吞吐量：（a）使用慢启动；（b）不使用慢启动。

15.3 查看在过程tcpacked中调用tcpostate的代码段，并建议一个优化方案。

15.4 如果发送方有16K字节的缓冲区，且最大报文段长度为1K字节，在拥塞窗口达到1MSS之前，已丢失了多少个确认？

15.5 Karn算法规定，当报文段必须被重发时，应忽略对其往返时延的估算。如果TCP总是认为接收到的确认对应于某报文段的第一次传输，会出现什么情况？如果它认为所有确认都是对应于最近的一次重发，又会出现什么情况？

15.6 阅读Jacobson [1988]，这是一份有关避免拥塞的文件。如果在成倍递减时，使用的倍数为0.8而不是0.5，根据文件的建议将会出现什么情况？

15.7 考虑缓冲区大小对吞吐量的影响。假设TCP的两种不同的实现软件分别使用了16K字节和64K字节的输出缓冲区，而且两种实现软件都是通过一个惟一的、拥塞的路径，向一个具有较大的接收缓冲区的机器传输数据，试问：一个报文段的丢失会分别对两个连接带来多大影响？如果连续丢失10个报文段呢？

15.8 当TCP接收到一份对应于某个活动的连接的ICMP源站抑制报文后，会发生什么？为什么？提示：考虑源站抑制与拥塞之间的关系。

15.9 阅读有关获取MTU的技术，解释TCP将如何利用它们。

15.10 有人说非本地连接要求MSS的值为536会为TCP带来不必要的限制，请提出反对意见。

15.11 如果范例实现中MTU只比输入（或输出）缓冲区的尺寸稍微小一点，它还能在本地网络

上正确工作吗？为什么？

15.12 实现方案应用了主动确认原则（即，它在发送确认时，并不为等待窗口的改变或相反方向的数据流而延迟）。主动确认是如何改善TCP性能的？延迟的确认又将如何影响慢启动或拥塞控制？

15.13 TCP在协议中并没有提供keepalive功能。然而，一些应用程序既使在它们没有发送数据时，也要求知道连接是否出现故障。请在范例实现中加入keepalive的功能。

15.14 一些应用程序更愿意关闭那些已闲置N分钟的连接，其中N是由应用程序给出的参数值。请为闲置连接实现自动关闭机制。

15.15 查看TCP有限机状态机中的每一个状态（图 11.2）。并考虑如果一方的计算机崩溃后没有重启动，连接的另一方会出现什么情况？你认为哪些状态在删除TCB时需要较长的超时时间？

15.16 某些TCP的实现方案中在发送确认报文之前采用小时段时延，这个时延会引起吞吐量的增加吗？为什么？

第16章　紧急数据处理和推功能

16.1　引言

前面的章节讨论了常规的TCP输入输出，也讨论了如何利用经验改善流量控制和自适应重传机制。本章考虑TCP的其余两个方面：TCP发送带外（out-of-band）指示的能力及它越过正常的数据缓存处理的能力。带外指示增强了TCP的能力，并且允许建立异常事件处理程序，如“程序中断”（program interruption）或“程序终止”（program abort）函数。数据缓存是很重要的，它提高了TCP的效率，并且降低了网络额外开销。但是，对数据的缓存处理增加了数据传送的时延。需要立即传送数据的程序可以越过缓存处理，强迫TCP无延迟地传送数据。后面几节将考虑TCP如何提供带外指示，以及一个应用程序如何才能越过数据的缓存处理。

16.2　带外信令

TCP使用一种“数据流模式”（stream paradigm）完成正常数据传送。位于某个已建立起来的连接一端的应用程序生成一个数据流，并将数据流交付给本机上的TCP。TCP将数据流中的数据发送出去，穿过互联网，到达位于连接另一端机器上的TCP模块，最后由该TCP模块交付给其上层应用程序。对于许多应用程序来说，数据流模式工作良好。但它并不能满足所有通信方式的需要，因为它要求接收者必须顺序地处理所有数据。在某些情况下，一个应用程序需要传送带外指示，该指示信令必须越过一般的数据流。例如，当一个程序因执行不当而必须终止时，远程的登录协议需要使用带外指示来通知远程节点。即使在该程序停止处理正常数据流中的数据时，也必须处理发送来的终止指示信令。因此，该信号不能够作为正常数据流的一部分传递。

16.3　紧急数据

TCP使用“紧急数据”（urgent data）这一术语来代表带外指示。在发送一个带外指示时，发送者将报文段首部代码字段中的紧急数据位TCPF_URG置1，将紧急数据置入报文段，并将“紧急数据指针”字段赋值为报文段数据区中的紧急数据结束处的偏移量。由于根据协议标准，报文段首部只设置一个指针，因此报文段中没有明确地指向紧急数据起始处的指针。

当一个携带紧急数据的报文段到达时，接收方必须立即通知接收该报文段的应用程序，它不能等待应用程序处理那些可以稍后处理的正常数据。为了做到这一点，TCP将应用程序置为“紧急模式”（urgent mode），通知应用程序有紧急数据存在。应用程序收到有紧急数据的通知后，就从连接中不断地读取数据直到紧急数据的结尾。最后，TCP通知应用程序紧急数据已经结束。

16.4　标准的解释

有关紧急数据的标准规范是TCP标准中讨论得最少的，也是最难以理解的部分。第一，接收

者如何知道紧急数据是从哪里开始的？第二，TCP 如何通知一个应用程序有紧急数据到达？第三，当所有的紧急数据处理完毕后，TCP 又是如何通知应用程序的？第四，如果携带紧急数据的多个报文段并没有按顺序到达，将会出现什么情况？第五，紧急数据指针是指向紧急数据的最后一个八位组还是指向紧急数据末尾后面一个八位组[①]？此外，标准中假设了一个“报文传递接口”（message-passing interface），但并没有描述许多实现方案中应用“打开-读-写-关闭”（open-read-write-close）模式的TCP语义，而且随后的标准修正也只解决了部分问题，但不管怎样，有关紧急数据的阐释已经有了比较统一的认识。我们将在讨论完一个可选方案后再研究这些默认的观点。

16.4.1　带外数据的解释法

实现者们对于紧急数据的处理有两种看法。第一种，将紧急数据看成是“带外数据”（out-of-band data），这种观点目前已不受重视。从大的方面来看，任何持带外数据观点的实现方案都提供两个独立的数据流：正常数据流和紧急数据流。当紧急数据到达时，TCP 应立即通知应用程序。当应用程序读取数据时，TCP 交付给它的是紧急数据。当应用程序处理完所有紧急数据后，TCP 通知应用程序，再将正常数据交付给它。

要理解带外数据处理时的细节，可以研究一下图 16.1 的概念示意图。

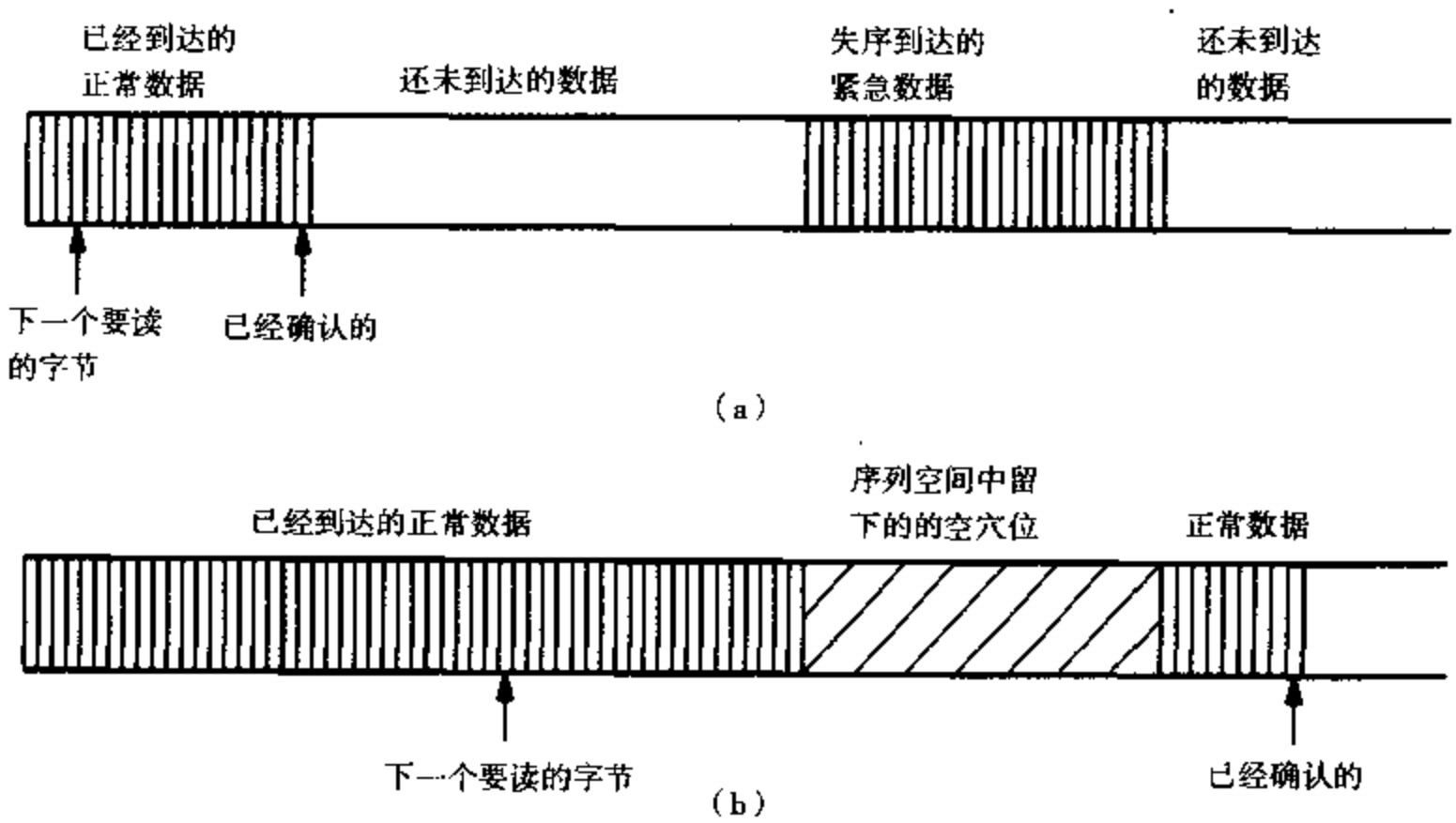


图 16.1　（a）当紧急数据比所有的一般数据早到达时的 TCP 序列空间；（b）同样的序列空间，此时所有的紧急数据已经处理完毕，并且有更多的正常数据到达。由紧急数据占据的空间在序列里形成一个空穴

紧急数据占据正常数据传送时所用的部分序列空间。在大多数情况下，紧急数据与其他数据一起按顺序到达，接收者也顺序地接收和确认数据，直到紧急报文的开头。然而，如 16.1（a）所示，由于下层的 IP 不保证报文段按序传送，一个携带紧急数据的报文段可能比某些携带正常数据的报文段要早到达，虽然这些携带正常数据的报文段在序列空间中可能位于较低的位置。实际上，由于

① 尽管现有标准对于指针问题有很明确的描述（紧急数据指针指向紧急数据的最后一个位置），但大多数的实现方案，包括从 BSD UNIX 中发展起来的方案，都将该指针解释为指向紧急数据后面一个位置。

紧急报文往往很短，而一些网关将传送优先权赋予短数据报，这种非顺序传送的情况是很常见的。

图16.1（b）给出了带外数据解释法带来的后果。紧急数据占据了一部分序列空间，并可以不按序传送。当紧急数据到达时，TCP立即将它交给从连接中读取数据的应用程序。处理完紧急数据后，应用返回到正常的数据处理，继续从紧急数据到达时被中断的地方开始读取数据。其结果是在序列空间中留下了一个曾经由紧急数据占用的“空穴”，当应用程序读取到空穴处时，TCP必须跳过它，好像这块空间不存在一样。

使用带外数据解释法的实现方案需要额外的数据结构。例如，当一个程序跳过正常数据，先读取紧急数据时，TCP需要具有一个保存了地址空间中的空穴的列表。另外，该实现方案必须面对一个基本问题：紧急数据可能位于非现有窗口所对应的序列空间中[1]。由于这些原因，带外数据解释法没有得到广泛应用。

16.4.2 数据标记解释法

我们的例子中的代码使用另一种普遍采纳的紧急数据解释法。这种方案被称为数据标记解释法。使用数据标记解释法的实现方案不在带外传送数据。实际上，它将一个报文段中的紧急数据指针只当作是数据流中某一位置的指针。当携带紧急数据的报文段到达后，接收方TCP记录下数据流中指针的位置，并通知应用程序数据流中包含有紧急数据。应用程序必须从数据流中读取数据，一直读到紧急数据标记。然后TCP通知应用程序，所有的紧急数据已经阅读完毕。

紧急数据的数据标记解释法在实现过程中比带外数据解释法要简单得多。为了实现紧急数据的数据标记解释法，TCP只需要记住收到一个紧急数据指针时序列空间的位置，而不需要将数据分为两个流，也不需要记住紧急数据处理后留下的空穴的位置。

重要的是，数据标记解释法将紧急数据的处理，以及紧急数据与正常数据的区分工作留给应用程序完成。尽管TCP要通知应用程序在数据流的某处有紧急数据到达，但它并不立即将紧急数据交付给应用程序，也不区分以报文段的形式到达的数据流中的哪些数据中含有紧急数据。

数据标记解释法对于那些在紧急数据到达时，可以读取并丢弃正常数据的应用程序是最合适的。例如，一个TELNET服务器接收到一个作为紧急数据的终止（abort）字符时，可以读取并丢弃终止字符前的键盘信号，因为一个终止操作可以结束一个读取数据的程序。然而，对于一个接收并处理完紧急数据后还要继续处理数据流中的正常数据的应用程序来说，它必须要设计成能够在搜索紧急数据的同时存储它所遇到的正常数据。当应用程序读取到紧急数据的结束处后，它必须返回并处理那些已被储存的正常数据。

16.5 为Berkeley紧急指针解释法而进行的配置

为了使我们的程序符合标准解释法，或者符合Berkeley UNIX的紧急指针解释法（即紧急数据指针是指向紧急数据的最后一个八位组还是指向超过末尾的一个八位组），程序代码包含一个配置常量（BSDURG）。当对其定义后，符号常量BSDURG使TCP认为该指针指向紧急数据末尾后面的一个八位组。

当编译代码时，BSDURG必须被定义，而这个定义要适合于整个系统。因此，紧急指针的解释

① 公布接收窗口为0时，这一点对于接收者处理紧急数据尤其重要，因为该紧急数据用于终止一个已停止读取数据的应用程序。

被编译到软件中，而不是分别为每一个单个的连接来设定。我们选择使BSDURG成为全局配置的一个部分，因为TCP不提供一种方法来协商选择或检测使用何种解释法。

16.6　通知应用程序

在实现过程中，采用一种被动方法通知某个应用程序有紧急数据到达。它等待应用程序发送read请求。当紧急数据到达时，应用程序的下一个读取数据请求将使进程转入紧急模式，返回一个特别码TCP_URGENTMODE，随后的read调用以八位组形式返回数据流中的数据，直到所有的紧急数据传递完毕。应用程序处理完所有紧急数据后，下一个read调用将进程转换为正常模式，并返回一个特殊码TCP_NORMALMODE，通知应用程序已到达紧急数据的末尾。如果没有进程调用read过程，紧急数据将不被处理。总结如下：

> 我们的实现方案以被动的形式处理紧急数据。在通知应用程序有紧急数据到达之前，它必须等待某个应用程序从已建立的连接中读取数据。TCP软件对紧急数据既不做任何处理，它也不会生成一个进程来解释紧急报文。

一次read调用可以申请得到任意大小的数据量。一般来讲，大的数据量需求不会带来什么问题。但是对于紧急数据来说，一个发出申请的应用程序可能申请的数据量会超过紧急数据的边界。例如，考虑下列事件序列：

1. 某个连接对应的接收缓冲区中含有10个未被处理的正常数据八位组，序号从1到10。
2. 一个含有紧急数据的报文段到达，分配的序号为11。
3. 一个含有10八位组正常数据的报文段到达，分配的序号从12到21。
4. 应用程序调用read，TCP返回代码TCPE_URGENTMODE，通知该应用程序有紧急数据到达。
5. 该应用程序调用read，并设定一个足够容纳所有21八位组的缓冲区。

标准中规定TCP在转换回正常模式之前，只向应用程序返回紧急数据。在前面所举的例子中，TCP只能够返回开头的11八位组数据。为了保证应用程序在调用read时不会读过紧急数据的结束点，TCP为数据读取增加了一个人为的限制：

> 尽管一个应用程序可以请求读取大量数据，TCP不允许一次对read的调用所读取的数据超过紧急数据的结束点。

16.6.1　多个并发应用程序

尽管一个并发处理系统允许多个进程从一个连接中读取数据，TCP标准并没有给出这些进程如何处理紧急数据的语义。我们决定实现以下策略：对每一个进程，系统保留一个独立的记录，标明该进程是否处于紧急模式。当某个进程从一个TCP连接中读取数据时，系统检查进程和连接的状态。如果该进程执行于正常模式下，它从只含有正常数据的连接中读取报文，TCP则向其传送所请求的数据。此时，如果一个进程执行于紧急模式下，则它从含有到达的紧急数据的连接中读取信息，TCP向其返回紧急数据。然而，如果进程的模式和连接状态不匹配，TCP更改进程状态并返回某个特殊代码。某个执行在正常模式下的进程如果要从一个有紧急数据到达的连接中读取信息，则该进程将进入紧急模式，并接收TCPE_URGENTMODE返回代码。同样，如果某个处于紧急模式下的进

程要从一个已经没有等待处理的紧急数据的连接中读取信息，则该进程进入正常模式，并接收返回代码TCPE_NORMALMODE。然而，所有进程都共享同一份数据流，一个进程读取的八位组不会再交给另一个进程。

关键问题是，并发处理策略假设协调进程共享一个数据流的工作是由程序员来完成的。这也意味着如果多个进程在不加以协调的情况下读取一个数据流，那么有可能一个进程接收到紧急数据到达指示，而与此同时，另外一个进程调用read已取走了该紧急数据。因此，任何允许并发使用一个TCP连接的应用程序都应该知道，当它接收到一个紧急数据指示而进入紧急模式后，所收到的紧急数据长度却有可能是0。

16.7 从TCP中读取数据

既然对紧急数据已有了较全面的了解，那么一个应用程序是如何从TCP中获取数据的，也就变得容易理解。过程tcpgetdata完成这一任务。

```
/* tcpgetdata.c - tcpgetdata */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpgetdata  -  copy data from a TCP receive buffer to a user buffer
 *------------------------------------------------------------------------
 */
int tcpgetdata(ptcb, pch, len)
struct tcb   *ptcb;
u_char       *pch;
unsigned          len;
{
        tcpseq          seq;
        unsigned        cc;

        seq = ptcb->tcb_rnext - ptcb->tcb_rbcount; /* start sequence   */
        if (ptcb->tcb_flags & TCBF_RUPOK) {
                int nbc, ubc; /* normal & urgent byte coutnts  */

                nbc = ptcb->tcb_rnext - ptcb->tcb_rupseq -1;
                if (nbc >= 0) {     /* urgent boundary in buffer    */
                        ubc = ptcb->tcb_rbcount - nbc;
                        if ((int)len >= ubc) {
                                len = ubc;
                                ptcb->tcb_flags &= ~TCBF_RUPOK;
                        }
                }
        }
```

```
for (cc=0; ptcb->tcb_rbcount && cc< len;) {
        *pch++ = ptcb->tcb_rcvbuf[ptcb->tcb_rbstart];
        --ptcb->tcb_rbcount;
        if (++ptcb->tcb_rbstart >= ptcb->tcb_rbsize)
                ptcb->tcb_rbstart = 0;
        ++cc;
}
if (ptcb->tcb_rbcount == 0)
        ptcb->tcb_flags &= ~TCBF_PUSH;
/*
 * open the receive window, if it's closed and we've made
 * enough space to fit a segment.
 */
if (SEQCMP(ptcb->tcb_cwin, ptcb->tcb_rnext) <= 0 &&
                        tcprwindow(ptcb)) {
                        ptcb->tcb_flags |= TCBF_NEEDOUT;
                        tcpkick(ptcb);
        }
        return cc;
}
```

tcpgetdata 首先计算从缓冲区中取出的数据块的第一个八位组序号。为了做到这一点，它从收到的最高一个八位组的序号中减去缓冲区中的八位组数。然后 tcpgetdata 检查 TCBF_RUPOK 位以判定是否有携带紧急数据的报文段到达。如果有，tcpgetdata 必须保证一个 read 操作不会超出紧急数据和正常数据之间的边界。

当有紧急数据存在时，为了限制一个 read 操作的数据读取量，tcpgetdata 首先计算在紧急数据后，缓冲区内还有多少八位组的正常数据，并将结果保存在变量 nbc 中。如果紧急数据的边界在缓冲区内，则 tcpgetdata 通过用缓冲区内的八位组总数减去 nbc，来计算紧急数据的八位组数 ubc。最后，tcpgetdata 将 ubc 与所请求的数据长度相比较，并将长度限制在 ubc 以内。

一旦 tcpgetdata 确定了起始序号并计算出将要传送的数据长度，它就进入主循环。只要缓冲区内还有字符，并且获取到的字符数比调用所要求的数目少，该循环就一直工作。在每一次循环时，tcpgetdata 从输入缓冲区内提取一个八位组的数据，并将其复制到调用者缓冲区内。

tcpgetdata 也包含了第 15 章里描述的窗口操作代码。在它返回之前，tcpgetdata 将检查 tcb_rnext 是否超出了目前的已通告的窗口，同时检查 tcprwindow 是否返回一个非零的窗口通告值。如果窗口通告值非零，tcpgetdata 就将 TCB 中的 TCBF_NEEDOUT 位置 1，并调用 tcpkick 产生一个无偿确认。tcpgetdata 产生的确认与到达的数据无关，TCP 使用它们仅仅是在没有等待探询的情况下向发送者报告，窗口大小已经增加。当然，如果 tcprwindow 计算出窗口大小为 0，tcpgetdata 将不会发送一个不必要的确认。

16.8　发送紧急数据

对紧急数据的不同解释不但会影响 TCP 紧急数据的传入处理，也会影响它的外发处理。特别

是在数据标记解释法里，负责接收的TCP并没有明确区分是否有多次紧急数据到达，它仅仅告诉负责接收的应用程序有关序列空间中紧急数据的最高点，也就是紧急数据的结束处。其结果是，一个使用数据标记来解释紧急数据的发送程序只需要存储一个值，该值标明了序列空间中紧急数据结束的位置。发送TCP进程不需要将外发数据区分为紧急数据流和正常数据流，也不用判别应用程序是否分多次写入紧急数据。也就是说，如果一个发送应用程序通过一个连接发送紧急数据，随后向同一连接发送正常数据，然后再次发送紧急数据，则TCP只需要将第二次发送的紧急数据结尾报告给接收应用程序。因此，发送TCP可以立即更改它的紧急数据地址指针，即使现有的紧急数据还没有被确认，只要到达的报文段中紧急数据指针有一个新的值，接收进程就立即更新紧急数据指针记录。

除了前面讨论的以外，设计思想中还有两个规定帮助我们简化了程序代码。第一，我们的接口层要求应用程序用一次调用写入紧急数据。这就是说，程序必须组装一个完整的紧急报文，并通过一次过程调用交付给TCP。第二，应用程序必须按顺序写入紧急数据。尤其对应用程序来说，它不能够跳过一段距离设置紧急数据标记，然后再填写数据直到数据标记点。因此，紧急数据是与正常数据一起放入TCP输出缓冲区的。

这两个设计规定帮助我们排除了特殊事件，并使我们能够继续使用大多数为处理正常数据输出而编写的程序代码。由于TCP在一次调用中接收一个完整的紧急报文，因此它不需要实现某种特殊的机制，用以在传送之前将所有的报文段汇集起来。由于紧急数据存放于序列空间和输出缓冲区的位置没有特殊性，它的存放可以完全像正常的输出数据一样。还有，由于紧急数据在序列空间中经常与正常数据连续存放，发送进程可以将其视为正常数据进行重发。然而，在传送紧急数据时确实会有一种特殊情况发生：根据协议规定，即使接收者已经关闭了窗口，发送者必须强制性传送紧急数据。

过程tcpsend发送紧急数据[1]。当发送一个报文段时，tcpsend检查TCB内的TCBF_SUPOK位，来判断发送者紧急指针字段是否有效。如果有效，则字段tcb_supseq中包含了通过该连接发送的紧急数据最高一个八位组在序列空间中所处的位置。tcpsend必须将序列空间位置转换成相对偏移量。计算相对偏移量的方法是：tcpsend减去tcp_seq，其中tcp_seq是报文段中第一个八位组的序号。然后tcpsend将结果储存在报文段的紧急数据字段中，并将报文段首部代码字段中的TCPF_URG位设置为1。

16.9 TCP的推功能

通常，TCP都要对输入和输出做缓存处理。当处理输出时，它将尽量多的数据封装在一个报文段中，用于增加吞吐量和减少额外开销。当处理输入时，它从单个报文段中收集数据，并将数据存放在输入缓冲区中，等待应用程序来提取它们。缓存处理改善了整体的效率。对一个主机或是一个网关来说，它减少了上下文切换的次数及过程调用的额外开销。而对计算机系统以外来说，缓存处理使更多的数据放在一个分组内传送，降低了网络的负荷。

总的说来，缓冲通过增加时延来改善吞吐量。在某些时候，由于缓存而引起的时延过长可能会导致应用程序之间的通信出现错误。为了允许应用程序越过缓存处理，TCP提供了一个推（push）功能。发送应用进程执行推请求TCP无延迟地发送所有数据。TCP在发送数据时，会设置报文段代

① 参见13.13节的tcpsend.c源程序清单。

码字段中的push位为1，因此接收者也能够得知该请求。当一个push位为1的报文段携带数据到达时，TCP立即无延迟地通知应用程序有数据到达。

16.10　在失序交付时对推数据的解释

原则上，当一个传入报文段到达时，对push位做出的响应很简单：当一个push位为1的报文段到达时，TCP应该马上使这些数据能够被应用程序使用。然而，由于报文段可能失序到达，因此对推数据处理的规定必须小心对待。这里有两种极端状况。第一，一个push位为1的报文段可能比某些具有较小序号，并携带正常数据的报文段提早到达。由于TCP必须按顺序将数据交付给应用程序，所以它不可能马上将push位为1的报文段中的数据交付给应用程序。标准规定push请求不仅仅指出序列空间的某一点，而且也指明了缓冲区中的数据。因此，TCP必须记住某些数据是推数据，一旦中间延迟的数据到达目的站，马上交付所有数据。第二，一个push位置1的报文段可能比其他报文段晚到达目的站，而这些报文段中携带的数据的序号比推数据的序号要高。也就是说，当这个push位为1的报文段到达时，它将填补序列空间中的一块空穴。协议标准规定，当一个push位置为1的报文段到达时，TCP必须将所有可用的数据交付给应用程序。因此，push位为1的报文段到达后，需要TCP立即交付的数据可能会超出该报文段所携带的数据。

对推功能的定义所隐含的某种思想是非常重要的：TCP不注意数据传送的边界。程序员不能够依靠TCP来交付恰好到推数据结束点为止的数据，因为所交付的数据量取决于缓冲区的容量及报文段到达的时间。总结如下：

> 推数据处理不标定数据的边界。当一个发送进程调用推操作时，TCP会发送并交付其缓冲区内的所有数据。因此，如果一个push位设置为1的报文段S来迟了，接收方的TCP将会把S携带的数据及其他额外数据全部交付给应用程序。

然而，尽管推数据功能不能明确标定信息的边界，协议标准却规定一个接收进程必须保留一些有关推数据的状态信息。所遵循的规则其实是很简单：接收方进程或者立即传送全部的现有数据，或者记住有一个推数据已经到达，那么当接收方收到了序列空间中在推数据之前的所有数据后，立即向应用程序交付所有数据，包括那个push位为1的报文段中的所有数据。TCP必须记住有多少数据应该立即交付，然后在到达序列空间中的相应点时，停止推技术模式下的数据传递。

16.11　输入时推功能的实现

为了记录一个push位为1的报文段已经到达，我们的系统在TCB中使用了两个变量。字段tcb_pushseq记录了push位为1的报文段所携带的数据后面一个八位组的序号。字段tcb_code只有一个位，用以指示是否有推数据到达。代码字段中的位可以认为是一个模式（mode）位。当TCBF_PUSH未被置1时，TCP使用正常数据交付。当TCBF_PUSH被置1后，表示已经接收到推数据，应立即交付，直到缓冲区全空为止。

我们已经考察了完成推数据操作的过程。过程tcpdodat[1]在从一个输入报文段中提取数据时处理push位。此时要考虑两种情况：要么到达的报文段扩展了目前已确认的序列（例如，在含有push

① 参见12.13节的tcpdodat.c程序清单。

的字段数据前的序列空间中没有间隙），要么到达的报文段已经失序。在这两种情况下，tcpdodat都要将TCB代码字段中的TCBF_PUSH位置1，表示TCP应该在push模式下交付数据。如果新来的报文段扩展了可交付数据的序列空间长度，则这些数据应该立即交付给等待的应用程序，因此tcpdodat调用tcpwakeup来唤醒应用程序。如果序列空间含有间隙，则tcpdodat不能够立即将数据传送给应用程序。实际上，它将在稍后对push进行处理。要做到这一点，它计算出此报文段后面第一个八位组的序号（将报文段的序号与报文段中的数据长度相加），并将结果存放在TCB的字段tcb_pushseq中。然后它将push位置0，等待其他报文段填补序列空间中的间隙。

过程tfcoalesce[1]对延迟的推数据进行处理。tfcoalesce重构数据流，将到达的报文段按序号的大小顺序地插入到队列中。在插入报文段时，tfcoalesce检查所接收到的数据的序号是否已经达到被保存的推数据序号。如果是，则含有推数据的报文段中的数据以及它以前的所有数据都必须交付给应用程序，此时tfcoalesce重新设置push位并重置推数据序号字段。

一旦重新设置后，将一直保持推技术模式，直到接收应用程序取出了缓冲区里所有的数据，此时缓冲区的八位组计数器值为0。关闭推技术模式的代码可以在过程tcpgetdata中找到，该过程在本章的16.7节中讨论。

16.12 小结

TCP支持紧急报文机制，以允许通信中的应用程序发送带外指示。对紧急数据的解释有两种可选方案，其中用数据标记来解释紧急数据目前已被广泛采纳。数据标记解释法认为紧急数据指针指向正常数据流中的某个位置。

当应用程序从某个有紧急数据到达的连接中读取数据时，TCP将应用程序置于紧急模式。此后应用程序保持紧急模式，直到完全读取了数据流中的紧急数据以及它前面的数据。当应用程序读取到紧急数据的最后一个八位组时，TCP将应用程序置为正常模式。数据标记解释法把对正常数据进行缓冲的工作交给应用程序来完成，在应用程序寻找紧急数据时，它应该将所遇到的正常数据缓冲存放。

在传送紧急数据时，TCP将紧急数据八位组放在标准输出缓冲区中，并使用与正常数据相同的确认机制。然而TCP必须立即发送紧急数据，即使接收方已经关闭了它的接收窗口。

TCP推功能允许发送方指定外发的数据应该被无延迟地发送和交付。报文段的失序到达使推数据的处理复杂化。如果一个push位为1的报文段比其他携带较高序号数据的报文段晚到达，那么协议规定TCP必须将所有数据立即交付。而如果一个push位置1的报文段比其他携带低序号数据的报文段早到达，TCP将记录push位为1的报文段序号，当中间的那些数据到达时，TCP再交付所有数据。

深入研究

Postel［RFC 793］对紧急数据的处理和推数据操作做了规范。Braden［RFC 1122］提供了更为精细的规范。

---

① 参见t2.14节的tfcoalesce.c程序清单。

习题

16.1 假设发送方错误地发送了具有同样序号的紧急数据和正常数据，我们的样本代码将如何反应？应用程序收到的是什么？

16.2 假设两个携带紧急数据的报文段失序到达，例如第一个到达的报文段的序号为 1000，而第二个报文段的序号为900。如果一个应用程序在第一个报文段到达后、第二个报文段到达前，从连接中读取数据，它将读到什么？如果它等到两个报文段都到达之后再读取数据，又会读到什么？

16.3 如果发送方设置了推数据序列，同时以发送更多的报文段使接收方的缓冲区一直保持非空，那么将会发生什么情况？

16.4 假设当一个 push 位为 1 的报文段到达时，接收缓冲区中已经有一些数据，但是在序列空间中，已经接收的数据和新到的报文段之间有间隙，此时，协议标准是禁止、建议还是要求 TCP 以推技术模式交付现有数据？

16.5 在上题中，如果 tcpdodat 在给 tcb_pushseq 赋值时将 TCB 的 push 位置为 1，将会发生什么情况？

16.6 假设一个携带紧急数据的报文段丢失了。在我们的系统里是如何反应的？对于这种情况，协议标准是如何规定的？

第17章　套接层的接口

17.1　引言

每个操作系统都具体地定义了应用程序和协议软件之间的接口[1]。本章探讨了Xinu操作系统中的这一接口。虽然其中的过程以及过程的参数顺序是针对Xinu操作系统的，但接口的基本结构也存在于其他一些操作系统中。研究这些接口过程将有助于理解下层协议软件，同时也体现出它们与应用程序之间的交互作用。

17.2　通过设备形成的接口

与UNIX系统不同，Xinu不是将服务与设备合并入文件系统，而是将服务与文件合并到设备系统中。它为所有的输入、输出操作，包括应用程序和协议软件之间的通信，使用了“设备范例”（device paradigm）。要做到这一点，系统提供了一个“设备抽象”（device abstraction）机制，此外还定义了一套设备，其中绝大多数都有相应的外围I/O硬件设备。例如，系统利用设备抽象机制提供了一个CONSOLE（控制台），应用程序利用CONSOLE与控制台终端通信。除了与常规硬件设备直接对应的抽象设备之外，Xinu还定义了其他许多种设备，以允许应用程序访问系统服务体系。例如，Xinu提供了用于访问本地磁盘、远程磁盘上的各个文件以及各协议软件的设备。

不论应用程序所使用的抽象设备是对应于具体物理硬件的，还是对应于一个服务的，它都要遵循“打开－读－写－关闭”（open-read-write-close）的范例来使用该抽象设备。程序调用open时有三个入口参数：

```
d = open(device, name, other)
```

其中第一个参数是一个整型的设备标识符，用于指定被使用的设备。而第二个参数给出了一个与该设备相关的对象名称。第三个参数的含义取决于被打开的设备。对大多数设备来说，它指出被打开的对象应当可读，或是可写，或是两者皆可。open返回一个设备描述符，用于访问指定的对象。

一旦对象被打开，并生成了该对象的设备描述符之后，应用程序利用函数read和write完成与目标之间的数据传送。这两个函数都有三个入口参数：一个是该对象的设备描述符，其次是指向缓冲区的指针，最后一个是传输长度（以字节为单位）。绝大多数设备认为传输长度就是最大缓冲区大小。对read的调用指定了最大缓冲区长度，并返回读入的字节数。

```
len = read(device, buffer, buflen);
```

对write的调用返回一个状态代码。

```
status = write(device, buffer, buflen);
```

---

① 这里的接口一般是指API（application program interface）。

当应用程序完成使用对象的操作后，调用close来结束使用。因为Xinu使用了一个全局设备描述符空间，并允许进程通过共享存储器来交换设备描述符，所以它无法知道当前有多少个进程正在使用某个设备。因此，与许多系统不同，当应用程序结束后，Xinu不会自动关闭已打开的设备[①]。事实上，应用程序负责直接调用close来关闭已打开的设备。

17.2.1　单字节 I/O

由于一些程序认为一次只传送一个字节（字符）比较方便些，因而设备系统支持另外两个函数。函数getc[②]从设备中读取一个单字节，并将该字符作为函数值返回。getc最常用于终端设备中。

```
ch = getc(CONSOLE)
```

过程putc的入口参数是一个设备描述符和一个字符，它将该字符写入指定的设备中。

```
putc (CONSOLE, '\n'); /* move Console to next line */
```

17.2.2　其他一些非传送的函数

除了open和close操作以及传送数据的操作（read、write、getc和putc）之外，Xinu还支持两个用于控制设备的函数。程序使用的第一个函数是seek，它对物理设备（特别是磁盘）和抽象设备（特别是文件）执行定位操作。

```
seek(device, position);
```

其中参数device必须是一个整型的设备描述符，而参数position是一个长整数，给出设备定位处的偏移量。

第二个函数是更为通用的函数control，它处理其他所有非传送操作[③]。control的入口参数的数目是可变的。

```
control(device, FUNC, arg1, arg2, ... );
```

一个调用至少要有两个入口参数：一个用于指出被控制的设备的描述符，另一个是设备上使用的控制功能。一些control调用还需要另外一些参数。

控制功能包括任何与数据传送无关的操作，或涉及特殊处理的操作（例如，写TCP紧急数据）。举个例子说，control函数可被用来指定终端是否回显字符（通常在提示输入口令字或其他不宜显示出来的机密信息时，应用程序关闭终端的回显设施）。应用程序还可使用control函数来改变设备的操作模式。

程序员有必要参阅有关的资料来决定一组对于特定设备有效的control函数，并理解其真正含义。如果某个control函数需要更多的参数，在资料中会给出这些参数的类型及含义。

---

① Xinu确实支持UNIX的标准输入、标准输出和标准差错记法，当一个进程结束时，系统自动分别地对这三个描述符调用close函数。

② Xinu操作系统的getc和putc都是系统调用而不是库函数。

③ 许多Xinu操作系统的TCP control函数与BSD UNIX的系统调用功能等价。例如，BSD UNIX支持一个accept系统调用，而Xinu支持一个参数为TCPC_ACCEPT的控制函数，两者的效果是一样的。

17.3 作为设备的TCP连接

Xinu系统为TCP连接使用了两种类型的设备，即它提供了一个TCP“主设备”（master device，用于创建连接），以及一个TCP“从设备”（slave device，一旦连接已建立，通过从设备来进行通信）。不论是客户还是服务器都使用主设备来创建连接。

当一个客户程序希望建立TCP连接时，就调用TCP主设备中的open函数，并指定希望与之通信的远程目的站。此次调用为该连接分配一个TCP从设备，初始化与之相关的内部数据结构，并向调用者返回从设备的描述符。接着，调用者利用从设备描述符及read和write函数，通过连接传输数据。当客户完成数据传输之后，调用从设备中的close函数，以关闭连接，并释放从设备，使之能够重新使用。

17.4 TCP客户程序举例

仔细地阅读一个客户例子中的程序代码会是非常有用的。当调用时，过程finger实现了一个finger命令。finger功能允许某个机器上的用户找出在另一个机器上有哪些用户登录。要做到这一点，客户打开一个与远程服务器相连的TCP连接，发送一行文本，然后打印出所有从该服务器中返回的数据。被发送的这一行文本可能指定了一个被查找的用户登录名，或者只是一个空行，空行的含义是要求返回在该系统中登录的所有用户的信息。

```
/* finger.c-finger */

#include<conf.h>
#include<kernel.h>
#include<proc.h>
#include<network.h>

#define FINGERPORT      79          /* TCP port finger server uses   */

/*------------------------------------------------------------------------
 * finger - client procedure to print information about remote users
 *------------------------------------------------------------------------
 */
finger (rhost, user, outdev)
char   *rhost;              /* domain name of remote host to contact  */
char   *user;               /* name of specific user or null pointer  */
int    outdev;              /*device on which to print output         */
{
       IPaddr   addr        /* holds IP address of remote host        */
       int      dd;         /* device descriptor for connection       */
       int      cc;         /* count of characters read               */
       char     buf[2048];  /* buffer to hold finger information      */

       /* convert domain name to IP address and place in dotted       */
       /* decimal string "xxx.xxx.xxx.xxx:port"for call to open       */
```

```
name2ip(addr, rhost);
sprintf  (buf, "%u.%u.%u.%u:%d", BYTE(addr, 0), BYTE(addr, 1),
          BYTE(addr, 2), BYTE(addr, 3), FINGERRORT;
/* open connection, write one line (so it works with any server*/
/* and then repeatedly read and print data that arrives over   */
/* the connection.

dd=open(TCP, buf, ANYLPORT);
if (user)
       write(dd, user, strlen(uset));
write(dd, "\r\n", 2);
while ( (cc=read(dd, buf, sizeof(buf)))>0)
     write(outdev, buf, cc);
close(dd);
retun(OK);
}
```

在这段程序代码中，对 open 的调用带有三个入口参数：一个 TCP 主设备描述符（常量 TCP），一个给出了远程机器的 IP 地址和端口号的字符串，以及一个整数值，用以指出被使用的本地协议端口号（buf）。由于客户可以使用一个任意的本地端口号，因而它使用了常量 ANYLPORT。

一旦 finger 开通了一个连接，它通过该连接向服务器写入一行文本，然后不断地读出并打印由服务器返回的信息。在服务器做了应答，且有整整一缓冲区的数据可读之前，对 read 的调用一直保持阻塞状态。当然，如果所有数据都已到达，且服务器已关闭时，既便是缓冲区中的数据没有全满，调用也会返回。

17.5　TCP 服务器程序举例

服务器要比客户复杂得多，因为它必须在为一个现有连接服务的同时，能够将传入的连接请求以队列的形式保存。要做到这一点，服务器在主设备上调用 open，并指出它希望建立一个被动模式的 TCP 设备。服务器利用两次 control 调用来管理该被动设备。首先，服务器调用 control 时使用了功能代码 TCPC_LISTENQ 来设置传入请求队列的长度。然后，服务器进入一个循环，调用具有功能代码 TCPC_ACCEPT 的 control 函数，以接受下一个传入的连接。系统为每个新的连接分配一个从设备，并返回这些从设备的描述符。

考虑如下所示的例程 finger 服务器。finger 的服务提供了有关在本机中登录的用户的信息。服务器首先打开 TCP 设备，获取一个被动设备描述符，以便接受传入的连接请求。每当它接受了一个连接，该服务器从连接中读入一行输入，并通过发送有关在本机中登录的用户的信息进行响应。当它发送完毕之后，服务器关闭该连接。为了保持范例代码的简洁，在我们的服务器中仅返回一固定的信息，这些信息保存在程序一开始定义的字符串中。

```
/* fingerd.c - fingerd */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```

```
#define FINGERPORT      79              /* TCP port finger server uses */
#define BUFFERSIZ      120              /* size of input buffer        */
#define QUEUESIZE        5              /* size of conn. request queue */

str1 = "Login          Name       \n"; /* fake informatin to return   */
str2 = "dls       David L. Stevens\n";
str3 = "comer     Douglas E. Comer\n";
/*-----------------------------------------------------------------------
 *  fingerd  -  server to provide information about users logged in
 *-----------------------------------------------------------------------
 */
PROCESS fingerd()
{
        int     dd;                     /* descriptor for server       */
        int     dd2;                    /* descriptor for a connection */
        char    request[BUFFFRSIZ];     /* space to read request       */

        /* Open TCP in passive mode (no connection) for the server   */
        /* and set maximum queue size fcr incoming connections       */

        dd = open(TCP, ANYFPORT, FINGERPORT);
        control(dd, TCPC_LISTENQ, QUEUESIZE);

        /* Continually wait for next connection, read one line from it */
        /* and respond to the request                                  */

        while (TRUE) {
                dd2 = control (dd, TCPC_ACCEPT);

                /* Fake version: read and then ignore what client sends */

                if (read(dd2, request, sieof(request)) < 2) {
                        close(dd2);
                        continue;
                }
                write(dd2, str1, strlen(str1));
                wirte(dd2, str2, strlen(str2));
                wirte(dd2, str3, strlen(str3));
                close(dd2);
        }
}
```

在这段程序代码中，具有功能代码TCPC_ACCEPT的control调用返回一个设备描述符（dd2），它是为给定的连接服务的从设备描述符。然后服务器利用描述符dd2来“读”、“写”信息。同时，如果新的连接请求到达，TCP把它们与原设备联系起来并将它们放入队列中。当服务器结束使用某一连接时，它释放从设备描述符，使之可在新连接中再次使用。

17.6 TCP 主设备的实现

要实现一个设备，程序员必须提供对应于所有高层操作的过程：open、close、read、write、putc、getc、seek 和 control。对 TCP 主设备来说，其中只有两个操作是有意义的，对其余操作的调用只会导致返回一个差错代码。这两个操作分别是 open 和 control。open 用于构造一个连接，而 control 用于设置所有连接适用的默认参数。

17.6.1 TCP 主设备打开功能

tcpmopen 实现主设备上的 open 操作。

```
/* tcpmopen.c - tcpmopen */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <proc.h>

int tcpserver(struct tcb *, u_short);
int tcpbind(struct tcb *, char *, u_short);
int tcpcon(struct tcb *);

/*------------------------------------------------------------------------
 *  tcpmopen  -  open a fresh TCP pseudo device and return descriptor
 *------------------------------------------------------------------------
 */
int tcpmopen(pdev, fport0, lport0)
struct devsw *pdev;
void         *fport0;
void         lport0;
{
      char            *fport = (char*)fport0;
      int             lport = (int)lport0;
      struct  tcb     *ptcb;
      int             error;

      ptcb = (struct tcb *)tcballoc();
      if (ptcb == (struct tcb *)SYSERR)
           return SYSERR;
      ptcb->tcb_error = 0;
      proctab[currpid].ptcpumode = FALSE;
      if (fport == ANYFPORT)
           return tcpserver(ptcb, lport);

      if (tcpbind(ptcb, fport, lport) != OK ||
          tcpsync(ptcb) != OK) {
```

```
            ptcb->tcb_state = TCPS_FREE;
            sdelete(ptcb->tcb_mutex);
            return SYSERR;
        }
        if (error = tcpcon(ptcb))
            return error;
        return ptcb->tcb_dvnum;
}
```

为了打开一个新连接，tcpmopen调用tcballoc[1]来分配一个未用的TCB，并初始化主调进程的进程表项中的“紧急模式”（urgent mode）布尔变量。然后它检查外端口参数fport，以得知调用者要求的是一个被动打开还是主动打开。从finger的例程中可看出，服务器给出的fport的值为常量ANYFPORT，而客户则指定一个具体的对端目的站。如果外端口参数表明调用者是一个服务器，则tcpmopen将控制权移交给过程tcpserver。否则，它调用tcpbind来记录下该对端地址，并调用tcpsync[2]来初始化TCB中的字段，同时调用tcpcon来初始化一个主动连接。

17.6.2 被动TCP连接的形成

tcpmopen调用过程tcpserver来处理被动打开的细节。tcpserver填写预先分配的TCB，以便为接收连接请求并将它们放入队列中做好准备，然后它返回对应于该TCB的从设备描述符，使服务器能够使用该连接。

```
/* tcpserver.c - tcpserver */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ports.h>

/*------------------------------------------------------------------------
 *  tcpserver - do a TCP passive open
 *------------------------------------------------------------------------
 */
int tcpserver(ptcb, lport)
struct tcb   *ptcb;
u_short      lport;
{
        if (lport == ANYLPORT) {
            ptcb->tcb_state = TCPS_FREE;
            sdelete(ptcb->tcb_mutex);
            return SYSERR;
        }
        ptcb->tcb_type = TCPT_SERVER;
```

① 参见11.9.1节tcballoc.c的清单。
② 参见12.17节tcpsync.c的清单。

```
        ptcb->tcb_lport = lport;
        ptcb->tcb_state = TCPS_LISTEN;
        ptcb->tcb_lqsize = tcps_lqsize;
        ptcb->tcb_listenq = pcreate(ptcb->tcb_lqsize);
        ptcb->tcb_smss = 0;
        signal(ptcb->tcb_mutex);
        return ptcb->tcb_dvnum;
}
```

tcpserver首先检查本地端口的说明。尽管一个被动连接可以具有一个任意的、未指定的外端口，但它必须拥有一个特定的本地端口（即由该服务器操作的熟知端口）。如果调用者要求一个任意的本地端口，tcpserver释放互斥信号量并向调用者返回SYSERR。除此之外，tcpserver在已分配的TCB中为各字段赋值，为接受连接请求做好准备，并返回用于访问这个新分配的TCB的设备描述符。

17.6.3　主动TCP连接的形成

前面曾经提到，tcpmopen在需要形成一个主动连接时，调用三个过程：tcpbind、tcpsync和tcpcon。过程tcpbind在一个TCB中保存对端和本端的地址。

```
/* tcpbind.c - tcpbind */

#include <conf.h>
#include <kernel.h>
#include <network.h>

u_short tcpnxtp(void);

/*------------------------------------------------------------------------
 *  tcpbind - bind a TCP pseudo device to its addresses and port
 *------------------------------------------------------------------------
 */
int tcpbind(ptcb, fport, lport)
struct tcb   *ptcb;
char         *fport;
u_short      lport;
{
        struct  route   *prt;
        struct  tcb     *ptcb2;
        int             slot;

        if (dnparse(fport, &ptcb->tcb_rip, &ptcb->tcb_rport) == SYSERR)
                return SYSERR;
        prt = rtget(ptcb->tcb_rip, RTF_LOCAL);
        if (prt == 0)
                return SYSERR;
        if (prt->rt_ifnum == NI_LOCAL)
                ptcb->tcb_lip = ptcb->tcb_rip;
```

```
        else
             ptcb->tcb_lip = nif[prt->rt_ifnum].ni_ip;
        ptcb->tcb_pni = &nif[prt->rt_ifnum];
        rtfree(prt);
        if (lport == ANYLPORT) {
             ptcb->tcb_lport = tcpnxtp();    /* pick one */
             return OK;
        }
        ptcb->tcb_lport = lport;
        for (slot=0, ptcb2=&tcbtab[0]; slot<Ntcp; ++slot, ++ptcb2) {
             if (ptcb == ptcb2 ||
                 ptcb2->tcb_state == TCPS_FREE ||
                 ptcb->tcb_rport != ptcb2->tcb_rport ||
                 ptcb->tcb_lport != ptcb2->tcb_lport ||
                 ptcb->tcb_rip != ptcb2->tcb_rip ||
                 ptcb->tcb_lip != ptcb2->tcb_lip)
                  continue;
             return SYSERR;
        }
        return OK;
}
```

tcpbind首先调用过程dnparse来分析远程端点的说明，并将其分解为一个远程协议端口号（字段tcb_rport）和一个远程IP地址（字段tcb_rip）。tcpbind检查并确保有一条能够到达指定的远程机器的路由。它还要查看沿着该路由是否会进入伪网络接口（即，两个本地进程正在使用TCP进行通信的情况），如果进入伪网络端口，则它使用从接口中得到的IP地址。

在处理完远程机器端点后，tcpbind处理本地端口。如果调用者指定一个任意的本地端口，tcpbind调用函数tcpnxtp来分配一个未用的本地端口号。否则，它必须检查指定的端口，以确保没有其他连接正在使用该端口。要做到这一点，它在TCB中填写指定的端口，并遍历所有的TCB，将其中的连接端口号与为新分配的TCB指定的端口号相比较。只要远程的端口号不同，TCP允许多个连接使用相同的本地端口号。因此，tcpbind必须比较检查远程的IP地址和协议端口号。如果没有相同的，tcpbind返回OK；如果它发现了另一个连接两端的端口与新分配TCB的两端端口完全相同，tcpbind则返回差错码SYSERR。

17.6.4 分配一个未使用的本地端口

过程tcpnxtp分配一个未使用的本地端口并将其返回调用者。

```
/* tcpnxtp.c - tcpnxtp */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#define     IPPORT_RESERVED      1024  /* from BSD */
```

```
/*------------------------------------------------------------------------
 *  tcpnxtp  -  return the next available TCP local "port" number
 *------------------------------------------------------------------------
 */
u_short tcpnxtp()
{
        static    u_short lastport=1;  /* #'s 1-1023 */
        u_short           start;
        int               i;

        wait(tcps_tmutex);
        for (start=lastport++; start != lastport; ++lastport) {
             if (lastport == IPPORT_RESERVED)
                  lastport = 1;
             for (i=0; i<Ntcp; ++i)
                  if (tcbtab[i].tcb_state != TCPS_FREE &&
                            tcbtab[i].tcb_lport == lastport)
                       break;
             if (i == Ntcp)
                  break;
        }
        if (lastport == start)
             panic("out of TCP ports");
        signal(tcps_tmutex);
        return lastport;
}
```

tcpnxtp 利用静态变量 lastport 来保留一个整型的索引，指向历次调用中最近分配的端口号。因此当 tcpnxtp 开始时，lastport 的值与前一次调用时的值相同。

tcpnxtp 使用一个简单的算法。它遍历所有可能的本地端口号，直至找到一个未使用的端口号。对某次调用来说，变量 start 先记录下变量 lastport 的起始值，然后不断遍历，直至 tcpnxtp 已试遍了所有可能的值。

虽然 TCP 标准并未将 TCP 端口号限制在较小的值中，但我们例子中的代码遵从 BSD UNIX 系统使用的约定。这一类的系统保留 1 到 1024 号端口给特权程序。因此，在这样的范围内分配端口保证了客户也能高效率地通信，哪怕服务器要求它使用特殊端口。

17.6.5　主动连接的完成

一旦 tcpbind 将连接两端的端口说明保存在一个 TCB 中，并验证了没有其他连接具有同样的端口后，tcpmopen 调用 tcpsync 来初始化 TCB 中的大多数字段，然后调用过程 tcpcon 来形成一个连接。

```
/* tcpcon.c - tcpcon */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```

```
/*------------------------------------------------------------------------
 *  tcpcon  - initiate a connection
 *------------------------------------------------------------------------
 */
int
tcpcon(struct tcb *ptcb)
{
        struct netif  *pni =ptcb->tcb_pni;
        struct route  prt;
        Bool          local;
        int           error, mss;

        prt = (struct route *)rtget(ptcb->tcb_rip, RTF_REMOTE);
        local = prt && prt->rt_metric == 0;
        rtfree(prt);
        if (local)
                mss = ptcb->tcb_pni->ni_mtu-IPMHLEN-TCPMHLEN;
        else
                mss = 536;                      /* RFC 1122                */
        ptcb->tcb_smss = mss;                   /* default                 */
        ptcb->tcb_rmss = ptcb->tcb_smss;
        ptcb->tcb_swindow = ptcb->tcb_smss;     /* conservative            */
        ptcb->tcb_cwnd = ptcb->tcb_smss;        /* 1 segment               */
        ptcb->tcb_ssthresh = 65535;             /* IP Max window size      */
        ptcb->tcb_rnext = 0;
        ptcb->tcb_finseq = 0;
        ptcb->tcb_pushseq = 0;
        ptcb->tcb_flags = TCBF_NEEDOUT|TCBF_FIRSTSEND;
        ptcb->tcb_ostate = TCPO_IDLE;
        ptcb->tcb_state = TCPS_SYNSENT;
        tcpkick(ptcb);
        ptcb->tcb_listenq = SYSERR;
        TcpActiveOpens++;
        signal(ptcb->tcb_mutex);
        wait(ptcb->tcb_ocsem);
        if (error = ptcb->tcb_error)
                tcbdealloc(ptcb);
        return error;                           /* usually 0 */
}
```

tcpcon初始化最大报文段长度、序列空间计数器以及几个缓冲区指针。它调用tcpkick来启动连接，并返回调用者。

17.6.6 TCP 主设备的控制

过程 tcpmcntl 实现 TCP 主设备的 control 操作。

```
/* tcpmcntl.c - tcpmcntl */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpmcntl - control function for the master TCP pseudo-device
 *------------------------------------------------------------------------
 */
int
tcpmcntl(struct devsw *pdev, int func, void *arg, void *arg2)
{
        int  rv;

        if (pdev != &devtab[TCP])
                return SYSERR;

        switch (func) {
        case TCPC_LISTENQ:
                tcps_lqsize = (int)arg;
                rv = OK;
                break;
        default:
                rv = SYSERR;
        }
        return rv;
}
```

TCP 主设备的控制操作允许调用者设置参数或控制那些处理所有新创建的从设备的进程。tcpmcntl目前的实现方案仅提供了一种可能的控制功能——它允许调用者设置被动打开的连接在听状态下队列的默认长度。当该默认长度被设置后，所有的被动打开都将以新的队列长度工作。

17.7 TCP 从设备的实现

一旦主设备的 open 操作创建了一个从设备并为连接分配了一个新的 TCB 之后，应用程序就使用从设备来输入、输出数据。通常，应用程序唤醒从设备上的 read 和 write 操作。它还可以使用 getc 和 putc 来一次传输一个字节的数据，或者可以使用 control 分别控制每个设备。

17.7.1 由 TCP 从设备输入数据

过程 tcpread 实现 TCP 从设备上的 read 操作。

```
/* tcpread.c - tcpread */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

int tcpgetdata(struct tcb *, u_char *, unsigned);

/*------------------------------------------------------------------------
 *  tcpread - read one buffer from a TCP pseudo-device
 *------------------------------------------------------------------------
 */
int
tcpread(struct devsw *pdev, char *pch, unsigne len)
{
        struct  tcb     *ptcb = (struct tcb *)pdev->dvioblk;
        int             state = ptcb->tcb_state;
        int             cc;

        if (state != TCPS_ESTABLISHED && state != TCPS_CLOSEWAIT)
                return SYSERR;
retry:
        wait(ptcb->tcb_rsema);
        wait(ptcb->tcb_mutex);

        if (ptcb->tcb_state == TCPS_FREE)
                retrn SYSERR;                   /* gone         */
        if (ptcb->tcb_error) {
                tcpwakep(READERS, ptcb);        /* propagate it */
                signal(ptcb->tcb_mutex);
                return ptcb->tcb_error;
        }
        if (ptcb->tcb_flag & TCBF_RUPOK) {
                if (!proctab[currpid].ptcpumode) {
                        proctab[currpid].ptcpumode = TRUE;
                        cc = TCPE_URGENTMODE;
                } else
                        cc = tcpgetdata(ptcb, pch, len);
        } else {
                if (proctab[currpid].ptcpumode) {
                        proctab[currpid].ptcpumode = FALSE;
                        cc = TCPE_NORMALMODE;
                } else if (len > ptcb->tcb_rbcont &&
                        ptcb->tcb_flags & TCBF_BUFFER &&
                        (ptcb->tcb_flags & (TCBF_PUSH|TCBF_RDONE)) == 0) {
```

```
                        signal(ptcb->tcb_mutex);
                        goto retry;
                } else
                        cc = tcpgetdata(ptcb, pch, len);
        }
        tcpwakeup(READERS, ptcb);
        signal(ptcb->tcb_mutex);
        return cc;
}
```

如果TCB不在ESTABLISHED状态或CLOSE-WAIT状态，输入操作是被禁止的，因此tcpread返回SYSERR。在从输入缓冲区中提取数据之前，tcpread等待读操作信号量。当输入数据到达时，tcpwakeup释放读操作信号量，使应用程序能够继续执行。然后tcpread等待互斥信号量，以确保单独访问TCB。

一旦tcpread获得单独访问的权力，它先查看连接是否还存在（即TCB中的状态已变为TCPS_FREE），如果已不存在则返回SYSERR。接着，tcpread查看是否有差错发生，并调用tcpwakeup来唤醒下一个正在等待读操作的进程。然后，它向调用者返回差错代码。

如果上述检查一切正常，tcpread就已经准备好了从缓冲区中提取数据。tcpread将主调进程的紧急模式与连接的紧急状态相比较。它检查TCBF_RUPOK比特，以判断连接中是否含有紧急数据，并检查进程表中的字段ptcpumode，以判定主调进程当前是否在紧急模式下操作。如果这两个数值不相同，则tcpread改变进程的紧急模式状态，并返回一个代码，以通知调用者。

如果连接上有紧急数据到达，且主调进程已在紧急模式下做好准备，则tcpread调用tcpgetdata提取紧急数据。如果没有正在等待处理的紧急数据，tcpread查看缓冲器中的数据是否足够满足需要。通常，tcpread会保持阻塞状态，直至接收到足够的数据（即，直至缓冲区至少包含len个字节的数据）。然而，在三种特殊情况下，tcpread不会阻塞。第一种情况是：如果已到达数据的push位为1，tcpread将立即交付该数据。第二种情况是：如果发送方已经结束了传输，并关闭了连接，此时tcpread必须交付这些最后的数据，否则它将被永远阻塞（再不会有数据到达了）。第三种情况是：如果应用程序指定了不经过缓冲区的交付方式（即，TCB标志字段中的TCBF_BUFFER比特为0），则tcpread不经等待便直接交付到达的数据。因此，如果缓冲区中有足够满足要求的数据量，或以上三种特殊情况中的一种发生时，tcpread不再等待其他数据的到达而直接交付数据。它调用tcpgetdata来提取数据，并将其复制到应用程序的缓冲区中。

一旦tcpread完成了提取数据的任务，就调用tcpwakeup，使下一个正在等待的读进程来判断缓冲区中是否还有其他数据。然后它释放互斥信号量并返回。tcpread或者返回一个差错代码（它的值小于0），或者把提取出的字符数量作为函数值返回。因此调用了read操作的应用程序可精确地知道一共接收了多少字节的数据。

17.7.2　由TCP从设备输入单字节

函数tcpgetc实现TCP从设备上的getc操作。它只是调用tcpread来读一个单字符，并将结果返回调用者。

```
/* tcpgetc.c - tcpgetc */
```

```
#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpgetc  -  read one character from a TCP pseudo-device
 *------------------------------------------------------------------------
 */
int tcpgetc(pdev)
struct devsw *pdev;
{
        char ch;
        int  cc;

        cc = tcpread(pdev, &ch, 1);
        if (cc < 0)
             return SYSERR;
        else if (cc == 0)
             return EOF;
        /* else, valid data */
        return ch;
}
```

17.7.3 通过TCP从设备的输出

过程tcpwrite实现write操作，同时过程tcpputc实现单字符（字节）的输出。它们都要调用一个基本过程tcpwr。

```
/* tcpwrite.c - tcpwrite */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpwrite  -  write one buffer from a TCP pseudo-device
 *------------------------------------------------------------------------
 */
int
tcpwrite(struct  devsw *pdev, unsigned char *pch, unsigned len)
{
        return tcpwr(pdev, pch, len, TWF_NORMAL);
}

/* tcpputc.c - tcpputc */

#include <conf.h>
```

```
#include <kernel.h>
#include <proc.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpputc - write one character to a TCP pseudo-device
 *------------------------------------------------------------------------
 */
int
tcpputc(struct devsw *pdev, unsigned char ch)
{
        return tcpwr(pdev, &ch, 1, TWF_NORMAL);
}

/* tcpwr.c - tcpwr */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpwr  -  write urgent and normal data to TCP buffers
 *------------------------------------------------------------------------
 */
int
tcpwr(struct devsw     *pdev, u_char *pch, unsigned len, Bool isurg)
{
        struct tcb      *ptcb = (struct tcb *)pdev->dvioblk;
        int             state = ptcb->tcb_state;
        unsigned        sboff;
        int             tocopy;

        if (state != TCPS_ESTABLISHED && state != TCPS_CLOSEWAIT)
                return SYSERR;
        tocopy = tcpgetspace(ptcb, len);          /* acquires tcb_mutex */
        if (tocopy <= 0)
                return tocopy;
        sboff = (ptcb->tcb_sbstart+ptcb->tcb_sbcount) % ptcb->tcb_sbsize;
        if (isurg) {
                ptcb->tcb_supseq = ptcb->tcb_snext + len - 1;
                ++ptcb->tcb_flags |= TCBF_SUPOK;
        }
        while (tocopy--) {
                ptcb->tcb_sndbuf[sboff] = *pch++;
```

```
                ++ptcb->tcb_sbcount;
                if (++sboff >= ptcb->tcb_sbsize)
                        sboff = 0;
        }
        ptcb->tcb_flags |= TCBF_NEEDOUT;
        tcpwakeup(WRITERS, ptcb);
        signal(ptcb->tcb_mutex);

        if (isurg || ptcb->tcb_snext == ptcb->tcb_suna)
                tcpkick(ptcb);
        return len;
}
```

如果连接既不在ESTABLISHED状态也不在CLOSEWAIT状态，输出操作是不允许的，因此tcpwr会返回一个SYSERR。否则，tcpwr调用tcpgetspace在输出缓冲区中分配空间。如果tcpgetspace返回一个差错，则tcpwr向调用者返回该差错代码。如若不是，tcpwr为将要“写”的数据计算它们在输出缓冲区中的位置（输出缓冲区中已存在的字符数目加上这些字符的起始地址，当指针到达缓冲区末尾时，再绕回到头部）。然后tcpwr从计算得到的起始位置开始，将数据复制到缓冲区中。

最后，当tcpwr生成了将被发送出去的新输出数据之后，将TCBF_NEEDOUT比特置为1，以表明有新的数据正在等待输出，并调用tcpwakeup来唤醒其他“写”进程。如果调用者要求使用紧急传送或此时已没有正在进行中的输出，tcpwr调用tcpkick立即启动输出（发送方预防糊涂窗口）。

17.7.4 TCP连接的关闭

一旦某个应用程序结束了对一个TCP连接的使用，它就在从设备上调用tcpclose来关闭该连接。tcpclose还要释放该从设备。

```
/* tcpclose.c - tcpclose */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpclose - close a TCP connection
 *------------------------------------------------------------------------
 */
int tcpclose(pdev)
struct devsw*pdev;
{
 struct  tcb     *ptcb = (struct tcb *)pdev->dvioblk;
 int             error;

 wait(ptcb->tcb_mutex);
 switch (ptcb->tcb_state) {
      case TCPS_LISTEN:
```

```
        case TCPS_ESTABLISHED:
        case TCPS_CLOSEWAIT:
             break;
        case TCPS_FREE:
             return SYSERR;
        default:
             signal(ptcb->tcb_mutex);
             return SYSERR;
        }
        if (ptcb->tcb_error || ptcb->tcb_state == TCPS_LISTEN)
             return tcbdealloc(ptcb);
        /* to get here, we must be in ESTABLISHED or CLOSE_WAIT */

        TcpCurrEstab--;
        ptcb->tcb_flags |= TCBF_SNDFIN;
        ptcb->tcb_slast = ptcb->tcb_suna + ptcb->tcb_sbcount;
        if (ptcb->tcb_state == TCPS_ESTABLISHED)
             ptcb->tcb_state = TCPS_FINWAIT1;
        else /* CLOSE_WAIT */
             ptcb->tcb_state = TCPS_LASTACK;
        ptcb->tcb_flags |= TCBF_NEEDOUT;
        tcpkick(ptcb);
        signal(ptcb->tcb_mutex);
        wait(ptcb->tcb_ocsem);          /* wait for FIN to be ACKed     */
        error = ptcb->tcb_error;
        if (ptcb->tcb_state == TCPS_LASTACK)
             tcbdealloc(ptcb);
        return error;
}
```

只有在ESTABLISHED、CLOSE-WAIT或LISTEN状态下才能关闭连接。在其他状态下，tcpclose返回SYSERR。

在LISTEN状态下关闭一个连接，只需要释放该TCB（没有正在工作中的连接）。类似地，当应用程序在发生差错之后关闭一个连接时，tcpclose释放该TCB。

对于ESTABLISHED状态，关闭一个连接意味着进入FIN-WAIT-1状态，而在CLOSE-WAIT状态下，则要变迁到LAST-ACK状态。对这两种情况，tcpclose将TCBF_SNDFIN比特置为1，表示需要一个FIN，同时将TCBF_NEEDOUT比特置为1，使输出进程发送FIN，并调用tcpkick来启动该输出进程（它将发送FIN）。

最后，tcpclose在与该TCB相关的“打开-关闭”信号量上被阻塞，以等待向对方发送FIN而返回的确认。当该确认到达时，输入过程tcpfin1或输入过程tcplastack将会释放该“打开-关闭”信号量。当tcpclose恢复执行后，它释放该TCB并返回调用者。如果在发送出去的FIN被确认之前，连接异常中断，则tcpclose向应用程序返回相应的差错代码。

17.7.5 TCP从设备的控制操作

TCP允许应用程序通过为从设备提供控制(control)功能来控制各个连接上的参数。过程tcpcntl实现控制操作。

```
/* tcpcntl.c - tcpcntl */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ports.h>

int tcpuopt(struct tcb *, int, u_int);

/*------------------------------------------------------------------------
 *  tcpcntl - control function for TCP pseudo-devices
 *------------------------------------------------------------------------
 */
int
tcpcntl(struct devsw *pdev, int func, void *arg, void *arg2)
{
        struct  tcb   *ptcb = (struct tcb *)pdev->dvioblk;
        int           rv;

        if (ptcb == NULL || ptcb->tcb_state == TCPS_FREE)
                return SYSERR;

        wait(ptcb->tcb_mutex);
        if (ptcb->tcb_state == TCPS_FREE) /* verify no state change */
                return SYSERR;

        switch (func) {
        case TCPC_ACCEPT:       if (ptcb->tcb_type != TCPT_SERVER) {
                                        rv = SYSERR;
                                        break;
                                }
                                signal(ptcb->tcb_mutex);
                                return preceive(ptcb->tcb_listenq);
        case TCPC_LISTENQ:      rv = tcplq(ptcb, (int)arg);
                                break;
        case TCPC_STATUS:       rv = tcpstat(ptcb, (struct tcpstat *)arg);
                                break;
        case TCPC_SOPT:
        case TCPC_COPT:         rv = tcpuopt(ptcb, func, (u_int)arg);
                                break;
        case TCPC_SENDURG:      /*
```

```
                                        * tcpwr acquires and releases tcb_mutex
                                        * itself.
                                        */
                                       signal(ptcb->tcb_mutex);
                                       return tcpwr(pdev, (u_char *)arg,
                                              (unsigned)arg2, TWF_URGENT);
        default:
                rv = SYSERR;
        }
        signal(ptcb->tcb_mutex);
        return rv;
}
```

当tcpcntl开始时，它先验证该TCB是有效的，并获得单独访问TCB的权力。然后tcpcntl检查参数func，以得知调用者需要的是哪种控制操作。

17.7.6　接受来自被动设备的连接

17.5节中的例子展示了服务器是如何利用TCPC_ACCEPT功能接受来自LISTEN状态下的TCB中的各个连接的。TCPC_ACCEPT的实现方案是比较简单的。tcpcntl验证该TCB是为服务器的使用而打开的，并释放互斥信号量，接着调用preceive来为下一个传入的连接请求获取从设备描述符。tcpcntl必须释放互斥信号量，以允许TCP软件处理传入的SYN请求。然而在一个新的连接被建立之前，它不能释放信号量。在成功建立连接且从设备准备好之前，preceive一直处于阻塞状态。

17.7.7　改变LISTEN队列的长度

功能TCPC_LISTENQ允许调用者改变传入的连接请求的队列长度。回想一下，应用程序能够在主设备上使用同样的功能来改变一个默认的长度，所有服务器请求被动打开时都会收到这个默认长度。这里的区别在于tcpcntl仅为一个从设备改变队列长度。tcpcntl调用过程tcplq来实现这个改变。

```
/* tcplq.c - tcplq */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <mark.h>
#include <ports.h>

/*------------------------------------------------------------------------
 *  tcplq - set the listen queue size for a TCP pseudo device
 *------------------------------------------------------------------------
 */
int
tcplq(struct tcb *ptcb, unsigned lqsize)
{
```

```
        if (ptcb->tcb_state == TCPS_FREE)
                return SYSERR;
        ptcb->tcb_lqsize = lqsize;
        if (ptcb->tcb_type == TCPT_SERVER) {
                pdelete(ptcb->tcb_listenq, PTNODISP);
                ptcb->tcb_listenq = pcreate(ptcb->tcb_lqsize);
        }
        return OK;
}
```

为了改变连接队列的长度，tcplq以新的值更换保存在TCB中的队列长度，并删除现存队列，创建一个新队列。

17.7.8 获取从设备中的统计数据

网络管理软件经常需要提取并报告有关各个连接状态的情况。控制功能TCPC_STATUS提供了完成这些任务的机制。它调用过程tcpstat来收集有关一个连接的统计数据，并向调用者报告。

```
/* tcpstat.c - tcpstat */

#include <conf.h>
#include <kernel.h>
#include <network.h>

/*------------------------------------------------------------------------
 *  tcpstat - return status information for a TCP pseudo device
 *------------------------------------------------------------------------
 */
int tcpstat(ptcb, tcps)
struct tcb  *ptcb;
struct tcpstat   *tcps;
{
        tcps->ts_type = ptcb->tcb_type;
        switch (ptcb->tcb_type) {
        case TCPT_SERVER:
                /* should increase to entire TCP MIB */
                tcps->ts_connects = TcpActiveOpens;
                tcps->ts_aborts = TcpEstabResets;
                tcps->ts_retrans = TcpRetransSegs;
                break;
        case TCPT_CONNECTION:
                tcps->ts_laddr = ptcb->tcb_lip,;
                tcps->ts_lport = ptcb->tcb_lport;
                tcps->ts_faddr = ptcb->tcb_rip;
                tcps->ts_fport = ptcb->tcb_rport;
                tcps->ts_rwin = ptcb->tcb_rbsize - ptcb->tcb_rbcount;
```

```
            tcps->ts_swin = ptcb->tcb_swindow;
            tcps->ts_state = ptcb->tcb_state;
            tcps->ts_unacked = ptcb->tcb_suna;
            tcps->ts_prec = 0;
            break;
        case TCPT_MASTER:
            break;
        }
        return OK;
}
```

tcpstat 假设调用者已经向其传递了保存各种统计数据的 tcpstat 结构的地址。文件 tcpstat.h 中包含了对 tcpstat 结构的说明，同时还定义了各种用来访问各个字段的缩写标识符。

```
/* tcpstat.h */

/*      The union returned by the TCP STATUS control call          */
struct tcpstat {
        int  ts_type;          /* which kind of TCP status?        */
        union {
                struct {
                        long tsu_connects;   /* # connections    */
                        long tsu_aborts;     /* # aborts         */
                        long tsu_retrans;    /* # retransmissions */
                } T_unt;
                struct {
                        IPaddr  tsu_laddr;   /* local IP         */
                        short   tsu_lport;       /* local TCP port    */
                        IPaddr  tsu_faddr;   /* foreign IP       */
                        short   tsu_fport;   /* foreign TCP port */
                        short   tsu_rwin;    /* receive window   */
                        short   tsu_swin;    /* peer's window    */
                        short   tsu_state;   /* TCP state        */
                        long    tsu_unacked; /* bytes unacked    */
                        int     tsu_prec;    /* IP precedence    */
                } T_unc;
                struct {
                        long tsu_requests;   /* # connect requests*/
                        long tsu_qmax;       /* max queue length  */
                } T_uns;
        } T_un;
};

#define  ts_connects   T_un.T_unt.tsu_connects
#define  ts_aborts     T_un.T_unt.tsu_aborts
#define  ts_retrans    T_un.T_unt.tsu_retrans
```

```
#define  ts_laddr      T_un.T_unc.tsu_laddr
#define  ts_lport      T_un.T_unc.tsu_lport
#define  ts_faddr      T_un.T_unc.tsu_faddr
#define  ts_fport      T_un.T_unc.tsu_fport
#define  ts_rwin       T_un.T_unc.tsu_rwin
#define  ts_swin       T_un.T_unc.tsu_swin
#define  ts_state      T_un.T_unc.tsu_state
#define  ts_unacked    T_un.T_unc.tsu_unacked
#define  ts_prec       T_un.T_unc.tsu_prec

#define  ts_requests   T_un.T_uns.tsu_requests
#define  ts_qmax       T_un.T_uns.tsu_qmax

int tcpstat(struct tcb *, struct tcpstat *);
```

17.7.9 设置或清除TCP选项

控制操作还允许应用程序清除或设置tcb_flags字段中的选项比特。虽然这些比特中大多数是为了内部使用的，但至少其中有两个与应用程序有关。首先，TCBF_BUFFER比特决定了“读”的行为是同步的还是异步的。需要特别指出的是，如果调用者要求TCP“读”n个字节数据，那么在n个字节数据到达之前，同步的调用将被阻塞，而异步的调用只要有数据到达就会返回，哪怕其中包含的数据少于n个字节（当然，如果发送方关闭了连接或指定了推数据，则“读”总是马上返回而不等待）。其次，应用程序能将TCBF_DELACK比特置1，以导致TCP延迟确认的发送。虽然对于普通的应用[1]，我们一般不推荐使用延迟的确认，但某些连接使用它们来减少通信量（因为确认是附加在外发的数据报文段中的）。

为了清除或设置选项比特，当应用程序调用control时，以TCPC_COPT或TCPC_SOPT作为函数参数，另外还有一个比特的掩码作为第三个参数。当tcpcntl发现清除或是设置请求时，它将该请求传递给过程tcpuopt。

```
/* tcpuopt.c - tcpuopt, ISUOPT */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#define ISUOPT(flags)    (!((flags) & ~(TCBF_DELACK|TCBF_BUFFER)))

/*------------------------------------------------------------------------
 *  tcpuopt - set/clear TCP user option flags
 *------------------------------------------------------------------------
 */
int
tcpuopt(struct tcb *ptcb, u_int func, u_int flags)
```

① 遗憾的是，主机需求文档RFC 1122，确实建议了延迟确认模式，而大多数研究者都认为使用延迟确认将给TCP往返时间估算带来混乱，并且除了极个别情况外，该方案往往导致低劣的系统性能。

```
{
        if (!ISUOPT(flags))
                return SYSERR;
        if (func == TCPC_SOPT)
                ptcb->tcb_flags |= flags;
        else
                ptcb->tcb_flags &= ~flags;
        return OK;
}
```

tcpuopt 利用宏 ISUOPT 来检查用户是否已经指定了任何不是 TCBF_DELACK 或 TCBF_BUFFER 的比特。如果是这样，那么它将拒绝该请求。然后 tcpuopt 检查功能代码，以判断它应该设置还是清除指定的比特。只要 TCB 未被释放，这些比特将一直置 1。

17.8　从设备的初始化

当系统启动时，操作系统初始化每个设备，包括用于连接的 TCP 从设备。过程 tcpinit 处理一个从设备的初始化。

```
/* tcpinit.c - tcpinit */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <mark.h>

static MARKER tcpmark;

/*------------------------------------------------------------------------
 *  tcpinit  -  initialize TCP slave pseudo device marking it free
 *------------------------------------------------------------------------
 */
int tcpinit(pdev)
struct devsw *pdev;
{
        struct    tcb  *tcb;

        if (unmarked(tcpmark)) {
                mark(tcpmark);
                tcps_tmutex = screate(1);
                tcps_lqsize = 5;          /* default listen Q size        */
        }
        pdev->dvioblk = (char *) (tcb = &tcbtab[pdev->dvminor]);
        tcb->tcb_dvnum = pdev->dvnum;
        tcb->tcb_state = TCPS_FREE;
        return OK;
```

```
}

#ifdef Ntcp
struct tcb  tcbtab[Ntcp];          /* tcp device control blocks    */
#endif
```

初始化包括将对应的TCB的状态字段设置为常量TCPS_FREE。一旦TCB初始化完成，它就可以被过程tcballoc使用了。

17.9 小结

TCP并不给出协议软件和应用程序之间的精确细节。事实上，它允许操作系统选择一个接口。许多操作系统沿用BSD UNIX使用的“套接字”（socket）接口。

在我们的例子中，接口遵循“打开-读-写-关闭”的范例来使用设备抽象机制。应用程序在TCP主设备上调用open操作来分配属于各个连接的设备描述符。当一个客户“打开”该TCP主设备时，它接收到一个用于数据传输的连接描述符。当一个服务器“打开”该TCP主设备时，它接收到一个部分描述符，只能用于接收传入的连接。然后服务器重复不断地在该部分描述符上发出control调用，用以为每个连接获取一个描述符。

一旦连接建立，应用程序利用read、write、getc或putc来传送数据。最后，应用程序调用close来结束使用该连接，并使得新的连接能够再次使用该描述符。

本章详述了在TCP主设备和TCP从设备上提供I/O操作过程的具体实现。

深入研究

Comer［1996］提供了对BSD UNIX套接字的概述，而Leffler、Mckusick、Karel和Quarterman［1989］讨论了套接字的实现。Stevens［1990］详述了应用程序如何使用不同的UNIX协议接口。Comer［1984］提供了Xinu中设备结构的细节，并描述了诸如read和write之类的操作是如何映射到其底层函数，如tcpread和tcpwrite上的。

习题

17.1 TCP主设备使用的远程机器的定义也许看起来很繁琐。请列举几个原因，解释一下为什么会选择这样的说明。

17.2 作为另一种与本章给出的设计方案不同的设计思想，试想若TCP主设备对任一控制操作都会向当前所有活动的从设备传递的设计方案。以ABORT控制操作为例，当一个TCP从设备使用它时仅终止该从设备上的连接，而当TCP主设备使用它时，则所有从设备上的所有正在工作中的连接都被终止。这种策略有什么优点，又有什么缺点？

17.3 仔细阅读tcpcntl过程开头的有关互斥操作的程序代码。调用wait有可能导致差错吗？它会不会对输出有所影响？为什么？

17.4 如果在服务器上通过control为它的从设备改变监听队列长度之前，已有一个或多个连接请求已经到达，这时会发生什么情况？

17.5 列出更多的对应用程序来说可能有用的控制功能。

17.6 我们的程序代码利用control来实现所有非传输操作。这种方法与为每个特殊的操作使用不同的系统调用（如 accept 和 listen 等）相比较，各有什么优缺点？

17.7 大缓冲区通常意味着较高的吞吐量，但有时候它却无法做到这一点。假设多个应用程序不断地试图在一个 TCBF_BUFFER 比特为 1 的连接做“读”操作，试论证，不论它们哪一个使用最大的缓冲区都只会得到最少的服务。

17.8 如上一题所述，多个应用程序试图在一个连接中做“读”操作，在什么情况下其中的一个应用程序根本接收不到任何数据，而其他的应用程序却连续不断地接收数据？

第18章 RIP：主动路由传播和被动获取

18.1 引言

前面的章节描述了在IP向目的站传送数据报时，所使用的IP选路表结构及相应过程。通过简单的互联网拓扑结构相互连接的主机或网关，在系统启动阶段对它们的IP路由表进行初始化，在表内插入几个永远不会更改的表项。然而，在大多数的环境下，网关将动态地传播选路信息，以支持最小路径的自动计算及自动从短暂的网络或网关故障中恢复。接收到传播信息的网络或网关随即更改它们的选路表表项。

本章讨论"选路信息协议"（Routing Information Protocol），这是一个在网关和主机之间传播路由信息时最常用到的协议[1]。尽管RIP表面上看起来简单，但我们将会发现，其中存在着许多细微的规则，它们决定了传播哪些路由，以及何时传播这些路由。这些规则帮助我们避免了形成路由环路，并且使路由传播更快、更可靠。

路由传播是互联网中最复杂的任务之一。对标准的微小变动或缺乏严密的推断都可能导致严重的问题，例如非优化的路由和不稳定性。还有，由于协议软件的大多数错误只影响运行该软件的机器，而低劣的路由传播软件可以影响它所在的互联网上的所有机器，因此它的危害性就显得极其严重。与所有的选路协议相同，一个正确的RIP实现方案需要对细节仔细研究。

18.2 主动和被动模式的参与者

TCP/IP互联网遵循一个假设：由于网关能够与其他网关之间交换选路信息，因面它能够了解正确的路由。另一方面，主机只是从网关处获取路由信息，它们的信息可能是不完整的，或者不是权威性的。因此，不允许主机通知其他机器有关路由的信息。总之：

网关参与主动传播选路信息，而主机则被动地接受路由信息并且永远不会传播这些信息。

RIP协议为遵循这条规则提供了操作的两种基本模式。主机以被动模式使用RIP，它被动地听取网关发送的RIP报文，从中提取选路信息，并更改自己的选路表。被动RIP并不传播本地选路表中的信息。网关以主动模式使用RIP。它主动地参与听取其他网关传来RIP报文，将新路由添加到选路表中，并发送包含有更改后的选路表表项的报文。因此，主动参与者主要完成两个工作（传输和接收），而被动参与者只完成一项工作（接收）。下一节我们讨论主动参与者。

18.3 基本的RIP算法和费用度量

RIP使用矢量距离（vector-distance）算法[2]传播路由，并利用本地网络广播交付报文。每个网

① 从技术上讲，RIP即内部网关协议（Interior Gateway Protocol, IGP），因为在自动系统内部使用它来传递路由。

② 近来也有一些文献使用distance-vector（DV）这种说法。

关定期地将它现有的IP选路表中路由广播发送给其上的所有网络接口。与其他的矢量距离协议相似，RIP报文中包含了一些由目的网络和到该网络的距离组成序偶（pair）。

当一个RIP更新报文到达时，接收方检查选路表中的每一个表项，对每一个具有相同目的站D的表项比较当前路由。接收方使用三角形不等式来判断到目的站D的通告路由是否优先于现有路由。这就是说，当接收方检查一个来自网关G的表项时，它会提出这样一个问题：从本机到G的费用加上从G到D的费用，是否要比目前从本机到D的耗费高。以数学形式来表示，接收者R询问是否：

$$\text{cost}(R,\ G) + \text{cost}(G,\ D) < \text{cost}(R,\ D)$$

其中cost(i, j)表示从i到j的最少路由费用。如果经由网关G发送报文的费用比目前费用低，接收方只需更改它的选路表表项。在更改一个路由时，接收方给该路由分配的费用值等于：

$$\text{cost}(R,\ G) + \text{cost}(G,\ D)$$

由于到一个相邻网关的费用为1，则新的费用成为：

$$\text{cost}(R,\ D) = \text{cost}(G,\ D) + 1$$

尽管上面的描述看起来简单，但最后还有一个细节给其增加了复杂性。假设目前由R到D的路由都经过G。当从G传来一个新的更新信息时，R必须更改它对该路由设置的费用值，而不管G发来的信息将会增加还是减少费用。因此，算法的最终描述应该为：

> 当网关G发来一个费用为M，目的站为D的RIP更新信息时，本机R将其与现有路由相比较。如果在本机中没有目的站为D的路由存在，则生成一个路由，且下一跳为G，费用为M + 1。如果目前路由指出G就是下一跳，则该路由的费用更改为M + 1。否则，如果目前路由的费用高于M + 1，则设置费用为M + 1，并设置下一跳为G。

18.4　不稳定性及解决方案

18.4.1　计数到无穷大

大多数矢量距离算法都存在一个共同的问题：它们允许短时间的路由环路。当两个或更多个网关以环状顺序锁定时，一个以D为目的站的路由环路就会产生。此时每一个网关都认为到目的站D的最优路由是选择以环状顺序排列的下一个网关。最简单的路由环路只包含两个网关，每一个都认为另外一个网关是到达某目的站的最佳下一跳。

使用RIP的网关无法轻易地检测出路由环路。当一个到目的站D的路由环路确实产生后，RIP协议对该路由环路涉及到的网关每次将费用值加1，缓缓增加其费用值。它们将持续增加到度量值为无穷大，软件将这样大的值解释为："到该目的站没有路由存在"。为了减少路由环路造成的破坏，RIP定义的"无穷大"是一个比较小的数值。我们可以总结如下：

> 为了减少一个路由环路存在的时间，RIP定义无穷大的值为16。当某个路由费用达到16时，RIP将其解释为："没有路由存在"。

18.4.2 网关瘫痪和路由超时

RIP要求所有参与的网关和主机为所有的路由设置一个定时时限值。当超时事件发生时，该路由必须被删除。要理解设置超时的原因，让我们考虑一下RIP中的主动参与者网关G突然瘫痪会发生什么情况。相邻网关已经从网关G收到了更新报文，并装载了将G作为下一跳的路由。当G发生故障时，相邻网关无从得知将G作为下一跳是错误的。从本质上讲，此时路由的费用为无穷大，由于本该负责发送路由更新信息的网关已瘫痪，使得相邻网关无法得知这一信息。因此，从RIP接收信息的网关要担负起保证它仍然正常工作的责任。

当建立或更新路由时，使每个路由具有一个关联定时器，如果在定时器超时之前没有确认路由有效的信息到达，则宣布该路由无效。

18.4.3 水平划分

路由环路产生的一个最常见的原因是：网关将所有选路报文发往所有的网络接口。要理解这一问题，可考虑与同一个以太网相连的三个网关A、B和C。假设网关A有一个到目的站D的费用为1的路径，并已经为此广播了RIP路由更新分组。B和C都收到了更新信息，也都建立了到目的站D的费用为2的路由。如果B和C通告其路由，由于这些路由比A广播的路由费用要高，因此不会出现问题。

现在假设A瘫痪了。如果B和C继续长时间通告它们到达目的站D且费用为2的路由，网络上的机器最终会发现A通告的路由超时，因而接受费用为2的路由。事实上，一旦网关A通告的路由超时，B和C其中之一必然会选择对方所通告的路由，并形成一个暂时的路由环路。

要避免路由环路问题，RIP使用了“水平划分”（split horizon）技术。它的规则很简单：

当向某个网络接口发送RIP更新信息时，决不能包含从该接口处得到的选路信息。

要了解这一规则，可以从网关内部是如何选择路由的观点来分析。如果网关G从与网络N的接口中得到一条到目的站D的路由，那么G的路由必须指明在网络N中的下一跳。现在假设G在向网络N发送RIP更新信息时，包含了到目的站D的路由。如果网络N上的网关或主机中没有以D为目的站的已存在路由（可能是因为出现过一次故障），它将建立所通告的路由，并将所有目的站为D的数据报发送到网关G。如果一个以D为目的站的数据报到达G，G将该数据报转发到位于网络N的下一跳。因此，G将那些来自网络N的数据报又送还给网络N。水平划分法避免了可以引起这一问题的路由通告，解决了这一问题。

18.4.4 毒性逆转

总的来说，像RIP这样的矢量距离协议允许路由环路持续一段时间，因为它们并不迅速地传播路由丢失的报文。一种被称为“毒性逆转”（poison reverse，或者是具有毒性逆转的水平划分）的试探策略解决了这一问题。它对水平划分法做了修正。该方法并不像水平划分那样，避免向路由信息到达时所经过的网络接口传播路由更新信息，而是向来时经过的接口发送一个携带着否认信息的更新。

在通过某个网络接口发送RIP更新信息时，应包括所有路由，但对于从该接口上获取的那些路由报文，要将其度量设置为无穷大。

毒性逆转将迅速打断路由环路。如果两个机器分别具有一个指向对方的、目的站为D的路由，那么应当安排它们发送费用为无穷大的更新信息。只要其中一个机器发送出这样的更新信息，路由环路就会被打破。

当然，使用具有毒性逆转的水平划分法有一个缺点：它增加了更新报文的长度（因此将占用更多的网络带宽）。然而对大多数网关来说，增加更新报文的长度并不会带来问题。

18.4.5　具有毒性逆转的路由超时

我们说RIP要求网关给每一个路由设置一个定时器，并在定时器时限超过时使此路由成为无效的。最简单的实现方案是在定时器超时后，仅仅将对应路由从选路表中删除。然而，当RIP使用毒性逆转时，程序不能够在路由无效时删除该路由。相反，它还必须保存一个记录，说明该路由曾存在且现在的费用是无穷大。

RIP只需要保持超时路由一段时间，以保证外发的报文已经将此信息传播给了相邻网关。从原则上讲，RIP只需要在一个更新周期内保持超时的路由。当然由于底层的UDP和IP协议可能丢失数据报，RIP保存一个超时路由记录的时间将长达4个更新周期。在4个更新周期后RIP假设相邻网关已经收到了至少一个报告了路由费用为无穷大的报文，因此它删除该路由。

18.4.6　触发更新

另外有一项技巧可以帮助RIP在有较大的路由环路存在时，仍然能够保持较好的健壮性。该技术称为“触发更新”（triggered update），它使用快速的更新来加速变化后的收敛过程。

> 一旦某个网关改变了一条路由的度量，它必须立即向所有相邻网关发送更新报文，而不用等待通常循环的更新周期。

触发更新帮助RIP打破涉及两个网关以上的路由环路，因为它使RIP立即传播费用为无穷大的路由，而并不等待周期性的广播。要理解触发更新的具体工作过程，让我们考虑一个路由环路内的n个网关（n > 2）。第一个网关将其路由通告给第二个，而第二个网关又将路由通告第三个，以此类推，直到最后一个网关将其路由通告给第一个网关。在这种情况下，当更新信息在路由环路中走完一圈时，路由的度量将会增加n。因此，计数到无穷大可能会需要很长时间，即便对无穷大的定义是很小的。单靠水平划分法是无法打破涉及多个网关的路由环路，因为对每一对网关来说，路由通告仅仅向一个方向传播，且决不会反过来。

触发更新通过迅速传播路由以增加系统的健壮性。特别是，如果一个网关G失去了到目的站D的连接，它将发送一个触发更新，以传播到D的路由，该路由的费用被设为无穷大。任何依靠G到达目的站D的相邻网关都将接收到更新报文，并更改其到D的费用为无穷大。G的相邻网关的变动将触发下一轮更新报文，而这些更新报文是向这些网关的相邻网关们发出的。以此类推，触发更新将导致连锁性的更新。事实上，如果触发更新产生得足够快[1]，它们将能完全避免路由环路的产生。

18.4.7　随机化以防止广播风暴

协议标准规定RIP必须对触发更新的传输随机化。这就是说：

---

① “足够快”的含义是连锁的更新过程必须在正常更新事件发生之前完成

一旦某个网关更改了一条路由的度量，它必须随机地延迟一小段时间后，才能向所有的相邻网关发送更新报文，而并不用等待通常周期性的更新。

要理解随机延迟是如何协助系统工作的，不要忘了RIP是通过硬件广播来传送更新报文，并假设有多个网关共享一个以太网。我们将毒性逆转机制考虑在内。只要有一个网关发送了目的站为D的某条路由的更新报文，这个以太网上的所有其他网关都将建立该更新记录，同时触发更新（包括发送毒性逆转报文给以太网上消息起源的网关）。因此，所有网关将试图同时广播其触发更新报文。一场广播风暴就出现了。事实上，如果一个网点中的所有网关都是从同一个销售商处购得，这些网关将使用同样的硬件和运行同样的软件，使得它们同时生成触发响应。为了避免同时传送，RIP规定一个网关在发送触发更新时必须等待一段短暂的、随机的时延。

18.5 报文类型

RIP提供两种基本的报文类型。它允许一个客户发送request报文来询问某特定路由，并提供response报文用以应答request报文或是周期性地通告路由。协议标准还定义了其他一些报文类型，但它们都已经过时不用了。

总的来说，很少有客户来轮询更新。事实上，大多数的实现方案依赖网关来产生一个周期性的更新报文。周期性广播报文的术语是“无偿响应”（gratuitous response），因为这是在没有request报文使它产生的情况下，仍然使用response报文类型。

18.6 协议特性

RIP具有下列特性：

- 在应用层工作
- 对所有的传输使用UDP
- 限制报文大小
- 是无连接的
- 采用n中取k算法达到可靠性
- 规定固定的超时值
- 使用协议的熟知端口号
- 使用单一的报文格式

下面一段将逐个讨论以上的特性。

层次和传送方式。RIP在应用层上工作，并使用UDP来传送请求和响应。使用UDP的主要优点是它允许客户向远程网关发送请求（即网关不与客户连接在同一个网络上）。使用UDP的主要缺点是RIP必须保证自己的可靠性。

较短的报文长度。RIP限制单个的更新报文的数据长度不大于512八位组。如果一个网关需要传播多于一个报文可以容纳的内容时，它就发送多个报文，但并不对报文进行编号或是将这些报文标志为一组。因此，如果出现分组丢失，接收方就有可能只收到部分更新信息而不知道丢失了数据。

无连接通信（connectionless communication）。RIP不维持一个连接，同时也不提供确认。

n 中取 k 的可靠性。RIP 通过使用 n 中取 k 的方法保证可靠性。每一个接收方监视周期性的广播，如果每 6 个更新至少收到了 1 个，就认为选路信息是有效的，否则则认为路由信息无效。接收方从来不轮询或验证一个路由是否过时，而仅仅被动地等待新的更新。

固定的超时时限。RIP 对所有的超时时限使用固定值。它每 30 秒广播一次更新报文，在收到一个更新报文后等待 180 秒，如果这期间仍没有报文到达，则宣布该路由无效。同时它还会保留该费用为无穷大的路由 120 秒（对毒性逆转）。这些超时时限值独立于应用 RIP 的底层物理网络或是往返时延，因此对于具有特别高的时延或是特别高的分组丢失率的网络，使用 RIP 是不太实际的。

协议的熟知端口。大多数通信的 RIP 使用协议端口 520。客户通常发送请求给 520，而服务器通常通过端口 520 应答。此外，所有周期性的更新广播从 520 发出，客户可以从 520 端口广播一个请求，要求本地网中所有网关产生一个响应。可是有些客户也常使用 520 以外的端口，并且用此端口向一个特定网关发送请求报文，服务器则从端口 520 将响应发送到发起请求的端口。

单一的报文格式。RIP 对所有的通信使用一个单一的报文格式。在请求报文中包含了客户要求得到的选路信息的目的站地址，同时并没有使用度量。在响应中，目的站地址和度量包含的信息来自发送方选路表。

18.7　RIP 的具体实现

18.7.1　实现的两种形式

从广义上讲，一个设计者必须从以下两种实现形式中选择。在第一种实现形式下，RIP 本身的选路数据库与 IP 用来转发数据报的选路表是分开的。当一个新的路由报文到达时，RIP 更新其选路数据库并将这一变动装载到 IP 的选路表中。在第二种形式下，RIP 与 IP 共同使用一张选路表。由于不管路由是来自什么地方的，都被装载在 IP 路由表内，因此选路表的表项需要额外的、供 RIP 使用的字段。

使用一个独立的选路表的好处是有利于功能分割。RIP 可以完全独立于 IP 操作，而只在需要更新 IP 选路表时才与 IP 协作。将 RIP 数据与 IP 选路数据库分开，使设计者能够做到在修改 RIP 数据结构，或是更改实现 RIP 的程序时，不用对 IP 部分做相应的变动。

将 RIP 和 IP 信息分割开的主要缺点是：这种分割状况使得 RIP 难以了解现有的路由状况或建立新的路由。例如，如果网关的其他协议获得一条路由，或者管理者人为地建立了一条路由，RIP 就无法知道这一变化。因此，RIP 发送的路由更新报文可能并不准确反映现有的选路表状况。同样，RIP 可能在一条路由被删除后仍然通告该路由，直到管理者将所有的变动都通知 RIP。

尽管我们采纳了第二种方案，但涉及到定时器管理问题时，将 RIP 信息存储在 IP 选路表内的形式是有缺点的。由于单一的选路表存储了所有的选路信息，它也必须存储 RIP 在毒性逆转时要用到的信息。特别是，它必须保留那些已超时但用以生成毒性逆转更新的选路表项，即使这些路由在转发数据报时绝对不能使用。因此，IP 不能够使用选路表中那些费用为无穷大的表项。

18.7.2　定义

文件 rip.h 包含了 RIP 报文格式的定义及在整个代码段内使用的常量。

```
/* rip.h */

#if !defined(_RTP_H) && defined(RIP)
#define _RIP_H_

#define RIPHSIZE        4           /* size of the header in octets      */
#define RIP_VERSION     1           /* RIP version number                */
#define AF_INET         2           /* Address Family for IP             */

/* RIP commands codes: */

#define RIP_REQUEST     1           /* message requests routes           */
#define RIP_RESPONSE    2           /* message contains route advertisement*/

/* one RIP route entry */

struct riprt {
        short   rr_family;          /* 4BSD Address Family               */
        short   rr_mbz;             /* must be zero                      */
        u_long  rr_ipa;             /* the part we use for IP            */
        char    rr_pad[8];          /* remaining address for non-IP      */
        int     rr_metric;          /* distance (hop count) metric       */
};

#define MAXRIPROUTES    25          /* MAX routes per packet             */
#define RIP_INFINITY    16          /* dest is unreachable               */

#define RIPINT          300         /* interval to send (1/10 secs)      */
#define RIPDELTA        50          /* stray RIPINT +/- this (1/10 secs) */
#define RIPOUTMIN       50          /* min interval between updates (1/10s)*/

#define RIPRTTL         180         /* route ttl (secs)                  */
#define RIPZTIME        120         /* zombie route lifetime (secs)      */

/* RIP packet structure */
struct rip {
        char    rip_cmd;            /* RIP command                       */
        char    rip_vers;           /* RIP_VERSION, above                */
        short   rip_mbz;            /* must be zero                      */
        struct riprt    rip_rts[MAXRIPROUTES];
};

/* RIP process definitions */

extern PROCESS          rip(), ripout();/* processes to implement RIP  */
#define RIPISTK         49152           /* RIP process stack size      */
#define RIPOSTK         4096            /* RIP process stack size      */
#define RIPPRI          50              /* RIP process priority        */
#define RIPNAM          "rip"           /* RIP main process name       */
```

```
#define RIPONAM        "ripout"        /* RIP output process name      */
#define RIPARGC        0               /* Num. args for main RIP proc. */
#define RIPOARGC       0               /* Num. args for RIP out. proc. */

#define MAXNRIP        5               /* max # of packets per update  */

/* Per-interface RIP send data */
struct  rq {
        Bool            rq_active;      /* TRUE => this data is active  */
        IPaddr          rq_ip;          /* destination IP address       */
        unsigned short rq_port;         /* destination port number      */
/* what we've built already:            */
        struct    ep    *rq_pep[MAXNRIP];/* packets to send             */
        int             rq_len[MAXNRIP]; /* length of each              */
/* in-progress packet information       */
        int             rq_cur;         /* current working packet       */
        int             rq_nrts;        /* # routes this packet         */
        struct rip      *rq_prip;       /* current rip data             */
};

extern int  riplock;                    /* ripin/ripout synchronization */
extern int  rippid;                     /* PID for ripout() process     */
extern Bool dorip;                      /* TRUE => do active RIP        */

int ripmetric(struct route *, unsigned);
int ripstart(struct rq *);
int ripcheck(struct rip *, unsigned);
int ripprecv(struct rip *, unsigned, IPaddr);
int ripprepl(struct rip *, unsigned, IPaddr, unsigned short);
int ripsend(IPaddr, unsigned short);
Bool ripok(struct riprt *);
int ripifset(struct rq rqinfo[], IPaddr, unsigned short);
int ripadd(struct rq rqinfo[], struct route *);
#endif /* !defined(_RIP_H_ && defined(RIP) */
```

数据结构 rip 定义了 RIP 报文格式，而结构 riprt 定义报文内的单个路由。为了与 BSD UNIX 系统兼容，定义报文类型的常量被命名为 AF_INET。

18.7.3　输出的理论结构

由于实用技巧，如水平划分法和毒性逆转，要求根据不同的接口对更新报文的内容做相应变化，因此 RIP 不能够产生一个单一的更新报文并将其发送给所有接口。事实上，RIP 同时为每一个网络接口产生一个更新报文的单独副本。在向更新报文里增加路由时，它应用如毒性逆转这样的规则，来判断应该向不同的副本中添加些什么。

RIP 使用结构 rq 来保存每个更新报文副本中的内容[1]。由于并不是所有的网络都使用 RIP，因

① 输出过程将单个的 rq 结构组织成数组 rqinfo，该数组的每个数组元素对应一个网络接口。

而该结构有一个布尔字段rq_active，允许管理者决定RIP是否应该给某接口发送更新报文。该结构看起来很复杂，因为RIP可能需要多个数据报来携带更新报文。它包含一个指向分组的指针数组（字段rq_pep）、一个对应的分组长度数组（字段rq_len）、一个整数说明RIP目前已经往报文里添加了多少分组（字段rq_cur）、RIP往现有的分组内添加了多少条路由（字段rq_nrts），同时还包含一个指针，指向分组里下一个路由应该放置的地方（字段rq_prip）。

18.8 基本RIP进程

过程rip完成两部分工作：它初始化RIP数据结构并处理所有的输入RIP分组。

```
/* ripin.c - rip */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <proc.h>

#ifdef RIP
/*------------------------------------------------------------------------
 *  rip  -  do the RIP route exchange protocol
 *------------------------------------------------------------------------
 */
PROCESS rip(void)
{
        struct  xgram    ripbuf;
        struct  rip     *prip;
        int             fd, len;

        fd = open(UDP, ANYFPORT, (void *)UP_RIP);
        if (fd == SYSERR)
                panic("rip: cannot open rip port");
        riplock = screate(1);
        if (gateway)
                resume(create(ripout, RIPSTK, RIPPRI, RIPONAM, RIPOARGC));

        while (TRUE) {
                len = read(fd, &ripbuf, sizeof(ripbuf));
                if (len == SYSERR)
                        continue;
                prip = (struct rip *)ripbuf.xg_data;
                if (ripcheck(prip, len) != OK)
                        continue;
                switch (prip->rip_cmd) {
                case RIP_RESPONSE:
                        if (ripbuf.xg_fport == UP_RIP)
```

```
                        riprecv(prip, len, ripbuf.xg_fip);
                    break;
                case RIP_REQUEST:
                    if (gateway || ripbuf.xg_fport != UP_RIP)
                        riprepl(prip, len, ripbuf.xg_fip,
                            ripbuf.xg_fport);
                    break;
                defaulit:
                    break;
                }
            }
    }

    #endif  /* RIP */

    Bool    dorip = FALSE;
    int     rippid = BADPID;
    int     riplock;
```

在初始化 RIP 时，过程 rip 打开 UDP 端口 UP_RIP，并产生一个锁定信号量。随后它检查该代码是在网关还是在主机上运行的。如果是在网关上，则紧接着启动 RIP 输出进程。

完成初始化工作后，rip进入一个无限循环。在循环中，它将下一个到达的报文读取到结构ripbuf里进行处理。根据标准规定，RIP 必须验证所有标记为“必须为零”的字段中包含的都是零。为了完成这项工作，rip 调用函数 ripcheck，并丢弃不正确的报文。

在验证完报文后，rip 使用报文类型来判断报文是一个请求（RIP_REQUEST）还是一个响应（RIP_RESPONSE），并相应地调用 riprepl 或 riprecv 来处理它。rip 遵守协议标准，忽略任何没有经过熟知端口（520）的响应报文。标准规定网关必须应答所有的 RIP 请求，不管它们来自于哪个端口，但是运行被动 RIP 的主机必须忽略任何来自熟知端口的请求报文。

18.8.1 “必须为零”的字段内容必须为零

过程 ripcheck 验证报文内协议规定为零的字段是否确实为零。

```
/* ripcheck.c - ripcheck */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef RIP

/*------------------------------------------------------------------------
 *  ripcheck  -  check a RIP packet for proper format
 *------------------------------------------------------------------------
 */
int
```

```
ripcheck(struct rip *prip, unsigned len)
{
        unsigned  i, j, nrts;

        switch (prip->rip_vers) {
        case 0:                 /* never OK          */
                return SYSERR;
        case 1:                 /* more checks below */
                break;
        default:                /* >1 always ok      */
                return OK;
        }
        /* check all "must be zero" fields */

        if (prip->rip_mbz)
                return SYSERR;
        nrts = (len - RIPHSIZE)/sizeof(struct riprt);
        for (i=0; i<nrts; ++i) {
                struct    riprt *prr = &prip->rip_rts[i];

                if (prr->rr_mbz)
                        return SYSERR;
                for (j=0; j<sizeof(prr->rr_pad); ++j)
                        if (prr->rr_pad[j])
                                return SYSERR;
        }
        return OK; /* this one's ok in my book... */
}
#endif  /* RIP */
```

根据标准规定，ripcheck拒绝接受协议版本号为零的报文。它接受版本号大于零的报文，但只检查版本号等于1的报文中为零的字段。当它确实要检查为零字段时，ripcheck检查报文首部中相应的字段。然后对报文内部的所有选路表项进行遍历，检查并确保IP用不到的地址八位组为零。

18.8.2 处理一个传入响应

过程riprecv处理传入的响应报文。入口参数包含指向RIP报文的一个指针、报文长度和发送方的IP地址。

```
/* riprecv.c - riprecv */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef RIP
```

```
/*------------------------------------------------------------------------
 *  riprecv  -  process a received RIP advertisement
 *------------------------------------------------------------------------
 */
int
riprecv(struct rip *prip, unsigned len, IPaddr gw)
{
        struct  route   *prt;
        IPaddr          mask;
        int             nrts, rn, ifnum;

        nrts = (len - RIPHSIZE)/sizeof(struct riprt);
        prt = rtget(gw, RTF_REMOTE);    /* find the interface number    */
        if (prt == NULL)
                return SYSERR;
        ifnum = prt->rt_ifnum;
        rtfree(prt);
        wait(riplock);          /* prevent updates until we're done  */
        for (rn=0; rn<nrts; ++rn) {
                struct riprt *rp = &prip->rip_rts[rn];

                rp->rr_family = net2hs(rp->rr_family);
                rp->rr_metric = net2hl(rp->rr_metric);
                if (!ripok(rp))
                        continue;
                mask = netmask(rp->rr_ipa)
                rtadd(rp->rr_ipa, mask, gw, rp->rr_metric,
                        ifnum, RIPRTTL);
        }
        signal(riplock);
        return OK;
}
#endif  /* RIP */
```

在计算完报文中有多少表项之后，riprecv 调用 rtget 寻找本机到发送网关的路由。如果机器中没有到该网关的路由，它不能够使用更新报文，因而返回一个 SYSERR。从返回的路由中，riprecv 提取要到达指定网关必须经过的网络接口号。并将其保存在变量 ifnum 中，在建立路由时要使用该值。

riprecv 遍历更新报文内的各个表项。对每一个表项，它调用 ripok 检查是否有畸形成分或无效地址，调用 netmask 来计算网络掩码，并调用 rtadd 来增加或更新本地 IP 选路表内的路由。注意 rtadd[1]应用了 RIP 更新规则：如果到某个目的站没有现存路由，则生成一个新路由，如果已经有一个经过发送网关到某个目的站的路由存在，则它更改其费用值。如果它已经拥有了一个费用更低的路由，则忽略新的路由。最后，rtadd 更新选路表。

① 参见 6.7.1 节 rtadd.c 的程序清单。

18.8.3 在更新期间的锁定

我们将要看到，RIP输出进程在形成和发送一个更新报文时要周期性地扫描选路表。为了保证输出软件在一个输入分组里的所有表项都处理完之前不发送触发更新，riprecv在做任何改变之前都要等待信号量 riplock。在所有的变动工作都完成后，它释放该信号量。

18.8.4 验证一个地址

前面说过riprecv调用过程ripok来验证一个地址的格式。ripok要验证以下几个方面：地址表项指定了一个IP地址类型；被通告的费用不会超过无穷大；被通告的地址的类别不是D或E；地址的网络部分不会有为零的八位组出现；主机部分的八位组也不为零；同时地址不能指定本地环回网络（127），其他的地址都表示合法的目的站，并将在选路中使用。

```
/* ripok.c - ripok */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef RIP
/*------------------------------------------------------------------------
 *  ripok  -  determine if a received RIP route is ok to install
 *------------------------------------------------------------------------
 */
Bool ripok(rp)
struct riprt  *rp;
{
        u_char net;

        if (rp->rr_family != AF_INET)
                return FALSE;
        if (rp->rr_metrio > RIP_INFINITY)
                return FALSE;
        if (IP_CLASSD(rp->rr_ipa) || IP_CLASSE(rp->rr_ipa))
                return FALSE;
        net = (net2hl (rp->rr_ipa) & 0xff000000>>24;
        if (net == 0 && rp->rr_ipa != ip_anyaddr)
                return FALSE;           /* net 0, host non-0       */
        if (net == 127)
                return FALSE;           /* loopback net            */
        return TRUE;
}
#endif  /* RIP */
```

18.9　对输入请求的响应

当一个请求报文到达时，rip 调用过程 riprepl 生成一个响应。

```
/* riprepl.c - riprepl */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef RIP
/*------------------------------------------------------------------------
 *  riprepl  -  process a received RIP request
 *------------------------------------------------------------------------
 */
int riprepl(pripin, len, gw, port)
struct rip        *pripin;
unsigned          len;
IPaddr            gw;
unsigned short    port;
{
        struct  ep      *pep;
        struct  rip     *prip;
        struct  route   *prt;
        int             rn, nrts;

        nrts = (len - RIPHSIZE)/sizeof(struct riprt);
        if (nrts == 1 && pripin->rip_rts[0].rr_family == 0 &&
           net2hl(pripin->rip_rts[0].rr_metric) == RIP_INFINITY)
                return ripsend(gw, port);     /* send the full table     */
        pep = (struct ep *)getbuf(Net.netpool);
        /* get to the RIP data... */
        prip = (struct rip *)((struct udp *)
                ((struct ip *)pep->ep_data)->ip_data)->u_data;
        memcpy(prip, pripin, len);
        for (rn = 0; rn < nrts; ++rn) {
                struct riprt *rp = &prip->rip_rts[rn];

                if (net2hs(rp->rr_family) != AF_INET)
                        continue;
                prt = rtget(rp->rr_ipa, RTF_LOCAL);
                if (prt) {
                        rp->rr_metric = hl2net(prt->rt_metric);
                        rtfree(prt);
                } else
                        rp->rr_metric = hl2net(RIP_INFINITY);
```

```
        }
        prip->rip_cmd = RIP_RESPONSE;
        prip->rip_vers = RIP_VERSION;
        prip->rip_mbz = 0;
        udpsend(gw, port, UP_RIP, pep, len, 1);
        return OK;
}
#endif  /* RIP */
```

RIP允许请求报文中存在一种特殊情况。根据标准规定，如果请求报文中恰有一个表项，其地址族标识符为0，且费用为无穷大，则接收方应该产生一个完整的更新报文作为响应。否则，接收方的响应报文里应该对请求报文中指定的每一个地址提供接收方的本地路由。

riprepl调用过程ripsend来产生一个完整的更新报文以响应特殊请求事件。过程ripsend也被输出进程使用。对正常的请求来说，riprepl申请一块缓冲区以存放响应报文。在它将传入的报文复制到新的缓冲区后，riprepl遍历其中的每个表项。它使用rtget查找表项对应的本地路由，并将该路由的费用复制到响应报文内，或者在没有路由存在的情况下赋值RIP_INFINITY。在发送响应之前，riprepl为报文首部的命令字段（使其成为响应报文）和版本字段（以指示一个合适的版本号）赋值。随后它调用udpsend来传送该数据报。

18.10 生成更新报文

过程ripsend生成和发送一个更新报文。

```
/* ripsend.c - ripsend */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef RIP
/*------------------------------------------------------------------------
 * ripsend - send a RIP update
 *------------------------------------------------------------------------
 */
int ripsend(gw, port)
IPaddr      gw;
unsigned short   port;
{
        struct  rq      *prq, rqinfo[NIF];   /* here, so reentrant      */
        struct  route   *prt;
        int             i, pn;

        if (ripifset(rqinfo, gw, port) != OK)
                return SYSERR;

        wait(Route.ri_mutex);
```

```
for (i=0; i<RT_TSIZE; ++i)
        for (prt=rttable[i]; prt; prt=prt->rt_next)
                ripadd(rqinfo, prt);
if (Route.ri_defauIt)
        ripadd(rqinfo, Route.ri_default);
signal(Route.ri_mutex);

for (i=0; i<Net.nif; ++i)
        if (rqinfo[i].rq_active) {
                prq = &rqinfo[i];
                for (pn=0; pn<=prq->rq_cur; ++pn)
                        udpsend(prq->rq_ip, prq->rq_port, UP_RIP,
                            prq->rq_pep[pn], prq->rq_len[pn], 1);
        }
return OK;
}
#endif  /* RIP */
```

ripsend首先调用ripifset初始化数组rqinfo，rqinfo数组包含了每一个网络接口对应的更新报文。随后它等待信号量Route.ri_mutex以获得选路表的独立使用权。并对所有可能的路由进行循环处理。对每一条路由，ripsend调用过程ripadd。ripadd实现诸如水平划分法这样的试探策略，它对网络接口循环操作，判断是否应该向各接口对应的更新报文副本里添加路由。最后，在添加完所有路由后，ripsend遍历网络接口并调用udpsend来发送对应的更新报文副本。由于对应于本地主机的数组元素的rq_active字段值为FALSE，因而不给本地主机发送更新报文。

18.11　初始化一个更新报文的副本

过程ripifset初始化数组rqinfo。过程的第二个参数或者包含发送特殊请求报文的客户的IP地址，或者是全1地址，即要求对全网广播。

```
/* ripifset.c - ripifset */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef RIP
/*------------------------------------------------------------------------
 * ripifset - set the per-interface data for a RIP update
 *------------------------------------------------------------------------
 */
int ripifset(rqinfo, gw, port)
struct rq   rqinfo[];
IPaddr      gw;
unsigned short   port;
```

```
{
        struct  route   *prt;
        int             ifn;

        if (gw != ip_maskall) {
                for (ifn=0; ifn<Net.nif; ++ifn)
                        rqinfo[ifn].rq_active = FALSE;
                prt = rtget(gw, RTF_LOCAL);
                if (prt == 0)
                        return SYSERR;
                ifn = prt->rt_ifnum;
                rtfree(prt);
                if (ifn == NI_LOCAL)
                        return SYSERR;
                rqinfo[ifn].rq_ip, gw;
                rqinfo[ifn].rq_port = port;
                rqinfo[ifn].rq_active = TRUE;
                rqinfo[ifn].rq_cur = -1;
                rqinfo[ifn].rq_nrts = MAXRIPROUTES;
                return OK;
        }
        /* else, all interfaces */
        for (ifn=0; ifn<Net.nif; ++ifn) {
                rqinfo[ifn].rq_ip, nif[ifn].ni_brc;
                rqinfo[ifn].rq_port = port;
                rqinfo[ifn].rq_active = TRUE;
                rqinfo[ifn].rq_cur = -1;
                rqinfo[ifn].rq_nrts = MAXRIPROUTES;
        }
        rqinfo[NI_LOCAL].rq_active = FALSE;  /* never do this one */
        return OK;
}
#endif  /* RIP */
```

如果请求报文指明了一个具体的地址，ripifset就取消其他所有接口上的RIP处理工作，只保留一个通往该地址的接口。否则，它指示完成除本地主机的伪网络接口以外的接口上的RIP处理工作。当打开一个端口时，ripifset将当前数据报里的路由索引初始化为-1，表示现在没有数据报存在。

18.11.1 向更新报文的副本中添加路由

过程ripadd向一个更新报文的各副本中添加路由。

```
/* ripadd.c - ripadd */

#include <conf.h>
#include <kernel.h>
```

```
#include <network.h>

#ifdef RIP
/*-----------------------------------------------------------------------
 *  ripadd  -  add the given route to the RIP packets yet to send
 *-----------------------------------------------------------------------
 */
int ripadd(rqinfo, prt)
struct rq    rqinfo[];
struct route *prt;
{
        IPaddr     net;
        int  i, metric, pn, rn;

        for (i=0; i<Net.nif; ++i) {
                struct rq     *prq = &rqinfo[i];
                struct riprt *rp;

                if (!rqinfo[i].rq_active || nif[i].ni_state != NIS_UP)
                        continue;
                metric = ripmetric(prt, i);
                if (metric == SYSERR)
                        continue;
                if (prq->rq_nrts >= MAXRIPROUTES &&
                        ripstart(prq) != OK)
                            continue;
                pn = prq->rq_cur;
                rn = prq->rq_nrts++;
                rp = &prq->rq_prip->rip_rts[rn];
                rp->rr_family = hs2net(AF_INET);
                rp->rr_mbz = 0;
                net = netnum(prt->rt_net);
                memset(rp->rr_pad, 0, sizeof(rp->rr_pad));
                if (nif[i].ni_net == net ||
                        prt->rt_mask == ip_maskall)
                            rp->rr_ipa = prt->rt_net;
                else    /* send the net part only (esp. for subnets)    */
                            rp->rr_ipa = net;
                rp->rr_metric = hl2net(metric);
                prq->rq_len[pn] += sizeof(struct riprt);
        }
        return OK;
}
#endif  /* RIP */
```

给出一个路由作为参数，ripadd对一组网络接口进行循环处理，检查该路由是否应该添加到某

个网络接口的更新报文中。如果对应于该网络接口的RIP表项是活动的，且接口正常工作，则ripadd向其添加路由。它调用ripmetric来计算费用。它在正在构建的报文内申请空间，如果现有的数据报全满的话，它调用ripstart来申请另一个数据报。它为该路由填写路由族“必须为0”字段。

ripadd找出路由在传送时所要经过的网络接口的IP地址，并将该地址与路由的目的地址比较，以决定采用地址的确定格式。通常，RIP屏蔽掉子网号和目的地址的主机部分，而只传播网络部分。然而，它在一个子网范围内传播子网信息。因此，如果一个负责该路由发送的端口位于目的地址的子网内，则它同时传播地址的子网部分及网络部分。一旦ripadd计算出正确的等待广播的IP地址和要使用的费用值，就填写报文内的下一个表项，继续对所有路由的遍历。

18.11.2 计算一个待通告的费用值

过程ripmetric用于计算将要与给定路由一起通告的费用值。

```
/* ripmetric.c - ripmetric */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef RIP
/*------------------------------------------------------------------------
 * ripmetric - compute the RIP metric for a route we advertise
 *------------------------------------------------------------------------
 */
int ripmetric(prt, ifnum)
struct route     *prt;
unsigned         ifnum;
{
        /* only advertise the net route for our interfaces */

        if (prt->rt_ifnum == NI_LOCAL &&
                        prt->rt_mask == ip_maskall)
                return SYSERR;

        if (prt->rt_ifnum == ifnum)
                return RIP_INFINITY;   /* poison reverse */
        /* else, add one to our cost */
        return prt->rt_metric + 1;
}
#endif  /* RIP */
```

ripmetric实现了毒性逆转功能，如果数据报沿着路由又回到获取该路由报文的网络接口，则ripmetric将路由的费用变为无穷大。在其他情况下，它将本地费用转换为标准的RIP费用，其中直接连接的本地费用值为0，而标准RIP费用值为1。

18.11.3　为 RIP 报文分配数据报

我们已经看到，ripadd将选路通告收集到一个更新报文里，并给每个网络接口生成一个副本。当它发现已经完全填满了一个 RIP 报文时，ripadd 调用 ripstart 为该 RIP 报文分配一个新的数据报。

```
/* ripstart.c - ripstart */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef RIP
/*------------------------------------------------------------------------
 * ripstart - initialize an interface's RIP packet data
 *------------------------------------------------------------------------
 */
int ripstart(prq)
struct rq   *prq;
{
        struct  ep      *pep;
        struct  ip      *pip;
        struct  udp     *pudp;
        struct  rip     *prip;
        int             pn;

        pn = ++prq->rq_cur;
        if (pn >= MAXNRIP)
                return SYSERR;
        prq->rq_nrts = 0;
        prq->rq_pep[pn] = pep = (struct ep *)getbuf(Net.netpool);
        if ((int)pep == SYSERR)
                return SYSERR;
        pip = (struct ip *)pep->ep_data;
        pudp = (struct udp *)pip->ip_data;
        prip = (struct rip *)pudp->u_data;

        prq->rq_prip = prip;
        prq->rq_len[pn] = RIPHSIZE;
        prip->rip_cmd = RIP_RESPONSE;
        prip->rip_vers = RIP_VERSION;
        prip->rip_mbz = 0;
        return OK;
}
#endif  /* RIP */
```

ripstart 只有一个入口参数，该参数包含一个指向 rqinfo 数组元素的指针。它将该数组元素中的现有数据报的索引值加 1，为另一个数据报申请一块缓冲区，在新分配的缓冲区内填写 RIP 首部的

各个字段值，并将新数据报中“必须为零”的字段设置为零。

18.12 生成定期的RIP输出

网关定期发送RIP响应。要做到这一点，它们生成一个执行过程ripout的进程。输出进程进入一个无限循环，首先延迟30秒钟，随后调用ripsend在所有的接口上生成一个更新报文。所编写的代码允许触发更新使用报文传递。如果在延迟期间有一个报文到达，ripout就放弃延迟并立即发送更新报文。

```
/* ripout.c - ripout */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <sleep.h>
#include <proc.h>

#ifdef RIP
/*------------------------------------------------------------------------
 *  ripout  -  do the RIP route exchange protocol, output side
 *------------------------------------------------------------------------
 */
PROCESS ripout(void)
{
        int  rnd;

        rippid = getpid();
        dorip = TRUE;
        /* advertise our routes */

        rnd = 0;   /* "randomizer" */
        while (TRUE) {
                sleep10(RIPOUTMIN);     /* minimum time between each    */
                if (++rnd > RIPDELTA)
                        rnd = -RIPDELTA;
                recvtim(RIPINT-RIPOUTMIN+rnd);
                wait(riplock);
                ripsend(ip_maskall, UP_RIP);
                signal(riplock);
        }
}
#endif  /* RIP */
```

尽管代码看起来简单，但它处理了三个细节。第一，ripout在调用ripsend之前等待信号量riplock，以避免在其形成并发送了一个完整的更新报文之前发生触发更新。第二，ripout在更新报文之间插

人短暂的 RIPOUTMIN 个 0.1 秒的时延（即使这些更新是被触发的）。为了保证最短时延，它先直接调用 sleep10 达到 RIPOUTMIN 个 0.1 秒，然后在余下的时间里调用 recvtim。

在计算到调用 recvtim 的时延的尚需时间时，ripout 以标准的 RIP 更新周期开始，减去已经发生的最小时延，最后添加一个小的“随机”整数。这个小的随机时延帮助 RIP 避免由触发更新带来的广播风暴。

代码通过保留一个全局整数并在每次调用时将其值加 1 来模拟随机时延。当整数大于 RIPDELTA 时，ripout 将其设为负的 RIPDELTA 并继续工作。

18.13　RIP 的局限性

RIP 的设计限制了它适用的环境。第一，由于 RIP 将 16 作为无穷大，它不能够在“直径”大于 15 的互联网中使用。第二，由于 RIP 在更新周期和超时机制里使用固定值，因而不能在丢失率高的网络里使用。第三，由于 RIP 在传播路由时使用固定的费用值，因而不能在使用动态方法选择路由（例如，根据当前时延和目前负荷选择路由）的互联网中使用。

18.14　小结

选路信息协议 RIP 使用矢量距离算法来传播路由。为了维持其稳定性，RIP 应用了诸如水平划分法、路由超时和毒性逆转等技巧。

我们研究了一种分别由输入进程和输出进程来处理 RIP 的实现方案。输入进程接受一个直接查询，一个对直接查询的应答，或者是给本地网上所有机器的周期响应广播。输出进程生成更新报文的周期性广播或触发更新报文。

要触发一个更新报文，任何程序可以直接发送一个报文给输出进程。为了避免在路由变动期间形成被触发报文，RIP 软件使用一个信号量 riplock。输出进程在准备一个更新报文时等待该信号量。

深入研究

Hedrick［RFC 1058］定义了 RIP 协议，并说明了本章中讨论到的技巧。它同时还提供处理输入请求和响应的算法。

习题

18.1 过程 ripsend 将数组 rqinfo 作为自动的堆栈变量。如果它将数组作为静态的全局变量，会出现什么情况？

18.2 仔细阅读程序代码，它是如何处理特殊请求的？

18.3 考虑一下 RIP 和 IP 之间的交互作用。描述一个设计方案，将 RIP 用来选择路由的策略和 IP 用来转发数据报的策略分开。

18.4 我们的 RIP 软件能够传播多个默认路由吗？试解释之。

18.5 假设一个网络管理员需要限制路由传播。你将如何修改程序代码，以允许管理员声明一些路由不能够被传播？

第19章 OSPF：使用SPF算法的路由传播

19.1 引言

第18章描述了一种内部网关协议，它使用矢量距离算法在网关和主机中传播路由信息。本章将详细讲解另一种称为“开放最短路径优先”（Open Shortest Path First）的内部网关协议。OSPF使用链路状态算法来传播选路信息。

对使用矢量距离算法的协议来说，每个网关都会传播一张列表，表中的内容是该网关可达的网络及它与网络之间的距离。对使用链路状态的协议来说，每个网关都会分别传播它与网络的每一个连接的状态。该协议向所有有关的网关传播每个链路的状态信息。当链路状态信息到达特定的网关后，该网关将信息加入它当前的拓扑数据库中。如果输入的链路状态信息改变了网关数据库中的内容，该网关必须重新计算路由。为了计算通过最短的路径到达每个目的站的下一跳，网关运行一个本地计算程序。

19.2 OSPF配置和选项

由于OSPF试图包容范围很广的各种互联网的构造和路由配置，因而协议中包括了许多选择。例如，试想在一个互联网中，两个网关之间由一条串行线路连接。常规的IP实现方案将每个这样的连接线路当作网络对待，并要求该线路也被赋予一个IP地址。然而一些IP实现方案通过允许两个网关通过匿名连接（anonymous connection），也就是一条没有IP地址的线路相互通信来节省协议地址。与大多数选路协议不同，OSPF不但能传播常规连接的路由信息，同时也可以传播匿名连接的路由信息。

OSPF的通用性是有代价的，该协议大而复杂，并且它的规范不容易理解。最重要的是，为了做到完全的通用性，设计OSPF软件时必须要使它能够为特殊的互联网做特殊的配置。事实上，协议包含了许多可能的变化和特例。例如，虽然OSPF是作为匿名系统中使用的内部网关协议而设计的，但它允许系统管理员把该匿名系统分割成许多子系统，称作“区域”（area）。每个网关必须被放置在其中的一个区域内，并且OSPF对一个区域边界上的网关如何与另一个区域边界上的网关相互传递选路信息做了具体规定。

为了有助于理解OSPF，而不在其通用性和特例中纠缠不清，本章将分别讲解每个协议的性能。下一节描述了用图解表示的OSPF网络连通性的理论模型，并定义了一些重要的概念和本章中使用的术语。后几节体现了一个OSPF网关如何寻找发现相邻网关；如何维护管理有关它们的信息；一组与某个网络相连的OSPF网关如何决定惟一的、特殊的网关来为该网络处理链路状态的更新；OSPF如何传播链路状态信息以及如何计算路由。

19.3 OSPF的图论模型

正像大多数链路状态算法一样，OSPF利用网络拓扑的图论模型来计算最短路径。每个网关定

期广播有关它与各网络的连接状态信息，OSPF用洪泛法将每一个状态报文发给所有参与的网关。网关利用接收到的链路状态信息组成一张图。当一个网关接收到改变其拓扑图的信息后（例如，由于连接中断），它运行一种常规的图算法来计算图中的最短路径，并利用其结果建立新的下一跳选路表。

OSPF 将互联网的模型用一张有向图表示。因为 OSPF 图将互联网的连接模型化，所以有时它被称为拓扑图。OSPF 拓扑图中的每个节点或者对应一个网关，或者对应一个网络。如果互联网中两实体之间存在物理连接，则OSPF图在代表实体的两个节点之间有一对有向边（每个方向一条）。例如，图 19.1 表示出了位于一个网关和一个网络之间的连接，以及该连接在 OSPF 图中对应的有向边。

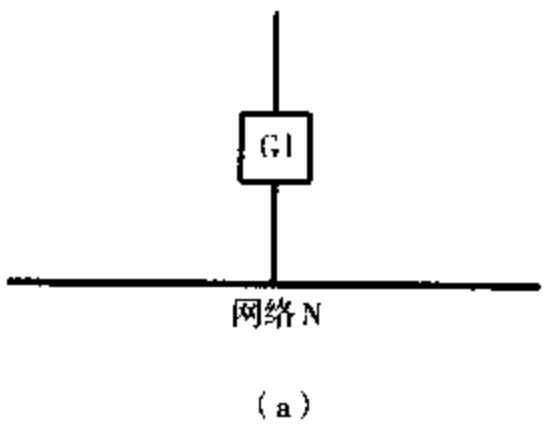


图 19.1　（a）一个网关和一个网络之间的连接的例子；（b）在 OSPF 图中对应的一对有向边

OSPF使用术语“多重接入网络”（multiaccess network）表示连接有多个网关的网络（例如，连接两个网关的以太网）。多重接入网络的 OSPF 图包括代表该网络的一个节点、代表每个网关的节点，以及代表网关和网络之间的每个连接的一对有向边。例如，图19.2表示了一个连接多个网关的多重接入网络及其对应的 OSPF 图。

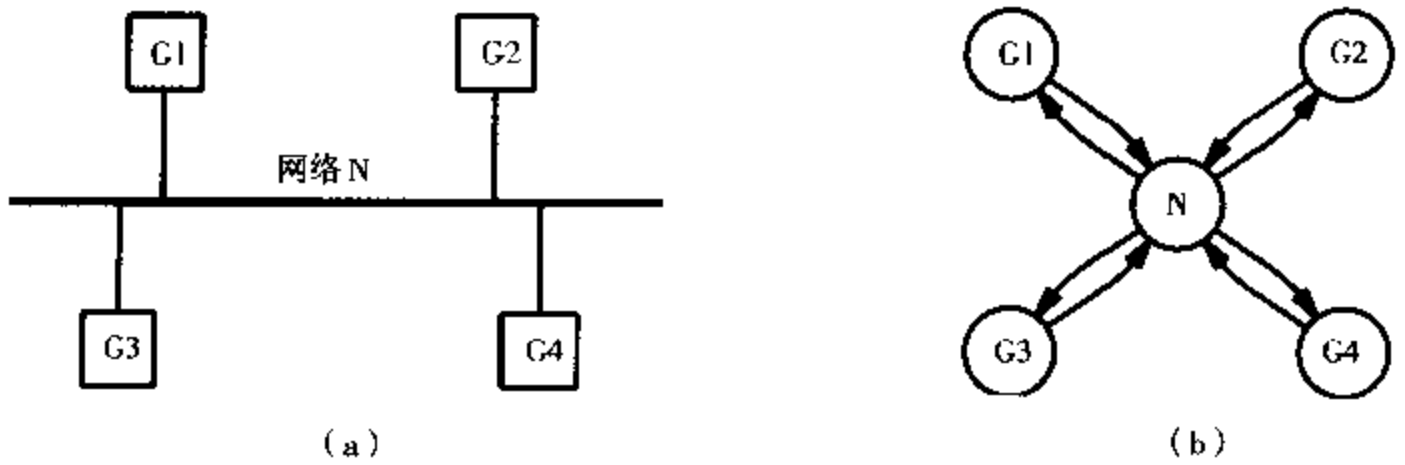


图 19.2　（a）与四个网关连接的多重接入网络；（b）这四个连接对应的 OSPF 图

OSPF还允许用图来表示一对网关之间的匿名点到点的连接模型。在使用图表示匿名串行连接时，OSPF 使用一对有向边来连接代表两个网关的节点。因此与大多数的多重接入网络不同，匿名串行连接的图中不包含代表该串行线路的节点。

为了使网关能够计算出最短的路径，OSPF 图中的每个边都有一个“权”（weight），它对应于使用该路径所需的花费。设置OSPF的网络管理员通过给每个网络接口一个正数的方式来设置该权值。为每个具体接口设置的权的大小可根据管理或技术的需要来选取。例如，该权值可能反映了使用接口所需的实际费用，或是通过接口的网络带宽，或者是出于某种管理策略，鼓励或节制人们使用该接口。图 19.3 体现了一个为每个接口都设置了费用的互联网的例子。

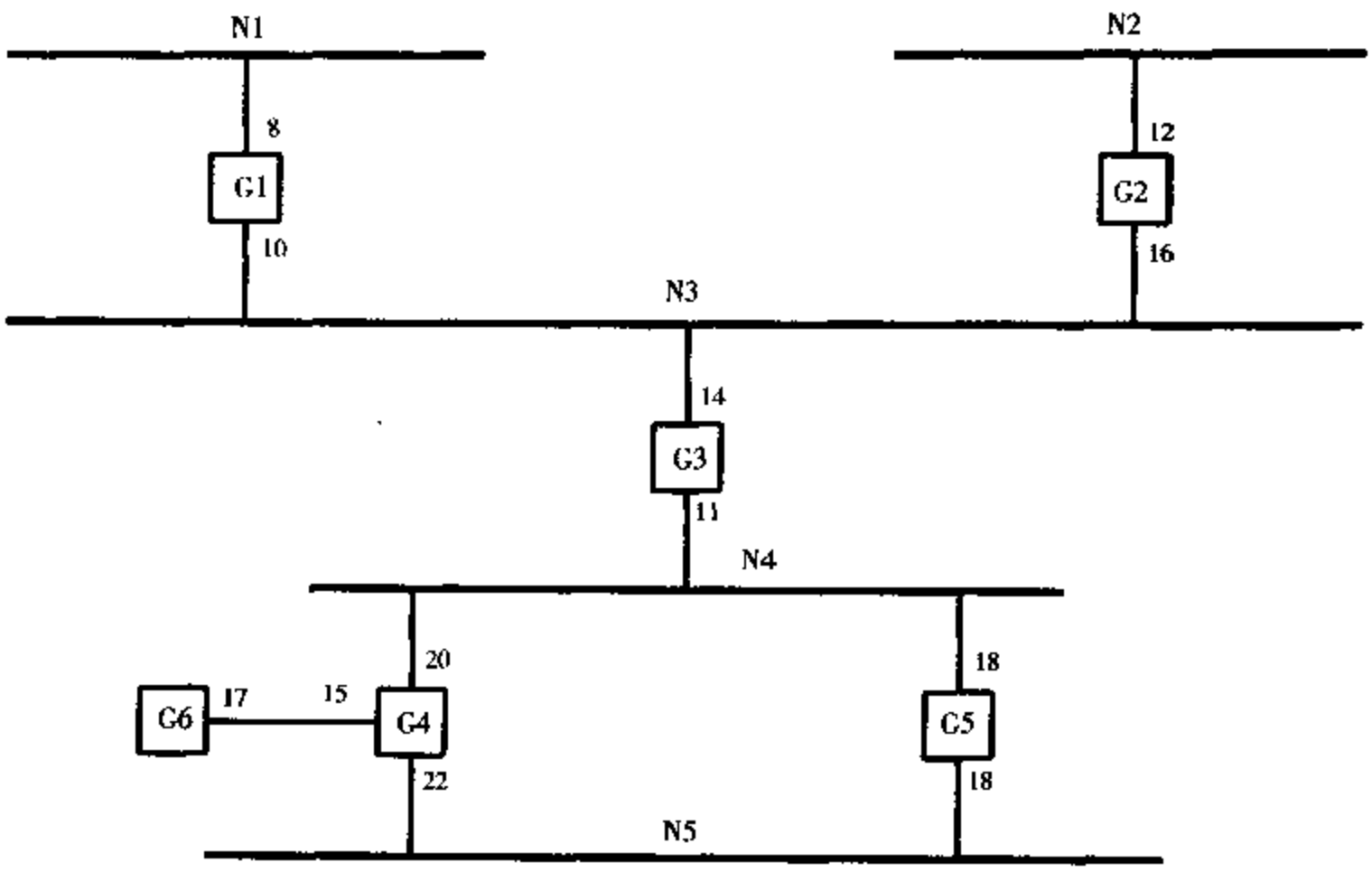


图 19.3 每个接口都被赋予费用的网络和网关组成的互联网的例子

像大多数选路算法一样，OSPF根据沿着花费最小的路径转发数据报的原则建立选路表。因此虽然图19.3中的网关G4和G5提供了网络N4的N5之间的连接，但OSPF为数据报选择的路由将通过网关G5，因为它具有较低的权值。

图19.4所示的是图19.3中的互联网的图论模型。图中包含了代表每个网关的节点和代表每个多通路网络的节点，但却没有代表网关G4和G6之间的匿名连接的节点。事实上，有一对边直接连接G4节点和G6节点。

如图所示，OSPF图中的每一条有向边（从代表一个网关的节点出发，到代表一个网络的节点结束）都有一个权，该权值的大小等同于使用该接口所需的花费。然而每条从代表网络的节点出发，到代表网关的节点结束的有向边的权值都为零。使用这种不对称的权值设置的原因很简单：为一条从网关节点到网络节点的边分配一个独立的权值就可达到为每条连接设置一个费用的目的。

为每条从网络节点到网关节点的有向边设置一个零权，保证了在网络中经过任一路径时只计算一次费用。因此网络管理员可以使经过一个网关接入某个网络的费用大于经过另一个网关接入同样的网络的费用，以此来达到管理上的控制。另外，零权的存在使得费用更容易被了解，因此管理员能够沿着一条从源站到目地站的路径计算所需的总费用。总结如下：

由于OSPF使用了互联网的有向图论模型来计算最短路径，而且网络管理员在设置费用时不依赖于网络硬件，所以从代表网络的节点出发，到代表网关的节点结束的有向边的权值为零。

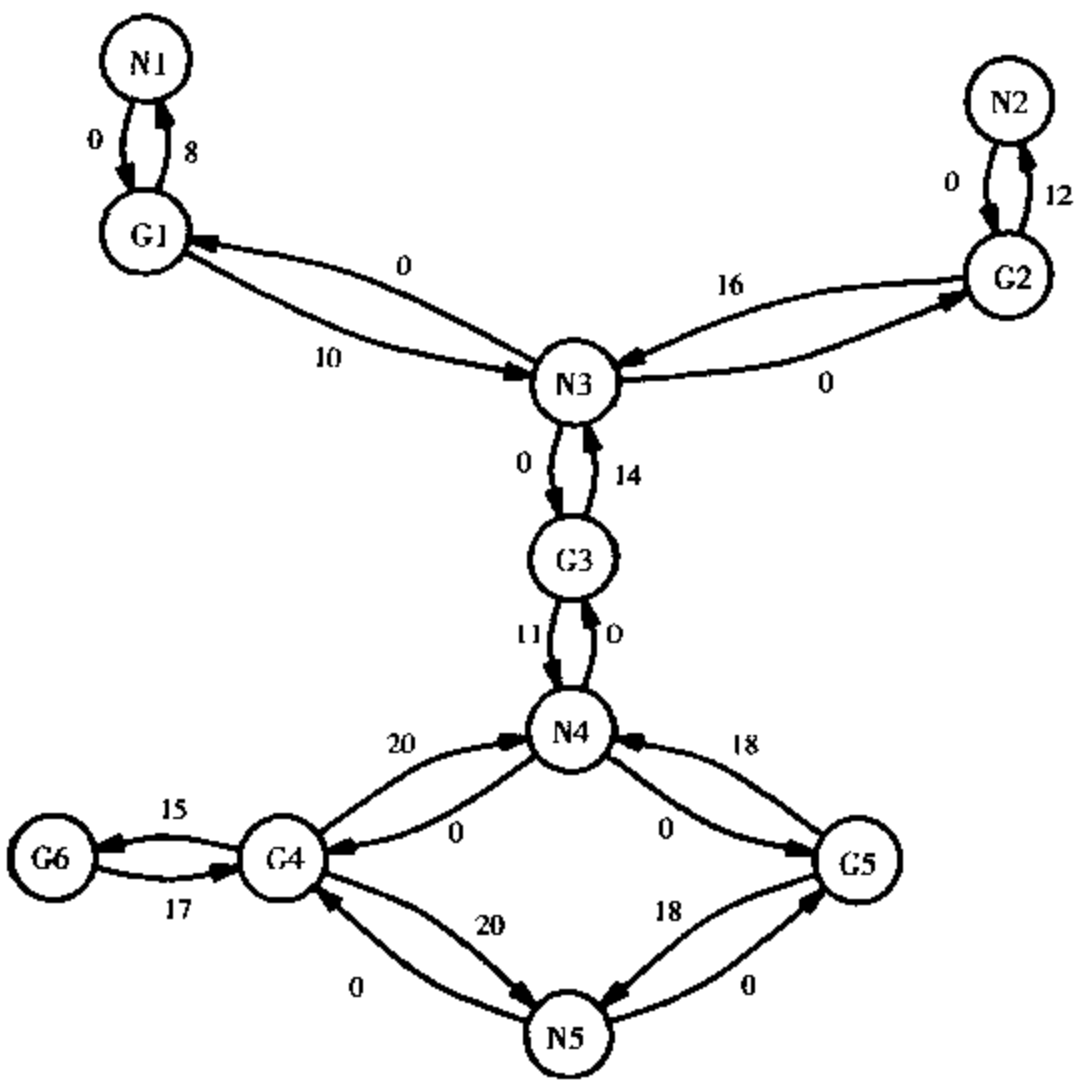


图 19.4　图 19.3 所示的互联网的 OSPF 图。图中每条边上的数字对应于该边所代表的网络连接的权值

19.4　OSPF 的说明

为了解 OSPF 在传播信息和计算最短路径时是如何使用图论模型的，有必要先理解包括 OSPF 分组格式和内部数据结构在内的许多细节问题。这一节给出了三个定义文件，适用于整个程序代码段。

19.4.1　OSPF 分组格式的说明

文件 ospf_pkt.h 中包含了一些说明和常量，这些说明和常量定义了 OSPF 分组格式和首部常量。

```
/* ospf_pkt.h */

/* OSPF packet format */

struct ospf {
        unsigned char  ospf_version;        /* Version Number          */
        unsigned char  ospf_type;           /* Packet Type             */
        unsigned short ospf_len;            /* Packet Length           */
        unsigned long  ospf_rid;            /* Router Identifier       */
        unsigned long  ospf_aid;            /* Area Idenifier          */
        unsigned short ospf_cksum;          /* Check Sum               */
        unsigned short ospf_authtype;       /* Authentication Type     */
```

```
        unsigned char  ospf_auth[AUTHLEN];/* Authentication Field        */
        unsigned char  ospf_data[1]
};

#define MINHDRLEN      24              /* OSPF base header length      */

/* OSPF Packet Types */

#define T_HELLO        1               /* Hello packet                 */
#define T_DATADESC     2               /* Database Description         */
#define T_LSREQ        3               /* Link State Requet            */
#define T_LSUPDATE     4               /* Link State Update            */
#define T_LSACK        5               /* Link State Acknowledgement   */
/* OSPF Authentication Types */

#define AU_NONE        0               /* No Authentication            */
#define AU_PASSWD      1               /* Simple Password              */

/* OSPF Hello Packet */

struct ospf_hello {
        unsigned long  oh_netmask;     /* Network Mask                 */
        unsigned short oh_hintv;       /* Hello Interval(seconds)      */
        unsigned char  oh_opts;        /* Options                      */
        unsigned char  oh_prio;        /* Sender's Router Priorty      */
        unsigned long  oh_rdintv;      /* Seconds Before Declare Dead  */
        unsigned long  oh_drid;        /* Designated Router ID         */
        unsigned long  oh_brid;        /* Backup Designated Router ID  */
        unsigned long  oh_neighbor[1]; /* Living Neighbors             */
};

#define MINHELLOLEN    (MINHDRLEN+20)

/* OSPF Database Description Packet */

struct ospf_dd {
        unsigned short  dd_mbz;        /* Must Be Zero                 */
        unsigned char   dd_opts;       /* Options                      */
        unsigned char   dd_control;    /* Control Bits (DDC_* below)   */
        unsigned long   dd_seq;        /* Sequence Number              */
        struct ospf_lss dd_lss[1];     /* Link State Advertisements    */
};

#define MINDDLEN       (MINHDRLEN+8)

#define DDC_INIT       0x04            /* Initial Sequence             */
#define DDC_MORE       0x02            /* More to follow               */
#define DDC_MSTR       0x01            /* This Router is Master        */
```

19.4.2 OSPF 接口说明

网关在结构ospf_if中保存诸如为它的每个接口设置费用之类的信息。文件ospf_if.h中包含了这些定义。

```
/* ospf_if.h */

/* OSPF Interface Information */

struct ospf_if {
        unsigned int    if_type;        /* one of IFT_* below          */
        unsigned char   if_state;       /* one of IFS_* below          */
        unsigned char   if_event;       /* one of IFE_* below          */
        unsigned int    if_hintv;       /* Hello Packet Interval       */
        unsigned int    if_rdintv;      /* Router Dead Interval        */
        unsigned int    if_rintv;       /* Retransmit Interval         */
        timer_t         if_twait;       /* Wait Timer for WAITING state */
        unsigned int    if_xdelay;      /* Estimated Transmission Delay */
        IPaddr          if_dipa;        /* IP Address of desig. router */
        IPaddr          if_bipa;        /* IP address of backup router */
        unsigned int    if_metric;      /* Cost to use this Interface  */
        unsigned char   if_auth[AUTHLEN];   /* Authentication Key      */
        int             if_nbmutex;     /* Mut. Excl. Semaphor for nbtab */
        struct ospf_ar  *if_area;       /* Area Structure Back-pointer */
        struct ospf_if  *if_next;       /* Next Interface for Area     */
        struct ospf_nb  if_nbtab[MAXNBR+1]  /* Neighbor  list this net */
};

#define if_rid          if_nbtab[0].nb_rid
#define if_prio         if_nbtab[0].nb_prio
#define if_drid         if_nbtab[0].nb_drid
#define if_brid         if_nbtab[0].nb_brid
#define if_opts         if_nbtab[0].nb_opts

/* Interface Types */

#define IFT_BROADCAST 0                 /* Interface Supports Broadcast */
#define IFT_MULTI     1                 /* Interface Supports Multicast */
#define IFT_PT2PT     2                 /* Interface is Point-to-point  */
#define IFT_VIRTUAL   3                 /* Interface is a Virtual Link  */

/* Interface States */

#define IFS_DOWN      0                 /* Interface non-functional     */
#define IFS_LOOPBACK  1                 /* Interface in loopback mode   */
#define IFS_WAITING   2                 /* Electing Backup Desig.Rtr.   */
#define IFS_PT2PT     3                 /* Interface is Point-to-Point  */
```

```
#define IFS_DROTHER    4                /* not Designated Router        */
#define IFS_BACKUP     5                /* Is Backup Designated Router  */
#define IFS_DR         6                /* Is Designated Router         */

/* Scheduled Interface Events */

#define IFE_BSEEN  0x01                 /* Have Seen Backup DR Claim    */
#define IFE_NCHNG  0x02                 /* A Neighbor State Changed     */
```

19.4.3 全局常量和数据结构的说明

文件ospf.h中定义了其余的适用于整个OSPF的数据结构和常量，还引用了其他的.h文件。

```
/* ospf.h */

/* OSPF process info */
extern  PROCESS  ospf();
#define OSPFSTK  4096                   /* stack size for OSPF input    */
#define OSPFPRI    90                   /* OSPF priority                */
#define OSPFNAM  "ospfinp"              /* name of OSPF input process   */
#define OSPFARGC    0                   /* count of args to ospfinp     */

extrn PROCESS    ospfhello();
#define OSPFHSTK 4096                   /* stack size for OSPF hello    */
#define OSPFHPRI   90                   /* OSPE priority                */
#define OSPFHNAM  "ospfhello"           /* name of OSPF hello process   */
#define OSPFHARGC   0                   /* count of args to ospfhello   */

#define OSPFQLEN   20                   /* OSPF input port queue length */
#define MAXNBR     32                   /* Max # active neighbors       */

/* Manifest OSPF Configuration Parameters */

#define OSP_VERSION  2                  /* protocol version number      */
#ifdef notdef
#define HELLOINTV    10                 /* Hello Interval (seconds)     */
#endif
#define HELLOINTV    30                 /* hello Interval (seconds)     */
#define DEADINTV     4*HFLLOINTV        /* Router Dead Interval (secs)  */
#ifdef notdef
#define RXMTINTV     5                  /* Retransmit Interval (secs)   */
#endif
#define RXMTINTV     15                 /* Retransmit Interval (secs)   */
#define AUTHLEN      8                  /* 64-bit Password              */

typedef int     timer_t;                /* OSPF count-down timers(secs) */

/* Area Information */
```

```
struct ospf_ar {
        unsigned long  ar_id;              /* Area Identifier          */
        unsigned char  ar_authtype;        /* Authentication Type      */
        unsigned char  ar_auth[AUTHLEN];   /*Password, if ARA_PASSWD */
        int            ar_dbmutex;         /* Mutex for Top.Database */
        int            ar_hmod;            /* Hash Table Modulus       */
        struct ospf_db **ar_htable;        /* LSA Hash Table           */
        struct ospf_db *ar_dblhead;        /* Head of Top.Database     */
        struct ospf_if *ar_if;  /* List of Interfaces This Area        */
};

#define ARA_NONE      0              /* No Authentication for Area   */
#define ARA_PASSWD    1              /* Simple-password Auth.        */

/* Neighbor Information */

struct ospf_nb {
        unsigned char  nb_state;      /* neighbor conversation state  */
        unsigned long  nb_seq;        /* DD packer sequence number    */
        unsigned long  nb_rid;        /* neighbor's router ID         */
        unsigned char  nb_prio;       /* neighbor's router priority   */
        IPaddr         nb_ipa;        /* neighbor's IP address        */
        unsigned char  nb_opts;       /* options                      */
        unsigned long  nb_drid;       /* neighbor's designated router*/
        unsigned long  nb_brid;       /* backup designated router     */
        Bool           nb_master;     /* nonzero if we are master     */
        timer_t        nb_lastheard;  /* seconds since last HELLO     */
        timer_t        nb_trexmt;     /* Retransmit Timer (secs)      */
        timer_t        nb_tlastdd;    /* Slave Last DD pkt. Timer     */
        timer_t        nb_tlsr;       /* LSR Retransmit Timer         */
        int            nb_lsal;       /* Link Status Adv. List        */
        int            nb_dsl;        /* Database Summary List        */
        int            nb_lsrl;       /* Link Status Req. List        */
};

/* Neighbor States */

#define NBS_DOWN  0
#define NBS_ATTEMPT   1
#define NBS_INIT      2
#define NBS_2WAY      3
#define NBS_EXSTART   4
#define NBS_EXCHNG    5
#define NBS_LOADING   6
#define NBS_FULL      7

/* Neighbor Options */
```

```
#define NBO_T          0x01            /* Router Does Supports IP TOS  */
#define NBO_E          0x02            /* Can Use External Routes      */

#define NBMAXLIST      10              /* Max Retrans.Queue Lengths    */

#include "ospf_ls.h"
#include "ospf_pkt.h"
#include "ospf_if.h"
#include "ospf_db.h"

extern struct ospf_ar   ospf_ar[];
extern struct ospf_if   ospf_if[];
extern int              ospf_iport;
extern int              ospf_lspool;
extern IPaddr           AllSPFRouters, AllDRouters;

void ospfinit(void, if_elect(unsigned);
int ospfcheck(struct ep *);
int nb_rexmt(struct ospf_if *, struct ospf_nb *);
int nb_aok(struct ospf_if *, struct ospf_nb *);
int nb_makel(struct ospf_nb *), nb_clearl(struct ospf_nb *);
int nb_mismatch(struct ospf_if *, struct ospf_nb *);
struct ospf_nb *nb_add(struct ospf_if *, struct ospf *);
struct ospf *ospfnet2h(struct ospf *);
void ospf_hin(struct ep *), ospf_ddin(struct ep *), lsr_in(struct ep *);
void lsu_in(struct ep *),  lsack_in(struct ep *);
int lsr_xmt(struct ospf_if *,  struct ospf_nb *);
void dd_queue(struct ospf_if *, struct ospf_nb *);
void dd_xmit(struct ospf_if *, struct ospf_nb *);
struct ep *ospfddtmpl(struct ospf_if *);
void ddi_exstart(struct ospf_if *, struct ospf_nb *, struct ep *);
int ddi_exchng(struct ospf_if *, struct ospf_nb *, struct ep *);
int ddi_full(struct ospf_if *, struct ospf_nb *, struct ep *);
```

19.5 邻接关系和链路状态信息的传播

一组运行OSPF的网关使用洪泛法来传播链路状态报文（即，它们向所有有关的网关传播每个链路的状态报文）。为了保证该组网关中的所有成员都接收到某个报文，这些网关在它们自身之间形成逻辑连接，并沿着这些连接传递链路状态报文的副本。通常两个网关之间的每条逻辑连接都对应着一条物理连接（例如，连接起两个网关的一条线路），但是也可以设置OSPF，使得在两个网关之间形成一条跨越中间网关和网络的逻辑连接。

最基本的洪泛法往往是只要需要就复制报文，因而能够迅速地扩散链路状态报文，当网关从某条连接上接收到一个报文时，就在它的其他所有连接上发送该报文的副本。然而，OSPF的洪泛法却不会盲目地将接收到的每一条链路状态报文进行很多次复制并继续转发。特别是，如果多个网关

之间的连接形成环路时，OSPF 保证这些网关不会沿着环路永久地发送某个报文的副本。

为了防止网关发送多余的、不必要的报文副本，OSPF 检查接收到的每个报文。当到达网关的一个报文中携带有新信息时，该网关用洪泛法在它的每条逻辑连接上发送报文的副本。当一个重复的报文到达时，网关丢弃该报文，而不发送它的副本。因此，OSPF 并不依靠网络管理员来设置不会构成环路的连接，而是由协议自动地消除了其重复性。

两个网关之间的每一条逻辑连接称作一个邻接（adjacency）。理解OSPF邻接的最简单的办法是想像一张无向图，其中每个节点对应一个网关，而每条边对应OSPF的一个邻接[1]。OSPF 网关沿着由邻接图规定的路径传播路由信息，因此在协议正确地运行时，所有网关必须能够以通过邻接相互通信。也就是，对于任意两个网关 G1 和 G2，G1 必须能够使用完全由一些邻接网关构成的路径到达 G2。

在大多数情况下，网络管理员设置OSPF使每两个相邻的网关之间形成一个邻接。例如，由一条串行线路连接的两个相邻网关通常就形成一个邻接，并且通过该连接发送路由信息。然而邻接关系是动态的——当一个网络或网关出现故障时，邻接可能会改变，以保证所有有关的网关能继续接收路由信息。

看起来似乎每一对相邻的网关之间都应当形成一个邻接，然而如果所有的网关对之间都形成邻接，就会产生不必要的链路状态报文的副本被继续转发，以致于浪费带宽。过多的邻接所引起的问题在类似以太网的多重接入网络中尤其严重，因为与某个网络相连接的任一对网关之间都存在有物理路径。如果某个网络上接有 N 个网关，则它们能够形成 N(N−1)/2 个可能的邻接。每当某个网关传送一个报文时，它会向所有 N−1 个邻接网关发送该报文的副本。在最糟糕的情况下，每个相邻的网关都立即向它的所相邻网关发送该报文。因此，在网关察觉这种重复性的行动，并停止发送副本之前，该报文会两次通过大多数的邻接，其结果是大约有 N^2 个报文的副本被发送，虽然事实上只需要 N 个副本就足够了。

19.6　用 Hello 发现相邻网关

为了找到相邻的网关并维护适当的邻接关系，OSPF 使用 Hello 协议[2]。使用 Hello 协议的一对网关定期地交换分组，以保证双方都处于活动状态，并且在邻接关系上达成共识。

有两个原因使 Hello 对类似以太网的多重接入网络特别重要，首先，虽然网关随时都可能发生崩溃或重启动，但网络硬件并不检查或报告这一类的事件。因此，为了互相保留对方的状态信息，与某个多重接入网络相连接的两个网关必须交换分组。第二，与某个多重接入网络相连接的任意两个网关之间都能够直接通信，因此 OSPF 必须防止这些网关形成过多的邻接。

OSPF标准使用了一个有限状态机来规范使用Hello的网关如何与相邻网关交互作用。图19.5所示为有限状态机的状态和它们之间主要的状态变迁。

一般说来，所有相邻网关最初都处于 DOWN 状态，表示并未准备通信。当一个网关接收到相邻网关发出的 Hello 分组后，它将相邻网关从 DOWN 状态变迁到 INIT 状态。在此之后，相邻网关或者进入 2-WAY 状态，或者进入 EXSTART 状态。其中 2-WAY 状态表示通信已经建立，但相邻网关与该网关之间没有邻接关系，EXSTART 状态表示不但已经建立通信，而且两个网关之间存在经双方协商同意的邻接关系。

---

① 邻接的概念是从图论中借用而来的。

② OSPF 的 Hello 协议与具有同样名字的一个旧协议不同。

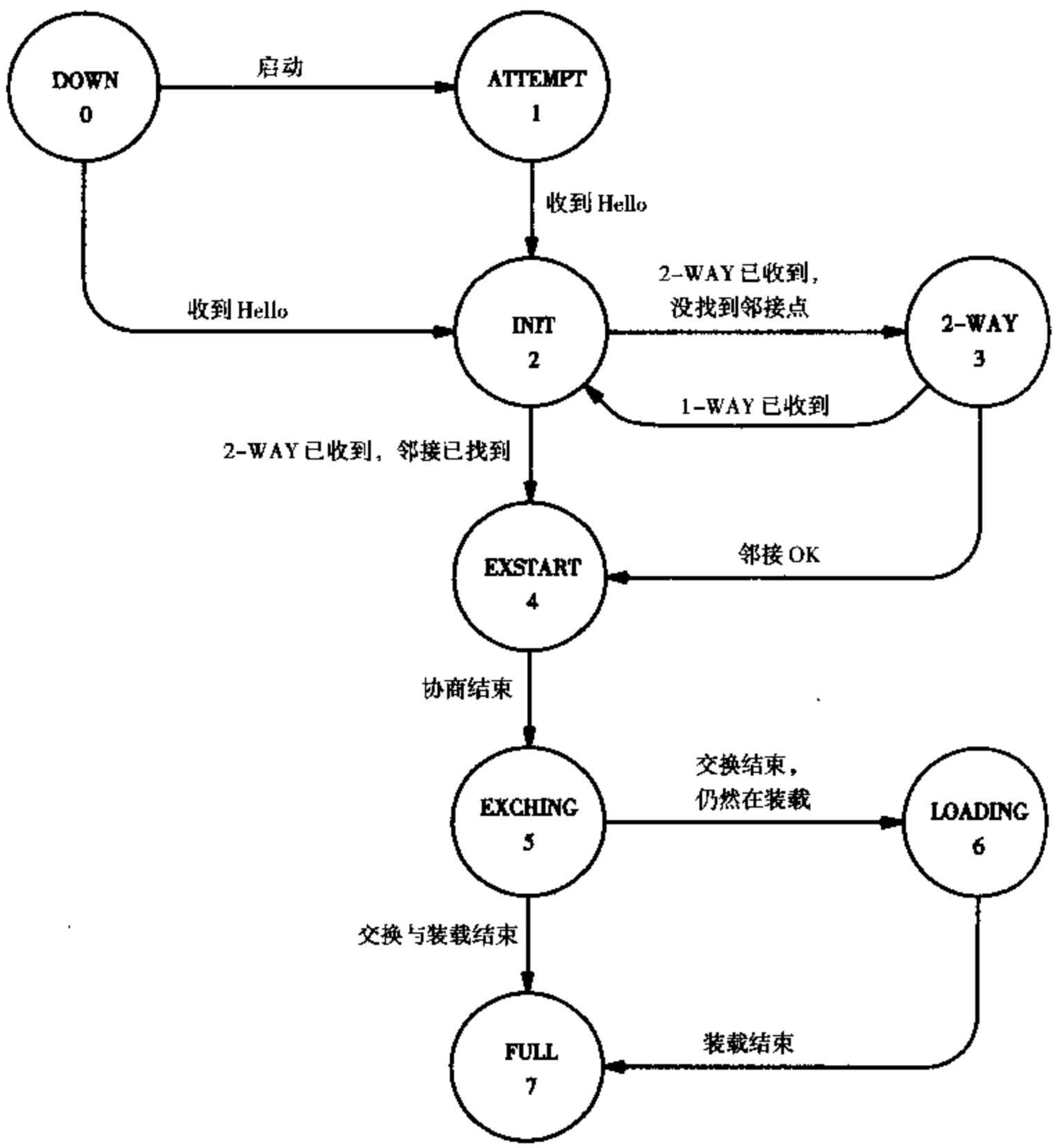


图 19.5 根据 OSPF 的 Hello 协议，相邻网关有八种可能状态。如图所示，范例软件遵循标准规定用数字为这些状态排定一个线性顺序

当协商结束后，网关开始交换拓扑数据库中的信息，以确保它们有完全相同的底层互联网拓扑图。两个邻接网关中的一个成为“主网关”，它查询另一个网关数据库中的信息。非主网关返回数据库描述分组，以通知主网关最近接收到的该拓扑图中每条链路的信息。在建立邻接关系时，交换信息尤其重要，因为在网络断连期间，某个网关中的信息可能变为过时的信息。每个拓扑信息分组中包含一个序号，因此网关能够知道相邻网关数据库中的描述信息是否比该网关自身数据库中的信息更新。在交换完成且所有拓扑信息都已装载后，网关进入 FULL 状态。在 FULL 状态中，两个网关定期交换分组，以保持连接。

OSPF 使用的相邻网关有限状态机与第 11 章和第 12 章中所介绍的 TCP 有限状态机之间非常相似。这两种有限状态机都为软件规定了几种基本状态，而且它们都描述了当一个消息到达时，软件应当如何响应。同时在两种状态机中，每个状态都有与之相关的复杂的行动说明。因此，相

邻网关有限状态机的实现与TCP有限状态机的实现是类似的。虽然协议标准规定了在每个状态中OSPF应当如何操作，但处理给定状态的程序代码却不能独立地用一个过程来完成，该软件中还存有很多细节问题。

19.7 Hello 分组的发送

在过程 ospf_hsend 中可找到使用状态信息的例子，它通过指定的接口发送 Hello 分组。

```
/* ospf_hsend.c - ospf_hsend */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

struct ep *ospfhtmpl(struct ospf_if *);

/*------------------------------------------------------------------------
 * ospf_hsend - send OSPF hello packer for an interface
 *------------------------------------------------------------------------
 */
int
ospf_hsend(ifn)
unsigned    ifn;
{
        struct ospf_if          *pif = &ospf_if[ifn];
        struct ep               *pep;
        struct ip               *pip;
        struct ospf             *po;
        struct ospf_hello       *poh;
        struct ospf_nb          *pnb;
        int                     i, nn, len;

        pep = ospfhtmpl(pif);
        wait(pif->if_nbmutex);
        pip = (struct ip *) pep->ep_data;
        po = (struct ospf *) pip->ip_data;
        poh = (struct ospf_hello *) po->ospf_data;
        pnb = &pif->if_nbtab[1];
        for (i=0, nn=0; i<MAXNBR; ++i, ++pnb)
                if (pnb->nb_state >= NBS_INIT) {
                        poh->oh_neighbor[nn++]=pnb->nb_rid;
                        po->ospf_len += sizeof(poh->oh_neighbor[0]);
                }
        signal (pif->if_nbmutex);
```

```
poh->oh_netmask = nif[ifn].ni_mask;
po->ospf_authtype = net2hs(pif->if_area->ar_authtype);
len = po->ospf_len;
po->ospf_len = net2hs(po->ospf_len);
pep->ep_order &= ~EPO_OSPF;     /* all in net order now */
memset(po->spf_auth, 0, AUTHLEN);
po->ospf_cksum = 0;
po->ospf_cksum = cksum((WORD *)po, len);
memcpy(po->ospf_auth, pif->if_area->ar_auth, AUTHLEN);
pip->ip_src = nif[ifn].ni_ip;
if (ifn == NI_PRIMARY)
        ipsend(AllSPFRouters , pep, len, IPT_OSPF,
                IPP_INCTL, 1);
else
        ipsend(nif[ifn].ni_brc, pep, len, IPT_OSPF,
                IPP_INCTL, 1); }
}
#if    BYTE_ORDER == LITTLE_ENDIAN
IPaddr AllSPFRouters   = 0x050000E0;    /* 224.0.0.5 */;
IPaddr AllDRouters     = 0x060000E0;    /* 224.0.0.6 */;
#else   /* BYTE_ORDER == LITTLE_ENDIAN */
IPaddr AllSPFRouters   = 0xE0000005;    /* 224.0.0.5 */;
IPaddr AllDRouters     = 0xE0000006;    /* 224.0.0.6 */;
#endif  /* BYTE_ORDER == LITTLE_ENDIAN */
```

ospf_hsend调用ospfhtmpl为Hello分组分配缓冲区，并填写首部中的字段。然后，它等待信号量if_nbmutex以确保单独地访问该接口的相邻网关表中的数据结构。在获得单独访问权后，ospf_hsend检查该接口的相邻网关表中的每一个表项，对于每个处于NBS_INIT状态或大于该状态的相邻网关来说，ospf_send在Hello分组中加入该相邻网关。

一旦分组构成，odpf_hsend计算校验和并调用ipsend发送该分组。只要有可能，OSPF使用IP组播来发送Hello分组，它的IP组播地址为AllSPFRouters，在无法使用组播时，OSPF使用硬件广播发送Hello分组。由于实现方案的主要网络接口支持组播，因此ospf_hsend检查该接口，以决定是使用组播还是广播来发送外发分组。

19.7.1 Hello分组的一个模板

ospf_hsend调用ospfhtmpl生成一个Hello分组。

```
/* ospfhtmpl.c - ospfhtmpl */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>
```

```
/*------------------------------------------------------------------------
 * ospfhtmpl - fill in OSPF HELLO packet template
 *------------------------------------------------------------------------
 */
struct ep *ospfhtmpl(pif)
struct ospf_if *pif;
{
        struct ep           *pep,
        struct ip           *pip;
        struct ospf         *po;
        struct ospf_hello   *poh;

        pep = (struct ep *)getbuf(Net.netpool);
        if ((int)pep == SYSERR)
                return 0;
        pep->ep_order = -0;
        pip = (struct ip *) pep->ep_data;
        po = (struct ospf *) pip->ip_data;
        poh = (struct ospf_hello *) po->ospf_data;
        po->ospf_version = OSPF_VERSION;
        po->ospf_type = T_HELLO;
        po->ospf_len = MINHELLOLEN;
        po->ospf_rid = pif->if_rid;
        po >ospf_aid = pif->if_area->ar_id;
        po->ospf_authtype = pif->if_area->ar_authtype;
        memcpy(po->ospf_auth, pif->if_auth, AUTHLEN);
        poh->oh_hintv = net2hs(pif->if_hintv);
        poh->oh_opts = pif->if_opts;
        poh->oh_rdintv = net2hl(pif->if_rdintv);
        poh->oh_prio = pif->if_prio;
        poh->oh_drid = pif->if_drid;
        poh->oh_brid = pif->if_brid;
        return pep;
}
```

ospfhtmpl 调用 getbuf 分配一个缓冲区，然后填写 Hello 分组的首部字段。由于分组中包含了某个接口的信息，调用者必须以入口参数的形式指定该接口。在范例代码中，参数 pif 包含了一个指向接口结构的指针，ospfhtmpl 从该接口结构中为首部字段提取信息。例如，接口的字段 if_hintv 中包含有 Hello "间隔时间"（interval），指出连续两次传送 Hello 分组之间的间隔时间长度。ospfhtml 将该数据复制到 Hello 分组的 oh_hintv 字段中。

19.7.2　Hello 分组输出进程

Hello 分组输出进程处理所有接口的定期性 Hello 分组传输。程序代码在文件 ospfhello.c 中。

```
/* ospfhello.c - ospfhello */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <sleep.h>
#include <ospf.h>

#define OSPFDELTA       10   /* OSPF "stagger"bounds (1/10 secs)    */

int ospf_hsend(unsigned);

/*------------------------------------------------------------------------
 * ospfhello - send OSPF hello packets
 *------------------------------------------------------------------------
 */
PROCESS
ospfhello(void)
{
        struct ospf_if *pif;
        int             ifn, rnd;
        /* select initial "random" offset to stagger hello's */
        rnd = nif[NI_PRIMARY]. ni_ip% OSPFDELTA;

        /* do state changes first so we can receive while waiting */
        for (ifn=0; ifn<Net.nif; ++ifn) {
                if (ifn == NI_LOCAL)
                        continue;
                if (nif[ifn].ni_state != NIS_UP)
                        continue;
                pif = &ospf_if[ifn];
                switch (pif->if_type) {
                case IFT_MULTI:
                case IFT_BROADCAST:
                        if (pif->if_prio > 0) {
                                pif->if_twait = pif->if_rdintv;
                                pif->if_state = IFS_WAITING;
                        } else
                                pif->if_state = IFS_DROTHER;
                        break;
                case IFT_PT2PT:
                case IFT_VIRTUAL:
                        pif->if_state = IFS_PT2PT;
                         break;
                default:
                        break;
                }
```

```
        }
        while (1) {
                sleep10(HELLOINTV*10 + rnd);
                if (++rnd == OSPFDELTA)
                        rnd = -OSPFDELTA;
                for (ifn=0; ifn<Net.nif; ++ifn) {
                        if (ifn == NI_LOCAL)
                                continue;
                        if (nif[ifn].ni_state != NIS_UP)
                                continue;
                        ospf_hsend(ifn);
                }
        }
}
```

ospfhello遍历所有的网络接口。对于每个接口，ospfhello检查网络类型并设置接口状态值[①]。ospfhello将非广播网络的状态设置为IFS_PT2PT。对于广播网络，ospfhello利用设置的优先级if_prio，在接口状态IFS_WAITING和IFS_DROTHER之中选取一个状态值。状态值IFS_DROTHER意味着该网关没有资格成为指定路由器，我们将在下一节中介绍。

ospfhello在接口初始化之后，进入无限循环。每次循环都会延迟HELLOINTV秒，然后调用ospf_hsend通过下个接口发送Hello分组。为了防止网关之间的同步，ospfhello为延迟时间增加一个伪随机值rnd，并在下次循环时改变rnd的值。

19.8　指定路由器

虽然OSPF网关沿着邻接路径用洪泛法向其他网关发送链路状态信息，但并不是所有网关之间都有邻接关系。特别是，为了防止在类似以太网的广播网络中产生不必要的通信量，与该网络相连的网关会在它们之中选出一个成员作为指定路由器[②]（designated router）。每个网关都变得与指定路由器有邻接关系，而与其他网关不存在邻接关系。这样，与特定网络相连的N个网关之间仅有N-1个邻接关系[③]。

除了作为邻接关系的聚集点，与网络相连的指定路由器还有另一项任务——它为该网络发送链路状态通告。为了理解发送网络链路状态通告的必要性，考虑图19.4中的拓扑图。图中有几个节点代表网络（例如节点N4）。为了正确地计算最短路径，网关必须接收到图中所有边的链路状态分组，包括从网络到网关的每一条边。由于该网络上的所有网关都与指定路由器有邻接关系，因而由指定路由器接收有关它们与网络的连接的信息。另外，Hello协议允许指定路由器决定其他网关是否可用。因此，某个网络的指定路由器能够发送链路状态更新数据，其中每一项代表图中一条从网络节点到网关节点的边。

① 本地接口的接口状态的概念与前面讨论的邻接状态无关。

② 回忆一下，IP路由器是与IP网关同步的。

③ 实际上，OSPF的概念还包括备用指定路由器，也就是说，N个网关之间可能有2N-3个邻接关系。

19.9 选出一个指定路由器

多重接入网络上的网关必须选出一个指定路由器和一个后备的指定路由器。要做到这一点，每个网关运行选取算法来检查与同一网络相连的相邻网关表。要成为指定路由器，相邻网关必须已建立了2-way通信（即在2-Way或更高的状态中），且必须具有非零的优先级。选取过程可能要求不只一次地检查这一组相邻网关。在一次检查之后，如果没有选出指定路由器，则被选为后备指定路由器的相邻网关成为指定路由器，然后开始第二轮检查，以选出新的后备指定路由器。过程if_elect1完成对相邻网关表的第一轮检查，在此期间它选出一个指定路由器和一个后备指定路由器。

```
/* if_elect1.c - if_elect1 */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*------------------------------------------------------------------------
 * if_elect1 - make one neighbor list pass in the election algorithm
 *------------------------------------------------------------------------
 */
int
if_elect1(struct ospf_if *pif,struct ospf_nb **ppdr,struct ospf_nb **ppdr)
{
        unsigned long       brid, drid;
        Bool                bdec1, cdec1;
        struct ospf_nb     *pnb, *pdr, *pbr;
        int                 i;

        drid = pif->if_drid;
        brid = pif->if_brid;
        bdec1 = FALSE;
        pdr = pdr = 0;
        pnb = &pif->if_nbtab[0];
        for (i=0; i<=MAXNBR; ++i, ++pnb) {
                if (pnb->nb_state < NBS_2WAY)
                        continue;
                if (pnb->nb_rid == pnb->nb_drid) {
                        /* neighbor claims designated routership */

                        if (pdr) {
                                if (pnb->nb_prio <pdr->nb_prio)
                                        continue;
                                if ((pnb->nb_prio == pdr->nb_prio) &&
                                    (net2hl(pnb->nb_rid <
                                     net2hl(pdr->nb_rid)))
```

```
                                continue;
                        }
                        pdr = pnb;
                        continue;
                } /* else pnb is a backup candidate */

                cdecl = pnb->nb_rid == pnb->nb_brid;
                if (bdecl && !cdecl)
                        continue;
                if (bdecl && !bdecl) {
                        bdecl = TRUE;
                        pbr = pnb;
                        continue;
                }
                if (pbr == 0) {
                        pbr = pnb;
                        continue;
                }
                if (pnb->nb_prio < pbr->nb_prio)
                        continue;
                if ((pnb->nb_prio == pbr->nb_prio) &&
                    (net2hl(pnb->nb_rid < net2hl(pbr->nb_rid)))
                        continue;
                pbr = pnb;      /* new backup */
        }
        *ppbr = pbr;
        *ppdr = pdr;
}
```

要被选为指定路由器，相邻网关必须处在2-Way或更高的状态。当if_elect1遍历这组相邻网关时，它检查字段nb_state来判断相邻网关的状态。然后它查看相邻网关是否认为自己是指定路由器。如果是，该相邻网关在字段pnb->nb_rid中的路由器ID，应当和字段pnb->nb_drid中的由相邻网关报告给指定路由器的ID相同。如果该相邻网关认为自己是指定路由器，而if_elect1又有另一个候选的指定路由器，则应当对这两个网关进行比较。If_elect1选取优先级最高的。当多个候选路由器具有相同的优先级时，它选取具有较高路由器ID的那一个。

如果if_elect1发现一个网关没有资格成为指定路由器，则这个网关可考虑作为后备指定路由器。为了成为候选后备指定路由器，该相邻网关必须具有非零的优先级，且当前也不是指定路由器。接下来if_elect1根据优先级在候选的后备指定路由器中选取。在具有相同优先级的相邻网关之中，if_elect1选取具有最高路由器ID的那一个。

如果需要，过程if_elect在一组相邻网关中循环经历一或两遍，以选出指定路由器和后备指定路由器。对于选取结果的种种变化，if_elect还要做出处理。

```
/* if_elect.c - if_elect */

#include <conf.h>
```

```
#include <kernel.h>
#include <network.h>
#include <ospf.h>

int if_elect1(struct ospf_if *, struct ospf_nb **, struct ospf_nb **);
void db_resync(struct ospf_if *), nb_reform(struct ospf_if *);

/*-----------------------------------------------------------------------
 * if_elect - elect a designated router and backup designated router
 *-----------------------------------------------------------------------
 */
void
if_elect (unsigned int ifn)
 {
        struct ospf_if      *pif = &ospf_if[ifn];
        struct ospf_nb      *pnb, *pdr, *pbr;
        unsigned long       odrid, obrid;

        wait(pif->if_nbmutex);
        odrid = pif->if_drid;
        obrid = pif->if_brid;
        if_elect1(pif, &pdr, &pbr);
        pif->if_brid = pbr ? pbr->nb_rid : 0;
        if (pdr)
                pif->if_drid = pdr->nb_rid;
        else {
                pif->if_drid = pif->if_brid;
                pif->if_brid = 0
        }
        /*
         * if designate or backup has changed in this pass and this
         * router is old or new designate or backup, run again to get
         * a backup designate. Also update DB if we were/are designate
         */
        if ((odrid != pif->if_drid &&
            (odrid == pif->if_rid || pif->if_drid == pif->if_rid))) {
                if_elect1(pif, &pdr, &pbr);
                pif->if_brid = pbr ? pbr->nb_rid : 0;
                db_resync(pif);
        }
        if ((odrid != pif->if_brid &&
            (odrid == pif->if_rid || pif->if_brid == pif->if_rid))) {
                if_elect1(pif, &pdr, &pbr);
                pif->if_brid = pbr ? pbr->nb_rid : 0;
        }
        signal(pif->if_nbmutex);
```

```
        if (obrid != pif->if_brid || odrid != pif->if_drid) {
                if (pif->if_drid == pif->if_rid)
                        pif->if_state = IFS_DR;
                else if (pif->if_brid == pif->if_rid)
                        pif->if_state = IFS_BACKUP;
                else
                        pif->if_state = IFS_DROTHER;
                nb_reform(pif);
        }
}
```

在一开始，if_elect记录下旧的指定路由器和后备指定路由器的ID，然后if_elect调用if_elect1。通常if_elect1会选出指定路由器和后备指定路由器。然而如果这组相邻网关都没有资格成为指定路由器，if_elect1 就无法选出指定路由器。在这种情况下，if_elect将后备指定路由器提升到指定路由器的地位。

如果在选取时改变了运行选取算法的网关的状态，选取算法必须再执行一次。特别指出的是，如果一个相邻网关代替该网关变成了指定路由器，则该网关可能有资格成为后备指定网关。因此，第二次调用 if_elect1 时，不再是指定路由器的网关将成为候选的后备指定路由器中的一员。

19.10　变动之后重建邻接关系

网络中的每个网关与指定路由器和后备指定路由器之间都有邻接关系。在选取之前，if_elect保存了当前指定路由器和后备指定路由器的ID。在选取之后，if_elect 对新的和旧的路由器 ID 进行比较，以判断它们是否有所变动。如果的确有变动，if_elect在接口结构中记录下新的指定路由器和后备指定路由器 ID。然后调用 nb_reform 来重建邻接关系。这段程序代码包含在文件 nb_reform.c 中。

```
/* nb_reform.c - nb_reform */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

SYSCALL gettime(unsigned long *);

/*------------------------------------------------------------------------
 * nb_reform - reform adjacencies after a DR or BDR change
 *------------------------------------------------------------------------
 */
int
nb_reform(struct ospf_if *pif)
{
        struct ospf_nb   *pnb = &pif->if_nbtab[1];
        int              nn;
```

```
        for (nn=0; nn<MAXNBR; ++nn, ++pnb) {
                if (phb->nb_state >= NBS_EXSTART) {
                        if (!nb_aok(pif, pnb)) {
                                /* break an adjacency */
                                pnb->nb_state = NBS_2WAY;
                                nb_clearl(pnb);
                        }
                } else if (pnb->nb_state == NBS_2WAY) {
                        if (nb_aok(pif, pnb)) {
                                /* form an adjacency */
                                gettime(&pnb->nb_seq);
                                pnb->nb_state = NBS_EXSTART;
                                nb_makel(pnb);
                                dd_queue(pif, pnb);
                        }
                }
        }
}
```

为了重新计算邻接关系，nb_reform遍历一组相邻网关。每个状态字段为NBS_EXSTART或更高状态的相邻网关，都存在或正在形成一个邻接关系。nb_reform调用nb_aok查看某个相邻网关是否应当有邻接关系存在。如果当前的邻接关系变得无效，则nb_reform将该网关的状态变迁为NBS_2WAY以清除该邻接关系。nb_reform还调用nb_clearl删除正在等待被发往该网关的链路状态分组。如果某个在NBS_2WAY状态下的相邻网关有邻接关系存在，则nb_reform将该网关变迁到NBS_EXSTART状态，并调用nb_makel来建立一张列表，其中的内容是外发的链路状态信息和重传事件。作为邻接关系初始化的最后一个步骤，nb_reform调用dd_queue开始向相邻网关发送拓扑数据库中的信息。下面给出的是nb_clearl和nb_makel的程序代码。在后面的章节中将讨论dd_queue。

```
/* nb_clearl.c - nb_clearl */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*------------------------------------------------------------------------
 * nb_clearl - clear lists of pending messages for a given neighbor
 *------------------------------------------------------------------------
 */
int
nb_clearl(struct ospf_nb *pnb)
{
        struct ep       *pep;

        while (pep = (struct ep *)deq(pnb->nb_lsal))
```

```
                freebuf(pep);
        freeq(pnb->nb_lsal);
        pnb->nb_lsal = EMPTY;

        while (pep = (struct ep *)deq(pnb->nb_dsl))
                freebuf(pep);
        freeq(pnb->nb_dsl);
        pnb->nb_dsl = EMPTY;

        while (pep = (struct ep *)deq(pnb->nb_lsrl))
                freebuf(pep);
        freeq(pnb->nb_lsrl);
        pnb->nb_lsrl = EMPTY;
}

/* nb_makel.c - nb_makerl */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>
#include <q.h>

/*------------------------------------------------------------------------
 * nb_makel - create a new adjacency's retransmission lists
 *------------------------------------------------------------------------
 */
int nb_makel(pnb)
struct ospf_nb *pnb;
{
        pnb->nb_lsal = newq(NBMAXLIST, QF_WAIT);
        pnb->nb_dsl = newq(NBMAXLIST, QF_WAIT);
        pnb->nb_lsrl = newq(NBMAXLIST, QF_WAIT);
}
```

19.11 处理到达的Hello分组

当一个Hello分组到达时，OSPF调用过程ospf_hin来处理。

```
/* ospf_hin.c - ospf_hin */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>
```

```
struct ospf_nb *nb_add(struct ospf_if *, struct ospf *);
int nb_switch(struct ospf_if *, struct ospf_nb *, struct ep *);

/*-----------------------------------------------------------------------
 * ospf_hin - handled input of OSPF HELLO packets
 *-----------------------------------------------------------------------
 */
void
ospf_hin(struct ep *pep)
{
        struct ospf_if      *pif = &ospf_if[pep->ep_ifn];
        struct ip           *pip;
        struct ospf_nb      *pnb, *nb_add( );
        struct ospf         *po;
        struct ospf_hello   *poh;

        pip = (struot ip *)pep->ep_data;
        po = (struct ospf *)((char *)pip + IP_HLEN(pip));
        poh = (struct ospf_hello *)po->ospf_data;

        if (net2hs(poh->oh_hintv) != pif->if_hintv ||
            net2hl(poh->oh_rdintv) != pif->if_rdintv)
                return;
        if (po->ospf_rid == pif->if_rid)
                return;          /* one of our own packets */

        pnb = nb_add(pif, po);
        if (pnb == 0)
                return;          /* neighbor list overflowed */
        pnb->nb_ipa = pip->ip_src;
        if (nb_switch(pif, pnb, pep) == 0)
                return;
        if (poh->oh_prio !=pnb->nb_prio) {
               pif->if_event |= IFE_NCHNG;
               pnb->nb_prio = poh->oh_prio;
        }
        if(poh->oh_drid == pnb_>nb_rid) {    /* Neighbor claims DR    * /
               if (poh->oh_brid == 0 && pif->if_state == IFS_WAITING)
                       pif->if_event |= IFE_BSEEN;
                else if (pnb->nb_drid != pnb->nb_rid)
                        pif->if_event |= IFE_NCHNG;
        } else if (pnb->nb_drid == pnb->nb_rid)
                pif->if_event |= IFE_NCHNG;
        pnb->nb_drid = poh->oh_drid;
        if (poh->oh_brid == pnb->nb_rid) {
                if (pif->if_state == IFS_WAITING)
```

```
                        pif->if_event |= IFE_BSEEN;
                else if (pnb->nb_brid != pnb->nb_rid)
                        pif->if_event |= IFE_NCHNG;
        } else if (pnb->nb_brid == pnb->nb_rid)
                pif->if_event |= IFE_NCHNG;
        pnb->nb_brid = poh->oh_brid;
}
```

如果相邻网关的Hello间隔时间值或路由器关闭间隔时间值与该网关中对应的时间值不同，或者如果接收到的Hello报文最初是由该网关自身生成并发送的，则Hello报文会被丢弃。如果Hello报文有效，ospf_hin调用nb_add将发送者填入相邻网关表中，然后ospf_hin根据协议标准实现状态变迁的语义。

19.12　在相邻网关表中增加一个网关

过程nb_add在相邻网关表中增加一个网关。

```
/* nb_add.c - nb_add */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*------------------------------------------------------------------------
 * nb_add - add a neighbor to our neighbor list (update timer if present)
 *------------------------------------------------------------------------
 */
struct ospf_nb *
nb_add(struct ospf_if  *pif, struct ospf *po)
{
        struct ospf_hello   *poh = (struct ospf_hello *)po->ospf_data;
        struct ospf_nb      *pnb, *pnbfree;
        int                 nn;

        wait(pif->if_nbmutex);
        pnb = &pif->if_nbtab[I];
        pnbfree = 0;
        for (nn=0; nn<MAXNBR;++nn, ++pnb) {
                if (pnbfree == 0 && pnb->nb_state == NBS_DOWN) {
                        phbfree = pnb;
                        continue;
                }
                if(pnb->nb_rid == po->ospf_rid)
                        break;
```

```
        }
        if (nn >= MAXNBR)
                pnb = pnbfree;
        if (pnb) {
                if (pnb->nb_state < NBS_INIT) {
                        pnb->nb_rid = po->ospf_rid;
                        pnb->nb_prio = 0;
                        pnb->nb_drid = 0;
                        pnb->nb_brid = 0;
                        pnb->nb_lsal = pnb->nb_dsl = EMPTY;
                        pnb->nb_lsrl = EMPTY;
                        pnb->nb_state = NBS_INIT;
                        pnb->nb_tlastdd = 0;
                }
                        pnb->nb_lastheard = pif->if_rdintv;
        }
        signal(pif->if_nbmutex);
        return pnb;
}
```

在获得单独使用相邻网关表的权力之后，nb_add顺序搜索相邻网关表。它在变量pnbfree中记录下第一个空闲表项的位置。如果它遇到了将要被加入到表中的项目后，搜索结束。如果搜索的结果是一个新表项，则nb_add将该表项的状态初始化为NBS_INIT，并填写其余字段。

19.13 相邻网关状态的变迁

对于一组事件和相邻网关状态，OSPF标准不但规定了状态变迁，还规定了动作说明。事件包括由本地网关引发的动作和到达的报文引发的动作。过程nb_switch处理几个当Hello报文到达时会发生的动作。

```
/* nb_switch.c - nb_switch */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

SYSCALL gettime(unsigned long *);

/*------------------------------------------------------------------------
 * nb_switch - handle neighbor state changes on HELLO packet input
 *------------------------------------------------------------------------
 */
int
nb_switch(struct ospf_if *pif, struct ospf_nb *pnb, struct ep *pep)
```

```
{
        struct ip           *pip;
        struct ospf         *po;
        struct ospf_hello   *poh;
        Bool                found = FALSE;
        int                 nn, maxn;

        pip = (struct ip *) pep->ep_data;
        po = (struct ospf *)((char *)pip + IP_HLEN(pip));
        poh = (struct ospf_hello *)po->ospf_data;

        maxn = (po->ospf_len - MINHELLOLEN) / sizeof(long);
        for (nn=0; nn<maxn; ++nn)
                if (found = (pif->if_rid == poh->oh_neighbor[nn]))
                        break;
        if (!found) {
                if (pnb->nb_state >= NBS_2WAY) {
                        pnb->nb_state = NBS_INIT;
                        nb_clearl(pnb);
                }
                return 0;
        } else if (pnb->nb_state == NBS_INIT) {
                pnb->nb_state = NBS_2WAY;
                if (nb_aok(pif, pnb)) {
                        gettime(&pnb->nb_seq); /* set initial seq */
                        pnb->nb_state = NBS_EXSTART;
                        nb_makel(pnb);
                        dd_queue(pif, pnb);
                }
        }
        return 1;
}
```

在计算出到达的报文中通告的相邻网关的数目之后，nb_switch遍历该列表。如果接收网关在这一组通告的网关中没有找到自己，则相邻网关不再保持2-way通信。因此，该网关将相邻网关状态变迁为NBS_INIT。如果接收网关在这组通告的网关中发现了自己，且在此之前并没有建立2-way通信，则它进入NBS_2WAY状态，并查看是否应当协商确定邻接关系。

19.14 OSPF定时器事件和重发

OSPF协议利用定时器定期发送Hello报文，设置超时重发，并在没有收到响应的情况下，完成相邻网关状态之间的变迁。程序代码调用过程ospftimer以定期处理定时事件。

```
/* ospftimer.c - ospftimer */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*------------------------------------------------------------------------
 * ospftimer - update neighbor time-out valuse
 *------------------------------------------------------------------------
 */
int
ospftimer(int delay)
{
        struct ospf_if      *pif;
        struct ospf_nb      *pnb;
        int                 ifn, i;

        pif = &ospf_if[0];
        for (ifn=0;ifn<NIF;++ifn, ++pif) {
                switch (pif->if_state) {
                case IFS_DOWN:
                        continue;
                case IFS_WAITING:
                        pif->if_twait -= delay;
                        if (pif->if_twait <=0)
                                if_elect(ifn);
                        break;
                default:
                        break;
                }
                wait(pif->if_nbmutex);
                pnb = &ospf_if[ifn].if_nbtab[1];
                for (i=0; i<MAXNBR; ++i, ++pnb) {
                        if (pnb->nb_state == NBS_DOWN)
                                continue;
                        pnb->nb_lastheard -= delay;
                        if (pnb->nb_lastheard <= 0) {
                              pnb->nb_state = NBS_DOWN;
                              pif->if_event |=IFE_NCHNG;
                }
                if (pnb->nb_state == NBS_EXSTART) {
                        pnb->nb_trexmt -= delay;
                        if (pnb->nb_trexmt <= 0)
                                nb_rexmt(pif, pnb);
                }
```

```
            if (pnb->nb_state >= NBS_FULL &&
                pnb->nb_tlastdd > 0) {
                    pnb->nb_tlastdd -= dealy;
                    if(pnb->nb_tlastdd <0)
                            freebuf(deq(pnb->nb_dsl));
            }
            if (headq(pnb->nb_lsrl)) {
                    pnb->nb_tlsr -=delay;
                    if (pnb->nb_tlsr <= 0)
                            lsr_xmit(pif, pnb);
            }
    }
    signal(pif->if_nbmutex);
    if (pif->if_event & IFE_NCHNG) {
                    if_elect(ifn);
                    pif->if_event &= ~IFE_NCHNG;
            }
        }
    }
```

除了处理超时事件之外，ospftimer 调用 nb_rexmt 在 NBS_EXSTART 状态下重发数据描述报文。重发和分组序列结合起来，保证了两个网关在建立邻接关系时，它们的拓扑数据库保持可靠同步。

```
/* nb_rexmt.c - nb_rexmt */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*------------------------------------------------------------------------
 * nb_rexmt - handle neighbor retransmit timer event
 *------------------------------------------------------------------------
 */
int
nb_rexmt(struct ospf_if *pif, struct ospf_nb *pnb)
{
        if (pnb->nb_state == NBS_EXSTART) {
                dd_xmit(pif, pnb);
                pnb->nb_trexmt = pif->if_rintv;
        }
}
```

19.15　判断是否允许邻接关系

过程 nb_aok 判断一个网关是否应当与一个相邻网关存在邻接关系。

```
/* nb_aok.c - nb_aok */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*------------------------------------------------------------------------
 * nb_aok - determine if adjacency with a neighbor is ok
 *------------------------------------------------------------------------
 */
int
nb_aok(struct ospf_if *pif, struct ospf_nb *pnb)
{
        if (pnb->if_type == IFT_PT2PT ||pif->if_type == IFT_VIRTUAL)
                return TRUE;

        if (pif->if_drid == pif->if_rid || pif->if_brid == pif->if_rid)
                return TRUE;
        if (pif->if_drid == pnb->nb_rid || pif->if_brid == pnb-> nb_rid)
                return TRUE;
        return FALSE;
}
```

一个网关总是与任何通过点到点链路或虚链路连接的相邻网关建立邻接关系。对于多重接入网络，指定路由器或后备的指定路由器与所有相邻网关建立邻接关系，而其余的网关只能与指定路由器或后备指定路由器有邻接关系。程序代码检查各种情况，并返回TRUE表示允许建立邻接关系，或返回FALSE表示拒绝。

19.16 OSPF输入的处理

当携带OSPF分组的IP数据报到达时，IP调用过程ospf_in来处理它。

```
/* ospf_in.c - ospf_in */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ports.h>
#include <ospf.h>

/*------------------------------------------------------------------------
 * ospf_in - deliver an inbound OSPF packet to the OSPF input process
 *------------------------------------------------------------------------
 */
```

```
int
ospf_in(struct netif *pni, struct ep *pep)
{
        /* drop instead of blocking on psend */

        if (pcount(ospf_iport) >= OSPFQLEN) {
                freebuf(pep);
                return SYSERR;
        }
        psend(ospf_iport, (int)pep);
        return OK;
}
```

在我们的系统中，当携带 OSPF 的分组到达时，中断处理程序调用过程 ospf_in。ospf_in 调用 psend 将分组放入端口 ospf_iport 的队列中，OSPF 的输入进程将从该队列中提取分组。由于中断处理程序调用的过程不能被阻塞，所以如果端口已满，ospf_in 绝对不能调用 psend。为了避免阻塞，ospf_in 调用 pcount，并在端口已满时，丢弃该分组。

OSPF 输入进程从端口 ospf_iport 中提取分组，执行过程 ospf。

```
/* ospf.c - ospf */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ports.h>
#include <proc.h>
#include <ospf.h>

/*------------------------------------------------------------------------
 * ospf - start OSPF processes and become OSPF input process
 *------------------------------------------------------------------------
 */
PROCESS ospf(void)
{
        struct ep           *pep;
        struct ip           *pip;
        struct ospf         *po;
        struct ospf_if      pif;

        if (!gateway)
                return SYSERR;
        ospfinit( );
        ospf_iport = poreate(OSPFQLEN);
        if (ospf_iport == SYSERR)
                panic("ospf: cannot open ospf port");
        resume(create(ospfhello, OSPFHSTK, OSPFHPRI, OSPFHNAM, OSPFHARGC));
```

```
        while (TRUE) {
                pep = (struct ep *)preceive(ospf_iport);
                if (pep == (struct ep *)SYSERR)
                        continue;
                pip = (struct ip *)pep->ep_data;
                po = (struct ospf *) ((char *)pip + IP_HLEN(pip));
                if (ospfcheck(pep) != OK)
                        continue;
                ospfnet2h(po);
                pep->ep_order |= EPO_OSPF;
                switch (po->ospf_type) {
                case T_HELLO:           ospf_hin(pep);
                                        break;
                case T_DATADESC:        ospf_ddin(pep);
                                        break;
                case T_LSREQ:           lsr_in(pep);
                                        break;
                case T_LSUPDATE;        lsu_in(pep);
                                        break;
                case T_LSACK:           lsack_in(pep);
                                        break;
                default:
                        break;
                }
                pif = &ospf_if[pep->ep_ifn];
                if ((pif->if_state == IFS_WAITING &&
                    (pif->if_event&IFE_BSEEN)) ||
                    (pif->if_state >= IFS_DROTHER &&
                    (pif->if_event &IFE_NCHNG))) {
                        if_elect(pep->ep_ifn);
                        pif->if_event &= ~(IFE_BSEEN|IFE_NCHNG);
                }
                freebuf(pep);
        }
        return OK;      /* not reached */
}

Bool            doospf = FALSE;
int             ospfpid = BADPID;
int             ospf_iport;
```

ospf作为一个独立进程运行。它一开始就调用ospfinit来初始化所有的数据结构，并调用pcreate为传入分组创建一个同步队列。ospf还要创建一个生成Hello分组的进程，然后它进入一个无限循环。

在每次经过主循环时，ospf提取一个传入的OSPF报文，并调用ospfnet2h将它转变为本地字节

顺序，并且调用 ospfcheck 以验证分组有效。然后，ospf 检查字段 ospf_type 判断分组的类型，并调用适当的输入例程。最后，ospf 检查传入分组经过的接口，以决定该报文的到达是否会触发对一个新指定路由器的选取。

19.17　链路状态处理中的说明和过程

在文件 ospf_ls.h 中包含与链路状态报文有关的分组内容、结构和常量的说明。

```
/* ospf_ls.h */

/* OSPF Link State Request Packet */

struct  ospf_lsr {
        unsigned long  lsr_type;       /* Link State Type              */
        unsigned long  lsr_lsid;       /* Link State Identifier        */
        unsigned long  lsr_rid;        /* Advertising Router           */
};

#define LSRLEN        12

/* OSPF Link State Summary */

struct  ospf_lss {
        unsigned short lss_age;        /* Time (secs) Since Originated*/
        unsigned char  lss_opts;       /* Options Supported           */
        unsigned char  lss_type;       /* LST_* below                 */
        unsigned long  lss_lsid;       /* Link State Identifier       */
        unsigned long  lss_rid;        /* Advertising Router Identifier*/
        unsigned long  lss_seq;        /* Link State Adv. Sequence #  */
        unsigned short lss_cksum;      /* Fletcher Cheksum of LSA     */
        unsigned short lss_len;        /* Length of Advertisement     */
};

#define LSSHDRLEN  20

/* Link State Advertisement Types */

#define LST_RLINK      1               /* Router Link                 */
#define LST_NLINK      2               /* Network Link                */
#define LST_SLINK      3               /* IP Network Summary Link     */
#define LST_BRSLINK    4               /* AS Border Router Summary    */
#define LST_EXTERN     5               /* AS External Link            */

/* Link State Advertisement (min) Lengths */

#define LSA_RLEN  (LSSHDRLEN + 4)
```

```
#define LSA_NLEN  (LSSHDRLEN + 4)

#define LSA_ISEQ  0x80000001

/* LSS Type of Service Entry */

struct  tosent {
        unsigned char  tos_tos;        /* IP Type of Service          */
        unsigned char  tos_mbz;        /* Must Be Zero                */
        unsigned short tos_metric;     /* Metric for This TOS         */
};

/* OSPF Link State Advertisement */

#define MAXLSDLEN      64              /* Max LS Data Len (configurable)*/

struct ospf_lsa {
        struct ospf_lss lsa_lss;       /* Link State Adv. Header      */
        char            lsa_data[MAXLSDLEN];/* Link-Type Dependent Data*/
};

/* Convenient Field Translations */

#define lsa_age        lsa_lss.lss_age
#define lsa_opts       lsa_lss.lss_opts
#define lsa_type       lsa_lss.lss_type
#define lsa_lsid       lsa_lss.lss_lsid
#define lsa_rid        lsa_lss.lss_rid
#define lsa_seq        lsa_lss.lss_seq
#define lsa_cksum      lsa_lss.lss_cksum
#define lsa_len        lsa_lss.lss_len

/* Router Links Advertisement */

struct  ospf_ra {
        unsigned char  ra_opts;        /* RAO_* Below                 */
        unsigned char  ra_mbz;         /* Must Be Zero                */
        unsigned short ra_nlinks;      /* # of Links This Advertisement */
        unsigned char  ra_data[1];     /* nlinks rlink structs        */
};

struct ospf_rl{
        unsigned long  rl_lid;         /* Link ID                     */
        unsigned long  rl_data;        /* Link Data                   */
        unsigned char  rl_type;        /* Link Type (RAT_* Below)     */
        unsigned char  rl_ntos;        /* # of Types-of-Service Entries */
        unsigned char  rl_metric;      /* TOS 0 Metric                */
        unsigned long  rl_tosent[1];   /* TOS Entries ra_ntos Times   */
```

```
};

#define MINRLLEN        12

#define RAO_ABR         0x01            /* Router is Area Border Router */
#define RAO_EXTERN      0x02            /* Router is AS Boundary Router */

#define RAT_PT2PT       1               /* Point-Point Connection       */
#define RAT_TRANSIT     2               /* Connection to Transit Network*/
#define RAT_STUB        3               /* Connection to Stub Network   */
#define RAT_VIRTUAL     4               /* Virtual Link                 */

/* Network Links Advertisement */

struct  ospf_na {
        IPaddr  na_mask;                /* Network Mask                 */
        unsigned long  na_rid[1];       /* IDs of All Attached Routers  */
};

/* Link State Update Packet Format */

struct  ospf_lsu {
        unsigned long lsu_nads;         /* # Advertisments This Packet  */
        char          lsu_data[1];      /* 1 or mcre struct ospf_lsa's  * /
};

#define MINLSULEN  (MINHDRLEN + 4) /* Base LSU Length   */

void lss_build(struct ospf_if *, struct ospf_nb *, struct ep *);
int lsr_xmit(struct ospf_if *, struct ospf_nb *);
void lsa_add(struct ospf_if *, struct ep *, struct ospf_db *);
struct ep *ospflstmpl(struct ospf_if *);
struct ep *lsr_add(struct ospf_if *, struct ospf_lss *, struct ep *);
int lsr_queue(struct ospf_if *, struct ospf_nb *, struct ep *);
int lsa_send(struct ospf_if *, IPaddr, struct ep *);
int lsr_check(struct ospf_nb *);
```

19.18　数据库描述分组的生成

当一个网关第一次与某个相邻网关初始化了一个邻接关系时，它将相邻网关转到 NBS_EXSTART 状态，并开始交换数据库描述分组。在相邻网关进入状态 NBS_EXCHNG 之后，这种分组的交换继续进行。

一旦 OSPF 软件需要生成一个数据库描述分组，它调用过程 dd_queue。

```
/* dd_queue.c - dd_queue */

#include <conf.h>
```

```
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*-----------------------------------------------------------------------
 * dd_queue - generate Date Description packests
 *-----------------------------------------------------------------------
 */
void
dd_queue(struct ospf_if *pif, struct ospf_nb *pnb)
{
        struct ep       *pep;
        struct ip       *pip;
        struct ospf     *po;
        struct ospf_dd  *pdd;

        pep = ospfddtmpl(pif);
        if (pep == 0)
                return;
        pip = (struct ip *)pep->ep_data;
        po = (struct ospf *)pip->ip_data;
        pdd = (struct ospf_dd *)po->ospf_data;

        if (pnb->nb_state == NBS_EXSTART) {
                pdd->dd_control = DDC_INIT | DDC_MORE | DDC_MSTR;
                pdd->dd_seq = h12net(pnb->nb_seq);
                if (enq(pnb->nb-dsl, pep, 0) < 0)
                        freebuf(pep);
                dd_xmit(pif, pnb);
                pnb->nb_trexmt = pif->if_rintv;
                return;
        }
        /* else we're in EXCHANGE state */
        lss_build(pif, pnb, pep);
        dd_xmit(pif, pnb);
        if (pnb->nb_master)
                pnb->nb_trexmt = pif->if_rintv;
}
```

dd_queue调用ospfddtmpl为分组分配缓冲区和填写首部中的字段。然后它检查相邻网关记录中nb_state字段，并判断相邻网关的状态。如果该相邻网关的状态是NBS_EXSTART，dd_queue创建一个初始分组并将其放入队列以待传输。将初始分组的dd_control字段中的DDC_MORE、DDC_MSTR和DDC_INIT比特置为1，用以通知相邻网关分组是初始的数据库描述分组，以引起相邻网关协商决定哪一个网关可作为主网关。dd_queue调用enq将初始分组放入数据库概要列表，然后调用dd_xmit发送该分组。最后，在返回调用者之前，dd_queue设置字段nb_trexmt为重传事件定时。

在相邻网关确认了该初始分组后，网关将相邻网关的状态转为NBS_EXCHNG并记录下两者之中哪一个是主网关。对于在状态NBS_EXCHNG下的相邻网关，dd_queue创建并发送数据库概要列表。dd_queue 调用 lss_build 建立它的拓扑数据库概要列表并将它放入队列以待发送，然后调用 dd_xmit 来发送该表。如果发送方是主网关，则在返回调用者前，调用 dd_queue 安排一次重传。

19.19　创建一个模板

当OSPF需要创建数据库描述分组，它调用ospfddtmpl分配缓冲区并填写首部中的字段。

```
/* ospfddtmpl.c - ospfddtmpl */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*------------------------------------------------------------------------
 * ospfddtmpl - fill in OSPF Data Description packet template
 *------------------------------------------------------------------------
 */
struct ep *
ospfddtmpl (struct ospf_if *pif)
{
        struct ep       *pep;
        struct ip       *pip;
        struct ospf     *po;
        struct ospf_dd  *pdd;

        pep = (struct ep *)getbuf(Net.netpool);
        if ((int)pep == SYSERR)
                return 0;
        pip = (struct ip *)pep->ep_data;
        po = (struct ospf *)pip->ip_data;
        pdd = (struct ospf_dd *)po->ospf_data;
        po->ospf_version = OSPF_VERSION;
        po->ospf_type = T_DATADESC;
        po->ospf_len = MINDDLEN;
        po->ospf_rid = pif->if_rid;
        po->ospf_aid = pif->if_area->ar_id;
        po->ospf_authtype = pif->if_area->ar_authtype;
        memcpy(po->ospf_auth, pif->if_auth, AUTHLEN);
        pdd->dd_mbz = 0;
        pdd->dd_opts = pif->if_opts;
        pdd->dd_control = 0;
        return pep;
}
```

ospfddtmpl以一个指向特定接口结构的指针为入口参数。它先分配一个缓冲区，然后完成填写首部字段的任务。内容为常量的字段可用符号变量来设置（例如OSPF版本及分组类型）。其他字段的信息必须从接口结构中提取（例如发送路由器的ID及区域号）。一旦ospfddtmpl填完了首部之后，它返回指向该分组的指针。

19.20 传送数据库描述分组

当两个网关交换各自拓扑数据库中的信息时，会发送数据库描述分组及确认，这由过程dd_xmit来完成。它发送来自数据库概要列表的数据库描述分组，或在该表为空时创建一个确认。

```
/* dd_xmit.c - dd_xmit */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*------------------------------------------------------------------------
 * dd_xmit - transmit pending Database Description packets
 *------------------------------------------------------------------------
 */
void
dd_xmit(struct ospf_if *pif, struct ospf_nb *pnb)
{
        struct ep       *pephead, *pep;
        struct ip       *pip;
        struct ospf     *po;
        struct ospf_dd  *pdd;
        int             len;

        if(pephead = (struct ep *)headq(pnb->nb_dsl)) {
                pep = (struct ep *)getbuf(Net.netpool);
                if ((int)pep == SYSERR)
                        return;
                /* make a copy */
                pip = (struct ip *)pephead->ep_data;
                po = (struct ospf *)pip->ip_data;
                len = EP_HLEN + IPMHLEN + po->ospf_len;
                memcpy(pep, pephead, len);
                pip = (struct ip *)pep->ep_data;
                po = (struct ospf *)pip->ip_data;
                pdd = (struct ospf_dd *)po->ospf_data;
        } else {
                /* no DD's to send;create an ACK-only */
                pep = ospfddtmpl(pif);
```

```
        pip = (struct ip *)pep->ep_data;
        po = (struct ospf *)pip->ip_data;
        pdd = (struct ospf_dd *)po->ospf_data;
        if (pnb->nb_master)
                pdd->dd_control = DDC_MSTR;
        else
                pdd->dd_control = 0;
    }
    pdd->dd_seq = hl2net(pnb->nb_seq);
    po->ospf_authtype = net2hs(pif->if_area->ar_authtype);
    len = po->ospf_len;
    po->ospf_len = net2hs(po->ospf_len);
    pep->ep_order &= ~EPO_OSPF;
    memset(po->ospf_auth, 0, AUTHLEN);
    po->ospf_cksum = 0;
    po->ospf_cksum = cksum((WORD *)po, len);
    memcpy(po->ospf_auth, pif->if_area->ar_auth, AUTHLEN);
    pip->ip_src = ip_anyaddr;
    ipsend(pnb->nb_ipa, pep, len, IPT_OSPF, IPP_INCTL,
            IP_TTL);
}
```

dd_xmit先通过调用headq来检查数据库概要列表的第一项，即字段nb_dsl。如果该表中有一个表项，headq返回指向该表项的指针，否则返回0。如果调用headq时找到了一个项目，dd_xmit分配一个缓冲区用于保存一个外发分组，并将表中第一个项目的信息复制到分组中。如果这时发现表为空，dd_xmit调用ospfddtmpl创建仅包含一个确认的分组。

在创建一个分组之后，dd_xmit在首部字段dd_seq中填写序号，并在字段ospf_authtype中填写鉴别类型。OSPF在校验和中不包括鉴别信息。因此在计算校验和之前，dd_xmit在校验和字段和鉴别字段中填充0。然后dd_xmit再将计算得到的校验和及鉴别信息移入首部中，并调用ipsend来发送最终得到的数据报。

19.21 处理到达的数据库描述分组

当两个网关建立了邻接关系之后，它们就开始交换拓扑数据库中的信息。具有较大路由器ID的网关变为主网关，并询问从网关以得到数据库中的信息。为了处理输入的数据库描述分组，OSPF调用过程ospf_ddin。

```
/* ospf_ddin.c - ospf_ddin */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>
```

```
/*-----------------------------------------------------------------------
 * ospf_ddin - handled input of OSPF Data Description packets
 *-----------------------------------------------------------------------
 */
void
ospf_ddin (struct ep *pep)
{
        struct ospf_if  *pif = &ospf_if[pep->ep_ifn];
        struct ospf_nb  *pnb;
        struct ip       *pip;
        struct ospf     *po;
        struct ospf_dd  *pdd;

        pip = (struct ip *)pep->ep_data;
        po = (struct ospf *)((char *)pip + IP_HLEN(pip));

        pdd = (struct ospf_dd *)po->ospf_data;
        pdd->dd_seq = net2hl(pad->dd_seq);

        pnb = nb_add(pif, po);
        if (pnb == 0)
                return 0;
        switch (pnb->nb_state) {
        case NBS_INIT:
                pnb->nb_state = NBS_2WAY;
                if (nb_aok(pif, pnb)) {
                        pnb->nb_seq++;
                        pnb->nb_state = NBS_EXSTART;
                        nb_makel(pnb);
                        dd_queue(pif, pnb);
                }
                break;
        case NBS_2WAY:
                return;
        case NBS_EXSTART:
                ddi_exstart(pif, pnb, pep);
                break;
        case NBS_EXCHNG:
                ddi_exchng(pif, pnb, pep);
                break;
        case NBS_LOADING:
        case NBS_FULL:
                ddi_full(pif, pnb, pep);
                break;
        }
}
```

ospf_ddin利用相邻网关的状态来决定如何处理。对处在低于NBS_2WAY状态的相邻网关来说，它必须在初始化邻接关系之前判断是否允许这种邻接关系。对处于比NBS_2WAY状态要高的网关，ospf_ddin根据网关状态决交是否需要调用过程ddi_exstart、ddi_exchng或ddi_full来处理已经到达的分组。

19.21.1　处理 EXSTART 状态下的分组

过程 ddi_exstart 处理来自当前处于 NBS_EXSTART 状态的相邻网关的分组。

```
/* ddi_exstart.c - ddi_exstart */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*--- ------------------------------------------------------------------
 * ddi_exstart - handle Data Descrip. input for EXSTART state neighbors
 *-----------------------------------------------------------------------
 */
void
ddi_exstart(struct ospf_if *pif, struct ospf_nb *pnb, struct ep *pep)
{
        struct ip       *pip = (struct ip *)pep->ep_data;
        struct ospf     *po = (struct ospf *)((char *)pip + IP_HLEN(pip));
        struct ospf_dd  *pdd = (struct ospf_dd *)po->ospf_data;
        struct ep       *peptmp;
        unsigned int    cbits = DDC_INIT | DDC_MORE | DDC_MSTR;

        if (((pdd->dd_control & cbits) == cbits) &&
            po->ospf_len == MINDDLEN &&
            net2hl(pnb->nb_rid > net2hl(pif->if-rid)) {
                pnb->nb_master = FALSE;
                pnb->nb_seq = pdd->dd_seq;
                pnb->nb_opts = pdd->dd_opts;
                pnb->nb_state = NBS_EXCHNG;
        } else if ((pdd->dd_control&(DDC_INIT|DDC_MSTR)) == 0)&&
            pdd->dd_seq == pnb->nb_seq &&
            net2hl(pnb->nb_rid) <net2hl(pif->if_rid)) {
                pnb->nb_master = TRUE;
                pnb->nb_opts = pdd->dd_opts;
                pnb->nb_state = NBS_EXCHNG;
        } else
                return;
        if (peptmp = (struct ep *)deq(pnb->nb_dsl))
                freebuf(peptmp);
```

```
        if (pnb->nb_master)
                pnb->nb_seq++;
        lsr_queue(pif, pnb, pep);
        dd_queue(pif, pnb);
}
```

ddi_exstart不仅检查传入报文中的控制比特，还要检查报文的长度。如果相邻网关发送了最短的分组，且相邻网关ID大于接收网关的ID，接收方就将相邻网关转到NBS_EXCHNG状态并声明相邻网关为主网关。否则，如果传入数据库描述分组的序号与初始序号相符，ddi_exstart就接受该分组，并将相邻网关转到NBS_EXCHNG状态，且声明自己是主网关。如果相邻网关改变了状态，ddi_exstart调用dd_queue将分组放入队列，以待传输。

19.21.2 处理EXCHNG状态下的分组

ospf_ddin调用过程ddi_exchng来处理来自处于NBS_EXCHNG状态下的相邻网关的数据库描述分组。

```
/* ddi_exchng.c - ddi_exchng */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*------------------------------------------------------------------------
 * ddi_exchng - handle Data Descrip. input for EXCHANGE state neighbors
 *------------------------------------------------------------------------
 */
int
ddi_exchng(struct ospf_if *pif, struct ospf_nb *pnb, struct ep *pep)
{
        struct ip       *pip = (struct ip *)pep->ep_data;
        struct ospf     *po = (struct ospf *)((char *)pip + IP_HLEN(pip));
        struct ospf_dd  *pdd = (struct ospf_dd *)po->ospf_data;
        struct ep       *peptmp;

        if (((pdd->dd_control & DDC_MSTR) & pnb->nb_master) ||
            (!(pdd->dd_control & DDC_MSTR) & !pnb->nb_master) ||
            (pdd->dd_control & DDC_INIT) ||
            (pdd->dd_opts !=pnb->nb_opts))
                return nb_mismatch(pif, pnb);
        if ((pnb->nb_master && pdd->dd_seq == pnb->nb_seq-1) ||
            (!pnb->nb_master && pdd->dd_seq == pnb->nb_seq))
                return 0; /* duplicate */
        if ((pnb->nb_master && pdd->dd_seq != pnb->nb_seq+1) ||
            (pnb->nb_master && pdd->dd_seq != pnb->nb_seq))
```

```
        return nb_mismatch(pif, pnb);
if (!pnb->nb_master && lenq(pnb->nb_dsl) == 1) {
        /* slave must save the last packet DEADINTV secs */
        if (pnb->nb_tlastdd == 0)
                pnb->nb_tlastdd = DEADINTV;
} else if (peptmp = (struct ep *)deq(pnb->nb_dsl))
        freebuf(peptmp);        /* has been acked */
if (pnb->nb_master
        pnb->nb_seq++;
else
        pnb->nb_seq = pdd->dd_seq;
lsr_queue(pif, pnb, pep);
if ((pdd->dd_control & DDC_MORE) == 0 &&
    lenq(pnb->nb_dsl) <=1) {
                if (headq(pnb->nb_lsrl))
                        pnb->nb_state = NBS_LOADING;
                else
                        pnb->nb_state = NBS_FULL;
                if (!phb->nb_master)
                        dd_xmit(pif, pnb);
                return 1;
        }
        dd_xmit(pif, pnb);
        return ok;
}
```

根据协议标准规定，ddi_exchng检查传入分组的序号。主网关接受序号与主网关发送的最后一个分组的序号相同的分组，从网关接受比从网关发送的最后一个分组的序号大1的分组。具有较低序号的分组是复制分组，主网关丢弃复制分组，但从网关会做出响应，它会重发刚才发出的最后一个分组。主网关接受分组后，会将字段nb_seq中的序号计数器加1，从网关使序号等于接受的分组的序号。

如果传入分组的主网关的状态说明与本地数据结构中记录的不一致，或者二者的序号不一致，ddi_exchng调用nb_mismatch从头开始重新建立邻接关系。

19.21.3 处理FULL状态下的分组

当ospf_ddin接收到来自当前处于NBS_FULL状态的相邻网关的数据库描述分组后，它调用过程ddi_full来处理该分组。

```
/* ddi_full.c - ddi_full */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>
```

```
/*------------------------------------------------------------------------
 * ddi_full - handle DD packet input for FULL and LOADING state neighbors
 *------------------------------------------------------------------------
 */
int ddi_full(pif, pnb, pep)
struct ospf_if   *pif;
struct ospf_nb   *pnb;
struct ep        *pep;
{
        struct ip        *pip = (struct ip *)pep->ep_data;
        struct ospf      *po = (struct ospf *)((char *)pip + IP_HLEN(pip));
        struct ospf_dd   *pdd = (struct ospf_dd *)po->ospf_data;

        if (((pdd->dd_control & DDC_MSTR) & pnb->nb_master) ||
            (!(pdd->dd_control & DDC_MSTR) & !pnb->nb_master) ||
            (pdd->dd_control & DDC_INIT) ||
            (pdd->dd_opts !=pnb->nb_opts))
                return nb_mismatch(pif, pnb);
        if ((pnb->nb_master && pdd->dd_seq == pnb->nb_seq-1)
                return 0;                       /* duplicate;master discards */
        if (!pnb->nb_master && pdd->dd_seq == pnb->nb_seq)) {
                dd_xmit(pif, pnb);              /* duplicate; slave responds * /
                return 1;
        }
        /* else, NOT a duplicate;Sequence mismatch */
        return nb_mismatch(pif, pnb);
}
```

当分组到达时，ddi_full将分组中主网关控制比特与本地数据结构中的字段nb_master相比较。如果相邻网关认为主网关状态与接收方记录的状态不符，ddi_full调用nb_mismatch从头开始建立邻接关系。主网关丢弃到达的复制分组，而从网关则发送一个确认报文。

19.22 处理链路状态请求分组

在建立邻接关系时，一对相邻网关互相交换各自掌握的链路的概要信息。如果到达的概要信息所指的链路接收方不知道，或者是因为发送方有更近期的信息，接收网关生成链路状态请求报文，要求对方网关发送链路状态描述分组。当一个链路状态请求报文到达时，ospf调用过程lsr_in进行处理。

```
/* lsr_in.c - lsr_in */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>
```

```
/*-----------------------------------------------------------------------
 * lsr_in - handle a received link state request packet
 *-----------------------------------------------------------------------
 */
void
lsr_in(pep)
struct ep  *pep;
{
        struct ip       *pipout, *pip = (struct ip *)pep->ep_data;
        struct ospf     *poout, *po;
        struct ospf_if  *pif = &ospf_if[pep->ep_ifn];
        struct ospf_nb  *pnb;
        struct ospf_lsr *plsr;
        struct ospf_lsu *plsu;
        struct ospf_db  *pdb;
        struct ep       *pepout, *ospflstmpl(struct ospf_if *);
        unsigned        i, nlsr, maxlsapp;

        if (pif->if_state <= IFS_WAITING)
                return;
        pnb = &pif->if_nbtab[1];
        po = (struct ospf *)((char *)pip+IP_HLEN(pip));
        for (i=0; i<MAXNBR; ++i, ++pnb)
                if (pnb->nb_rid == po->ospf_rid)
                        break;
        if (i == MAXNBR || pnb->nb_state < NBS_EXCHNG)
                return;
        maxlsapp = (nif[pep->ep_ifn].ni_mtu - IPMHLEN - MINLSULEN);
        maxlsapp /= (LSSHDRLEN + MAXLSDLEN);
        pepout = ospflstmpl(pif);
        if (pepout == 0)
                return;
        pipout = (struct ip *)pepout->ep_data;
        poout = (struct ospf *)pipout->ip_data;
        plsu = (struct ospf_lsu *)poout->ospf_data;

        nlsr = (po->ospf_len - MINHDRLEN) / sizeof (struct ospf_lsr);
        plsr = (struct ospf_lsr *)po->ospf_data;
        for (i=0; i<nlsr;++i, ++plsr) {
                pdb = db_lookup(pif->if_area, plsr->lsr_type,
                        plsr->lsr_lsid);
                if (pdb == 0) {
                        freebuf(pepout);
                        nb_mismatch(pif, pnb);
```

```
                        return;
                }
                if (plsu->lsu_nads >= maxlsapp) {
                        lsa_send(pif, pip->ip_src, pepout);
                        if (!(pepout = ospflstmpl(pif)))
                                return;
                        pipout = (struct ip *)pepout->ep_data;
                        poout = (struct ospf *)pipout->ip_data;
                        plsu = (struct ospf_lsu *)poout->ospf_data;
                }
                lsa_add(pif, pepout, pdb);
        }
        lsa_send(pif, pip->ip_src, pepout);
}
```

lsr_in 首先查看 OSPF 是否在等待从该传入报文经过的接口上接收一个链路状态请求，然后搜索相邻网关找出发送该报文的相邻网关。lsr_in 检查相邻网关的状态以确保它接受的链路请求报文只来自与它有邻接关系的相邻网关的。若发现相邻发送网关的状态有效，则 lsr_in 遍历链路状态请求报文中的项目，并从拓扑数据库中找出对应的链路状态信息，添加到作为响应发送回相邻网关的链路状态报文中。

19.23 建立链路状态概要信息

过程 lss_build 建立链路状态概要信息，并将它插入相邻网关的数据库概要列表中。

```
/* lss_build.c - lss_build */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

#define LSS_ABORT(par, pnb, pep) { \
                signal(par->ar_dbmutex);        \
                if (pep)                        \
                        freebuf(pep);           \
                pnb->nb_state = NBS_2WAY;       \
                nb_clearl(pnb); }

/*------------------------------------------------------------------------
 * lss_build - build link-state summaries for DD packets
 *------------------------------------------------------------------------
 */
void
lss_build(pif, pnb, pep)
```

```
struct ospf_if  *pif;
struct ospf_nb    *pnb;
struct ep  *pep;
{
        struct ospf_ar *par = pif->if_area;
        struct ip      *pip = (struct ip *)pep->ep_data;
        struct ospf    *po = (struct ospf *)pip->ip_data;
        struct ospf_dd *pdd = (struct ospf_dd *)po->ospf_data;
        struct ospf_db *pdb;
        struct ospf_lss *plss;
        int            nlsr, ifn, maxlspp;

        ifn = pif - &ospf_if[0];
        maxlspp = (nif[ifn].ni_mtu - IPMHLEN - MINDDLEN) / LSSHDRLEN;
        wait(par->ar_dbmutex);
        nls = 0;
        for (pdb = par->ar_ablhead; pdb; pdb = pdb->db_lnext) {
                if (nls >= maxlspp) {
                        pdd->dd_control = DDC_MORE;
                        if (pnb->nb_master)
                                pdd->dd_control |= DDC_MSTR;
                        if (enq(pnb->nb_dsl, pep, 0) < 0) {
                                LSS_ABORT(par, pnb, pep);
                                return;
                        }
                        pep = ospfddtmpl(pif);
                        if (pep == 0) {
                                LSS_ABORT(par, pnb, pep);
                                return;
                        }
                        pip = (struct ip *)pep->ep_data;
                        po = (struct ospf *)pip->ip_data;
                        pdd = (struct ospf_dd *)po->ospf_data;
                        nls = 0;
                }
                plss = &pdd->dd_lss[nls];
                memcpy(plss, &pdb->db_lsa.lsa_lss, LSSHDRLEN);
                plss->lss_age = hs2net(plss->lss_age);
                plss->lss_seq = hl2net(plss->lss_seq);
                plss->lss_len = hs2net(plss->lss_len);
                po->ospf_len += LSSHDRLEN;
                ++nls;
        }
        pdd->dd_control = 0;
        if (pnb->nb_master)
```

```
                pdd->dd_control |= DDC_MSTR;
        if (enq(pnb->nb_dsl, pep, 0) < 0) {
                LSS_ABORT(par, pnb, pep);
                return;
        }
        signal (par->ar_dbmutex);
}
```

19.24 OSPF实用过程

在接收到一个分组后，OSPF将首部的整型字段转换为网络字节顺序。过程ospfnet2h完成这项任务，很简单。

```
/* ospfnet2h.c - ospfnet2h */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ospf.h>

/*------------------------------------------------------------------------
 * ospfnet2h - convert OSPF header from network to host byte order
 *------------------------------------------------------------------------
 */
struct ospf *
ospfnet2h(po)
struct ospf      *po;
{

        /* NOTE: only includes fields in the main header */

        po->ospf_len = net2hs(po->ospf_len);
        po->ospf_cksum = net2hs(po->ospf_cksum);
        po->ospf_authtype = net2hs(po->ospf_authtype);
}
```

在将传入分组首部整数字段转换为本地字节顺序后，OSPF对其值进行检查，并决定该分组是否有效（例如，可检查其校检和是否正确）。过程ospfcheck用于其细节的处理。

```
/* ospfcheck.c - ospfcheck */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <ports.h>
#include <ospf.h>
```

```
/*------------------------------------------------------------------------
 * ospfcheck - check if a packet is a valid OSPF packet
 *------------------------------------------------------------------------
 */
int
ospfcheck(struct ep *pep)
{
        struct ip       *pip = (struct ip *)pep->ep_data;
        struct ospf     *po = (struct ospf *)((char *)pip + IP_HLEN(pip));
        struct ospf_if  *pif = &ospf_if[pep->ep_ifn];
        if (pif->if_state == IFS_DOWN)
                return FALSE;
        if (po->ospf_version != OSPF_VERSION)
                return FALSE;
        if (net2hs(po->ospf_authtype) != pif->if_area->ar_authtype)
                return FALSE;
        if (pif->if_area->ar_authtype &&
            memcmp(po->ospf_auth, pif->if_area->ar_auth, AUTHLEN)
                return FALSE;
        memset(po->ospf_auth, 0, AUTHLEN);
        if (cksum(WORD*)(po, net2hs(po->ospf_len)))
                return FALSE;
        return TRUE;
}
```

ospfcheck首先检查接口状态。如果系统管理员将该接口设置为IFS_DOWN状态，OSPF无法使用该接口，且不能接受分组。如果OSPF版本号与软件版本号不符，或到达的分组鉴别类型与设置不符，则ospfcheck拒绝接受分组。此外，如果鉴别字段中携带的口令与该区域使用的口令不符，OSPF也要拒绝接受该分组。

在检查过鉴别字段之后，ospfcheck验证校验和。由于校验和不包括鉴别字段，因此在计算校验和之前，ospfcheck用0来填入鉴别字段。就像其他TCP/IP协议中一样，ospfcheck在计算时包括了校验和字段，并且如果计算结果不为0，就拒绝该分组。ospfcheck产生的结果是一个布尔值。如果其中任何一项检测失败，则ospfcheck立即返回FALSE，对通过所有检测的分组，它返回TRUE。

另一个实用程序处理两个相邻网关在交换信息时发生状态或序号出错时的情况。主网关发送一个序号，从网关利用该序号确认已接收到的分组。任何时候都仅允许一个分组未被确认，因此不存在分组重新排序的问题。如果到达的分组具有错误的序号，则OSPF拆除邻接关系，使相邻网关返回EXSTART状态，且重新交换信息[1]。过程nb_mismatch处理其细节问题。

```
/* nb_mismatch.c - nb_mismatch */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```

① 当一个序号过小的复制分组达到时会发生惟一的例外。

```
#include <ospf.h>

/*-----------------------------------------------------------------------
 * nb_mismatch - handle sequence # mismatch event
 *-----------------------------------------------------------------------
 */
int nb_mismatch(pif, pnb)
struct ospf_if *pif;
struct ospf_nb *pnb;
{
        struct ep       *pep;

        pnb->nb_state = NBS_EXSTART;

        /* empty lists */
        while (pep = (struct ep *)deq(pnb->nb_lsal))
                freebuf(pep);
        while (pep = (struct ep *)deq(pnb->nb_dsl))
                freebuf(pep);
        while (pep = (struct ep *)deq(pnb->nb_lsrl))
                freebuf(pep);
        pnb->nb_seq++;
        dd_queue(pif, pnb);
        return 0;
}
```

程序代码比较简单。nb_mismatch 首先使相邻网关的状态降为 NBS_EXSTART。然后它清除三个与正在等待处理的请求有关的列表。nb_dsl 中包含的是数据库概要信息，nb_lsal 中的内容是链路状态描述，而nb_lsrl 中包含的是链路状态请求。在清除了这三个列表之后，nb_mismatch 递增序号，并调用 dd_queue 重新开始发送新的数据库描述分组。

19.25 小结

OSPF 利用链路状态算法传播路由信息。每个参与网关都保留一张底层互联网的有向图模型，其中每个节点对应于一个网关或一个多重接入网络。每个网关定期向其他所有网关广播其链路（即拓扑图中的有向边）的状态信息。

为了确保所有有关的网关都接收到每一个链路的状态报文的副本，OSPF 维护并管理一对具有邻接关系的相邻网关之间的连接。OSPF 使用 Hello 报文来找出相邻网关，管理邻接关系，并交换拓扑信息。

在任何时候，一个网关必定处在八种状态之一。多数状态控制邻接关系的建立和数据库描述分组的建立。协议标准使用有限状态机描述相邻两个网关之间的交互作用并规范其行为。就像TCP有限状态机一样，相邻网关的状态机具有复杂的语义，无法由完全独立的一个个过程来完成，事实上，它们渗透在整个软件代码内。

深入研究

Moy［RFC 2178］包含了 OSPF 版本 2 的规范，其余信息可在［RFC 1245］和［RFC 1246］中找到。Moy［RFC 1584］进一步讨论了利用组播进行选路。

习题

19.1 RIP 和 OSPF 哪个更新路由的速度更快？为什么？

19.2 是否每一对由点到点线路连接的串行网关都应建立邻接关系？请说明原因？

19.3 研读协议标准，了解有关 OSPF 的“虚链路”（virtual link）的概念。什么时候应当使用“虚链路”？它的优点是什么？

19.4 考虑选取指定路由器的过程。请描述在什么情况下指定路由器会有所变动。如果其中的一个路由器定期地断连和重新连接，指定路由器是否仍然稳定？请解释。

19.5 仔细阅读 OSPF 协议标准，考虑选取指定路由器的过程。这个过程稳定吗？为什么？

第20章 SNMP：MIB变量、表示形式和绑定

20.1 引言

简单网络管理协议SNMP（Simple Network Management Protocol）帮助网络管理员找出并纠正TCP/IP互联网中的故障。管理员在本地计算机上（通常是一个工作站）调用SNMP客户，利用客户与一个或多个运行在远程机器（通常为网关）上的SNMP服务器取得联系。SNMP使用"取-存"（fetch-store）模式，其中每个服务器上都具有一组概念性变量，它们不仅包括简单的统计变量，例如计算接收到多少个分组的计数器，还有复杂的、与TCP/IP数据结构相对应的变量，例如ARP高速缓存和IP选路表。SNMP报文或者要求服务器提取变量的值，或者要求它将一个值保存到变量中，而服务器会将这两种请求转换成等价的在本地数据结构上的操作。由于协议不包括其他操作，因而所有控制必须在"取-存"模式中完成。除了SNMP协议之外，还有一个独立的"管理信息库"（Management Information Base）标准，它定义了一组由SNMP服务器维护的变量以及每个变量的含义。MIB变量记录了每一个相连网络的状态、通信量的统计数据、发生差错的次数以及内部数据结构的当前内容，例如该机器的IP选路表的内容。

SNMP定义了在客户和服务器之间交换的SNMP报文的语法和语义。它使用"抽象语法表示法1"（Abstract Syntax Notation One，ASN.1）来定义SNMP报文的格式和MIB变量的名称[1]。因此与绝大多数TCP/IP协议不同，SNMP报文不含固定字段，也不能用固定的结构来定义。

本章描述了SNMP软件的总体结构。它讨论了一组MIB变量及其ASN.1名字，并讲解服务器是如何将ASN.1名字与相应的内部变量绑定的。本章关心的是SNMP允许的操作，并解释了服务器为什么必须将它提供的ASN.1名字按字典顺序排列。下一章所关心的是客户如何生成请求分组，以及服务器是如何处理它们的。

20.2 服务器的组织和名字的映射

SNMP服务器必须接受输入的请求，执行指定的操作，并返回一个响应。有必要从基础上理解服务器是如何处理SNMP报文的，因为这有助于理解在本章中今后不断会遇到的完成变量名字映射的软件。图20.1描述了经过一个服务器的SNMP报文的流程。

如图20.1所示，服务器首先分析SNMP报文并将其转换为内部格式。然后它将MIB变量与本地数据项——映射，在这些本地数据项中保存了所需的信息，最后执行取出或存回的操作。对取出操作来说，它将获取到的值保存在SNMP报文的数据区中。如果该报文指定了多个变量，服务器为每个变量循环执行第三和第四个步骤。一旦所有操作执行完毕，服务器将应答分组从内部格式转换为外部格式，并将其返回给服务器。

后面各节描述MIB变量，并重点讲述变量名映射的细节。在今后的章节中将给出服务器软件的其余部分。

---

① ASN.1中名字被称为对象标识符。

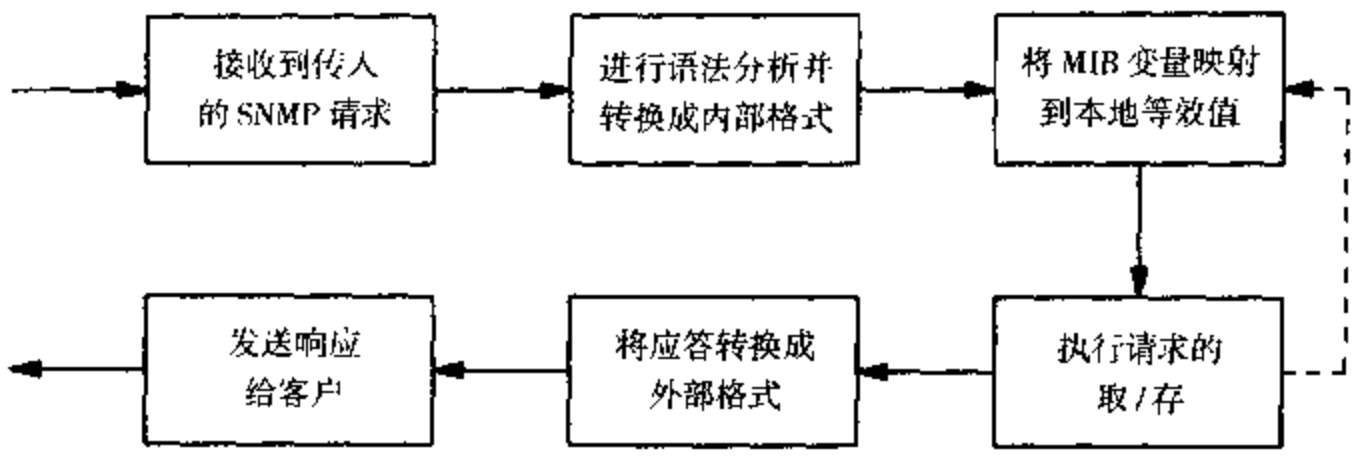


图 20.1　经过一个服务器的 SNMP 报文的流程。对每一个由该报文指定的变量，服务器重复执行第三和第四个步骤

20.3　MIB 变量

MIB 定义了 SNMP 服务器必须维护的所有变量。更精确的说，MIB 还定义了一组必须能够被 SNMP服务器访问到的概念性变量。在大多数情况下，使用常规变量就能够保存MIB要求的数据项。然而，在某些情况下，TCP/IP 协议使用的内部数据结构并不完全与MIB要求的变量相符。在这种情况下，SNMP 必须能够从可用的数据结构中计算得出必要的 MIB 变量值。可以考虑网关如何保存系统操作所经历的时间，这是一个使用计算而非使用变量的例子。有很多系统仅仅记录下系统的启动时间，并通过从当前时间中减去系统启动时间来得到系统操作经历的时间。这样 SNMP 软件能够模拟一个 MIB "变量"，该变量记录了自最近一次启动以来经历的时间。一旦有需要"读取"这个 MIB 变量值的请求到达，SNMP 就执行如上所述的计算。总结如下：

> MIB 定义了一些概念性变量，它们并不直接对应于网关使用的数据结构。SNMP 软件可以通过完成一些计算来模拟这些概念性变量，但远程网点却不会知道这些计算的存在。

大体说来，MIB 中的变量可划分为两部分：简单变量（simple variable）和表格（table）。简单变量包括诸如有符号的或无符号的整型变量及字符串之类，也包括一些数据集合，它们对应于诸如 C 程序设计语言中的"结构"或 Pascal 程序设计语言中的"记录"。通常，网关为每个简单变量维护一个实例（例如，为计算 IP 总共接收到多少个数据报而维护的一个整型变量）。表格对应于一维数组，一张表格可以包含变量的多个实例。例如，MIB 定义了一张对应于与该机器相连的一组网络接口的表格，每个网络接口在该表中对应一个表项。MIB 还定义了另外一些概念性表格，它们对应于服务器上的 IP 选路表、ARP 高速缓存以及一组 TCP 连接。

虽然简单变量的长度是预先可知的，但表格的长度却可能随时变化。例如，对应于 ARP 高速缓存的表格的长度，随着删除旧的表项以及增添新的表项而变化着。在任意时刻，ARP 高速缓存中的每一个绑定对应于 MIB 地址转换表格中的一个表项。如果某个 ARP 绑定的寿命超时，高速缓存管理软件就删除它，而对应的 MIB 表格也就少了一个表项。

20.3.1　表格中的字段

MIB 表格中的每个表项可以具有多个字段，这些字段本身有可能是简单变量，也有可能是表格。这样，可以定义基本的数据集合，如一个整型数组，也可以定义更复杂的，如地址绑定序偶的一个数组。

20.4 MIB变量名

MIB根据ASN.1为所有的变量命名。ASN.1定义了一个分级的名字空间，每个变量名都反映出它在分级结构中的位置。ASN.1变量名分级的关键在于其仔细地将“权限”（authority）分配给各个组织用于命名。该策略虽然保证了多个组织并行地分配名字，但结果得到的MIB变量名必定是惟一的（unique）和绝对的（absolute）。例如，MIB名字的分级结构从国际标准组织ISO开始，相继经过组织（organization）子级、美国国防部（Department of Defense）子级、Internet子级、管理（management）子级，最终可到达MIB子级。分级结构中每个部分都分配有一个标号（label），变量名用一系列的代表各子级的标号写成，标号之间用句点相隔，其中最高层的标号位于最左边。因此，在ip子级中用于输入IP数据报的计数变量ipInReceives的MIB变量名是：

iso.org.dod.internet.mgmt.mib.ip.ipInReceives

如上所示，MIB变量名有时会相当长。显然，表格中表项的名字比简单变量的名字还要长，因为它们还包括由该表项索引值和表项中所需字段名编码而成的附加标号。

20.4.1 变量名的数字表示形式

在发送和接收报文时，SNMP并不是以文本字符串的形式保存变量名。事实上，它使用ASN.1中的数字格式来表示每个变量名，因为数字表示形式比文本表示形式更为精炼，它节省了分组的空间。

ASN.1中的数字格式为变量名中的每个标号分配一个惟一的（通常是很小的）整数，并用一个整数序列来表示该变量名。例如，用一组序列标号来表示变量ipInReceives的名字：

1.3.6.1.2.1.4.3

当它们出现在SNMP报文中时，简单变量名的数字表示形式在尾部附加一个0，以指明该名字代表MIB变量的惟一的一个实例，因此其精确格式为：

1.3.6.1.2.1.4.3.0

20.5 名字之间的字典顺序

ASN.1定义了名字之间的字典顺序，因而提供了SNMP功能的基础。该字典顺序使客户能够向服务器请求一组当前可用的变量，并且能够在不知道表格长度的情况下搜索表格。

名字之间的字典顺序与字典中各单字之间的顺序相同，并且根据规定使用它们的数字表示形式。如果数字表示形式完全相同，则我们说这几个变量名的字典序相同。否则通过比较它们的标号来判定它们之间的顺序。如果一个名字恰好是另一个名字的前缀，则我们认为，较短的名字按照字典顺序比较长的名字要小。如果两个名字既不相同，也无包容关系，则它们中至少有一个标号是不同的。用n_1、n_2、n_3…来表示名字n中标号的数字形式，并用m_1、m_2、m_3…来表示名字m的数字形式的标号。找出第一个n_i与m_i不同的标号i。如果n_i小于m_i，我们认为按照字典顺序n小于m，否则n大于m。总结如下：

MIB 变量名之间的字典顺序以 ASN.1 对象标识符的数字表示形式定义。如果两个名字具有完全相同的表示形式，则它们的字典顺序相同。如果其中的某个是另一个的前缀，或者如果两个名字在其第一个不相同的标号处相比较时，某个标号小于另一个标号，则这个名字小于另一个名字。

20.6　除去前缀

因为SNMP软件只需要处理MIB变量，又因为所有MIB变量名都有相同的前缀，所以该软件通过提供省略了共同前缀的内部变量，来消除不必要的计算并节省空间。特别是对分组中的每个变量来说，它们的前缀必定是MIB变量名序列：

iso.org.dod.internet.mgmt.mib

或以数字形式表示为：

1.3.6.1.2.1

一旦某个SNMP服务器通过检查前缀验证了一个变量名确实是MIB变量名后，就会忽略该前缀，并且在内部只需要使用省略了前缀后剩余的变量名部分即可。正如我们将要看到的，这样做不但可以节省时间，而且被保存的内部表示形式也较短。类似地，某个客户可以仅仅在发送报文的前一刻才为变量名添加共同的前缀，以节省空间。总结如下：

SNMP软件通过保存并管理MIB变量名的后缀，以达到改善性能的目的。它在向另一个机器发送报文的前一刻，才为报文中的变量名添加它们共同的前缀，而在报文到达后（在验证这是MIB变量之后），除去这个共同前缀。

20.7　在 MIB 变量上执行的操作

在考察那些被SNMP软件用来保存信息的数据结构之前，先考虑一下服务器将如何使用这些数据结构是十分必要的。一个客户可以向服务器发出三种类型的基本命令，其中有两条命令比较显而易见，并且只需要使用简单的映射。客户发送一个 set-request（设置请求）为某个变量赋值，而当它发送一个 get-request（读取请求）时是为了读取保存在变量中的当前值。在服务器执行请求命令之前，它必须将输入的请求分组中数字编码的ASN.1变量名转换为保存着所求信息的内部变量名，并保证每个变量没有被指定为只读。

客户还可发出 get-next-request（读取下一个请求）命令。与 set-request 或 get-request 命令不同，get-next-request命令指定的变量名不是要获取的数据项的变量名。事实上，它指定了一个变量名，并要求服务器响应按字典顺序紧跟在指定变量后的变量的名字和值。服务器找出按字典顺序大于指定变量的下一个变量，并在该变量上执行 get-request 操作，以取出所需的变量值。

get-next-request命令对访问未知长度的表格尤其有用。客户可以通过不断发出get-next-request命令，使服务器自动地经过表中所有表项。由于服务器能够在响应分组中提供下一个表项的名字，因而每次请求中指定的变量名是上一次请求时对方响应返回的变量名。每次一个地通过表中所有的表项称为“走过表格”（walking the table）。

20.8 表格名

为了便于使用get-next-request命令，有些MIB名字对应于整个表格，而不是表格中的个别表项。这些变量名没有单独的值，并且客户无法在get-request命令中直接使用这些名字。然而客户可以在get-next-request命令中指定这样的变量名，以取出表格中的第一个表项。例如，MIB定义了变量名：

iso.org.dod.internet.mgmt.mib.ip.ipAddrTable

该变量名指向一个概念上的IP地址表格，通过它可找出运行服务器进程的机器[①]，其数字表示形式为：

1.3.6.1.2.1.4.20

由于表格名并不对应于某个具体的表项，因此使用表格名的get-request命令是错误的。然而，表格名的存在又是十分必要的，因为它使得客户在不知道上一个MIB变量名的情况下，也能够找到表格中的第一个表项。客户发出get-next-request命令，用表格名即可提取表格中的第一个表项。然后，它利用该命令返回的响应中的名字，通过不断地发出get-next-request命令，逐个遍历表格中的所有表项。

使得用于支持get-next-request命令的数据结构的实现复杂化的最后一项规则是：get-next-request总是按字典顺序移向下一个简单变量。更重要的是，某个服务器上变量的当前可用内容决定了该服务器可能跳过去的一组变量名。尤其对空表来说，get-next-request命令总要略过它。因此，如果当客户向一个服务器的ARP高速缓存发送get-next-request报文时，该高速缓存恰好为空，则服务器将按照字典顺序移向下一个变量，而这个变量可能与所请求的变量毫无关系。总结如下：

> 服务器在响应get-next-request命令时，仅返回简单变量的值。当这个请求到达而表格为空时，它会略过该空表格。

因此，服务器无法简单地按字典顺序排列的MIB变量表来判断哪一个变量才是get-next-request命令所需要的。事实上，它必须具有检查按字典顺序排列的各个数据项的一段程序代码，该段代码略过所有空表格，并找到下一个非空数据项中的第一个简单变量。

20.9 名字体系概念上的线索

如果将MIB变量根据其ASN.1名字安排成一个体系，则这些变量定义了一个树结构。如图20.2所示，对“字典顺序的下一个”（lexically next）的定义可被认为是树中的一组线索（thread）。

---

① 标准使用“实体代理”（entity agent）一词表示运行服务器进程的机器。

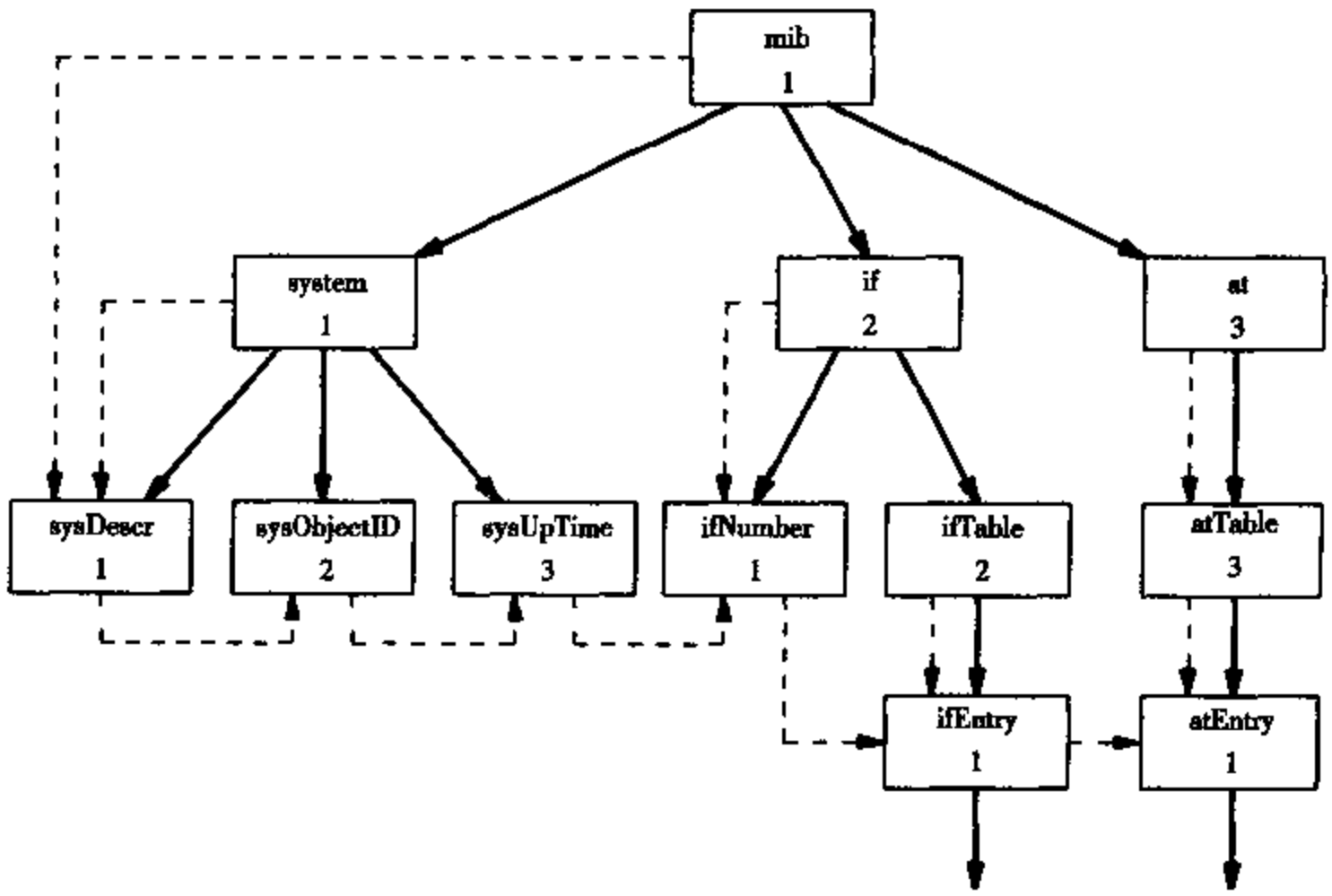


图 20.2　MIB 变量命名树的一部分。虚线表示树中的线索，它们定义了“字典顺序的下一个”。按字典排序时，跳过对应于表格的结点

20.10　MIB 变量的数据结构

SNMP必须为每个概念性MIB变量保存一些信息。在实现中，使用了一个数组来保存这些信息，数组中每个数组元素对应于一个 MIB 变量。文件 mib.h 中的结构 mib_info 定义了这些数组元素的内容。

```
/* mib.h */

#ifndef _MIB_H_
#define _MIB_H_

/* mib information structure: 1 per mib entry;  Initialized in snmib.c */
struct mib_info {
        char     *mi_name;        /* name of mib variable in English    */
        char     *mi_prefix;      /* prefix in English (e.g., "tcp.")   */
        struct   oid mi_objid;    /* object identifier                  */
        int      mi_vartype;      /* type: integer, aggregate, octet str  */
        Bool     mi_writable;     /* is this variable writable?         */
        Bool     mi_varleaf;      /* is this a leaf with a single value   */
        int      (*mi_func)(struct snbentry *, struct mib_info *, int);
                                  /* function to implement get/set/getnext*/
        char     *mi_param;       /* parameter used with function       */
        struct   mib_info *mi_next; /* pointer to next var. in lexi-order*/
```

```
};

extern struct mib_info mib[];     /* array with one entry per MIB variable*/
extern int mib_entries;           /* number of entries in mib array     */

/* Information about MIB tables.  Contains functions to implement     */
/* operations upon variables in the tables.                           */
struct tab_info {
        int (*ti_get)(struct snbentry*, int);    /* get operation      */
        int (*ti_getf)(struct snbentry*, struct mib_info*, int);
                                                 /* get first operation*/
        int (*ti_getn)(struct snbentry*, struct mib_info*, int);
                                                 /* getnext operation  */
        int (*ti_set)(struct snbentry*, struct mib_info*, int);
                                                 /* set operation      */
        int (*ti_match)(struct snbentry*, ...);
                                /* match operation: is a given oid    */
                                /*    found in the current table?     */
        struct mib_info *ti_mip;/* pointer to mib information record  */
};

extern struct tab_info tabtab[]; /* table of MIB table information     */

#define LEAF          1     /* This MIB variable is a leaf.            */
#define NLEAF         0     /* This MIB variable is not a leaf.        */

#define T_TCPTABLE          0x0  /* var is the TCP conn. table        */
#define T_RTTABLE           0x1  /* var is the routing table          */
#define T_AETABLE           0x2  /* var is the address entry tbl      */
#define T_ATTABLE           0x3  /* var is the addr translat tbl      */
#define T_IFTABLE           0x4  /* var is the interface table        */
#define T_UDPTABLE          0x5  /* var is the UDP table              */
#define T_NMTABLE           0x6  /* var is the IP NetToMedia tbl      */
/* #define T_EGPTABLE       0x7 */     /* var is the egp table        */
#define T_AGGRIGATE         0x8       /* var is an aggregate         */

/* this type specifies in mib.c that the object is a table.  The value is
   different than any of the ASN.1 types that SNMP uses. */
#define T_TABLE             01

#endif  /* _MIB_H_ */
```

数组中的每个数据元素都为识别一个变量提供了足够的信息，包括该变量的ASN.1名字（对象标识符）、类型以及是否可写的特征。

20.10.1　使用独立的函数完成操作

大多数SNMP服务器的数据结构将重点放在提高变量名转换的效率上。当含有ASN.1名字的请求分组到达时，服务器必须能够识别该名字，并调用一个能够满足该请求的过程。实现方案并不试图将所有有关如何满足该请求的信息编码保存，而是调用一个函数。该函数有三个入口参数：一个请求、一个参数（通常是存储器地址）以及一个指针，指向该ASN.1名字在mib表中对应的表项。结构 *mib_info* 中的字段 *mi_func* 的内容是为指定的变量而调用的函数的地址。

很多要访问变量的函数相当简单：它们只需从内部表示形式转换成SNMP报文使用的ASN.1格式。在这种情况下，可以对转换类型和mib表项中的变量地址进行编码，并安排一个过程来处理这种转换。然而，如果某个MIB变量在服务器中没有明确有效的表示形式，则对该变量的存取操作可能需要做更多的计算，而不仅仅是表示形式上的转换。当然，访问概念性MIB表中表项的请求需要根据计算才能从 MIB 名字转换为用于保存信息的本地数据结构。最后，SNMP 必须提供某种方法，以计算哪一个 *mib* 表项可满足 get-next-request 命令。为了保持实现方案的统一，并具有相当的通用性，使之能适用于所有请求分组，实现方案总是利用一个函数来处理请求。我们总结如下：

> 以某个 MIB 变量为对象的某个请求所需的计算量取决于该变量及请求。使用一个独立的函数处理访问请求使得实现方案统一化，且使该服务器能够只用一种机制就可处理所有请求。

20.11　用于快速查找的数据结构

后面将要看到的，数组 mib 中包含有 ASN.1 名字（对象标识符）和内部变量之间的绑定。如果使用线性查找，则查找其中的一个表项将花费相当长的时间。为了提高变量名的转换效率，软件使用如图 20.3 所示的辅助散列表。

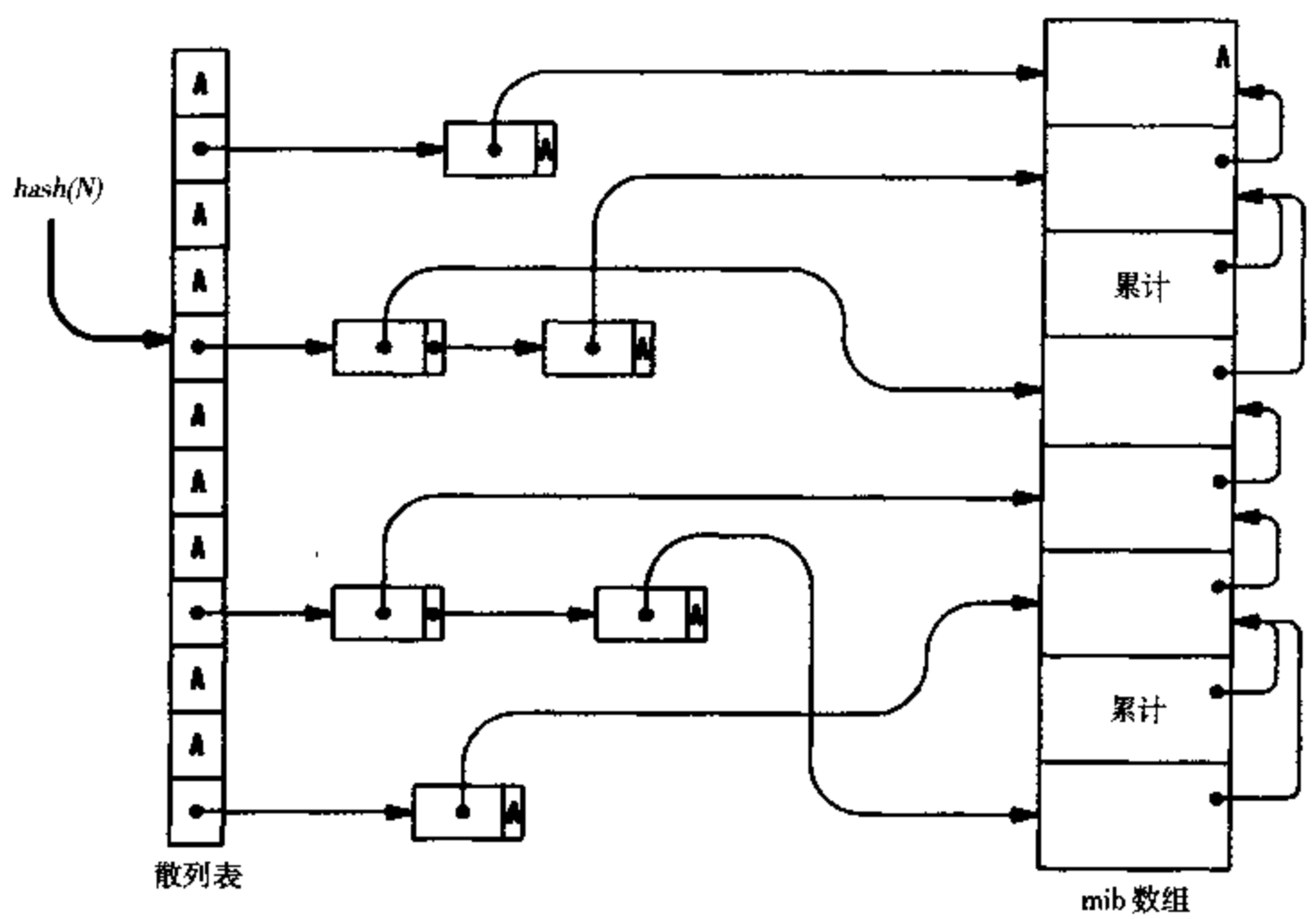


图 20.3　mib 数组以及用于提高 MIB 变量查找速度的散列表 snhtab。右边的箭头表示 get-next-request 命令使用的字典顺序链接，字典顺序跳过无叶的表项

这里使用的技术被称为“桶散列”(bucket bashing)。散列表本身由一个指针数组组成，其中每个指针给出一个链表的地址，链表中的结点代表可进行散列寻址的MIB变量。要找出某个MIB变量的信息，软件利用该变量名的数字表示形式计算散列值，从散列表中选出一个链表，然后搜索该链表。链表中的每个表项不仅包含一个指向它所代表的MIB变量的指针，还有一个指向链表中下一个表项的指针。

在实际中，一种简单的散列技术就可以得到很好的效果。对MIB中定义的122[1]个变量来说，用散列技术将它们置入具有101个元素的数组，其结果是链表的平均长度比1稍大一些。因此，大多数查找能立即找到所需的值，并不需要软件搜索链表。

20.12 散列表的实现

文件snhash.h中包含了对散列表和链表结点的说明。

```
/* snhash.h */

struct snhnode {                        /* hash table node structure    */
        struct mib_info *sh_mip;        /* points to a mib record       */
        struct snhnode *sh_next;        /* next node in this hash bucket*/
};

#define S_HTABSIZ       101             /* hash table size - a prime #  */
#define S_HTRADIX       21              /* hash table radix             */

#define S_NUMTAB        7               /* number of table objects      */

extern  struct snhnode *snhtab[];       /* the hash table               */
```

结构snhnode定义了链表中结点的内容。每个结点仅含有两个指针，其中一个指向mib数组的一个数组元素，另一个指向该链表的下一个结点。

20.13 MIB绑定的描述

在文件snmib.c中包含的程序代码用于初始化mib数组，使数组中的所有元素与MIB变量一一对应。在这些数组元素中包含了所有与该MIB对象有关的信息，包括它的ASN.1名字的数字表示形式，一个含有与MIB变量相关值的内部变量的地址，该内部变量的值与MIB变量相关，以及一个函数地址，调用该函数可以用来完成SNMP在该变量上的操作。注意在所有这些静态定义中，按字典顺序指向下一个表项的指针均被设置为0。虽然字典顺序列表指针没有在此次定义中初始化，但所有MIB对象必须以字典逆序给出。SNMP软件将在系统启动时初始化这些字典顺序列表指针。

```
/* snmib.c */

#include <conf.h>
```

① 范例代码中不包括EGP变量。

```
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>
#include <snmpvars.h>

extern struct oid SysObjectID;

/* All table and leaf variables that are found in the MIB */
struct mib_info mib[] = {
 /* System Group - 1 */
     { "system", "", { {1}, 1}, T_AGGREGATE,
          FALSE, NLEAF, 0, 0, 0},
     { "sysDescr", "system.", { {1, 1, 0}, 3}, ASN1_OCTSTR,
          FALSE, LEAF, snleaf, SysDescr, 0},
     { "sysObjectID", "system.", { {1, 2, 0}, 3}, ASN1_OBJID,
          FALSE, LEAF, snleaf, (char *) &SysObjectID, 0},
     { "sysUpTime", "system.", { {1, 3, 0}, 3}, ASN1_TIMETICKS,
          FALSE, LEAF, snleaf, (char *) &SysUpTime, 0},
     { "sysContact", "system", { {1, 4, 0}, 3}, ASN1_OCTSTR,
          TRUE, LEAF, snleaf, SysContact, 0},
     { "sysName", "system.", { {1, 5, 0}, 3}, ASN1_OCTSTR,
          TRUE, LEAF, snleaf, SysName, 0},
     { "sysLocation", "system.", { {1, 6, 0}, 3}, ASN1_OCTSTR,
          TRUE, LEAF, snleaf, SysLocation, 0},
     { "sysServices", "system.", { {1, 7, 0}, 3}, ASN1_INT,
         FALSE, LEAF, snleaf, (char *) &SysUpTime, 0},
/* Interfaces Group - 2 */
     { "if", "", { {2}, 1}, T_AGGREGATE,
          FALSE, NLEAF, 0, 0, 0},
     { "ifNumber", "if.", { {2, 1, 0}, 3}, ASN1_INT,
          FALSE, LEAF, snleaf, (char *) &IfNumber, 0},
     { "ifTable", "if.", { {2, 2}, 2}, T_AGGREGATE,
          TRUE, NLEAF, 0, 0, 0},
     { "ifEntry", "if.ifTable.", { {2, 2, 1}, 3}, T_TABLE,
          TRUE, NLEAF, sntable, (char *) &tabtab[T_IFTABLE], 0},
/* Address Translation Group - 3  (deprecated in RFC 1213) */
     { "at", "", { {3}, 1}, T_AGGREGATE,
          TRUE, NLEAF, 0, 0, 0},
     { "atTable", "at.", { {3, 1}, 2}, T_AGGREGATE,
          TRUE, NLEAF, 0, 0, 0},
     { "atEntry", "at.atTable.", { {3, 1, 1}, 3}, T_TABLE,
```

```
            TRUE, NLEAF, sntable, (char *) &tabtab[T_ATTABLE], 0},
/* IP Group - 4 */
    { "ip", "", { {4}, 1}, T_AGGREGATE,
            TRUE, NLEAF, 0, 0, 0},
    { "ipForwarding", "ip.", { {4, 1, 0}, 3}, ASN1_INT,
            FALSE, LEAF, snleaf, (char *) &IpForwarding, 0},
    { "ipDefaultTTL", "ip.", { {4, 2, 0}, 3}, ASN1_INT,
            TRUE, LEAF, snleaf, (char *) &IpDefaultTTL, 0},
    { "ipInReceives", "ip.", { {4, 3, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IpInReceives, 0},
    { "ipInHdrErrors", "ip.", { {4, 4, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IpInHdrErrors, 0},
    { "ipInAddrErrors", "ip.", { {4, 5, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IpInAddrErrors, 0},
    { "ipForwDatagrams", "ip.", { {4, 6, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IpForwDatagrams, 0},
    { "ipInUnknownProtos", "ip.", { {4, 7, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IpInUnknownProtos, 0},
    { "ipInDiscards", "ip.", { {4, 8, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IpInDiscards, 0},
    { "ipInDelivers", "ip.", { {4, 9, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IpInDelivers, 0},
    { "ipOutRequests", "ip.", { {4, 10, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IpOutRequests, 0},
    { "ipOutDiscards", "ip.", { {4, 11, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IpOutDiscards, 0},
    { "ipOutNoRoutes", "ip.", { {4, 12, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IpOutNoRoutes, 0},
    { "ipReasmTimeout", "ip.", { {4, 13, 0}, 3}, ASN1_INT,
            FALSE, LEAF, snleaf, (char *) &IpReasmTimeout, 0},
    { "ipReasmReqds", "ip.", { {4, 14, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IpReasmReqds, 0},
    { "ipReasmOKs", "ip.", { {4, 15, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IpReasmOKs, 0},
    { "ipReasmFails", "ip.", { {4, 16, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IpReasmFails, 0},
    { "ipFragOKs", "ip.", { {4, 17, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IpFragOKs, 0},
    { "ipFragFails", "ip.", { {4, 18, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IpFragFails, 0},
    { "ipFragCreates", "ip.", { {4, 19, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IpFragCreates, 0},
    { "ipAddrTable", "ip.", { {4, 20}, 2}, T_AGGREGATE,
        FALSE, NLEAF, 0, 0, 0},
    { "ipAddrEntry", "ip.ipAddrTable.", { {4, 20, 1}, 3}, T_TABLE,
```

```
            FALSE, NLEAF, sntable, (char *) &tabtab[T_AETABLE], 0},
        { "ipRouteTable", "ip.", { {4, 21}, 2}, T_AGGREGATE,
            TRUE, NLEAF, 0, 0, 0},
        { "ipRouteEntry", "ip.ipRouteTable.", { {4, 21, 1}, 3}, T_TABLE,
            TRUE, NLEAF, sntable, (char *) &tabtab[T_RTTABLE], 0},
        { "ipNetToMediaTable", "ip.", { {4, 22}, 2}, T_AGGREGATE,
            TRUE, NLEAF, 0, 0, 0},
        { "ipNetToMediaEntry", "ip.ipNetToMediaTable.", { {4, 22, 1}, 3}, T_TABLE,
            TRUE, NLEAF, sntable, (char *) &tabtab[T_NMTABLE], 0},
        { "ipRoutingDiscards", "ip.", { {4, 23, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, sntable, (char *) &IpRoutingDiscards, 0},
/* ICMP Group - 5 */
        { "icmp", "", { {5}, 1}, T_AGGREGATE,
            FALSE, NLEAF, 0, 0, 0},
        { "icmpInMsgs", "icmp.", { {5, 1, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IcmpInMsgs, 0},
        { "icmpInErrors", "icmp.", { {5, 2, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IcmpInErrors, 0},
        { "icmpInDestUnreachs", "icmp.", { {5, 3, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IcmpInDestUnreachs, 0},
        { "icmpInTimeExcds", "icmp.", { {5, 4, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IcmpInTimeExcds, 0},
        { "icmpInParmProbs", "icmp.", { {5, 5, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IcmpInParmProbs, 0},
        { "icmpInSrcQuenchs", "icmp.", { {5, 6, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IcmpInSrcQuenchs, 0},
        { "icmpInRedirects", "icmp.", { {5, 7, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IcmpInRedirects, 0},
        { "icmpInEchos", "icmp.", { {5, 8, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IcmpInEchos, 0},
        { "icmpInEchoReps", "icmp.", { {5, 9, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IcmpInEchoReps, 0},
        { "icmpInTimestamps", "icmp.", { {5, 10, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IcmpInTimestamps, 0},
        { "icmpInTimestampReps", "icmp.", { {5, 11, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IcmpInTimestampReps, 0},
        { "icmpInAddrMasks", "icmp.", { {5, 12, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IcmpInAddrMasks, 0},
        { "icmpInAddrMaskReps", "icmp.", { {5, 13, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IcmpInAddrMaskReps, 0},
        { "icmpOutMsgs", "icmp.", { {5, 14, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IcmpOutMsgs, 0},
        { "icmpOutErrors", "icmp.", { {5, 15, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &IcmpOutErrors, 0},
        { "icmpOutDestUnreachs", "icmp.", { {5, 16, 0}, 3}, ASN1_COUNTER,
```

```
        FALSE, LEAF, snleaf, (char *) &IcmpOutDestUnreachs, 0},
    { "icmpOutTimeExcds", "icmp.", { {5, 17, 0}, 3}, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (char *) &IcmpOutTimeExcds, 0},
    { "icmpOutParmProbs", "icmp.", { {5, 18, 0}, 3}, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (char *) &IcmpOutParmProbs, 0},
    { "icmpOutSrcQuenchs", "icmp.", { {5, 19, 0}, 3}, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (char *) &IcmpOutSrcQuenchs, 0},
    { "icmpOutRedirects", "icmp.", { {5, 20, 0}, 3}, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (char *) &IcmpOutRedirects, 0},
    { "icmpOutEchos", "icmp.", { {5, 21, 0}, 3}, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (char *) &IcmpOutEchos, 0},
    { "icmpOutEchoReps", "icmp.", { {5, 22, 0}, 3}, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (char *) &IcmpOutEchoReps, 0},
    { "icmpOutTimestamps", "icmp.", { {5, 23, 0}, 3}, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (char *) &IcmpOutTimestamps, 0},
    { "icmpOutTimestampReps","icmp.", { {5, 24, 0}, 3}, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (char *) &IcmpOutTimestampReps, 0},
    { "icmpOutAddrMasks", "icmp.", { {5, 25, 0}, 3}, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (char *) &IcmpOutAddrMasks, 0},
    { "icmpOutAddrMaskReps", "icmp.", { {5, 26, 0}, 3}, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (char *) &IcmpOutAddrMaskReps, 0},
/* TCP Group - 6 */
    { "tcp", "", { {6}, 1}, T_AGGREGATE,
        FALSE, NLEAF, 0, 0, 0},
    { "tcpRtoAlgorithm", "tcp.", { {6, 1, 0}, 3}, ASN1_INT,
        FALSE, LEAF, snleaf, (char *) &TcpRtoAlgorithm, 0},
    { "tcpRtoMin", "tcp.", { {6, 2, 0}, 3}, ASN1_INT,
        FALSE, LEAF, snleaf, (char *) &TcpRtoMin, 0},
    { "tcpRtoMax", "tcp.", { {6, 3, 0}, 3}, ASN1_INT,
        FALSE, LEAF, snleaf, (char *) &TcpRtoMax, 0},
    { "tcpMaxConn", "tcp.", { {6, 4, 0}, 3}, ASN1_INT,
        FALSE, LEAF, snleaf, (char *) &TcpMaxConn, 0},
    { "tcpActiveOpens", "tcp.", { {6, 5, 0}, 3}, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (char *) &TcpActiveOpens, 0},
    { "tcpPassiveOpens", "tcp.", { {6, 6, 0}, 3}, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (char *) &TcpPassiveOpens, 0},
    { "tcpAttemptFails", "tcp.", { {6, 7, 0}, 3}, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (char *) &TcpAttemptFails, 0},
    { "tcpEstabResets", "tcp.", { {6, 8, 0}, 3}, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (char *) &TcpEstabResets, 0},
    { "tcpCurrEstab", "tcp.", { {6, 9, 0}, 3}, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (char *) &TcpCurrEstab, 0},
    { "tcpInSegs", "tcp.", { {6, 10, 0}, 3}, ASN1_COUNTER,
        FALSE, LEAF, snleaf, (char *) &TcpInSegs, 0},
    { "tcpOutSegs", "tcp.", { {6, 11, 0}, 3}, ASN1_COUNTER,
```

```
            FALSE, LEAF, snleaf, (char *) &TcpOutSegs, 0},
        { "tcpRetransSegs", "tcp.", { {6, 12, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &TcpRetransSegs, 0},
        { "tcpConnTable", "tcp.", { {6, 13}, 2}, T_AGGREGATE,
            FALSE, NLEAF, 0, 0, 0},
        { "tcpConnEntry", "tcp.tcpConnTable.", { {6, 13, 1}, 3}, T_TABLE,
            FALSE, NLEAF, sntable, (char *) &tabtab[T_TCPTABLE], 0},
        { "tcpInErrs", "tcp.", { {6, 14, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &TcpInErrs, 0},
        { "tcpOutRsts", "tcp.", { {6, 15, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &TcpOutRsts, 0},
/* UDP Group - 7 */
        { "udp", "", { {7}, 1}, T_AGGREGATE,
            FALSE, NLEAF, 0, 0, 0},
        { "udpInDatagrams", "udp.", { {7, 1, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &UdpInDatagrams, 0},
        { "udpNoPorts", "udp.", { {7, 2, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &UdpNoPorts, 0},
        { "udpInErrors", "udp.", { {7, 3, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &UdpInErrors, 0},
        { "udpOutDatagrams", "udp.", { {7, 4, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf, (char *) &UdpOutDatagrams, 0}
        { "udpTable", "udp.", { {7, 5}, 2}, T_AGGREGATE,
            FALSE, NLEAF, 0, 0, 0},
        { "uctpTableEntry", "udp.udpTable.", { {7, 5, 1}, 3}, T_TABLE,
            FALSE, NLEAF, sntable,  (char *)&tabtab[T_UDPTABLE], 0},
/* EGP Group - 8  (not used by Xinu)  */
/* CMOT Group- 9  (historical  (hysterical))  */
/* TRANSMISSION Group - 10  (experimental, not yet defined)  */
/* SNMP Group - 11  */
        { "snmp",  ""  , { {11}, 1}, T_AGGREGATE,
            FALSE, NLEAF, 0, 0, 0},
        { "snmpInPkts", "snmp.", { {11, 1, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf,  (char *)&snmpInPkts, 0},
        { "snmpOutPkts", "snmp.", { {11, 2, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf,  (char *)&snmpOutPkts, 0},
        { "snmpInBadVersions",  "snmp.",  { {11, 3, 0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf,  (char *) &snmpInBadVersions,  0},
        { "snmpInBadCommunityNames",  "snmp.",  { {11,4,0},3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf,  (char *)&snmpInBadCommunityNames, 0},
        { "snmpInBadCommunityUses", "snmp.",  { {11,5,0}, 3}, ASN1_COUNTER,
            FALSE, LEAF,  snleaf,  (char *)&snmpInBadCommunityUses,  0},
        { "snmpInASNParseErrs",  "snmp.",  { {11,  6,  0}, 3}, ASN1_COUNTER,
            FALSE, LEAF, snleaf,  (char *)&snmpInASNParseErrs, 0},
/* snmp 7 is not defined by the RFC */
```

```
{ "snmpInTooBigs", "snmp.", { {11, 8, 0}, 3}, ASN1_COUNTER,
    FALSE  LEAF, snleaf,  (char *)&snmpInTooBigs, 0},
{ "snmpInNoSuchNames",  "snmp.",  { {1I, 9, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf,  (char *)&snmpInNoSuchNames,  0},
{ "snmpInBadValues",  "snmp.",  { {11, 10, 0}, 3}, ASN1_COUNTER,
    FALSE  LEAF, snleaf,  (char *)&snmpInBadValues,  0},
{ "snmpInReadOnlys", "snmp.", { {I1, 11, 0}, 3}, ASN1_COUNTER,
    FALSE  LEAF, snleaf,  (char *)&snmpInReadOnlys,  0},
{ "snmpInGenErrs", "snmp.",  { {1I, 12, 0}, 3}, ASN1_COUNTER,
    FALSE  LEAF, snleaf,  (char *)&snmpInGenErrs,  0},
{ "snmpInTotalReqVars",  "snmp.",  { {11, 13, 0}, 3}, ASN1_COUNTER,
    FALSE  LEAF, snleaf,  (char *)&snmpInTotalReqVars,  0},
{ "snmpInTotalSetVars",  "snmp.",  { {11, 14, 0}, 3}, ASN1_COUNTER,
    FALSE  LEAF, snleaf,  (char *)&snmpInTotalSetVars,  0},
{ "snmpInGetRequests", "snmp.", { {11, 15, 0}, 3}, ASNI_COUNTER,
    FALSE  LEAF, snleaf,  (char * ) &snmpInGetRequests, 0},
{ "snmpInGetNexts", "snmp.", { {11, 16, 0}, 3}, ASN1_COUNTER,
    FALSE  LEAF, snleaf,  (char *)&snmpInGetNexts,  0},
{ "snmpInSetRequests", "snmp.", { {11, 17, 0}, 3}, ASN1_COUNTER,
    FALSE  LEAF, snleaf,  (char *)&snmpinSetRequests,  0},
{ "snmpInGetResponses",  "snmp.",  { {11, 18, 0}, 3}, ASN1_COUNTER,
    FALSE  LEAF, snleaf,  (char *)&snmpInGetResponses,  0},
{ "snmpInTraps", "snmp.", { {11, 19, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF,  snleaf,  (char *)&snmpInTraps,  0},
{ "snmpOutTooBigs", "snmp.", { {11, 20, 0}, 3}, ASN1_COUNTER,
    FALSE  LEAF, snleaf,  (char *)&snmpOutTooBigs,  0},
{ "snmpOutNoSuchNames",  "snmp.",  { i11, 21, 0}, 3}, ASN1_COUNTER,
    FALSE  LEAF, snleaf,  (char *)&snmpOutNoSuchNames,  0},
{ "snmpOutBadValues",  "snmp.",  { {11, 22, 0}, 3}, ASN1_COUNTER,
    FALSE  LEAF, snleaf,  (char *)&snmpOutBadValues, 0},
/* snmp 23 is not defined by the RFC */
{ "snmpOutGenErrs", "snmp.", { {11, 24, 0}, 3}, ASN1_COUNTER,
    FALSE, LEAF, snleaf,  (char *)&snmpOutGenErrs, 0},
{ "snmpOutGetRequests",  "snmp.", { {11, 25,  0}, 3}, ASN1_COUNTER,
};
    FALSE  LEAF, snleaf,  (char *)&snmpOutGetRequests, 0},
{ "snmpOutGetNexts",  "snmp.",  { {11, 26, 0}, 3}, ASN1_COUNTER,
    FALSE  LEAF, snleaf,  (char *)&snmpOutGetNexts,  0},
{ "snmpOutGetRequests", "snmp.", { {11, 27, 0}, 3}, ASN1_COUNTER,
    FALSE  LEAF, snleaf,  (char * ) &snmpOutSetRequests, 0},
{ "snmpOutGetResponses", "snmp.", { {11, 28, 0}, 3}, ASN1_COUNTER,
    FALSE  LEAF, snleaf,  (char *)&snmpOutGetResponses,  0},
{ "snmpOutTraps", "snmp.", { {11, 29, 0}, 3}, ASN1_COUNTER,
    FALSE  LEAF, snleaf,  (char *)&snmpOutTraps,  0},
{ "snmpEnableAuthenTraps", "snmp.", { {11, 30, 0}, 3}, ASN1_INT,
```

```
            TRUE, LEAF, snleaf,  (char *)&snmpEnableAuthenTraps, 0},
};

int mib_entries = sizeof(mib) / sizeof(struct mib_info);

/* Funcs that implement get,getfirst,getnext,set, and match for tables*/

struct tab_info tabtab[] = {
        { stc_get, stc_getf, stc_getn, stc_set, stc_match, 0, },
        { srt_get, srt_getf, srt_getn, srt_set, srt_match, 0, },
        { sae_get, sae_getf, sae_getn, sae_set, sae_match, 0, },
        { sat_get, sat_getf, sat_getn, sat_set, sat_match, 0, },
        { sif_get, sif_getf, sif_getn, sif_set, sif_match, 0  },
        { sud_get, sud_getf, sud_getn, sud_set, sud_match, 0  },
        { snm_get, snm_getf, snm_getn, snm_set, snm_match, 0  },
};
#endif  /* SNMP */
```

20.14 绑定中使用的内部变量

文件snmpvars.h中定义了MIB绑定中使用的内部变量的类型。

```
/* snmpvars.h */

/* System & Interface MIB */

extern char SysDescr[], sysContart[], SysName[], SysLocation[];
extern unsigned
        SysUpTime, SysServices, IfNumber;

/* IP MIB */
extern unsigned
        IpForwarding, IpDefaultTTL, IpInReceives, IpInHdrErrors,
        IpInAddrErrors, IpForwDatagrams, IpInUnknownProtos, IpInDiscards,
        IpInDelivers, IpOutRequests, IpOutDiscards, IpOutNoRoutes,
        IpReasmTimeout, IpReasmReqds, IpReasmOKs, IpReasmFails, IpFragOKs,
        IpFragFails, IpFragCreates, IpRouting Discards;

/* ICMP MIB */
extern unsigned
        IcmpInMsgs, IcmpInErrors, IcmpInDestUnreachs, IcmpInTimeExceds,
        IcmpInParmProbs, IcmpInSrcQuenchs, IcmpInRedirects, IcmpInEchos,
        IcmpInEchoReps, IcmpInTimestamps, IcmpInTimestampReps,
        IcmpInAddrMasks, IcmpInAddrMaskReps, IcmpOutMsgs, IcmpOutErrors,
        IcmpOutDestUnreachs, IcmpOutTimeExceds, IcmpOutParmProbs,
        IcmpOutSrcQuenchs, IcmpOutRedirects, IcmpOutEchos,
        IcmpOutEchoReps, IcmpOutTimestamps, IcmpOutTimestampReps,
```

```
        IcmpOutAddrMasks, IcmpOutAddrMaskReps;

/* UDP MIB */
extern unsigned
        UdpInDatagrams, UdpNoPorts, UdpInErrors, UdpOutDatagrams;

/* TCP MIB */
extern unsigned
        TcpRtoAlgorithm, TcpRtoMin, TcpRtoMax, TcpMaxConn, TcpActiveOpens,
        TcpPassiveOpens, TcpAttemptFails, TcpEstabResets, TcpCurrEstab,
        TcpInSegs, TcpOutSegs, TcpRetransSegs TcpInErrs, TcpOutRsts;

/* SNMP MIB */
extern unsigned
        snmpInPkts, snmpOutPkts, snmpInBadVersions,
        snmPInBadCommunityNames, snmpInBadCommunityUses,
        snmpInASNParseErrs, snmpInTooBigs, snmpInNoSuchNames,
        snmpInBadValues, snmpInReadOnlys, snmpInGenErrs,
        snmpInTotalReqVars, snmpInTotalSetVars, snmpInGetRequests,
        snmpInGetNexts, snmpInSetRequests, snmpInGetResponses,
        snmpInTraps, snmpOutTooBigs, snmpOutNoSuchNames,
        snmpOutBadValues, snmpOutGenErrs, snmpOutGetRequests,
        snmpOutGetNexts, snmpOutSetRequests, snmpOutGetResponses,
        snmPOutTraps, snmpEnableAuthenTraps;
```

20.15 散列表的查找

函数getmib利用散列表查找对应于指定名字（对象标识符）的MIB表项。它调用函数hashoid来计算该名字的散列值。为了形成一个代表该对象标识符的整型数值，hashoid将标识符看作是以S_HTRADIX为基数（radix）的数字序列。它循环经过这个对象标识符，在每一次循环时，将其乘以基数并加上一个新的数字。

getmib和hashoid的源代码都在文件snhash.c中，该文件还包含有散列表的初始化过程hashinit。

```
/* snhash.c - getmib, hashoid, hashinit */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <mem.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <snhash.h>

struct snhnode *snhtab[S_HTABSIZ];
```

```
extern struct tab_info tabtab[];

/*-----------------------------------------------------------------------
 * getmib -find mib record for the given object id
 *-----------------------------------------------------------------------
 */
struct mib_info *
getmib(struct oid *oip)
{
        struct snhnode  *hp;
        int             loc, i;
        loc = hashoid(oip);         /* try the regular hash table */
        for (hp = snhtab[loc]; hp; hp = hp->sh_next)
                if (oidequ(oip, &hp->sh_mip->mi_objid)) {
                        return hp->sh_mip;
                }
        for (i = 0; i < S_NUMTABS; ++i)       /* try the table table */
                if (!memcmp(tabtab[i].ti_mip->mi_objid.id, oip->id,
                            tabtab[i].ti_mip->mi_objid.len * 2)) {
                        rentrun tabtab[i].ti_mip;
                }
        return 0;

/* ----------------------------------------------------------------------
 * hashoid-hash the object id
 * ----------------------------------------------------------------------
 */
int
hashoid(struct oid *oip)
{
        register unsigned tot;
        register int i;

        for (tot = 0, i = oip->len - 1; i >= 0, i--)
                tot = tot * S_HTRADIX + oip->id[i];
        return tot % S_HTABSIZ;
}

/*-----------------------------------------------------------------------
 * hashinit - initialize the hash table
 *-----------------------------------------------------------------------
 */
void
hashinit(void)
{
        int i;
```

```
        register struct snhnode    **ht;
        register struct mib_info   *mp;
        struct mib_info            *lastnodep;
        struct snhnode             *hp;
        int loc, tabtabct;

        /* clear the hash table */
        ht = snhtab;
        for (i = 0; i < S_HTABSIZ; i++)
                *ht++ = 0;
        lastnodep = 0;
        tabtabct = 0;
        for (i=0, mp = &mib[mib_entries - 1]; i<mib_entries; i++,  mp--) {
                loc = hashoid(&mp->mi_objid);
                hp = (struct snhnode *) getmem(sizeof(struct snhnode));
                hp->sh_mip = mp;
                hp->sh_next = snhtab[loc];
                snhtab[loc] = hp;
                mp->mi_next = lastnodep;
                /* (node == table) ==> insert into array of tables */

                if (mp->mi_vartype == T_TABLE)
                        tabtab[tabtabct++].ti_mip = mp;
                if (mp->mi_varleaf || mp->mi_vartype == T_TABLE)
                        lastnodep = mp;
        }
}
#endif  /* SNMP */
```

20.16 SNMP的结构和常量

文件snmp.h包含了整个程序代码所使用的数据结构和符号常量的定义。

```
/* snmp.h - strequ, oidequ */

#define SNMPD           snmpd           /* SNMP server code          */
extern  int             SNMPD();        /* SNMP server daemon        */
#define SNMPSTK         8000            /* SNMP server stack size    */
#define SNMPPRI         20              /* SNMP server priority      */
#define SNMPDNAM        "snmpd"         /* SNMP server daemon name   */

#define SMAXOBJID       32              /* max # of sub object ids   */
#define OBJSUBIDTYPE    unsigned short  /* type of sub object ids    */

/* strequ - return TRUE if strings x and y are the same              */
#define strequ(x,y)     (strcmp((x), (y)) == 0)
```

```
#define u_char          unsigned char

struct oid {                            /* object identifier            */
        OBJSUBIDTYPE    id[SMAXOBJID];  /* array of sub-identifiers     */
        int             len;            /* length of this object id     */
};

/*
 * oidequ - check if the lengths of the oid's are the same, then check
 *          the contents of the oid's
 */
#define oidequ(x,y)     ((x)->len == (y)->len && \
                        !memcmp((x)->id, (y)->id, (y)->len * 2))

/* Structure that contains the value of an SNMP variable.               */
struct snval {
        unsigned char  sv_type;         /* variable type                */
        union {                         /* value of var is one of these */
                int     sv_int;         /* variable is one of: integer, */
                                        /* counter, gauge, timeticks    */
                struct {                /* variable is a (octet) string */
                        char *sv_str;   /* string's contents            */
                        int sv_len;     /* string's length              */
                } sv_str;
                struct  oid sv_oid;     /* variable is an object id     */
                IPaddr  sv_ipaddr;      /* variable is an IP address    */
        } sv_val;
};

/* Functions to access parts of the above snval structure               */
#define SVTYPE(bl)      ((bl)->sb_val.sv_type)
#define SVINT(bl)       ((bl)->sb_val.sv_val.sv_int)
#define SVSTR(bl)       ((bl)->sb_val.sv_val.sv_str.sv_str)
#define SVSTRLEN(bl)    ((bl)->sb_val.sv_val.sv_str.sv_len)
#define SVOID(bl)       ((bl)->sb_val.sv_val.sv_oid.id)
#define SVOIDLEN(bl)    ((bl)->sb_val.sv_val.sv_oid.len)
#define SVIPADDR(bl)    ((bl)->sb_val.sv_val.sv_ipaddr)

/*
 * Each snblist node contains an SNMP binding in one of 2 forms: ASN.1
 * encoded form or internal form.  The bindings list is doubly-linked
 */
struct snbentry {
        struct  oid sb_oid;             /* object id in internal form   */
        struct  snval sb_val;           /* value of the object          */
        u_char  *sb_alstr;              /* ASN.1 string containing the  */
```

```
                                        /* object id and its value      */
        short     sb_alslen;            /* length of the ASN.1 string   */
        Bool      Sb_aldynstr;          /* need alstr be freemem ()ed?  */
        struct    snbentry *sb_next;    /* next node in the bind list   */
        struct    snbentry *sb_prev;    /* previous node in the list    */
};

/* Structure that holds a complete description of an SNMP request     */
struct req_desc {
        u_char    reqtype;              /* request type                 */
        u_char    reqid[10];            /* request identifier           */
        int       reqidlen;             /* length of the identifier     */
        u_char    err_stat;             /* error status                 */
        u_char    err_idx;              /* error index                  */
        int       err_stat_pos;         /* position of error status in  */
                                        /* the ASN.1 encoding           */
        int       err_idx_pos;          /* position of error index      */
        int       pdutype_pos;          /* position of pdu type         */
        struct    snbentry *bindlf;     /* front of bindings list       */
        struct    snbentry *bindle;     /* end of bindings list         */
};

#define SNMPMAXSZ      U_MAXLEN         /* max SNMP request size        */
#define SNMPPORT       161              /* SNMP server UDP port         */

/* SNMP error types                                                     */
#define SNMP_OK        0                /* no error                     */
#define SERR_TOO_BIG   1                /* reply would be too long      */
#define SERR_NO_SUCH   2                /* no such object id exists     */
#define SERR_BAD_VALUE 3                /* bad value in a set operation */

#define SVERS_LEN      1                /* SNMP version is 1 byte long  */
#define SVERSION       0                /* current SNMP version         */

#define SCOMM_STR      "public"         /* SNMP community string        */

/* operations to be applied to an SNMP object                           */
#define SOP_GET        1                /* get operation                */
#define SOP_GETN       2                /* getnext operation            */
#define SOP_GETF       3                /* get first operation          */
#define SOP_SET        4                /* set operation                */

/* standard version and community string -- backwards */
#define SNVCBACK_CHARS {\
        'c', 'i', 'l', 'b', 'u', 'p', 0x06, 0x04 /* ASN1_OCTSTR */,\
        SVERSION, SVERS_LEN, 0x02 /* ASN1_INT */\
        };
```

```
extern char SNVBACK[];

/* SNMP client return codes                                          */
#define SCL_OK        0          /* good response -- no errors       */
#define SCL_OPENF     1          /* open fails                       */
#define SCL_WRITEF    2          /* write fails                      */
#define SCL_NORESP    3          /* no response from server          */
#define SCL_READF     4          /* read fails                       */
#define SCL_BADRESP   5          /* bad response                     */
#define SCL_BADREQ    6          /* bad request                      */

/* Table specific constants                                          */
#define SNUMF_AETAB   5          /* 5 fields in an Addr Entry        */
#define SNUMF_ATTAB   3          /* 3 fields in an Addr Transl. Entry */
#define SNUMF_IFTAB   22         /* 22 fields in an Interface Entry  */
#define SNUMF_RTTAB   13         /* 13 fields in a Route Table Entry */
#define SNUMF_TCTAB   5          /* 5 fields in a TCP Connection Entry */
#define SNUMF_UDTAB   2          /* 2 fields in a UDP Listener Entry */
#define SNUMF_NMTAB   4          /* 4 fields in a IP NetToMedia Entry */

#define SAE_OIDLEN    3          /* all sae variables start with 4.20.1 */
#define SAT_OIDLEN    3          /* all sat variables start with 3.1.1  */
#define SIF_OIDLEN    3          /* all sif variables start with 2.2.1  */
#define SRT_OIDLEN    3          /* all srt variables start with 4.21.1 */
#define STC_OIDLEN    3          /* all stc variables start with 6.13.1 */
#define SUD_OIDLEN    3          /* all srt variables start with 7.5.1  */
#define SNM_OIDLEN    3          /* all stc variables start with 4.22.1 */

#include <mib. h>

/* differences between Xinuv7 and Xinuv7 * /
#define IP2I (ipaddr) ((int) ipaddr)


/* internal routines * /
int alreadint(u_char *pack, unsigned len);
int alreadlen(unsigned char *pack, unsigned *lenlen);
int alreadoid(unsigned char *pack, int objidlen, struct oid *objid);
int alreadval(struct snval *val, int type, int vallen,
              unsigned char *pack);
int alwriteint(int val, u_char *buffp, int altype);
int alwritelen(unsigned char *pp, unsigned len);
int alwriteoid(unsigned char *packp, struct oid *oidp);
int alwriteval (struct snbentry *bl, u_char *bp);
void hashinit (void);
```

```
int hashoid(struct oid *oip);
struct mib_info *getmib(struct oid *oip);
int mksnmp(struct req_desc *rqdp, u_char *snmppack, u_char pdutype);
void sip2ocpy(u_short *oidp, IPaddr pip);
int sna2b (struct req_desc *rqdp);
int snb2a (struct req_desc *rqdp);
int snleaf (struct snbentry *bindl, struct mib_info *mip, int op);
int snparse(struct req_desc *rqdp, u_char *snmppack, int len);
int snrslv (struct req_desc *rqdp);
void so2ipcpy (u_char *pip, u_short *oidp);
int soipequ(u_short *oidp, IPaddr ip, int len);
int sntable(struct snbentry *bindI, struct mib_info *mip, int op);

/* IP Address Entry Table (ipAdEnt) * /
int sae_findnext (int iface, int numifaces);
int sae_get (struct snbentry *bindl, int numifaces);
int sae_getf(struct snbentry *bindl, struct mib_info *mip, int numifaces);
int sae_getn(struct snbentry *bindl, struct mib_info *mip, int numifaces);
int sae_match(struct snbentry *bindl, ...);
int sae_set (struct snbentry *bindl, struct mib_info *mip, int numifaces);

/* Interface Table access (if) */
int sif_get(struct snbe ntry *bindl, int numifaces);
iht sif_getf(struct snbe ntry *bindl, struct mib_info *mip, int numifaces);
int sif_getn(struct snbentry *bindl, struct mib_info *mip, int numifaces);
int sif_match(struct snbentry *bindl, ...);
int sif_set(struct snbentry *bindl, struct mib_info *mip, int numifaces);

/* ARP table access (at) */
int sat_findnext(int entry, int iface);
int sat_get(struct snbentry *bindl, int numifaces);
int sat_getf(struct snbentry *bindl, struct mib_info *mip, int numifaces);
iht sat_getn(struct snbentry *bindl, struct mib_info *mip), int numlfaces);
int sat_match(struct snbentry *bindl, ...);
int sat_set(struct snbentry *bindl, struct mib_info *mip, int numifaces);
int satcmp(struct arpentry *pael, struct arpentry *pae2);

/* IP NetToMedia access (nm)  */
int snm_findnext(int entry, int iface);
int snm_get(struct snbentry *bindl, int numifaces);
int snm_getf(struct snbentry *bindl, struct mib_info *mip, int numifaces);
int snm_getn(struct snbentry *bindl, struct mib_info *mip, int numifaces);
int snm_match(struct snbentry *bindl, ...);
int snm_set(struct snbentry *bindl, struct mib_info *mip, int numifaces);
int snmcmp(struct arpentry *pael, struct arpentry *pae2);

/* Route table access  (ipRoute)  */
```

```
struct route *srt_findnext(struct route *rtp,  int *rtl);
int srt_get(struct snbentry *bindl, int numifaces);
int srt_getf(struct snbentry *bindl, struct mib_info *mip, int numifaces);
int srt_getn(struct snbentry *bindl, struct mib_info *mip, int numifaces);
int srt_match(struct snbentry *bindl, ...);
int srt_set(struct snbentry *bindl, struct mib_info *mip,  int numifaces);

/* TCP connection table access (tcp) */
int stc_findnext(int tcbn);
int stc_get(struct snbentry *bindl, int numifaces);
int stc_getf(struct snbentry *bindl, struct mib_info *mip, int numifaces);
int stc_getn(struct snbentry *bindl, struct mjb_info *mip, int numifaces);
int stc_match(struct snbentry *bindl, ...);
int stc_set(struct snbentry *bindl, struct mib_info *mip), int numifaces);
int sntcpcmp (int tcbl, int tcb2 );

/* UDP connection table access (udp) */
int sud_findnext (int udpn);
int sud_get(struct snbentry *bindl, int numifaces);
int sud_getf(struct snbentry *bindl, struct mib_info *mip,  int numifaces);
int sud_getn(struct snbentry *bindl, struct mib_info *mip,  int numifaces);
int sud_match(struct snbentry *bindl, ...);
int sud_set(struct snbentry *bindl, struct mib_info *mip, int numifaces);

/* client access routines */
void sninit (void);
int snclient(struct req_desc *rqdp, char *fport,  int stdout);
void snerr(int stdout, struct req_desc *rqdp);
void snfreebl(struct snbentry **bl);

/* printing routines */
void snmpprint(int stdout, struct snbentry *bindl);
void snmpprint_val (int stdout, struct snval *val );
void snmpprint_objid(int stdout, struct oid *objidp);
void snmpprint_objname(int stdout, struct oid *oip);

/* printing macros * /
#define SOP2STR(op)  (op==SOP_GET?"GET" :op==SOP_GETF?"GETF" :\
op==SOP_GETN? "GETN": op==SOP_SET? "SET": "<unknown>" )
```

ASN.1为所有随SNMP报文发送来的对象定义了完整的表示法，包括变量名（对象标识符）、整型数值、序列、IP地址以及SNMP命令等。通常，ASN.1用类型（type）、长度（length）和值（value）来表示每个对象。其中类型指出了对象的类型，可分为整数、命令和计数器类型。长度指出了表示形式所占的八位组数，而值由包含了对象的八位组组成。

20.17 ASN.1表示形式的处理

文件asn1.h中包含对符号常量的定义，这些符号常量用于指定ASN.1类型，大多数从字面上可以理解。类型“序列”用于指出数据项中的重复（例如，一组整数），并被认为对应于程序设计语言中的数组结构。在不需要任何值时使用类型NULL。

```
/* asn1.h - A1_SIGNED */

/* constants for parsing an SNMP packet, according to ASN.1          */

/* ASN.1 object types */
#define ASN1_SEQ            0x30        /* sequence object            */
#define ASN1_INT            0x02        /* integer                    */
#define ASN1_OCTSTR         0x04        /* octet string               */
#define ASN1_NULL           0x05        /* null                       */
#define ASN1_OBJID          0x06        /* object identifier          */
#define ASN1_IPADDR         0x40        /* ip address                 */
#define ASN1_COUNTER        0x41        /* counter                    */
#define ASN1_GAUGE          0x42        /* gauge                      */
#define ASN1_TIMETICKS      0x43        /* time ticks                 */

/* Protocol Data Unit types -- SNMP specific                          */
#define PDU_GET             0xA0        /* get request                */
#define PDU_GETN            0xA1        /* get-next request           */
#define PDU_RESP            0xA2        /* response                   */
#define PDU_SET             0xA3        /* set request                */
#define PDU_TRAP            0xA4        /* trap message               */

/* Constants used for conversion of objects to/from ASN.1 notation    */
#define CHAR_BITS      8                /* number of bits per char    */
#define CHAR_HIBIT     0x80             /* octet with the high bit set */
#define BYTE2_HIBIT    0x8000           /* 2 bytes with high bit set  */
#define BYTE3_HIBIT    0x800000         /* 3 bytes with high bit set  */
#define BYTE4_HIBIT    0x80000000       /* 4 bytes with high bit set  */

#define A1_SIGNED(x)   ((x) == ASN1_INT)

/* the standard MIB prefix - 1.3.6.1.2.1                              */
extern char MIB_PREFIX[];

/* the standard MIB prefix is encoded by ASN.1 into 5 octets          */
#define MIB_PREF_SZ    5
```

20.17.1 长度表示法

如图20.4所示，ASN.1使用两种方式表示对象结构中的长度字段。如果对象所需的长度小于128八位组，ASN.1使用短格式，其中只用一个八位组来表示对象的长度。因为对小于128的8位

二进制数来说，其最高位为0，所以程序通过检查最高位来判断是否为短格式。

在长格式中，ASN.1使用多个八位组整数长度对长度进行编码。第一个八位组的最高位为1（以说明它为长格式），而低7位的内容是一个整数K（K>1）。后面的K个八位组中所包含的一个二进制整数指出了后面跟随的对象的长度。因此，为了提取长格式表示形式的长度值，程序首先读一个八位组，找出K的值，然后读入K个八位组的二进制数。函数alreadlen执行该操作。同一个文件中的函数alwritelen创建ASN.1的长度编码。它通过直接保存长度值来处理短格式，并为需要一或两个八位组的整数处理长格式（即，它仅能处理小于65536的整数）。将SNMP目标的长度限制在64K范围之内是合理的，因为更大的对象根本无法置入一个数据报中。

图 20.4　ASN.1 对象长度的编码。图（a）中的短格式被用于表示小于 128 的长度，而图（b）中的长格式被用于表示比 128 大的长度。根据第一个字节的最高位区分这两种格式

```
/* alrwlen.c - alreadlen, alwritelen */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <asn1.h>

/*------------------------------------------------------------------------
 * alreadlen - read and return the length of an ASN.1 encoded object
 *------------------------------------------------------------------------
 */
int alreadlen(pack, lenlen)
unsigned char     *pack;
unsigned          *lenlen;
{
        unsigned  totlen;
        unsigned  i;

        /* if the high bit is NOT set, then len is in short form */
        if (!((*pack) & CHAR_HIBIT)) {
             *lenlen = 1;
             return (*pack) & ~CHAR_HIBIT;   /* use only low bits */
        }
        /*
         * else, using long form where bit 7 = 1, and bits 6 - 0 encode
```

```
        * the number of subsequent octets that specify the length
        */
        *lenlen = (*pack++ & ~CHAR_HIBIT) + 1;

        for (i = 0, totlen = 0; i < (*lenlen) - 1; i++)
             totlen = (totlen << CHAR_BITS) | (int) *pack++;
        return totlen;
}

/*-----------------------------------------------------------------------
 * alwritelen - write the length of an object in ASN.1 encoded form
 *-----------------------------------------------------------------------
 */
int alwritelen(pp, len)          /* return number of bytes required */
unsigned char    *pp;
unsigned         len;
{
        /* if len < 128 then use short form */
        if (len < CHAR_HIBIT) {
             *pp = len;
             return 1;
        }
        /* use long form, where bit 7 = 1, and bits 6 - 0 encode the
             number of subsequent octets that specify the length */
        if (len <= 255) {
             *pp++ = CHAR_HIBIT | 1;
             *pp = len & 0xff;
             return 2;
        }
        /* else, assume len <= 65535 (2^16 - 1) */
        *pp++ = CHAR_HIBIT | 2;
        *pp++ = len >> CHAR_BITS;
        *pp = len & 0xff;
        return 3;
}
#endif  /* SNMP */
```

20.17.2 将整数转换为ASN.1格式

ASN.1就像表示变量的长度字段一样表示其他所有的数值。为了给一个整数编码，首先必须找出其有意义的(非0的)字节数，然后向编码字段中复制这么多字节的内容。函数alreadint和alwriteint完成该转换。

```
/* alrwint.c - alreadint, alwriteint */

#include <conf.h>
```

```
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <asn1.h>
#define h2asn    memcpy

/*-----------------------------------------------------------------------
 * alreadint - convert an ASN.1 encoded integer into a machine integer
 *-----------------------------------------------------------------------
 */
int alreadint(pack, len)
u_char      *pack;
unsigned    len;
{
        register int    tot;
        u_char          neg;
        unsigned        tlen;

        if ((tlen = len) > sizeof(int))
             return 0;
        tot = *pack & ~CHAR_HIBIT;
        neg = *pack & CHAR_HIBIT;
        for (tlen--, pack++ ; tlen > 0 ; tlen--, pack++)
             tot = (tot << CHAR_BITS) | (int) *pack;
        if (neg)
             tot -= (1 << ((len * CHAR_BITS) - 1));
        return tot;
}

/*-----------------------------------------------------------------------
 * alwriteint - convert an integer into its ASN.1 encoded form
 *-----------------------------------------------------------------------
 */
int alwriteint(val, buffp, altype)
int    val;
u_char *buffp;
int    altype;
{
        unsigned  tmp, numbytes;
        register u_char *bp;

        bp = buffp;
        tmp = val;
```

```
        if (Al_SIGNED(altype) && val < 0)
                tmp = -val;
        if (tmp < (unsigned) CHAR_HIBIT)
                *bp++ = numbytes = (u_char) 1;
        else if (tmp < (unsigned) BYTE2_HIBIT)
                *bp++ = numbytes = (u_char) 2;
        else if (tmp < (unsigned) BYTE3_HIBIT)
                *bp++ = numbytes = (u_char) 3;
        else if (tmp < (unsigned) BYTE4_HIBIT)
                *bp++ = numbytes = (u_char) 4;
        else {     /* 5 bytes for unsigned with high bit set */
                *bp++ = (u_char) 5;      /* length */
                *bp++ = (u_char) 0;
                numbytes = 4;
        }
        /* works for big endian & little-endian */
        switch (numbytes) {
            /* each case falls through to the next */
          case 4: *bp++ = (u_char)(0xff&((val&0xff000000)>>24));
          case 3: *bp++ = (u_char)(0xff&((val&0x00ff0000)>>16));
          case 2: *bp++ = (u_char)(0xff&((val&0x0000ff00)>>8));
          case 1: *bp++ = (u_char)(0xff&((val&0x000000ff)));
        }

        return bp - buffp;
}
#endif  /* SNMP */
```

20.17.3 将对象标识符转换为ASN.1格式

软件在内部使用结构oid（在文件snmp.h[1]中定义）来保存ASN.1对象标识符。过程alreadoid和alwriteoid完成从标准ASN.1格式到内部表示形式的转换及逆转换。

```
/* alrwoid.c - alreadoid, alwriteoid */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <asn1.h>

char MIB_PREFIX[] = { 0x2b, 0x6, 0x1, 0x2, 0x1 };   /* 1.3.6.1.2.1 */
```

① 文件snmp.h见20.16节。

```
/*-----------------------------------------------------------------------
 * alreadoid - convert an ASN.1 encoded object id into internal form
 *-----------------------------------------------------------------------
 */
int alreadoid(pack, objidlen, objid)
unsigned char     *pack;
int               objidlen;
struct oid        *objid;
{
        int  val;
        u_char     *pp;

        objid->len = 0;
        pp = pack;

        /* verify the required 1.3.6.1.2.1 prefix */
        if (memcpy(MIB_PREFIX, pp, MIB_PREF_SZ))
             return SYSERR;
        }
        pp += MIB_PREF_SZ;

        for (; pp < pack + objidlen; objid->len++) {
             if (! (*pp & CHAR_HIBIT)) {
                  objid->id[objid->len] = *pp++;
                  continue;
             }
             /*
              * using long form, where bits 6 - 0 of each
              * octet are used; (bit 7 == 0) ==> last octet
              */
             val = 0;
             do
                  val = (val << 7) | (int) (*pp & ~CHAR_HIBIT);
                  while (*pp++ & CHAR_HIBIT);    /* high bit set */
                  objid->id[objid->len] = val;
        }
        return OK;
}

/*-----------------------------------------------------------------------
 * alwriteoid - convert an object id into ASN.1 encoded form
 *-----------------------------------------------------------------------
 */
int alwriteoid(packp, oidp)
unsigned char     *packp;
```

```
struct oid        *oidp;
{
        register u_char *pp;
        int             i;
        u_char          *objidp, *lenp;

        pp = packp;
        lenp = pp++;    /* save location of objid len */
        objidp = pp;
        /* prepend the standard MIB prefix. */
        memcpy(pp, MIB_PREFIX, MIB_PREF_SZ);
        pp += MIB_PREF_SZ;

        for (i=0; i < oidp->len; i++)
                if (oidp->id[i] < CHAR_HIBIT)   /* short form */
                        *pp++ = oidp->id[i];
                else {                          /* long form */
                        if (oidp->id[i] >= (u_short) (BYTE2_HIBIT >> 1))
                                *pp++ = (u_char) (oidp->id[i] >> 14) |
                                        CHAR_HIBIT;
                        *pp++ = (u_char) (oidp->id[i] >> 7) | CHAR_HIBIT;
                        *pp++ = (u_char) (oidp->id[i] & ~CHAR_HIBIT);
                }
        *lenp = pp - objidp; /* assign the length of the objid */
        return pp - packp;
}
#endif  /* SNMP */
```

由于客户和服务器管理的所有对象标识符都以相同的前缀开始，因此软件通过在输入时删除前缀，而在输出时添加前缀的方法以提高效率。ASN.1表示形式使得前缀难以辨别，因为它将对象标识符的前两个数字形式的标号编码成为一个八位组。因此，前缀：

1.3.6.1.2.1

被编码成以43（十六进制的0x2b）[1]为第一个八位组的5个八位组串，后四个八位组的内容分别为6、1、2和1。

如果对象标识符中有一个数字形式的标号大于127[2]，则ASN.1使用扩展表示法来保存该值。每个八位组中只有7位是含有数据的。ASN.1使用最高位来表示子标识符的结束。其编码如图20.5所示。

① ASN.1使用表达式a*40+b将前两个子标识符a和b结合在一起

② 目前永久性赋值给MIB变量的ASN.1名字中没有一个数字标号值大于127。但是，某些表格中的对象标识符可能有更大一些的值。

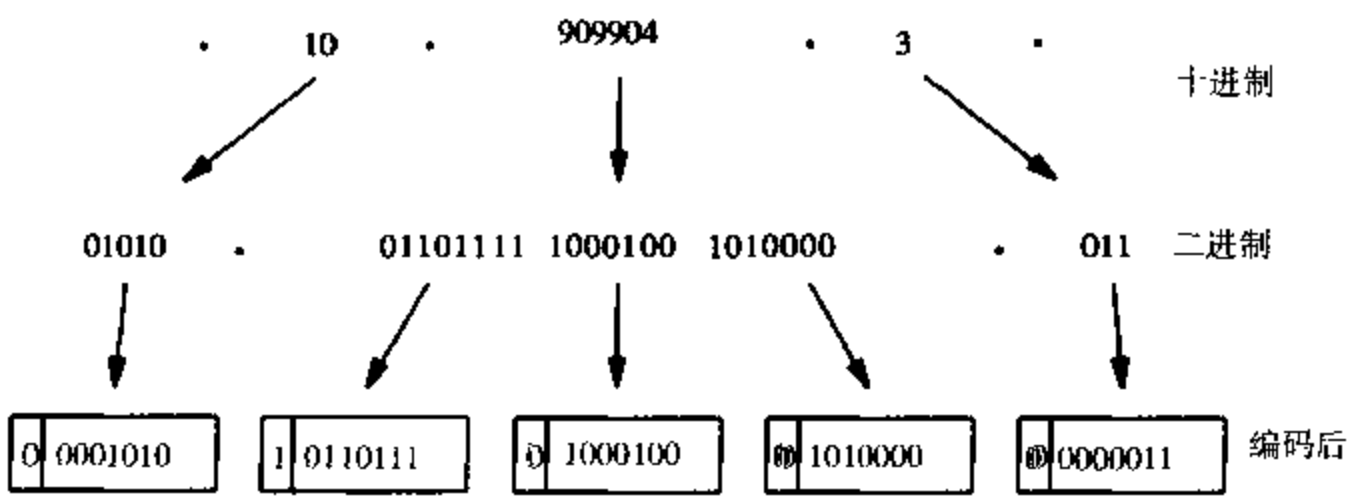


图 20.5　ASN.1 对象标识符的一部分，用十进制、二进制和编码后的格式表示。对于大于 127 的整数，ASN.1 编码法使用多个八位组。一个八位组的最高位为 0 标志首子标识符的结束

20.17.4　用于转换对象值的例程

前面已给出了用于内部表示法与在 SNMP 报文中使用的 ASN.1 表示法之间互相转换的低级例程，现在可以了解如何使用这些例程了。函数 alreadval 的一个入口参数指定 ASN.1 对象类型，另一个参数是指向对象值的指针。它利用该类型来决定应用哪一个转换例程来完成从 ASN.1 到内部格式的转换，而另一个与此相关的例程 alwriteval 完成输出时的转换。这两个例程都要调用本章前面所描述的那些完成转换操作的函数。

```
/* alrwval.c - alreadval, alwriteval */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <mem.h>

#ifdef SNMP

#include <snmp.h>
#include <asn1.h>

/*------------------------------------------------------------------------
 * alreadval - convert object in ASN.1 encoded form into internal form
 *------------------------------------------------------------------------
 */
int alreadval(val, type, vallen, pack)
struct snval      *val;
int               type;
int               vallen;
unsigned char     *pack;
{
        val->sv_type = type;

        switch (type) {
```

```
        case ASN1_INT:
        case ASN1_COUNTER:
        case ASN1_GAUGE:
        case ASN1_TIMETICKS:
             val->sv_val.sv_int = alreadint(pack, vallen);
             return OK;
        case ASN1_OCTSTR:
             val->sv_val.sv_str.sv_len = vallen;
             val->sv_val.sv_str.sv_str = (char *) pack;
             return OK;
        case ASN1_NULL:
             return OK;
        case ASN1_OBJID:
             return alreadoid(pack, vallen, &(val->sv_val.sv_oid));
        case ASN1_IPADDR:
             memcpy(&val->sv_val.sv_ipaddr, pack, vallen);
             return OK;
        default:
             return SYSERR;
        }
}

/*-----------------------------------------------------------------------
 * alwriteval - convert the value of a variable into ASN.1 equivalent.
 *-----------------------------------------------------------------------
 */
int alwriteval(bl, bp)
struct snbentry  *bl;
u_char      *bp;
{
        u_char     *origbp;

        origbp = bp;
        *bp++ = SVTYPE(bl);

        switch(SVTYPE(bl)) {
        case ASN1_INT:
        case ASN1_COUNTER:
        case ASN1_GAUGE:
        case ASN1_TIMETICKS:
             bp += alwriteint(SVINT(bl), bp, SVTYPE(bl));
             break;
        case ASN1_NULL:
             *bp++ = (u_char) 0;
             break;
        case ASN1_OCTSTR:
```

```
            bp += alwritelen(bp, SVSTRLEN(bl));
            memcpy(bp, SVSTR(bl), SVSTRLEN(bl));
            bp += SVSTRLEN(bl);
            freemem(SVSTR(bl), SVSTRLEN(bl));
            break;
        case ASN1_IPADDR:
            *bp++ = IP_ALEN;
            memcpy(bp, &SVIPADDR(bl), IP_ALEN);
            bp += IP_ALEN;
            break;
        case ASN1_OBJID:
            bp += alwriteoid(bp, &bl->sb_val.sv_val.sv_oid);
            break;
        default:
            break;
        }

        return bp - origbp;
}
#endif  /* SNMP */
```

20.18　小结

SNMP允许管理员询问和控制网关。管理员可以调用客户软件与一个或多个在远程网关上运行的SNMP服务器取得联系。SNMP服务器在概念上维持着一组变量，这些变量与网关中的TCP/IP数据结构一一对应，SNMP应用“取/存”模式对这些变量进行操作。

管理信息库（MIB）规定了网关必须维持的一组变量。SNMP应用ASN.1语法来表示SNMP报文，并且用ASN.1对象标识符为MIB变量命名。当服务器收到一个报文时，它必须将ASN.1变量名的数字表示形式映射到保存相应值的本地变量中。我们的实现方案中用一个数组来保存MIB变量，而利用散列表来提高对该数组的查找速度。另外，因get-next-request操作的需要，它还利用一组指针维护了变量名之间的字典顺序。

我们详述了将变量值从SNMP报文使用的ASN.1表示形式到网关使用的本地格式之间的转换函数。虽然ASN.1语法有些冗长，但并不难理解。

深入研究

Case、Fedor、Schoffstall和Davin［RFC 1157］中包含了SNMP标准。ISO［May 87a］和［May 87b］中不但有ASN.1标准，且具体规定了编码方法。McCloghrie和Rose［RFC 1213］给出了MIB规范，这也就是著名的MIB Ⅱ，Rose和McCloghrie［RFC 1212］说明了规范中所用的语言，而McCloghrie和Rose［RFC 1155］中包含了为MIB变量命名的SMI（Structure of Management Information）规则。

习题

20.1 假设当前的一组MIB变量是按顺序从1到122编号的，而不是赋值为ASN.1对象标识符，可省略多少程序代码？

20.2 比较ASN.1分级命名方案和上一题中假设的编号方案。它们各有什么优缺点？

20.3 在文件snmib.c中定义的、为维护MIB变量之间的字典顺序而设置的指针有什么优点及缺点？

20.4 考虑在文件snmp.h中定义的宏oidequ。为什么它要非常明确地检查对象长度？

20.5 阅读协议规范，找出community字段代表什么？为什么服务器在该字段中发送的和希望接收的值为public？

20.6 阅读协议标准，ASN.1对象标识符的前两个标号是如何被编码成一个单个的八位组？为什么ASN.1要规定这样的编码法？

20.7 请解释为什么像alreadint这样的过程固有地依赖于设备。对不同的体系alreadint可能要做哪些变动？

第21章 SNMP：客户与服务器

21.1 引言

上一章讲述SNMP软件，并描述了在查找时使用的数据结构，以及从MIB变量使用的ASN.1名字绑定为本地网关中的变量名时使用的数据结构。本章给出SNMP服务器的实现方案，并详述它是如何分析处理SNMP报文的。另外，本章还考察一个SNMP客户，它可以与服务器共同享用许多实用过程。下一章将集中讲述如何实现在特定表格上的存取操作，以继续我们的讨论。

21.2 服务器中数据的表示形式

从理论上讲，SNMP服务器很简单。它由一个单一的进程组成，该进程反复不断地等待输入的报文，执行指定的操作，并返回结果。报文中最重要的部分是：一系列的get、set和get-next[1]请求，以及每个请求执行的ASN.1对象标识符。服务器在请求序列中找出每一个对象标识符，并逐一地完成指定的对象的操作。

在实际中，因许多细节问题使程序代码变得复杂。SNMP报文中所有字段都使用ASN.1表示形式，因此服务器不能使用固定的结构来描述报文格式。事实上，它必须检查整个报文，并逐步分析每个字段。另外，因为每个值都以ASN.1格式表示，所以服务器还必须将它们转换为适当的内部格式。

为简化实现取或存操作的各个过程，我们的服务器将诸如整型之类的数值采用本地机器自身的编码形式表示。服务器分析每个传入报文，并将字段从ASN.1表示形式转换为相应的内部数据结构。服务器处理内部格式，并在发送应答分组之前，将其转换回所需的ASN.1表示形式。我们总结如下：

> 为了优化性能，SNMP服务器将传入报文从ASN.1表示形式转换为内部的固定字段的格式，以机器自身的硬件表示形式来保存大多数的值。它使用内部格式完成各种操作，并在发送应答分组之前，将其转换回ASN.1表示形式。

21.3 服务器的实现

过程snmpd实现SNMP服务器的主要算法。文件snmp.h[2]中包含了所需数据结构的说明。

```
/* snmpd.c - snmpd */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```

① 在今后的章节中，我们将get-request、set-request、get-next-request分别简写为get、set和get-next。

② 参见20.16节的文件snmp.h。

```
#include <mem.h>
#include <io.h>

#ifdef SNMP

#include <snmp.h>
#include <asn1.h>

/*-----------------------------------------------------------------------
 * snmpd - open the SNMP port and handle incoming queries
 *-----------------------------------------------------------------------
 */
int snmpd()
{
        int             snmpdev, len;
        struct xgram    *query;
        struct req_desc rqd;
        int             reqtype;

        sninit();
        rqd.bindlf = (void*) NULL;
        query = (struct xgram *) getmem(sizeof (struct xgram));
        /* open the SNMP server port */
        if ((snmpdev = open(UDP, ANYFPORT, (void *)SNMPPORT)) == SYSERR)
             return SYSERR;
        while (TRUE) {
             /*
              * In this mode, give read the size of xgram, it returns
              * number of bytes of *data* in xgram.
              */
             len = read(snmpdev, query, sizeof(struct xgram));
             ++snmpInPkts;
             /* parse the packet into the request desc. structure */
             if (snparse(&rqd,
                  (u_char *)(query->xg_data, len) == SYSERR) {
                  snfreebl(&rqd.bindlf);
                  continue;
             }
             /* convert ASN.1 representations to internal forms */
             if (sna2b(&rqd) == SYSERR) {
                  snfreebl(&rqd.bindlf);
                  ++snmpInASNParseErrs;
                  continue;
             }
             if (snrslv(&rdq) == SYSERR) {
                  /* canberra bugfix, mksnmp cleans up dynamic */
```

```
                /* memory that would otherwise be lost */
                query->xg_data[rqd.err_stat_pos] = rqd.err_stat;
                query->xg_data[rqd.err_idx_pos] = rqd.err_idx;
                len = mksnmp(&rqd, (u_char *)query->xg_data,
                             PDU_RESP);
                ++snmpOutPkts;
                if (rdq.err_stat == SERR_TOO_BIG)
                    ++snmpOutTooBigs;
                if (rdq.err_stat == SERR_NO_SUCH)
                    ++snmpOutNoSuchName;
                if (rdq.err_stat == SERR_BAD_VALUE)
                    ++snmpOutBadValues;
                if (write(snmpdev, query, len) == SYSERR)
                        return SYSERR;
                snfreebl(&rqd.bindlf);
                continue;
        }
        reqtype = rqd.reqtype;
        len = mksnmp(&rqd, (u_char *)query->xg_data, PDU_RESP);
        if (len == SYSERR) {
                query->xg_data[rqd.pdutype_pos] = (u_char)PDU_RESP;
                query->xg_data[rqd.err_stat_pos] = rqd.err_stat;
                query->xg_data[rqd.err_idx_pos] = rqd.err_idx;
                ++snmpOutPkts;
                if (rdq.err_stat == SERR_TOO_BIG)
                    ++snmpOutTooBigs;
                if (rdq.err_stat == SERR_NO_SUCH)
                    ++snmpOutNoSuchNames;
                if (rdq.err_stat == SERR_BAD_VALUE)
                    ++snmpOutBadValues;
                if (write(snmpdev, query, len) == SYSERR)
                        return SYSERR;
                snfreebl(&rqd.bindlf);
                continue;
        }
        ++snmpOutPkts;
        switch (reqtype) {
          case PDU_GETN:
              ++snmpInTotalGetVars;
              ++snmpInGetNexts;
              break;
          case PDU_GET:
              ++snmpInTotalReqVars;
              ++snmpInGetRequests;
              break;
```

```
                case PDU_SET:
                    ++snmpInTotalSetVars;
                    ++snmpInSetRequests;
                    break;
            }
            if (write(snmpdev, query, len) == SYSERR) {
                return SYSERR;
            }
            snfreebl(&rqd.bindlf);
        }
}
#endif  /* SNMP */
```

snmpd首先打开一个SNMP使用的UDP端口（程序代码中的常量SNMPPORT），然后进入一个无限循环，此时它调用read等待下一个传入的报文。当一个SNMP报文到达时，snmpd调用snparse分析该报文，将其转换为内部格式，并保存到请求描述块结构（req_desc）中。除了提取首部中的字段之外，snmpd调用函数sna2b从报文中提取对象标识符序列，并把它们转换到一个链表中。链表中的每个结点对应于一个绑定，这些结点被定义为文件snmp.h中的snbentry结构。在转换时，sna2b把每个ASN.1对象标识符转换成内部表示形式。

一旦报文和变量名序列都已转换为内部格式之后，snmpd调用snrslv解决报文中提出的请求，包括为链表中的每个对象标识符执行指定的get、set或get-next操作。在完成这一工作后，snmpd调用mksnmp生成应答报文，并用write操作向客户发送应答报文。服务器发送了一个应答报文后，它调用snfreebl释放变量名链表，并返回到主循环的头部，重新等待下一个输入的报文。

如果因出现差错而无法顺利完成上述工作，则服务器生成一个差错应答报文，该报文中含有差错类型代码（error type code）和差错索引（error index）。差错类型码给出差错发生的原因，而差错索引指出在请求中导致差错的变量名。

21.4 对SNMP报文的分析

函数snparse提取SNMP报文中的字段，并解析这些字段。由于ASN.1表示法允许每个字段都具有可变的长度，因此这项工作十分繁琐。负责分析工作的进程必须利用一个指针遍历报文中的所有字段，找出字段长度并提取字段值。

```
/* snparse.c - snparse */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <mem.h>
#include <string.h>

#ifdef SNMP

#include <snmp.h>
```

```
#include <asn1.h>
/*-----------------------------------------------------------------------
 * snparse - convert the ASN.1-encoded SNMP packet into internal form
 *   N.B.   the binding list rqdp->bindlf must either be NULL or a valid
 *          binding list, in which case I'll free it up.
 *-----------------------------------------------------------------------
 */
int snparse(rqdp, snmppack, len)
struct req_desc  *rqdp;
u_char           *snmppack;
int              len;
{
        struct snbentry *bl, *lastbl = 0;
        register u_char *packp;
        int             totpacklen = 0, commlen = 0;
        int             pdulen, totbindlen = 0;
        u_int           lenlen;
        int             varbindlen = 0;
        u_char          *packendp;

        packp = snmppack;
        packendp = snmppack + len;

        /* sequence operator and total packet length */
        if (*packp++ != ASN1_SEQ ||
            (totpacklen = alreadlen(packp, &lenlen)) < 0) {
             return SYSERR;
        }
        packp += lenlen;

        /* verify total length, version, community */
        if (packendp != packp + totpacklen ||
            *packp++ != ASN1_INT ||
            *packp++ != SVERS_LEN ||
            *packp++ != SVERSION ||
            *packp++ != ASN1_OCTSTR ||
            (commlen = alreadlen(packp, &lenlen)) < 0) {
             return SYSERR;
        }
        packp += lenlen;
        if (strncmp((char *)packp, SCOMM_STR, commlen) != 0) {
             return SYSERR;
        }
        packp += commlen;
```

```
/* PDU type and length */
if (*packp == PDU_TRAP)
     return SYSERR;
rqdp->pdutype_pos = packp - snmppack;
rqdp->reqtype = *packp++;
if ((pdulen = alreadlen(packp, &lenlen)) < 0) {
     return SYSERR;
}
packp += lenlen;

/* verify PDU length */
if (packendp != packp + pdulen)
     return SYSERR;
}
/* request id */
if (*packp++ != ASN1_INT ||
    (rqdp->reqidlen = alreadlen(packp, &lenlen)) < 0) {
     return SYSERR;
}
packp += lenlen;
memcpy(rqdp->reqid, packp, rqdp->reqidlen);
packp += rqdp->reqidlen;

/* error status */
if (*packp++ != ASN1_INT || *packp++ != 1) {
     return SYSERR;
}
rqdp->err_stat = *packp;
rqdp->err_stat_pos = packp++ - snmppack;

/* error index */
if (*packp++ != ASN1_INT || *packp++ != 1) {
     return SYSERR;
}
rqdp->err_idx = *packp;
rqdp->err_idx_pos = packp++ - snmppack;

/* sequence of variable bindings */
if (*packp++ != ASN1_SEQ ||
    (totbindlen = alreadlen(packp, &lenlen)) < 0) {
     return SYSERR;
}
packp += lenlen;

/* canberra bug fix - rqdp might already have old dynamic */
/* memory attached, free before overwriting pointers */
```

```
    snfreebl(&rqdp->bindlf);
    /* verify variable bindings length */
    if (packendp != packp + totbindlen)
        return SYSERR;
    /* build doubly-linked bindings list; fill in only sb_alstr's  */
    rqdp->bindlf = rqdp->bindle = (struct snbentry *) NULL;
    do {
        bl = (struct snbentry *) getmem(sizeof(struct snbentry));
        bl->sb_next = 0;
        bl->sb_prev = 0;
        if (rqdp->bindlf) {
            lastbl->sb_next = bl;
            bl->sb_prev = lastbl;
            lastbl = bl;
        } else
            lastbl = rqdp->bindlf = bl;
        bl->sb_alstr = packp;
        bl->sb_aldynstr = FALSE; /* not dynamically allocated */
        if (*packp++ != ASN1_SEQ ||
            (varbindlen - alreadlen(packp, &lenlen)) < 0) {
            return SYSERR;
        }
        packp += lenlen;
        bl->sb_alslen = varbindlen;
        packp += varbindlen;
    } while (packp < packendp);
    /* check that the entire packet has now been parsed */
    if (packp != packendp) {
        return SYSERR;
    }
    rqdp->bindle = lastbl;
    return OK;
}
#endif  /* SNMP */
```

SNMP报文以一个序列操作码和报文的总长度作为起始。如果该长度与数据报长度不符，snparse返回一个差错码。接着，snparse验证第二个字段中的整数是否包含正确的SNMP版本号。第三个字段被称作“团体”（community），用于验证该报文的身份。我们的实现范例只承认值为字符串public的“团体”，这是服务器使用的、不要求客户鉴别自己的标准值。

snparse还要确定报文的第四个字段没有指定“陷阱”（trap）操作（即一个get、set或get-next请求），因为陷阱报文是由服务器，而不是客户生成的。

snparse检查第五个字段，以确保它正确地给出了该报文剩余部分的长度。接着从第六个字段中提取请求标识符。它还要验证第七个和第八个字段中的差错状态和差错索引值都为0。

在检查了差错状态和差错索引字段之后，snparse开始处理绑定序列。不论是在请求分组中，还

是在响应分组中，每个绑定都由一对数据组成。在get请求或get-next请求中，每个绑定含有一个变量名（ASN.1对象标识符）和一个关联值NULL。在响应中，服务器用客户所请求的变量值取代NULL。在set请求中，绑定为每个变量名提供一个非0的值，服务器将这些非0值赋予指定的变量。

snparse遍历绑定序列，并为每个绑定分配一个snbentry结点。每个结点中含有一个指针，指向源报文中的、用ASN.1表示法表示的对象标识符。这些结点被链接入一个链表中，链表的头指针和尾指针都在该报文的req_desc结构中，如图21.1所示。

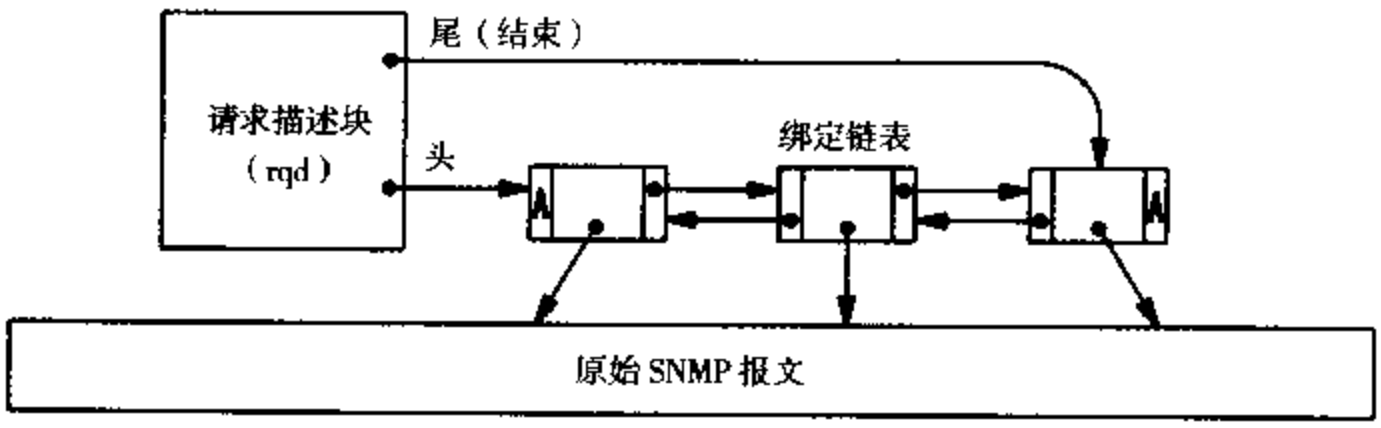


图21.1 由snparse创建的数据结构，绑定链表的每个结点对应于一个MIB变量。结点指针指向原报文中的一个ASN.1对象标识符

一旦snparse从报文中提取了所有的字段后，它还要验证报文中再没有别的剩余八位组了。

21.5 绑定链表中ASN.1名字的转换

当snparse从输入的SNMP报文中提取出所有的字段之后，请求描述块结构req_desc中就有了内容。为了便于访问，服务器调用函数sna2b将报文中的对象标识符和对象值转换为固定长度的内部表示形式。sna2b遍历绑定链表，并为每个对象进行ASN.1语法转换。snparse调用alreadoid提取对象标识符的后缀，并将其保存在结点的sb_oid字段中。它还要从原报文中提取相关的对象值，并将其保存在结点的sb_val字段中。

```
/* sna2b.c - sna2b */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <asn1.h>

/*------------------------------------------------------------------------
 * sna2b - convert an ASN.1 encoded binding into internal form
 *------------------------------------------------------------------------
 */
int sna2b(rqdp)
struct req_desc  *rqdp;
```

```
{
        register u_char *sp;
        struct snbentry *bl;
        int             lenlen, objidlen, vallen;
        u_char          type;

        for (bl = rqdp->bindlf; bl; bl = bl->sb_next) {
             sp = bl->sb_alstr;
             /* match the sequence operator and length of bindings */
             if (*sp++ != ASN1_SEQ || alreadlen(sp, (unsigned *)&lenlen) < 0) {
                  return SYSERR;
             }
             sp += lenlen;
             /* object identifier type, length, objid */
             if (*sp++ != ASN1_OBJID)
                  return SYSERR;
             if((objidlen = alreadlen(sp, (unsigned *)&lenlen)) < 0) {
                  return SYSERR;
             sp += lenlen;
             if (alreadoid(sp, objidlen, &bl->sb_oid) == SYSERR) {
                  return SYSERR;
             }
             sp += objidlen;
             /* object's value */
             type = *sp++;
             if ((vallen = alreadlen(sp, (unsigned *)&lenlen)) < 0) {
                  return SYSERR;
             }
             sp += lenlen;
             if (alreadval(&bl->sb_val, type, vallen, sp) == SYSERR) {
                  return SYSERR;
             }
             sp += vallen;
             bl->sb_alslen = 0;
        }
        return OK;
}
#endif  /* SNMP */
```

21.6 解析一个请求

只要一个输入的报文被成功地转换为内部格式后，服务器调用 snrslv 对其进行解析。

```
/* snrslv.c - snrslv */
```

```
#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/* Set the error status and error index in a request descriptor. */
#define seterr(errval)      rqdp->err_stat = errval;   \
                            rqdp->err_idx - i;

/*------------------------------------------------------------------------
 * snrslv - resolve the list of specified variable bindings
 *------------------------------------------------------------------------
 */
int snrslv(rqdp)
struct req_desc *rqdp;
{
        struct     snbentry *bl;
        struct     mib_info *np;
        int        i, op, err;

        for (bl = rqdp->bindlf, i = 1; bl; bl = bl->sb_next, i++) {


            /* special case, getNext on 0 (or NULL) */
            /* returns getfirst on 1 */
            if (((bl->sb_oid.len == 0) ||
                ((bl->sb_oid.len == 1) &&
                 (bl->sb_oid.id[0] == 0))) &&
               (rqdp->reqtype == PDU_GETN)) {
               bl->sb_oid.len = 1;
               bl->sb_oid.id[0] = 1;
            }

            /* use getmib to look up object id */
            if ((np = getmib(&bl->sb_oid)) == 0) {
                 seterr(SERR_NO_SUCH);
                 return SYSERR;
            }
            /* call function to apply specified operation        */
            if (np->mi_func == 0) { /* objid is an aggregate     */
                 /* only getnext allows nonexistent names  */
```

```
                if (rqdp->reqtype != PDU_GETN) {
                        seterr(SERR_NO_SUCH);
                        return SYSERR;
                }
                /* use getfirst for getnext on an aggregate */
                if (err = ((*np->mi_next->mi_func)
                        (bl, np->mi_next, SOP_GETF))) {
                        seterr(err);
                        return SYSERR;
                }
        } else { /* function in table ==> single item or table    */
                switch (rqdp->reqtype) {
                case PDU_GET:   op = SOP_GET;   break;
                case PDU_GETN:  op = SOP_GETN;  break;
                case PDU_SET:   op = SOP_SET;   break;
                default:
                    return(SYSERR);
                }
                /* use getfirst for getnext on table entry      */
                if (oidequ(&bl->sb_oid, &np->mi_objid) &&
                    np->mi_vartype == T_TABLE) {
                        if (op == SOP_GETN)
                            op = SOP_GETF;
                }
                if (err = ((*np->mi_func)(bl, np, op))) {
                        seterr(err);
                        return SYSERR;
                }
        }
    }
    return OK;
}
#endif  /* SNMP */
```

解析包括以绑定链表中的每个变量为对象，执行请求中指定的操作。因此，snrslv遍历绑定链表。它调用getmib在MIB表中查找每一个对象标识符。如果没有找到匹配项，则snrslv返回差错代码。

一旦它在MIB表中找到了匹配项，snrslv检查该表项的字段mi_func。如果字段mi_func的内容是NULL，则这个表项对应的是一个聚合体（aggregate），而不是变量。由于聚合体名字只能用在get-next请求中，因此，对于不是get-next的请求，snrslv返回差错代码。如果字段mi_func的内容不是NULL，则它包含的是一个函数的地址，并由该函数解释对变量进行何种操作。

21.7 对get-next操作的解释

在get-next请求的情况下，snrslv将操作变换为get-first，并以按字典顺序排列的下一个MIB表项为对象，执行get-first操作。get-first操作是实现中的一部分，而不是SNMP协议标准的一部分。为了有助于理解为什么会出现get-first这样的操作，可以仔细地考虑一下对get-next操作的解释。当get-next请求给出一个对象标识符时，服务器必须对按字典顺序紧跟在指定的变量名后的MIB变量执行取操作。如果服务器利用MIB表中的字典顺序指针来找出下一个变量，然后试图对这个新变量继续执行get-next操作，其结果是无限地循环遍历。另一方面，如果服务器在MIB表中查找出指定的变量，利用字典顺序指针找出下一个表项，然后执行get操作，如果按字典顺序排列的下一个表项恰好对应于一张空表格，而不是一个变量，操作就会出现问题，因此服务器应当通过指针找出一个简单变量。范例程序代码中的get-first操作恰好可以解决上述问题。当对象是简单变量时，get-first和get操作完全一样，而对聚合体进行操作时，get-first和get-next操作完全一样。总结如下：

> get-first操作可为任意一个按字典顺序排序的MIB表项提供get-next操作，它先找出MIB中按字典顺序排列的下一个变量，并以该变量为对象，执行get操作。

为了理解snrslv如何应用get-first操作，再仔细研究一下程序代码。如果给出的变量名对应于一个聚合体，snrslv会发现字段mi_func为空，因而它以按字典顺序排列的下一个MIB表项为对象，执行get-first操作。如果snrslv发现一张表格（mi_vartype中的内容为T_TABLE），它在调用函数以实现操作之前，先将get-next操作改为get-first操作。

21.8 操作的间接执行

我们已经知道snrslv要根据mib数组来决定它调用什么样的函数对指定的变量完成某种操作。它向该函数传递一个操作码，一个指向绑定链表中某个结点的指针，一个指向对应于该变量的mib表项的指针。使用这种间接方法是为了避免重复，因为若为每个MIB变量各创建一组函数，会带来大量的重复。事实上，少数几个函数就可以容纳所有用于差错检查和完成共同的操作的程序代码。例如，函数snleaf可处理以简单变量为对象的各种操作。

```
/* snleaf.c - snleaf */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <mem.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>
```

```
/*-----------------------------------------------------------------------
 * snleaf - perform the requested operation on the leaf SNMP variable
 *-----------------------------------------------------------------------
 */
int snleaf(bindl, mip, op)
struct snbentry *bindl;
struct mib_info *mip;
int                op;
{
        int             len;
        char            *strp;
        struct oid      *oip;

        if (op == SOP_GETN) {
            if (mip->mi_next)
                return((*mip->mi_next->mi_func)
                        (bindl, mip->mi_next, SOP_GETF));
            return SERR_NO_SUCH;
        }
        if (op == SOP_SET) {
            if (! mip->mi_writable)
                return SERR_NO_SUCH;
            switch(mip->mi_vartype) {
            case ASN1_INT:
                if (SVTYPE(bindl) != ASN1_INT)
                    return SERR_BAD_VALUE;
                if (mip->mi_param == 0)
                    return SERR_NO_SUCH;
                *((int *) mip->mi_param) = SVINT(bindl);
                break;
            case ASN1_OCTSTR:
                if (SVTYPE(bindl) != ASN1_OCTSTR)
                    return SERR_BAD_VALUE;
                strp = (char *)mip->mi_param;
                memcpy(strp, SVSTR(bindl), SVSTRLEN(bindl));
                *(strp + SVSTRLEN(bindl)) = '\0';
                break;
            case ASN1_OBJID:
                if (SVTYPE(bindl) != ASN1_OBJID)
                    return SERR_BAD_VALUE;
                op = (struct oid *)mip->mi-param;
                oip->len = SVSTRLEN(bindl);
                memcpy(oip->id, SVSTR(bindl), oip->len * 2);
                break;
            }
```

```
        return SNMP_OK;
    }
    if (op == SOP_GETF) {
        /* put the correct objid into the binding list. */
        bindl->sb_oid.len = mip->mi_objid.len;
        memcpy(bindl->sb_oid.id, mip->mi_objid.id,
            mip->mi_objid.len * 2);
    }
    SVTYPE(bindl) = mip->mi_vartype;

    switch(mip->mi_vartype) {
    case ASN1_INT:
    case ASN1_TIMETICKS:
    case ASN1_GAUGE:
    case ASN1_COUNTER:
        SVINT(bindl) = *((int *) mip->mi_param);
        break;
    case ASN1_OCTSTR:
        strp = mip->mi_param;
        if (strp == NULL) {
            SVSTRLEN(bindl) = 0;
            SVSTR(bindl) = NULL;
            break;
        }
        len = SVSTRLEN(bindl) = strlen(strp);
        SVSTR(bindl) = (char *) getmem(len);
        memcpy(SVSTR(bindl), strp, len);
        break;
    case ASN1_OBJID:
        oip = (struct oid *) mip->mi_param;
        SVOIDLEN(bindl) = oip->len;
        memcpy(SVOID(bindl), oip->id, oip->len * 2);
        break;
    case ASN1_OBJID:
        oip = (struct oid *)mip->mi_param;
        SVOIDLEN(bindl) = oip->len;
        memcpy(SVOID(bindl), oip->id, oip->len * 2);
    }
    return SNMP_OK;
}
#endif  /* SNMP */
```

snleaf通过检查操作码参数来选择某种动作。它对按字典顺序排列的下一个MIB表项调用get-first，以完成get-next操作。snleaf通过把绑定链表结点中的变量值存放到内存中适当的变量中来完成set操作（mib表中的字段mi_param给出了内存地址）。它通过将内存中某个变量的值复制到绑定

链表结点中来完成 get（以及 get-first）操作。

21.9　表格的间接寻址

在mib数组中，对应于表格的表项，利用字段mi_param保存一个指针，指向该表格对应的tabtab表项。某个表格的 tabtab 表项含有以该表格为对象，用以实现各种 SNMP 操作的函数。因此过程 sntable 利用 tabtab 表项和指定的操作来选择并调用某个函数，实现对指定表格的操作。

```
/* sntable.c - sntable */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>

/*------------------------------------------------------------------------
 * sntable - call function to operate on a table-embedded variable
 *------------------------------------------------------------------------
 */
int sntable(bindl, mip, op)
struct snbentry  *bindl;
struct mib_info  *mip;
int              op;
{
      struct tab_info *pti = (struct tab_info *)mip->mi_param;
      int             numifaces = Net.nif - 1;

      /*
       * mip->mi_param holds a pointer to an entry in tabtab that
       * contains the pointers to functions for each table
       */
      switch (op) {
      case SOP_GET:
            return pti->ti_get)(bindl, numifaces);
      case SOP_GETF:
            return pti->ti_getf)(bindl, mip, numifaces);
      case SOP_SET:
            return pti->ti_set)(bindl, mip, numifaces);
      case SOP_GETN:
            return pti->ti_getn)(bindl, mip, numifaces);
      }
```

```
        return SYSERR;
}
#endif  /* SNMP */
```

21.10 应答报文的反向生成

一旦服务器成功地对请求报文中的所有项目都进行了解析之后，就生成一个应答报文，并将应答报文返回客户。应答报文的格式与请求报文的格式相同，每个字段均使用ASN.1表示形式。该表示形式要求首部字段中含有报文的长度，但是只有在计算得出了所有字段的ASN.1表示形式之后，才有可能知道报文的总长度。再者，由于报文长度字段本身的长度取决于报文中其余字段的长度，因此在生成应答报文时，不可能事先得知需要为长度字段预留多大的空间。

为了简化报文的构造，范例程序代码通过反向生成应答报文，避免了由于不知道长度而引起的问题。它按逆序生成报文中的字段，并且在各个字段中，它也是以逆序生成每个八位组的。因此，一旦生成了完整的报文，只需要将其反转回正序再进行传送即可。总结如下：

> 由于报文中所有的字段，包括长度字段本身，都具有可变长度的格式，因而ASN.1表示形式使报文的生成变得困难。为了简化构造，报文以逆序生成，并在传送前反转回正序。

过程mksnmp既处理请求报文的创建，也处理应答报文的生成，并且无论是客户还是服务器都可以调用它。

```
/* mksnmp.c - mksnmp */

#include <conf.h>
#include <kernel.h>
#include <sleep.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <asn1.h>

char SNVCBACK[] = SNVCBACK_CHARS;
#define SNVCLEN        sizeof(SNVCBACK) /* length of SNVCBACK


#define SNMAXHLEN 32   /* length of a "maximum" SNMP header */

u_char snmpbuff[SNMPMAXSZ]; /* global scratch buffer        */

/*------------------------------------------------------------------------
 * mksnmp - make an snmp packet and return its length
 *------------------------------------------------------------------------
 */
int mksnmp(rqdp, snmppack, pdutype)
```

```
struct req_desc  *rqdp;
u_char           *snmppack;
u_char           pdutype;
{
        register u_char      *pp, *cp;
        struct     snbentry  *bl;
        u_char               tmpbuff[40];
        int                  len, mtu, estlen;

        pp = snmpbuff;
        if (rqdp->reqidlen == 0) {  /* if id len == 0, get new reqid   */
             memcpy(rqdp->reqid, (char *) &clktime, sizeof(clktime));
             rqdp->reqidlen = sizeof(clktime);
        }
        snb2a(rqdp);    /* convert bindings to ASN.1 notation          */

        /* check total length of the packet to be created          */
        mtu = IP_MAXLEN;

        /* add up total length of ASN.1 representations of variables   */
        for (estlen=0, bl=rqdp->bindlf; estlen<mtu && bl; bl=bl->sb_next)
             estlen += bl->sb_alslen;
        /*
         * if too long, or if adding the header makes it too long,
         * set error status to tooBig and return
        */
        if (bl || (estlen + SNMAXHLEN >= mtu)) {
             rqdp->err_stat = SERR_TOO_BIG;
             rqdp->err_idx = 0;
             return SYSERR;
        }
        /* go backwards through snbentry, writing out the bindings     */
        for (bl=rqdp->bindle; bl; bl=bl->sb_prev) {
             cp = &bl->sb_alstr[bl->sb_alslen-1];
             while (cp >= bl->sb_alstr)
                  *pp++ = *cp--;
        }
        /* write the length of the bindings and an ASN1_SEQ type  */
        len = alwritelen(tmpbuff, pp - snmpbuff);
        for (cp = &tmpbuff[len-1]; cp >= tmpbuff; )
             *pp++ = *cp--;
        *pp++ = ASN1_SEQ;

        /* write the error index and error status -- 1 byte integers */
        *pp++ = (u_char) rqdp->err_idx;
        *pp++ = (u_char) 1;
        *pp++ = ASN1_INT;
```

```
        *pp++ = (u_char) rqdp->err_stat;
        *pp++ = (u_char) 1;
        *pp++ = ASN1_INT;

        /* write the request id, its length, and its type */
        for (cp = &rqdp->reqid[rqdp->reqidlen-1]; cp >= rqdp->reqid; )
             *pp++ = *cp--;
        *pp++ = rqdp->reqidlen;
        *pp++ = ASN1_INT;

        /* write the packet length and pdutype */
        len = a1writelen(tmpbuff, pp - snmpbuff);
        for (cp = &tmpbuff[len-1]; cp >= tmpbuff; )
             *pp++ = *cp--;
        *pp++ = pdutype;

        /* write the community and the version */
        memcpy(pp, SNVCBACK, SNVCLEN);
        pp += SNVCLEN;

        /* write the total packet length */
        len = a1writelen(tmpbuff, pp - snmpbuff);
        for (cp = &tmpbuff[len-1]; cp >= tmpbuff; )
             *pp++ = *cp--;
        *pp++ = ASN1_SEQ;

        /* reverse the entire finished packet */
        for (--pp, cp = snmppack; pp >= snmpbuff; )
             *cp++ = *pp--;
        return cp - snmppack;
}
#endif  /* SNMP */
```

mksnmp首先检查报文的请求标识符字段（reqid）。在响应中，标识符的值和客户在请求中发送的值相同。对于由客户生成的请求报文，该标识符字段的值应当是0，因此mksnmp利用当日时间生成具有惟一性的标识符。

在检查了报文的请求标识符之后，mksnmp调用snb2a将绑定链表中所有对象标识符转换为ASN.1格式。然后它将转换时计算出的长度加上报文首部长度的估计值，得到报文总长度的估计值。如果总长度的估值超过最大UDP数据报缓冲区长度，mksnmp返回差错代码。

输出报文的形成与输入报文的确定是并行的。mksnmp从请求描述块结构req_dese中取出固定字段，并将它们转换为ASN.1表示形式。最后，它将整个报文反转成正序。

21.11 将内部格式转换为ASN.1表示形式

过程snb2a是sna2b的逆过程，它将对象标识符从内部格式转换为对应的ASN.1表示形式。

```
/* snb2a.c - snb2a */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <mem.h>

#ifdef SNMP

#include <snmp.h>
#include <asn1.h>

extern u_char snmpbuff[];

/*------------------------------------------------------------------------
 * snb2a - convert the list of bindings from internal form into ASN.1
 *------------------------------------------------------------------------
 */
int snb2a(rqdp)
struct req_desc  *rqdp;
{
        register u_char *bp;
        int             len;
        struct snbentry *bl;
        u_char          *ap;

        for (bl = rqdp->bindlf; bl; bl = bl->sb_next) {
                bp = snmpbuff;  /* used for temporary working space */
                *bp++ = ASN1_OBJID;
                bp += alwriteoid(bp, &bl->sb_oid);
                bp += alwriteval(bl, bp);
                /*
                 * We need to allocate bytes in sb_alstr but can't do it
                 * until we know how many bytes it takes to write the
                 * length of the binding,  so we write that length into
                 * snmpbuff at the end of the binding. Then we can alloc
                 * space, and transfer the data.
                 */
                len = alwritelen(bp, bp - snmpbuff);
                bl->sb_alslen = bp - snmpbuff + len + 1;
                ap = bl->sb_alstr = (u_char *) getmem(bl->sb_alslen);
                bl->sb_aldynstr = TRUE; /* dynamically allocated */
                *ap++ = ASN1_SEQ;
                memcpy(ap, bp, len);  /* write in the length spec.   */
                ap += len;
                memcpy(ap, snmpbuff, bp - snmpbuff);
```

```
        }
        return OK;
}
#endif  /* SNMP */
```

21.12 服务器使用的实用函数

服务器在向发出请求报文的客户返回一个应答报文之后，调用实用过程snfreebl释放绑定链表。

```
/* snfreebl.c - snfreebl */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <mem.h>

#ifdef SNMP

#include <snmp.h>

/*------------------------------------------------------------------------
 * snfreebl - free memory used for ASN.1 strings and snbentry nodes
 *------------------------------------------------------------------------
 */
void snfreebl(bl)
struct snbentry  *bl;
{
        register struct snbentry  *pbl;
        register struct snbentry  *pbl_next;

        if (*bl == (void *)NULLPTR)
            return;
        for (pbl = *bl, pbl_next = pbl->sb_next; pbl_next;
            pbl = pbl_next, pbl_next = pbl->sb_next) {
            if (pbl->sb_aldynstr) {
                freemem(pbl->sb_alstr, pbl->sb_alslen);
            }
            freemem(pbl, sizeof(struct snbentry));
        }
        if (pbl->sb_aldynstr) {
            freemem(pbl->sb_alstr, pbl->sb_alslen);
        }
        freemem(pbl, sizeof(struct snbentry));

        /* erase the list head */
```

```
        *bl = (struct snbentry *)NULLPTR;
}
#endif  /* SNMP */
```

snfreebl 以一个指向绑定链表的指针作为入口参数，利用该指针遍历链表并释放用于保存该绑定的 ASN.1 表示形式的存储区和结点本身。

21.13　一个 SNMP 客户的实现

SNMP客户必须生成和向服务器发送请求报文，等待服务器的响应，并验证该响应是否与请求相对应。过程 snclient 执行客户功能。它以一个请求描述块的地址为入口参数，在请求描述块中包含了报文的内容，其中包括需要完成的操作和一张操作执行对象的绑定表。它调用 mksnmp 生成以 ASN.1 格式表示的报文，调用 open 打开一个用于发送数据报的描述块，并调用 write 发送该报文。

```
/* snclient.c - snclient */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <io.h>

#ifdef SNMP

#include <snmp.h>

/*------------------------------------------------------------------------
 * snclient - send an SNMP request and wait for the response
 *------------------------------------------------------------------------
 */
int snclient(rqdp, fport, stdout)
struct req_desc  *rqdp;
char             *fport;
int              stdout;
{
    u_char          buff[SNMPMAXSZ], reqidsave[10], reqidsavelen;
    int             snmpdev, len;

    rqdp->reqidlen = 0;
    rqdp->err_stat = 0;
    rqdp->err_idx = 0;

    if ((len = mksnmp(rqdp, buff, rqdp->reqtype)) == SYSERR)
        return SCL_BADREQ;
    memcpy(reqidsave, rqdp->reqid, rqdp->reqidlen);
    reqidsavelen = rqdp->reqidlen;

    /* open the SNMP port and put into data mode */
```

```
    if ((snmpdev = open(UDP, fport, ANYLPORT)) == SYSERR ||
        control(snmpdev, DG_SETMODE, DG_DMODE | DG_TMODE) == SYSERR) {
            close(snmpdev);
            return SCL_OPENF;
    }
    if (write(snmpdev, buff, len) == SYSERR) {
            close(snmpdev);
            return SCL_WRITEF;
    }
    /* retry once, on timeout */
    if ((len = read(snmpdev, buff, SNMPMAXSZ)) == TIMEOUT)
            len = read(snmpdev, buff, SNMPMAXSZ);
    if (len == TIMEOUT) {
            close(snmpdev);
            return SCL_NORESP;
    } else if (len == SYSERR) {
            close(snmpdev);
            return SCL_READF;
    }
    if (snparse(rqdp, buff, len) == SYSERR) {
            close(snmpdev);
            return SCL_BADRESP;
    }
    if (reqidsavelen != rqdp->reqidlen ||
            memcpy(reqidsave, rqdp->reqid, reqidsavelen)) {
            close(snmpdev);
            return SCL_BADRESP;
    }
    /* convert the sb_alstr's to objid's and their values */
    if (sna2b(rqdp) == SYSERR) {
            close(snmpdev);
            return SCL_BADRESP;
    }
    close(snmpdev);
    return SCL_OK;
}
#endif  /* SNMP */
```

由于UDP是不可靠的，因而SNMP客户必须实现自己的超时和重传策略。在本例的客户中实现了超时控制，但只有一次重传。为了做到这一点，snclient调用control将UDP描述块置于定时模式（DG_TMODE）。定时模式下，如果在任何数据报到达之前，定时器就已经超时，read操作或者返回一个数据报，或者返回一个特殊值TIMEOUT。如果snclient在两次超时时间内都没有接收到响应，则客户关闭描述块并返回差错代码。

如果snclient收到了响应，它调用snparse将响应转换为内部格式，然后它将响应中的标识符字段与请求中的标识符字段相比较，以验证该报文是对客户刚才发送的请求的响应。如果一切正常，

snclient 调用 sna2b 将每个对象标识符从 ASN.1 表示形式转换为内部格式。

21.14　变量初始化

由 SNMP 使用的简单计数器和变量，若在系统正常启动时没有初始化，则都由过程 sninit 进行初始化。sninit 还要调用 hashinit 初始化用于优化 MIB 名字查找的散列表。当然对应于表格（如选路表）的 MIB 变量则由 TCP/IP 软件初始化。

```
/* sninit.c - sninit */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <ctype.h>
#include <snmp.h>
#include <mib.h>
#include <snmpvars.h>

extern char vers[];

struct oid SysObjectID = { {0}, 1};
#endif  /* SNMP */

/* System & Interface MIB */
char sysDescr[256], SysCortact[256], SysName[256], SysLocation[256];
unsigned SysUpTime, SysServices, IfNumber;

/* IP MIB */
unsigned IpForwarding, IpDefaultTTL, IpInReceives, IpInHdrErrors,
        IpInAddrErrors, IpForwDatagrams, IpInUnknownProtos, IpInDiscards,
        IpInDelivers, IpOutRequests, IpOutDiscards, IpOutNoRoutes,
        IpReasmTimeout, IpReasmReqds, IpReasmOKs, IpReasmFails, IpFragOKs,
        IpFragFails, IpFragCreates, IpRotingDiscards;

/* ICMP MIB */
unsigned IcmpInMsgs, IcmpInErrors, IcmpInDestUnreachs, IcmpInTimeExcds,
        IcmpInParmProbs, IcmpInSrcQuenchs, IcmpInRedirects, IcmpInEchos,
        IcmpInEchoReps, IcmpInTimestamps, IcmpInTimestampReps,
        IcmpInAddrMasks, IcmpInAddrMaskReps, IcmpOutMsgs, IcmpOutErrors,
        IcmpOutDestUnreachs, IcmpOutTimeExcds, IcmpOutParmProbs,
        IcmpOutSrcQuenchs, IcmpOutRedirects, IcmpOutEchos,
        IcmpOutEchoReps, IcmpOutTimestamps, IcmpOutTimestampReps,
        IcmpOutAddrMasks, IcmpOutAddrMaskReps;
```

```
/* UDP MIB */
unsigned UdpInDatagrams, UdpNoPorts, UdpInErrors, UdpOutDatagrams;

/* TCP MIB */
unsigned TcpRtoAlgorithm, TcpRtoMin, TcpRtoMax, TcpMaxConn,
        TcpActiveOpens, TcpPassiveOpens, TcpAttemptFails, TcpEstabResets,
        TcpCurrEstab, TcpInSegs, TcpOutSegs, TcpRetransSegs;
        TcpInErrs, TcpOutAddrMaskReps;

/* SNMP MIB */
unsigned snmpInPkts, snmpOutPkts, snmpInBadVersions,
        snmpInBadCommunityNames, snmpInBadCommunityUses,
        snmpInASNParseErrs, snmpInTooBigs, snmpInNoSuchNames,
        snmpInBadValues, SnmpInReadOnlys, snmpInGenErrs,
        snmpInTotalReqVars, snmpInTotalSetVars, snmpInGetRequests,
        snmpInGetNets, snmpInSetRequests, snmpInGetResponses,
        snmpInTraps, snmpOutTooBigs, snmpOutNoSuchNames,
        snmpOutBadValues, snmpOutGenErrs, snmpOutGetRequests,
        snmpOutGetNexts, snmpOutSetRequests, snmpOutGetResponses,
        snmpOutTraps, snmpEnableAuthenTraps;

#ifdef  SNMP
int snmpinitialized = FALSE;

/*-----------------------------------------------------------------------
 * sninit - initialize the data structures for the SNMP server and client
 *-----------------------------------------------------------------------
 */
void
sninit(void)
{
      if (snmpinitialized)
              return; /* if SNMP data structures already initialized */
      snmpinitialized = TRUE;
      hashinit();

      /* initialize most SNMP variables */
      strcpy(SysDescr, vers); Strcpy(SysContact, CONTACT);
      strcpy(SysLocation, LOCATION); getname(SysName);

      IfNumber = Net.nif - 1;

      /* non-zero int/counter initializations */
      IpDefaultTTL = IP_TTL;          IpInReceives = 0;
      IpReasmTimeout = IP_FTTL;
      TcpRtoAlgorithm = 4;
      TcpRtoMin = TCP_MINRXT*10;
```

```
        TcpRtoMax = TCP_MAXRXT*10;
        TcpMaxConn = Ntcp;

}
#endif  /* SNMP */
```

21.15　小结

SNMP服务器接受输入的请求报文，执行指定的操作，并向客户返回结果。虽然理论上简单，但服务器程序代码中，用于处理SNMP报文中的ASN.1表示形式和对应的内部表示形式之间的转换的代码占了大部分。服务器分析传入报文，并将其转换到一个具有固定字段的结构中。然后，它利用mib表进行从对象标识符和操作到完成指定操作的函数之间的映射。服务器使用不同的数据结构为每个表格指定各自的函数。而通过间接的手段，服务器可以避免为每个简单变量使用一个独立的函数。

客户软件与服务器共享许多过程。客户生成一个报文，发送该报文并等待响应。它必须实现超时机制，并且如有必要，还需完成重传。当一个响应报文到达时，客户分析响应报文，并将其从ASN.1表示形式转换为内部表示形式。客户还要比较响应报文和请求报文中的标识符字段，以保证响应与请求是相对应的。

深入研究

Case、Fedor、Schoffstall 和 Davin［RFC 1157］描述了客户和服务器中的操作。

习题

21.1　过程 snparse 将 SNMP 报文格式编码成一系列用于分析报文的语句。你能设计出另一种策略，将报文格式编码为一种数据结构吗？提示：可考虑常规的编译程序。

21.2　snmib 数据结构为实现 get、set 和 get-next 操作只提供了一个函数指针。另一种设计思想是：snmib 数据结构为每个操作保存一个独立的指针，类似于数组 tabtab 中的元素，试比较这两种设计思想，它们各有什么优、缺点？

21.3　估计一下用于分析报文和完成在ASN.1和内部表示法之间的转换的程序代码所占的比例。它在整个SNMP程序代码中的比例是多少？并估计在IP软件中这类程序代码所占的百分比又是多少？由此可得出什么结论？

21.4　当客户指定一个 get-next 操作并给出数据报聚合体的名字时，snrslv 利用字典顺序指针，并对下一个项目调用函数（使用表达式 *np->mi_next->mi_func）。请解释为什么 snrslv 不需要检查字段 mi_func 是否为空？

21.5　设计一个用户界面，使管理员在不知道 MIB 中有哪些变量的情况下也能够使用 SNMP。

21.6　客户会等待对一个 SNMP 请求的响应报文，它应等待多长时间？客户的重发次数又应当是多少？

21.7　在一个 SNMP get 请求中的一组绑定给出了由对象标识符和值 NULL 组成的序偶。当不

必要使用值 NULL 时，为什么协议中还要包括这些值？

21.8 阅读SNMP协议规范。它是否为诸如IpForwarding之类的变量规定了取值范围？如果是，举出这些变量和它们可能取的值的例子。本章描述的服务器有没有检查SNMP set请求中变量的取值范围？为什么？

21.9 SNMP协议标准指定当set-request与多变量相关时是原子性的，也就是说，要么所有的变量都被设置，要么都不被设置。解释其合理性。

21.10 我们的范例的实现是否遵从了上面习题中所描述的原子设置？为什么？

21.11 假设某个实体不支持一个给定的MIB变量，那么对这个变量的get请求，SNMP应返回什么值？对get-next请求呢？为什么？

21.12 协议标准指明，当客户试图写入一个只读的MIB变量时，SNMP服务器应返回NO SUCH错误。考虑这是为什么？

第 22 章　SNMP：表格访问函数

22.1　引言

第21章详述了SNMP服务器，从中可了解到SNMP服务器是如何利用mib表项中的函数指针来直接调用一个操作的。当一个SNMP报文到达时，服务器将其转换为内部格式，并保存到请求描述块结构中。请求描述块结构中包含有一个指针，指向某个操作执行的对象的对象标识符。对于如整数这样的简单MIB变量来说，访问并不复杂。被调用的访问函数只需要把一个值从TCP/IP软件的数据结构中复制到绑定链表结点中。一旦服务器完成了指定的操作，它将请求描述块以及绑定链表中的数据项，再次转换回外部格式，然后发送一个应答。

本章将集中讲述被调用来处理以表格为对象的get、set和get-next请求的函数。我们可以看到软件是如何将概念上的MIB表格与TCP/IP使用的数据结构一一对应，以及被调用的访问函数是如何在表格上实现这些操作的。最后，它详述了SNMP在实现概念表格中那些与现有数据结构并不对应的表项时，需要用到的数据结构。

22.2　表格访问

和那些对应于一个个存储区的简单变量不同，表格需要更多的软件将SNMP概念表格与内部数据结构一一对应。对MIB表格来说，服务器提供一种机制，允许每个表格具有三个函数，分别实现get、set和get-next操作。如第21章中指出的，服务器软件利用tabtab数组[1]中的指针，以间接的方式选取适当的访问函数，在tabtab数组中，每个操作各有一个指针。

22.3　表格的对象标识符

在数组tabtab中，对应于表格的五个表项并不包含完整的对象标识符。事实上，它们仅具有表格对象标识符的前缀部分。原因很简单：表格中每个数据项的完整的对象标识符由用于标识表格的前缀、用于标识表格中某个表项以及指定该表项中某一字段的后缀组成。在检查标识符时，getmib[2]首先通过在散列表中搜索该标识符来判断它是不是一个简单变量。散列表的搜索要通过比较找出完全的匹配。如果没有找到完全的匹配，getmib将该标识符与一组对应于MIB表格的标识符前缀相比较。

一旦找到前缀匹配的标识符，服务器间接地调用访问函数。访问函数分析出对象标识符的后缀部分，并利用它来选择一个表项，以及表项中的一个字段。对许多表格来说，MIB使用IP地址来选择表项。IP地址被编码在对象标识符中，它在对象标识符中以点分十进制形式出现。下一节中将举例说明。

---

① 见第20.13节定义tabtab的文件snmib.c。

② 见第20.15节出现getmib的文件snhash.c。

22.4 地址入口表函数

MIB定义了一张概念上的地址入口表（address entry table），它对应于机器上的一组接口的IP地址。每个表项有五个字段：IP地址（ipAdEntAddr）、网络接口索引值（ipAdEntIfIndex）、子网掩码（ipAdEntMask）、以及广播地址（ipAdEntBcastAddr）以及能被重组的IP数据报的最大长度（ipAdEntReasmMaxSize）。虽然TCP/IP软件中没有一个数据结构是完全按照这种形式定义的，但网络接口数组nif中包含了所需的信息。

为了识别概念性地址入口表中的一个表项，客户软件生成一个ASN.1对象标识符，该标识符的前缀指定一张表格，后缀指定表格中某个表项中的一个字段。例如，对象标识符

1.3.6.1.2.1.4.20.1.1.128.10.2.3

指定了标准的MIB前缀（1.3.6.1.2.1）、ip子级（4）、ip地址表格ipAddrTable（20）、ip地址表项ipAddrEntry（1）、表项中的字段（1）以及作为该表项索引的IP地址（128.10.2.3）。因此我们认为该对象标识符代表了：

standard-MIB-prefix.ip.ipAddrTable.ipAddrEntry.field.IPaddress

每个概念表格的访问程序中都包括有一个匹配函数，用于检测该表格中是否存在给出的对象。如我们所知，当指定了表格中的一个变量的请求报文到达时，服务器使用对象标识符的前缀选择适当的表格，然后调用表格的匹配函数判断指定的表项是否存在。例如，函数sae_mateh完成地址入口表中的匹配检测。

```
/* sae_match.c - sae_match */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asnl.h>
#include <snhash.h>
#include <stdarg.h>

/*------------------------------------------------------------------------
 * sae_match - check if a variable exists in the IP Address Entry Table
 *------------------------------------------------------------------------
 */
int
sae_match(struct snbentry *bindl,
          .../* int *iface, int *field, int numifaces */)
{
```

```
    int  oidi;
    va_list ap;
    int *iface, *field, numifaces;

    va_strart(ap,bindl);
    iface = va_arg(ap,int *);
    field = va_arg (ap,int *);
    numifaces = va_arg(ap,int);
    va_end(ap);

    oidi = SAE_OIDLEN;   /* skip over fixed part of objid */

    if ((*field = bindl->sb_oid.id[oidi++]) > SNUMF_AETAB)
         return SYSERR;
    for (*iface = 1; *iface <= numifaces; (*iface)++)
            if (soipequ(&bindl->sb_oid.id[oidi],
                        nif[*iface].ni_ip, IP_ALEN))
                break;
    if (*iface > numifaces)
         return SYSERR;
    oidi += IP_ALEN;
    if (oidi != bindl->sb_oid.len)  /* verify oidi at end of objid  */
         return SYSERR;
    return OK;
}
#endif  /* SNMP */
```

上述程序代码并不复杂。sae_match 略过地址入口表格和地址入口结构的标识符，提取标识了所需表项中某个字段的整型数值，并将该值保存在由参数field给出的存储区中。如果这个数值超出有效范围，sae_match 返回一个差错代码，表示该对象标识符无法对应于有效表项。

一旦它认为字段定义值有效，sae_match 遍历所有的网络接口，将对象标识符中剩余的四个值与每个接口的IP地址相比较。如果它找到匹配项，sae_match 在由参数 iface 指定地址的存储区中保存接口号，并返回一个代码，指出它发现了匹配项。如果没有找到匹配项，sae_match 返回一个差错代码，指出对象标识符无法与有效的表项相对应。

22.4.1　对地址入口表的 get 操作

过程 sae_get 实现以地址入口表中某个表项为对象的 get 操作。

```
/* sae_get.c - sae_get */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP
```

```
#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*-----------------------------------------------------------------------
 * sae_get - perform a get on a variable in the IP Address Entry Table
 *-----------------------------------------------------------------------
 */
int sae_get(bindl, numifaces)
struct snbentry  *bindl;
int          numifaces;
{
        int  iface, field;

        if (sae_match(bindl, &iface, &field, numifaces) == SYSERR)
                return SERR_NO_SUCH;
        switch (field) {
        case 1:         /* ipAdEntAddr */
                SVTYPE(bindl) = ASN1_IPADDR;
                SVIPADDR(bindl) = nif[iface].ni_ip;
                return SNMP_OK;
        case 2:         /* ipAdEntIfIndex */
                SVTYPE(bindl) = ASN1_INT;
                SVINT(bindl) = iface;
                return SNMP_OK;
        case 3:         /* ipAdEntNetMask */
                SVTYPE(bindl) = ASN1_IPADDR;
                SVIPADDR(bindl) = nif[iface].ni_mask;
                return SNMP_OK;
        case 4:         /* ipAdEntBcastAddr */
                SVTYPE(bindl) = ASN1_INT;
                SVINT(bindl) = IP2I(nif[iface].ni_brc) & 0x01;
                return SNMP_OK;
        case 5:         /* ipAdEntReasmMaxSize */
                SVTYPE(bindl) = ASN1_INT;
                SVINT(bindl) = nif[iface].ni_maxreasm;
                return SNMP_OK;
        default:
                break;
        }
        return SERR_NO_SUCH;
}
#endif  /* SNMP */
```

sae_get调用sae_match找出与对象标识符匹配的网络接口。如果该标识符无法与有效的表项相匹配，则返回一个差错代码。如果它找到了匹配项，sae_get进一步处理对所需信息的访问。

程序代码中的switch语句利用由sae_match设置的field的值，选取概念表项中的一个字段。实现了某个字段上取操作的程序代码在绑定链表的结点中存放该字段的类型和字段值，并返回到调用者。

22.4.2　对地址入口表的get-first操作

函数sae_getf实现以地址入口表中的一个表项为对象的get-first操作。

```
/* sat_getf.c - sat_getf */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*------------------------------------------------------------------------
 * sat_getf - perform a getfirst on a variable in the IPAddr Entry Table
 *------------------------------------------------------------------------
 */
int
sat_getf(struct snbentry *bindl, struct mib_info *mip, int numifaces)
{
        int  iface, oidi;

        iface = sae_findnext(-1, numifaces);

        /* write the objid into the bindings list and call get func */
        memcpy(bindl->sb_oid.id, mip->mi_objid.id, mip->mi_objid.len*2);
        oidi = mip->mi_objid.len;

        bindl->sb_oid.id[oidi++] = (u_short) 1;   /* field   */
        sip2ocpy(&bindl->sb_oid.id[oidi], nif[iface].ni_ip);
        bindl->sb_oid.len = oidi + IP_ALEN;

        return sat_get(bindl, numifaces);
}
#endif  /* SNMP */
```

为了在表中找出按字典顺序排序的第一个表项，sae_getf调用函数sae_findnext，向其传递-1作为起始接口。sae_findnext找出第一个接口并返回该接口的索引值。一旦sae_getf知道了接口号，它为该接口计算生成正确的对象标识符，并利用它来取代绑定链表结点中的标识符。为了生成这个标识符，sae_getf在字段定义值中插入1（用以识别第一个字段），并调用过程sip2ocpy将表项的IP地

址复制到对象标识符中。最后，sae_getf 调用 sae_get 来取出所需的信息。

22.4.3 对地址入口表的 get-next 操作

函数 sae_getn 实现以地址入口表中某个表项为对象的 get-next 操作。

```
/* sae_getn.c - sae_getn */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*------------------------------------------------------------------------
 * sae_getn - perform a getnext on a variable in the IPAddr Entry Table
 *------------------------------------------------------------------------
 */
int sae_getn(bind1, mip, numifaces)
struct snbentry  *bind1;
struct mib_info  *mip;
int         numifaces;
{
      int  field, iface, oidi;

      if (sae_match(bind1, &iface, &field, numifaces) == SYSERR)
           return SERR_NO_SUCH;
      if ((iface = sae_findnext(iface, numifaces)) == -1) {
           iface = sae_findnext(-1, numifaces);
           if (++field > SNUMF_AETAB)
                return (*mip->mi_next->mi_func)
                           (bind1, mip->mi_next, SOP_GETF);
      }
      /* The fixed part of the objid is correct. Update the rest      */
      oidi = SAE_OIDLEN;

      bind1->sb_oid.id[oidi++] = (u_short) field;
      sip2ocpy(&bind1->sb_oid.id[oidi], nif[iface].ni_ip);
      bind1->sb_oid.len = oidi + IP_ALEN;

      return sae_get(bind1, numifaces);
}
#endif  /* SNMP */
```

get-next 操作的实现需要三个步骤。首先，sae_getn 调用 sae_match 在地址入口表中找出由绑定链表结点中的对象标识符指定的表项。第二，它找出按字典顺序排列的“下一个”（next）地址入口表中元素的位置。第三，执行 get 操作。

为了找到地址入口表中按字典顺序排列的下一个表项，sae_getn 调用函数 sae_findnext，以入口参数的形式向其传递起始接口。如果 sae_findnext 返回一个有效的接口号，就不再需要更进一步的搜索。然而，如果 sae_findnext 返回了一个值 -1，它意味着对象标识符指定的表项是表格中的最后一个表项。

当 sae_getn 到达表格的末尾时，它必须为字段定义值加 1，并回到地址入口表中按字典顺序排列的第一个表项。最后，当它递增地经过最后一个表项中的最后字段后，sae_getn 就以 MIB 表中按字典顺序排列的下一个表项为对象，调用 get-first 操作。如果字段定义值加 1 后得到的仍是一个有效的字段值，则 sae_getn 必须找出地址入口表中按字典顺序排列的第一个表项。为了找出按字典顺序的第一个表项，它调用 sae_findnext，向其传递 -1 作为起始接口。

22.4.4　地址入口表中的递增搜索

过程 sae_findnext 搜索地址入口表，查找按字典顺序排列在给定表项后的下一个表项。

```
/* sat_findn.c - sat_findnext */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

/*------------------------------------------------------------------------
 * sae_findnext - find the next interface in the lexical ordering
 *------------------------------------------------------------------------
 */
int sae_findnext(iface, numifaces)
int    iface;
int    numifaces;
{
        int  i, nextif;

        for (nextif = -1, i = 1; i <= numifaces; ++i) {
                if (iface >= 0 &&
                   memcmp(&nif[i].ni_ip, &nif[iface].ni_ip,IP_ALEN) <= 0)
                       continue;
                if (nextif < 0 ||
                    memcmp(&nif[i].ni_ip,&nif[nextif].ni_ip,IP_ALEN) < 0)
                       nextif = i;
        }
        return next;
}
```

```
#endif  /* SNMP */
```

我们认为参数iface指定了起始接口。当iface值为-1时，sae_findnext从头开始查找，并定位在具有最小字典顺序的对象标识符的表项上（即表中按字典顺序排列的第一个表项）。当参数iface的值为一个正数时，sae_findnext从该接口开始，并定位在按字典顺序排列在该接口表项之后的“下一个”表项上（即表中位于iface之后的“下一个”表项）。

如果sae_findnext在表中找到所需的表项，则返回该表项的索引值。否则返回-1，以指出起始接口是表中按字典顺序排列的最后一个表项。

22.4.5 对地址入口表的set操作

由于不允许管理人员对地址入口表做任何改动，所以set操作的实现不复杂。函数sae_set仅在调用时返回一个差错代码。

```
/* sae_set.c - sae_set */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>

/*------------------------------------------------------------------------
 * sae_set - return error: the IP Address Entry Table is read-only
 *------------------------------------------------------------------------
 */
int
sae_set(struct snbentry *bindl, struct mib_info *mip, int numifaces)
{
        return SERR_NO_SUCH;
}
#endif  /* SNMP */
```

22.5 地址转换表函数

MIB定义了一张概念性地址转换表，它对应于ARP高速缓存[1]。该表中每个表项具有四个字段：从中获得该表项的网络接口号（ipNetToMediaIfIndex），该表项中的物理（或MAC）地址（ipNetToMediaPhysAddress），表项中的IP地址（ipNetToMediaNetAddress）和该表项的类型（ipNetToMediaType）。地址转换表中某表项的ASN.1名是对该表项的IP地址字段的编码。

地址转换表允许管理员询问或控制ARP高速缓存。如果set请求中包含IP地址和物理地址，序偶就被插入ARP高速缓存。如果ipNetToMediaType字段包含invalid（值为2），相应的表项则被标

① 在目前的MIB版本中，net-to-media表（ipNetToMedia）取代了address translation表。

记为无效（其实，应从高速缓存中将它取出）。

一个地址转换表表项的通用格式为：

standard-MIB-prefix.ip.nmTable.nmEntry.field.iface.1.IPaddress

前缀用于标识表格和地址转换表结构，而剩余八位组定义了某个表项中的field、与该表项相关的接口、类型（1）以及一个 IP 地址。

当对象标识符指定了地址转换表中的一个数据项时，SNMP服务器只对标识了表格的标识符前缀进行检查。然后它调用专门处理表格的函数，该函数必须完成对该对象标识符剩余部分的分析，并验证它对应于一个有效的表项。函数snm_match将对象标识符的后缀与表格中的表项相比较，以判断是否有某个表项与给出的 ASN.1 名相匹配。

```
/* snm_match.c - snm_match */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <stdarg.h>

/*------------------------------------------------------------------------
 * snm_match - check if a variable exists in the IP NetToMedia Table
 *------------------------------------------------------------------------
 */
int
snm_match(struct snbentry *bindl,
          ... /*int *iface, int *entry, int *field, int numiface */
{
      int     oidi;
      struct  arpentry *pae
      va_list ap;
      int *iface, *entry, *field, numifaces;

      va_start(ap, bindl);
      iface = va_arg(ap, int *);
      entry = va_arg(ap, int *);
      field = va_arg(ap, int *);
      numifaces = va_arg(ap, int);
      va_end(ap);

      oidi = SNM_OIDLEN;
      if ((*field = bindl->sb_oid.id[oidi++]) > SNUMF_NMTAB)
```

```
        return SYSERR;
if((*iface = bindl->sb_oid.id[oidi++])>numifaces)
        return SYSERR;
oidi++;         /* skip over the 1 */
/*
 * oidi now points to IPaddr.  Read it and match it against
 * the correct arp cache entry to get entry number
 */
for (*entry = 0; *entry < ARP_TSIZE; (*entry)++) {
        pag = &arptable[*entry];
        if (pae->ae_state != AS_FREE &&
            pae->ae_pni == &nif[*iface] &&
            soipequ(&bindl->sb_oid.id[oidi],
                    *((IPaddr *)pae->ae_pra),IP_ALEN))
                break;
        }
        if (*entry >= ARP_TSIZE)
                return SYSERR;
        if (oidi + IP_ALEN != bindl->sb_oid.len)
                return SYSERR;    /* oidi is not at end of objid */
        return OK;
}
#endif  /* SNMP */
```

在提取了字段和接口的定义值之后，snm_match将四个八位组的IP地址与保存在ARP高速缓存中的每个表项的IP地址相比较。如果找到匹配项，它返回OK，否则返回SYSERR。

22.5.1 对地址转换表的get操作

函数snm_get实现以地址转换表为对象的get操作。在调用snm_match找出一个正确的ARP表项之后，它应用一个switch语句来选择所需的字段，并将字段中的信息复制到请求描述块中。

```
/* snm_get.c - snm_get */

#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <mem.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*------------------------------------------------------------------------
 * snm_get - do a get on a variable in the IP NetToMedia Table
```

```
 *------------------------------------------------------------------
 */
int
snm_get(struct snbentry *bind1, int numifaces)
{
        int     iface, entry, field;
        if (snm_match(bind1, &iface, &entry, &field, numifaces) == SYSERR)
                return SERR_NO_SUCH;
        switch(field) {
        case 1:         /* ipNetToMediaIfIndex */
                SVTYPE(bind1) = ASN1_INF;
                SVINT(bind1) = iface;
                return SNMP_OK;
        case 2:         /* ipNetToMediaPhysAddress */
                SVTYPE(bind1) = ASN1_OCTSTR;
                SVINT(bind1) = (char *) getmem(EP_ALEN);
                memcpy(SVSTR(bind1), arptable[entry].ae_hwa,
                        EP_ALEN);
                SVSTRLEN(bind1) = EP_ALEN;
                return SNMP_OK;
        case 3:         /* ipNetToMediaNetAddress */
                SVTYPE(bind1) = ASN1_IPADDR;
                memcpy(&SVIPADDR(bind1), arptable[entry].ae_pra,
                        IP_ALEN);
                return SNMP_OK;
        case 4:         /* ipNetToMediaType */
                SVTYPE(bind1) = ASN1_INT;
                if (arptable[entry].ae_state == AS_FREE)
                    SVINT(bind1) = 2;  /* invalid(2) */
                else
                    SVINT(bind1) = 3;  /* dynamic(3) */
                return SNMP_OK;
        default:
                break;
        }
        return SERR_NO_SUCH;
}
#endif  /* SNMP */
```

22.5.2　对地址转换表的 get-first 操作

函数 snm_getf 实现以地址转换表为对象的 get-first 操作。

```
/* snm_getf.c - snm_getf */

#include <conf.h>
```

```
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>
#include <snhash.h>

/*------------------------------------------------------------------------
 * snm_getf - do a getfirst on a variable in the IP NetToMedia Table
 *------------------------------------------------------------------------
 */
int snm_getf(bindl, mip, numifaces)
struct snbentry  *bindl;
struct mib_info  *mip;
int          numifaces;
{
        int  iface, entry, oidi;

        for (iface=1; iface <= numifaces; ++ifaoe)
             if ((entry = snm_findnext(-1, iface)) >= 0)
                  break;
        if (iface > numifaces) {  /* no active interface found */
             if (mip->mi_next)
                  return (*mip->mi_next->mi_func)
                          (bindl, mip->mi_next, SOP_GETF);
             return SERR_NO_SUCH;       /* no next node */
        }
        memcpy(bindl->sb_oid.id, mip->mi_objid.id, mip->mi_objid.len*2);
        oidi = mip->mi_objid.len;

        bindl->sb_oid.id[oidi++] = (u_short) 1;          /* field   */
        bindl->sb_oid.id[oidi++] = (u_short) iface;
        bindl->sb_oid.id[oidi++] = (u_short) 1;
        sip2ocpy(&bindl->sb_oid.id[oidi],
                 *((IPaddr *)arptable[entry].ae_pra));
        bindl->sb_oid.len = oidi + IP_ALEN;

        return snm_get(bindl, numifaces);
}
#endif  /* SNMP */
```

地址转换概念表将其中的表项划分为几组，分别与各个接口相对应，并将对应于某个接口的所有表项按字典顺序相邻排列。因此，为了找出按字典顺序排序的第一个表项，sat_getf必须一个个地循环经过所有可能的接口。在每次循环中，调用sat_findnext查看高速缓存中是否存在与给定接口

相关的表项。一旦 snm_findnext 报告说找到了一个有效的表项，则 snm_getf 停止查找。然而，即便是 ARP 高速缓存完全为空，在检查过所有接口之前，循环还是要继续执行。如果确认 ARP 高速缓存为空，snm_getf 以 MIB 表中按字典顺序排列的下一个表项（假设还有一个这样的表项存在）执行 get-first 操作。

如果在 ARP 高速缓存中找到一个有效表项，snm_getf 就生成一个对应于该表项的对象标识符，并调用 snm_get 提取其中的值。在生成对象标识符时，它使用字段 1，因为第一个字段的对象标识符的字典顺序总是最小的。

22.5.3　对地址转换表的 get_next 操作

函数 snm_getn 提供了以地址转换表为对象的 get-next 操作。

```
/* snm_getn.c - snm_getn */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*------------------------------------------------------------------------
 * snm_getn - do a getnext on a variable in the IP NetToMedia Table
 *------------------------------------------------------------------------
 */
int snm_getn(bindl, mip, numifaces)
struct snbentry  *bindl;
struct mib_info  *mip;
int         numifaces;
{
        int  entry, iface, field, oidi;
        if (snm_match(bindl, &iface, &entry, &field, numifaces) == SYSERR) {
             return SERR_NO_SUCH;
        }
        while (field <= SNUMF_ANMTAB) {
             if ((entry = snm_findnext(entry, iface)) >= 0)
                  break;
             if (++iface > numifaces) {
                  iface = 1;
                  ++field;
             }
        }
```

```
        if (entry < 0)
             return (*mip->mi_next->mi_func)
                        (bindl, mip->mi_next, SOP_GETF);
        }
        oidi = SNM_OIDLEN;    /* 4.22.1 */

        bindl->sb_oid.id[oidi++] = (u_short) field;
        bindl->sb_oid.id[oidi++] = (u_short) iface;
        bindl->sb_oid.id[oidi++] = (u_short) 1;
        sip2ocpy(&bindl->sb_oid.id[oidi],
                 *((IPaddr *)arptable[entry].ae_pra));
        bindl->sb_oid.len = oidi + IP_ALEN;

        return snm_get(bindl, numifaces);
}
#endif   /* SNMP */
```

snm_getn调用snm_match查找与给出的对象标识符匹配的表项。然后它调用函数snm_findnext在ARP高速缓存中搜索具有相同接口的下一个有效表项。如果ARP高速缓存中没有这样的表项存在，则它试图在下一个接口的相关表项中搜索。如果试过了所有的接口，snm_getn将字段定义值（内部变量field）加1，并重新开始搜索表格。这种循环在两种情况下终止，或者是因为snm_findnext找到了一个紧跟在指定的起始表项后的有效表项，或者是因为没有这样的表项存在。如果snm_getn搜索过了所有可能的接口而没有找到一个表项，则它以MIB表中按字典顺序紧跟在地址转换表后的变量为对象执行get-first操作。否则，snm_getn为该表项生成新的对象标识符，并调用snm_get提取其值。

22.5.4 地址转换表中的递增搜索

函数snm_findnext根据MIB字典顺序在ARP高速缓存中搜索下一个表项。

```
/* snm_findn.c - snm_findnext, snmcmp */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
/*------------------------------------------------------------------------
 * snm_findnext - for given iface, find next resolved arp entry
 *------------------------------------------------------------------------
 */
int snm_findnext(entry, iface)
int    entry;
int    iface;
{
```

```
        int   i, next;

        next = -1;
        for (i = 0; i < ARP_TSIZE; ++i) {
             struct arpentry *pae = &arptable[i];

             if (pae->ae_state == AS_FREE ||
                 pae->ae_pni != &nif[iface] ||
                 (entry >= 0 && snmcmp(pae, &arptable[entry]) <= 0))
                  continue;
             if (next < 0 || snmcmp(pae, &arptable[next]) < 0)
                  next = i;
        }
        return next;
}

/*-----------------------------------------------------------------------
 * snmcmp - compare two ARP table entries in SNMP,lexicographic order
 *-----------------------------------------------------------------------
 */
int snmcmp(pae1, pae2)
struct arpentry   *pae1, *pae2;
{
      int   rv;

      if ((rv = (pae1->ae_prlen - pae2->ae_prlen)))
       return rv;
      return memcmp(pae1->ae_pra, pae2->ae_pra, pae1->ae_prlen);
}
#endif   /* SNMP */
```

snm_findnext 利用参数 entry 作为指定的起始点索引值，并利用参数 iface 选取与某个接口对应的一组表项。它搜索 ARP 高速缓存，并且只查看那些与 iface 相关的有效表项。如果参数 entry 包含的值为-1，则snm_findnext记下与iface匹配的第一个表项。一旦它找到候选的表项后，snm_findnext 仅在找出另一个字典顺序大于初始表项又小于候选表项的新表项之后，才会替换候选表项值。

22.5.5　乱中有序

MIB地址转换表体现了一种有趣的思想：那就是可以为任何系统数据结构定义结构及顺序。虽然 ARP 实现方案将所有网络接口的所有表项都保存在一张高速缓存表中，但 MIB 定义了一张以接口为索引的概念性表格，就像每个接口都具有一张 ARP 高速缓存表。地址转换表的访问函数通过实施某种表项搜索顺序，使人们觉得似乎有多张独立存在的表格。这样，在访问地址转换表时，SNMP 客户完全不用知道具体的实现方案。

22.5.6 对地址转换表的set操作

与前面讲述的地址入口表不同，地址转换表不但允许管理员取出变量值，也允许为变量赋值。函数snm_set提供了set操作。

```
/* snm_set.c - snm_set */

#include <conf.h>
#include <kernel.h>
#include <network.h>


#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*------------------------------------------------------------------------
 * snm_set - do a set on a variable in the IP NetToMedia Table
 *------------------------------------------------------------------------
 */
int snm_set(bind1, mip, numifaces)
struct snbentry  *bind1;
struct mib_info  *mip;
int         numifaces;
{
      int  iface, entry, field;

      if (snm_match(bind1, &iface, &entry, &field, numifaces) == SYSERR)
           return SERR_NO_SUCH;
      switch (field) {
      case 1:         /* ipNetToMediaIfIndex */
           if (SVTYPE(bind1) != ASN1_INT)
                return SERR_BAD_VALUE;
           if (SVINT(bind1) <= 0)
                return SERR_BAD_VALUE;
           if (SVINT(bind1) > numifaces)
                return SERR_BAD_VALUE;
           arptable[entry].ae_pni = &nif[SVINT(bind1)];
           return SNMP_OK;
      case 2:         /* ipNetToMediaphysAddress */
           if (SVTYPE(bind1) != ASN1_OCTSTR)
                return SERR_BAD_VALUE;
           if (SVSTRLEN(bind1) != EP_ALEN)
                return SERR_BAD_VALUE;
```

```
        memcpy(arptable[entry].ae_hwa, SVSTR(bindl), EP_ALEN);
        return SNMP_OK;
    case 3:         /* ipNetToMediaNetAddress */
        if (SVTYPE(bindl) != ASN1_IPADDR)
            return SERR_BAD_VALUE;
        memcpy(arptable[entry].ae_pra, &SVIPADDR(bindl), IP_ALEN);
        return SNMP_OK;
    case 4:         /* ipNetToMediaType */
        if (SVTYPE(bindl) != ASN1_INT)
            return SERR_BAD_VALUE;
        /* only invalid(2) may be set */
        if (SVINT(bindl) != 2)
            return SERR_BAD_VALUE;
        /* invalidate this entry */
            arptable[entry].ae_state = AS_FREE;
            return SNMP_OK;
    default:
            break;
    }
    return SERR_NO_SUCH;
}
#endif  /* SNMP */
```

snm_set检查值的类型，以验证对象的类型与它将要设置的值的类型一致。在绝大多数情况下，snm_set 在赋值之前，还要查看该值是否在合法的范围内。

22.6　网络接口表的函数

MIB 定义了一张保存有关每个网络接口的信息的概念性网络接口表（network interface table），服务器利用数组 nif[1]中的数组元素保存该 MIB 概念表格所需的信息。

22.6.1　接口表标识符的匹配

网络接口表中表项的ASN.1对象标识符的构造与前面所述的表格中的一样。变量名的前缀定义了表格，而后缀定义了表格中的一个字段以及网络接口。通用格式为：

standard-MIB-prefix.interfaces.ifTable.ifTableEntry.field.iface

函数 sif_match 将后缀部分进行解码，并验证这些值的有效性。

```
/* sif_match.c - sif_match */

#include <conf.h>
```

① 参看 3.2.1 节，文件 netif.h 中包含有对 nif 的定义。

```
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <stdarg.h>

/*------------------------------------------------------------------------
 * sif_match - check if a variable exists in the Interfaces Table.
 *------------------------------------------------------------------------
 */
int sif_match(bindl, iface, field, numifaces)
struct snbentry  *bindl;
int          *iface;
int          *field;
int          numifaces;
{
        int  oidi;
        va_list ap;
        int *iface, *field, numifaces;

        va_start(ap, bindl);
        iface = va_arg(ap,int *);
        field = va_arg(ap,int *);
        numifaces = va_arg(ap,int);
        va_end(ap);

        oidi = SIF_OIDLEN;
        if ((*field = bindl->sb_oid.id[oidi++]) > SNUMF_IFTAB)
             return SYSERR;
        if ((*iface = bindl->sb_oid.id[oidi++]) > numifaces)
             return SYSERR;

        if (oidi != bindl->sb_oid.len)
             return SYSERR;  /* oidi is not at end of objid */
        return OK;
}
#endif  /* SNMP */
```

22.6.2 对网络接口表的 get 操作

由于MIB接口表中，每个表项具有21个字段，因此像get和set这样的操作需要的程序代码比前面提到的同类程序代码要多。例如，函数sif_get提供了get操作。

```
/* sif_get.c - sif_get */
```

```
#include <conf.h>
#include <kernel.h>
#include <network.h>
#include <mem.h>
#include <q.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*------------------------------------------------------------------------
 * sif_get - perform a get on a variable in the Interfaces Table
 *------------------------------------------------------------------------
 */
int sif_get(bindl, numifaces)
struct snbentry  *bindl;
int          numifaces;
{
        int  iface, field, sl;

        if (sif_match(bindl, &iface, &field, numifaces) == SYSERR){
             return SERR_NO_SUCH;
        }
        switch (field) {
        case 1:          /* ifIndex */
             SVTYPE(bindl) = ASN1_INT;
             SVINT(bindl) = iface;
             return SNMP_OK;
        case 2:          /* ifDescr */
             SVTYPE(bindl) = ASN1_OCTSTR;
             if(nif[iface].ni_descr){
                SVSTRLEN(bindl) = sl = strlen(nif[iface].ni_descr);
                SVSTR(bindl) = (char *) getmem(sl);
                memcpy(SVSTR(bindl), nif[iface].ni_descr, sl);
             } else {
                char *descr = "<unknown>";
                SVSTRLEN(bindl) = sl = strlen(descr);
                SVSTR(bindl) = (char *)getmem(sl);
                memcpy (SVSTR(bindl), descr, sl);
             }return SNMP_OK;
        case 3:          /* ifType */
             SVTYPE(bindl) = ASN1_INT;
             SVINT(bindl) = nif[iface].ni_mtype;
```

```
        return SNMP_OK;
    case 4:         /* ifMtu */
        SVTYPE(bindl) = ASN1_INT;
        SVINT(bindl) = nif[iface].ni_mtu;
        return SNMP_OK;
    case 5:         /* ifSpeed */
        SVTYPE(bindl) = ASN1_GAUGE;
        SVINT(bindl) = nif[iface].ni_speed;
        return SNMP_OK;
    case 6:         /* ifPhysAddress */
        SVTYPE(bindl) = ASN1_OCTSTR;
        SVSTR(bindl) = (char *) getmem(nif[iface].ni_hwa.ha_len);
        memcpy(SVSTR(bindl), nif[iface].ni_hwa.ha_addr,
             SVSTRLEN(bindl) = nif[iface].ni_hwa.ha_len);
        return SNMP_OK;
    case 7:         /* ifAdminStatus */
        SVTYPE(bindl) = ASN1_INT;
        SVINT(bindl) = nif[iface].ni_admstate;
        return SNMP_OK;
    case 8:         /* ifOperStatus */
        SVTYPE(bindl) = ASN1_INT;
        SVINT(bindl) = nif[iface].ni_state;
        return SNMP_OK;
    case 9:         /* ifLastChange */
        SVTYPE(bindl) = ASN1_TIMETICKS;
        SVINT(bindl) = nif[iface].ni_lastchange;
        return SNMP_OK;
    case 10:        /* ifInOctets */
        SVTYPE(bindl) = ASN1_COUNTER;
        SVINT(bindl) = nif[iface].ni_ioctets;
        return SNMP_OK;
    case 11:        /* ifInUcastPkts */
        SVTYPE(bindl) = ASN1_COUNTER;
        SVINT(bindl) = nif[iface].ni_iucast;
        return SNMP_OK;
    case 12:        /* ifInNUcastPkts */
        SVTYPE(bindl) = ASN1_COUNTER;
        SVINT(bindl) = nif[iface].ni_inucast;
        return SNMP_OK;
    case 13:        /* ifInDiscards */
        SVTYPE(bindl) = ASN1_COUNTER;
        SVINT(bindl) = nif[iface].ni_idiscard;
        return SNMP_OK;
    case 14:        /* ifInErrors */
        SVTYPE(bindl) = ASN1_COUNTER;
```

```
        SVINT(bindl) = nif[iface].ni_ierrors;
        return SNMP_OK;
    case 15:        /* ifInUnknownProtos */
        SVTYPE(bindl) = ASN1_COUNTER;
        SVINT(bindl) = nif[iface].ni_iunkproto;
        return SNMP_OK;
    case 16:        /* ifOutOctets */
        SVTYPE(bindl) = ASN1_COUNTER;
        SVINT(bindl) = nif[iface].ni_ooctets;
        return SNMP_OK;
    case 17:        /* ifOutUcastPkts */
        SVTYPE(bindl) = ASN1_COUNTER;
        SVINT(bindl) = nif[iface].ni_oucast;
        return SNMP_OK;
    case 18:        /* ifOutNUcastPkts */
        SVTYPE(bindl) = ASN1_COUNTER;
        SVINT(bindl) = nif[iface].ni_onucast;
        return SNMP_OK;
    case 19:        /* ifOutDiscards */
        SVTYPE(bindl) = ASN1_COUNTER;
        SVINT(bindl) = nif[iface].ni_odiscard;
        return SNMP_OK;
    case 20:        /* ifOutErrors */
        SVTYPE(bindl) = ASN1_COUNTER;
        SVINT(bindl) = nif[iface].ni_oerrors;
        return SNMP_OK;
    case 21:        /* ifOutQLen */
        SVTYPE(bindl) = ASN1_GAUGE;
        SVINT(bindl) = lenq(nif[iface].ni_outq);
        return SNMP_OK;
    case 22:        /* ifSpecific*/
        {
            struct oid IfSpec = { {0,0}, 2};
                SVTYPE(bindl) = ASN1_OBJID;
                /* set to 0.0, per rfc 1213 */
                SVOIDLEN(bindl) = IfSpec.len;
                memcpy(SVOID(bindl), IfSpec.id, Ifspec.len * 2);
            }
            return SNMP_OK;
    default:
            break;
    }
    return SERR_NO_SUCH;
}
#endif  /* SNMP */
```

正如程序代码所示，接口表的概念字段包括：硬件类型、最大传输单元、物理地址以及像计算输入和输出时发生差错次数的计数器。程序代码将每个概念性字段转换为本地软件用来保存接口信息的数组中对应的字段。

22.6.3 对网络接口表的get-first操作

函数sif_getf定义以网络接口表为对象的get-first操作。

```
/* sif_getf.c - sif_getf */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*------------------------------------------------------------------------
 * sif_getf - perform a getfirst on a variable in the Interfaces Table
 *------------------------------------------------------------------------
 */
int sif_getf(bind1, mip, numifaces)
struct snbentry  *bind1;
struct mib_info  *mip;
int    numifaces;
{
        int  oidi;

        memcopy(bind1->sb_oid.id, mip->mi_objid.id, mip->mi_objid.len*2);
        oidi = mip->mi_objid.len;

        bind1->sb_oid.id[oidi++] = (u_short) 1;          /* field   */
        bind1->sb_oid.id[oidi++] = (u_short) 1;          /* interface    */
        bind1->sb_oid.len = oidi;

        return sif_get(bind1, numifaces);
}
#endif  /* SNMP */
```

与前面讨论的表格不同，网络接口表不用某个数据值作为索引。事实上，我们认为接口表是一个一维数组，它以从1到最大接口数之间的一个整数为索引。对这样的表格，get-first操作的实现极为简单。sif_getf生成一个对象标识符，指明了网络接口表中接口1上的字段1。然后它调用get操作取出该变量的值。

22.6.4　对网络接口表的 get-next 操作

函数 sif_getn 提供了以网络接口表为对象的 get-next 操作。

```
/* sif_getn.c - sif_getn */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>

/*------------------------------------------------------------------------
 * sif_getn - perform a getnext on a variable in the Interfaces Table.
 *------------------------------------------------------------------------
 */
int
sif_getn(bindl, mip, numifaces)
struct snbentry  *bindl;
struct mib_info  *mip;
int         numifaces;
{
        int  oidi, field, iface;

        if (sif_match(bindl, &iface, &field, numifaces) == SYSERR) {
                return SERR_NO_SUCH;
        }
        if (++iface > numifaces) {
                iface = 1;
                if (++field > SNUMF_IFTAB) {
                        return (*mip->mi_next->mi_func)
                                (bindl, mip->mi_next, SOP_GETF);
                }
        }
        oidi = SIF_OIDLEN;   /* 2.2.1 */

        bindl->sb_oid.id[oidi++] = (u_short) field;
        bindl->sb_oid.id[oidi++] = (u_short) iface;
        bindl->sb_oid.len = oidi;

        return sif_get(bindl, numifaces);
}
#endif  /* SNMP */
```

由于表项的对象标识符可以在不知道表格内容的情况下构造，因而sif_getn不需要做搜索工作。它首先对接口号加1。如果此时接口号的值超过了最大接口数，则sif_getn将接口号置为1，然后给字段号加1。如果字段号超过了最大的字段数目，则再没有其他的表项存在，所以sif_getn以MIB字典顺序排列的下一变量为对象执行get-first操作。

与其他表格一样，get-next操作在找到下一个对象之后，需要执行get操作。因此如果sif_getn找到下一个有效的对象后，就为该数据项构造一个新的对象标识符，然后调用sif_get取出它的值。

22.6.5 对网络接口表的set操作

网络接口表允许管理员设置管理状态字段，而其他字段仅允许进行读访问。因此实现方案对以其他字段为对象的set操作返回一个差错代码。函数sif_set包含了这段程序代码。

```
/* sif_set.c - sif_set */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*------------------------------------------------------------------------
 * sif_set - perform a set on a variable in the Interfaces Table
 *------------------------------------------------------------------------
 */
int sif_set(bindl, mip, numifaces)
struct snbentry  *bindl;
struct mib_info  *mip;
int              numifaces;
{
        int iface, field;

        if (sif_match(bindl, &iface, &field, numifaces) == SYSERR)
                return SERR_NO_SUCH;
        /* the only settable object here is ifAdminStatus (ifEntry 7) */
        if (field != 7)
                return SERR_NO_SUCH;
        if (SVTYPE(bindl) != ASN1_INT)
                return SERR_BAD_VALUE;
        if (SVINT(bindl) < 0 || SVINT(bindl) > 1)
                return SERR_BAD_VALUE;
        /* value is OK, so install it */
        nif[iface].ni_admstate = SVINT(bindl);
```

```
        return SNMP_OK;
}
#endif  /* SNMP */
```

在验证了请求中指明的字段是ifAdminStatus之后，sif_set检查该值，以确保它是一个正数，然后检查接口定义值，以保证它指定的是一个有效的接口。如果一切正常，sif_set完成赋值操作。

22.7　选路表函数

MIB定义了一张对应于网关的IP选路表的概念性表格。与地址入口表项的对象标识符一样，选路表的对象标识符中隐含了两个数据，一个是字段标识符，另一个是被SNMP当作选路表索引使用的IP地址。该对象标识符的通用格式如下：

standard-MIB-prefix.ip.ipRouteTable.ipRouteEntry.field.IPdestaddr

标识符中的IPdestaddr部分给出了一个4八位组的IP地址，用于识别路由。

匹配函数srt_match提取字段定义值，并将对象标识符的后缀与选路表项的IP地址相匹配。

```
/* srt_match.c - srt_match */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <stdarg.h>

/*------------------------------------------------------------------------
 * srt_match - check if a variable exists in the current Routing Table
 *------------------------------------------------------------------------
 */
int srt_match(bindl, rtp, rtl, field)
struct snbentry  *bindl;
struct route     **rtp;
int              *rtl;
int              *field;
{
        int  oidi, i;
        Bool found;
        struct route **rtp;
        va_list ap;
        int *rtl, *field;
```

```
    va_start(ap, bindl);
    rtp = va_arg(ap, struct route **);
    rtl =  va_arg(ap, int *);
    field = va_arg(ap,int *);
    va_end(ap);

    oidi = SRT_OIDLEN;
    if ((*field = bindl->sb_oid.id[oidi++]) > SNUMF_RTTAB)
         return SYSERR;
    /* oidi points to IP address to match in the routing table. */
    for (found = FALSE, i = 0; !found && i < RT_TSIZE; i++)
         for (*rtp = rttable[i]; *rtp; *rtp = (*rtp)->rt_next)
              if (found = soipequ(&bindl->sb_oid.id[oidi],
                             (*rtp)->rt_net, IP_ALEN))
                        break;
    if (!found ||                        /* not there          */
        oidi + IP_ALEN != bindl->sb_oid.len) /* not end of object id*/
         return SYSERR;
    *rtl = i - I;
    return OK;
}
#endif  /* SNMP */
```

srt_match将参数field赋值为对象标识符中的字段标识符部分，并在IP选路表中搜索，找出与对象标识符中给出的IP地址相匹配的表项。在搜索时，它循环经过选路表中的所有表项，并且遍历每个表项中扩展出来的路由链表。如果找到了完全匹配项，srt_match返回OK，否则返回SYSERR。

22.7.1 对选路表的get操作

MIB选路表变量包含了13个字段。最明显的字段分别对应于IP目的地址（ipRouteDest）、到该目的站经过的下一跳地址（ipRouteNextHop）、送往下一跳的数据流经过的接口的索引值（ipRouteIfIndex）。在选路表中还有另外一些字段，分别给出了建立路由的协议（ipRouteProto）、路由类型（ipRouteType）、路由的寿命（ipRouteAge）。函数srt_get实现以选路表为对象的get操作。它调用srt_match找出与给出的对象标识符相匹配的路由，然后利用给出的字段定义值选择正确的程序代码段，以满足请求。

```
/* srt_get.c - srt_get */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
```

```
#include <asn1.h>

/*-----------------------------------------------------------------------
 * srt_get - perform a get on a variable in the Routing Table
 *-----------------------------------------------------------------------
 */
int srt_get(bind1, numifaces)
struct snbentry  *bind1;
int          numifaces;
{
        struct     route *rtp;
        int        rt1, field;

        if (srt_match(bind1, &rtp, &rt1, &field) == SYSERR)
            return SERR_NO_SUCH;
        switch(field) {
        case 1:          /* ipRouteDest */
            SVTYPE(bind1) = ASN1_IPADDR;
            SVIPADDR(bind1) = rtp->rt_net;
            return SNMP_OK;
        case 2:          /* ipRouteIfIndex */
            SVTYPE(bind1) = ASN1_INT;
            SVINT(bind1) = rtp->rt_ifnum;
            return SNMP_OK;
        case 3:          /* ipRouteMetric1 */
            SVTYPE(bind1) = ASN1_INT;
            SVINT(bind1) = rtp->rt_metric;
            return SNMP_OK;
        case 4:          /* ipRouteMetric2 */
        case 5:          /* ipRouteMetric3 */
        case 6:          /* ipRouteMetric4 */
            SVTYPE(bind1) = ASN1_INT;
            SVINT(bind1) = -1;
            return SNMP_OK;
        case 7:          /* ipRouteNextHop */
            SVTYPE(bind1) = ASN1_IPADDR;
            SVIPADDR(bind1) = rtp->rt_gw;
            return SNMP_OK;
        case 8:          /* ipRouteType */
            SVTYPE(bind1) = ASN1_INT;
            if (rtp->rt_metric)
                SVINT(bind1) = 4;      /* remote */
            else
                SVINT(bind1) = 3;      /* direct */
            return SNMP_OK;
```

```
        case 9:         /* ipRouteProto */
                SVTYPE(bindl) = ASN1_INT;
                SVINT(bindl) = 1;           /* other */
                return SNMP_OK;
        case 10:        /* ipRouteAge */
                SVTYPE(bindl) = ASN1_INT;
                SVINT(bindl) = rtp->rt_ttl;
                return SNMP_OK;
        case 11:        /* ipRouteMask */
                SVTYPE(bindl) = ASN1_IPADDR;
                SVIADDR(bindl) = rtp->rt_mask;
                return SNMP_OK;
        case 12:        /* ipRouteMetric5 */
                SVTYPE(bindl) = ASN1_INT;
                SVINT(bindl) = -1;
                return SNMP_OK;
        case 13:        /* ipRouteInfo */
                {
                        struct oid RouteInfo = { {0,0}, 2};
                        SVTYPE(bindI) = ASN1_OBJID;
                        /* set to 0.0, per rfc 1213 */
                        SVOIDLEN(bindl) = RouteInfo.len;
                        memcpy(SVOID(bindl), RouteInfo.id,
                                RouteInfo.len * 2);
                }
                return SNMP_OK;
        default:
                break;
        }
        return SERR_NO_SUCH;
}
#endif  /* SNMP */
```

srt_get通过访问IP选路表结构中适当的字段，处理了大多数常见请求。由于本地选路表不包括多度量路由，因而srt_get对那些具有2到5个度量的路由的请求返回-1。当客户请求取出字段ipRouteProto的值时，srt_get总是返回other（1）代码。虽然表中路由可能是由本地系统建立的，也可能是由RIP或ICMP建立的，但srt_get对此一无所知，因为本地选路表对此不做区分。我们说MIB概念表包含的变量不仅可能不存在，而且可能不可计算，除非对底层系统做巨大的变动。

22.7.2 对选路表的get-first操作

函数srt_getf实现以选路表为对象的get-first操作。

```
/* srt_getf.c - srt_getf */

#include <conf.h>
```

```
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*-----------------------------------------------------------------------
 * srt_getf - perform a getfirst on a variable in the Routing Table
 *-----------------------------------------------------------------------
 */
int srt_getf(bindl, mip, numifaces)
struct snbentry  *bindl;
struct mib_info  *mip;
int              numifaces;
{
        int             rtl, oidi;
        struct route    *rtp = (void *)NULLPTR;

        rtl = -1;  /* use first field, first route */
        if ((rtp = srt_findnext(rtp, &rtl)) == 0) {
             if (mip->mi_next)
                  return (*mip->mi_next->mi_func)
                       (bindl, mip->mi_next, SOP_GETF);
             return SERR_NO_SUCH; /* no next node */
        }
        memcpy(bindl->sb_oid.id, mip->mi_objid.id, mip->mi_objid.len*2);
        oidi = mip->mi_objid.len;

        bindl->sb_oid.id[oidi++] = (u_short) 1;         /* field */
        sip2ocpy(&bindl->sb_oid.id[oidi], rtp->rt_net);
        bindl->sb_oid.len = oidi + IP_ALEN;
        return srt_get(bindl, numifaces);
}
#endif   /* SNMP */
```

当srt_getf在表中搜索第一个路由时，必须确保选出的路由具有字典顺序最小的对象标识符。它调用函数srt_findnext搜索选路表，并取出这样的一个路由。如果表为空，则srt_findnext返回-1，并且srt_getf对MIB中按字典顺序排列的下一个数据项调用get_first函数。如果它找到一个数据项，则srt_getf生成一个正确的对象标识符，并对该数据项调用get操作。

22.7.3　对选路表的get-next操作

选路表上的get-next操作与以前提及的所有表格上的get-next操作都有一个明显的区别：对一个给定的目的站，IP选路表可能含有多个路由。原因很简单：虽然MIB指定了一个目的站地址作为

查找时的关键字，但选路表中可能有多个路由与一个关键字相对应。尤其因为在选路表中，每个目的站都可能含有指定主机的路由、指定子网的路由以及指定网络的路由。如果选路表中某个目的站D的这三种路由全有，会出现什么情况？以D为对象的get-next操作首先找出指定主机的路由，然后使用表中跟在它后面的“下一个”路由。遗憾的是，下一个路由可能恰好是同一目的站的指定子网的路由。当get-next为响应报文生成一个对象标识符时，显然它与请求中的对象标识符将是相同的。因此，接下来的get-next操作会再次与同一目的站的指定主机的路由匹配，导致客户永远无法通过这张表格。

为了解决这个模糊不清的目的地址所引起的问题，我们在get-next的实现中忽略那些与对象标识符中给出的目的地址相同的路由表项。总结如下：

> 如果在MIB选路表中仅使用IP目的地址作为关键字，则无法访问完整的表格，因为IP选路表中一个指定的目的地址可能具有多个路由。在执行get-next操作时，我们的实现会略过与请求具有相同的目的站地址的路由表项。

函数srt_getn实现以选路表为对象的get-next操作。它调用函数srt_findnext处理上述问题。

```
/* srt_getn.c - stc_getn */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*------------------------------------------------------------------------
 * srt_getn - perform a getnext on a variable in the Routing Table
 *------------------------------------------------------------------------
 */
int
srt_getn(struct snbentry *bindl, struct mib_info *mip, int numifaces)
{
        struct route    *rtp;
        int             rtl, field, oidi;

        if (srt_match(bindl,&rtp, &rtl, &field) == SYSERR)
                return SERR_NO_SUCH;
        if ((rtp = srt_findnext(rtp, &rtl)) == 0) {
                rtp = (struct route *) NULL;
                rtl = 0;            /* set route hash table list to 0 */
                rtp = srt_findnext(rtp, &rtl);
                if (++field > SNUMF_RTTAB)
                        return (*mip->mi_next->mi_func)
```

```
                                (bindl, mip->mi_next, SOP_GETF);
        }
        oidi = SRT_OIDLEN;    /* 4.21.1 */

        bindl->sb_oid.id[oidi++] = field;
        sip2ocpy(&bindl->sb_oid.id[oidi], rtp->rt_net;
        bindl->sb_oid.len = oidi + IP_ALEN;

        return stc_get(bindl, numifaces);
}
#endif  /* SNMP */
```

22.7.4　选路表中的递增搜索

srt_findnext 在选路表中查找按字典顺序排列的“下一个”表项。

```
/* srt_findn.c - stc_findnext */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

/*------------------------------------------------------------------------
 * stc_findnext - find next route in the Iexicographic ordering
 *------------------------------------------------------------------------
 */
struct route *
srt_findnext(struct route *rtp, int *rtl)
{
        struct route    *nextrtp = (void *)NULLPTR;
        struct route    *trtp;
        int             i, nextrtl;

        for (i = 0, nextrtl = -1; i < RT_TSIZE; i++)
                for (trtp = rttable[i]; trtp; trtp = trtp->rt_next) {
                        if (*rtl >= 0 &&
                            trtp->rt_net <= rtp->rt_net)
                                continue;
                        if(nextrtl < 0 ||
                            trtp->rt_net < nextrtp->rt_net) {
                                nextrtp = trtp;
                                nextrtl = i;
                        }
                }
        if (nextrtl == -1)    /* no next route found */
```

```
                return 0;
        *rtl = nextrtl;
        return nextrtp;
}
#endif  /* SNMP */
```

如果调用时入口参数rtp的值等于-1，则srt_findnext找出字典顺序最小的表项。而当在调用中指定了特定的路由时，它略过多个具有相同目的地址的路由表项，找出下一个路由。

22.7.5 对选路表的set操作

函数srt_set实现以选路表为对象的set操作。

```
/* srt_set.c - srt_set */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*------------------------------------------------------------------------
 * srt_set - perform a set operation on a variable in the Routing Table
 *------------------------------------------------------------------------
 */
int srt_set(bindl, mip, numifaces)
struct snbentry   *bindl;
struct mib_info   *mip;
int         numifaces;
{
        struct     route *rtp;
        int                rtl, field;

        if (srt_match(bindl, &rtp, &rtl, &field) == SYSERR)
             return SERR_NO_SUCH;
        switch (field) {
        case 1:          /* ipRouteDest */
             if (SVTYPE(bindl) != ASN1_IPADDR)
                 return SERR_BAD_VALUE;
             rtp->rt_net = SVIPADDR(bindl);
             return SNMP_OK;
        case 2:          /* ipRouteIfIndex */
             if (SVTYPE(bindl) != ASN1_INT ||
```

```
            SVINT(bindl) <= 0 || SVINT(bindl) > numifaces)
             return SERR_BAD_VALUE;
        rtp->rt_ifnum = SVINT(bindl);
        break;
    case 3:         /* ipRouteMetric1 */
        if (SVTYPE(bindl) != ASN1_INT || SVINT(bindl) < 0)
             return SERR_BAD_VALUE;
        rtp->rt_metric = SVINT(bindl);
        break;
    case 4:         /* ipRouteMetric2 */
    case 5:         /* ipRouteMetric3 */
    case 6:         /* ipRouteMetric4 */
        break;
    case 7:         /* ipRouteNextHop */
        if (SVTYPE(bindl) != ASN1_IPADDR)
             return SERR_BAD_VALUE;
        rtp->rt_gw = SVIPADDR(bindl);
        break;
    case 8:         /* ipRouteType */
        /* route type is invalid (2) ==> should remove route
           from routing table */
        if (SVTYPE(bindl) != ASN1_INT ||
            SVINT(bindl) < 1 || SVINT(bindl) > 4)
             return SERR_BAD_VALUE;
        if (SVINT(bindl) == 2)     /* route invalid */
             (void) rtdel(rtp->rt_net, rtp->rt_mask);
        break;
    case 9:         /* ipRouteProto */
        break;
    case 10:        /* ipRouteAge */
        if (SVTYPE(bindl) != ASN1_INT ||
            SVINT(bindl) < 0)
             return SERR_BAD_VALUE;
             rtp->rt_ttl = SVINT(bindl);
             break;
        defaulIt:
             return SERR_NO_SUCH;
        }
        return SNMP_OK;
}
#endif  /* SNMP */
```

对选路表中的大多数字段来说，srt_set 将设置请求转换为适当的对选路表中某个字段的设置。对于本地软件没有提供的字段的设置（如多度量路由）将不起任何作用。srt_set 在设置之前检查绝大多数给定的值，以确保这些值有效。

srt_set代码中包含了一个有趣的特例。协议标准规定，将字段ipRouteType设置为invalid，表示该路由应当从表中删除。如果某个请求将ipRouteType字段设置为invalid(2)，则srt_set调用rtdel来删除该路由。

22.8 TCP连接表函数

MIB定义了一个包含所有活动的(active)TCP连接的表格。TCP连接表的对象标识符具有如下的通用格式：

standard-MIB-prefix.tcp.tcpConnTable.tcpConnEntry.remainder

其中remainder具有五个字段：

field.localIP.localport.remoteIP.remoteport

正如我们所预料的，对象标识符包含有标准的MIB前缀，指出数据项位于TCP子级(subhierarchy)，然后指明连接表以及连接表中的表项。标识符中的localIP、localport、remoteIP和remoteport给出了连接两端点的IP地址和协议端口号，而field用于选择表项中的字段。与其他表格一样，有一个匹配函数用于验证对象标识符是否在连接表中，并提取字段值。过程stc_match完成这一匹配任务。

```
/* stc_match.c - stc_match */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asnl.h>
#include <stdarg.h>

/*------------------------------------------------------------------------
 * stc_match - check if a variable exists in the TCP connections table
 *------------------------------------------------------------------------
 */
int stc_match(struct snbentry *bindl,
                    /* int *field, int *tcbn */)
{
        int         oidi = STC_OIDLEN;
        IPaddr      lip, rip;
        int         lport, rport;
        va_list     op;
        int *field, *tcbn;
```

```
    va_start(ap,bindl);
    field = va_arg(ap, int *);
    tcbn = va_arg(ap, int *);
    va_end(ap);

    if ((*field = bindl->sb_oid.id[oidi++]) > SNUMF_TCTAB)
         return SYSERR;
    so2ipcpy((u_char *)&lip, &bindl->sb_oid.id[oidi]);
    oidi += IP_ALEN;
    lport = bindl->sb_oid.id[oidi++];
    so2ipcpy((u_char *)&rip, &bindl->sb_oid.id[oidi]);
    oidi += IP_ALEN;
    rport = bindl->sb_oid.id[oidi++];

    for (*tcbn = 0; *tcbn < Ntcp; ++(*tcbn)) {
         if (tcbtab[*tcbn].tcb_state == TCPS_FREE)
              continue;
         if (lport == tcbtab[*tcbn].tcb_lport &&
             rport == tcbtab[*tcbn].tcb_rport &&
             rip = tcbtab[*tcbn].tcb_rip &&
             lip = tcbtab[*tcbn].tcb_lip)
              break;
    }
    if (*tcbn >= Ntcp || oidi != bindl->sb_oid.len)
         return SYSERR;
    return OK;
}
#endif  /* SNMP */
```

在提取了字段定义值之后，stc_match 在 TCB 表中搜索，找出与对象标识符指明的连接两端点相匹配的有效表项。因为 TCP 使用了连接的两端点来识别一个连接，所以 stc_match 要对本地和远程的地址和端口对都进行比较。如果 stc_match 找到一个与对象标识符相匹配的连接，它将参数 tcbn 设置为 tcbtab 表的一个索引值，从该表项中可找到有关这个连接的信息，并返回 OK。否则返回差错代码 SYSERR。

22.8.1　对 TCP 连接表的 get 操作

在 MIB 的 TCP 连接表中，每个表项具有五个字段，分别对应于连接状态、本地 IP 地址、本地 TCP 端口号、远程 IP 地址以及远程 TCP 端口号。过程 stc_get 通过字段选择器在这五个数据项中选取操作对象，以实现连接表上的 get 操作。

```
/* stc_get.c - stc_get */

#include <conf.h>
#include <kernel.h>
#include <network.h>
```

```
#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*-----------------------------------------------------------------------
 * stc_get - perform a get on a variable in the TCP connections table
 *-----------------------------------------------------------------------
 */
int stc_get(bindl, numifaces)
struct snbentry  *bindl;
int              numifaces;
{
        int  field, tcbn;

        if (stc_match(bindl, &field, &tcbn) == SYSERR)
                return SERR_NO_SUCH;


        switch (field) {
        case 1:         /* tcpConnState */
                SVTYPE(bindl) = ASN1_INT;
                SVINT(bindl) = tcbtab[tcbn].tcb_state;
                break;
        case 2:         /* tcpConnLocalAddress */
                SVTYPE(bindl) = ASN1_IPADDR;
                SVIPADDR(bindl) = tcbtab[tcbn].tcb_lip;
                break;
        case 3:         /* tcpConnLocalPort */
                SVTYPE(bindl) = ASN1_INT;
                SVINT(bindl) = tcbtab[tcbn].tcb_lport;
                break;
        case 4:         /* tcpConnRemAddress */
                SVTYPE(bindl) = ASN1_IPADDR;
                SVIPADDR(bindl) = tcbtab[tcbn].tcb_rip;
                break;
        case 5:         /* tcpConnRemPort */
                SVTYPE(bindl) = ASN1_INT;
                SVINT(bindl) = tcbtab[tcbn].tcb_rport;
                break;
        default:
                return SERR_NO_SUCH;
        }
        return SNMP_OK;
}
```

```
#endif  /* SNMP */
```

22.8.2　对 TCP 连接表的 get-first 操作

过程 stc_getf 提供了以 TCP 连接表为对象的 get-first 操作。stc_getf 调用 stc_findnext 在 tcbtab 表中搜索，直到找出具有最小字典顺序标识符的已分配 TCB。如果没有任何 TCB 被分配，则 MIB 连接表为空，所以 stc_getf 对 MIB 表中按字典顺序排列的下一个变量执行 get-first 操作，并返回其结果。如果 stc_findnext 找到一个有效的连接，则 stc_getf 为该连接生成一个新的对象标识符，以它为对象执行 get 操作，并返回其结果。

```
/* stc_getf.c - stc_getf */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/*------------------------------------------------------------------------
 * stc_getf - do a getfirst on a variable in the TCP connection table
 *------------------------------------------------------------------------
 */
int stc_getf(bindl, mip, numifaces)
struct snbentry *bindl;
struct mib_info *mip;
int         numifaces;
{
        int  oidi, tcbn;

        /* find first connection, if any */
        tcbn = stc_findnext(-1);
        if (tcbn < 0) {
             if (mip->mi_next)
                  return((*mip->mi_next->mi_func)
                       (bindl, mip->mi_next, SOP_GETF));
             return SERR_NO_SUCH;
        }
        memcpy(bindl->sb_oid.id, mip->mi_objid.id, mip->mi_objid.len*2);

        oidi = mip->mi_objid.len;

        bindl->sb_oid.id[oidi++] = (u_short) 1;   /* field */
        sip2ocpy(&bindl->sb_oid.id[oidi], tcbtab[tcbn].tcb_lip);
```

```
        oidi += IP_ALEN;
        bindl->sb_oid.id[oidi++] = (u_short) tcbtab[tcbn].tcb_lport;
        sip2ocpy(&bindl->sb_oid.id[oidi], tcbtab[tcbn].tcb_rip);
        oidi += IP_ALEN;
        bindl->sb_oid.id[oidi++] = (u_short) tcbtab[tcbn].tcb_rport;
        bindl->sb_oid.len = oidi;

        return stc_get(bindl, numifaces);
}
#endif  /* SNMP */
```

22.8.3 对TCP连接表的get-next操作

过程stc_getn实现以TCP连接表为对象的get-next操作。

```
/* stc_getn.c - stc_getn */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asnl.h>

/*------------------------------------------------------------------------
 * stc_getn - do a getnext on a variable in the TCP connection table
 *------------------------------------------------------------------------
 */
int stc_getn(bindl, mip, numifaces)
struct snbentry  *bindl;
struct mib_info  *mip;
int              numifaces;
{
        int  oidi, field, tcbn;

        if (stc_match(bindl,&field,&tcbn) == SYSERR) {
                return SERR_NO_SUCH;
        }
        /* search for next connection */
        if ((tcbn = stc_findnext(tcbn)) < 0) {
                tcbn = stc_findnext(-1);
                if (++field > SNUMF_TCTAB)
                        return((*mip->mi_next->mi_func)
```

```
                    (bindl, mip->mi_next, SOP_GETF));
    }
    oidi = STC_OIDLEN;    /* 6.13.1 */

    bindl->sb_oid.id[oidi++] = (u_short) field;
    sip2ocpy(&bindl->sb_oid.id[oidi], tcbtab[tcbn].tcb_lip);
    oidi += IP_ALEN;
    bindl->sb_oid.id[oidi++] = (u_short) tcbtab[tcbn].tcb_lport;
    sip2ocpy(&bindl->sb_oid.id[oidi], tcbtab[tcbn].tcb_rip);
    oidi += IP_ALEN;
    bindl->sb_oid.id[oidi++] = (u_short) tcbtab[tcbn].tcb_rport;
    bindl->sb_oid.len = oidi;

    return stc_get(bindl, unmifaces);
}
#endif  /* SNMP */
```

stc_getn利用stc_match找出与指定标识符匹配的TCB。然后调用过程stc_findnext找出下一个有效的连接。一旦stc_findnext在tcbtab表中找到了有效的表项，它会返回该表项的索引值。如果stc_findnext找到了一个有效的连接，stc_getn为该表项生成一个对象标识符，执行get操作，并返回结果。

若没有有效的表项，stc_getn将字段定义值加1，回到tcbtab表的头部，继续搜索。最后，当循环通过了所有有效连接上的所有字段之后，stc_getn以MIB表中按字典顺序排列的下一个MIB变量执行get-first操作，并返回结果。

22.8.4　TCP 连接表中的递增搜索

过程stc_findnext在TCB表中搜索，找出按字典顺序排列在由参数tcbn指定的TCB之后的下一个TCB。

```
/* stc_findn.c - stc_findnext, sntcpcmp */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP
#include <snmp.h>

/*------------------------------------------------------------------------
 * stc_findnext - search the TCP connection table for the next valid entry
 *------------------------------------------------------------------------
 */
int stc_findnext(tcbn)
int    tcbn;
{
```

```
        int  i, next;

        for (i = 0, next = -1; i < Ntcp; ++i) {
             if (tcbtab[i].tcb_state == TCPS_FREE ||
                 (tcbn >= 0 && sntcpcmp(i, tcbn) <= 0))
                   continue;
             if (next < 0 || sntcpcmp(i, next) < 0)
                   next = i;
        }
        return next;
}

/*------------------------------------------------------------------------
 * sntcpcmp - compare two TCP connections in SNMP lexical ordering
 *------------------------------------------------------------------------
 */
int sntcpcmp(tcb1, tcb2)
int    tcb1, tcb2;
{
        int  rv;

        if ((rv = (tcbtab[tcb1].tcb_lip - tcbtab[tcb2].tcb_lip)))
             return rv;
        if ((rv = (tcbtab[tcb1].tcb_lport - tcbtab[tcb2].tcb_lport)))
             return rv;
        if ((rv = (tcbtab[tcb1].tcb_rip - tcbtab[tcb2].tcb_rip)))
             return rv;
        if ((rv = (tcbtab[tcb1].tcb_rport - tcbtab[tcb2].tcb_rport)))
             return rv;
        return 0;
}
#endif  /* SNMP */
```

要得出MIB为两个连接之间定义的字典顺序，软件必须对连接两端点的四个因素都进行比较。本地IP地址是最重要的字段，本地协议端口和远程IP地址稍次之，而远程协议端口号的份量最轻。函数sntcpcmp根据字典顺序比较连接两端点。如果它们相等，则返回0。如果第一个连接按字典顺序小于第二个连接，则返回一个小于0的值。如果第一个连接按字典顺序大于第二个连接，则返回一个大于0的值。

22.8.5 对TCP连接表的set操作

MIB允许将TCP连接的状态改变为deleteTCB（值为12）来关闭这个连接。过程stc_set实现了set操作。

```
/* stc_set.c - stc_set */
```

```
#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>
#include <io.h>

/*------------------------------------------------------------------------
 * stc_set - set TCP table entities
 *------------------------------------------------------------------------
 */
int
stc_set(struct snbentry *bindl, struct mib_info *mip, int numifaces)
{
        int tcbn, field;

        if (stc_match(bindl, &field, &tcbn) == SYSERR)
                return SERR_NO_SUCH;

        swtch (field) {
        case 1:         /* tcpConnState */
                /* the only thing that you can legally change
                   this to is the integer deleteTCB(12) */
                if (SVTYPE(bindl) != ASN1_INT)
                        return SERR_BAD_VALUE;
                if (SVINT(bindl) != 12)
                        return SERR_BAD_VALUE;
                close(tcbtab[tcbn].tcb_dvnum);
                return SNMP_OK;
        default:
                return SERR_NO_SUCH;
        }
        return SNMP_OK;

}
#endif  /* SNMP */
```

过程 stc_set 表明 set 操作将 tcpConnState 字段改为整型变量 deleteTCB（12），然后关闭相关连接。

22.9 UDP Listener 表

MIB定义了一个与监听UDP端口的服务器设置一致的列表。UDP listener表的对象标识符具有如下形式：

standard-MIB-prefix.udp.udpTable.udpEntry.remainder

其中remainder包含三个字段：

field.localIP.localport

正如所有的MIB变量，对象标识符以标准的MIB前缀起始，然后说明了UDP分层结构、UDP listener表及其表项。对象标识符的localIP和localport部分说明了服务器正在监听的IP地址和协议端口号，而field在表项的各字段中查找。一般来说，一个匹配的函数确认标识符与链接表正确匹配，然后提取字段值。该过程为sud_match。

```
/* sud_match.c - sud_match */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef  SNMP

#include <snmp.h>
#include <mib.h>
#include <asnl.h>
#include <stdarg.h>

/* ------------------------------------------------------------------------
 * sud_match - check if a variable exists in the UDP listener table
 * ------------------------------------------------------------------------
 */
int
sud_match(struct snbentry *bindl,
          ... /* int *field, int *udpn */ )
{
        int     oidi = SUD_OIDLEN;
        int     lport;
        va_list ap;
        int *field, *udpn;
        va_start(ap, bindl );
        field = va_arg(ap, int *);
        udpn = va_arg(ap, int *);
        va_end(ap);

        if ((*field = bindl->sb_oid.id[oidi++]) > SNUMF_UDTAB)
```

```
                return SYSERR;

        oidi += IP_ALEN;    /* skip the IP address */
        lport = bindl->sb_oid.id[oidi++];

        for (*udpn = 0; *udpn < UPPS; ++(*udpn))  {
                if (!upqs[*udpn].up_valid)
                        continue;
                if (lport == upqs[*udpn].up_port)
                        break;
        }
        if (*udpn >= UPPS || oidi != bindl->sb_oid.len)
                return SYSERR;
        return OK;
}

#endif    /* SNMP */
```

sud_match 必须检查对象标识符中说明的端口号，以证实它已在有效端口的当前列表中。为达到这个目的，sud_match 需要检查 upqs 中所有表项的 up_port 字段是否有效。如果是则返回 OK，否则返回 SYSERR。

22.9.1　对 UDP Listener 表的 get 操作

过程 sud_get 实现了 get 操作。它利用字段选择器在 IP 地址和协议端口字段之间进行选择。

```
/* sud_get.c - sud_get */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef  SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.b>

/* ------------------------------------------------------------------------
 * sud_get - perform a get on a variable in the UDP listener table
 * ------------------------------------------------------------------------
 */
int
sud_get(struct snbentry *bindl, int numifaces)
{
        int      field, udpn;

        if (sud_match(bindl, &field, &udpn) == SYSERR)
                return SERR_NO_SUCH;
```

```
        switch (field) {
        case 1:         /* udpLocalAddress */
                {
                IPaddr ip_allzero = 0;
                SVTYPE(bind1) = ASN1_IPADDR;
                /* accept any, rfc 1213 */
                SVIPADDR (bind1) =ip_allzero;
                }
                break;
        case 2:         /* udPLocalPort */
                SVTYPE(bind1) = ASN1_INT;
                SVINT(bind1) = upqs[udpn].up_port;
                break;
        default:
                return SERR_NO_SUCH;
        }
        return SNMP_OK;
}
#endif   /* SNMP */
```

sud_get 总是返回 0.0.0.0 的 IP 地址，用以响应字段 udpLocalAddress 的 get 操作。协议标准这样规定是因为 UDP listener 随时准备接受发向任何主机 IP 地址的 UDP 数据报。

22.9.2 对 UDP Listener 表的 get-first 操作

过程 sud_getf 是处理 UDP listener 表中 get-first 操作的。它调用 sud_findnext 在所有活动的 UDP 端口中查找端口号最小的。如果没有活动的 UDP 端口，sud_getf 返回 SERR_NO_SUCH。否则，它将为端口创建一个有效的对象标识符，然后执行 get 操作，返回结果。

```
/* sud_getf.c - sud_getf */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef  SNMP

#include <snmp.h>
#include <mib.h>
#include <asn1.h>

/* ------------------------------------------------------------------------
 * sud_getf - do a getfirst on a variable in the UDP listener table
 * ------------------------------------------------------------------------
 */
int
sud_getf(struct snbentry *bind1, struct mib_info *mlp, int numifaces)
```

```
{
        int     oidi, udpn;
        IPaddr ip_allzero = 0;


        /* find first connection, if any */
        udpn = sud_findnext (-1);
        if (udpn < 0) {
                if (mip->mi_next)
                        return (( *mip->mi_next->mi_func)
                                (bindl, mip->mi_next, SOP_GETF));
                        return SERR_NO_SUCH;
        }
        memcpy(bindl->sb_oid.id, mip->mi_objid.id, mip->mi_objid.len*2);
        oidi = mip->mi_objid.len;

        bindl->sb_oid.id[oidi++] = (u_short)  1;            /* field */
        sip2ocpy(&bindl->sb_oid.id[oidi], ip_allzero);
        oidi += IP_ALEN;
        bindl->sb_oid.id[oldi++] = (u_short) upqs[udpn].up_port;
        bindl->sb_oid.len = oidi;

        return sud_get (bindl, numifaces);
}
#endif   /* SNMP */
```

22.9.3　对 UDP Listener 表的 get-next 操作

过程 sud_getn 实现 UDP listener 表中 get-next 操作。它调用 sud_match 查找 udps 数组中与对象标识符一致的 UDP 活动端口；然后调用 sud_findnext 获得下一个有效表项。如果找到有效表项，则为该表项创建对象标识符，调用 sud_get 执行 get 操作，然后返回结果。否则，在字段说明中增值，重新开始查找。

```
/* sud_getn.c - sud_getn * /

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef  SNMP

#include <snmp.h>
#include <mib.h>
#include <asnl.h>
/* ------------------------------------------------------------------------
 * sud_getn - do a getnext on a variable in the UDP listener table
 * ------------------------------------------------------------------------
 */
```

```
int
sud_getn(struct snbentry *bindl, struct mib_info *mip, int numifaces)
{
        int     oidi, field, udpn;
        IPaddr ip_allzero = 0;
        if (sud_match(bindl,&field,&udpn) == SYSERR) {
                return SERR_NO_SUCH;
        }
        /* search for next connection */
        if ((udpn = sud_findnext(udpn)) < 0) {
                udpn = sud_findnext(-1);
                if (++field > SNUMF_UDTAB) {
                        /* end of table, go to next variable */
                        if (mip->mi_next == (void *)NULLPTR) {
                            /* end of mib */
                            return SERR_NO_SUCH;
                        }
                        return (( *mip->mi_next->mi_func )
                                (bindl, mip->mi_next, SOP_GETF));
                }
        }
        oidi = SUD_OIDLEN;          /* 7.5.1 */

        bindl->sb_oid.id[oidi++] = (u_short) field;

        sip2ocpy(&bindl->sb_oid.id[oidi], ip_allzero);
        oidi += IP_ALEN;
        bindl->sb_oid.id[oidi++] = (u_short)upqs[udpn].up_port;
        bindl->sb_oid.len = oidi;

        return sud_get(bindl, numifaces);
}
#endif   /* SNMP */
```

22.9.4 UDP Listener表中的递增搜索

在活动的UDP端口列表中，过程sud_findn搜索按字典顺序紧接着由参数udpn说明的那个活动的端口。

```
/* sud_findn.c - sud_findnext,snudpcmp */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef  SNMP
#include <snmp.h>
```

```
/* ------------------------------------------------------------------------
 * sud_findnext - search the UDP connection table for the next valid entry
 * ------------------------------------------------------------------------
 */
int
sud_findnext(int udpn)
{
        int     i, next;


        for (i = 0, next = -1; i < UPPS; ++i)  {
                if ( !upqs[i].up_valid)
                        continue;
                /* had one before, and this one isn't bigger */
                if ((udpn >= 0) &&
                    (upqs[i].up_port <= upqs[udpn].up_port) )
                        continue;
                 /* haven't found one yet, or we did, but this */
                 /* one is smaller */
                 if ((next < 0) ||
                     (upqs[i].up_port < upqs[next].up_port))
                         next = i;
        }
        return next;
}
#endif   /* SNMP */
```

与 TCP 链接表的查找过程相似，sud_findnext 跳过与活动端口不一致的表项（即表项中的 up_valid 字段为 false）。但是与 TCP 链接表的查找过程不同的是，UDP listener 表中查找过程不检查多变量，相反，sud_findnext 只查看在字段 up_port 中的 UDP 端口号，选择比 udpn 值更大的下一个活动端口。

22.9.5　对 UDP Listener 表的 set 操作

协议标准规定 UDP listener 列表中的所有变量都是只读的。这样，当服务器接到 set 请求时必须返回差错值。过程 sud_set 返回差错值。

```
/* sud_set.c - sud_set */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef  SNMP

#include <snmp.h>
```

```
#include <mib.h>
#include <asn1.h>

/* ------------------------------------------------------------------------
 * sud_set - return error: the UDP Listener Table is read-only
 * ------------------------------------------------------------------------
 */
int
sud_set (struct snbentry *bindl, struct mib_info *mip, int numifaces)
{
        return SERR_NO_SUCH;
}
#endif   /* SNMP */
```

22.10 IP地址转换的实用例程

范例代码中的许多过程都需要在以32位二进制值表示的IP地址与SNMP所用的形式之间进行转换。文件snoip.c包含了三个这样的实例。sip2ocpy连续提取IP地址中的八位组，然后将每一个都复制到对象ID结构中。so2ipcpy执行相反的操作，将IP地址中的八位组从对象ID的形式转换为以32位二进制值表示的形式。最后，soipequ将以这两种形式表示的IP地址进行比较，判断它们是否包含相同的值。

```
/* snoip.c - sip2ocpy, so2ipcpy, soipequ */

#include <conf.h>
#include <kernel.h>
#include <network.h>

#ifdef  SNMP

#include <snmp.h>
/* ------------------------------------------------------------------------
 * sip2ocpy - copy an IP address into the object id structure
 * ------------------------------------------------------------------------
 */
void
sip2ocpy(u_short *oidp, IPaddr pip)
{
        int     i;
        for (i = 0; i < IP_ALEN; i++) {
            *oidp++ = ((u_short)  ((char *)&pip)i]) & 0xff;
        }

}
```

```
/* ----------------------------------------------------------------------
 * so2ipcopy - copy 4 sub-object ids into an IP address array
 * ----------------------------------------------------------------------
 */
void
so2ipcpy(u_char *pip, u_short *oidp)
{
        int     i;

        for (i = 0; i < IP_ALEN; i++)
            pip[i] = (u_char) oidp[i] & 0xff;

/* ----------------------------------------------------------------------
 * soipequ - test equality between an object id and an IP address
 * ----------------------------------------------------------------------
 */
int
soipequ(u_short *oidp, IPaddr ip, int len)
{
        int     i;
        for (i = 0; i < len; i++) {
            u_char *pip = (u_char *)&ip;
            if ((u_short) pip[i] != (u_short) oidp[i])
                return (FALSE);
        }
        return (TRUE);
}
#endif  /* SNMP */
```

22.11　小结

表格访问函数与访问简单变量的函数不同，因为服务器必须将部分对象标识符解释为表格的索引。get 和 set 操作要求客户提供表项的全名。get-next 操作使客户能够在不知道确切名称的情况下，跨过表格中的所有数据项。

我们讨论了 MIB 地址入口表、网络接口表、地址转换表、IP 选路表以及 TCP 连接表。每张表格都要求具有支持 get、get-next 和 set 操作的实例程序。另外，在我们的实现中，每个表格还具有一个 match 函数、一个 findnext 函数和一个 get-first 函数。match 函数用于将对象标识符与表中的有效表项相匹配，findnext 函数用于找出按字典顺序排序的下一个表项，而 get-first 函数处理以空表为对象的 get-next 操作。

深入研究

在 McCloghrie 和 Rose［RFC 1213］和［RFC 1155］中可找到有关 MIB 变量名的更多细节。

McCloghrie [RFC 2233] 讨论了MIB-II中的接口群。McCloghrie和Rose [RFC 1213] 对本书用到的变量进行了定义。许多RFC中都为具体的硬件设备使用的MIB变量提出了建议。可以就本书中的范例参阅RFC索引。

习题

22.1 网络管理员是否可以设置地址转换表项中的硬件地址？为什么？

22.2 在ipRoutingTable表项中，IP地址值为128.10.2.3的ipRouteNextHop字段的ASN.1对象标识符的精确格式是什么？

22.3 SNMP能访问IP选路表中所有表项的所有字段吗？为什么？

22.4 如果客户发出的SNMP get-next请求中的对象标识符，指定的是地址转换表中的最后一个表项，服务器将返回什么？

22.5 SNMP能够用来为本章中提到的某个表格创建新表项吗？请解释。

22.6 我们的实现方案中没有提供EGP信息表。阅读MIB规范，并写出EGP表要求的所有字段的列表。

第23章　实现的回顾

23.1　引言

本章简要地对书中提供的实现范例做一回顾。在本章中，我们将分析程序代码，从一个实现范例的角度指出协议中的优势与弱点，并得出有关协议软件设计的几点结论。

23.2　程序代码统计分析

本书所涉及的软件系统十分庞大，对其进行分析是比较困难的，因为只用一种手段无法提供准确的评估。此外，我们也不可能对其“难度”或“力度”进行估价，因为设计所需的“人月”工作量取决于设计人员的技巧能力与背景。因此我们有意回避这一类的评估，而对最终得到的程序代码做一客观测度。

有两种软件评估方法可以证实我们直觉上的估计，并提供了足够的数据以支持我们得出的几点结论。其中第一种方法是计算用于实现某一协议的函数和过程的数目，第二种方法是计算程序代码行数。虽然我们知道在一个软件中各个过程所占比例的大小取决于程序员，且程序代码行数取决于编程风格，但是通过这种方法还是可以得知实现主要协议的相对复杂性。

23.3　各个协议的程序代码行数

软件评估的第一种方法是考虑实现每个协议所需程序代码的行数。这些代码不仅包括本书中讨论的所有程序段，还包括一些本书中未提供的依赖于机器的支持过程。它们被分为11组[1]：TCP、SNMP、OSPF、IP（包括所有处理选路、数据报的分片和组装以及广播等方面）、NET（网络设备驱动程序和网络接口例程）、OTHER、ARP、IGMP、RIP、ICMP和UDP。其中OTHER组中包括一些实用程序，如校验和过程及初始化例程。在评估时不包括其他操作系统的函数或其他一些协议的程序代码（例如，我们有意忽略的rwho协议的程序代码）。

整个程序代码包括大约16 000行。图23.1中所示为每组程序的代码行数在总代码中所占的百分比。从图中可看出，TCP所需要代码最多，约占19%。当然，在所有有关的协议中，TCP提供的功能最强，并且处理的差错类型也最多。与此相反，因为IP不需要做复杂的重发和确认工作，所以IP需要的代码在行数上只是TCP的一半稍多一些。甚至IP、ICMP和UDP三个协议的程序代码的总和也没有TCP所需的代码多。

网络接口的程序代码似乎有些过多，但读者应当想到，它不仅包括设备驱动程序代码，还有IP和设备之间的接口程序代码。设备驱动程序本质上是依赖于硬件的。通常它们需要许多行的代码用以处理DMA存储器、硬件接口以及硬件中断处理。这样看来，接口的确需要大量的代码。

---

① 所列的各组根据程序大小降序排列。

惟一令人惊奇的是SNMP和OSPF所需的程序代码量。虽然SNMP承担的任务要比TCP简单得多，但SNMP的程序代码却几乎与TCP一样多。原因有两个：首先，SNMP规定MIB变量要与TCP/IP栈中的数据结构相一致，因此实现代码必须包含独立的例程，将SNMP操作译成本地数据结构。再者，与TCP不同，SNMP设计方案中为每个字段使用了可变长度的编码值（包括长度字段本身也是如此）。因此，从SNMP报文中提取一个简单的整型值也需要许多计算，虽然当前的计算机可能无法处理大于32位的整数。事实上，SNMP程序包含了为冗长的程序设计语言使用的特定的分析程序。总结如下：

因为SNMP使用可变长度的字段，这使它的程序代码段比其他TCP/IP协议的要大。

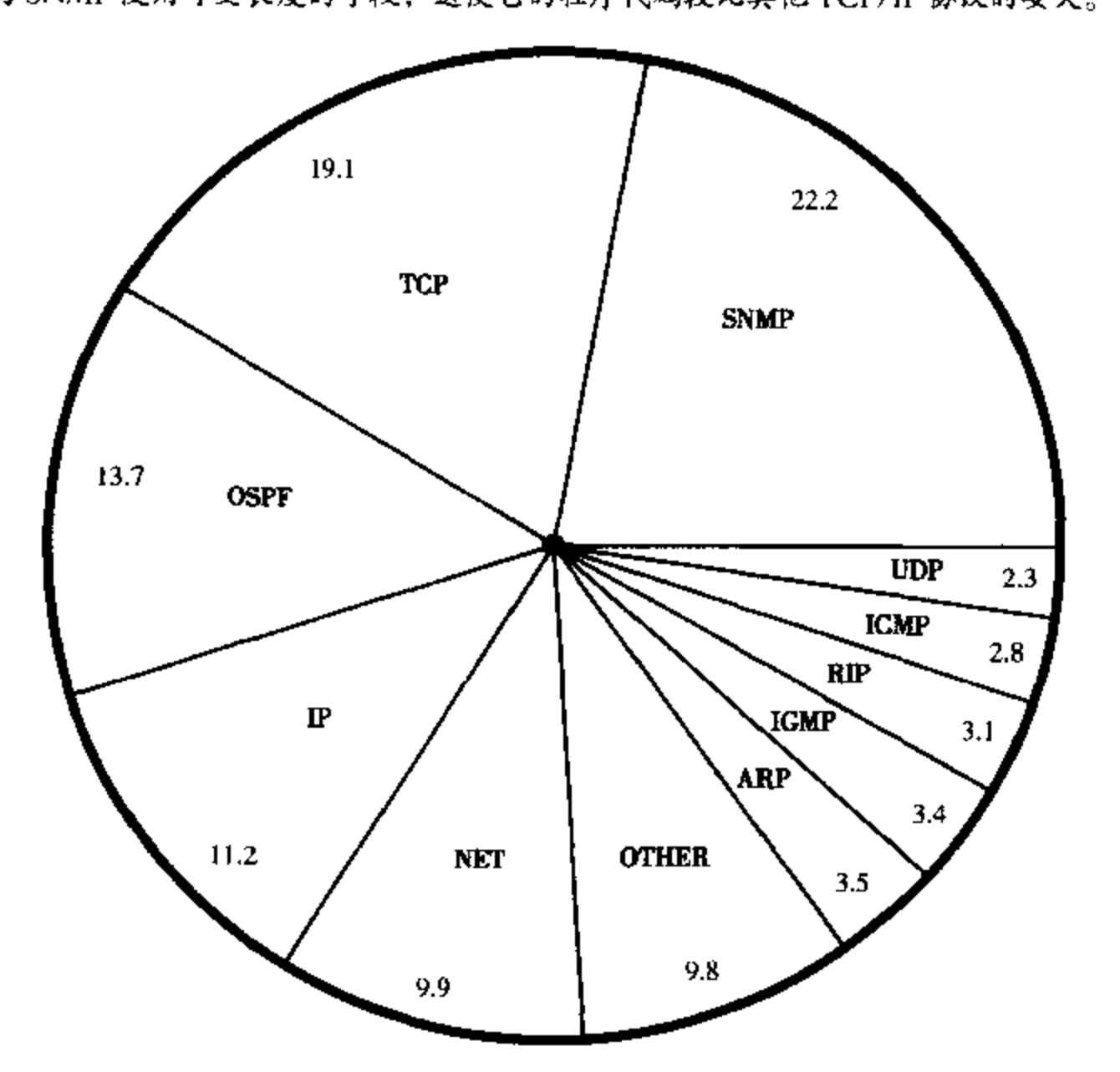


图23.1 各个主要协议在实现时需要的程序代码在总程序中所占的比例。在OTHER组中包括混杂的实用函数，譬如用于计算校验和的程序以及初始化程序代码

OSPF程序代码的大小似乎也是惊人的，它大约是RIP所需的程序代码的4倍。因为OSPF试图容纳各种各样的互联网拓扑图，它必须处理多种选择。OSPF的过程代码可能与比我们通常见到的版本要大，因为它必须允许网络管理员将匿名系统划分为多个区域并引入外部路由。

23.4 每个协议所需的函数和过程

图23.2所示为另一种评估程序代码的方式。它报告了实现每个协议所需的过程和函数数目在整个系统使用的约385个例程中所占的百分比。这个数目不包括操作系统中那些与TCP/IP软件无关

的过程和通用函数库例程。

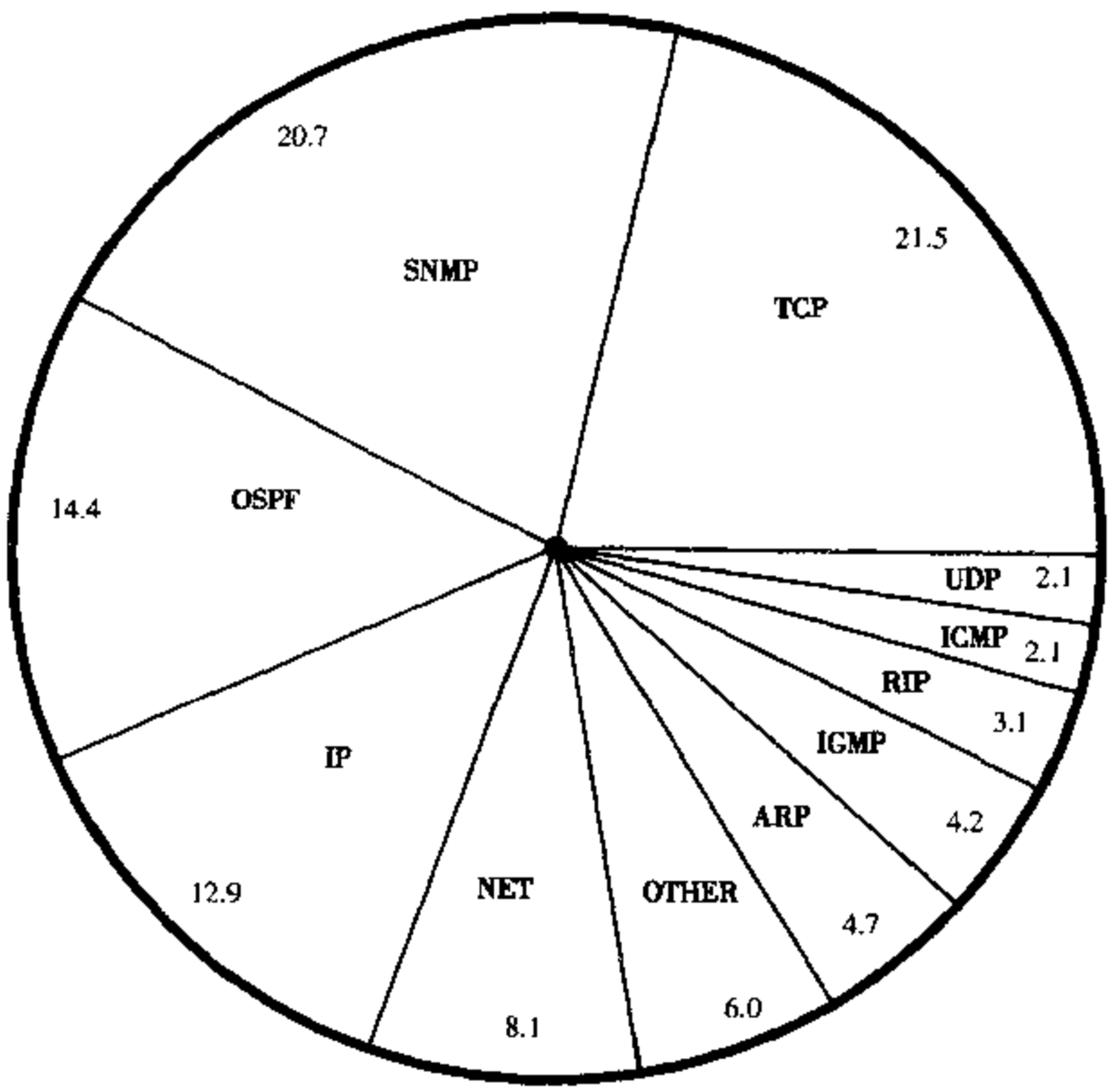


图 23.2　用于实现各个协议的函数和过程的数目在总数中所占的比例。图中顺序与图 23.1 相同

当然，各人的编程风格会影响使用的过程数目。虽然一些程序员就本书的要求编写了一些程序代码，但他们都是从研究 Xinu 系统开始的，并采用了 Xinu 的风格。某个成员编写绝大多数的程序代码，而所有代码都被审查过，以保证它适合于所要求的风格。因此，看来假设本书中编程风格相对统一是有理由的，并且各个程序人员之间能力技巧的不同也不会带来太大的影响。

23.5　小结

对程序代码的分析结果并不令人感到意外。正如我们所料，SNMP需要大量的程序代码，因为MIB定义了许多变量，每一个都需要一个独立的过程，而且可变长度的编码使得程序代码更加繁重。TCP协议所需的代码最多，因为它提供最大的服务量并处理最多的故障。事实上，TCP比IP、ICMP、UDP 和 ARP 的程序代码的总和还要多一些。网络接口的程序代码也占了很大的比例，因为其中包括了设备驱动程序。

习题

23.1 本章提供的分析报告来自一份 1998 年版的代码。找一份机读软件，并对本章中提供的数据与从软件的近期版本得出的数据进行比较。

23.2 在计算SNMP代码量时是否应当包括ASN.1过程？为什么？

23.3 范例代码提供了网关功能。如果该程序运行在只与一个网络相连的主机上时，会减少多少百分比的代码？

23.4 找一份本书中描述的机读软件。在OTHER组中，域名系统软件所占的比例为多少？请解释。

23.5 在前面的分析中，每个协议用直接插入代码来为SNMP中所用的变量增加记数。估计一下，如果不用计数器，应该减少多少行代码？

23.6 通过估算每个协议使用的数据结构的大小，估计每个协议一共需要多大的存储空间。

23.7 为了验证各个协议的程序说明的风格是一致的，可以找一份机器可读软件，删除程序中所有的说明行和不必要的空格后，得到一个新程序，并对所有过程做这一处理。对不同的协议来说，压缩后的代码与未压缩的代码所占的比例有什么不同吗？

23.8 将我们的TCP/IP实现范例与另一种实现（如以BSD UNIX系统为基础）相比较。它们之间在程序代码行数以及函数、过程数量上有很大的不同吗？

23.9 扩充我们的OSPF实现方案，使系统管理员能够对区域和接口费用进行设置。这又需要多少代码？

附录1　过程调用交叉参考表

引言

由于任何一个像TCP这样庞大的软件系统都是由保存在上百个文件中的上百个过程组成的，因此研究这样的程序代码并不容易。阅读一个庞大的系统内的程序代码时，仅仅为了找出某个过程或变量而带来的问题就十分令人头痛。尤其是对查找那些调用了特定过程的某个过程来说是最困难的，因为这个过程可能出现在任意的地方。

计算机科学家们用“调用图”（call graph）对过程之间的调用关系进行形式化处理。一组过程的调用图是一张有向图，其中每个结点对应一个过程，而一条从结点A到结点B的有向边的含义是过程A调用过程B。对简单的程序来说，完全可以用图的形式来表示调用图。但对大型系统来说，调用图的图形表示会因为太大、太复杂而用处不大。因此有必要用文本形式表示调用图。

附录1提供了本书中使用到的所有过程的调用图的文本表示形式。它不仅表达了“调用”（call）关系，也表达了“被调用”（is called by）关系。这些信息被组织成一张按字母顺序排列的过程列表。为了便于查找，所有的名称在排序时字母不分大小写，并忽略名称中的下划线。因此你可以发现TCP_HLEN紧跟在tcphowmuch之后，列表中包含了所有的过程以及所有在TCP范例代码中用到的内建宏。该表的每一项给出一个过程或宏的名称、在哪个文件中定义、该文件在本书的第几页出现（如果有）、调用该过程或宏的所有文件的名称以及定义该过程或宏的文件中调用到的所有过程的名称。交叉参考表使用了文件而非过程，是因为比较容易找到该文件在书中的位置。在包含了多个过程的文件中，所有的调用关系和被调关系列于每个过程之后。

alreadint in alrwint.c (pg 384)
called in: alrwval.c
calls: sizeof switch

alreadlen in alrwlen.c (pg 383)
called in: sna2b.c snparse.c

alreadoid in alrwoid.c (pg 386)
called in: alrwval.c sna2b.c
calls: memcmp memcpy

alreadval in alrwval.c (pg 389)
called in: sna2b.c
calls: alreadint alreadoid alwriteint alwritelen alwriteoid freemem memcpy switch

Al_SIGNED macro in asn1.h (pg 382)

alwriteint in alrwint.c (pg 384)
called in: alrwval.c
calls: sizeof switch

alwritelen in alrwlen.c (pg 383)
called in: alrwval.c mksnmp.c snb2a.c

alwriteoid in alrwoid.c (pg 386)
called in: alrwval.c snb2a.c
calls: memcmp memcpy

alwriteval in alrwval.c (pg 389)
called in: snb2a.c
calls: alreadint alreadoid alwriteint alwritelen alwriteoid freemem memcpy switch

addarg in addarg.c
called in: shell.c
calls: isbadpid strcpy strlen

arpadd in arpadd.c (pg 38)
called in: arp_in.c
calls: arpalloc memcpy

arpalloc in arpalloc.c (pg 42)
called in: arpadd.c hgarpadd.c
netwrite.c
calls: arpdq

arpdq in arpdq.c (pg 44)
called in: arpalloc.c arptimer.c
calls: deq freebuf freeq icmp

arpfind in arpfind.c (pg 34)
called in: arp_in.c hgarpdel.c
netwrite.c

arp_in in arp_in.c (pg 40)
called in: ni_in.c
calls: arpadd arpfind arpqsend
freebuf hs2net memcmp memcpy
net2hs write

arpinit in arpinit.c (pg 45)
called in: netstart.c
calls: screate

arpprint in arpprint.c
called in: x_arp.c
calls: index sprintf strcpy strlen write

arpqsend in arpqsend.c (pg 39)
called in: arp_in.c
calls: deq freeq netwrite

arpsend in arpsend.c (pg 35)
called in: arptimer.c netwrite.c
calls: getbuf hs2net memcpy memset
write

arptimer in arptimer.c (pg 43)
called in: slowtimer.c
calls: arpdq arpsend disable restore

ascdate in ascdate.c
called in: lfing.c x_date.c x_who.c
calls: isleap sprintf

BLKEQU macro in string.h

BTOP macro in dma.h

BYTE macro in network.h (pg 504)

cksum in cksum.c (pg 57)
called in: dd_xmit.c icmp.c
icmp_in.c igmp.c
igmp_in.c ipfsend.c
ipproc.c ipputp.c
lsa_send.c lsr_xmit.c
ospf_hsend.c ospfcheck.c

create in proc.h
called in: echod.c hginit.c
netstart.c ospf. c ripin.c
rwho.c shell.c x_create.c

DBHASH macro in ospf_db.h

db_init in db_init.c
called in: ospfinit.c
calls: getmem screate sizeof

db_lookup in db_lookup.c
called in: db_update.c lsack_in.c
lsr_in.c lsr_queue.c lsu_in.c

db_new in db_new.c
called in: db_update.c
calls: getbuf memset sizeof

db_nlink in db_nlink.c
called in: db_resync.c
calls: db_update

db_resync in db_resync.c
called in: if_elect.c
calls: db_nlink

db_rlink in db_rlink.c
calls: switch

db_update in db_update.c
called in: db_nlink.c
calls: db_lookup db_new

ddi_exchng in ddi_exchng.c (pg 348)
called in: ospf_ddin.c
calls: dd_xmit deq freebuf headq
lenq lsr_queue nb_mismatch

ddi_exstart in ddi_exstart.c (pg 347)
called in: ospf_ddin.c
calls: dd_queue deq freebuf
lsr_queue net2hl

ddi_full in ddi_full.c (pg 349)
called in: ospf ddin.c
calls: dd_xmit nb_mismatch

dd_queue in dd_queue.c (pg 341)
called in: ddi_exstart.c
nb_mismatch.c nb_reform.c
nb_switch.c
ospf_ddin.c
calls: dd_xmit enq freebuf hl2net
lss_build ospfddtmpl

dd_xmit in dd_xmit.c (pg 344)
called in: dd_queue.c
ddi_exchng.c ddi_full.c
nb_rexmt.c
calls: cksum getbuf headq hl2net
ipsend memcpy memset net2hs
ospfddtmpl

DELAY macro in slu.h

dn_cat macro in domain.h
called in: ip2name.c name2ip.c

dnparse in dnparse.c
called in: tcpbind.c
calls: dot2ip index isdigit

dot2ip in dot2ip.c
called in: dnparse.c initgate.c
name2ip.c netstart.c x_conf.c
x_igmp.c x_route.c x_snmp.c
calls: atoi index

dsdirec macro in disk.h

dssync macro in disk.h

echod in echod.c
calls: close control create open read
resume sizeof write

ECHOMAX macro in icsetdata.c (pg 114)

echop in echod.c
calls: close control create open read
resume sizeof write

efaceinit in initgate.c
called in: inithost.c
calls: dot2ip ethmcast memepy
memset mkpool netmask netnum
rtadd screate setmask sizeof switch

egp in egp.c

ethcntl in ethcntl.c
calls: disable ethstrt kprintf restore
switch

ethdemux in ethdemux.c
called in: ethinter.c
calls: blkcopy blkequ freebuf kprintf
nbgetbuf net2hs ni_in

ethinit in ethinit.c
calls: blkcopy bzero ethstrt getdmem

getidprom hi8 kprintf le_md lowl6
newq panic sizeof sm_ethint

ethint in ethint.c
calls: disable ethinter restore

ethinter in ethinter.c
called in: ethint.c
calls: ethdemux ethwstrt printcsr0

ethmatch in ethmatch.c
calls: memcmp

ethmcast in ethmcast.c (pg 125)
called in: initgate.c
calls: control memcpy net2h1 switch

ethread in ethread.c
calls: deq kprintf signal suspend wait

ethstrt in ethstrt.c
called in: ethcntl.c ethinit.c
calls: hi8 lowl6 panic printcsr0
set_evec sm_ethint

ethwrite in ethwrite.c
calls: blkcopy enq ethwstrt freebuf
hs2net kprintf pdump

ethwstrt in ethwstrt.c
called in: ethinter.c ethwrite.c
othwrite.c
calls: blkcopy blkequ deq disable
freebuf restore

EVENT macro in tcpfsm.h (pg 161)

fcksum in fcksum.c

fd_to_dd macro in io.h

fgetc macro in io.h

findfd macro in io.h

finger in fclient.c (pg 262)
calls: close name2ip open read
sizeof sprintf strlen write

fingerd in fingerd.c
called in: fserver.c
calls: close control index ip2dot
ip2name lfing memset open read
sizeof syslog

firstid macro in q.h

firstkey macro in q.h

fopen macro in name.h

fputc macro in io.h

freestk macro in mem.h

getaddr in getaddr.c
calls: blkcopy rarpsend signal wait

get_bit in ethcntl.c
calls: disable ethstrt kprintf restore
switch

getchar macro in io.h

get_hashbit in ethcntl.c
calls: disable ethstrt kprintf restore
switch

getiaddr in getaddr.c
called in: getiaddr.c getname.c
getnet.c netstart.c
calls: blkcopy rarpsend signal wait

getiname in getname.c
called in: netstart.c
calls: getiaddr ip2name

getinet in getnet.c
calls: getiaddr

getmib in snhash.c (pg 374)
called in: snprint.c snrslv.c

calls: getmem memcmp oidequ sizeof

getname in getname.c
called in: login.c name2ip.c rwhod.c
shell.c sninit.c x_uptime.c
x_who.c
calls: getiaddr ip2name

getnet in getnet.c
calls: getiaddr

GETPHYS macro in dma.h

getsim in inithost.c
calls: efaceinit memset mkpool
ofaceinit read screate sizeof strlen
switch write

getutim in getutim.c
called in: login.c netstart.c rwhod.c
shell.c
calls: close control net2hl net2xt
open panic read signal wait write

gname in name2ip.c
calls: close control dn_cat dot2ip
freemem getmem getname hs2net
index net2hs open read sizeof
sprintf strcpy strlen write

H2IO macro in ee.h

hashinit in snhash.c (pg 374)
called in: sninit.c
calls: getmem memcmp oidequ sizeof

hashoid in snhash.c (pg 374)
calls: getmem memcmp oidequ sizeof

hgadd in hgadd.c (pg 122)
called in: hgjoin.c
calls: hgarpadd signal wait

hgarpadd in hgarpadd.c (pg 124)
called in: hgadd.c
calls: arpalloc disable memcpy restore

hgarpdel in hgarpdel.c (pg 126)
called in: hgleave.c
calls: arpfind disable restore

hginit in hginit.c (pg 136)
called in: netstart.c
calls: create hgjoin resume rtadd screate signal

hgjoin in hgjoin.c (pg 127)
called in: hginit.c ospfifinit.c
x_igmp.c
calls: hgadd hglookup igmp rtadd
signal sleepl0 wait

hgleave in hgleave.c (pg 135)
called in: x_igmp.c
calls: hgarpdel hglookup rtdel signal tmclear wait

hglookup in hglookup.c (pg 121)
called in: hgjoin.c hgleave.c
igmp_in.c
udpsend.c

hgprint in hgprint.c
calls: index ip2dot printone sprintf
strcpy strlen write

hgrand in hgrand.c (pg 131)
called in: igmp_settimers.c

hi8 macro in network.h (pg 504)
called in: ethinit.c ethstrt.c

hl2net macro in network.h (pg 504)
called in: dd_queue.c
dd_xmit.c lss_build.c
netmask.c netnum.c
ripadd.c riprepl.c
rwhod.c tcph2net.c

hs2net macro in network.h (pg 504)
called in: arp_in.c arpsend.c

ethwrite.c ip2name.c
iph2net.c lsr_xmit.c
lss_build.c name2ip.c
rarpsend.c ripadd.c
tcpcksum.c tcph2net.c
udpcksum.c udph2net.c

ibdisp macro in iblock.h

ibtodb macro in iblock.h

icerrok in icerrok.c (pg 111)
called in: icmp.c
calls: isbrc switch

icmp in icmp.c (pg 109)
called in: arpdq.c icmp_in.c
ipftimer.c ipproc.c
ipputp.c ipredirect.c
local_out.c netstart.c
udp_in.c x_ping.c
calls: cksum freebuf getpid icerrok
icsetbuf icsetdata icsetsrc ipsend
memcpy

icmp_in in icmp_in.c (pg 103)
called in: local_out.c
calls: cksum freebuf icmp icredirect
icsetsrc ipsend netmask send
setmask switch

icredirect in icredirect.c (pg 106)
called in: icmp_in.c
calls: freebuf netmask rtadd rtdel
rtfree rtget

icsetbuf in icsetbuf.c (pg 112)
called in: icmp.c
calls: freebuf getbuf kprintf memcpy
switch

icsetdata in icsetdata.c (pg 114)
called in: icmp.c
calls: isodd memcpy switch

icsetsrc in icsetsrc.c (pg 108)
called in: icmp.c icmp_in.c
calls: netmatch

if_elect in if_elect.c (pg 325)
called in: ospf. c ospftimer.c
calls: db_resync if_electl nb_reform signal wait

if_electl in if_electl.c (pg 324)
called in: if_elect.c
calls: net2hl

igmp in igmp.c (pg 129)
called in: hgjoin.c igmp_update.c
calls: cksum getbuf ipsend

igmp_in in igmp_in.c (pg 133)
called in: local_out.c
calls: cksum freebuf hglookup
igmp_settimers signal switch tmclear wait

igmp_settimers in igmp_settimers.c (pg 130)
called in: igmp_in.c
calls: hgrand signal tmset wait

igmp_update in igmp_update.c (pg 132)
calls: igmp pcreate preceive signal wait

IG_TYP macro in igmp.h (pg 120)

IG_VER macro in igmp.h (pg 120)

initgate in initgate.c
called in: netstart.c
calls: dot2ip ethmcast memcpy
memset mkpool netmask netnum
rtadd screate setmask sizeof switch

inithost in inithost.c
called in: netstart.c
calls: efaceinit memset mkpool
ofaceinit read screate sizeof strlen
switch write

int in local_out.c
called in: tcpswitch.c
calls: icmp icmp_in igmp_in
ipdstopts ipnet2h ipreass ospf_in
switch tcp_in udp_in

IO2H macro in ee.h

ip2dot in ip2dot.c
called in: fingerd.c hgprint.c
ip2name.c netstart.c x_conf.c
x_dumper.c x_finger.c x_ifstat.c
x_net.c x_ns.c x_ping.c x_routes.c
x_snmp.c
calls: sprintf strlen

IP2I macro in snmp.h (pg 376)

ip2name in ip2name.c
called in: fingerd.c getname.c
x_conf.c x_routes.c
calls: close control dn_cat freemem
getmem hs2net ip2dot net2hs open
panic read sizeof sprintf strcat
strlen strncat write

IP_CLASSA macro in ip.h (pg 54)

IP_CLASSB macro in ip.h (pg 54)

IP_CLASSC macro in ip.h (pg 54)

IP_CLASSD macro in ip.h (pg 54)

IP_CLASSE macro in ip.h (pg 54)

ipdbc in ipdbc.c (pg 57)
called in: ipproc.c
calls: getbuf ipputp isbrc memcpy
rtfree rtget

ipdoopts in ipdoopts.c (pg 82)

ipdstopts in ipdstopts.c (pg 82)
called in: local_out.c
calls: memcpy

ipfadd in ipfadd.c (pg 93)
called in: ipreass.c
calls: deq enq freebuf freeq

ipfcons in ipfcons.c (pg 96)
called in: ipfjoin.c
calls: deq freebuf freeq getbuf
memcpy

ipfhcopy in ipfhcopy.c (pg 89)
called in: ipfsend.c ipputp.c
calls: memcpy

ipfinit in ipfinit.c (pg 99)
called in: netstart.c
calls: screate

ipfjoin in ipfjoin.c (pg 94)
called in: ipreass.c
calls: deq freebuf freeq ipfcons seeq

ipfsend in ipfsend.c (pg 88)
called in: ipputp.c
calls: cksum getbuf ipfhcopy iph2net
memcpy netwrite

ipftimer in ipftimer.c (pg 98)
called in: slowtimer.c
calls: deq freebuf freeq icmp signal
wait

ipgetp in ipgetp.c (pg 50)
called in: ipproc.c
calls: receive recvclr

iph2net in iph2net.c (pg 60)
called in: ipfsend.c ipproc.c
ipputp.c
calls: hs2net

IP_HLEN macro in ip.h (pg 54)

ip_in in ip_in.c (pg 63)
called in: ni_in.c
calls: enq freebuf send

ipnet2h in ipnet2h.c (pg 61)
called in: ipproc.c local_out.c
calls: net2hs

ipproc in ipproc.c (pg 51)
calls: cksum freebuf getpid icmp ipdbc ipgetp iph2net ipnet2h ipputp ipredirect rtfree rtget signal

ipputp in ipputp.c (pg 86)
called in: ipdbc.c ipproc.c
calls: cksum freebuf icmp ipfhcopy ipfsend iph2net memcpy net2hs netwrite

ipreass in ipreass.c (pg 91)
called in: local_out.c
calls: freebuf ipfadd ipfjoin newq signal wait

ipredirect in ipredirect.c (pg 116)
called in: ipproc.c
calls: icmp netmask rtfree rtget

ipsend in ipsend.c (pg 61)
called in: dd_xmit.c icmp.c icmp_in.c igmp.c lsa_send.c lsr_xmit.c ospf_hsend.c tcpackit.c tcpreset.c tcpsend.c udpsend.c
calls: enq freebuf send

isalnum macro in ctype.h

isalpha macro in ctype.h
called in: x_snmp.c

isascii macro in ctype.h

isbaddev macro in io.h
called in: x_mount.c

isbadpid macro in proc.h
called in: addarg.c rarp_in.c udp_in.c x_stacktrace.c

isbadport macro in ports.h

isbadsem macro in sem.h

isbrc in isbrc.c (pg 59)
called in: icerrok.c ipdbc.c netmatch.c netwrite.c

iscntrl macro in ctype.h

isdigit macro in ctype.h
called in: dnparse.c x_snmp.c

isempty macro in q.h

isleap macro in date.h
called in: ascdate.c

islower macro in ctype.h

isodd macro in kernel.h (pg 505)
called in: icsetdata.c

isprint macro in ctype.h
called in: snprint.c

isprshort macro in ctype.h

ispunct macro in ctype.h

isspace macro in ctype.h

ISUOPT macro in tcpuopt.c (pg 282)

isupper macro in ctype.h

ISVALID macro in dma.h

isxdigit macro in ctype.h

lastkey macro in q.h

level macro in io.h

lexan in lexan.c

called in: shell.c

lfing in lfing.c
called in: fingerd.c x_finger.c
calls: asctime gettime getuser sprintf
strcat strlen write

local_out in local_out.c
called in: netwrite.c
calls: icmp icmp_in igmp_in
ipdstopts ipnet2h ipreass ospf_in
switch tcp_in udp_in

login in login.c
calls: control fprintf getname getuid
getutim mark read strncpy

low16 macro in network.h (pg 504)
called in: ethinit.c ethstrt.c

lsa_add in lsa_add.c
called in: lsr_in.c
calls: memcpy

lsack_in in lsack_in.c
called in: ospf_c
calls: db_lookup

lsa_send in lsa_send.c
called in: lsr_in.c
calls: cksum ipsend memcpy memset

lsa_xmit in lsa_xmit.c

lsr_add in lsr_add.c
called in: lsr_queue.c
calls: net2hl ospflsrtmpl

lsr_check in lsr_check.c
called in: lsu_in.c

lsr_in in lsr_in.c (pg 350)
called in: ospf_c
calls: db_lookup freebuf lsa_add
lsa_send nb_mismatch ospflstmpl sizeof

lsr_queue in lsr_queue.c
called in: ddi_exchng.c
ddi_exstart.c
calls: db_lookup enq freebuf lsr_add lsr_xmit

lsr_xmit in lsr_xmit.c
called in: lsr_queue.c ospftimer.c
calls: cksum getbuf headq hs2net
ipsend memcpy memset

LSS_ABORT macro in lss_build.c (pg352)

lss_build in lss_build.c (pg 352)
called in: dd_queue.c
calls: enq freebuf hl2net hs2net
memcpy nb_clearl ospfddtmpl signal wait

lsu_in in lsu_in.c
called in: ospf_c
calls: db_lookup deq freebuf headq
lsr_check

ltim2ut macro in date.h

major macro in systypes.h

makedev macro in systypes.h

marked macro in mark.h
called in: rwhod.c x_who.c

max macro in kernel.h (pg 505)
called in: tcprwindow.c

min macro in kernel.h (pg 505)
called in: tcppersist.c tcprexmt.c tcpsmss.c
tcpsndlen.c tcpwinit.c tcpxmit.c

minor macro in systypes.h

mkarp in rarpsend.c
calls: disable getbuf getpid hs2net
memcpy panic recvclr recvtim
restore rtadd write

MKEVENT macro in tcpfsm.h (pg 161)

mksnmp in mksnmp.c (pg 408)
called in: snclient.c snmpd.c
calls: alwritelen memcpy sizeof snb2a

MOVC macro in dma.h

MOVL macro in dma.h

MOVSB macro in dma.h

MOVSL macro in dma.h

MOVSW macro in dma.h

name2ip in name2ip.c
called in: fclient.c
x_dumper.c x_finger.c x_ns.c x_ping.c
x_snmp.c
calls: close control dn_cat dot2ip
freemem getmem getname hs2net
index net2hs open read sizeof
sprintf strcpy strlen write

nb_add in nb_add.c (pg 331)
called in: ospf_ddin.c ospf_hin.c
calls: signal wait

N_BADMAG macro in a.out.h

nb_aok in nb_aok.c (pg 336)
called in: nb_reform.c
nb_switch.c ospf_ddin.c

nb_clearl in nb_clearl.c (pg 328)
called in: lss_build.c nb_reform.c nb_switch.c
calls: deq freebuf freeq

nb_makel in nb_makel.c (pg 329)
called in: nb_reform.c nb_switch.c ospf_ddin.c
calls: newq

nb_mismatch in nb_mismatch.c (pg 355)
called in: ddi_exchng.c ddi_full.c lsr_in.c
calls: dd_queue deq freebuf

nb_reform in nb_reform.c (pg 327)
called in: if_elect.c
calls: dd_queue gettime nb_aok
nb_clearl nb_makel

nb_rexmt in nb_rexmt.c (pg 335)
called in: ospftimer.c
calls: dd_xmit

N_BSSADDR macro in a.out.h

nb_switch in nb_switch.c (pg 332)
called in: ospf_hin.c
calls: dd_queue gettime nb_aok
nb_clearl nb_makel sizeof

N_DATADDR macro in a.out.h

net2hl macro in network.h (pg 504)
called in: ddi_exstart.c ethmcast.c getutim.c
if_electl.c lsr_add.c ospf_ddin.c ospf_hin.c
ospfhtmpl.c ripok.c riprecv.c riprepl.c
rwhoind.c tcpnet2h.c tcpsmss.c x_ospf.c
x_routes.c

net2hs macro in network.h (pg 504)
called in: arp_in.c dd_xmit.c
ethdemux.c ip2name.c ipnet2h.c ipputp.c
name2ip.c ospf_hin.c ospf_hsend.c ospfcheck.c
ospfhtmpl.c ospfnet2h.c rarp_in.c riprecv.c
riprepl.c tcpnet2h.c tcpsmss.c udp_in.c
udpnet2h.c x_ospf.c x_rls.c

net2xt macro in date.h
called in: getutim.c

netdump in netdump.c
calls: kprintf pcount

netmask in netmask.c (pg 71)
called in: icmp_in.c icredirect.c initgate.c
ipredirect.c rarp_in.c riprecv.c setmask.c
calls: hl2net netnum

netmatch in netmatch.c (pg 71)
called in: icsetsrc.c rtget.c

calls: isbrc

netnum in netnum.c (pg 70)
called in: initgate.c netmask.c
rarp_in.c ripadd.c
calls: hl2net

netstart in netstart.c
calls: arpinit create dot2ip getiaddr
getiname getutim hginit icmp
initgate inithost ip2dot ipfinit newq
panic recvtim resume rtadd rwho
screate sleep sprintf strlen udpecho
wait write

netwrite in netwrite.c (pg 36)
called in: arpqsend.c ipfsend.c ipputp.c
calls: arpalloc arpfind arpsend
disable enq freebuf isbrc kprintf
local_out memcpy newq pdump
restore write

NIGET macro in netif.h (pg 19)

ni_in in ni_in.c (pg 27)
called in: ethdemux.c
calls: arp_in freebuf ip_in memcmp
rarp_in switch

nonempty macro in q.h

N_PAGSIZ macro in a.out.h

N_SEGSIZ macro in a.out.h

N_STROFF macro in a.out.h

N_SYMOFF macro in a.out.h

N_TXTADDR macro in a.out.h

N_TXTOFF macro in a.out.h

ofaceinit in initgate.c
called in: inithost.c
calls: dot2ip ethmcast memcpy
memset mkpool netmask netnum
rtadd screate setmask sizeof switch

oideqn macro in snmp.h (pg 376)
called in: snhash.c snrslv.c

ospf in ospf.c (pg 337)
calls: create freebuf if_elect lsack_in
lsr_in lsu_in ospf_ddin ospf_hin
ospfcheck ospfinit ospfnet2h panic
pcreate preceive resume switch

ospfcheck in ospfcheck.c (pg 354)
called in: ospf. c
calls: cksum memcmp memset net2hs

ospf_ddin in ospf_ddin.c (pg 345)
called in: ospf. c
calls: dd_queue ddi_exchng
ddi_exstart ddi_full nb_add nb_aok
nb_makel net2hl switch

ospfddtmpl in ospfddtmpl.c (pg 343)
called in: dd_queue.c dd_xmit.c lss_build.c
calls: getbuf memcpy

ospfhello in ospfhello.c (pg 322)
calls: ospf_hsend sleepl0 switch

ospf_hin in ospf_hin.c (pg 329)
called in: ospf. c
calls: nb_add nb_switch net2hl net2hs

ospf_hsend in ospf_hsend.c (pg 319)
called in: ospfhello.c
calls: cksum ipsend memcpy memset
net2hs ospfhtmpl signal sizeof wait

ospfhtmpl in ospfhtmpl.c (pg 320)
called in: ospf_hsend.c
calls: getbuf memcpy net2hl net2hs

ospfifinit in ospfifinit.c
called in: ospfinit.c

calls: bgjoin memset screate signal

ospf_in in ospf_in.c (pg 336)
called in: local_out.c
calls: freebuf pcount psend

ospfinit in ospfinit.c
called in: ospf. c
calls: db_init mkpool ospfifinit sizeof

ospflsrtmpl in ospflsrtmpl.c
called in: lsr_add.c
calls: getbuf memcpy

ospflstmpl in ospflstmpl.c
called in: lsr_in.c
calls: getbuf memcpy

ospfnet2h in ospfnet2h.c (pg 354)
called in: ospf. c
calls: net2hs

ospftimer in ospftimer.c (pg 334)
called in: slowtimer.c
calls: deq freebuf headq if_elect
lsr_xmit nb_rexmt signal switch wait

othinit in othinit.c
calls: control kprintf memcpy panic

othwrite in othwrite.c
calls: enq ethwstrt freebuf kprintf memcpy pdump

pdelete in ports.h
called in: tcbdealloc.c tcplq.c

pidle in lfing.c
calls: ascdate gettime getuser sprintf
strcat strlen write

preset in ports.h

printcsr0 in ethinit.c
called in: ethinter.c ethstrt.c
calls: blkcopy bzero ethstrt getdmem
getidprom hi8 kprintf le_md lowl6
newq panic sizeof sm_ethint

printone in arpprint.c
called in: hgprint.c
calls: index sprintf strcpy strlen write

PTOB macro in dma.h

putchar macro in io.h

rarp_in in rarp_in.c
called in: ni_in.c
calls: disable freebuf isbadpid
memcmp memcpy net2hs netmask
netnum restore send setmask

rarpsend in rarpsend.c
called in: getaddr.c getiaddr.c
calls: disable getbuf getpid hs2net
memcpy panic recvclr recvtim
restore rtadd write

resolve in name2ip.c
calls: close control dn_cat dot2ip
freemem getmem getname hs2net
index net2hs open read sizeof
sprintf strcpy strlen write

rip in ripin.c (pg 294)
calls: create open panic read resume
ripcheck riprecv riprepl screate
sizeof switch

ripadd in ripadd.c (pg 302)
called in: ripsend.c
calls: hl2net hs2net memset netnum
ripmetric ripstart sizeof

ripcheck in ripcheck.c (pg 295)
called in: ripin.c
calls: sizeof switch

ripifset in ripifset.c (pg 301)
called in: ripsend.c

calls: rtfree rtget

ripmetric in ripmetric.c (pg 304)
called in: ripadd.c

ripok in ripok.c (pg 298)
called in: riprecv.c
calls: net2hl

ripout in ripout.c (pg 306)
calls: getpid recvtim ripsend signal
sleepl0 wait

riprecv in riprecv.c (pg 296)
called in: ripin.c
calls: net2hl net2hs netmask ripok
rtadd rtfree rtget signal sizeof wait

riprepl in riprepl.c (pg 299)
called in: ripin.c
calls: getbuf hl2net memcpy net2hl
net2hs ripsend rtfree rtget sizeof udpsend

ripsend in ripsend.c (pg 300)
called in: ripout.c riprepl.c
calls: ripadd fipifset signal udpsend wait

ripstart in ripstart.c (pg 305)
called in: ripadd.c
calls: getbuf

rlok in db_rlink.c
calls: switch

roundew macro in mem.h

roundmb macro in mem.h

rtadd in rtadd.c (pg 77)
called in: hginit.c hgjoin.c icredirect.c initgate.c
netstart.c rarpsend.c riprecv.c
setmask.c x_route.c
calls: rthash rtinit rtnew send signal wait

rtdel in rtdel.c (pg 80)
called in: hgleave.c icredirect.c setmask.c
srt_set.e x_route.c
calls: rthash signal wait

rtdump in rtdump.e
calls: kprintf rtinit signal wait

rtfree in rtfree.c (pg 81)
called in: icredirect.c ipdbc.c
ipproc.c ipredirect.c ripifset.c riprecv.c
riprepl.c tcpbind.c tcpcon.c tcpwinit.c
udpsend.c x_route.c
calls: signal wait

rtget in rtget.c (pg 73)
called in: icredirect.c ipdbc.c
ipproc.c ipredirect.c ripifset.c riprecv.c
riprepl.c tcpbind.c tcpcon.c tcpwinit.c
udpsend.c x_route.c
calls: netmatch rthash rtinit signal wait

rthash in rthash.c (pg 72)
called in: rtadd.c rtdel.c rtget.c

rtinit in rtinit.c (pg 74)
called in: rtadd.c rtdump.c rtget.c
calls: mkpool screate sizeof

rtnew in rtnew.c (pg 79)
called in: rtadd.c
calls: getbuf

rttimer in rttimer.c (pg 75)
called in: slowtimer.c
calls: send signal wait

rwho in rwho.c
called in: netstart.c
calls: create resume

rwhod in rwhod.c
calls: disable getbuf getname getutim
hl2net index marked memcpy
restore sizeof sleep strcpy strncpy udpsend

rwhoind in rwhoind.c
calls: control disable gettime net2hl open panic read restore sizeof strncmp strncpy

sae_findnext in sae_findn.c (pg 425)
called in: sae_getf.c sae_getn.c
calls: memcmp

sae_get in sae_get.c (pg 421)
called in: sae_getf.c sae_getn.c
calls: sae_match switch

sae_getf in sae_getf.c (pg 423)
calls: memcpy sae_findnext sae_get sip2ocpy

sae_getn in sae_getn.c (pg 424)
calls: sae_findnext sae_get sae_match sip2ocpy

sae_match in sae_match.c (pg 420)
called in: sae_get.c sae_getn.c
calls: soipequ va_arg va_end va_start

sae_set in sae_set.c (pg 426)

SECYEAR macro in clock.h

SEQCMP macro in tcp.h (pg 158)

set_bit in ethcntl.c
calls: disable ethstrt kprintf restore switch

seterr macro in snrslv.c (pg 401)

setmask in setmask.c (pg 107)
called in: icmp_in.c initgate.c rarp_in.c
calls: netmask rtadd rtdel

SHA macro in arp.h (pg 31)

shell in shell.c
calls: addarg close control create fprintf getname getpid getutim index lexan open read receive recvclr resume setdev setnok sizeof strcmp strlen

sir_get in sif_get.c (pg 436)
called in: sif_getf.c sif_getn.c
calls: getmem lenq memcpy sif_match strlen switch

sif_getf in sif_getf.c (pg 440)
calls: memcpy sif_get

sif_getn in sif_getn.c (pg 441)
calls: sif_get sif_match

sif_match in sif_match.c (pg 435)
called in: sif_get.c sif_getn.c sif_set.c
calls: va_arg va_end va_start

sif_set in sif_set.c (pg 442)
calls: sif_match

sip2ocpy in snoip.c (pg 466)
called in: sae_getf.c sae_getn.c snm_getf.c snm_getn.c srt_getf.c srt_getn.c stc__getf.c stc_getn.c sud_getf.c sud_getn.c

slowtimer in slowtimer.c (pg 64)
calls: arptimer gettime ipftimer ospftimer rttimer signal sleep

sna2b in sna2b.c (pg 400)
called in: snclient.c snerr.c snmpd.c
calls: alreadlen alreadoid alreadval

snb2a in snb2a.c (pg 411)
called in: mksnmp.c
calls: alwritelen alwriteoid alwriteval getmem memcpy

snclient in snclient.c (pg 413)
called in: x_snmp.c
calls: close control memcmp memcpy mksnmp open read sna2b snparse write

snerr in snerr.c
called in: x_snmp.c

calls: sna2b snmpprint_objid sprintf
strlen write

snfreebl in snfreebl.c (pg 412)
called in: snmpd.c snparse.c x_snmp.c
calls: freemem sizeof

sninit in sninit.c (pg 415)
called in: snmpd.c x_snmp.c
calls: getname hashinit strcpy

snleaf in snleaf.c (pg 404)
calls: getmem memcpy strlen switch

snmcmp in snm_findn.c (pg 432)
calls: memcmp

snm_findnext in snm_findn.c (pg 432)
called in: snm_getf.c snm_getn.c
calls: memcmp

snm_get in snm_get.c (pg 428)
called in: snm_getf.c snm_getn.c
calls: getmem memcpy snm_match switch

snm_getf in snm_getf.c (pg 430)
calls: memcpy sip2ocpy snm_findnext snm_get

snm_getn in snm__getn.c (pg 431)
calls: sip2ocpy snm_findnext snm_get snm_match

snm_match in snm_match.c (pg 427)
called in: snm_get.c snm_getn.c snm_set.c
calls: soipequ va_arg va_end va_start

snmpd in snmpd.c (pg 393)
calls: getmem mksnmp open read
sizeof sna2b snfreebl sninit snparse
snrslv switch write

snmpprint in snprint.c
called in: x_snmp.c
calls: getmib isprint sprintf strcat
strcpy strlen switch write

snmpprint_objid in snprint.c
called in: snerr.c
calls: getmib isprint sprintf strcat
strcpy strlen switch write

snmpprint_objname in snprint.c
calls: getmib isprint sprintf strcat
strcpy strlen switch write

snmpprint_val in snprint.c
calls: getmib isprint sprintf strcat
strcpy strlen switch write

snm_set in snm_set.c (pg 434)
calls: memcpy snm_match switch

snparse in snparse.c (pg 396)
called in: snclient.c snmpd.c
calls: alreadlen getmem memcpy
sizeof snfreebl strncmp

snrslv in snrslv.c (pg 401)
called in: snmpd.c
calls: getmib oidequ switch

sntable in sntable.c (pg 407)
calls: switch ti_get ti_getf ti_getn ti_set

sntcpcmp in stc_findn.c (pg 457)

so2ipcpy in snoip.c (pg 466)
called in: stc_match.c

soipequ in snoip.c (pg 466)
called in: sae_match.c snm_match.c
srt_match.c

SOP2STR macro in snmp.h (pg 376)

SPA macro in arp.h (pg 31)

srt_findnext in srt_findn.c (pg 449)
called in: srt_getf.c srt_getn.c

srt_get in srt_get.c (pg 444)
called in: srt_getf.c srt_getn.c

calls: memcpy srt_match switch

srt_getf in srt_getf.c (pg 446)
calls: memcpy sip2ocpy srt_findnext srt_get

srt_getn in srt_getn.c (pg 448)
calls: sip2ocpy srt_findnext srt_get srt_match

srt_match in srt_match.c (pg 443)
called in: srt_get.c srt_getn.c srt_set.c
calls: soipequ va_arg va_end va_start

srt_set in srt_set.c (pg 450)
calls: rtdel srt_match switch

stc_findnext in stc_findn.c (pg 457)
called in: stc_getf.c stc_getn.c

stc_get in stc_get.c (pg 453)
called in: stc_getf.c stc_getn.c
calls: stc_match switch

stc_getf in stc getf.c (pg 455)
calls: memcpy sip2ocpy stc_findnext stc_get

stc_getn in stc_getn.c (pg 456)
calls: sip2ocpy stc_findnext stc_get stc_match

stc_match in stc_match.c (pg 452)
called in: stc_get.c stc_getn.c stc_set.c
calls: so2ipcpy va_arg va_end va_start

stc_set in stc_set.c (pg 458)
calls: close stc_match switch

strequ macro in snmp.h (pg 376)
called in: x_snmp.c

sud_findnext in sud_findn.c (pg 464)
called in: sud_getf.c sud_getn.c

sud_get in sud_get.c (pg 461)
called in: sud_getf.c sud_getn.c
calls: sud_match switch

sud_getf in sud_getf.c (pg 462)
calls: memcpy sip2ocpy sud_findnext sud_get

sud_getn in sud_getn.c (pg 463)
calls: sip2ocpy sud_findnext sud_get sud_match

sud_match in sud_match.c (pg 460)
called in: sud_get.c sud_getn.c
calls: va_arg va_end va_start

sud_set in sud_set.c (pg 465)

SVINT macro in snmp.h (pg 376)

SVIPADDR macro in snmp.h (pg 376)

SVOID macro in snmp.h (pg 376)

SVOIDLEN macro in snmp.h (pg 376)

SVSTR macro in snmp.h (pg 376)

SVSTRLEN macro in snmp.h (pg 376)

SVTYPE macro in snmp.h (pg 376)

TCB macro in tcpfsm.h (pg 161)

tcballoc in tcballoc.c (pg 163)
called in: tcplisten.c tcpmopen.c
calls: screate signal wait

tcbdealloc in tcbdealloc.c (pg 164)
called in: tcpclose.c tcpclosing.c tcpcon.c
tcpout.c tcpsynrcvd.c tcptimewait.c
calls: freemem freeq pdelete sdelete
signal switch tcpkilltimers

tcpabort in tcpabort.c (pg 191)
called in: tcpclosewait.c tcpestablished.c
tcpfin 1.c tcpfin2.c tcplastack.c tcprexmt.c
tcpsynrcvd.c
calls: tcpkilltimers tcpwakeup

tcpacked in tcpacked.c (pg 243)
called in: tcpclosewait.c
tcpclosing.c tcpestablished.c

tcpfin1.c tcpfin2.c tcplastack.c
tcpsynrcvd.c tcpsynsent.c tcptimewait.c
calls: disable restore scount signal
tcpackit tcpostate tcpreset tcprtt

tcpackit in tcpackit.c (pg 245)
called in: tcpacked.c tcpinp.c
calls: getbuf ipsend tcpcksum tcph2net tcprwindow

tcpbind in tcpbind.c (pg 267)
called in: tcpmopen.c
calls: dnparse rtfree rtget tcpnxtp

tcpcksum in tcpcksum.c (pg 168)
called in: tcpackit.c tcpinp.c tcpreset.c tcpsend.c
calls: hs2net

tcpclose in tcpclose.c (pg 276)
calls: signal switch tcbdealloc tcpkick wait

tcpclosed in tcpclosed.c (pg 175)
calls: tcpreset

tcpclosewait in tcpclosewait.c (pg 181)
calls: tcpabort tcpacked tcpreset tcpswindow

tcpclosing in tcpclosing.c (pg 178)
calls: signal tcbdealloc tcpacked tcpreset tcpwait

tcpcntl in tcpcntl.c (pg 278)
calls: preceive signal switch tcplq
tcpstat tcpuopt tcpwr wait

tcpcon in tcpcon.c (pg 269)
called in: tcpmopen.c
calls: rtfree rtget signal tcbdealloc tcpkick wait

tcpdata in tcpdata.c (pg 284)
called in: tcpestablished.c tcpfinl.c tcpfin2.c
tcplisten.c tcpsynrcvd.c tcpsynsent.c
tcptimewait.c
calls: tcpdodat tcpkick

tcpdemux in tcpdemux.c (pg 170)
called in: tcpinp.c
calls: signal wait

tcpdodat in tcpdodat.c (pg 186)
called in: tcpdata.c
calls: tcpwakeup tfcoalesce tfinsert

tcpestablished in tcpestablished.c (pg 183)
calls: tcpabort tcpacked tcpdata
tcpreset tcpswindow

tcpfinl in tcpfin1.c (pg 180)
calls: signal tcpabort tcpacked
tcpdata tcpreset tcpswindow tcpwait

tcpfin2 in tcpfin2.c (pg 179)
calls: tcpabort tcpacked tcpdata tcpreset tcpwait

tcpgetc in tcpgetc.c (pg 273)
calls: tcpread

tcpgetdata in tcpgetdata.c (pg 254)
called in: tcpread.c
calls: tcpkick tcprwindow

tcpgetspace in tcpgetspace.c (pg 213)
called in: tcpwr.c
calls: signal tcpwakeup wait

tcph2net in tcph2net.c (pg 213)
called in: tcpackit.c tcpreset.c tcpsend.c
calls: hl2net hs2net

TCP_HLEN macro in tcp.h (pg 158)

tcphowmuch in tcphowmuch.c (pg 211)
called in: tcpxmit.c

tcpidle in tcpidle.c (pg 203)
calls: tcpxmit

tcp_in in tcp_in.c (pg 165)
called in: local_out.c
calls: freebuf pcount psend

tcpinit in tcpinit.c (pg 283)

calls: mark screate unmarked

tcpinp in tcpinp.c (pg 166)
calls: freebuf pcreate preceive signal
tcpackit tcpcksum tcpdemux
tcpnet2h tcpok tcpopts tcpreset

tcpiss in tcpiss.c (pg 216)
called in: tcpsync.c

tcpkick in tcpkick.c (pg 227)
called in: tcpclose.c tcpcon.c tcpdata.c
tcpgetdata.c tcpostate.c tcpswindow.c tcpwr.c
calls: pcount psend tmleft tmset

tcpkilltimers in tcpkilltimers.c (pg 223)
called in: tcbdealloc.c tcpabort.c tcpsynsent.c
tcpwait.c
calls: tmclear

tcplastack in tcplastack.c (pg 182)
calls: signal tcpabort tcpacked tcpreset

tcplisten in tcplisten.c (pg 196)
calls: signal tcballoc tcpdata tcpreset
tcpsync tcpwinit

tcplq in tcplq.c (pg 279)
called in: tcpcntl.c
calls: pcreate pdelete

tcpmcntl in tcpmcntl.c (pg 271)
calls: switch

tcpmopen in tcpmopen.c (pg 265)
calls: sdelete tcballoc tcpbind tcpcon
tcpserver tcpsync

tcpnet2h in tcpnet2h.c (pg 168)
called in: tcpinp.c
calls: net2hl net2hs

tcpnxtp in tcpnxtp.c (pg 268)
called in: tcpbind.c
calls: panic signal wait

tcpok in tcpok.c (pg 171)
called in: tcpinp.c

tcpopts in tcpopts.c (pg 236)
called in: tcpinp.c
calls: memcpy switch tcpsmss

tcpostate in tcpostate.c (pg 246)
called in: tcpacked.c
calls: tcpkick

tcpout in tcpout.c (pg 202)
calls: pcreate preceive signal
tcbdealloc wait

tcppersist in tcppersist.c (pg 204)
calls: min tcpsend tmset

tcpputc in tcpputc.c (pg 274)
calls: tcpwr

tcpread in tcpread.c (pg 272)
called in: tcpgetc.c
calls: signal tcpgetdata tcpwakeup wait

tcpreset in tcpreset.c (pg 211)
called in: tcpacked.c
tcpclosed.c tcpclosewait.c tcpclosing.c
tcpestablished.c tcpfinl.c tcpfin2.c tcpinp.c
tcplastack.c tcplisten.c tcpsynrcvd.c
tcpsynsent.c tcptimewait.c
calls: getbuf ipsend tcpcksum tcph2net

tcprexmt in tcprexmt.c (pg 230)
called in: tcpxmit.c
calls: min tcpabort tcpsend tmset

tcprmss in tcprmss.c (pg 238)
called in: tcpsend.c
calls: sizeof

tcprtt in tcprtt.c (pg 240)
called in: tcpacked.c
calls: tmclear

tcprwindow in tcprwindow.c (pg 232)
called in: tcpackit.c
tcpgetdata.c tcpsend.c
calls: max

tcpsend in tcpsend.c (pg 207)
called in: tcppersist.c
tcprexmt.c tcpxmit.c
calls: getbuf ipsend tcpcksum
tcph2net tcprmss tcprwindow tcpsndlen tcpwait

tcpserver in tcpserver.c (pg 266)
called in: tcpmopen.c
calls: pcreate sdelete signal

tcpsmss in tcpsmss.c (pg 235)
called in: tcpopts.c
calls: min net2hl net2hs sizeof switch

tcpsndlen in tcpsndlen.c (pg 210)
called in: tcpsend.c
calls: min

tcpstat in tcpstat.c (pg 280)
called in: tcpcntl.c
calls: switch

tcpswindow in tcpswindow.c (pg 233)
called in: tcpclosewait.c
tcpestablished.c tcpfin 1.c
calls: tcpkick tmclear

tcpsync in tcpsync.c (pg 191)
called in: tcplisten.c tcpmopen.c
calls: getmem screate tcpiss

tcpsynrcvd in tcpsynrcvd.c (pg 194)
calls: pcount psend signal tcbdealloc
tcpabort tcpacked tcpdata tcpreset wait

tcpsynsent in tcpsynsent.c (pg 193)
calls: signal tcpacked tcpdata
tcpkilltimers tcpreset

tcptimer in tcptimer.c (pg 220)
calls: freemem getpid pcount psend
screate signal sizeof sleepl0 suspend wait

tcptimewait in tcptimewait.c (pg 177)
calls: tcbdealloc tcpacked tcpdata tcpreset tcpwait

tcpuopt in tcpuopt.c (pg 282)
called in: tcpcntl.c

tcpwait in tcpwait.c (pg 176)
called in: tcpclosing.c tcpfinl.c tcpfin2.c
tcpsend.c tcptimewait.c
calls: tcpkilltimers tmset

tcpwakeup in tcpwakeup.c (pg 215)
called in: tcpabort.c
tcpdodat.c tcpgetspace.c tcpread.c tcpwr.c
calls: disable restore scount signal

tcpwinit in tcpwinit.c (pg 197)
called in: tcplisten.c
calls: min rtfree rtget

tcpwr in tcpwr.c (pg 275)
called in: tcpcntl.c tcpputc.c tcpwrite.c
calls: signal tcpgetspace tcpkick tcpwakeup

tcpwrite in tcpwrite.c (pg 274)
calls: tcpwr

tcpxmit in tcpxmit.c (pg 205)
called in: tcpidle.c
calls: min tcphowmuch tcprexmt
tcpsend tmclear tmleft tmset

tfcoalesce in tfcoalesce.c (pg 189)
called in: tcpdodat.c
calls: deq enq freemem freeq sizeof

tfinsert in tfinsert.c (pg 188)
called in: tcpdodat.c
calls: enq freemem getmem newq sizeof

THA macro in arp.h (pg 31)

tmclear in tmclear.c (pg 222)
called in: hgleave.c igmp_in.c tcpkilltimers.c
tcprtt.c tcpswindow.c tcpxmit.c tmset.c
calls: freemem signal sizeof wait

tmleft in tmleft.c (pg 224)
called in: tcpkick.c tcpxmit.c
calls: signal wait

tmset in tmset.c (pg 225)
called in: igmp_settimers.c tcpkick.c tcppersist.c
tcprexmt.c tcpwait.c tcpxmit.c
calls: getmem resume signal sizeof tmclear wait

toascii macro in ctype.h

tolower macro in ctype.h

toupper macro in ctypè.h

TPA macro in arp.h (pg 31)

tqdump in tqdump.c
calls: kprintf sprintf strlen write

tqwrite in tqdump.c
called in: x_timerq.c
calls: kprintf sprintf strlen write

truncew macro in mem.h

truncmb macro in mem.h

udpcksum in udpcksum.c (pg 148)
called in: udp_in.c udpsend.c
calls: hs2net

udpecho in udpecho.c
called in: netstart.c
calls: open printf read write

udph2net in udph2net.c (pg 146)
called in: udpsend.c
calls: hs2net

udp_in in udp_in.c (pg 146)
called in: local_out.c
calls: freebuf icmp isbadpid net2hs
pcount psend send signal udpcksum
udpnet2h wait

udpnet2h in udpnet2h.c (pg 145)
called in: udp_in.c
calls: net2hs

udpnxtp in udpnxtp.c (pg 149)

udpsend in udpsend.c (pg 150)
called in: riprepl.c ripsend.c rwhed.c
calls: freebuf hglookup ipsend rtfree
rtget signal udpcksum udph2net wait

UEPAGE macro in ue.h

unmarked macro in mark.h
called in: tcpinit.c

upalloc in upalloc.c (pg 144)
calls: pcreate signal wait

ut2ltim macro in date.h
called in: x_who.c

va_arg macro in stdarg.h
called in: sae_match.c
sif_match.c snm_match.c
srt_match.c stc_match.c sud_match.c

va_end macro in stdarg.h
called in: sae_match.c
sif_match.c snm_match.c
srt_match.c stc_match.c sud_match.c

va_start macro in stdarg.h
called in: sae_match.c
sif_match.c snm_match.c
srt_match.c stc_match.c sud match.c

xt2net macro in date.h

ZSTimeConst macro in zsreg.h

附录2　程序代码中使用到的C数据结构交叉参考表

引言

本附录包含了程序代码中用到的C数据结构交叉参考。每个数据结构名后是它所被引用的文件名称。此处并未列出本书中所有的代码，而有意略去了操作系统中的那部分。如果本书中的任何一个文件在交叉参考表中出现，列表中则给出了这个文件所在的页码。若没有，读者则需要参考一下源代码。

arp def. in arp.h (pg 31)

arpentry def. in arp.h (pg 31)

bert_field def. in dma.h

bootparam def. in sunromvec.h

boottab def. in sunromvec.h

bpool def. in bufpool.h

bre
refs: ethdemux.c ethinit.c ethinter.c

cbl def. in ee.h

cmdent def. in cmd.h

comsoft def. in com.h

core11
refs: x_snap.c

cx_field def. in dma.h

datinfo def. in date.h
refs: ascdate.c

devinfo def. in sunromvec.h

devsw def. in conf. h (pg 507)
and in saio.h
refs: com.h

dgblk def. in dgram.h (pg 142)

dinode
refs: saio.h

dir def. in dir.h

dirent def. in x_rls.c

dn_mesg def. in domain.h

dn_qsuf def. in domain.h

dreq def. in disk.h

dsblk def. in disk.h

dtc def. in dtc.h
refs: disk.h

eh def. in ether.h (pg 31)

enable_field def. in dma.h

ep def. in ether.h (pg 31)
refs: arpdq.c arpqsend.c arpsend.c dd_queue.c
dd_xmit.c ddi_exchng.c ddi_exstart.c
ethdemux.c

etblk def. in ether.h (pg 21)
refs: ethcntl.c ethdemux.c

etdev def. in ether.h (pg 21)

exec def. in a.out.h

fdes def. in dir.h

flblk def. in lfile.h

fpacket def. in fserver.h

fphdr def. in fserver.h

freeblk def. in dir.h

fs
refs: saio.h

globram
refs: sunromvec.h

hg def. in igmp.h (pg 120)
refs: hgadd.c hginit.c hgjoin.c hgleave.c, hglookup.c hgprint.c

hginfo def. in igmp.h (pg 120)
refs: hginit.c

hwa def. in netif.h (pg 19)

iblk def. in iblock.h

icmp def. in icmp.h (pg 101)
refs: icerrok.c icmp.c

id_field def. in dma.h

idprom def. in idprom.h
refs: ethinit.c

idt def. in i386.h

igmp def. in igmp.h (pg 120)
refs: igmp.c

interrupt_field def. in dma.h

intersil7170 def. in clock.h

intmap def. in io.h

iob def. in saio.h

ip def. in ip.h (pg 54)
refs: arpdq.c dd_queue.c dd_xmit.c ddi_exchng.c ddi_exstart.c, ddi_full.c ethwrite.c icerrok.c icmp.c icmp_in.c icredirect.c igmp.c igmp_in.c

ipfq def. in ipreass.h (pg 90)
refs: ipftimer.c ipreass.c

iscp def. in ee.h

keybuf
refs: sunromvec.h

le_device def. in lereg.h
refs: ethinit.c ethstrt.c ethwstrt.c

le_drp def. in lereg.h

le_init_block def. in lereg.h
refs: ethcntl.c ethinit.c

le_md def. in lereg.h
refs: ethdemux.c ethinit.c ethinter.c ethwstrt.c

led_field def. in dma.h

mblock def. in mem.h

mc68k_vector def. in dma.h

mib_info def. in mib.h (pg 363)

nam def. in name.h

nament def. in name.h

netif def. in netif.h (pg 19)
refs: arp.h arpqsend.c arpsend.c ethinter.c getaddr.c getiaddr.c getname.c getnet.c hgarpadd.c hgarpdel.c ipputp.c local_out.c

netinfo def. in net.h

netmon def. in netmon.h

netq
refs: netdump.c

nlist def. in a.out.h

oid def. in snmp.h (pg 376)
refs: mib.h sif_get.c sninit.c snleaf.c snmib.c

ospf def. in ospf_pkt.h (pg 311)
refs: dd_queue.c dd_xmit.c ddi_exchng.c ddi_exstart.c ddi_full.c ethwrite.c lsa_add.c lsa_send.c lsack_in.c lsr_add.c lsr_in.c lsr_queue.c lsr_xmit.c lss_build.c lsu_in.c nb_switch.c netwrite.c ospf. c ospf_ddin.c ospf_hin.c ospf_hsend.c

ospf_ar def. in ospf. h (pg 314)
refs: db_nlink.c db_resync.c db_rlink.c lss_build.c

ospf_db def. in ospf_db.h
refs: db_lookup.c db_new.c db_nlink.c db_resync.c db_rlink.c db_update.c lsr_in.c lsr_queue.c lss_build.c ospf. h

ospf_dd def. in ospf_pkt.h (pg 311)
refs: dd_queue.c dd_xmit.c ddi_exchng.c ddi_exstart.c ddi_full.c lsr_queue.c lss_build.c ospf_ddin.c

ospf_hello def. in ospf_pkt.h (pg 311)
refs: nb_add.c nb_switch.c ospf_hin.c ospf hsend.c

ospf_if def. in ospf_if.h (pg 313)
refs: if_elect.c lsack_in.c lsr_in.c lsu_in.c ospf. c ospf. h ospf_ddin.c ospf_hin.c ospf_hsend.c

ospf_lsa def. in ospf_ls.h (pg 339)
refs: db_nlink.c db_resync.c db_rlink.c, ospf_db.h

ospf_lsr def. in ospf_ls.h (pg 339)
refs: lsr_add.c lsr_in.c

ospf_lss def. in ospf_ls.h (pg 339)
refs: lsr_queue.c lss_build.c

ospf_lsu def. in ospf_ls.h (pg 339)
refs: lsa_add.c lsack_in.c lsr_in.c lsu_in.c

ospf_na def. in ospf_ls.h (pg 339)
refs: db_nlink.c db_resync.c

ospf_nb def. in ospf. h (pg 314)
refs: db_nlink.c if_elect.c if_electl.c lsack_in.c lsr_in.c lsu_in.c nb_add.c nb_reform.c

ospf_ra def. in ospf_ls.h (pg 339)
refs: db_rlink.c

ospf_rl def. in ospf_ls.h (pg 339)
refs: db_rlink.c

otblk def. in ether.h (pg 21)
refs: ethdemux.c

outmp def. in unix-rwho.h

parity_field def. in dma.h

pentry def. in proc.h
refs: addarg.c

pg_field def. in dma.h

pt def. in ports.h

ptnode def. in ports.h

qent def. in q.h

rbd def. in ee.h

relocation_info def. in a.out.h

req_desc def. in snmp.h (pg 376)

rfblk def. in rfile.h

rfd def. in ee.h

rfinfo def. in ffile.h

rip def. in rip.h (pg 292)

riprt def. in rip.h (pg 292)

route def. in route.h (pg 68)
refs: icredireet.c ipdbc.c ipproc.c ipredirect.c, ripifset.c riprecv.c riprepl.c ripsend.c

rq def. in rip.h (pg 292)

rtinfo def. in route.h (pg 68)

rw_who def. in rwho.h

rwent def. in rwho.h

rwhopac def. in rwho.h

rwinfo def. in rwho.h
refs: rwho.c

scb def. in ee.h

scp def. in ee.h

sd def. in i386.h

segtr def. in i386.h

sema_stat def. in sem.h

sentry def. in sem.h

shvars def. in shell.h
refs: shell.c

sm_field def. in dma.h

snbentry def. in snmp.h (pg 376)
refs: mksnmp.c sna2b.c snb2a.c snerr.c

snhnode def. in snhash.h (pg 366)
refs: snhash.c

snval def. in snmp.h (pg 376)

stat
refs: saio.h

station def. in netmon.h

sunromvec def. in sunromvec.h

tab_info def. in mib.h (pg 363)

tbd def. in ee.h

tcb def. in tcb.h (pg 154)

tchars def. in tty.h

tcp def. in tcp.h (pg 158)

tcpfrag def. in tcb.h (pg 154)

tcpstat def. in tcpstat.h (pg 280)
refs: fingerd.c

tosent def. in ospf_ls.h (pg 339)

tqent def. in tcptimer.h (pg 219)
refs: tcptimer.c

tty def. in tty.h
refs: lfing.c

udp def. in udp.h (pg 141)
refs: ripstart.c

upq def. in udp.h (pg 141)

uqe
refs: tcpsndlen.c

urb def. in ue.h

user def. in user.h
refs: lfing.c

utdev def. in ether.h (pg 21)

vector def. in io.h

whod def. in unix-rwho.h

whoent def. in unix-rwho.h

xbdcb
refs: disk.h dtc.h

xgram def. in dgram.h (pg 142)

zscc_device def. in zsreg.h
refs: sunromvec.h

附录3　程序代码中使用到的Xinu函数和常量

引言

本书中所有程序代码使用的常量、过程以及函数由Xinu操作系统提供。许多函数对应于Xinu的系统调用（system call），而其他一些对应于库函数。虽然在不知道这些Xinu过程操作的内部细节的情况下也有可能理解TCP/IP协议软件，但若要了解每个函数提供的服务，就必须首先了解TCP/IP是如何操作的。

本章简要地描述了本书中出现的所有过程和函数，解释它们的目的以及它们使用的参数。另外，我们还列出了文件kernel.h、conf.h和network.h，在许多过程中都要包括这些文件。

从某种程度上看，Xinu系统调用和库程序通常都是“函数”，因为它们总是会返回一个值。然而，很少系统调用是数学计算意义上的函数，因为几乎所有的系统调用都要完成一些不是数学计算的任务。通常出现差错将导致返回一个SYSERR（在某些情况下是具体的差错代码），操作时未出现差错的过程返回OK。

按字母顺序排列的表

函数与过程表（按字母顺序排列）在以下按字母排序的列表中给出了本书中未提供的，但在程序代码中用到的所有过程和函数的名称和参数。由于这些实用程序如何操作的具体细节并不重要，因而没有给出库函数和Xinu系统调用之间的区别。对每个函数或过程的简要描述是为了有助于读者理解本书中的协议软件，并没有给出可能出现的问题和例外情况。因此程序员在编写调用到这些实用例程的程序之前，应当先了解更多的有关参数以及调用规定之类的信息。

atoi(string)

提取字符串string的ASCII值，以函数值形式返回一个整数。

blkcmp(ptr1, ptr2, nbytes)

比较从地址ptr1开始的nhytes个字节和从地址ptr2开始的nbytes个字节的内容，如果相同，返回值0。如果这两个数据块不同，找出第一个不同的字节，如果第一个数据块中的这个字节比第二个数据块中对应的字节小，则返回一个负整数，否则返回一个正整数。

blkcopy(toptr, fromptr, nbytes)

从地址fromptr开始把nbytes字节复制到地址toptr。

blkequ(ptr1, ptr2, nbytes)

比较从地址ptr1开始的nbytes个字节和从地址ptr2开始的nbytes个字节的内容。如果它们相同，则返回TRUE，否则返回FALSE。

chprio(pid, newprio)
将进程 pid 的 CPU 调度优先级改为 newprio，并以函数值形式返回旧的优先级。

close(dev)
关闭设备 dev（对 TCP 来说，指删除连接）。

control(dev, func, arg1, arg2)
控制设备 dev，执行函数 func，该函数的入口参数为 arg1 和 arg2。控制操作依赖于设备。

create(caddr, ssize, prio, pn, nargs, arg...)
创建一个进程，执行以 caddr 为起始地址的代码，初始栈大小为 ssize，CPU 优先级为 prio，进程名为 pn，有 nargs 个参数，从 arg 开始都是参数值。

deq(indx)
从索引值为 indx 的列表中删除第一个表项，并返回该表项。

disable(ps)
将处理器状态字保存在 ps 中，并禁止 CPU 中断。

enq(item, indx, key)
在索引值为 indx 的有序列表中插入 item，利用整型参数 key 来选取 item 在列表中的位置。

fprintf(dev, fmt, value...)
根据格式 fmt，对一个或多个 value 格式化，并将结果发送到设备 dev。

freebuf(bufptr)
将地址为 bufptr 的缓冲区释放回缓冲池。

freemem(memptr, nbytes)
释放地址为 memptr 的 nbytes 个字节的存储区。

freeq(indx)
删除索引为 indx 的列表，将它的所有存储区返回给空闲列表。

getbuf(poolid)
从缓冲池 poolid 中获取一个缓冲区。

getc(dev)
从设备 dev 中读取一个字符。如果没有，则阻塞直至有一个字符到达。

getdmem(nbytes)
分配 nbytes 个字节的、能被用于 DMA I/O 的存储区。

getidprom(buf, size)
从硬件 ID prom 中获取 size 个字节，并将其保存在 buf 缓冲区中。

getmem(nbytes)
从空闲列表中分配nbytes个字节的存储区，并返回一个指向它的指针。

getpid( )
返回当前执行的进程的进程标识符。

getprio( )
返回当前执行的进程的CPU优先级。

gettime(tptr)
获取本地时间，用从新纪元（1970年1月1日）开始经过的秒数来表示，并以长整型保存在地址为tptr的存储区中。

index(str, ch)
返回字符串str中，字符ch第一次出现的位置，如果str中没有字符ch，则返回0。

initq( )
在系统启动时初始化通用列表的机制（必须在newq之前完成）。

kill(pid)
删除标识符为pid的进程。

Kprintf(fmt, value...)
根据格式fmt将一个或多个value转换成ASCII串，并像printf一样在控制台上显示，但它会越过中断系统，因此可以被用来调试操作系统内核。

mark(ptr)
导致内核记住存储区ptr已经被初始化了，并且可以用函数unmarked来测试。

mkpool(bufsiz, numbufs)
创建一个含有numbufs个缓冲区的缓冲池，每个缓冲区的长度为bufsiz字节，并返回缓冲池标识符。

mount(prefix, dev, replace)
向文件名空间增加新的文件名prefix，使其与设备dev及替代字符串replace相关联。

nammap(name, newname)
在文件名空间中找出name的映射名，将新名称写入newname中，并以函数值的形式返回设备标识符。

newq(size, type)
分配一个可容纳size个结点的新列表，通过type决定列表是利用信号量还是通过不允许中断的方式来控制互斥操作的，并返回新列表的索引。

open(dev, name, mode)
使用设备dev打开名为name的文件或对象，并以mode模式访问。返回用于访问该对象的新设

备的设备描述符。

panic(message)
在控制台上显示字符串 message，同时立即暂停（终止）操作系统及所有应用程序。

pcount(portid)
返回当前在端口 portid 上等待处理的报文数目，如果在端口上有 n 个进程因等待报文的到达而阻塞，则返回负值 n。

pcreate(count)
创建一个能容纳 count 个报文的新端口，并返回它的标识号。

pdelete(portid, dispose)
删除标识号为 portid 的端口，调用过程 dispose 释放每个正在等待的报文。

preceive(portid)
从端口 portid 中提取下一个报文，如果端口为空，进入阻塞状态直到有一个报文到达。

printf(fmt, value...)
根据格式 fmt 将一个或多个 value 转换为 ASCII 串，并将其写入控制台设备。

psend(portid, message)
将整数 message 置入标识号为 portid 的端口，如果端口已满，进入阻塞状态，直至端口空间可用。

putc(dev, ch)
向设备 dev 写单个字符 ch。

qsort(darray, n, isize, cmp)
为数组 darray 中的 n 个数值快速排序，每个数值具有 isize 个字节，利用函数 cmp 完成数组元素之间的比较。

read(dev, buf, len)
从设备 dev 中读取 len 个字节的数据，并保存在缓冲区 buf 中，返回读取的字节数。read 的精确含义与设备有关，但大多数设备会在数据到达之前使调用者阻塞。

receive( )
阻塞主调进程，直至该进程有报文到达，然后向主调进程返回该报文。

recvclr( )
如果主调进程有报文到达，则向调用进程返回该报文，否则返回 OK。在两种情况下均不阻塞主调进程。

recvtim(maxdelay)
阻塞主调进程，除非以下两种情况出现：或者有一个报文到达，或者经过了 maxdelay 个 0.1 秒。

并返回一个报文或TIMEOUT。

remove(fname, key)
删除名为fname的文件，key为保护字。

rename(file1, file2)
将文件名file1改为file2。

restore(ps)
将CPU从中断状态恢复到由disable保存在ps中的状态。

resume(pid)
恢复早先被挂起的标识符为pid的进程。

rindex(str, ch)
返回字符串str中字符ch出现的最后位置。如果str中没有字符ch则返回0。

scount(sid)
返回信号量sid的当前计数值。若返回一个负值n，则表示有n个进程在该信号量上被阻塞。

screate(icount)
创建一个新信号量，其初始计数值为icount，并返回它的标识号。

sdelete(sid)
删除标识号为sid的信号量，并释放所有可能在该信号量上被阻塞的进程。

seek(dev, pos)
在设备dev上找到位置pos。其精确含义与设备有关。

seeq(indx)
在索引为indx的列表中逐个表项地搜索，不删除任何表项。在任何时候，列表仅记住一个搜索位置。

send(pid, msg)
向标识号为pid的进程发送报文msg，但如果该进程已经有未读取的报文正在等待，则丢弃报文msg。

sendf(pid, msg)
向标识号为pid的进程发送报文msg，覆盖旧的未读取的报文。

set_evec(vec, func)
设置硬件异常处理矢量vec为指向异常处理函数func的指针。

setdev(pid, idev, odev)
为标识号为pid的进程设置标准输入设备idev，以及标准输出设备odev。

setnok(nok, pid)

为标识号为pid的进程设置“下一个同类（next-of-kin）”为nok，允许在进程pid结束时通知nok。

signal(sid)

释放标识号为sid的信号量，如果有任何一个进程在该信号量上被阻塞，则允许该进程继续执行。

sleep(sdelay)

在返回主调进程前延迟sdelay秒。

sleep10(tsdelay)

在返回调用进程前延迟tsdelay个0.1秒。

sprintf(str, fmt, value)

根据格式fmt将一个或多个value转换为ASCII字符串，并将结果保存在字符串变量str内。

strcat(tostr, fromstr)

将一个以NULL结束的字符串fromstr的副本连接到另一个以NULL结束的字符串tostr的后面。

strcmp(str1, str2)

比较两个以NULL结束的字符串str1和str2，并返回一个负整数、0或一个正整数，分别表示str1按字典顺序小于、等于或大于str2。

strcpy(tostr, fromstr)

将以NULL结束的字符串fromstr的内容复制到另一个以NULL结束的字符串tostr中。

strlen(str)

返回以NULL结束的字符串str的长度，单位为字节，但不包括结束符NULL。

strncat(tostr, fromstr, maxlen)

将以NULL结束的字符串fromstr的副本连接到以NULL结束的另一个字符串tostr的后面，但不能超过最大长度maxlen个字节。

strncmp(str1, str2, maxlen)

比较两个以NULL结束的字符串str1和str2中前maxlen个字节的内容，并返回一个负整数、0或一个正整数，分别表示str1按字典顺序小于、等于或大于str2。

strncpy(tostr, fromstr, maxlen)

将以NULL结束的字符串fromstr复制到另一个以NULL结束的字符串tostr中，但不能超过最大长度maxlen个字节。

suspend(pid)

挂起（阻塞）标识号为pid的进程。

unmount(prefix)

删除名称空间中的一个以prefix为字首的名称映射。

wait(sid)

将信号量sid的计数值减1，如果结果得到的计数值是一个负数，则在信号量sid上阻塞主调进程。只有在另一个进程调用signal释放信号量sid之后，这个被阻塞的进程才能继续执行。

write(dev, buf, len)

将缓冲区buf中的len个字节写入设备dev。write的精确含义依赖于设备，但大多数设备将会阻塞主调进程，直至所有字节都能够被写入。

Xinu系统中的包含文件

本书提供完整的程序代码中包括三个来自Xinu系统的文件：network.h, kernel.h和conf.h，它们定义了符号常量，类型名称以及宏。文件network.h中包括了所有与网络应用代码有关的.h文件。另外，它还定义了诸如完成一个数值在网络字节顺序和本机字节顺序之间的转换的net2hs之类的宏。

```
/* network.h */

#include <systypes.h>
#include <string.h>
#include <sem.h>
#include <mark.h>
#include <bufpool.h>
#include <q.h>
#include <io.h>

/* All includes needed for the network */

#include <lereg.h>
#include <ether.h>
#include <ip.h>
#include <netif.h>
#include <ipreass.h>
#include <icmp.h>
#include <udp.h>
#include <tcp.h>
#include <tcpfsm.h>
#include <tcb.h>
#include <tcpstat.h>
#include <tcptimer.h>
#include <net.h>
#include <dgram.h>
#include <arp.h>
#include <fserver.h>
```

```
#include <rfile.h>
#include <domain.h>
#include <dma.h>
#include <route.h>
#ifdef RIP
#include <rip.h>
#ifdef /* RIP */
#include <daemon.h>
#include <snmpvars.h>

/* Declarations data conversion and checksum routines */

extern unsigned short cksum();    /* 1s comp of 16-bit 1s comp sum*/

#if BYTE_ORDER == LITTLE_END1AN
#define hs2net(x) (unsigned) ((((x)>>8) &0xff)|(((x) &0xff)<<8))
#define net2hs(x) hs2net(x)

#define hl2net(x)      ((((x) & 0xff)<<24)|((x)>>24) & 0xff)|\
        (((x) & 0xff0000)>>8) | (((x) & 0xff00)<<8))
#define net2hI(x) hI2net(x)
#endif

#if  BYTE_ORDER == BIG_ENDIAN
#define hs2net(x) (x)
#define net2hs(x) (x)
#define hl2net(x) (x)
#define net2hl(x) (x)
#endif

/* network macros */
#define hi8(x)   (unsigned char)  (((long) (x) >> 16) & 0x00ff)
#define low16(x) (unsigned short) ((long) (x) & 0xffff)
#define     BYTE(x,y) ((x)[(y)] & 0xff)  /* get byte "y" from ptr "x" */
```

文件kernel.h中包含被所有操作系统函数使用的各种定义。它定义了返回代码SYSERR和OK以及其他在程序代码中出现的常量的值。它还定义了用于定义过程的标号PROCESS和LOCAL。

```
/* kernel.h - disable, enable, halt, restore, isodd, min, max */

#include <stdlib.h>

/* Symbolic constants used throughout Xinu */

typedef char          Bool;      /* Boolean type                    */
#define FALSE         0          /* Boolean constants               */
#define TRUE          1
#define EMPTY         (-1)       /* an illegal gpq                  */
```

```
#define NULL        0          /* Null pointer for linked lists     */
#define NULLCH      '\0'       /* The null character                */
#define NULLSTR     ""         /* Pointer to empty string           */
#define SYSCALL     int        /* System call declaration           */
#define LOCAL       static     /* Local procedure declaration       */
#define COMMAND     int        /* Shell command declaration         */
#define BUILTIN     int        /* Shell builtin " "                 */
#define INTPROC     void       /* Interrupt procedure  "            */
#define PROCESS     int        /* Process declaration               */
#define RESCHYES    1          /* tell ready to reschedule          */
#define RESCHNO     0          /* tell ready not to resch.          */
#define MININT      0x80000000
#define MAXINT      0x7fffffff
#define LOWBYTE     0377       /* mask for low-order 8 bits         */
#define HIBYTE      0177400    /* mask for high 8 of 16 bits        */
#define LOW16       0177777    /* mask for low-order 16 bits        */
#define MINSTK      3072       /* minimum process stack size        */
#define NULLSTK     MINSTK     /* process 0 stack size              */
#define MAGIC       0125251    /* unusual value for top of stk      */

/* Universal return constants */

#define OK        1            /* system call ok                    */
#define SYSERR   -1            /* system call failed                */
#define EOF      -2            /* End-of-file (usu. from read)      */
#define TIMEOUT  -3            /* time out  (usu. recvtim)          */
#define INTRMSG  -4            /* keyboard "intr" key pressed       */
                               /*  (usu. defined as ^B)             */
#define BLOCKERR -5            /* non-blocking op would block       */

/* Initialization constants                                         */

#define INITSTK     1024       /* initial process stack size   */
#define INITPRIO    20             /* initial process priority     */
#define INITNAME    "main"         /* initial process name         */
#define INITARGS    1,0            /* initial count/arguments      */
#define INITRET     userret        /* processes return address     */
#define INITREG     0              /* initial register contents    */
#define QUANTUM     10             /* clock ticks until preemption */

/* Machine size definitions                                         */

typedef char    CHAR;     /* sizeof the unit the holds a character  */
typedef int     WORD;     /* maximum of (int, char *)               */
typedef char    *PTR;     /* sizeof a char. or fcnt. pointer        */
typedef int     INT;      /* sizeof compiler integer                */
typedef int     REG;      /* sizeof machine register                */
```

```
#define MAXLONG        0x7fffffff
#define MINLONG        0x80000000

typedef short  STATWORD[1]; /* machine status for disable/restore     */
                            /* by declaring it to be an array, the    */
                            /* name provides an address so forgotten  */
                            /* &'s don't become a problem             */

/* Miscellaneous utility inline functions */
#define     isodd(x)    (01&(WORD)(x))
#define     min(a,b)    ( (a) < (b) ? (a) : (b) )
#define     max(a,b)    ( (a) > (b) ? (a) : (b) )

extern int  rdyhead, rdytail;

extern int  preempt;

extern int  noint;

void disable(STATWORD), restore(STATWORD);
void kprintf(const char*, ...);
void panic(const char*, ...);
SYSCALL getpid(void);
```

在文件conf.h中定义的常量用于某个Xinu系统的设置。例如，它定义了TCP用于发送和接收数据的缓冲区的大小（TCPSBS和TCPRBS）。另外，conf.h文件中还包括组成设备驱动程序的各个函数的外部（extern）定义。

```
/* conf.h (GENERATED FILE; DO NOT EDIT) */

#define NULLPTR (char *)0

/* Devioe table declarations */
struct devsw     {          /* device table entry */
       int       dvnum;
       char      *dvname;
       int       (*devinit) (struct devsw *);
       int       (*dvopen) (struct devsw *, void *, void *);
       int       (*dvclose) (struct devsw *);
       int       (*dvread) (struct devsw *, char *, unsigned);
       int       (*dvwrite) (struct devsw *, unsigned char *, unsigned);
       int       (*dvseek) (struct devsw *, long);
       int       (*dvgetc) (struct devsw *);
       int       (*dvputc) (struct devsw *, unsigned char);
       int       (*dvcntl) (struct devsw *, int, void *, void *);
       void      *dvcsr;
       int       dvivec;
```

```
        int         dvovec;
        void        (*dviint) (struct devsw *, unsigned char);
        void        (*dvoint) (struct devsw *);
        void        *dvioblk;
        int         dvminor;
        };

extern  struct  devsw  devtab[];        /* one entry per device */

/* Device name definitions */

#define KBMON     0                 /* type kbmon   */
#define SERIAL0   1                 /* type serial  */
#define SERIAL1   2                 /* type serial  */
#define CONSOLE   3                 /* type tty     */
#define TTY0      4                 /* type tty     */
#define TTY1      5                 /* type tty     */
#define TTY2      6                 /* type tty     */
#define ETHER     7                 /* type eth     */
#define UDP       8                 /* type dgm     */
#define DGRAM0    9                 /* type dg      */
#define DGRAM1    10                /* type dg      */
#define DGRAM2    11                /* type dg      */
#define DGRAM3    12                /* type dg      */
#define DGRAM4    13                /* type dg      */
#define DGRAM5    14                /* type dg      */
#define DGRAM6    15                /* type dg      */
#define DGRAM7    16                /* type dg      */
#define DGRAM8    17                /* type dg      */
#define DGRAM9    18                /* type dg      */
#define DGRAMA    19                /* type dg      */
#define DGRAMB    20                /* type dg      */
#define DGRAMC    21                /* type dg      */
#define DGRAMD    22                /* type dg      */
#define DGRAME    23                /* type dg      */
#define DGRAMF    24                /* type dg      */
#define TCP       25                /* type tcpm    */
#define TCP0      26                /* type tcp     */
#define TCP1      27                /* type tcp     */
#define TCP2      28                /* type tcp     */
#define TCP3      29                /* type tcp     */
#define TCP4      30                /* type tcp     */
#define TCP5      31                /* type tcp     */
#define TCP6      32                /* type tcp     */
#define TCP7      33                /* type tcp     */
#define TCP8      34                /* type tcp     */
```

```
#define TCP9      35              /* type tcp     */
#define TCPA      36              /* type tcp     */
#define TCPB      37              /* type tcp     */
#define TCPC      38              /* type tcp     */
#define TCPD      39              /* type tcp     */
#define TCPE      40              /* type tcp     */
#define TCPF      41              /* type tcp     */
#define RFILSYS   42              /* type rfm     */
#define RFILE1    43              /* type rf      */
#define RFILE2    44              /* type rf      */
#define RFILE3    45              /* type rf      */
#define RFILE4    46              /* type rf      */
#define NAMESPACE 47              /* type nam     */

/* Control block sizes */

#define Nkbmon    1
#define Nserial   2
#define Ntty      4
#define Neth      1
#define Ndgm      1
#define Ndg       16
#define Ntcpm     1
#define Ntcp      16
#define Nrfm      1
#define Nrf       4
#define Nnam      1

#define NDEVS     48

/* Declarations of I/O routines referenced */

extern  int     kbminit(struct devsw *);
extern  int     noopen(struct devsw *, void *, void *);
extern  int     noclose(struct devsw *);
extern  int     kbmread(struct devsw *, char *, unsigned);
extern  int     kbmwrite(struct devsw *, unsigned char *, unsigned);
extern  int     noseek(struct devsw *, long);
extern  int     kbmcntl(struct devsw *, int, void *, void *);
extern  int     kbmgetc(struct devsw *);
extern  int     kbmputc(struct devsw *, unsigned char);
extern  void    kbmint(struct devsw *, unsigned char);
extern  void    nooint(struct devsw * );
extern  int     cominit(struct devsw * );
extern  int     noread(struct devsw *, char *, unsigned);
extern  int     comwrite(struct devsw *, unsigned char *, unsigned);
extern  int     comcntl(struct devsw *, int, void *, void *);
```

```
extern  int     comgetc(struct devsw *);
extern  int     computc(struct devsw *, unsigned char);
extern  void    comint(struct devsw *, unsigned char);
extern  int     ttyinit(struct devsw *);
extern  int     ttyopen(struct devsw *, void *, void *);
extern  int     ttyread(struct devsw *, char *, unsigned);
extern  int     ttywrite(struct devsw *, unsigned char *, unsigned);
extern  int     ttycntl(struct devsw *, int, void *, void *);
extern  int     ttygetc(struct devsw * );
extern  int     ttyputc(struct devsw *, unsigned char);
extern  void    ttyiin(struct devsw *, unsigned char);
extern  void    ttyoin(struct devsw *);
extern  int     ue_init(struct devsw * );
extern  int     ue_write(struct devsw *, unsigned char *, unsigned);
extern  int     ue_cntl(struct devsw *, int, void *, void *);
extern  int     nogetc(struct devsw *);
extern  int     noputc(struct devsw *, unsigned char);
extern  void    ue_intr(struct devsw *);
extern  int     ionull(struct devsw *);
extern  int     dgmopen(struct devsw *, void *, void *);
extern  int     nowrite(struct devsw *, unsigned char *, unsigned);
extern  int     dgmcntl(struct devsw *, int, void *, void *);
extern  void    noiint(struct devsw *, unsigned char);
extern  int     dginit(struct devsw *);
extern  int     dgclose(struct devsw *);
extern  int     dgread(struct devsw *, char *, unsigned);
extern  int     dgwrite(struct devsw *, unsigned char *, unsigned);
extern  int     dgcntl(struct devsw *, int, void *, void *);
extern  int     tcpmopen(struct devsw *, void *, void *);
extern  int     tcpmcntl(struct devsw *, int, void *, void *);
extern  int     tcpinit(struct devsw * );
extern  int     tcpclose(struct devsw *);
extern  int     tcpread(struct devsw *, char *, unsigned);
extern  int     tcpwrite(struct devsw *, unsigned char *, unsigned);
extern  int     tcpcntl(struct devsw *, int, void *, void *);
extern  int     tcpgetc(struct devsw *);
extern  int     tcpputc(struct devsw *, unsigned char);
extern  void    ioerr(void);
extern  int     rfopen(struct devsw *, void *, void *);
extern  int     rfcntl(struct devsw *, int, void *, void *);
extern  int     rfinit(struct devsw *);
extern  int     rfclose(struct devsw *);
extern  int     rfread(struct devsw *, char *, unsigned);
extern  int     rfwrite(struct devsw *, unsigned char *, unsigned);
extern  int     rfseek(struct devsw *, long);
```

```
extern  int     rfgetc(struct devsw *);
extern  int     rfputc(struct devsw *, unsigned char);
extern  int     naminit(struct devsw *);
extern  int     namopen(struct devsw *, void *, void *);
extern  int     nocntl(struct devsw *, int, void *, void *);

/* Configuration and Size Constants */

#define LITTLE_ENDIAN   0x1234

#define BIG_ENDIAN      0x4321

#define BYTE_ORDER      LITTLE_ENDIAN

#define CONTACT  "dis@purdue.edu"     /* administrative contact       */
#define LOCATION "Xinu Lab"           /* Host's physical location     */

#define NPROC           50            /* number of user processes     */
#define NSEM            100           /* number of semaphores         */
#define MEMMARK                       /* define if memory marking used*/
#define RTCLOCK                       /* now have RTC support         */
#define STKCHK                        /* resched checks stack overflow*/
#define NETDAEMON                     /* Ethernet network daemon runs */
#define OSPF                          /* define if using OSPF         */
#ifdef notdef
#define NETMON                        /* define for network monitor   */
#endif
#define SNMP                          /* define for SNMP & MIB        */
#define RIP                           /* define if using RIP          */
#define MULTICAST                     /* define if using multicasting */

#ifdef  OSPF
#define MULTICAST                     /* OSPF uses multicasting       */
#endif   /* OSPF */

#define IPADDR1 "10.3.0.3"            /* interface 1 IP address       */
#define DEFRTR  "10.3.0.1"            /* default router               */
#define TSERVER "10.3.0.1:37"         /* Time server address          */
#define RSERVER "10.3.0.1:2001"       /* Remote file server address   */
#define NSERVER "10.3.0.1:53"         /* Domain Name server address   */
#define LOGHOST "10.3.0.1:514"        /* syslog server address        */

#define BINGID   9                    /* Othernet simlated net param. */
#define SMALLMTU 400                  /* for OTHER2; sim. small MTU   */

#define BPMAXB   8192                 /* max buffer size for mkpool   */
#define BPMAXN   128                  /* max # buffers in a buf pool  */
#define TCPSBS   4096                 /* TCP send buffer sizes        */
```

```
#define TCPRBS    8192                    /* TCP receive buffer sizes     */

#define NPORTS    100
#define TIMERGRAN      1                  /* timer granularity, secs/10   */
#define NAREA          1                  /* max number of OSPF Areas     */

#ifndef Noth
#define Noth      0
#endif  !Noth
#ifndef Ndg
#define Ndg       0
#endif
```

conf.h有点与众不同，因为它是由程序根据系统配置规范自动生成的。例如，它定义Ntcp为16，因为原先的输入规范定义了TCP连接的16个表位置。本书中提供的程序代码不包含定义大小的常量，它由文件conf.h中的诸如Ntcp这样的符号常量提供。只要原先的输入规范发生了改变，Xinu会自动生成新版本的conf.h。因此只要程序员改变了某个参数，如缓冲区大小，则支持软件会自动生成新版本的系统。

参考文献

ABRAMSON, N. [1970], The ALOHA System – Another Alternative for Computer Communications, *Proceedings of the Fall Joint Computer Conference.*

ABRAMSON, N. and F. KUO (EDS.) [1973], *Computer Communication Networks,* Prentice Hall, Englewood Cliffs, New Jersey.

ANDREWS, D. W., and G. D. SHULTZ [1982], A Token-Ring Architecture for Local Area Networks: An Update, *Proceedings of Fall 82 COMPCON,* IEEE.

BALL, J. E., E. J. BURKE, I. GERTNER, K. A. LANTZ, and R. F. RASHID [1979], Perspectives on Message-Based Distributed Computing, *IEEE Computing Networking Symposium,* 46-51.

BBN [1981], A History of the ARPANET: The First Decade, *Technical Report* Bolt, Beranek, and Newman, Inc.

BBN [December 1981], Specification for the Interconnection of a Host and an IMP (revised), *Technical Report 1822,* Bolt, Beranek, and Newman, Inc.

BERTSEKAS D., and R. GALLAGER [1991], *Data Networks,* 2nd edition, Prentice-Hall, Englewood Cliffs, New Jersey.

BIAGIONI E., E. COOPER, and R. SANSOM [March 1993], Designing a Practical ATM LAN, *IEEE Network,* 32-39.

BIRRELL, A., and B. NELSON [February 1984], Implementing Remote Procedure Calls, *ACM Transactions on Computer Systems,* 2(1), 39-59.

BLACK, U., [1995], *ATM: Foundation for Broadband Networks,* Prentice-Hall, Englewood Cliffs, New Jersey.

BOGGS, D., J. SHOCH, E. TAFT, and R. METCALFE [April 1980], Pup: An Internetwork Architecture, *IEEE Transactions on Communications.*

BORMAN, D., [April 1989], Implementing TCP/IP on a Cray Computer, *Computer Communication Review,* 19(2), 11-15.

BROWN, M., K. KOLLING, and E. TAFT [November 1985], The Alpine File System, *ACM Transactions on Computer Systems,* 3(4), 261-293.

BROWNBRIDGE, D., L. MARSHALL, and B. RANDELL [December 1982], The Newcastle Connections or UNIXes of the World Unite!, *Software – Practice and Experience,* 12(12), 1147-1162.

CASNER, S., and S. DEERING [July 1992], First IETF Internet Audiocast, *Computer Communications Review*, 22(3), 92-97.

CERF, V., and E. CAIN [October 1983], The DOD Internet Architecture Model, *Computer Networks*.

CERF, V., and R. KAHN [May 1974], A Protocol for Packet Network Interconnection, *IEEE Transactions of Communications*, Com-22(5).

CERF, V. [October 1989], A History of the ARPANET, *ConneXions, The Interoperability Report*, 480 San Antonio Rd, Suite 100, Mountain View, California.

CHERITON, D. R. [1983], Local Networking and Internetworking in the V-System, *Proceedings of the Eighth Data Communications Symposium*.

CHERITON, D. R. [April 1984], The V Kernel: A Software Base for Distributed Systems, *IEEE Software*, 1(2), 19-42.

CHERITON, D. [August 1986], VMTP: A Transport Protocol for the Next Generation of Communication Systems, *Proceedings of ACM SIGCOMM '86*, 406-415.

CHERITON, D., and T. MANN [May 1984], Uniform Access to Distributed Name Interpretation in the V-System, *Proceedings IEEE Fourth International Conference on Distributed Computing Systems*, 290-297.

CHESSON, G. [June 1987], Protocol Engine Design, *Proceedings of the 1987 Summer USENIX Conference*, Phoenix, AZ.

CHESWICK, W., and S. BELLOVIN [1998], *Firewalls And Internet Security: Repelling the Wiley Hacker*, 2nd edition, Addison-Wesley, Reading, Massachusetts.

CLARK, D. [December 1985], The structure of Systems Using Upcalls, *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 171-180.

CLARK, D., M. LAMBERT, and L. ZHANG [August 1987], NETBLT: A High Throughput Transport Protocol, *Proceedings of ACM SIGCOMM '87*.

CLARK, D., V. JACOBSON, J. ROMKEY, and H. SALWEN [June 1989], An Analysis of TCP Processing Overhead, *IEEE Communications*, 23-29.

COHEN, D., [1981], On Holy Wars and a Plea for Peace, *IEEE Computer*, 48-54.

COMER, D. E. and J. T. KORB [1983], CSNET Protocol Software: The IP-to-X25 Interface, *Computer Communications Review*, 13(2).

COMER, D. E. [1984], *Operating System Design – The XINU Approach*, Prentice-Hall, Englewood Cliffs, New Jersey.

COMER, D. E. [1987], *Operating System Design Vol II. – Internetworking With XINU*, Prentice-Hall, Englewood Cliffs, New Jersey.

COMER, D. E., T. NARTEN, and R. YAVATKAR [April 1987], The Cypress Network: A Low-Cost Internet Connection Technology, *Technical Report TR-653*, Purdue University, West Lafayette, IN.

COMER, D. E., T. NARTEN, and R. YAVATKAR [1987], The Cypress Coaxial Packet Switch, *Computer Networks and ISDN Systems*, vol. 14:2-5, 383-388.

COMER, D. E. [1995] *Internetworking With TCP/IP: Volume 1: Principles, Protocols, and Architecture*, 3rd edition, Prentice-Hall, Englewood Cliffs, New Jersey.

COMER, D. E. and D. L. STEVENS [1996] *Internetworking With TCP/IP Volume III Client-Server Programming And Applications, BSD socket version*, 2nd edition, Prentice-Hall, Englewood Cliffs, New Jersey.

COMER, D. E. and D. L. STEVENS [1994] *Internetworking With TCP/IP Volume III – Client-Server Programming And Applications, AT&T TLI version*, Prentice-Hall, Englewood Cliffs, New Jersey.

COMER, D. E. and D. L. STEVENS [1997] *Internetworking With TCP/IP Volume III – Client-Server Programming And Applications, Windows Sockets version*, Prentice-Hall, Englewood Cliffs, New Jersey.

COTTON, I. [1979], Technologies for Local Area Computer Networks, *Proceedings of the Local Area Communications Network Symposium.*

CROWLEY, T., H, FORSDICK, M. LANDAU, and V. TRAVERS [June 1987], The Diamond Multimedia Editor, *Proceedings of the 1987 Summer USENIX Conference*, Phoenix, AZ.

CYPSER, R. [1991], Communications for Cooperating Systems: OSI, SNA, and TCP/IP, Addison-Wesley, Reading, Massachusetts.

DALAL Y. K., and R. S. PRINTIS [1981], 48-Bit Absolute Internet and Ethernet Host Numbers, *Proceedings of the Seventh Data Communications Symposium.*

DEERING S. E., and D. R. CHERITON [May 1990], Multicast Routing in Datagram Internetworks and Extended LANs, *ACM Transactions on Computer Systems*, 8(2), 85-110.

DEERING, S., D. ESTRIN, D. FARINACCI, V. JACOBSON, C-G LIU, and L. WEI [August 1994], An Architecture for Wide-Area Multicasting Routing, *Proceedings of ACM SIGCOMM '94*, 126-135.

DENNING P. J., [September-October 1989], *The Science of Computing: Worldnet*, in American Scientist, 432-434.

DENNING P. J., [November-December 1989], *The Science of Computing: The ARPANET After Twenty Years*, in American Scientist, 530-534.

DE PRYCKER, M. [1995], *Asynchronous Transfer Mode Solution for Broadband ISDN*, 3rd edition, Prentice-Hall, Englewood Cliffs, New Jersey.

DIGITAL EQUIPMENT CORPORATION., INTEL CORPORATION, and XEROX CORPORATION [September 1980], *The Ethernet: A Local Area Network Data Link Layer and Physical Layer Specification.*

DION, J. [Oct. 1980], The Cambridge File Server, *Operating Systems Review*, 14(4), 26-35.

DRIVER, H., H. HOPEWELL, and J. IAQUINTO [September 1979], How the Gateway Regulates Information Control, *Data Communications.*

EDGE, S. W. [1979], Comparison of the Hop-by-Hop and Endpoint Approaches to Network Interconnection, in *Flow Control in Computer Networks*, J-L. GRANGE and M. GIEN (EDS.), North-Holland, Amsterdam, 359-373.

EDGE, S. [1983], An Adaptive Timeout Algorithm for Retransmission Across a Packet Switching Network, *Proceedings of ACM SIGCOMM '83.*

ENSLOW, P. [January 1978], What is a 'Distributed' Data Processing System? *Computer*, 13-21.

ERIKSSON, H. [August 1994], MBONE: The Multicast Backbone, *Communications of the ACM*, 37(8), 54-60.

FALK, G. [1983], The Structure and Function of Network Protocols, in *Computer Communications, Volume I: Principles*, CHOU, W. (ED.), Prentice-Hall, Englewood Cliffs, New Jersey.

FARMER, W. D., and E. E. NEWHALL [1969], An Experimental Distributed Switching System to Handle Bursty Computer Traffic, *Proceedings of the ACM Symposium on Probabilistic Optimization of Data Communication Systems*, 1-33.

FEDOR, M. [June 1988], GATED: A Multi-Routing Protocol Daemon for UNIX, *Proceedings of the 1988 Summer USENIX conference*, San Francisco, California.

FEINLER, J., O. J. JACOBSEN, and M. STAHL [December 1985], *DDN Protocol Handbook Volume Two, DARPA Internet Protocols*, DDN Network Information Center, SRI International, 333 Ravenswood Avenue, Room EJ291, Menlo Park, California.

FLOYD, S. and V. JACOBSON [August 1993], Random Early Detection Gateways for Congestion Avoidance, *IEEE/ACM Transactions on Networking*, 1(4).

FRANK, H., and W. CHOU [1971], Routing in Computer Networks, *Networks*, 1(1), 99-112.

FRANK, H., and J. FRISCH [1971], *Communication, Transmission, and Transportation Networks*, Addison-Wesley, Reading, Massachusetts.

FRICC [May 1989], *Program Plan for the National Research and Education Network*, Federal Research Internet Coordinating Committee, US Department of Energy, Office of Scientific Computing report ER-7.

FULTZ, G. L., and L. KLEINROCK, [June 14-16, 1971], Adaptive Routing Techniques for Store-and-Forward Computer Communication Networks, presented at *IEEE International Conference on Communications*, Montreal, Canada.

GERLA, M., and L. KLEINROCK [April 1980], Flow Control: A Comparative Survey, *IEEE Transactions on Communications*.

GREEN, P. E. (ED.) [1982], *Computer Network Architectures and Protocols*, Plenum Press, New York.

HINDEN, R., J. HAVERTY, and A. SHELTZER [September 1983], The DARPA Internet: Interconnecting Heterogeneous Computer Networks with Gateways, *Computer*.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION [June 1986a], Information processing systems — Open Systems Interconnection — *Transport Service Definition*, International Standard number 8072, ISO, Switzerland.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION [July 1986b], Information processing systems — Open Systems Interconnection — *Connection Oriented Transport Protocol Specification*, International Standard number 8073, ISO, Switzerland.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION [May 1987a], Information processing systems — Open Systems Interconnection — *Specification of Basic Specification of Abstract Syntax Notation One (ASN.1)*, International Standard number 8824, ISO, Switzerland.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION [May 1987b], Information processing systems — Open Systems Interconnection — *Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, International Standard number 8825, ISO, Switzerland.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION [May 1988a], Information processing systems — Open Systems Interconnection — *Management Information Service Definition, Part 2: Common Management Information Service*, Draft International Standard number 9595-2, ISO, Switzerland.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION [May 1988a], Information processing systems — Open Systems Interconnection — *Management Information Protocol Definition, Part 2: Common Management Information Protocol*, Draft International Standard number 9596-2.

JACOBSON, V. [August 1988], Congestion Avoidance and Control, *Proceedings ACM SIGCOMM '88*.

JAIN, R. [January 1985], On Caching Out-of-Order Packets in Window Flow Controlled Networks, *Technical Report*, DEC-TR-342, Digital Equipment Corporation.

JAIN, R. [March 1986], Divergence of Timeout Algorithms for Packet Retransmissions, *Proceedings Fifth Annual International Phoenix Conference on Computers and Communications*, Scottsdale, AZ.

JAIN, R. [October 1986], A Timeout-Based Congestion Control Scheme for Window Flow-Controlled Networks, *IEEE Journal on Selected Areas in Communications*, Vol. SAC-4, no. 7.

JAIN, R., K. RAMAKRISHNAN, and D-M. CHIU [August 1987], Congestion Avoidance in Computer Networks With a Connectionless Network Layer. *Technical Report*, DEC-TR-506, Digital Equipment Corporation.

JAIN, R. [1991], The Art of Computer Systems Performance Analysis, John Wiley & Sons, New York.

JAIN, R. [May 1992], Myths About Congestion Management in High-speed Networks, *Internetworking: Research and Experience*, 3(3), 101-113.

JENNINGS, D. M., L. H. LANDWEBER, and I. H. FUCHS [February 28, 1986], Computer Networking for Scientists and Engineers, *Science* vol 231, 941-950.

JUBIN, J. and J. TORNOW [January 1987], The DARPA Packet Radio Network Protocols, *IEEE Proceedings*.

KAHN, R. [November 1972], Resource-Sharing Computer Communications Networks, *Proceedings of the IEEE*, 60(11), 1397-1407.

KARN, P., H. PRICE, and R. DIERSING [May 1985], Packet Radio in the Amateur Service, *IEEE Journal on Selected Areas in Communications*,

KARN, P., and C. PARTRIDGE [August 1987], Improving Round-Trip Time Estimates in Reliable Transport Protocols, *Proceedings of ACM SIGCOMM '87*.

KENT, C., and J. MOGUL [August 1987], Fragmentation Considered Harmful, *Proceedings of ACM SIGCOMM '87*.

KLINE, C. [August 1987], Supercomputers on the Internet: A Case Study, *Proceedings of ACM SIGCOMM '87*.

LABARRE, L. (ED.) [December 1989], OSI Internet Management: Management Information Base, *Internet Draft <IETF.DRAFTS>DRAFT-IETF-SNMP-MIB2-01.TXT*, DDN Network Information Center, SRI International, Ravenswood, CA.

LAMPSON, B. W., M. PAUL, and H. J. SIEGERT (EDS.) [1981], *Distributed Systems - Architecture and Implementation (An Advanced Course)*, Springer-Verlag, Berlin.

LANZILLO, A. L., and C. PARTRIDGE [January 1989], Implementation of Dial-up IP for UNIX Systems, *Proceedings 1989 Winter USENIX Technical Conference*, San Diego, CA.

LAQUEY, T. L., [July 1989], *Users' Directory of Computer Networks*, Digital Press, Bedford, MA.

LAZAR, A. [November 1983], Optimal Flow Control of a Class of Queuing Networks in Equilibrium. *IEEE Transactions on Automatic Control*, Vol. AC-28:11.

LEFFLER, S., M. McKUSICK, M. KARELS, and J. QUARTERMAN [1996], *The Design and Implementation of the 4.4BSD UNIX Operating System*, Addison Wesley, Reading, Massachusetts.

MCNAMARA, J. [1998], *Technical Aspects of Data Communications*, 2nd edition, Digital Press, Digital Equipment Corporation, Bedford, Massachusetts.

MCQUILLAN, J. M., I. RICHER, and E. ROSEN [May 1980], The New Routing Algorithm for the ARPANET, *IEEE Transactions on Communications*, (COM-28), 711-719.

MERIT [November 1987], Management and Operation of the NSFNET Backbone Network: A Proposal Funded by the National Science Foundation and the State of Michigan, *MERIT Incorporated*, Ann Arbor, Michigan.

METCALFE, R. M., and D. R. BOGGS [July 1976], Ethernet: Distributed Packet Switching for Local Computer Networks, *Communications of the ACM*, 19(7), 395-404.

MILLER, C. K., and D. M. THOMPSON [March 1982], Making a Case for Token Passing in Local Networks, *Data Communications*.

MILLS, D., and H-W. BRAUN [August 1987], The NSFNET Backbone Network, *Proceedings of ACM SIGCOMM '87*.

MITCHELL, J., and J. DION [April 1982], A Comparison of Two Network-Based File Servers, *Communications of the ACM*, 25(4), 233-245.

MORRIS, R. [1979], Fixing Timeout Intervals for Lost Packet Detection in Computer Communication Networks, *Proceedings AFIPS National Computer Conference*, AFIPS Press, Montvale, New Jersey.

NAGLE, J. [April 1987], On Packet Switches With Infinite Storage, *IEEE Transactions on Communications*, Vol. COM-35:4.

NARTEN, T. [Sept. 1989], Internet Routing, *Proceedings ACM SIGCOMM '89*.

NEEDHAM, R. M. [1979], System Aspects of the Cambridge Ring, *Proceedings of the ACM Seventh Symposium on Operating System Principles*, 82-85.

NELSON, J. [September 1983], 802: A Progress Report, *Datamation*.

OPPEN, D., and Y. DALAL [October 1981], The Clearinghouse: A Decentralized Agent for Locating Named Objects, Office Products Division, XEROX Corporation.

PARTRIDGE, C. [June 1986], Mail Routing Using Domain Names: An Informal Tour, *Proceedings of the 1986 Summer USENIX Conference*, Atlanta, GA.

PARTRIDGE, C. [June 1987], Implementing the Reliable Data Protocol (RDP), *Proceedings of the 1987 Summer USENIX Conference*, Phoenix, Arizona.

PARTRIDGE, C. [1994], *Gigabit Networking*, Addison-Wesley, Reading, Massachusetts.

PELTON, J. [1995], Wireless and Stellite Telecommunications, Prentice-Hall, Englewood Cliffs, New Jersey.

PERLMAN, R. [1992], *Interconnections: Bridges and Routers*, Addison-Wesley, Reading, Massachusetts.

PETERSON, L. [1985], *Defining and Naming the Fundamental Objects in a Distributed Message System*, Ph.D. Dissertation, Purdue University, West Lafayette, Indiana.

PETERSON, L., and B. DAVIE, [1996], *Computer Networks: A Systems Approach*, Morgan Kaufmann, San Francisco, CA.

PIERCE, J. R. [1972], Networks for Block Switching of Data, *Bell System Technical Journal*, 51.

POSTEL, J. B. [April 1980], Internetwork Protocol Approaches, *IEEE Transactions on Communications*, COM-28, 604-611.

POSTEL, J. B., C. A. SUNSHINE, and D. CHEN [1981], The ARPA Internet Protocol, *Computer Networks*.

QUARTERMAN, J. S. [1990], *The Matrix: Computer Networks and Conferencing Systems Worldwide*, Digital Press, Digital Equipment Corporation, Maynard, MA.

QUARTERMAN, J. S., and J. C. HOSKINS [October 1986], Notable Computer Networks, *Communications of the ACM*, 29(10).

REYNOLDS, J., J. POSTEL, A. R. KATZ, G. G. FINN, and A. L. DESCHON [October 1985], The DARPA Experimental Multimedia Mail System, *IEEE Computer*.

RITCHIE, D. M. [October 1984], A Stream Input-Output System, *AT&T Bell Laboratories Technical Journal*, 63(8), 1987-1910.

RITCHIE, D. M., and K. THOMPSON [July 1974], The UNIX Time-Sharing System, *Communications of the ACM*, 17(7), 365-375; revised and reprinted in *Bell System Technical Journal*, 57(6), [July-August 1978], 1905-1929.

ROSE, M. (ED.) [October 1989], Management Information Base for Network Management of TCP/IP-based Internets, *Internet Draft <IETF.DRAFTS>DRAFT-IETF-OIM-MIB2-00.TXT*, DDN Network Information Center, SRI International, Ravenswood, CA.

ROSE, M. [1993], *The Internet Message:Closing The Book with Electronic Mail*, Prentice-Hall, Englewood Cliffs, New Jersey.

ROSENTHAL, R. (ED.) [November 1982], *The Selection of Local Area Computer Networks*, National Bureau of Standards Special Publication 500-96.

SALTZER, J. [1978], Naming and Binding of Objects, *Operating Systems, An Advanced Course*, Springer-Verlag, 99-208.

SALTZER, J. [April 1982], Naming and Binding of Network Destinations, *International Symposium on Local Computer Networks*, IFIP/T.C.6, 311-317.

SALTZER, J., D. REED, and D. CLARK [November 1984], End-to-End Arguments in System Design, *ACM Transactions on Computer Systems*, 2(4), 277-288.

SCHWARTZ, M., and T. STERN [April 1980], *IEEE Transactions on Communications*, COM-28(4), 539-552.

SHOCH, J. F. [1978], Internetwork Naming, Addressing, and Routing, *Proceedings of COMPCON.*

SHOCH, J. F., Y. DALAL, and D. REDELL [August 1982], Evolution of the Ethernet Local Computer Network, *Computer.*

SNA [1975], *IBM System Network Architecture – General Information,* IBM System Development Division, Publications Center, Department E01, P.O. Box 12195, Research Triangle Park, North Carolina, 27709.

SOLOMON, J. [1997], *Mobile IP: The Internet Unplugged,* Prentice-Hall, Englewood Cliffs, New Jersey.

SOLOMON, M., L. LANDWEBER, and D. NEUHEGEN [1982], The CSNET Name Server, *Computer Networks* (6), 161-172.

STALLINGS, W. [1985], *Data and Computer Communications,* 5th edition, Prentice-Hall, Englewood Cliffs, New Jersey.

STALLINGS, W. [1997], *Local and Metropolitan Area Networks,* Prentice-Hall, Englewood Cliffs, New Jersey.

STALLINGS, W. [1998], *High-Speed Networks: TCP/IP and ATM Design Principles,* Prentice-Hall, Englewood Cliffs, New Jersey.

STEVENS, W. R. [1998], *UNIX Network Programming,* Prentice-Hall, Englewood Cliffs, New Jersey.

SWINEHART, D., G. MCDANIEL, and D. R. BOGGS [December 1979], WFS: A Simple Shared File System for a Distributed Environment, *Proceedings of the Seventh Symposium on Operating System Principles,* 9-17.

TANENBAUM, A. [1981], *Computer Networks: Toward Distributed Processing Systems,* Prentice-Hall, Englewood Cliffs, New Jersey.

TICHY, W., and Z. RUAN [June 1984], Towards a Distributed File System, *Proceedings of Summer 84 USENIX Conference,* Salt Lake City, Utah, 87-97.

TOMLINSON. R. S. [1975], Selecting Sequence Numbers, *Proceedings ACM SIGOPS/SIGCOMM Interprocess Communication Workshop,* 11-23, 1975.

WARD, A. A. [1980], TRIX: A Network-Oriented Operating System, *Proceedings of COMPCON,* 344-349.

WATSON, R. [1981], Timer-Based Mechanisms in Reliable Transport Protocol Connection Management, *Computer Networks,* North-Holland Publishing Company.

WEINBERGER, P. J. [1985], The UNIX Eighth Edition Network File System, *Proceedings 1985 ACM Computer Science Conference,* 299-301.

WELCH, B., and J. OSTERHAUT [May 1986], Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System, *Proceedings IEEE Sixth International Conference on Distributed Computing Systems,* 1845-189.

WILKES, M. V., and D. J. WHEELER [May 1979], The Cambridge Digital Communication Ring, *Proceedings Local Area Computer Network Symposium.*

XEROX [1981], Internet Transport Protocols, *Report XSIS 028112,* Xerox Corporation, Office Products Division, Network Systems Administration Office, 3333 Coyote Hill Road, Palo Alto, California.

ZHANG, L. [August 1986], Why TCP Timers Don't Work Well, *Proceedings of ACM SIGCOMM '86.*