

国外计算机科学教材系列

用 TCP/IP 进行网际互联

第三卷：客户—服务器编程与应用

Linux/POSIX 套接字版

Internetworking With TCP/IP Vol III:

Client-Server Programming And Applications

Linux/POSIX Sockets Version

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

赵刚 林瑞 蒋慧 等译
谢希仁 审校



电子工业出版社

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

第三卷：客户 – 服务器编程与应用（Linux/POSIX 套接字版）

Internetworking With TCP/IP Vol III :
Client-Server Programming And Applications
Linux/POSIX Sockets Version

[美] DOUGLAS E. COMER 著
DAVID L. STEVENS

赵 刚 林 瑶 蒋 慧 等译

谢希仁 审校

11582 63

电子工业出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

本书是一部计算机网络经典性教科书。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。第三卷主要讨论应用软件如何使用TCP/IP，重点研究了客户-服务器范例，并考察了分布式程序中的客户和服务器，举例说明了各种设计，讨论了应用网关和隧道技术。第三卷共31章，各章之后附有很多很好的习题。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，也可供从事科研和技术开发的人员参考。

Authorized translation from the English language edition published by Prentice-Hall, Inc. Copyright © 2001.
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3卷，客户-服务器编程与应用：Linux/POSIX套接字版／(美)科默(Comer, D.E.)著；赵刚译。—北京：电子工业出版社，2001.4

书名原文：Internetworking With TCP/IP Volume 3 Client-Server Programming And Applications Linux/POSIX Sockets
国外计算机科学教材系列

ISBN 7-5053-6590-8

I.用... II.①科... ②赵... III.①计算机网络－通信协议－教材 ②服务器－操作系统(软件), Linux－程序设计 IV.TN915.04

中国版本图书馆CIP数据核字(2001)第16806号

丛 书 名：国外计算机科学教材系列

书 名：用TCP/IP进行网际互联第三卷：客户-服务器编程与应用(Linux/POSIX套接字版)

原 书 名：Internetworking With TCP/IP Vol III: Client-Server Programming And Applications Linux/POSIX Sockets Version

著 者：[美] DOUGLAS E. COMER DAVID L. STEVENS

译 者：赵 刚 林 瑶 蒋 慧 等

审 校 者：谢希仁

责 任 编辑：谭海平

排 版 制 作：今日电子公司制作部

印 刷 者：北京天竺颖华印刷厂

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：28.75 字数：718千字

版 次：2001年4月第1版 2001年4月第1次印刷

书 号：ISBN 7-5053-6590-8

TP · 3652

定 价：48.00元

版权贸易合同登记号 图字：01-2000-3473

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

关于作者

Douglas Comer 博士是 TCP/IP 协议和因特网的国际公认专家。自 20 世纪 70 年代末、80 年代初形成因特网以来，他就一直致力于因特网的研究工作，他也是负责指导因特网开发的因特网体系小组（IAB）的成员，还是 CSNET 技术委员会的主席和 CSNET 执行委员会的成员。

Comer 为一些公司提供网络设计和实现的咨询，还给全世界的技术和非技术人员开 TCP/IP 和互联网络的专业讲座。他的操作系统 Xinu 以及 TCP/IP 协议的实现在他的书中都有介绍，并且应用到了商业产品中。

Comer 是 Purdue 大学计算机科学系的教授，他主要教授计算机网络、互联网络和操作系统的课程，并进行相关的研究。除了撰写一系列畅销的技术书籍外，他还是《Software – Practice and Experience》杂志的北美地区编辑。Comer 是 ACM 会员（Fellow）。

其他的信息可查询以下网址：

www.cs.purdue.edu/people/comer

David Stevens 从 Purdue 大学的计算机科学系获得了硕士（1985）和博士（1993）学位。从 1983 年起，他就是 BSD UNIX 系统的编程人员，主要从事于 BSD UNIX 内核的开发。他实现了大部分 TCP/IP 协议族，并和 Comer 博士一起合作编写了几本计算机科学的教科书。他感兴趣的专业方向是操作系统、计算机网络和大规模软件系统设计。

近年来，Stevens 在可扩缩联网（scalable networking）领域进行高性能多处理器系统的研究，为 Sequent Computer Systems 公司和 IBM 公司工作。他是 ACM 和 IEEE 的成员。

前　　言

Douglas E. Comer博士的系列著作——《用TCP/IP进行网际互联》是一套令人瞩目的丛书。能向开放源码（open source）读者介绍该书的第三卷，对我来说实在是荣幸之至。

开放源码和TCP/IP的历史是紧密相连的：没有网络把你和合作者连接起来，就不能进行协作！而且，最早一批开放源码软件就有TCP/IP协议的实现。我记得在20世纪80年代初，“开放源码”还不像现在那样受媒体青睐，理解网络体系结构和实现的研究者屈指可数，而Douglas就是其中的佼佼者——他是一项广泛研究计划的负责人，全线出击，对当时遇到的很多问题提出挑战。

记得在20世纪90年代初，我们已经看到将技术应用到大工程领域的巨大趋势，这些领域渴望着知识和解决方案。那时，为公司构造基于互联网的环境，对工程师来说还是一个巨大的挑战。于是，Douglas便开始教导他们，让他们能够掌握下层网络的复杂性，给他们提供辛勤耕耘得来的经验教训。

21世纪来临了，新一代的设计者正在为因特网编写分布式应用程序。当前，我们听到许多激动人心的因特网应用，如napster、gnutella以及infrasearch。但奇怪的是，现在的开发人员很少有人牢固掌握网络工程原理——坦率地说，他们缺乏对基础的理解，这种缺乏不可避免地造成了应用程序的适应能力不强或者干脆就不能工作。

正因为如此，Doug与David L. Stevens合著的第三卷：客户-服务器编程与应用才与今天的因特网息息相关。这本书教给我们如何设计和构建客户-服务器应用程序，而且更重要的是，它还教给我们如何理解每种设计决策中所蕴涵的利弊得失。

我希望读者能够像业界前辈一样，从Comer博士的智慧中获益。

Marshall T. Rose

Theorist, Implementor, and Agent Provocateur

Petaluma, California

序 言

Linux 操作系统声名正旺，作为服务器系统，它对联网界尤其重要。这本使用 Linux 的新版第三卷是为那些渴望了解如何创建联网应用的程序员撰写的。大致说来，本书考察这样的问题，“应用软件如何使用 TCP/IP 协议通过因特网进行通信？”。本书重点研究了客户-服务器范例，并考察了在分布式程序中客户和服务器这两部分所用的算法。本卷举例说明了每种设计，并讨论了包括应用层网关和隧道等技术。另外，本卷还重温了几个标准应用协议，并用它们来说明一些算法和实现技术。

尽管本卷可以单独阅读和使用，但它实际是和另外两卷共同构成了一套丛书。丛书第一卷考虑的问题是：“什么是 TCP/IP 互联网？”；第二卷考察的问题是：“TCP/IP 软件是如何工作的？”，它给出了更多的细节，考察了工作代码，比第一卷探讨得更深入。因此，虽然程序员可以只通过第三卷学习创建网络应用，但学习其他各卷可以更好地理解下层技术。

第三卷的这个新版本包含了最新的技术，如，有一章解释了 Linux 程序如何利用 POSIX 线程设施创建并发服务器；关于 NFS 的章节讨论了 NFS 的第三版，这一版将为 Linux 界采用。此外，还有部分章节解释了 slirp 等程序所蕴涵的概念，这种程序能通过拨号电话连接访问因特网，而不要求每台计算机有一个惟一的 IP 地址。

还有两章显得特别及时，它们集中讨论了流式概念以及相关的技术，这些技术用于通过因特网发送音频和视频数据。第 28 章描述了实时协议（RTP）、编码、抖动缓存等基本概念。第 29 章展示了用于接收和播放 MP3 音频的 RTP 实现。

本书代码可在线获得。要通过万维网得到一个副本，可在以下网址的联网书籍清单中查找第三卷：

<http://www.cs.purdue.edu/homes/comer/netbooks.html>

要通过 FTP 访问代码，使用以下网址：

<ftp://ftp.cs.purdue.edu/pub/comer/TCPIP-vol3.linux.dist.tar.Z>

本书前几章介绍了客户-服务器模型，以及应用程序用于访问 TCP/IP 协议软件的套接字（socket）接口。此外，还描述了并发进程和用于创建进程的操作系统函数。随后的几章介绍了客户和服务器设计。

本书阐明了各式各样可能的设计并不是没有规则的。实际上，这些设计都遵循了一种模式。在考虑了并行性和传输的选择后，就可以理解这一观点。例如，有一章讨论了使用面向连接传输（如 TCP）的非并发服务器设计，而另一章讨论了相似的设计，但它使用无连接传输（如 UDP）。

我们描述了每个设计如何适应于各种可能的实现，但是，并没有试图开发一种客户-服务器交互的抽象“理论”。我们只是强调实用的设计原则，以及对程序员很重要的技术。每种技术在某些情况下都有其优点，并且每种技术都已用于正在工作的软件中。我们相信，理解这些设计之间的概念联系，将有助于读者理解每种方法的优缺点，并更容易在它们之间进行选择。

本书包含了多个例子程序，他们展现了各种设计实际上是如何进行的。大多数例子实现了标准

的TCP/IP应用协议。在每一种情况下，我们都试图选择一个应用协议，使它可表达一种设计思路而又不太难理解。因此，虽然很少有令人激动的例子程序，但这里的每一个例子都说明了一个重要的概念。在第三卷的这个版本中，所有的例子程序都使用Linux套接字机制（即套接字API）；本书还有两个其他版本，他们含有相同的例子，只不过使用了微软的Windows Sockets和AT&T的TLI接口。

后几章集中讨论中间件，讨论了远程过程调用的概念，并描述它是怎样被用于构造分布式程序的。这些章将远程过程调用技术与客户-服务器模型相联系，并说明如何使用软件从远程过程调用描述生成出客户和服务器程序。有关TELNET的章节展现了细枝末节如何在一个实际工作的程序中占据支配地位，以及即使是实现一个简单的、面向字符的协议，其代码如何会变得复杂。本部分最后两章是关于流式传输协议的。

本书很大部分的重点在并发处理。编写过并发程序的学生可能熟悉我们所描述的许多概念，因为这些概念适用于所有的并发程序，而不仅仅是网络应用。没编写过并发程序的学生可能会觉得这些概念很难。

本书适于作为向高年级学生教授“套接字编程”，或向低年级研究生介绍分布式计算的一个学期的课程。由于本书重点是如何使用互联网，而不是互联网是如何工作的，因此学生几乎不需要太多的网络背景知识就能理解这些内容。只要教师按合适的进度循序渐进，本科生课程中不会有特别的概念令人感到太难。介绍操作系统概念或并发编程实际经验的基础课程，可提供最佳背景材料。

学生只有亲手使用教材后，才会欣赏它。因此，任何课程都应安排编程实践，强迫学生将其想法运用到实际程序中。大学本科生可通过反复设计其他的应用协议来学习基本概念。研究生则应构建更为复杂的分布式程序，这些程序强调一些细微的技术（如第16章中的并发管理技术和第18章和第19章中的互连技术）。

在此要感谢许多人的帮助。Purdue大学因特网研究小组的成员们给本书原稿提供了许多技术信息和建议。Michael Evangelista校对了本书并编写了RTP代码。Gustavo Rodriguez-Rivera阅读了本书的许多章节，并做了很多实验测试细节，还编辑了附录1。Dennis Brylow对本书许多章节提出了建议。Christine Comer进行了修订并改进了行文和一致性。

Douglas E. Comer
David L. Stevens

目 录

第1章 引言和概述	1
1.1 使用 TCP/IP 的因特网应用	1
1.2 为分布式环境设计应用程序	1
1.3 标准和非标准的应用协议	1
1.4 使用标准应用协议的例子	1
1.5 telnet 连接的例子	2
1.6 使用 TELNET 访问其他服务	3
1.7 应用协议和软件的灵活性	4
1.8 从提供者的角度看服务	4
1.9 本教材的其余部分	5
1.10 小结	5
深入研究	5
习题	5
第2章 客户 – 服务器模型与软件设计	7
2.1 引言	7
2.2 动机	7
2.3 术语和概念	8
2.3.1 客户和服务器	8
2.3.2 特权和复杂性	8
2.3.3 标准和非标准客户软件	9
2.3.4 客户的参数化	9
2.3.5 无连接的和面向无连接的服务器	10
2.3.6 无状态和有状态服务器	10
2.3.7 无状态文件服务器的例子	11
2.3.8 有状态文件服务器的例子	11
2.3.9 标识客户	12
2.3.10 无状态是一个协议问题	13
2.3.11 充当客户的服务器	13
2.4 小结	14
深入研究	14
习题	15
第3章 客户 – 服务器软件中的并发处理	16
3.1 引言	16
3.2 网络中的并发	16

3.3 服务器中的并发	17
3.4 术语和概念	18
3.4.1 进程概念	18
3.4.2 局部和全局变量的共享	19
3.4.3 过程调用	20
3.5 一个创建并发进程的例子	20
3.5.1 一个顺序执行的 C 实例	20
3.5.2 程序的并发版本	21
3.5.3 时间分片	22
3.5.4 单线程的进程	23
3.5.5 使各进程分离	23
3.6 执行新的代码	24
3.7 上下文切换和协议软件设计	25
3.8 并发和异步 I/O	25
3.9 小结	25
深入研究	26
习题	26
第 4 章 协议的程序接口	27
4.1 引言	27
4.2 不精确指明的协议软件接口	27
4.2.1 优点与缺点	27
4.3 接口功能	28
4.4 概念性接口的规约	28
4.5 系统调用	28
4.6 网络通信的两种基本方法	29
4.7 LINUX 中提供的基本 I/O 功能	29
4.8 将 Linux I/O 用于 TCP/IP	30
4.9 小结	31
深入研究	31
习题	31
第 5 章 套接字 API	32
5.1 引言	32
5.2 Berkeley 套接字	32
5.3 指明一个协议接口	32
5.4 套接字的抽象	33
5.4.1 套接字描述符和文件描述符	33
5.4.2 针对套接字的系统数据结构	34
5.4.3 主动套接字或被动套接字	35
5.5 指明端点地址	35

5.6	类属地址结构	35
5.7	套接字 API 中的主要系统调用	36
5.7.1	socket 调用	37
5.7.2	connect 调用	37
5.7.3	send 调用	37
5.7.4	recv 调用	37
5.7.5	close 调用	38
5.7.6	bind 调用	38
5.7.7	listen 调用	38
5.7.8	accept 调用	38
5.7.9	在套接字中使用 read 和 write	38
5.7.10	套接字调用小结	39
5.8	用于整数转换的实用例程	39
5.9	在程序中使用套接字调用	40
5.10	套接字调用的参数所使用的符号常量	40
5.11	小结	41
	深入研究	41
	习题	41
第 6 章 客户软件设计中的算法和问题		43
6.1	引言	43
6.2	不是研究细节而是学习算法	43
6.3	客户体系结构	43
6.4	标识服务器的位置	44
6.5	分析地址参数	45
6.6	查找域名	45
6.7	由名字查找某个熟知端口	46
6.8	端口号和网络字节顺序	47
6.9	由名字查找协议	47
6.10	TCP 客户算法	48
6.11	分配套接字	48
6.12	选择本地协议端口号	48
6.13	选择本地 IP 地址中的一个基本问题	49
6.14	将 TCP 套接字连接到某个服务器	49
6.15	使用 TCP 与服务器通信	50
6.16	从 TCP 连接中读取响应	50
6.17	关闭 TCP 连接	51
6.17.1	对部分关闭的需要	51
6.17.2	部分关闭的操作	51
6.18	UDP 客户的编程	51
6.19	连接的和非连接的 UDP 套接字	52

6.20 对 UDP 使用 connect.....	52
6.21 使用 UDP 与服务器通信.....	52
6.22 关闭使用 UDP 的套接字.....	53
6.23 对 UDP 的部分关闭.....	53
6.24 关于 UDP 不可靠性的警告.....	53
6.25 小结.....	53
深入研究.....	54
习题.....	54
第 7 章 客户软件举例	55
7.1 引言	55
7.2 小例子的重要性	55
7.3 隐藏细节	55
7.4 针对客户程序的过程库例子	56
7.5 connectTCP 的实现	56
7.6 connectUDP 的实现	57
7.7 构成连接的过程	57
7.8 使用例子库	60
7.9 DAYTIME 服务	60
7.10 针对 DAYTIME 的 TCP 客户实现	61
7.11 从 TCP 连接中进行读	62
7.12 TIME 服务	63
7.13 访问 TIME 服务	63
7.14 精确时间和网络时延	63
7.15 针对 TIME 服务的 UDP 客户	64
7.16 ECHO 服务	65
7.17 针对 ECHO 服务的 TCP 客户	66
7.18 针对 ECHO 服务的 UDP 客户	67
7.19 小结	69
深入研究	70
习题	70
第 8 章 服务器软件设计的算法和问题	71
8.1 引言	71
8.2 概念性的服务器算法	71
8.3 并发服务器和循环服务器	71
8.4 面向连接的和无连接的访问	72
8.5 传输协议的语义	72
8.5.1 TCP 语义	72
8.5.2 UDP 语义	73
8.6 选择传输协议	73

8.7 面向连接的服务器	73
8.8 无连接的服务器	74
8.9 故障、可靠性和无状态	74
8.10 优化无状态服务器	75
8.11 四种基本类型的服务器	76
8.12 请求处理时间	77
8.13 循环服务器的算法	77
8.14 一种循环的、面向连接的服务器的算法	78
8.15 用 INADDR_ANY 绑定熟知端口	78
8.16 将套接字置于被动模式	78
8.17 接受连接并使用这些连接	79
8.18 循环的、无连接的服务器的算法	79
8.19 在无连接的服务器中构造应答	79
8.20 并发服务器的算法	80
8.21 主线程和从线程	80
8.22 并发的、无连接的服务器的算法	81
8.23 并发的、面向连接服务器的算法	81
8.24 服务器并发性的实现	82
8.25 把单独的程序作为从进程来使用	82
8.26 使用单线程获得表面上的并发性	83
8.27 各服务器类型所适用的场合	83
8.28 服务器类型小结	84
8.29 重要问题——服务器死锁	85
8.30 其他的实现方法	85
8.31 小结	85
深入研究	86
习题	86
 第 9 章 循环的、无连接服务器 (UDP)	87
9.1 引言	87
9.2 创建被动套接字	87
9.3 进程结构	90
9.4 TIME 服务器举例	91
9.5 小结	92
深入研究	92
习题	93
 第 10 章 循环的、面向连接的服务器 (TCP)	94
10.1 引言	94
10.2 分配被动的 TCP 套接字	94
10.3 用于 DAYTIME 服务的服务器	95

10.4	进程结构	95
10.5	DAYTIME 服务器举例	95
10.6	关闭连接	98
10.7	连接终止和服务器的脆弱性	98
10.8	小结	98
	深入研究	99
	习题	99
第 11 章	并发的、面向连接的服务器 (TCP)	100
11.1	引言	100
11.2	并发 ECHO	100
11.3	循环与并发实现的比较	100
11.4	进程结构	101
11.5	并发 ECHO 服务器举例	101
11.6	清除游离 (errant) 进程	104
11.7	小结	105
	深入研究	105
	习题	105
第 12 章	将线程用于并发 (TCP)	106
12.1	引言	106
12.2	Linux 线程概述	106
12.3	线程的优点	106
12.4	线程的缺点	107
12.5	描述符、延迟和退出	107
12.6	线程退出	108
12.7	线程协调和同步	108
12.7.1	互斥	108
12.7.2	信号量	108
12.7.3	条件变量	109
12.8	使用线程的服务器实例	109
12.9	监控	113
12.10	小结	113
	深入研究	113
	习题	114
第 13 章	单线程、并发服务器 (TCP)	115
13.1	引言	115
13.2	服务器中的数据驱动处理	115
13.3	用单线程进行数据驱动处理	116
13.4	单线程服务器的线程结构	116
13.5	单线程 ECHO 服务器举例	117

13.6 小结	119
深入研究	119
习题	120
第 14 章 多协议服务器 (TCP, UDP)	121
14.1 引言	121
14.2 减少服务器数量的动机	121
14.3 多协议服务器的设计	121
14.4 进程结构	122
14.5 多协议 DAYTIME 服务器的例子	122
14.6 共享代码的概念	125
14.7 并发多协议服务器	125
14.8 小结	126
深入研究	126
习题	126
第 15 章 多服务服务器 (TCP, UDP)	127
15.1 引言	127
15.2 合并服务器	127
15.3 无连接的、多服务服务器的设计	127
15.4 面向连接的、多服务服务器的设计	128
15.5 并发的、面向连接的、多服务服务器	129
15.6 单线程的、多服务服务器的实现	129
15.7 从多服务服务器调用单独的程序	130
15.8 多服务、多协议设计	131
15.9 多服务服务器的例子	131
15.10 静态的和动态的服务器配置	137
15.11 UNIX 超级服务器, inetd	138
15.12 inetd 服务器的例子	140
15.13 服务器的几种变形清单	141
15.14 小结	141
深入研究	142
习题	142
第 16 章 服务器并发性的统一、高效管理	143
16.1 引言	143
16.2 在循环设计和并发设计间选择	143
16.3 并发等级	143
16.4 需求驱动的并发	144
16.5 并发的代价	144
16.6 额外开销和时延	144
16.7 小时延可能出麻烦	145

16.8 从线程/进程预分配	146
16.8.1 Linux 中的预分配	146
16.8.2 面向连接服务器中的预分配	147
16.8.3 互斥、文件锁定和 accept 并发调用	147
16.8.4 无连接服务器中的预分配	148
16.8.5 预分配、突发通信量和 NFS	149
16.8.6 多处理器上的预分配	149
16.9 延迟的从线程/进程分配	149
16.10 两种技术统一的基础	150
16.11 技术的结合	150
16.12 小结	151
深入研究	151
习题	151
 第 17 章 客户进程中的并发	153
17.1 引言	153
17.2 并发的优点	153
17.3 运用控制的动机	153
17.4 与多个服务器的并发联系	154
17.5 实现并发客户	154
17.6 单线程实现	155
17.7 使用 ECHO 的并发客户例子	156
17.8 并发客户的执行	160
17.9 例子代码中的并发性	161
17.10 小结	161
习题	162
 第 18 章 运输层和应用层的隧道技术	163
18.1 引言	163
18.2 多协议环境	163
18.3 混合网络技术	164
18.4 动态电路分配	165
18.5 封装和隧道技术	166
18.6 通过 IP 互联网的隧道技术	166
18.7 客户和服务器之间的应用级隧道技术	166
18.8 隧道技术、封装和电话拨号线	167
18.9 小结	168
深入研究	168
习题	168
 第 19 章 应用级网关	169
19.1 引言	169

19.2 在受约束的环境中的客户和服务器.....	169
19.2.1 限制访问的现实	169
19.2.2 有限功能的计算机	169
19.2.3 安全性引起的连通性约束	169
19.3 使用应用网关	170
19.4 通过邮件网关互操作	171
19.5 邮件网关的实现	171
19.6 应用网关与隧道技术的比较	172
19.7 应用网关和有限因特网连接	173
19.8 为解决安全问题而使用的应用网关.....	174
19.9 应用网关和额外跳问题	174
19.10 应用网关举例	176
19.11 一个应用网关的实现	176
19.12 应用网关的代码	178
19.13 网关交换的例子	179
19.14 使用 rfed 和.forward 或 slocal 文件	179
19.15 通用的应用网关	180
19.16 SLIRP 的运行	180
19.17 SLIRP 如何处理连接	181
19.18 IP 寻址和 SLIRP	181
19.19 小结	182
深入研究	182
习题	183

第 20 章 外部数据表示 (XDR)	184
20.1 引言	184
20.2 数据表示	184
20.3 N 平方转换问题	185
20.4 网络标准字节顺序	185
20.5 外部数据表示的事实上的标准	186
20.6 XDR 数据类型	186
20.7 隐含类型	187
20.8 使用 XDR 的软件支持	187
20.9 XDR 库例程	188
20.10 一次一片地构造报文	188
20.11 XDR 库中的转换例程	189
20.12 XDR 流、I/O 和 TCP	190
20.13 记录、记录边界和数据报 I/O	190
20.14 小结	191
深入研究	191
习题	191

第 21 章 远程过程调用 (RPC) 的概念	193
21.1 引言	193
21.2 远程过程调用模型	193
21.3 构建分布式程序的两种模式	193
21.4 常规过程调用的概念性模型	194
21.5 过程模型的扩充	194
21.6 常规过程调用的执行和返回	195
21.7 分布式系统中的过程模型	196
21.8 客户 - 服务器和 RPC 之间的类比	196
21.9 作为程序的分布式计算	197
21.10 Sun Microsystems 的远程过程调用定义	197
21.11 远程程序和过程	198
21.12 减少参数的数量	198 ¹
21.13 标识远程程序和过程	198
21.14 适应远程程序的多个版本	200
21.15 远程程序中的互斥	200
21.16 通信语义	200
21.17 至少一次语义	201
21.18 RPC 重传	201
21.19 将远程程序映射到协议端口	201
21.20 动态端口映射	202
21.21 RPC 端口映射器算法	202
21.22 ONC RPC 的报文格式	203
21.23 对远程过程进行参数排序	205
21.24 鉴别	205
21.25 RPC 报文表示的例子	206
21.26 UNIX 鉴别字段的例子	206
21.27 小结	207
深入研究	208
习题	208
第 22 章 分布式程序的生成 (rpcgen 的概念)	209
22.1 引言	209
22.2 使用远程过程调用	209
22.3 支持 RPC 的编程工具	210
22.4 将程序划分成本地过程和远程过程	211
22.5 为 RPC 增加代码	211
22.6 stub 过程	212
22.7 多个远程过程和分派	212
22.8 客户端的 stub 过程的名字	213
22.9 使用 rpcgen 生成分布式程序	213

22.10 rpcgen 输出和接口过程	214
22.11 rpcgen 的输入和输出	215
22.12 使用 rpcgen 构建客户和服务器	215
22.13 小结	215
深入研究	216
习题	216
第 23 章 分布式程序的生成 (rpcgen 的例子)	218
23.1 引言	218
23.2 说明 rpcgen 的例子	218
23.3 查找字典	218
23.4 分布式程序的八个步骤	219
23.5 步骤 1：构建常规应用程序	220
23.6 步骤 2：将程序划分成两部分	224
23.7 步骤 3：创建 rpcgen 规约	229
23.8 步骤 4：运行 rpcgen	230
23.9 rpcgen 产生的.h 文件	230
23.10 rpcgen 产生的 XDR 转换文件	232
23.11 rpcgen 产生的客户代码	233
23.12 rpcgen 产生的服务器代码	235
23.13 步骤 5：编写 stub 接口过程	238
23.13.1 客户端接口例程	238
23.13.2 服务器端接口例程	240
23.14 步骤 6：编译和链接客户程序	241
23.15 步骤 7：编译和链接服务器程序	244
23.16 步骤 8：启动服务器和执行客户	246
23.17 使用 make 实用程序	247
23.18 小结	249
深入研究	249
习题	249
第 24 章 网络文件系统 (NFS) 的概念	251
24.1 引言	251
24.2 远程文件存取和传送	251
24.3 对远程文件的操作	252
24.4 异构计算机之间的文件存取	252
24.5 无状态服务器	252
24.6 NFS 和 UNIX 的文件语义	252
24.7 UNIX 文件系统的回顾	253
24.7.1 基本定义	253
24.7.2 无记录界限的字节序列	253

24.7.3 文件拥有者和组标识符	253
24.7.4 保护和存取	253
24.7.5 打开—读—写—关闭范例	254
24.7.6 数据传送	255
24.7.7 搜索目录权限	255
24.7.8 随机存取	255
24.7.9 搜索超过文件的结束	256
24.7.10 文件位置和并发存取	256
24.7.11 在并发存取时的“写 (write)”语义	257
24.7.12 文件名和路径	257
24.7.13 索引节点 (inode): 存储在文件中的信息	257
24.7.14 stat 操作	259
24.7.15 文件命名机制	259
24.7.16 文件系统 mount	260
24.7.17 UNIX 文件名解析	261
24.7.18 符号链接	262
24.8 NFS 下的文件	262
24.9 NFS 的文件类型	262
24.10 NFS 文件模式	263
24.11 NFS 文件属性	263
24.12 NFS 客户和服务器	264
24.13 NFS 客户操作	265
24.14 NFS 客户与 UNIX 系统	266
24.15 NFS 安装	266
24.16 文件句柄	267
24.17 句柄取代路径名	267
24.18 无状态服务器的文件定位	268
24.19 对目录的操作	269
24.20 无状态地读目录	269
24.21 NFS 服务器中的多个分层结构	269
24.22 安装 (mount) 协议	270
24.23 NFS 的传输协议	270
24.24 小结	270
深入研究	271
习题	271
第 25 章 网络文件系统协议 (NFS, Mount)	272
25.1 引言	272
25.2 用 RPC 定义协议	272
25.3 用数据结构和过程定义协议	272
25.4 NFS 常数、类型和数据声明	273

25.4.1 NFS 常数	273
25.4.2 NFS 的 <code>typedef</code> 声明	274
25.4.3 NFS 数据结构	274
25.5 NFS 过程	277
25.6 NFS 操作的语义	277
25.6.1 <code>NFSPROC3_NULL</code> (过程 0).....	277
25.6.2 <code>NFSPROC3_GETATTR</code> (过程 1).....	278
25.6.3 <code>NFSPROC3_SETATTR</code> (过程 2).....	278
25.6.4 <code>NFSPROC3_LOOKUP</code> (过程 3).....	278
25.6.5 <code>NFSPROC3_ACCESS</code> (过程 4).....	278
25.6.6 <code>NFSPROC3_READLINK</code> (过程 5).....	278
25.6.7 <code>NFSPROC3_READ</code> (过程 6).....	278
25.6.8 <code>NFSPROC3_WRITE</code> (过程 7).....	278
25.6.9 <code>NFSPROC3_CREATE</code> (过程 8).....	278
25.6.10 <code>NFSPROC3_MKDIR</code> (过程 9).....	278
25.6.11 <code>NFSPROC3_SYMLINK</code> (过程 10).....	279
25.6.12 <code>NFSPROC3_MKNOD</code> (过程 11).....	279
25.6.13 <code>NFSPROC3_REMOVE</code> (过程 12).....	279
25.6.14 <code>NFSPROC3_RMDIR</code> (过程 13).....	279
25.6.15 <code>NFSPROC3_RENAME</code> (过程 14).....	279
25.6.16 <code>NFSPROC3_LINK</code> (过程 15).....	279
25.6.17 <code>NFSPROC3_READDIR</code> (过程 16).....	279
25.6.18 <code>NFSPROC3_READDIRPLUS</code> (过程 17).....	280
25.6.19 <code>NFSPROC3_FSSTAT</code> (过程 18).....	280
25.6.20 <code>NFSPROC3_FSINO</code> (过程 19).....	280
25.6.21 <code>NFSPROC3_PATHCONF</code> (过程 20).....	280
25.6.22 <code>NFSPROC3_COMMIT</code> (过程 21).....	280
25.7 安装协议	280
25.7.1 安装协议的常数定义	280
25.7.2 安装协议的类型定义	281
25.7.3 安装数据结构	281
25.8 安装协议中的过程	282
25.9 安装操作的语义	282
25.9.1 <code>MOUNTPROC3_NULL</code> (过程 0).....	282
25.9.2 <code>MOUNTPROC3_MNT</code> (过程 1).....	282
25.9.3 <code>MOUNTPROC3_DUMP</code> (过程 2).....	283
25.9.4 <code>MOUNTPROC3_UMNT</code> (过程 3).....	283
25.9.5 <code>MOUNTPROC3_UMNTALL</code> (过程 4).....	283
25.9.6 <code>MOUNTPROC3_EXPORT</code> (过程 5).....	283
25.10 NFS 和安装鉴别	283

25.11 文件加锁	284
25.12 NFS 第 3 版与第 4 版之间的变化	284
25.13 小结	285
深入研究	285
习题	285
第 26 章 TELNET 客户 (程序结构)	287
26.1 引言	287
26.2 概述	287
26.2.1 用户终端	287
26.2.2 命令和控制信息	287
26.2.3 终端、窗口和文件	288
26.2.4 对并发性的需要	288
26.2.5 TELNET 客户的过程模型	288
26.3 TELNET 客户算法	289
26.4 Linux 中的终端 I/O	289
26.4.1 控制设备驱动器	291
26.5 建立终端模式	291
26.6 用于保存状态的全局变量	292
26.7 在退出之前恢复终端模式	293
26.8 客户挂起与恢复	294
26.9 有限状态机的规约	295
26.10 在 TELNET 数据流中嵌入命令	296
26.11 选项协商	296
26.12 请求 / 提供的对称性	297
26.13 TELNET 字符定义	297
26.14 针对来自服务器数据的有限状态机	298
26.15 在各种状态之间转移	299
26.16 有限状态机的实现	300
26.17 压缩的有限状态机表示	300
26.18 在运行时维持压缩表示	302
26.19 压缩表示的实现	302
26.20 构造有限状态机转移矩阵	304
26.21 套接字输出有限状态机	305
26.22 套接字输出有限状态机的相关定义	306
26.23 选项子协商有限状态机	307
26.24 选项子协商有限状态机的相关定义	308
26.25 有限状态机初始化	309
26.26 TELNET 客户的参数	310
26.27 TELNET 客户的核心	311
26.28 主有限状态机的实现	314

26.29 小结	315
深入研究	315
习题	316
第 27 章 TELNET 客户 (实现细节)	317
27.1 引言	317
27.2 有限状态机动作过程	317
27.3 记录选项请求的类型	317
27.4 完成空操作	318
27.5 对回显选项的 WILL/WONT 做出响应	318
27.6 对未被支持的选项的 WILL/WONT 做出响应	320
27.7 对 no go-ahead 选项的 WILL/WONT 做出响应	320
27.8 生成用于二进制传输的 DO/DONT	321
27.9 对未被支持的选项的 DO/DONT 做出响应	322
27.10 对传输二进制选项的 DO/DONT 做出响应	323
27.11 对终端类型选项的 DO/DONT 做出响应	324
27.12 选项子协商	326
27.13 发送终端类型信息	326
27.14 终止子协商	328
27.15 向服务器发送字符	328
27.16 显示在用户终端上出现的传入数据	329
27.17 使用 termcap 控制用户终端	332
27.18 将数据块写到服务器	333
27.19 与客户进程交互	334
27.20 对非法命令做出响应	335
27.21 脚本描述文件	335
27.22 脚本描述的实现	336
27.23 初始化脚本描述	336
27.24 收集脚本文件名的字符	337
27.25 打开脚本文件	338
27.26 终止脚本描述	339
27.27 打印状态信息	340
27.28 小结	341
深入研究	341
习题	342
第 28 章 流式音频和视频传输 (RTP 概念和设计)	343
28.1 引言	343
28.2 流式传输服务	343
28.3 实时交付	343
28.4 抖动的协议补偿	343

28.5 重传、丢失和恢复	344
28.6 实时传输协议	344
28.7 流的转换和混合	345
28.8 迟延回放和抖动缓存	346
28.9 RTP 控制协议 (RTCP)	346
28.10 多种流同步	347
28.11 RTP 传输和多对多传输	348
28.12 会话、流、协议端口和分用	349
28.13 编码的基本方法	350
28.14 RTP 软件的概念性组织	350
28.15 进程/线程结构	351
28.16 API 的语义	352
28.17 抖动缓存的设计和重新缓存	353
28.18 事件处理	354
28.19 回放异常及时间戳的复杂性	354
28.20 实时库例子的大小	354
28.21 MP3 播放器的例子	355
28.22 小结	355
深入研究	356
习题	356
第 29 章 流式音频和视频传输 (RTP 实现示例)	357
29.1 引言	357
29.2 集成实现	357
29.3 程序结构	357
29.4 RTP 定义	358
29.5 时间值的处理	361
29.6 RTP 序列空间的处理	362
29.7 RTP 分组队列的处理	363
29.8 RTP 输入处理	365
29.9 为 RTCP 保存统计信息	367
29.10 RTP 初始化	368
29.11 RTCP 的定义	372
29.12 接收 RTCP 发送方的报告	373
29.13 产生 RTCP 接收方的报告	374
29.14 创建 RTCP 的首部	376
29.15 RTCP 时延的计算	376
29.16 RTCP Bye (再见) 报文的产生	377
29.17 集成实现的大小	378
29.18 小结	378
深入研究	378

习题	379
第 30 章 Linux 服务器中的实用技巧和技术	380
30.1 引言	380
30.2 后台操作	380
30.3 编写在后台运行的服务器	381
30.4 打开描述符和继承	382
30.5 对服务器编程以关闭所继承的描述符	382
30.6 来自控制 TTY 的信号	382
30.7 对服务器编程以改变它的控制 TTY	382
30.8 转移到一个安全的和已知的目录	383
30.9 对服务器编程以改变目录	383
30.10 Linux umask	383
30.11 对服务器编程以设置其 umask	384
30.12 进程组	384
30.13 对服务器编程以设置其进程组	384
30.14 标准 I/O 描述符	384
30.15 对服务器编程以打开标准描述符	385
30.16 服务器互斥	385
30.17 对服务器编程以避免多个副本	385
30.18 记录服务器的进程 ID	386
30.19 对服务器编程以记录其进程 ID	386
30.20 等待一个子进程退出	386
30.21 对服务器编程以等待每个子进程退出	387
30.22 外来信号	387
30.23 对服务器编程以忽略外来信号	387
30.24 使用系统日志设施	387
30.24.1 产生日志报文	387
30.24.2 间接方式和标准差错的优点	388
30.24.3 I/O 重定向的限制	388
30.24.4 客户 - 服务器的解决方案	388
30.24.5 syslog 机制	389
30.24.6 syslog 的报文类	389
30.24.7 syslog 的设施	389
30.24.8 syslog 的优先级	389
30.24.9 使用 syslog	390
30.24.10 syslog 配置文件举例	390
30.25 小结	391
深入研究	392
习题	392

第 31 章 客户—服务器系统中的死锁和资源缺乏	393
31.1 引言	393
31.2 死锁的定义	393
31.3 死锁检测的难度	393
31.4 避免死锁	394
31.5 客户和服务器间的死锁	394
31.6 在单个交互中避免死锁	395
31.7 一组客户和一个服务器之间的资源缺乏	395
31.8 忙连接和资源缺乏	395
31.9 避免阻塞的操作	396
31.10 进程、连接和其他限制	396
31.11 客户和服务器的循环	397
31.12 用文档确认依赖性	397
31.13 小结	398
习题	398
附录 1 系统调用与套接字使用的库例程	400
附录 2 Linux 文件和套接字描述符的操作	422
参考文献	425

第1章 引言和概述

1.1 使用 TCP/IP 的因特网应用

TCP/IP 提供了将计算机互联起来的技术，通过 TCP/IP 互联网以获得多种可用的应用。有些应用很明显是使用 TCP/IP 的，而有些应用却不太明显。我们最熟悉的是与整个因特网和万维网(World Wide Web)有关的应用：浏览、聊天室，还有通常被称为万维网广播（ webcasting ）的流式处理（ streaming ）。公司以其他的方式来使用 TCP/IP 技术。例如，一家公司可使用 TCP/IP 监视和控制海上石油平台，而另一家公司可使用 TCP/IP 控制库存数量。一些连锁经营的宾馆在他们的预订系统中使用了 TCP/IP——每次预订信息通过一个专用的 TCP/IP 互联网进行通信。另外，许多大规模的网络使用 TCP/IP 应用来监视和控制网络设备。除此之外，新的应用正不断涌现。

事实上，互联网技术已经十分普及。目前全世界范围内的大多数公用和专用网络都选用了 TCP/IP 协议。在欧洲、印度、南美和太平洋周边的一些国家， TCP/IP 的使用正迅速增长。

1.2 为分布式环境设计应用程序

随着网络技术已日渐成为所有软件的一部分，程序员必须掌握这样的基础知识：设计和实现分布式应用程序所用到的原则和技术。我们将会看到，分布式计算的一个主要目标就是透明性——所产生的分布式程序的行为应尽可能与同样程序的非分布式版本一样。因此，分布式计算的目标就是提供一个环境，该环境隐藏了计算机和服务的地理位置，使它们看上去就像是在本地一样。

1.3 标准和非标准的应用协议

TCP/IP 协议族包含许多应用协议，而且新的应用协议每日都在出现。事实上，只要一个程序员设计了两个使用 TCP/IP 通信的程序，这个程序员就已经发明了一种新的应用协议。当然，有些应用协议已经被记录和标准化，并被采纳为正式的 TCP/IP 协议族的一部分。我们称这种协议为标准应用协议。应用程序员发明的其他供私人使用的协议，则称为非标准协议。

只要有可能，多数网络管理员都会选择使用标准的应用协议。当现成的协议足够用时，就不必再去发明一种新的应用协议。例如， TCP/IP 包含了像文件传送、远程登录和电子邮件等服务的标准应用协议。因此，对这些服务，程序员应使用标准协议。

1.4 使用标准应用协议的例子

尽管一个给定的远程登录会话仅仅以人可以键人的速率产生数据，并以人可以阅读的速率接收数据，但在已连接的因特网中，远程登录的通信量名列前十名之中。许多用户依赖远程登录作为他

们工作环境的一部分；他们不必直接连接到他们用于完成大部分计算的那些机器上。

TCP/IP 协议族包含一个用于远程登录的标准应用协议，称为 TELNET。TELNET 协议定义了应用程序为登录到远程机器而必须发送给该机器的数据的格式，以及远程机器发回的报文的格式。它定义了字符数据在传输时应如何被编码，以及为控制会话或放弃一个远程登录应发送的特殊报文。

TELNET 协议如何对数据进行编码的内部细节，对大多数用户是不相关的。一个用户可以调用接入远程机器的软件而不需要知道或关心它的实现。实际上，使用远程服务常常和使用本地服务一样容易。例如，运行 TCP/IP 协议的计算机系统往往含有一个命令，用户调用此命令来运行 TELNET 软件，在很多系统中，该命令名为 telnet。为了调用它，用户键入：

```
telnet machine
```

参数 machine 表示期望远程登录接入到的机器的域名。因此，为构成一个到机器 example.com 的 TELNET 连接，用户键入：

```
telnet example.com
```

从用户的观点看，运行 telnet 将连接远程机器并使用户窗口中的命令可在远程机器上执行。一旦建立了连接，telnet 应用就提供一个双向通信信道。只要用户一直打开窗口，telnet 应用程序就会把用户键入的每个字符都发送给远程机器，并把远程机器发出的每个字符显示在用户的显示器上。

通常在 telnet 用户连接到一个远程系统后，该远程系统会要求用户键入登录标识符和口令进行验证。这个提供给远程用户的提示信息与提供给本地用户的提示符信息是一样的。因此，TELNET 提供给每个远程用户一种正在使用直连终端的错觉。

1.5 telnet 连接的例子

作为一个例子，我们考虑一下当某位用户调用 telnet 并连接到机器 purdue.edu 时，会发生些什么：

```
telnet purdue.edu
Trying...
Connected to purdue.edu.
Escape character is '^]'.

SunOS 5.6

login:
```

当 telnet 程序将机器名转换为 IP 地址并试图与该地址建立一个有效的 TCP 连接时，就出现了最开始的输出消息，Trying…。一旦建立了连接，telnet 便打印出第二行和第三行，它们通知用户连接尝试已经成功并指出一个特殊字符，在需要时，用户可以键入该字符临时从 telnet 应用程序中退出（例如，如果发生了故障，用户就要放弃连接）。符号] 表示用户必须按住 CONTROL 键同时键入右括号键。

输出的最后几行来自远程机器。它们表明其操作系统是 SunOS 5.6 版，还提供了一个标准的登录提示符。光标停在 login: 消息后，等待用户键入一个合法的登录标识符。用户必须在远程机器上

有一个账户, TELNET会话才能继续下去。在用户键入一个合法的登录标识符后, 远程机器会提示用户输入口令。只有在登录标识符和口令都有效的情况下, 才允许用户接入。

1.6 使用 TELNET 访问其他服务

TCP/IP使用协议端口号来标识一台特定机器上的应用服务。实现某个特定服务的软件在某个预先确定的(熟知)协议端口上等待请求。例如, 用TELNET协议访问的远程登录服务, 它被分配的端口号是23。在默认情况下, 当用户调用telnet程序时, 程序会连接指定机器上的端口23。

有趣的是, TELNET协议可以用来访问不同于标准的远程登录服务的其他服务。为此, 用户必须指定所期望访问服务的协议端口号。当在命令行上调用telnet时, 大多数系统会提供可选的第二个参数, 以便允许用户指定其他的协议端口号; 一些telnet版本提供一个图形接口让用户从一个菜单项中选择端口号。不管哪种情况, 若用户没有提供第二个参数, telnet就使用端口23。但是, 若用户指明了一个协议端口号, telnet就连接该端口号。例如, 如果用户键入:

```
telnet purdue.edu 13
```

telnet程序将建立与机器purdue.edu上的协议端口号13的一个连接。

端口13对应的服务并不是常规的远程登录服务, 而是将提供daytime服务, 报告远程机器上的本地日期和时间。在telnet首次连接端口13时, 远程机器不会返回任何提示。但是, telnet程序会产生三行输出:

```
telnet purdue.edu 13
Trying...
Connected to purdue.edu.
Escape character is '^]'.
```

而远程机器上的服务只产生了一行含日期和时间的输出。在它产生输出后, 远程机器将关闭连接。

```
Sun Jan 16 21:52:08 2000
Connection closed by foreign host.
```

Finger服务可从端口79上获得, 它是一种使用TELNET协议访问的交互式服务。Finger服务提供有关已登录到机器中的所有用户信息或某个指定用户的信息。输入可以是一个空行(提示要列出计算机上的所有用户)加回车符, 或一个用户名(提示只要提供指定用户的信息)加一个回车符。

虽然finger服务是交互式的, 但它不会自动产生提示信息, 而是等待用户输入一个请求再加以响应。例如, 为在机器purdue.edu上的端口79处对John Rice执行finger操作, 可在调用telnet后, 等待连接, 然后输入用户名。服务会提供该用户的信息加以响应。下面所示的输出说明了这一交互过程:

```
telnet purdue.edu 79
Trying ...
Connected to purdue.edu.
Escape character is '^]'.
John Rice
```

Output of your query: John Rice

Name	Dept/School	Phone	Status
Email			
John Richard Ricd	Computer Science	+1 765 49-46007	staff
jrr@cs.purdue.edu			

1.7 应用协议和软件的灵活性

前面的例子说明了如何用单个软件（在此例中是 telnet）访问多个服务。TELNET 协议的设计以及用它来访问 daytime 和 finger 服务说明了两个重要问题。其一，所有协议设计的目标都是寻找一个适用于多种应用的基本抽象。在实际应用中，由于 TELNET 提供了一种基本的交互通信手段，它适用于多种多样的服务。从概念上说，用来访问一种服务的协议和服务本身是保持分离的。其二，当设计师们考虑实现一些应用服务时，会尽可能地使用标准的应用协议。前面讲的 finger 服务就因为它使用了标准的 TELNET 协议进行通信，所以访问该服务很容易。而且，由于多数 TCP/IP 软件都包括一个可供调用的应用程序来执行 TELNET，所以不需要其他的客户软件来访问其他服务。设计人员在发明新的交互应用时，如果选用 TELNET 用作其访问协议，就可以重新使用这些软件。概括地说就是：

TELNET 协议提供了最大程度的灵活性，因为它只定义了交互通信而没有定义所访问服务的细节。除了远程登录以外，TELNET 还可以被许多交互服务用作通信机制。

1.8 从提供者的角度看服务

前面所给出的应用服务的例子说明了如何为单个用户提供服务。用户运行一个访问远程服务的程序时，期望只经过短时延或甚至没有时延就收到应答。

从提供服务的计算机的角度看，情况就大为不同了。在各个不同网点的用户可能在同一时间访问某个特定的服务。当他们这样做时，每个用户都期望能没有时延地收到响应。

为提供快速的响应并处理多个请求，提供应用服务的计算机系统必须使用并发处理。即服务提供者不能在它正处理前一用户的请求时，让新来的用户等待，这就必须使软件在同一时间处理多个请求。

并发应用程序的运行可能看上去有些不可思议。单个应用程序似乎在同一时刻处理多种活动。对 TELNET 来说，提供远程登录服务的程序必须允许多个用户登录到某台机器，而且必须能管理多个活动的登录会话。某一个登录会话所进行的通信一定不能影响其他的会话。类似地，实现 finger 服务的程序必须允许多个用户在同一时刻请求服务，而且相互之间互不影响。

这种对并发的需求使网络软件的设计、实现和维护复杂化了。它需要一些专用算法和编程技术。此外，由于并发使调试变得复杂了，程序员就必须很小心地为其设计进行整理归档，并遵循良好的编程习惯。最后，程序员必须谨慎管理并发——他们必须选用一种能提供最高吞吐能力的并发等级，因为太大或太小的并发等级都会使软件性能下降。

1.9 本教材的其余部分

本书有助于应用编程人员理解、构造和优化网络应用软件。它描述了一些应用协议的顺序和并发实现中的基本算法，并提供了每种实现细节的例子。虽然这些例子都使用了TCP/IP协议，但它所重点讨论的原理、算法以及一般性的技术是可以为大多数网络协议使用的。书中还介绍了各种技术的优缺点，说明了并发在服务器设计中的重要作用。后一部分章节讨论了并发管理的细节问题，并回顾了一些允许程序员自动优化吞吐能力的技术，概括地说：

提供对应用服务的并发访问是很重要的，但也是困难的；本书的许多章节解释并讨论了应用协议软件的并发实现。

本书的每一章中都介绍了一个应用的设计。前面的章节介绍客户-服务器模型、无连接和面向连接传输以及套接字应用编程接口（API）。后面的章节介绍客户和服务器软件中使用的算法和实现技术，以及管理并发所需的一些算法和技术组合。

除了描述客户和服务器软件的算法之外，本书还介绍了诸如隧道技术、应用级网关和远程过程调用等一般性技术。最后，本书还详细介绍了几个标准的应用协议，如NFS和TELNET，还探讨了一些支持协议，如RTP。

大多数章节都包含了一些软件实例，他们有助于说明所讨论的原理。这些软件应被看作本书的一部分。它们清楚地说明了工作程序中的细节是如何组织的，以及程序是如何体现这些概念的。

1.10 小结

许多程序员在构建分布式的应用程序时使用TCP/IP作为传输机制。在程序员设计和实现这些软件前，必须先理解计算的客户-服务器模型、传输协议使用的语义、应用程序用于访问协议软件的操作系统接口、实现客户和服务器软件的基本算法，以及诸如应用网关的其他技术。

多数网络服务允许多个用户同时对其进行访问。为使多个用户能同时访问服务，服务器软件必须是并发的。本书大部分内容的重点是应用协议的并发实现技术和对并发的管理问题。

深入研究

厂家在其操作系统所提供的手册中含有一些命令调用的信息，调用这些命令可以访问TELNET之类的服务。许多网络站点用本地定义的命令扩充标准的命令集，请与你的本地站点管理员一起找找这些本地可使用的命令。

习题

- 1.1 使用TELNET从你的本地机器登录到其他机器上。有多大时延？如果有，当第二台机器连接到同一个局域网时情况又如何？当连接到一台远程机器时，你注意到有多大的时延？
- 1.2 阅读一下厂家的手册，看看TELNET软件的本地版本是否允许连接远程机器的某个端口，而这个端口不是标准的远程登录端口。

- 1.3 确定一下本地计算机上可用的 TCP/IP 服务集。
- 1.4 使用 FTP 程序从远程站点读取一个文件。如果该软件不提供统计功能，请估计一下对一个大文件的传输速率。该速率较你期望的是大了还是小了？
- 1.5 使用 finger 命令获取某个远程站点上的用户信息。

第2章 客户-服务器模型与软件设计

2.1 引言

从应用的观点看，就像大多数计算机通信协议一样，TCP/IP仅仅提供传输数据的基本机制。具体地说，TCP/IP允许程序员在两个应用程序之间建立通信并来回传递数据。因此，我们说，TCP/IP提供了对等（peer-to-peer）或端到端（end-to-end）通信。这些对等应用程序可在同一机器上执行，也可在不同的机器上执行。

尽管TCP/IP规定了数据如何在一对正在进行通信的应用程序间传递，但它并没有规定对等的应用程序在什么时间以及为什么要进行交互，也没有规定程序员在一个分布式环境下应如何组织这样的应用程序。在TCP/IP的使用中，客户-服务器的组织方法占有主导地位，几乎所有的应用都使用了客户-服务器范例这种模型^①。实际上在对等网络系统中，客户-服务器的交互是相当基本的形式，大多数计算机通信都采用了此交互形式。

本书使用客户-服务器范例描述所有应用程序的编写。它考虑了客户-服务器模型背后所蕴涵的动机，描述了客户和服务器的功能，还说明了如何构造客户和服务器软件。

在考虑如何构造软件之前，首先要定义客户-服务器的概念和术语。下面几节定义了全书都要使用的术语。

2.2 动机

客户-服务器范例的基本动机来自会聚点（rendezvous）问题。为理解这一问题，设想一个人试图在两台独立的机器上启动两个程序并让它们进行通信。我们知道计算机的运行速度要比人快许多数量级。在这个人启动第一个程序后，该程序开始执行并向其对等程序发送报文。在几微秒内，它就判断出对等程序还不存在，于是就发出一条错误消息，然后退出。在这时，这个人启动了第二个程序。遗憾的是，当第二个程序开始执行时，它发现对等程序已经终止执行了。即便是两个程序继续尝试通信，但由于每个程序都执行得相当快，在同一时刻双方相互发送消息的概率还是很低的。

客户-服务器模型用一种直接的方式解决此会聚点问题：它要求在任何一对进行通信的应用进程中，有一方必须在启动执行后（无限期地）等待对方与其联系。这种解决方案减少了下层协议软件的复杂性，因为下层协议不必自己对收到的通信请求做出响应。问题在于：

由于客户-服务器模型负责处理应用的会聚点问题，因此不需要TCP/IP在报文到达后提供自动创建运行程序的任何机制。但是要求有一个程序在任何请求到来前就一直运行，等待其他程序与之通信。

^① 尽管近来的营销文献将客户-服务器计算称为应用-服务器范例，但我们将使用最初的术语。

为确保计算机已准备好进行通信，多数系统管理员都安排通信程序在操作系统引导时就自动启动。每个程序启动后就一直运行，等待下一个服务请求的到来并为其提供服务。

2.3 术语和概念

根据应用程序是等待通信的一方还是发起通信的一方，客户—服务器范例将通信应用分为两大类。本节将对这两类应用给出一个简明而全面的定义，在后面的章节中还会对其加以说明并解释其中的许多细节问题。

2.3.1 客户和服务器

客户—服务器范例根据通信发起的方向对程序进行分类，即区别一个程序是客户还是服务器。一般来说，发起对等通信的应用程序称为客户。终端用户往往在其使用网络服务时调用客户软件（例如万维网浏览器就是一个客户程序）。多数客户软件与常规的应用程序实现是一样的。客户应用程序每次执行时都要与服务器联系，发出请求并等待响应。客户收到响应后再继续处理。客户通常比服务器容易构建，它的运行往往并不需要系统特权。

与之相比，服务器是等待接收客户通信请求的一种程序^①。服务器接收一个客户的请求，执行必要的计算，然后将结果返回给客户。

2.3.2 特权和复杂性

为完成计算和返回结果，服务器软件经常要访问受操作系统保护的对象（如文件、数据库、设备或协议端口）。因此，服务器软件的执行通常带有一些系统特权。由于服务器在执行时带有特权，应注意不要将特权传递给使用服务的客户。例如，一个具有执行特权的文件服务器，必须仔细检查某个文件能否被某个客户访问。服务器不能依赖那些常规的操作系统检查，因为服务器的特权允许它访问任何文件。

通常，服务器含有处理以下安全问题的代码：

- 鉴别——验证客户的身份
- 授权——判断某个客户是否被允许访问服务器所提供的服务
- 数据安全——确保数据不被无意泄露或损坏
- 保密——防止未经授权访问信息
- 保护——确保网络应用程序不能滥用系统资源

在以后几章中我们会看到，服务器要执行高强度计算或处理大量数据，如果它能并发地处理请求，其运行效率会更高。这种特权和并发操作的结合使服务器的设计与实现较客户的要更加困难。后面的几章将提供许多例子，它们说明了客户和服务器的区别。

^① 从技术上讲，服务器是一个程序而不是一块硬件。然而计算机用户经常（错误地）将这一术语用于表示负责运行某个特定服务器程序的计算机。例如，他们可能说：“那台计算机是我们的文件服务器”，这时他们实际上指的是：“那台计算机上运行着我们的文件服务器程序”。

2.3.3 标准和非标准客户软件

第1章中描述了两大类客户应用程序：一类客户调用标准TCP/IP服务（如电子邮件）；另一类客户调用网点定义的服务（例如，某家公司的私有数据库系统）。标准应用服务包括TCP/IP定义的服务，这些服务都被指派了熟知的、普遍都能识别的协议端口标识符。我们将所有其他的应用服务称为本地定义的应用服务（locally-defined application service）或非标准的应用服务（nonstandard application service）。

标准服务与其他服务的区别只有在与外界环境通信时才变得明显。在某个特定环境下，系统管理员往往使得那些定义的服务名让用户不能区分服务是本地的还是标准的。但程序员在构建网络应用时却应记住这个区别，不要让程序依赖那些只在本地才能用的服务，否则那些应用就不能被其他网点使用了。

尽管TCP/IP定义了许多标准的应用协议，但多数商业计算机厂家在他们的TCP/IP软件中只提供了少部分标准应用的客户程序。例如，TCP/IP软件中通常包括使用标准TELNET协议的远程登录客户、使用标准SMTP协议或POP协议传输和接收邮件的电子邮件客户、使用标准FTP协议在两台机器之间传送文件的文件传送客户，以及使用标准HTTP协议访问万维网文档的万维网浏览器。

当然，许多组织为自己构建了特殊的应用程序，这些程序使用TCP/IP进行通信。这些定制的、非标准的应用程序有简单的，也有复杂的，它们涉及各种各样的服务，如音乐和视频传输、语音通信、远程实时数据采集、在线预订系统、分布式数据库访问、气象数据发布以及设备或机器的远程控制等。

2.3.4 客户的参数化

有一些客户软件提供了较高的通用性。具体地说，有些客户软件允许用户既指定运行服务器程序的远程机器，又指定服务器监听的协议端口号。例如，在第1章中我们已说明过，标准应用客户软件如何使用TELNET协议访问不同于常规TELNET远程终端服务的其他服务，前提是程序应允许用户指定目的协议端口以及远程机器。

从概念上说，允许用户指明协议端口号的软件比其他软件多一些输入参数，因此我们用一个术语——全参数化的客户（fully parameterized client）来描述这种软件。例如，一个全参数化的客户允许用户指定一个协议端口号。

并非所有的厂商为其客户应用软件提供全参数化。因此，在一些系统中，如果要TELNET客户使用一个非公认的远程登录端口是很困难的，甚至是不可能的。实际上，可能需要修改厂家的TELNET客户软件或者编写一个新的客户软件，使它接受一个端口参数并使用该端口。当然，在构建客户软件时，最好使用全参数化：

在设计客户应用软件时，最好让它包含一些允许用户全部指明目的机器和目的协议端口号的参数。

全参数化在测试新的客户或服务器时特别有用，因为它可以使测试过程独立于目前使用的软件。例如，某个程序员可以构建一对TELNET客户和服务器，并使用非标准的协议端口调用它们，这样在对该软件进行测试时不会打扰标准服务。在测试过程中，其他用户可以继续访问原来的TELNET服务而不受影响。

2.3.5 无连接的和面向无连接的服务器

在程序员设计客户—服务器软件时，必须在两种类型的交互中做出选择：无连接的风格或面向连接的风格。这两种风格的交互直接对应于 TCP/IP 协议族所提供的两个主要的传输协议。如果客户和服务器使用用户数据报（UDP）进行通信，那么交互就是无连接的；如果使用传输控制协议（TCP），则交互就是面向连接的。

从应用程序员的角度看，无连接的交互和面向连接的交互之间的区别是非常重要的，因为这在很大程度上决定了客户和服务器交互所采用的算法。如果使用 TCP/IP 协议通信，连接方式的选择还决定了下层系统所提供的可靠性等级。由于 TCP 考虑到了所有传输问题，它提供了完全可靠性。TCP 验证数据的到达，对未到达的报文段会自动进行重传。它还计算数据的校验和，以保证数据在传输过程中没有损坏。它使用序号确保数据按序到达并自动忽略重复的分组。它提供了流控制以确保发送方发送数据的速度不超过接收方的承受能力。最后，在下层网络因某种原因无法运作时，TCP 会通知发送方。

相反，使用 UDP 的客户和服务器在传输可靠性上没有任何保证。在客户发送请求时，这个请求可能丢失、重复、延迟或者交付失序。类似地，服务器发回给客户的响应也可能丢失、重复、延迟或交付失序。因此，使用 UDP 的客户或服务器软件中必须包含检测和纠正这些错误的代码。

由于 UDP 提供了最大努力交付（best effort delivery），有时程序员可能会被它蒙蔽。UDP 本身不会引入差错——它只是依靠下层的网际协议（IP）来交付分组。而 IP 则要依赖于下层的硬件网络和中间的一些网关。从程序员的角度来看，如果下层的互联网工作得好，UDP 也就工作得好。例如，在本地环境中可靠性差错很少发生，因此在一个本地环境中 UDP 工作得就好。通常差错只在一个广域互联网中通信时才会发生。

程序员有时会犯这样的错误，即选择了无连接的传输（例如 UDP）来构建应用程序，但只在一个局域网中测试这个应用软件。因为在局域网中分组很少甚至从不被迟延、丢失或者交付失序，应用软件好像工作得很好。然而，如果同样的软件要放在一个广域互联网中使用，就可能失败或产生不正确的结果。

初学者以及大多数有经验的专业人员喜欢使用面向连接风格的交互。面向连接的协议使编程更简单，程序员不必再负责检测和纠正差错。实际上，在 UDP 中加入可靠性不容易，它要求具有相当的协议设计经验。

通常，应用程序只在以下情况下使用 UDP：（1）应用协议指明必须使用 UDP（可假定应用协议已设计了处理可靠性和交付差错的内容），（2）应用程序协议要依靠硬件进行广播或组播，或者（3）应用协议在可靠的本地环境中运行，不需要额外的可靠性处理。我们可以概括如下：

在设计客户—服务器应用时，强烈建议初学者使用 TCP，因为 TCP 提供了可靠的、面向连接的通信。程序仅在以下情况使用 UDP：如果由应用协议处理可靠性，或应用协议需要用硬件进行广播或组播，或不需要额外的可靠性处理。

2.3.6 无状态和有状态服务器

服务器所维护的与客户交互活动的信息称为状态信息。不保存任何状态信息的服务器称为无状态服务器（stateless server），反之则称为有状态服务器（stateful server）。

获得高效率的愿望促使设计者在服务器中保存状态信息。在服务器中保存少量信息，就可减少客户和服务器间交换报文的大小，还能允许服务器快速地响应请求。从本质上讲，状态信息让服务

器记住了客户以前有过哪些请求，并在每个新请求到来时计算新响应。相反，采用无状态服务器的动机是协议的可靠性：如果报文丢失、重复或交付失序，或者如果客户计算机崩溃或重启动，则一个服务器中的状态信息就会变得不正确。在服务器计算响应时，若使用了不正确的状态信息，就可能产生不正确的响应。

2.3.7 无状态文件服务器的例子

举个例子有助于说明无状态和有状态服务器之间的区别。考虑一个文件服务器，它允许客户远程访问保存在本地磁盘中的文件信息。服务器作为一个应用程序运作，等待网络中的某个客户与它联系。客户发送的请求有两种类型：一是请求从某个指定文件中获取数据，或者请求在指定文件中存储数据。服务器执行所请求的操作并向客户发回响应。

如果文件服务器是无状态的，它不维护有关事务处理的信息。客户发出的每个报文都必须指定完整的信息。报文必须指定操作类型（是读文件还是写文件）、文件名、传输数据在文件中的位置和传输字节数。如果报文的操作类型是写数据，报文还应包含要写入文件的数据。图2.1是无状态请求报文中包含的字段。

项目	描述
op	操作（读或写）
name	文件名
pos	在文件中的位置
size	要传输的字节数
data	（只出现在写请求中）

图2.1 客户发给无状态文件服务器的报文中所需字段。服务器独立地解释每个报文

无状态请求中所用的文件名必须完整，不能有二义性——服务器必须能单独从报文中标识该文件。因此，文件名可能会很长。

2.3.8 有状态文件服务器的例子

另一方面，考虑文件服务器维护状态信息时所用的报文内容。由于服务器能区分各个客户，并保留着各个客户以前的请求，请求报文中就不必包含所有字段信息。具体来说，在某个客户开始访问一个文件后，服务器中保留着被访问的文件名、当前位置和客户以前的操作。在客户发出第一个读文件的请求之后，后续的读请求报文只需包含一个字段：读取字节数。类似地，在客户开始写数据后，后续的每个写请求报文只需包含两个字段：写入数据的字节数和写入数据本身。

一个有状态文件服务器如何管理状态呢？服务器维护着一张表，该表中含有每个客户的信息和当前正被访问文件的信息。图2.2所示的是一张可能出现的状态信息表。

在服务器为某个客户完成一次操作后，服务器会相应增加状态表中的文件位置，从而使该客户的下一个请求指向新数据。

客户	文件名	当前位置	上一次操作
1	test.program.c	0	read
2	tcp.book.text	456	read
3	dept.budget.text	38	write
4	tetris.exe	128	read

图 2.2 有状态文件服务器的状态信息表的例子。客户可能在报文中省略服务器中已有信息的字段

2.3.9 标识客户

有状态服务器标识客户有两种基本方法：端点和句柄。使用端点标识符的优点是好像能自动进行标识，因为此机制只与下层传输协议有关，不依赖于上层应用协议。为使用端点标识符，服务器要请下层传输协议软件在请求到达时提供识别客户的信息（如客户的 IP 地址和协议端口号）。服务器然后使用端点信息查找状态表进行定位。

但是端点信息有可能变化。例如，如果一个网络故障会使客户打开一个新的 TCP 连接，这时服务器不会将该新连接与此客户以前的状态信息关联起来。而另一种标识客户的句柄方法却能将多个连接与一个固定客户联系起来，这是句柄方法的一个优点。但是这种方法对应用而言也有一个缺点。一个句柄只在一个客户和一个服务器间使用。在客户发出第一个请求时，客户必须提供完整的信息。服务器收到请求后，将在状态表中增加一个条目，并为此条目生成一个称为句柄的短标识符（通常是一个小整数）。服务器将此句柄发回给客户以便在后续的请求中使用。当该客户发送新请求时，它可以在请求报文中使用此句柄代替长的文件名。值得注意的是，由于此机制与下层传输协议无关，传输连接的变化不会使句柄无效。

有状态服务器不可能永久维持状态。当一个客户开始访问时，服务器为该客户分配一小部分本地资源（如存储器）。如果服务器从不释放这些被占用的资源，服务器最终会耗尽一个或多个资源。因此，有状态的应用协议需要终止。在文件服务器的例子中，客户使用完某个文件后，必须发送一个报文通知服务器它不再用该文件了。在响应该请求时，服务器会删除存储的相应状态信息，然后使该表项可被其他客户使用。

使用状态的要点是效率。只要客户和服务器间的所有报文的传送都是可靠的，这种有状态的设计就会使传输的数据量减少。而且，由于大多数非分布式的程序使用状态，程序员采用有状态的设计是很自然的。这里的要点是：

在理想的情况下，只要网络能可靠地交付所有的报文，并且计算机从不崩溃，则在这种情况下，使服务器为每个进行着的交互保持少量状态信息，就可以使交互的报文小些，并且使分布式应用更像非分布式的应用。

尽管状态信息可以提高效率，但如果下层网络使报文重复、延时或者交付失序（例如，如果客户和服务器在因特网中使用 UDP 进行通信），状态信息就很难甚至不可能被正确地维护。考虑以下这样的情况，在我们所举的文件服务器的例子中，如果网络重复了一个 read 请求，那么会发生什么呢？回想一下，服务器在其状态信息中保持了一个文件位置项。假设客户每次从文件中获取数据后，服务器便更新该文件的位置信息。如果网络重复了一个 read 请求，服务器就会收到两份同样的 read 请求。在第一个 read 请求到来时，服务器从文件中读取数据，然后更新状态信息中的文件位置，并将结果返回给客户。当第二个重复的 read 请求到来时，服务器将读取另外一段数据，然后再一次更

新文件位置，并将新数据返回给客户。客户可能将第二个响应看作是重复的并将其丢弃，或者可能报告出了一个差错，因为它只发出一个请求却收到了两个不同的响应。无论哪种情况，在服务器中的状态信息都会变得不正确，因为它与客户的真实状态不一致。

当计算机重启动时，状态信息也可能变得不正确。如果一个客户与服务器联系后机器就崩溃了，而服务器已经为该客户建立了状态信息。服务器就可能永远不会收到让它丢弃这些状态信息的报文。最终，这些积累起来的状态信息会把服务器的存储器资源耗尽。在我们的文件服务器的例子中，如果某个客户打开了100个文件而后崩溃了，那么服务器将永远在状态表中保持这100个没用的条目。

有状态服务器如果使用端点标识符，机器崩溃或重启动也同样会使服务器产生混乱（或响应不正确）。一个新的客户程序在重启动后开始工作，而它与先前那个在系统崩溃前工作的客户程序正好使用相同的协议端口号，那么服务器不可能正确区分这两个客户。这个问题可能看上去很容易解决，只要在一个客户要求进行交互的新请求到达时，让服务器擦除以前来自某个客户的信息就可以了。然而要记住，下层的互联网可能会重复或延迟报文，因此，解决重启动后新客户重新使用协议端口问题的方案应当也能处理这种情况，即某个客户正常启动后，它发给服务器的第一个报文被重复了，并且其中一个报文副本被延迟了。

一般来说，保持正确状态信息这个问题只有用复杂的协议才能解决，这种协议能解决不可靠的交付和计算机系统重启动的问题。概括地说：

在实际的互联网中，机器可能崩溃或重启动，而报文可能丢失、重复或交付失序。采用有状态的设计会导致复杂的应用协议，而这种应用协议难于设计、理解和正确实现。

2.3.10 无状态是一个协议问题

尽管我们已经在服务器的环境中讨论了无状态的问题，但一个服务器到底是无状态的还是有状态的呢？这一问题的答案更多地取决于应用协议而不是实现。如果应用协议规定了某个报文的含义在某种方式上依赖于先前的一些报文，这样它就不可能提供无状态的交互。

从本质上说，无状态问题的重点是应用协议是否承担着可靠交付的责任。要避免出问题并使交互可靠，应用协议的设计者必须确保每个报文决无二义性。也就是说，一个报文既不能依赖于被按序交付，也不能依赖于前面的报文已被交付。关键是协议设计者必须这样构建交互，即无论一个请求何时到达或到达多少次，服务器都应给出同样的响应。数学家们用术语幂等（idempotent）指一个总是产生相同结果的数学运算。我们用这个术语来称呼这种协议，它让服务器对某个已知报文给出相同的响应，而不管该报文到达多少次。

如果一个互联网中的下层网络可能使报文重复、延迟或不按序交付，或者运行客户应用程序的计算机可能会意外崩溃，那么在这样的网络中服务器应是无状态的。只有当应用协议被设计成对操作是幂等的，服务器才能是无状态的。

2.3.11 充当客户的服务器

许多程序并不准确符合客户或服务器的定义。在服务器计算某个请求的响应时，它可能需要访问其他网络服务。因此服务器也可能充当客户。例如，假设我们的文件服务器程序需要获得当时的时间，以便在文件中打上访问时间的标记。我们还假设运行服务器的系统中没有时钟。为了获得时间信息，该服务器就作为客户向时间服务器发出请求，如图2.3所示。

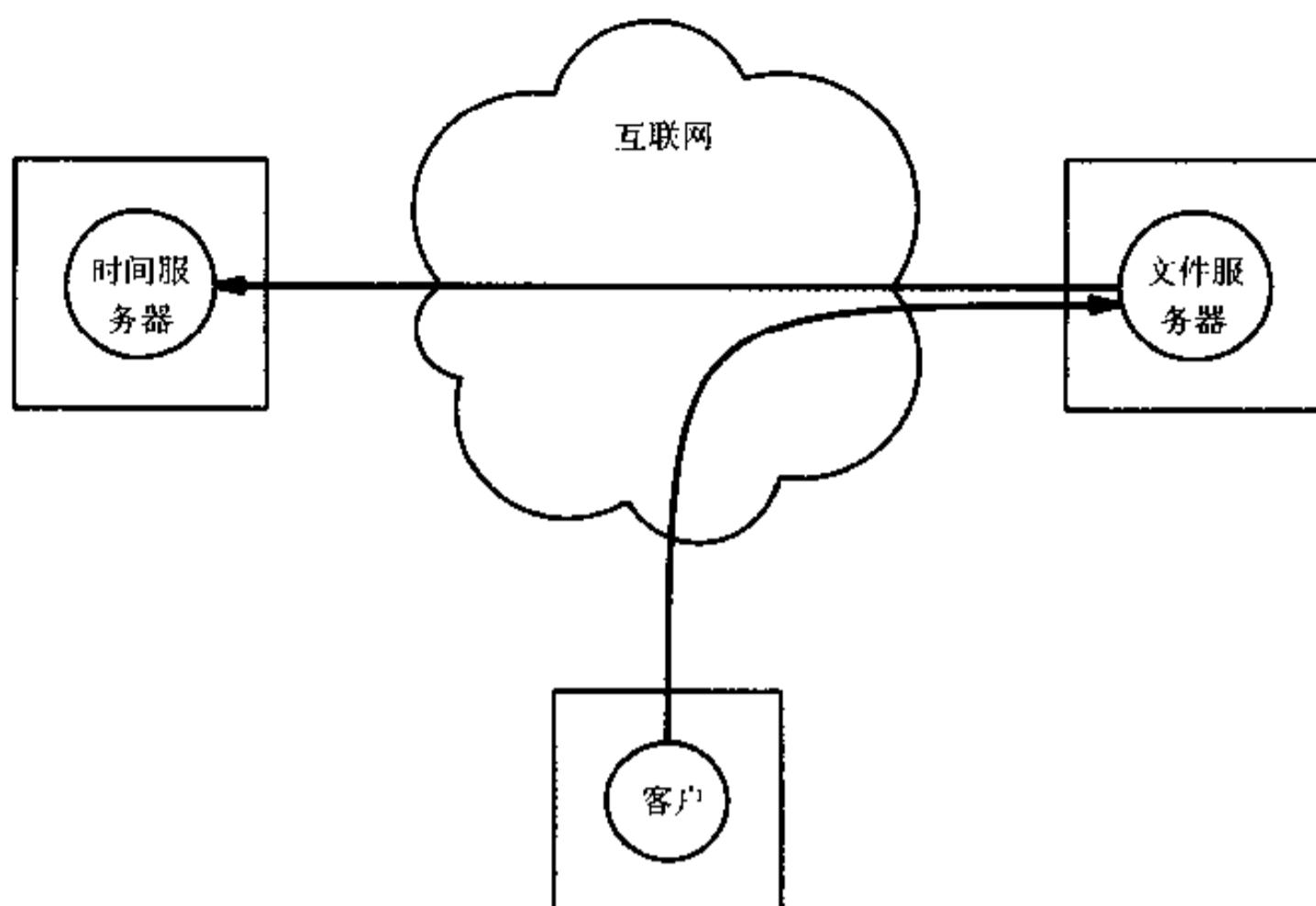


图 2.3 对时间服务器来说，文件服务器相当于一个客户。当时间服务器应答后，文件服务器将结束计算并将结果返回给原来的客户

在一个有许多可供使用的服务器的网络环境中，经常可以发现某一个应用的服务器对另一个应用则是客户。当然，设计人员必须小心从事以避免在这些服务器之间出现循环的依赖关系。

2.4 小结

客户 - 服务器范例将通信应用程序分为两类，即要么是客户，要么是服务器，这取决于它是否发起通信。除了为标准应用设计的客户和服务器软件外，许多 TCP/IP 用户为其自定义的非标准应用构建了客户和服务器软件。

初学者以及多数有经验的程序员使用 TCP 在客户与服务器之间传输报文，因为 TCP 提供了互联网环境所需要的可靠性。程序员只有在 TCP 不能解决问题时才选择 UDP（例如支持广播或组播交付）。

在服务器中维护状态信息可以提高效率。然而，如果客户的机器意外崩溃或者下层的传输网络让分组重复、延迟或丢失，状态信息会耗尽资源或变得不正确。因此，多数应用协议设计者努力减少状态信息。如果应用协议不能使操作成为幂等的，就可能不能使用无状态服务器。

程序不能简单地划分为客户和服务器这两类，这是因为许多程序同时具有客户和服务器这两种功能。一个程序对某个服务来说是服务器，但它又可作为客户请求其他的服务。

深入研究

Steven [1998] 简要描述了客户 - 服务器模型并给出了一些 UNIX 例子。其他例子可参考各个厂商的操作系统所提供的应用。

习题

- 2.1 在你的标准应用客户的本地实现中，有哪些是全参数化的？为什么需要全参数化？
- 2.2 标准的应用协议，如 TELNET、FTP、SMTP 以及 NFS（网络文件系统），它们是无连接的还是面向连接的？
- 2.3 当一个客户的请求到达时，如果不存在服务器，按照 TCP/IP 的规范将发生什么（提示：看看 ICMP）？在你的本地系统中情况是怎样的？
- 2.4 写出无状态文件服务器所需要的数据结构和报文格式。若两个或更多的客户访问同一个文件会发生什么？若客户在关闭某个文件前崩溃了又会如何？
- 2.5 服务器使用端点标识符标识客户比使用句柄安全吗？为什么（提示：中间的路由器可以看到因特网中传输的分组内容，而一台计算机可能崩溃和重启动）？
- 2.6 如果一个句柄只含有一个表的索引，指派给状态表中某个条目的一个句柄可能被重复指派。设计一种机制将一个句柄高效地映射到某个表条目，在每次重用一个表条目时都会为之指派一个新句柄。
- 2.7 写出有状态文件服务器所需要的数据结构和报文格式。使用 open、read、write 和 close 等操作来存取文件。让 open 操作返回一个整数，而该整数又被 read 和 write 操作用来读写文件。若一个客户发送一个 open 后崩溃了，然后重启动，并再次发出一个 open 请求，请问你如何区分这个客户重复发出的 open 请求？
- 2.8 在上一题中，如果有两个或更多的客户存取同一个文件，你的设计会如何？要是客户在关闭文件前崩溃了，情况又如何呢？
- 2.9 仔细检查 NFS 远程文件存取协议，识别哪些操作是幂等的。若报文丢失、重复或延迟将会产生什么差错？
- 2.10 请问幂等的协议报文是否应比非幂等的协议报文大一些？为什么？

第3章 客户–服务器软件中的并发处理

3.1 引言

上一章定义了客户–服务器范例。本章通过讨论并发来扩展这种客户–服务器交互的概念。并发提供了很多蕴藏在客户–服务器交互背后的能力，但也使软件的设计和构建变得更困难。在以后的各章中还会涉及并发的概念，在那些章节中我们将详细解释服务器如何提供并发访问。

除了讨论并发的一般概念，本章还回顾了操作系统为支持并发处理而提供的那些设施。理解本章描述的功能很重要，因为在后续章节的许多服务器实现中会用到这些功能。

3.2 网络中的并发

术语并发（concurrency）指真正的或表面呈现的同时计算。例如，一个多用户的计算机系统可以通过分时（time sharing）获得并发。分时是一种设计，它使单个处理器在多个计算任务之间足够快地切换，从表面上看这些计算是同时进行的。或者通过多处理（multiprocessing）获得并发，这种设计让多个处理器同时执行多个计算任务。

分布式计算的基础是并发处理，并发会以多种形式出现。在单个网络中的各台机器之间，许多成对的应用程序可以并发地通信，共享使它们互连的网络。例如，在一台机器中的应用进程A可能与另一台机器中的应用进程B进行通信，同时，在第三台机器中的应用进程C可能与第四台机器中的应用进程D通信。尽管这些进程都共享一个网络，但这些应用进程看上去像是在独立地运行。网络硬件执行一些访问规则，这些规则允许每对通信的机器之间相互交换报文，并且能防止某一应用进程占用了全部的网络带宽而排斥其他应用进程的通信。

在一个特定的计算机系统中也有并发产生。例如，一个分时系统中的多个用户可以各自调用一个客户应用，这些客户应用将与另外一台机器中的应用进行通信。一个用户在传送文件时，另一用户可以进行远程登录。从用户的观点看，好像所有的客户程序都在同时运行。

除在单台机器中的各个客户之间有并发外，在一组机器上的所有客户之间也可以并发执行。图3.1说明了运行于几台机器上的客户程序之间的并发。

要使客户软件具有并发，往往并不需要程序员为此特别花费工夫。应用程序员在设计和构建各个客户程序时不需要考虑其并发；由于操作系统允许多个用户在同一时间各自调用某个客户程序，这些客户程序间的并发与生俱有。因此，各个客户程序的运行很像普通的程序。概括地说就是：

大多数客户软件都能够并发运行，这是因为底层的操作系统允许用户并发地执行客户程序，或是因为多台机器上的用户在同一时间各自执行客户软件。单个客户程序就像普通的程序那样运行；它并不明显地管理并发。

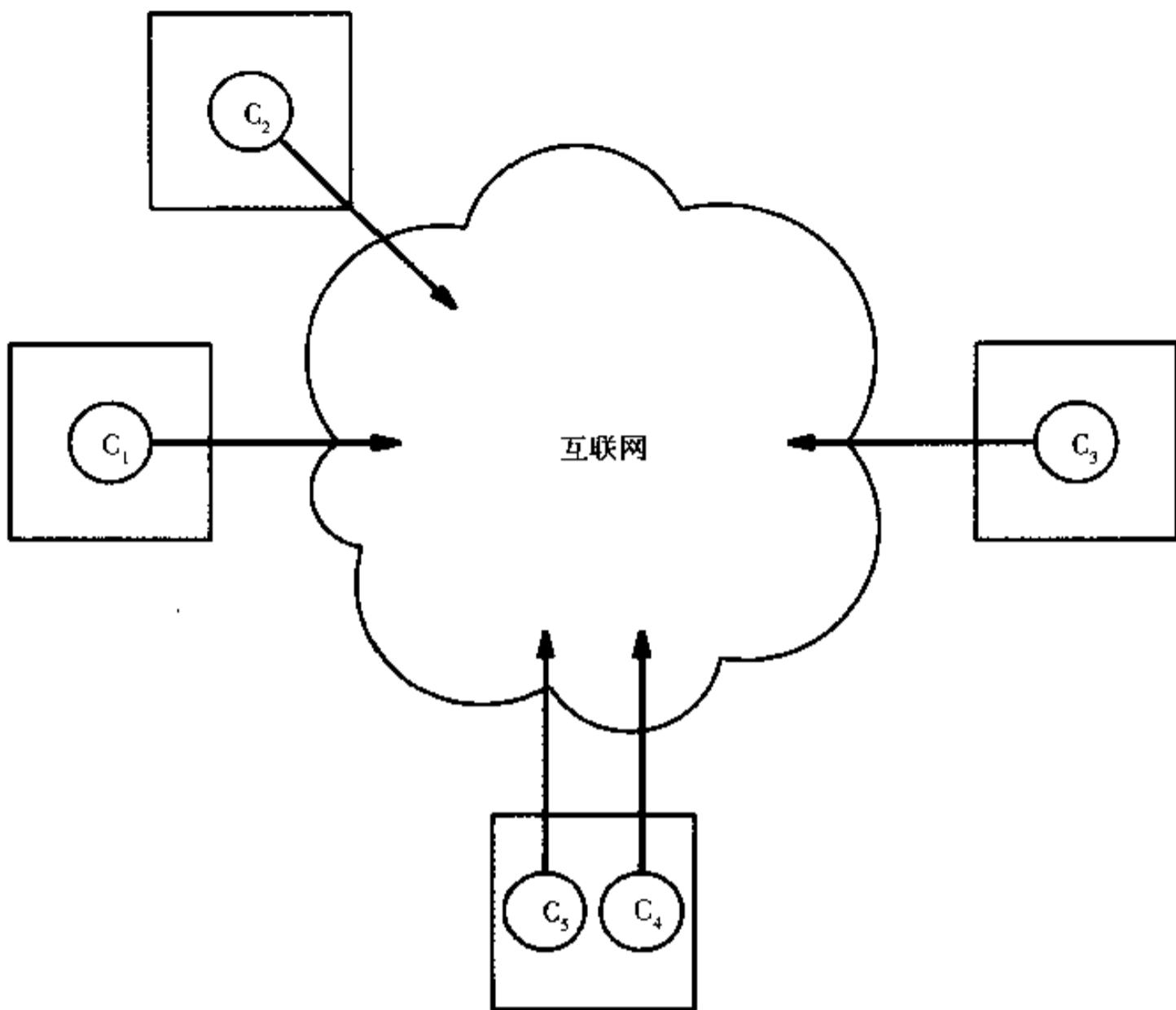


图 3.1 当用户在多台机器上同时执行客户程序时，或者当一个任务操作系统允许多个客户程序的副本在单台计算机中并发执行时，这些客户程序之间就产生了并发

3.3 服务器中的并发

与并发的客户软件不同的是，服务器中的并发实现需要花费相当的工夫。如图 3.2 所示，单个服务器必须并发地处理多个传入请求（incoming request）。

为理解并发的重要性，考虑一下需要大量计算或通信的服务器操作。例如，设想一个远程登录服务器，如果它不能并发运行，而是一次只能处理一个远程登录。一旦有一个客户与该服务器建立了联系，服务器必须忽略或拒绝后继的请求，直到第一个用户结束会话。很明显，这样的设计限制了服务器的使用，而且使得多个远程用户不能在同一时间访问某台机器。

第 8 章讨论了并发服务器的算法和设计问题，还说明了它们的运行规则。第 9 章到第 13 章各说明了一个并发算法，更详细地描述了其设计，并提供了可以使用的服务器代码。本章的其余部分将集中讨论全书所要使用的术语和基本概念。

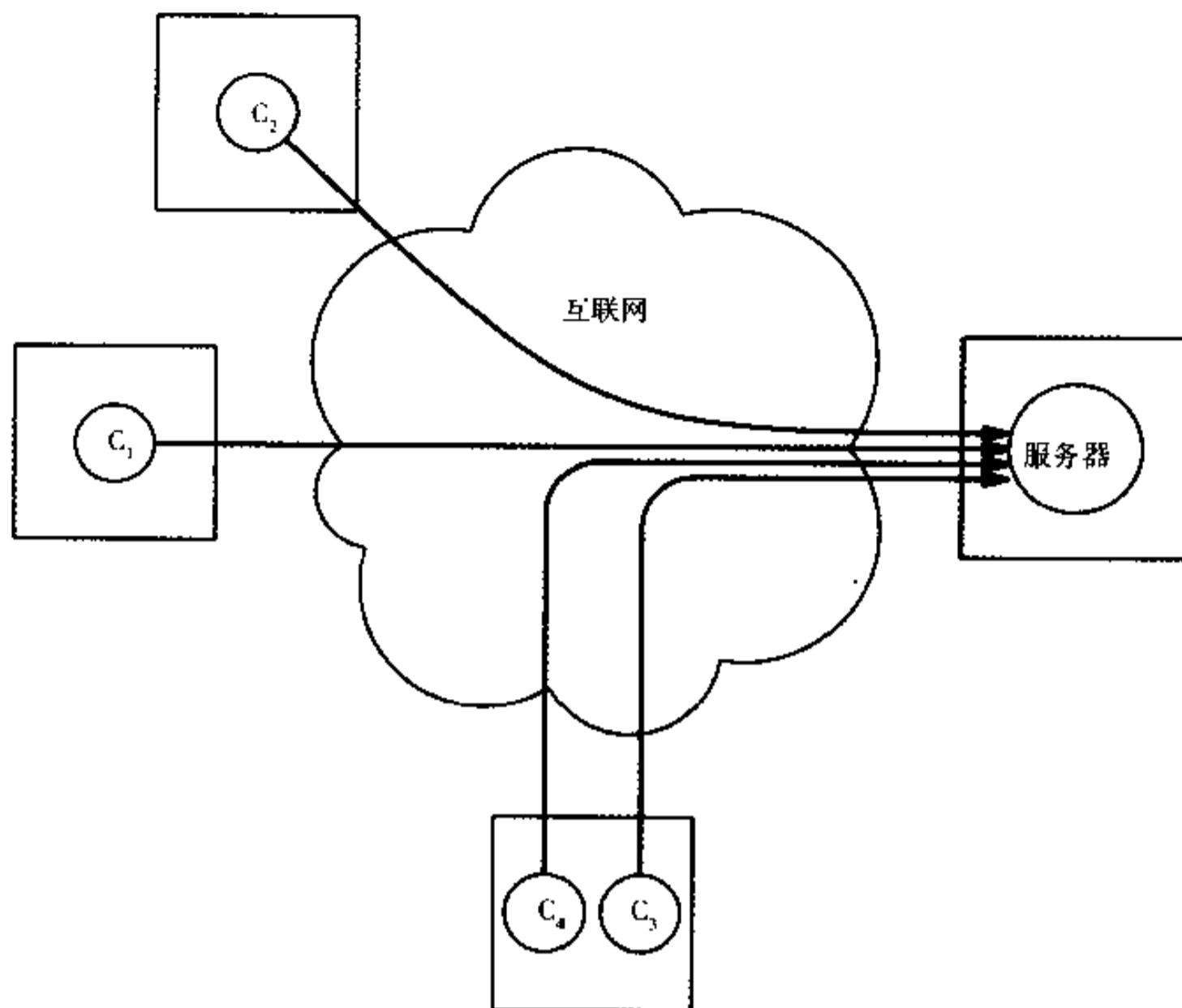


图 3.2 服务器软件必须在编程中处理好并发请求，因为多个客户使用服务器的一个熟知协议端口与服务器联系

3.4 术语和概念

由于具有设计并发程序经验的程序员不太够，所以理解服务器中的并发还是有一定难度的。本节解释并发处理的基本概念，并说明操作系统是如何提供并发的。本节还举了一些例子来说明并发，并定义了以后章节中要使用的术语。

3.4.1 进程概念

在并发处理系统中，进程（process）这种抽象定义了计算的基本单元^①。我们认为一个进程包括一段地址空间和至少一个执行的线程（thread）。线程最重要的信息是一个指令指针（instruction pointer），它指明该进程正在执行的地址。其他与进程相关的信息包括拥有该进程的用户标识、正在执行的已编译的程序，以及进程的程序文本和数据区的存储器位置。

进程与程序不同，因为进程的概念只包括一个计算的活动执行，而不是程序的静态版本。在用户创建一个进程后，操作系统将程序的一个副本装入到计算机中，然后启动一个线程执行程序。具体地说，一个并发处理系统允许多个线程（一个进程中的多个线程或多个进程中的多个线程）“在同一时间”执行同一段代码。各个线程都按照各自的步调运行，各自在任意时刻开始或结束运行。

^① 有些系统使用术语任务（task）、作业（job）来表示进程，有些系统中 process 采用大写形式：Process。

因此，每个线程执行的代码位置可能各不相同。由于每个线程有其各自的指令指针，该指针将指出线程下一步将执行的指令，因此线程之间决不会有任何冲突。

当然，对一个单处理器的体系结构来说，单个CPU在任一时刻只能执行一个线程。操作系统通过在所有正在执行的线程间快速切换CPU，使得计算机看上去好像在同时执行多个计算。从观察者的角度看，许多线程像是在同时运行的。实际上，一个线程执行了一小段时间后，另一个线程接着执行一小段时间，如此反复。我们用并发执行（concurrent execution）这个术语来描述此做法，它的意思是“从表面上看是在同时执行”。在单处理器系统中由操作系统处理并发，而在多处理器系统中，所有的CPU可以同时执行多个线程。

有一点很重要：

程序员在构建并发环境中使用的应用程序时，并不知道底层的硬件是由一个处理器还是由多个处理器构成的。

3.4.2 局部和全局变量的共享

在一个并发处理系统中，常规的应用程序只是一种特例：它包括一段代码，而这段代码在某一时刻仅被一个线程执行。在其他方式上，线程（thread）的概念与程序（program）的一般概念不同。例如，大多数应用程员认为程序中定义的变量是与代码放在一起的。然而，如果多个线程并发地执行这段代码，那么，每个线程就很有必要各自拥有这些变量的副本。为理解这样做的理由，考虑下面这段C语言代码，打印从1到10这几个整数：

```
for (i=1; i<=10; i++)
    printf("%d\n", i);
```

代码的循环部分使用了一个序号变量*i*。在常规的程序中，程序员认为变量*i*的存储位置与代码在同一个地方。然而，如果有两个或多个线程并发地执行这个代码段，其中的某一个线程可能在第六次循环时，另一个线程才开始第一次循环。每个线程都应该各有一个变量*i*的副本，否则就会产生冲突。概括而言：

当多个线程并发地执行同一段代码时，对这段代码所涉及的变量，每个线程都应各有一份独立的副本。

进程模型将独立变量副本的做法扩大到了包含全局变量。例如，下面这个程序的开头部分就包含一个全局变量*x*的声明：

```
int      x;
main (argc, argv)
    int argc;
    char *[] argv; {
```

操作系统为每个执行程序的进程创建变量*x*的一个独立副本。但是，一个进程内的多个线程可以共享同一个副本。总结如下：

每个进程拥有全局变量的副本；如果多个线程在同一个进程中执行，则它们各自拥有局部变量的副本，但都可共享进程的全局变量副本。

3.4.3 过程调用

在像 Pascal 或 C 这样的面向过程的语言中，执行码可以包含子程序的调用（过程或函数）。子程序接受参数、计算出结果，然后返回到调用点之后。如果多个线程并发地执行代码，同一时刻它们可能处于过程调用序列中的不同调用点上。一个线程 A 开始执行后调用某个过程，然后再调用第二级的过程，而这之后，另一个线程 B 才开始执行。当线程 B 从第一级过程返回时，线程 A 也许刚好从第二级调用返回。

面向过程编程语言的运行时（run-time）系统使用一种栈（stack）机制来处理过程调用。每进行一次过程调用，运行时系统就将一个过程激活记录（procedure activation record）压入栈中。这个激活记录存储了发生过程调用的指令在代码中的位置等信息。当过程执行完，运行时系统从栈顶弹出激活记录并返回发生这个调用的过程。与局部变量相似，并发编程系统提供了运行线程的过程调用之间的分离：

当多个线程并发地执行一段代码时，每个线程拥有自己的过程激活记录运行时栈（run-time stack）。

3.5 一个创建并发进程的例子

3.5.1 一个顺序执行的 C 实例

下面的例子说明了在 UNIX 操作系统中^①的并发处理。从大多数计算的角度而言，编程语言的语法并不重要；它只用了几行代码。例如，下面的 C 代码是一段常规的 C 程序，该程序打印从 1 到 5 的整数以及它们的和：

```

/* sum.c - A conventional C program that sums integers from 1 to 5 */
#include <stdlib.h>
#include <stdio.h>
int sum; /* sum is a global variable */

main() {
    int i; /* i is a local variable */

    sum = 0;
    for (i=1 ; i<=5;i++) { /* iterate i from 1 to 5 */
        printf("The value of i is %d\n", i);
        fflush(stdout); /* flush the buffer */
        sum += i;
    }
    printf("The sum is %d\n", sum);
}

```

^① 在整本书中，我们使用通用名词 UNIX 表示 UNIX 分时系统的若干变形版本，而使用 Linux 表示 UNIX 系统中的一个特定的版本。在下面的代码中，我们试图使用遵循 POSIX 标准的 Linux 调用的一个子集。

```
    exit(0);                                /* terminate the program      */
}
```

当程序执行完后，产生六行输出：

```
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
The sum is 15
```

3.5.2 程序的并发版本

为在 UNIX 中创建一个新进程，程序要调用系统函数 fork^①。本质上说，fork 将运行着的程序分成为两个（几乎）完全一样的进程，每个进程都启动一个从代码的同一位置开始执行的线程。这两个进程中的线程继续执行，就像是两个用户同时启动了该应用程序的两个副本。例如，下面这段程序是上例的一个修改版本，它调用 fork 创建一个新的进程（注意，尽管并发的引入彻底地改变了程序的含义，但 fork 调用只占用了一行代码）。

```
#include <stdlib.h>
#include <stdio.h>
int sum;

main() {
    int i;

    sum = 0;
    fork();                                /* create a new process */
    for (i=1 ; i<=5 ; i++) {
        printf("The value of i is %d\n", i);
        fflush(stdout);
        sum += i;
    }
    printf("The sum is %d\n", sum);
    exit(0);
}
```

当某个用户执行这个并发版本的程序时，系统创建一个含单个线程的进程执行代码。然而，当线程执行到 fork 调用时，系统会复制这个进程，在新进程中创建一个线程，而且让原来的线程和新创建的线程继续执行。当然，每个线程都各自拥有一份程序所要使用的变量的副本。实际上，想像

^① 对程序员来说，对 fork 的调用不光看上去像而且使用上也像一般 C 函数调用。其调用写为 fork()。然而，在运行时，控制权交给了操作系统，由它来创建一个新进程。

下一步的情况到底怎样，最简单的方法是设想系统建立了全部运行程序的第二份副本。然后，设想两个副本都在运行（就像两个用户都已同时执行了该程序）。概括而言：

可以这样理解 fork 函数，想像 fork 调用导致操作系统建立了执行程序的一份副本，并且允许两份副本同时执行。

在某个特定的单处理器系统上，我们的并发例子程序执行后产生了十二行输出：

```
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
The sum is 15
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
The sum is 15
```

在我们所使用的硬件平台上，第一个进程中的线程运行得如此之快，以至于在第二个进程中的线程开始执行前就能执行完毕。一个进程的所有线程都执行完毕后，操作系统将终止该进程。第一个进程终止后，操作系统就将处理器切换到第二个进程中的线程，它开始执行直到完成。整个执行过程花了不到1秒的时间。操作系统的开销主要在线程的切换、进程的终止以及系统调用的处理上，包括 fork 调用和写输出所用的调用，这两者所用的时间加起来不到全部时间的 20%。

3.5.3 时间分片

在我们这个例子程序中，每个线程在5次循环的过程中只执行了一些简单的计算。因此，一旦某个进程中的线程获得了 CPU 的控制权，它很快就能执行完。思考执行更多计算的并发线程，就会发现一个有趣的现象：操作系统每次只将很小一段时间的 CPU 资源分配给一个线程使用，然后就将 CPU 资源转移给下一进程。我们使用时间分片 (timeslicing) 这一术语来描述多个并发线程共享可用 CPU 的系统。例如，如果一个分时系统只有一个 CPU 可供分配，而程序被分为两个线程，这时，其中一个线程执行一段时间后，另一个线程也将执行一段时间，接着，第一个线程又将执行，如此反复。如果该分时系统有很多线程，在第一个线程再一次执行之前，每个线程都将执行一小段时间。

时间分片机制试图在所有线程间平均分配可用的处理器资源。如果只有两个线程有待执行，而计算机只有一个处理器，那么，这两个线程将各自获得大约 50% 的 CPU 处理时间。如果某个单处理器计算机上有 N 个线程有待执行，那么每个线程将各自获得大约 1/N 的 CPU 处理时间^①。因此，不管有多少个线程在执行，所有线程都好像是以相等的速率推进。当有很多线程执行时，速率很

^① 一些时间分片机制是在多个进程间平均分配 CPU 时间，然后再将各个进程获得的 CPU 时间分给该进程内的多个线程。

低；而只有很少线程执行时，速率就很高。

下面我们举一个例子来说明时间分片的效果，该程序中的每个线程的执行时间要比一个分配的时间片长。将以上并发程序中的循环次数由5次扩大到10 000次，我们得到如下程序：

```
#include <stdlib.h>
#include <stdio.h>
int sum;

main() {
    int i;

    sum = 0;
    fork();
    for (i=1 ; i<=10000 ; i++) {
        printf("The value of i is %d\n", i);
        fflush(stdout);
        sum +=i;
    }
    printf("The total is %d\n", sum);
    exit(0);
}
```

当这个并发程序在同样的系统上执行后，它会产生20 002行输出。然而，这时并不是所有来自第二个线程的输出都跟在第一个线程的输出后面，而是来自两个线程的输出相互混杂到一起了。在一次运行中，第一个线程循环了74次后，第二个线程才开始执行。接着，在第二个线程循环63次后，系统又切换回第一个线程。在接下来的时间片中，每个获得足够CPU时间的线程执行的循环次数在60到90之间。当然，这两个线程要与运行在该计算机上的其他线程竞争使用CPU，因此，随着系统正在运行之程序的数目的不同，进程看上去的执行速率也不同。

3.5.4 单线程的进程

虽然我们知道一个进程可包含多个执行的线程，`fork()`却不能复制该进程中的所有线程。在`fork`创建一个运行进程的副本时，新进程只含有一个线程——执行`fork()`的那个线程。因此，新创建的进程称为单线程的进程。

事实上，单线程的进程很普通。例如，当用户调用一个命令时，会产生一个单线程的进程。操作系统的shell在执行一个命令时只创建一个进程，该进程只启动一个线程执行命令。因此，除非程序员明确创建了多个线程，否则每个进程都是单线程的。要点是：

虽然一个进程可以包含多个线程，`fork`调用新创建的进程都是单线程的。

3.5.5 使各进程分离

至此，我们已知道了`fork`可以用来创建新的进程，这个新进程与原来的进程执行完全相同的代码，而且代码中的变量值也是相同的。为运行的程序创建一个完全一样的副本既没有意思也没有用处，因为这意味着两个副本执行完全一样的计算。在实际使用中，由`fork`创建的进程并不与原来的

进程完全相同：在一个小细节上它们是不同的。Fork 是个函数，要向它的调用者返回一个值。当这个函数调用返回时，返回给原来进程的值与返回给新创建进程的值是不同的。在新创建的进程里，fork 返回零；在原来的进程里，fork 返回一个正整数来标识新创建的进程。从技术角度而言，这个返回值称为进程标识符（process identifier）或进程 id^①。

并发程序利用 fork 的这个返回值决定如何继续运行。在最常见的情况下，代码中包含一个测试返回值是否非零的条件语句。

```
#include <stdlib.h>
int      sum;

main() {
    int pid;

    sum = 0;
    pid = fork();
    if (pid != 0) { /* original process */
        printf("The original process prints this.\n");
    } else {          /* newly created process */
        printf("The new process prints this.\n");
    }
    exit(0);
}
```

在以上的例子代码中，变量 pid 记录了调用 fork 返回的值。不要忘记，对所有变量，每个进程都有其副本，fork 或者返回零（在新创建的进程中），或者返回非零值（在原来的进程中）。在调用 fork 之后，if 语句检查变量 pid 的值，从而判断正在执行的进程是原来的进程还是新创建的进程。这两个进程各自打印一条标识消息并退出。因此在程序运行时会出现两条消息：一条来自原来的进程，一条来自新创建的进程。概括而言：

在原来的进程和新创建的进程里，fork 所返回的值是不同的；并发程序利用这个区别让新进程执行与原来进程不一样的代码。

3.6 执行新的代码

UNIX 系统提供了一种机制，这种机制允许任一进程各自执行一个独立编译的程序。此机制含有一个系统调用 execve^②，它带有 3 个参数：一个文件名，文件中包含一个可执行对象程序（即一个已编译过的程序）；一个字符串参数数组的指针；以及一个字符串数组的指针，这些字符串构成了所谓的环境（environment）。

execve 用新程序的代码来替代当前正在执行的进程所运行的代码。这个调用不影响任何其他的进程。因此，一个进程必须调用 fork 和 execve，才能让新创建的进程执行从某个文件得到的目的代

^① 许多程序员将 process id 简写为 pid。

^② 有些版本的 UNIX 使用其旧名字 exec。

码。例如，只要用户在任何一个可用的命令解释器中输入一个命令，此命令解释器就使用 fork 为此命令创建一个新的进程，同时使用 execve 来执行此代码。

对于需要处理不同服务的服务器来说，execve 就特别重要。只要把每一种服务的代码和其他服务的代码分开，程序员就能将每一种服务作为一个独立的程序来构建、编写和编译。当服务器需要处理一个特殊的服务时，它就使用 fork 和 execve 来创建一个进程，并运行其中的一个程序。以后的几章将更详细地讨论这一概念，并列举一些服务器怎样使用 execve 的例子。

3.7 上下文切换和协议软件设计

尽管操作系统所提供的这些并发处理机制使程序的功能更强并且更易于理解，但它们确实有一些计算开销。为保证所有线程并发执行，操作系统采用了时间分片机制，在线程间非常快速地切换 CPU（或多个 CPU），以至于在人看来这些线程像是在同时执行。

当操作系统暂时停止执行某个线程而切换到另一个线程时，会发生上下文切换（context switch）：在同一个进程内的多个线程间切换上下文的开销比不同进程中的线程间切换的开销少一些。不管哪种情况，线程间切换上下文都要使用 CPU，而且在 CPU 正忙于切换时，任何应用线程都不能得到任何服务。因此，我们把上下文切换看成支持并发处理所付出的代价。

为避免不必要的开销，设计协议软件时应设法将上下文切换的次数减到最少。特别是，程序员必须要多加留意，保证在服务器中引入并发处理所带来的好处比上下文切换的开销多。下面的章节讨论在服务器软件中的并发用法，既提供了非并发的设计，也有使用单线程的进程的并发设计，以及使用多线程的进程的并发设计，另外还描述了每种设计所适用的环境。

3.8 并发和异步 I/O

除了对并发使用 CPU 提供支持外，一些操作系统还允许单个应用线程启动，并控制并发的输入和输出操作。系统调用 select 提供了一个基本的操作，围绕着这个系统调用，程序员可以构建管理并发 I/O 的程序。从原理上说，select 很容易理解：它允许一个程序询问操作系统哪个 I/O 设备已准备就绪。

举一个例子，设想一个应用程序从一个 TCP 连接读取字符，并将这些字符写到显示屏上去。这个程序可能还允许用户从键盘输入命令，控制数据如何显示。因为用户很少（或者从来不）输入命令，程序不能等着从键盘来的输入，它必须连续地从 TCP 连接中读取文本并加以显示。然而，如果程序试图从 TCP 连接中读取数据而连接中却没有任何数据，程序将会阻塞。在程序等待 TCP 连接上的输入数据而被阻塞时，用户可能会输入命令。问题是：程序不可能知道输入的数据是先从键盘来还是先从 TCP 连接来。为了解决这个问题，程序可以调用 select，并为两个输入来源指定响应的描述符。然后，程序可以询问操作系统，了解两个输入源谁先能够使用。当某个输入源就绪后，select 调用立即返回，程序就可以从这个输入源中读取数据了。此处只需要理解 select 的含义，在下面的章节中会详细说明它的用法。

3.9 小结

并发是 TCP/IP 程序的基础，因为它使用户不必一个等着一个地接受服务。多个客户中的并发

很容易发生，因为多个用户可以在同一时间执行客户应用软件。然而，实现服务器中的并发就困难多了，服务器软件必须通过编程来并发地处理请求。

在 UNIX 系统中，一个程序使用系统调用 `fork` 来创建新的进程。我们想像调用 `fork` 会使操作系统复制程序，使第二个进程中新创建的线程和原来进程中的线程执行同样的程序。从技术角度看，`fork` 是个函数调用，因为它返回一个值。原来的进程和由 `fork` 创建的进程间的惟一区别就是这个调用所返回的值不同。在新创建的进程中，该调用返回零；而在原来的进程中，它返回一个小的正整数，即新创建进程的进程 id。并发程序可利用这个返回值使新创建的进程执行与原来进程不同的程序。一个进程可以调用 `execve` 使进程执行一段独立编译的程序代码。

并发并非不用付出代价。当操作系统从一个进程切换上下文到另一个进程时，系统要使用 CPU。在服务器设计中引入并发的程序员必须保证：并发设计所带来的好处要超过由于上下文切换所引起的额外开销。

`Select` 调用允许单个进程管理并发 I/O。进程使用 `select` 来查明哪个 I/O 设备首先就绪。

深入研究

许多关于操作系统的教材描述了并发处理。Galvin 和 Silberschatz [1999] 全面地描述了一般性的问题。Comer [1984] 讨论了进程的实现、报文传递以及进程协调机制。Leffler 等 [1989] 描述了 4.3 BSD UNIX，在此基础上派生出了一些新系统，如 Linux。

习题

- 3.1 在你的本地计算机系统上运行例子程序。在一个时间片内，一个线程大约可以执行多少次输出循环？
- 3.2 编写一个启动五个进程的并发程序，让每个进程中的线程打印几行输出，然后停止。
- 3.3 阅读 POSIX 线程原语，编写一个在单个进程中创建五个线程的程序。再编写一个创建五个独立进程执行同样任务的程序，比较两个程序的运行时间。将两种并发的开销之差表示为三个参数的函数：全局变量数、局部变量数和线程的执行时间。
- 3.4 看看其他操作系统如何创建并发进程或线程。
- 3.5 进一步阅读 `fork` 函数的说明。新创建的进程与原来的进程共享哪些信息？
- 3.6 编写一个程序，使用 `execve` 来改变一个进程所执行的代码。
- 3.7 编写一个程序，使用 `select` 从两个终端（串行线）读取数据，并将结果显示在屏幕上，每个结果用标号标明数据的来源。
- 3.8 就上一练习重写一个程序，不使用 `select`。哪个版本更易理解？哪个版本效率更高？哪个版本更易于彻底地终止？

第4章 协议的程序接口

4.1 引言

前面几章描述了进行通信的程序间交互的客户-服务器模型，还讨论了并发性和通信之间的关系。本章将考虑在客户-服务器模型中，应用程序间通信接口的一般特征。下一章则通过详细介绍一种特定接口来说明这些特性。

4.2 不精确指明的协议软件接口

在多数实现中，TCP/IP协议软件驻留在计算机的操作系统中。因此，只要应用程序使用TCP/IP通信，它就必须与操作系统交互并请求其服务。从程序员的观点看，操作系统所提供的那些例程定义了应用程序和协议软件之间的接口，即应用程序接口 API (Application Program Interface)。

TCP/IP被设计成能运行在多厂商的环境之中。为了与各种不同的机器保持兼容，TCP/IP的设计者们都尽量避免使用任何一家厂商的内部数据表示。另外，TCP/IP标准还尽量避免让接口使用那些只在某一家厂商的操作系统中可用的特征，因此，TCP/IP和其应用程序之间的接口是不精确指明的 (loosely specified)。换言之：

TCP/IP标准没有规定应用软件与TCP/IP协议软件如何接口的细节；这些标准只建议了所需的功能集，并允许系统设计者选择有关API的具体实现细节。

4.2.1 优点与缺点

对协议接口使用不精确的指明，有优点也有缺点。从好的方面说，它提供了灵活性和容错能力。它允许设计者使用各种操作系统实现TCP/IP，这里的操作系统可以是个人计算机中所提供的最简单的系统，也可以是超级计算机所使用的很复杂的系统。更重要的是，它意味着设计者既可以使用户过程的接口方式，也可以使用消息传递的接口方式（最适合其所用的操作系统的方式）。

从坏的方面说，不精确的指明意味着：设计者可以使得不同操作系统中接口的实现细节有所不同。当厂商增加了与现有API不同的新接口时，应用编程就会更困难，应用程序在不同机器间的移植性更差。因此，尽管系统设计者偏爱不精确指明，但应用程序员却期望有一个受限制的规范，因为这样就可使应用不用改变即可在新机器上编译。

实际上，目前只有几种可供应用程序使用TCP/IP协议的API。加利福尼亚大学伯克利分校为Berkeley UNIX操作系统定义了一种API，后来的一些系统（包括Linux）也采用了这种API，该API业已称为套接字接口 (socket interface)，或者套接字。Microsoft在其操作系统中采用了套接字接口，套接字API的这种变形称为Windows Socket。AT&T为其UNIX系统V (System V) 定义了一种API，简写为TLI^①。此外还定义了几种API，但都还没有获得普遍接受。

4.3 接口功能

尽管 TCP/IP 没有定义一种应用程序接口，但标准确实对接口所需要的功能提出了建议。接口必须支持如下概念性操作：

- 分配用于通信的本地资源
- 指定本地和远程通信端点
- (客户端) 启动连接
- (客户端) 发送数据报
- (服务器端) 等待连接到来
- 发送或接收数据
- 判断数据何时到达
- 产生紧急数据
- 处理到来的紧急数据
- 从容终止连接
- 处理来自远程端点的连接终止
- 异常终止通信
- 处理错误条件或连接异常终止
- 连接结束后释放本地资源

4.4 概念性接口的规约

TCP/IP 标准并非让实现者得不到有关 API 的任何帮助。它们为 TCP/IP 指明了一个概念性接口 (conceptual interface)，其作用是提供示例。由于多数操作系统使用过程机制把控制权从应用程序传送给系统，所以该标准把这个概念性接口定义为一组过程和函数。该标准还提出了每个过程或函数所要求的参数以及它们所执行操作的语义。例如，TCP 标准讨论了 SEND 过程，并列出了它的参数，应用进程为在已有的 TCP 连接上发送数据必须提供这些参数。

概念性操作的定义很简单：

TCP/IP 标准定义的概念性接口并不指明数据的表示或编程的细节；它仅仅提供了一种可能的 API 的例子，操作系统可将此 API 提供给使用 TCP/IP 的应用程序。

因此，这种概念性的接口并未精确说明应用进程如何与 TCP/IP 进行交互。由于它没有精确描述细节，操作系统的设计者就可以自由选择其他的过程名或参数，只要它们能提供相同功能就行。

4.5 系统调用

图 4.1 说明了系统调用机制，大多数操作系统使用这种机制在应用程序和提供服务的操作系统过程之间传递控制权。对程序员来说，系统调用无论看上去还是行为上都像是函数调用。

① TLI 表示运输层接口 (Transport Layer Interface)。

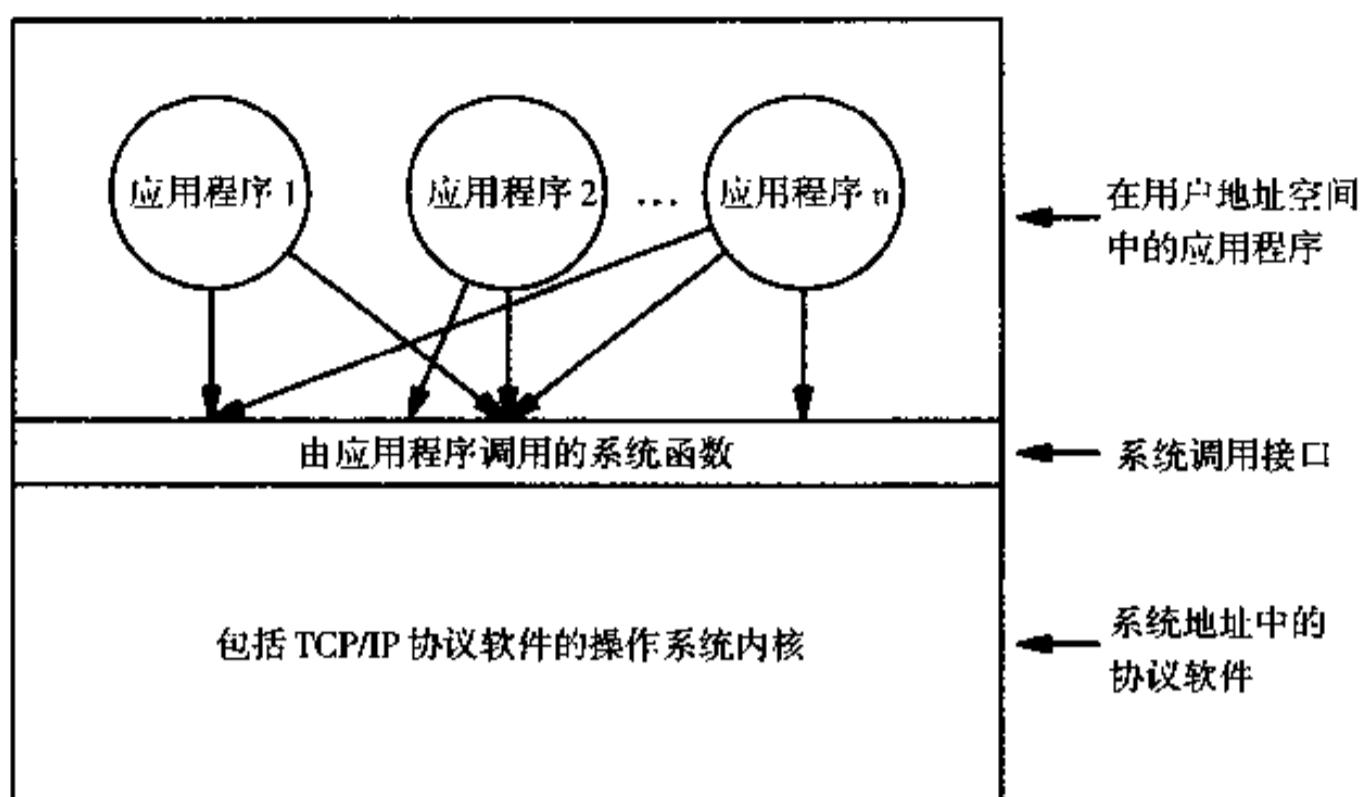


图 4.1 多个应用进程通过系统调用接口与 TCP/IP 协议软件进行交互。系统调用的行为与其他函数调用相似，只是系统调用将控制权传给了操作系统

如图所示，当某个应用进程启动系统调用时，控制权从应用进程传递给了系统调用接口。此接口又将控制权传递给了操作系统。操作系统将此调用转给某个内部过程，该过程执行所请求的操作。内部过程一旦完成，控制权通过系统调用接口返回给应用进程，然后应用进程将继续执行。从本质上说，只要应用程序需要从操作系统获得服务，执行这个应用程序的进程就将控制转给操作系统，执行必要的操作后就返回。由于进程要通过系统调用接口，它需要一定的特权，从而允许它读取或修改操作系统中的数据结构。然而，由于每个系统调用都会将控制转给一个由操作系统设计者所写的过程，操作系统还是要被保护的。

4.6 网络通信的两种基本方法

由于操作系统的设计者们把协议软件安装在操作系统中，他们就必须选择一组过程来访问 TCP/IP 协议。实现方法有两种：

- 设计者发明一组新的系统调用，应用程序用它们来访问 TCP/IP。
- 设计者使用一般的 I/O 调用访问 TCP/IP。

对第一种方法，设计者列举出所有的概念性的操作，为每个操作指定一个名字和参数，并分别把它们实现为一个系统调用。由于许多设计者认为，除非绝对必要，创建新的系统调用并不明智，所以这种方法很少采用。对第二种方法，设计者使用一般的 I/O 原语，但他们扩充了这些原语，使其既可用于网络协议，又可用于一般的 I/O 设备。当然，许多设计者选择了一种混合的方法，即尽可能地使用基本的 I/O 功能，但对那些不能常规表达的操作则增加其他的函数。

4.7 LINUX 中提供的基本 I/O 功能

为理解如何扩展一般的系统调用使其适用于TCP/IP，我们考虑一下Linux基本的I/O功能。Linux

和其他 UNIX 操作系统提供了一组（六个）基本的系统函数，用来对设备或文件进行输入/输出操作。图 4.2 中的表列出了这些操作以及它们通常的含义。

操作	含义
open	为输入或输出操作准备一个设备或文件
close	终止使用以前已打开的设备或文件
read	从输入设备或文件中获得数据，将数据放到应用程序的存储器中
write	将数据从应用程序的存储器传到输出设备或文件中
lseek	转到文件或设备中的某个指定的位置（此操作仅用于文件或类似磁盘的设备）
ioctl ^①	控制设备或用于访问该设备软件（比如，指明缓存的大小或改变字符集的映射）

图 4.2 LINUX 中提供的基本 I/O 功能

当应用程序调用 open 来启动输入或输出时，系统返回一个小整数，称为文件描述符（file descriptor），此应用程序在以后的 I/O 操作中会使用它。调用 open 带有三个参数：要打开的文件或设备的名字；一组位标志（bit flag），用来控制一些特殊的情况，例如，若文件不存在，是否要创建文件；一个访问模式，它为新创建的文件指定读/写保护。例如，代码段：

```
int      desc;
desc = open("filename", O_RDWR, 0)
```

打开一个现有的文件 filename，模式是既允许读又允许写。在获得了整数描述符 desc 后，应用程序在以后对这个文件的 I/O 操作中将使用这个标识符。例如，语句：

```
read(desc, buffer, 128);
```

从文件中读取 128 字节的数据并写入数组 buffer 中。

最后，当一个应用进程结束使用一个文件时，它将调用 close 来撤消标识符并释放相关的资源（例如，内部缓存）：

```
close(desc);
```

4.8 将 Linux I/O 用于 TCP/IP

当设计者们把 TCP/IP 协议加入到 UNIX 系统时，他们扩展了传统的 UNIX I/O 设施。首先，他们扩展了文件描述符集，使应用进程可以创建能被网络通信所使用的描述符。其次，他们扩展了 read 和 write 这两个系统调用，使其既可以同网络标识符一起使用，又可以同普通的文件标识符一起使用。这样，当应用进程需要通过 TCP 连接发送数据时，它就创建相应的标识符，并使用 write 传输数据。

然而，并非所有的网络通信都很容易地套用这种 UNIX 的 open-read-write-close 范例。应用进程必须指明本地的和远端的协议端口，远程 IP 地址，还有它将使用 TCP 还是 UDP，以及它是否要启动传输还是要等待传入连接（这就是说，它要作为客户还是要作为服务器）。如果应用进程是服务器，它必须指明在拒绝请求之前，可以接受多少传入连接请求被操作系统排队。此外，如果应用

① ioctl 代表输入输出控制（Input Output ConTroL）。

进程选择使用 UDP，它必须能传输 UDP 数据报，而不仅仅是字节流。套接字 API 的设计者提供了一些额外功能来适应这些特殊情况。在 Linux 中，与早期的 UNIX 系统一样，通过增加一些新的操作系统的系统调用来实现这些额外的功能。在下一章中我们将说明设计的细节。

4.9 小结

由于 TCP/IP 是为多厂商环境而设计的，协议标准没有精确指明应用程序使用的接口，并允许操作系统的设计师自由选择如何来实现这个接口。标准确实也讨论了概念性的接口，但它仅仅是作为一种说明性的示例。尽管这些标准把这种概念性接口定义为一组过程，但设计师可以自由选择不同的过程，或者干脆使用一种完全不同的交互风格（例如，消息传递）。

操作系统往往通过一种叫做系统调用接口的机制提供服务。当在系统中增加对 TCP/IP 的支持时，设计师们试图通过尽可能地扩展已有系统调用的功能，减少新增加的系统调用的数量。然而，由于网络通信所要求的一些操作不容易套用通常的 I/O 过程，大多数 TCP/IP 的接口还是需要几个新的系统调用。

深入研究

Linux 程序员手册（Programmer's Manual）的第 2 部分（Section 2）详细描述了每个套接字调用；4P 部分（Section 4P）详细描述了协议和网络设备接口。[AT&T 1989] 定义了 AT&T TLI 接口，它是 UNIX 系统 V 中所使用的套接字的替代品。

习题

- 4.1 考察一种提供消息传递的操作系统。你将如何扩展应用程序接口使其适用于网络通信？
- 4.2 比较 Berkeley UNIX 的套接字接口和 AT&T 的 TLI 接口。它们的主要区别是什么？它们有哪些相似点？有什么理由会使设计师选择某个设计方案而不是另一个？
- 4.3 有些硬件的体系结构把可能的系统调用的数量限制在一个很小的数量上（例如，64 或 128）。在你的本地操作系统中，指派了多少个系统调用？
- 4.4 考虑上一题中所讨论的硬件对系统调用数量的限制。系统设计者如何创建操作系统才能在其中增加新的系统调用而不改变硬件？
- 4.5 看看命令解释器脚本（shell script）如何使用设备（如 /dev/tcp）用作 TCP 的 API。写出脚本的例子。
- 4.6 看看 Perl 脚本语言。Perl 提供了什么 API，从而允许脚本使用 TCP/IP 协议？

第 5 章 套接字 API

5.1 引言

前面一章描述了应用程序和TCP/IP软件之间的接口，还说明了在大多数系统中是如何使用系统调用机制，将控制权传送给操作系统中的TCP/IP软件。我们还回顾了UNIX所提供的六个基本I/O函数：open、close、read、write、lseek和ioctl。本章将详细描述套接字API，并说明这些函数是如何与UNIX I/O函数集成到一起的。本章还涉及到一些通用概念，并给出了每个调用的使用方法。后面几章将说明客户和服务器是怎样使用这些调用的，并提供了一些说明许多细节的例子。

5.2 Berkeley 套接字

在20世纪80年代早期，远景研究规划局（Advanced Research Projects Agency, ARPA）资助了加利福尼亚大学伯克利分校的一个研究组，让他们将TCP/IP软件移植到UNIX操作系统中，并将结果提供给其他网点。作为项目的一部分，设计者们创建了一个接口，应用进程使用这个接口可以进行通信。他们决定，只要有可能就使用已有的系统调用，对那些不能方便地容入已有函数集的情况，就再增加新的系统调用以支持TCP/IP功能。这样就出现了套接字API（socket API）或称套接字接口（socket interface）^①，这个系统被称为Berkeley UNIX或BSD UNIX（TCP/IP首次出现于BSD 4.1版本（release 4.1 of Berkeley Software Distribution）；本书所描述的套接字函数来自BSD的4.4版本）。

由于许多计算机厂商，尤其是像Sun Microsystem、Tektronix以及Digital这样的工作站制造商，采用了Berkeley UNIX，于是在许多机器上都可以使用套接字接口。这样，套接字接口就已被广泛采用，以至于成为事实上的标准。微软也接受了这个标准，并为其操作系统开发了一个相应的实现版本。另外，Linux也使用了套接字。

5.3 指明一个协议接口

在考虑如何在操作系统中增加功能，使应用程序能够访问TCP/IP协议软件时，设计者们必须为函数选择名字，并指明每个函数所带的参数。为此，他们要决定各函数所提供的服务的范围，以及应用进程以何种方式来使用它们。设计者们还必须考虑，是让这个接口专门针对TCP/IP协议，还是使它能为其他协议所用。因此，设计者们必须在下列两大类方法中选择一个：

- 定义专门支持TCP/IP通信的一些函数。

^① 套接字接口有时也称为伯克利套接字（Berkeley socket）接口。

- 定义支持一般网络通信的函数，用参数使TCP/IP通信作为一种特例。

这两种方法的不同之处是很容易理解的，它们会影响到系统函数的名字以及这些函数所要求的参数。例如，对第一种方法，设计者可能会把一个系统函数取名为maketcpconnection，而对第二种方法，设计者也许会创建一个一般性的函数makeconnection并使用一个参数来指明使用TCP协议。

由于Berkeley的设计者想使接口适合多种通信协议，所以使用了第二种方法。实际上，纵观整个设计，他们提供了超出TCP/IP之外的通用性。他们允许使用多种协议族(family)，而把所有TCP/IP协议表示为单个族(PF_INET族)。他们还决定，让应用程序使用所要求的服务的类型(type of service)来指明操作，而不是指明协议名。因此，应用程序不是去指明它需要一个TCP连接，而是要求使用Internet协议族的流传送(stream transfer)类型的服务。我们可作如下概括：

套接字API提供了许多综合的功能，这些功能支持使用众多可能的协议进行网络通信。套接字调用把所有TCP/IP协议看作一个单一的协议族。这些调用允许程序员指明所要求的服务而不是指明某个特定协议的名字。

套接字的整个设计以及它们所提供的通用性从一开始就受到议论。一些计算机科学家认为，通用性是没有必要的，这只能使应用程序难于阅读。而另一些人则认为，让程序员指明服务的类型而不是指明协议，可使编程容易，因为这样做使程序员免于了解各个协议族的细节。最后，一些TCP/IP的商业厂商则强调更喜欢其他接口，因为，除非有了源代码，套接字不能加到操作系统中，而源代码往往需要一个特定的许可证以及附加的费用。

5.4 套接字的抽象

5.4.1 套接字描述符和文件描述符

需要执行I/O的应用程序要调用open函数，才能创建用于访问文件的文件描述符。如图5.1所示，操作系统将这些文件描述符实现为一个指针数组，这些指针指向内部的数据结构。系统为每个进程维护一个单独的文件描述符表。当一个进程打开某个文件后，系统就将一个指针(指向此文件的内部数据结构)写入进程的文件描述符表，并将这个表的下标返回给调用者。应用程序只需记住这个描述符，就可在以后要求对此文件进行操作的调用中使用该描述符。操作系统将此描述符作为该进程文件描述符表的下标来使用，并沿着指针可找到那个保存文件所有信息的数据结构。

套接字接口为网络通信增加了一个新的抽象，即套接字。就像文件一样，每个活动的套接字由一个小整数标识，称为套接字描述符。操作系统在与文件描述符相同的描述符表中分配套接字描述符。因此，一个应用进程不能拥有具有相同值的文件描述符和套接字描述符。

操作系统还有一个单独的系统函数socket，应用程序调用它来创建套接字；应用进程只使用open来创建文件描述符。套接字中所蕴涵的一般性的概念是：单个系统调用对创建任何套接字都是足够的。套接字一旦创建后，应用程序必须用其他的系统调用来指明准确使用此套接字的细节。在我们研究了系统所维护的数据结构后，这个范例将变得更清晰。

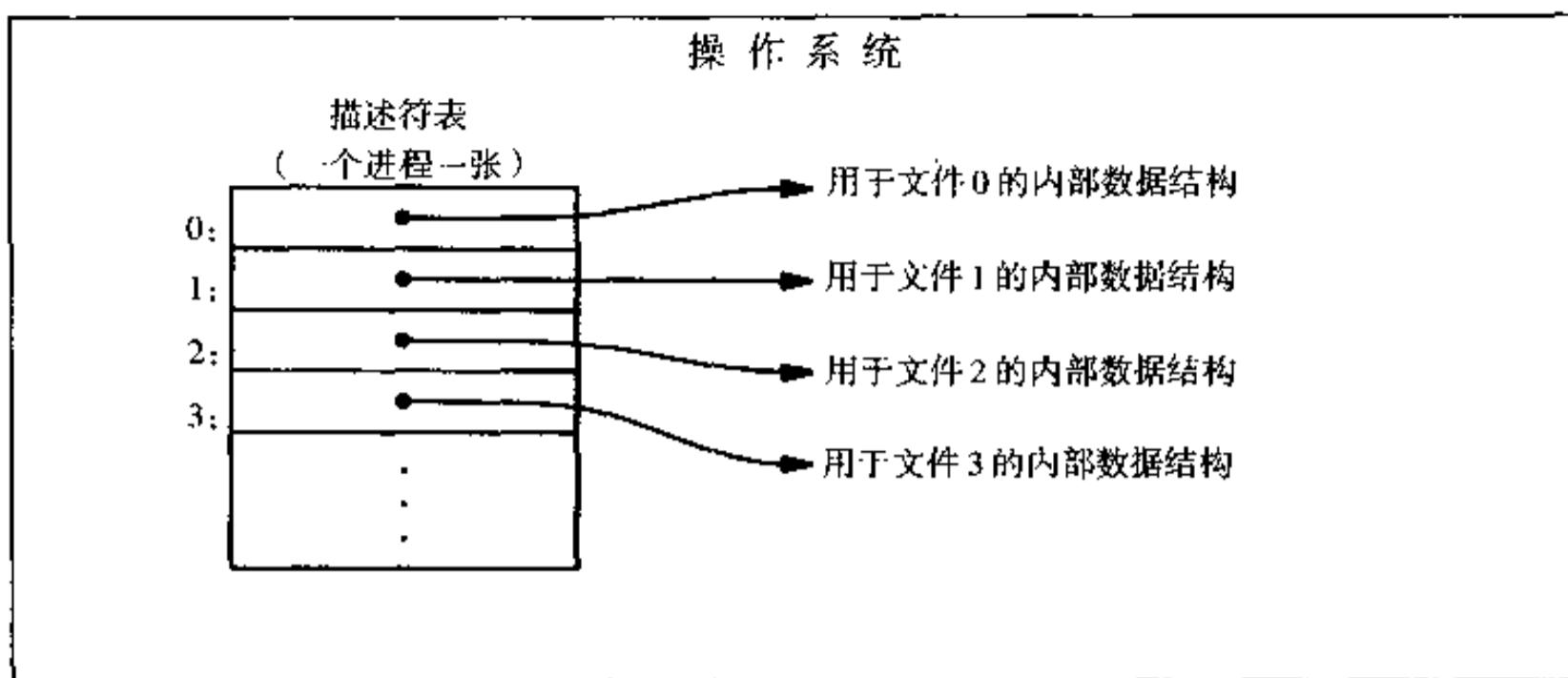


图 5.1 每个进程 (per-process) 的文件描述符表。操作系统使用进程的描述符表存储指向内部数据结构的指针，这些内部数据结构是针对那些进程业已打开的文件的。进程 (应用) 在访问这些文件时要使用此描述符

5.4.2 针对套接字的系统数据结构

了解套接字抽象的最简单的方法是想像一下操作系统中的数据结构。当应用进程调用 socket 后，操作系统就分配一个新的数据结构以便保存通信所需的信息，并在文件描述符表中填入了一个新的条目，该条目含有指向这个数据结构的指针。例如，图 5.2 说明了在调用 socket 后，图 5.1 中的描述符表会有什么变化^①。在本例中，socket 调用的参数指明协议族为 PF_INET、服务类型为 SOCK_STREAM。

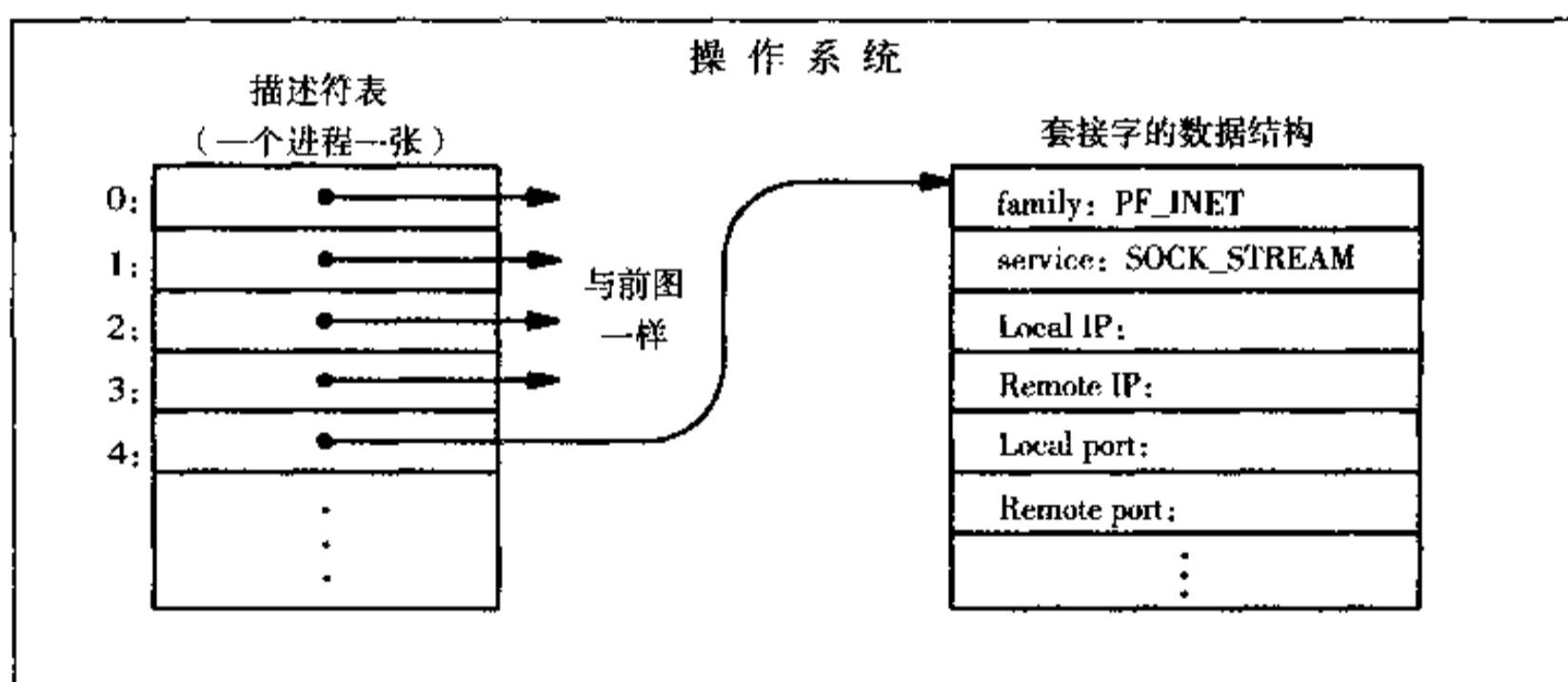


图 5.2 在调用 socket 后，操作系统的概念性的数据结构。
系统为 socket 和其他 I/O 使用同一个描述符表

尽管针对套接字的内部数据结构含有许多字段，在系统创建了套接字后，大多数字段中的值并没有填上。就像我们将看到的，在套接字能够被使用前，创建该套接字的应用程序必须用其他系统

^① 实际的数据结构要比图 5.1 所示的复杂；图中只说明了概念而不是细节。

调用把套接字数据结构中的这些信息填上。

5.4.3 主动套接字或被动套接字

套接字一旦创建，应用程序必须指定如何使用它，套接字本身是完全通用的，可以用来进行任意方式的通信。例如，服务器可以将套接字配置为等待传入连接，而客户可以将其配置为发起连接。

如果服务器将套接字配置为等待传入连接，就称此套接字为被动（passive）套接字；反之，客户用来发起连接的套接字就称为主动（active）套接字。其要点是：

主动套接字和被动套接字的惟一不同在于应用使用它们的方式；两种套接字最初的创建方式是相同的。

5.5 指明端点地址

在创建套接字时，并没有包含如何使用这个套接字的信息。具体说就是，套接字并没有包含本地或远程机器的协议端口号或者IP地址等信息。在应用进程使用一个套接字之前，它必须指明这些地址中的一个或者两个都指明。

TCP/IP协议定义了通信端点，它包括IP地址和协议端口号。其他协议族按照各自的方式定义端点地址。由于套接字抽象适用于多种协议族，所以它既没有指明如何定义端点地址，也没有定义一种特定的协议地址格式，而是改为允许每个协议族随其所愿地指明端点。

为允许协议族自由地选择其地址表示方式，套接字抽象为每种类型的地址定义了一个地址族。一个协议族可以使用一种或者更多的地址族来定义地址表示方式。TCP/IP各协议都使用一种单一的地址表示方式，其地址族用符号常量AF_INET表示。

实际上，TCP/IP协议族（表示为PF_INET）和它所使用的地址族（表示为AF_INET）引起了很大的混淆。主要的问题是：这两个符号常量都有相同的数字值（2），于是，在程序中，在本应使用某一个常量的时候却不慎用了另一个，而程序却能正确运作。甚至在最初Berkeley UNIX源代码中也含有这种误用。程序员应当注意到这个区别，这有助于澄清变量的意义并使程序更易移植。

5.6 类属地址结构

某些软件要使用协议地址，但它们不知道每种协议族定义其地址表示的细节。例如，可能有必要编写这样一个过程，它可以接受任意的协议端点说明，将它作为一个参数，并按照地址的类型在各种可能的动作中选择一个合适的动作。为适合这种程序，套接字系统定义了一般化的格式，它可以为所有端点地址使用。这种一般化的格式构成如下：

(地址族, 该族中的端点地址)

在这里，地址族（address family）字段包含一个常量，它表示预定义的地址类型。端点地址（endpoint address）字段包含有端点地址，它使用地址族所指明的那种地址类型的标准表示方式。

实际上，套接字软件为地址端点提供了预定义的C结构的声明。应用程序在需要声明存储端点地址的变量时，或要使用某种重叠来覆盖结构中的某个字段时，就要使用这种预定义的结构。最…

般的结构是 sockaddr 结构，它包含一个占 2 字节的地址族标识符，还有一个占 14 字节的数组存储地址^①：

```
struct sockaddr { /* struct to hold an address */
    u_char sa_len; /* total length */
    u_short sa_family; /* type of address */
    char sa_data[14]; /* value of address */
};
```

并不是所有的地址族都定义了适合这种 sockaddr 结构的端点。例如，某些 UNIX 定义了 AF_UNIX 地址族，它说明了一个地址族，程序员称其为命名管道（named pipe）。AF_UNIX 族中的端点地址由文件系统路径名构成，这个路径名可以比 14 字节长得多。这样，应用程序就不能在变量声明中使用 sockaddr 结构了，因为，声明为 sockaddr 类型的变量不能装下所有的可能的地址。

因为 sockaddr 结构适合于 AF_INET 族中的地址，这在实际当中常常引起混淆。于是，甚至在程序员把变量声明为 sockaddr 类型时，TCP/IP 软件也能正确工作。然而，为使程序可移植和可维护，TCP/IP 代码不能在声明中使用 sockaddr 结构。这种结构只能用于覆盖，而且代码只能引用该结构中的 sa_family 字段。

使用套接字的每个协议族都精确地定义了它的端点地址，套接字软件还提供了相应的结构声明。每个 TCP/IP 端点地址由下列字段构成：一个用来识别地址类型的 2 字节字段（必须包含 AF_INET），一个 2 字节的端口号（port number）字段，一个 4 字节的 IP 地址字段，一个还未使用的 8 字节字段。预定义的结构 sockaddr_in 指明了这种格式：

```
struct sockaddr_in { /* struct to hold an address */
    u_char sin_len; /* total length */
    u_short sin_family; /* type of address */
    u_short sin_port; /* protocol port number */
    struct in_addr sin_addr; /* IP address (declared to be
                               /* u_long on some systems) */
    char sin_zero[8]; /* unused (set to zero) */
};
```

只使用 TCP/IP 协议的应用程序可以只使用 sockaddr_in 结构；它不需要使用 sockaddr 结构^②。以上要点总结如下：

当表示一个 TCP/IP 通信端点时，应用程序使用结构 sockaddr_in，该结构包含了 IP 地址和协议端口号。在编写使用混合协议的程序时，程序员一定要注意，因为有些非 TCP/IP 的端点地址要求一个更大的结构。

5.7 套接字 API 中的主要系统调用

套接字调用可以分为两组：主调用（primary call），它提供对下层功能的访问；实用例程（utility

^① 本书所描述的结构与 Berkeley 软件版本 4.4 是一样的；sockaddr 结构的旧版本不包含 sa_len 字段。

^② sockaddr 结构用于改变指针的类型或改变系统函数结果的类型，以便使程序坚持严格的类型检查。

routine)，为程序员提供帮助。本节描述这些调用，它们提供客户和服务器所需要的主要功能。

套接字系统调用的细节、参数及其语义，都不是可避免的了。这很复杂，因为套接字带有一些参数，这些参数允许程序以许多方式使用它们。套接字可以被客户或服务器使用，可以使用流传送（TCP）通信或使用数据报（UDP）通信，可以使用特定的远程端点地址（往往为客户所使用）或使用非特定的远程端点地址（往往为服务器所用）。

为帮助了解套接字，我们将以考察套接字的主调用开始，并描述简单的客户和服务器如何使用这些调用和使用TCP进行通信。在以后的每一章中，我们都会讨论一种套接字的使用方式，并进行具体的说明。

5.7.1 socket 调用

应用程序调用socket函数创建一个新的套接字，这个新的套接字可以用于网络通信。该调用返回这个新创建的套接字的描述符。该调用的参数指明应用程序将使用的协议族（例如，TCP/IP使用PF_INET）以及将使用的协议或它所需要的服务的类型 [即流（stream）或数据报]。对一个使用Internet协议族的套接字，其协议或服务类型参数决定了它将使用TCP还是使用UDP。

5.7.2 connect 调用

在创建了一个套接字后，客户程序调用connect以便同远程服务器建立主动的（active）连接。connect的一个参数允许客户指明远程端点，它包括远程机器IP地址以及协议端口号。一旦连接建立，客户就可通过它传送数据了^①。

5.7.3 send 调用

客户和服务器都使用send在TCP连接上发送数据。客户常常使用send传输请求，而服务器使用send传输应答。send调用需要以下三个参数（应用程序要传递这些参数）：数据将要发往的套接字的描述符、数据要发往的地址、数据的长度。send往往要将外发数据（outgoing data）复制到操作系统内核中的缓存里，并允许应用程序在通过网络传输数据的同时继续执行下去。若系统的缓存满，send调用可能会暂时阻塞，直到TCP可以通过网络发送数据并在缓存中为新数据腾出空间为止。

5.7.4 recv 调用

客户和服务器都使用recv从TCP接收数据。在已经建立好连接后，服务器往往使用recv接收客户通过调用send而发送的请求。而客户在发送完请求后，使用recv接收应答。

为从连接中读取数据，应用程序要使用三个参数来调用recv。第一个参数指明所使用的套接字的描述符，第二个参数指明缓存地址，第三个参数指明缓存长度。recv调用抽出已到达该套接字的数据字节，并将其复制到用户的缓存中。若没有数据到达，调用recv将阻塞，直到有数据到达为止。如果到达的数据比缓存的容量多，recv只抽出能填满缓存的足够数据。若到达的数据比缓存的容量少，recv将抽出所有的数据并返回它所找到的字节数。

客户和服务器也可用recv接收来自套接字（使用UDP）的报文。如同面向连接时的使用情况，调用者提供三个参数，即：套接字标识符、缓存地址（数据将放置于此）、缓存的大小。对recv的

^① 下一章还会详细讨论连接的（connected）和未连接的（unconnected）套接字。

每次调用将取出一个进来的 UDP 报文（即一个用户数据报）。若缓存不能装下整个报文，recv 将填满缓存并把剩下的数据丢弃。

5.7.5 close 调用

客户或服务器一旦结束使用某个套接字，便调用 close 将该套接字撤消。若只有一个进程使用此套接字，close 就立即中止连接并撤消该套接字。若多个进程共享某个套接字，close 就把套接字的引用数减 1，当此引用数降到零时，就将该套接字撤消。

5.7.6 bind 调用

当套接字被创建时，它还没有任何关于端点地址的概念（本地地址和远程地址都还没有指派）。应用程序调用 bind 以便为一个套接字指明本地端点地址。对 TCP/IP 协议，端点地址使用 sockaddr_in 结构，它包含了 IP 地址和协议端口号。服务器主要使用 bind 来指明熟知端口号，它将在此熟知端口等待连接。

5.7.7 listen 调用

套接字被创建后，直到应用程序采取进一步行动前，它既不是主动的（准备由客户使用）也不是被动的（准备由服务器使用）。面向连接的服务器调用 listen 将一个套接字置为被动模式（passive mode），并使其准备接受传入连接。

大多数服务器由无限循环构成，该循环可以接受下一个传入连接，然后对其进行处理，完成后便返回准备接受下一个连接。处理一个连接哪怕只需要几个毫秒，在服务器正忙于处理这个现有的请求时，还是有可能刚巧又到来一个新的连接请求。为保证不会丢失连接请求，服务器必须给 listen 传递一个参数，告诉操作系统对某个套接字上的连接请求进行排队。因此，listen 的一个参数指明某个套接字将置于被动模式，而另一个参数将指明该套接字所使用的队列长度。

5.7.8 accept 调用

对 TCP 的套接字，服务器调用 socket 函数创建一个套接字，调用 bind 指明本地端点地址，调用 listen 将其置于被动模式，在这之后，服务器将调用 accept 以获取接下去的传入连接请求。accept 的一个参数指明一个套接字，将从该套接字上接受连接。

accept 为每个新的连接请求创建了一个新的套接字，并将这个新套接字的描述符返回给调用者。服务器只对这个新的连接使用该套接字，而用原来的套接字接受其他的连接请求。服务器一旦接受了一个连接后，它就可以在这个新的套接字上传送数据。在使用完这个新的套接字后，服务器将关闭该套接字。

5.7.9 在套接字中使用 read 和 write

如同大多数 UNIX 系统，在 Linux 中，程序员可以用 read 代替 recv，用 write 代替 send。对 TCP 和 UDP 套接字来说，read 具有和 recv 一样的语义，write 具有和 send 一样的语义。Read 和 write 的主要优点是程序员对它们已经很熟悉；而 send 和 recv 的主要优点是它们在程序中标记明显。

5.7.10 套接字调用小结

图5.3中的表给出了与套接字有关的系统函数的简要总结。

函数名	含义
socket	创建用于网络通信的描述符
connect	连接远程对等实体(客户)
send(write)	通过TCP连接外发数据(outgoing data)
recv(read)	从TCP连接中获得传入数据(incoming data)
close	终止通信并释放描述符
bind	将本地IP地址和协议端口号绑定到套接字上
listen	将套接字置于被动模式，并设置在系统中排队的TCP传入连接的个数(服务器)
accept	接收下一个传入连接(服务器)
recv(read)	接收下一个传入的数据报
recvmsg	接收下一个传入的数据报(recv的变形)
recvfrom	接收下一个传入的数据报并记录其源端点地址
send(write)	发送外发的数据报
sendmsg	发送外发的数据报(send的变形)
sendto	发送外发的数据报，往往是到预先记录下的端点地址
shutdown	在一个或两个方向上终止TCP连接
getpeername	在连接到达后，从套接字中获得远程机器的端点地址
getsockopt	获得套接字的当前选项
setsockopt	改变套接字的当前选项

图5.3 套接字函数及其含义的总结。read和write与recv和send是等价的

5.8 用于整数转换的实用例程

TCP/IP对协议首部(header)中所使用的二进制整数指明了一种标准的表示方式，称为网络字节顺序(network byte order)，它在表示整数时，让最高位字节在前。

尽管协议软件对应用程序隐藏了协议首部中所使用的大部分值，但程序员必须注意这个标准，因为有些套接字例程要求参数按网络字节顺序存储。例如，sockaddr_in结构中的协议端口域就使用网络字节顺序。

套接字例程中含有一些在网络字节顺序和本地主机字节顺序间进行转换的函数。程序应坚持调用这些转换例程，即使在本地主机字节顺序与网络字节顺序一样时也应如此，因为这样可以使源代码移植到任意体系结构的机器上。

这些转换例程分为短(short)和长(long)两类，用以分别处理16位和32位的整数。函数htons(host to network short，主机短整型到网络短整型的转换)将一个短整数从主机的本地字节顺序转换为网络字节顺序，而 ntohs(network to host short，网络短整型到主机短整型的转换)则反之。与之类似，htonl和ntohl把长整数从本地字节顺序变换为网络字节顺序或者反之。概括起来就是：

使用 TCP/IP 的软件调用函数 htons、 ntohs、 htonl 和 ntohl 将二进制整数在主机本地字节顺序和网络字节顺序之间进行转换。这样做使代码可以移植到任何机器上，而不管这台机器的本地字节顺序是什么。

5.9 在程序中使用套接字调用

图 5.4 说明了使用 TCP 的客户和服务器各自使用套接字函数的一种调用序列。客户创建套接字，调用 connect 连接服务器，交互时，使用 send（或者 write）发送请求，使用 recv（或者 read）接收应答。当使用连接结束时，客户调用 close。服务器使用 bind 指明它所使用的本地（熟知）协议端口，调用 listen 设置连接等待队列的长度，之后便进入了循环。在该循环中，服务器调用 accept 进行等待，直到下个连接请求到达为止，它使用 recv 和 send（或 read 和 write）同客户进行交互，最后使用 close 中止连接。之后，服务器回到 accept 调用，在那里等待下一个连接。

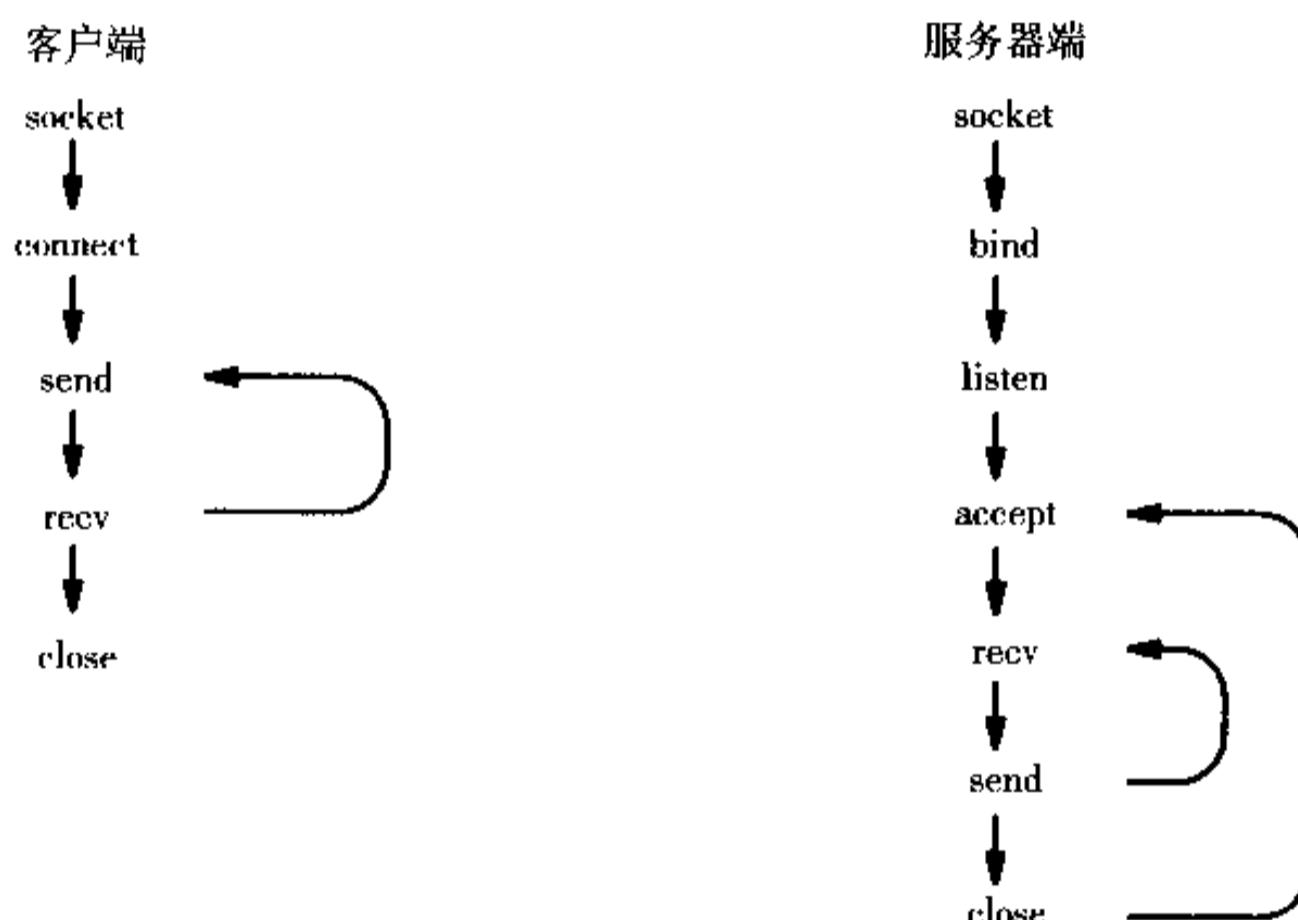


图 5.4 套接字系统调用的使用序列的例子，这个序列分别由采用 TCP 的客户和服务器所使用。服务器一直运行。它在熟知端口上等待新连接，然后接受这个连接，与客户通信，之后便关闭这个连接

5.10 套接字调用的参数所使用的符号常量

除了实现套接字接口的系统函数外，大多数 UNIX 提供了一组预定义的符号常量和数据结构声明，程序用这些常量和声明来声明数据和指明参数。例如，在说明是否使用数据报或流服务时，应用程序使用符号常量 SOCK_DGRAM 或 SOCK_STREAM。为此，应用程序必须使用 C 的预处理语句 include，以便把相应的定义合入每个程序中。include 语句往往出现在源文件的开始处，它们必须出现在使用这些（它们所定义的）常量之前。套接字所需要的那些 include 语句往往具有如下的形式：

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

在本书其余各章的程序里，我们假设都以这些语句开始，即使它们没有在例子中明确显示出来也是如此。概括起来就是：

套接字API的实现提供了套接字系统调用所要使用的符号常量和数据结构声明。引用这些常量的程序必须以C预处理语句`include`开始，该语句将引用出现这些定义的文件。

5.11 小结

BSD UNIX引入了作为一种机制的套接字抽象，它允许应用程序与操作系统中的协议软件接口。由于许多厂商采纳了套接字，套接字接口已经成为一种事实上的标准。

一个程序调用`socket`函数创建套接字并获得其描述符。`socket`调用的参数指明了所使用的协议和所要求的服务类型。所有的TCP/IP协议都是Internet协议族的一部分，它以符号常量`PF_INET`来说明。系统为套接字创建了一个内部的数据结构，并把协议族域填上，系统还使用服务类型参数来选择某个指定的协议（常常是`UDP`或`TCP`）。

其他的系统调用允许应用程序指明本地地址（`bind`），强迫套接字进入被动模式以便为某个服务器使用（`listen`），或者强迫套接字进入主动模式以便为某个客户使用（`connect`）。服务器可以进一步使用`accept`调用以获得入连接请求（`accept`），客户和服务器都可以发送或接收数据（`read`或`write`）。最后，在结束使用某个套接字后，客户和服务器都可以撤消（`close`）该套接字。

套接字结构允许每个协议族定义一个或多个地址表示方式。所有的TCP/IP协议使用Internet地址族，`AF_INET`指明含有IP地址和协议端口号的端点地址。当某个应用程序要为某个套接字函数指明通信端点时，它将使用预定义的结构`sockaddr_in`。若客户指明它需要一个任意的、没有使用的本地协议端口号，TCP/IP软件将为它挑选一个。

用C写的应用程序在能够使用那些与套接字相关的预定义的结构和常量之前，必须包含一些定义这些结构和常量的文件。具体说就是，我们假设所有的源程序都以这样的语句开始，即这些语句都包含文件`<sys/types.h>`和`<sys/socket.h>`。

深入研究

Leffler等人[1989]详细描述了Berkeley UNIX系统，并描述了用于套接字的内部数据结构。Presotto和Ritchie[June 1990]描述了一种针对TCP/IP协议的接口，它使用UNIX文件系统空间。Linux的联机文档含有套接字函数的规格说明，包括参数的精确描述和返回码。关于进程间通信的一节（节的标题是The IPC Tutorial）很值得一读。关于套接字的很多信息也可在附录A中找到。

习题

- 5.1 请查看那些针对套接字的`include`文件（往往是`/usr/include/sys/socket.h`），都允许什么类型的套接字？哪种套接字类型对TCP/IP协议不适用？
- 5.2 如果你的系统有一个精确度至少是微秒级的时钟，试测量一下每个套接字调用的执行时间有多长。为什么有些调用比其他调用所用的时间要长好几个数量级？
- 5.3 仔细阅读关于`connect`的手册页，若对`SOCK_DGRAM`类型的套接字调用了`connect`，会产

生什么样的网络通信量？

5.4 当一个应用程序对 SOCK_STREAM 类型的套接字执行了 connect 调用时，监视你的本地网络。你看到了多少个分组？

5.5 我们介绍过， send 和 write 都可以用于在套接字上上传输数据， read 和 recv 都可以用于从套接字上获取数据。检查有关程序采用了哪种调用？你偏好哪种？为什么？

第6章 客户软件设计中的算法和问题

6.1 引言

前面几章考虑了应用程序用来与 TCP/IP 软件接口的套接字抽象，还回顾了一些有关的系统调用。本章讨论客户软件所蕴涵的基本算法。在本章中，我们将讨论以下问题：应用程序是如何通过发起通信而成为客户的；这些应用程序是怎样使用 UDP 或 TCP 协议与服务器联系的；这些程序又是如何使用套接字调用与协议相交互的。下一章还将继续这个讨论，并展示一个实现这里所讨论的各种想法的完整的客户程序。

6.2 不是研究细节而是学习算法

TCP/IP 提供的丰富功能使得程序之间能够按照多种方式进行通信。使用 TCP/IP 的应用程序必须指明所期望的通信的许多细节。例如，应用程序必须指明它是作为客户还是作为服务器；它将使用的端点地址；通信时，它是使用一种无连接的协议还是一种面向连接的协议；它将如何强迫执行授权和防护准则；以及它所需要的缓存大小等详细内容。

到目前为止，我们业已讨论了提供给应用程序的那些操作，但没有讨论应用程序应该如何使用它们。遗憾的是，即使知道了所有可能的系统调用的低层细节以及它们的精确参数，也并不能使程序员懂得如何构建一个设计良好的、分布式程序。实际上，尽管对用于网络通信的系统调用进行一般了解是重要的，但很少有程序员能记住所有这些细节。取而代之的方法是，他们学习并牢记那些程序可以通过网络进行交互的各种可能的途径，而且了解每种可能设计的不足之处。从本质上讲，只要程序员对分布式计算中所蕴涵的算法有足够的了解，就可以迅速地做出设计决策，并在各种待选方案中做出选择。为编写在某个特定系统上实现某个特定算法的程序，程序员可以参考编程手册以便找到所需要的那些细节。关键之处在于，如果程序员知道一个程序能做些什么，找出如何这样做就简单了。

虽然程序员需要概念性地了解 API，但他们的主要精力应集中在学习构造通信程序的各种方法上，而不是记住某个特定 API 的细节。

6.3 客户体系结构

由于几个原因，作为客户的应用程序要比作为服务器的应用程序简单。首先，大多数的客户软件在与多个服务器进行交互时，不必明显地处理并发性。其次，大多数客户软件像常规的应用程序那样执行。客户软件不像服务器软件，它一般不要求一定的特权，这是因为它一般不访问特权协议端口。第三，大多数客户软件不需要强行保护。它可以依赖操作系统自动地强迫执行保护。实际上，

设计和实现客户软件是很容易的，有经验的程序员可以很快地学会编写基本的客户应用程序。下面几节将对客户软件进行一般讨论；再后面的几节将把注意力集中在使用 UDP 的客户和使用 TCP 的客户之间的区别上。

6.4 标识服务器的位置

客户软件可以使用多种方法找到某个服务器的 IP 地址和协议端口号。客户可以：

- 在编译程序时，将服务器的域名或 IP 地址说明为常量
- 要求用户在启动程序时指定服务器
- 从稳定的存储设备中获得关于服务器的信息（例如，从本地磁盘中的某个文件）
- 使用某个单独的协议来找到服务器（例如，组播或广播所有服务器都要响应的报文）

把服务器的地址指明为常量可以使客户软件执行得更快，并且可以使它对某个特定本地计算环境的依赖更小。然而，这也意味着在服务器被移动后，客户软件必须重新编译。更重要的是，这还意味着当更换一个服务器时，哪怕是暂时做测试之用，这个客户也不能使用。作为一种折衷，有些客户固定一个机器名，而不是一个 IP 地址。这种方法将绑定（binding）推迟到运行的时候。它允许某个网点为服务器选择一个类属名（generic name），并为这个名字在域名系统中增加一个别名。使用别名允许一个网点的管理员改变服务器的地址而不必改变客户软件。为了移动服务器，管理员只需要改变别名。例如，可以为 mailhost 在本地域中增加一个别名，并且让所有的客户查找字符串“mailhost”以取代某个特定的机器。因为所有的机器都引用这个类属名而不是某个机器，系统管理员可以改变邮件主机的位置而不必重新编译客户软件。

把服务器的地址存放在一个文件中可使客户软件更加灵活，但这意味着如果没有这个文件，客户软件就不能执行。因此，客户软件不能容易地转移到其他的机器中。

尽管在本地的小环境下，使用广播协议来发现服务器这种方法是可行的，但这种方法却不适合于大的互联网。此外，使用一种动态的查找机制会使客户和服务器两方面都招致额外的复杂性，同时还会对网络增加额外的业务量。

为避免不必要的麻烦和对计算环境的依赖，大多数客户使用一种简单的方式解决服务器的规约问题：在启动客户程序时，要求用户提供能标识服务器的参数。让客户软件接受作为服务器地址的参数，按这种方法构建客户软件可以使它更具一般性，并且消除了对计算环境的依赖。

允许用户在调用客户软件时指明服务器地址，可以使客户软件更具一般性，并且使改变服务器位置成为可能。

需要指出的一个要点是：用参数指明服务器的地址带来了最大的灵活性。接受地址参数的程序可以同其他程序相组合，而那些程序可以从磁盘中获取服务器地址，也可以使用远程名字服务器发现地址，或者通过广播协议查找地址。因此：

若客户软件把服务器地址作为参数来接受，那么，构建这样的客户软件使构建软件的扩展版本变得容易，这些扩展版本的软件可以用其他的途径找到服务器地址（例如，从磁盘中的文件中读取）。

有些服务要求显式服务器，而有些则可以使用任一可供使用的服务器。例如，当用户启动远程

登录客户时，在他心目中有一个特定的目标机器，而登录到别的机器往往没有意义。然而，如果用户仅仅想知道当前的时间，他不在乎到底是哪个服务器在响应。为适合这种服务，设计者可以修改以上所讨论的任何一种服务器查询方法，他们可以提供一组服务器名而不是单个服务器名。客户也必须改动，以便尝试一组服务器中的每一个，直到找到一个能响应的服务器为止。

6.5 分析地址参数

用户在调用客户程序时，常常在命令行中指明一些参数。在大多数系统中，传递给客户程序的每个参数由字符串构成。客户使用一种参数语法来解释其意义。例如，大多数客户软件既允许用户提供机器（服务器运行于其上）的域名：

```
merlin.cs.purdue.edu
```

也允许用户提供点分十进制表示的IP地址：

```
128.10.2.3
```

为确定用户是指明了IP地址还是名字，客户扫描这个参数，查看它是否含有字母。若含有，它一定是个名字。若它只含有数字和小数点，客户假设它是点分十进制数字地址，并按照这种方式对其进行语法检查。

当然，除服务器机器名或IP地址外，客户程序有时需要另外的信息。具体地说就是，全参数化的客户软件允许用户指明协议端口号和机器。为这样做，我们可以使用一个附加的参数或者在一个字符串中放入此信息的编码。例如，若要为某台机器上的SMTP服务指明相关联的协议端口，且该机器取名为merlin.cs.purdue.edu，则客户可以接受两个参数：

```
merlin.cs.purdue.edu smtp
```

或者将机器名和协议端口结合进单个参数：

```
merlin.cs.purdue.edu:smtp
```

尽管每个客户可以独立地选择其参数的语法细节，但让许多客户拥有各自一套语法就会引起混淆。从用户的观点看，一致性总是重要的。因此，我们劝告程序员：对客户软件，要遵循它们本地系统所使用的约定。例如，许多Linux实用程序使用单独的参数指明服务器的机器地址和协议端口。

6.6 查找域名

客户必须用sockaddr_in结构指明服务器的地址。这意味着将点分十进制表示的地址转换为用二进制表示的32比特IP地址。把点分十进制表示法转换为二进制表示法是很简单的。然而，要转换域名就相当费事了。套接字API包含了库例程(library routine)inet_addr和gethostbyname，这两个例程执行这种转换^①。inet_addr接受一个字符串，该字符串含有一个点分十进制表示的地址，而返回一个等价的二进制表示的地址。gethostbyname接受一个ASCII字符串，该字符串含有某台机器的域名，它返回一个hostent结构，该结构含有二进制表示的主机IP地址，当然还有一些其他的内

容。`hostent` 在文件 `netdb.h` 中声明：

```
struct hostent {
    char      *h_name;        /* official host name */
    char      **h_aliases;    /* other aliases */
    int       h_addrtype;     /* address type */
    int       h_length;       /* address length */
    char      **h_addr_list;  /* list of addresses */
};

#define h_addr h_addr_list[0]
```

包含名字和地址的字段一定是表(`list`)，这是因为拥有多个接口的主机也拥有多个名字和地址。为了与早先的版本兼容，文件还定义了标识符 `h_addr`，它指向主机地址表中的第一个位置。这样，程序就可以把 `h_addr` 作为结构中的一个字段来使用。

我们来考虑名字转换的简单例子。假设某个客户已经收到传递来的 一个字符串形式的域名 `merlin.cs.purdue.edu`，但它需要获得 IP 地址。客户就可以按如下方式调用 `gethostbyname`：

```
struct hostent *hptr;
char  *examplenam="merlin.cs.purdue.edu";

if ( hptr = gethostbyname( examplenam ) ) {
    /* IP address is now in hptr->h_addr */
} else {
    /* error in name-handle it */
}
```

若调用成功，`gethostbyname` 返回一个合法的 `hostent` 结构指针。若名字不能映射为某个 IP 地址，该调用就返回零。因此，客户程序检查 `gethostbyname` 返回的值以确定是否有差错发生。

6.7 由名字查找某个熟知端口

多数客户程序必须查找它们想要调用的特定服务的协议端口。例如，SMTP 邮件服务器的客户需要查找分配给 SMTP 的熟知端口。为此，客户调用库函数 `getservbyname`，它有两个参数：一个字符串，指明所期望的服务；还有一个字符串，指明所使用的协议。该函数返回 `servent` 类型的结构指针。该结构也包含在文件 `netdb.h` 中：

```
struct servent {
    char      *s_name;        /* official service name */
    char      **s_aliases;    /* other aliases */
    int       s_port;         /* port for this service */
    char      *s_proto;       /* protocol to use */
};

.
```

① 有些库例程返回的结果是指向静态数据结构的指针，这对线程来说是不安全的；第 12 章讨论了线程执行和库例程的关系。

若某个 TCP 客户需要查找 SMTP 的正式协议端口号，它便调用 getservbyname，如下例：

```
struct servent *sptr;

if (sptr = getservbyname("smtp", "tcp")){
    /* port number is now in sptr->s_port */
} else {
    /* error occurred-handle it */
}
```

6.8 端口号和网络字节顺序

函数 getservbyname 按网络字节顺序返回服务的协议端口。第 5 章解释了网络字节顺序的概念，还描述了用来将网络字节顺序转换为本地机器所使用的字节顺序的库例程。getservbyname 返回的端口号值的形式正是使用 sockadd_in 结构所需要的形式，但这种表示也许和本地机器经常使用的表示方法不一致。因此，如果一个程序打印 getservbyname 返回的值，但没有将其转换为本地字节顺序，这时可能会显示不正确的结果。

6.9 由名字查找协议

套接字接口提供了一种机制，允许客户或服务器将协议名映射为分配给该协议的整数常量。库函数 getprotobynam 执行这种查找。调用 getprotobynam 时以一个字符串参数的形式传递协议名，它返回一个 protoent 类型的结构的地址。若 getprotobynam 不能访问数据库或所指明的名字不存在，该函数就返回零。协议名数据库允许网点为每个名字定义别名。protoent 结构拥有一个字段针对正式的协议名，还有一个字段指向别名表（list of aliases）。C 的 include 文件 netdb.h 中含有该结构的声明：

```
struct protoent {
    char    *p_name;      /* official protocol name      */
    char    **p_aliases;   /* list of aliases allowed      */
    int     p_proto;      /* official protocol number    */
};
```

若某个客户需要查找 UDP 的正式协议号，它可以按下例调用 getprotobynam：

```
struct protoent *pptr;

if (pptr = getprotobynam("udp")){
    /* official protocol number is now in pptr->p_proto */
} else {
    /* error occurred-handle it */
}
```

6.10 TCP 客户算法

构建客户软件往往较构建服务器软件容易。因为TCP处理了全部的可靠性和流量控制问题，所以在所有网络编程工作中，构建使用TCP的客户程序是最简单不过的。客户按照算法6.1构造与某个服务器的连接并与该服务器通信。下面的几节将按照该算法，详细地讨论每一步骤。

算法 6.1

1. 找到期望与之通信的服务器的IP地址和协议端口号。
2. 分配套接字。
3. 指明此连接需要在本地机器中的、任意的、未使用的协议端口，并允许TCP选择一个这样的端口。
4. 将这个套接字连接到服务器。
5. 使用应用级协议与服务器通信（在此，往往包含发送请求和等待应答）。
6. 关闭连接。

算法 6.1 面向连接的客户。客户应用程序分配套接字并将它与某个服务器连接。接着，它通过该连接发送请求并接收发回的应答

6.11 分配套接字

前面几节业已讨论过用于找到服务器IP地址的方法，还讨论了用来分配通信套接字的socket函数。使用TCP的客户必须将协议族和服务分别说明为PF_INET和SOCK_STREAM。程序以include语句开始，这些语句引用了一些文件，这些文件包含了调用中要使用的常量定义以及保存套接字描述符的变量的声明。若协议族中不止一个协议，则由第一个参数指明协议族，第二个参数指明所要求的服务，而socket的第三个参数标识某个特定的协议。就Internet协议族来说，只有TCP提供SOCK_STREAM服务。因此，第三个参数就无所谓了，它就应当为零。

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

int      s;          /* socket descriptor */

s = socket( PF_INET, SOCK_STREAM, 0 );
```

6.12 选择本地协议端口号

在套接字能够用于通信之前，应用程序需要为它指明远程的和本地的端点地址。服务器运行于某个熟知协议端口之上，所有客户都需要知道该端口。然而，TCP客户并非工作于某个预分配的端口上，它必须为它的端点地址选择一个本地协议端口。一般来说，客户并不在乎它使用哪个端口，只要：(1)该端口并不与这台机器中正被其他进程所使用的端口相冲突，(2)该端口并未被分配给某个熟知服务。

当然，在客户需要一个本地协议端口时，它可以随机选择任意的端口，直到找到某个符合以上条件的端口为止。然而，套接字接口使选择客户端口简单了，这是因为它提供了一个途径，即客户可以允许 TCP 自动选择本地端口。`connect` 调用的一个副效应就是所选择的本地端口能满足上述这些准则。

6.13 选择本地 IP 地址中的一个基本问题

在构成连接端点时，客户必须选择一个本地 IP 地址以及一个本地协议端口号。对只挂在一个网络上的主机来说，选择本地 IP 是很简单的。然而，由于路由器或多接口（multi-homed）主机拥有多个 IP 地址，这就使这种选择困难了。

一般来说，选择 IP 地址之所以困难是因为正确的选择要依赖于选路信息，而应用程序却很少使用选路信息。为理解其中的原因，想像某台计算机拥有多个网络接口——因而有多个 IP 地址。在应用程序使用 TCP 之前，它必须具有这个连接的端点地址。TCP 与某个外界的目的地通信时，它将 TCP 报文段封装到 IP 数据报中，并将该数据报传递给 IP 软件。IP 使用远程目的地址和它的路由表来选择下一跳（next hop）的地址以及可以用来到达下一跳的网络接口。

这里有个问题：在外发（outgoing）数据报中的 IP 源地址，应当与网络接口的 IP 地址相匹配，IP 就是通过这个接口传送该数据报的。然而，如果应用程序随机地从机器的各 IP 地址中选择一个，它可能选择了一个与接口（IP 通过该接口传送数据报）并不匹配的地址。

在实际工作中，甚至在程序员选择了一个错误的地址时，客户可能看来还能工作，这是因为分组可能通过某个不同的（本该到达服务器的）路由转回客户。然而，使用不正确的地址违反了规约，使网络管理变得困难和混乱，还使程序的可靠性下降。

为解决这个问题，套接字调用可以让应用程序将本地地址字段放置不填，而允许 TCP/IP 软件在客户与某个服务器进行连接时自动地选取本地 IP 地址。

因为选取正确的本地 IP 地址要求应用程序与 IP 选路软件交互，TCP 客户软件往往将本地端点地址放置不填，而允许 TCP/IP 软件自动选取正确的本地 IP 地址和未使用的本地协议端口号。

6.14 将 TCP 套接字连接到某个服务器

系统调用 `connect` 允许 TCP 套接字发起连接。用下层协议的术语来说就是 `connect` 强迫执行了开始时的三次握手。除非它建立了连接，或者 TCP 到达超时门限并放弃建立连接，否则对 `connect` 的调用是不会返回的。如果连接尝试成功，该调用返回 0，否则返回 1。`connect` 有三个参数：

```
retcode = connect( s, remaddr, remaddrlen )
```

此处，`s` 是套接字的描述符，`remaddr` 是一个 `sockaddr_in` 类型的结构的地址，它指明期望与之连接的远程端点，`remaddrlen` 是第二个参数的长度（单位为字节）。

`connect` 指明四项任务。首先，它对指明的套接字进行检测，以保证它是有效的并且还没有建立连接。第二，它将第二个参数给出的端点地址填入到此套接字中。第三，若此套接字还没有本地端点地址，它便为连接选择一个（IP 地址和协议端口号）。第四，它发起一个 TCP 连接并返回一个值，以此告诉调用者连接是否成功。

6.15 使用 TCP 与服务器通信

假设 connect 成功地建立了连接，客户就可以使用这个连接与服务器进行通信。应用协议往往指明一种请求响应交互 (request-response interaction)，即客户发送一系列请求并等待对每个请求的响应。

客户常常调用 send 来发送各个请求，调用 recv 来等待响应。对最简单的协议来说，客户只发送一个请求，并且只接收一个响应。更复杂的应用协议则要求客户反复执行，发送一个请求，然后在发送下一请求前等待响应。下面的代码说明了这种请求响应交互，该代码说明了一个程序如何在 TCP 上写一个简单的请求并读取响应。

```
/* Example code segment */

#define BLEN 120      /* buffer length to use */
char *req="request of some sort";
char buf[BLEN];      /* buffer for answer */
char *bptr;          /* pointer to buffer */
int n;               /* number of bytes read */
int buflen;          /* space left in buffer */

bptr=buf;
buflen=BLEN;

/* send request */

send(s, req, strlen(req), 0);

/* read response (may come in many pieces) */

while ((n=recv(s, bptr, buflen, 0))>0){
    bptr += n;
    buflen -= n;
}
```

6.16 从 TCP 连接中读取响应

上一个例子的代码展示了客户发送一小报文到服务器，并期待一则响应（小于 120 字节）的过程。该代码含有对 send 的一次调用，但重复调用了 recv。只要调用 recv 返回了数据，代码就把缓存可用空间的计数减少，并将缓存的指针移过所接收的数据。在输入中，反复执行是必要的，即使在连接另一端的应用程序只发送了一小点数据，也应如此。这是因为 TCP 不是面向块的 (block-oriented) 协议，而是面向流的 (stream-oriented) 协议：它保证传递发送者所发出的字节序列，但是并不保证按照这些字节所写入时的组传送。TCP 可能会将一块数据分成若干片，并把每片数据在单独的报文段中传送（例如，它可能需要这样分割数据，使其每片都能填满报文段的最大容量，或者，如果接收者没有足够大的空间容纳一大块数据，它可能需要每次发送一小片数据）。另一种途

径是，TCP可能在发送报文段之前，要在其缓存中积累许多的字节（例如，为了填满一个数据报）。这样，即使发送应用程序使用一次send调用将数据传递给TCP，接收应用程序也可能接收到许多小块数据。或者，即使发送应用程序用了一串send调用将数据传递给TCP，接收应用程序却有可能接收到一大块数据。这是TCP编程中一个很基本的概念：

由于TCP并不保持记录的边界，所以从TCP连接中进行接收的任何程序都必须准备一次只接受几个字节的数据。即使在发送应用程序一次发送一大块数据时，此规则也成立。

6.17 关闭TCP连接

6.17.1 对部分关闭的需要

当某个应用程序完全结束使用一个连接时，可以调用close来从容地终止连接并释放该套接字。可是，关闭连接却很少会如此简单，这是因为TCP允许双向通信，因此，常常需要在客户和服务器之间进行协调。

为了理解这个问题，考虑一个使用上述请求响应交互的客户和服务器。客户软件重复地发送请求给进行响应的服务器。一方面，服务器不能关闭连接，因为它不知道客户是否还要发送其他的请求。另一方面，尽管客户可以知道何时将没有请求要发送了，但它不知道是否所有从服务器来的数据均已到达。后者对某些应用协议尤其重要，例如，这些协议在对某个请求的响应中有大量数据要传递的情况（例如，响应数据库查询）。

6.17.2 部分关闭的操作

为解决连接关闭问题，多数套接字接口的实现包含附加的原语，允许应用程序在一个方向上关闭TCP连接。系统调用shutdown有两个参数，即套接字描述符和方向说明，它在所指明的方向上关闭该套接字：

```
errcode = shutdown( s, direction );
```

参数direction是个整数。若其值为0，将不再允许输入。若其值为1，将不再允许输出。最后，若其值为2，连接将在两个方向上关闭。

部分关闭的优点现在就清楚了：当客户结束发送请求时，它可以使用shutdown来指明没有数据要发送了，但并不释放套接字。下层的协议向远程机器报告这个关闭，使服务器应用程序知道将要接收end-of-file信号。服务器一旦检测到end-of-file，就知道不会再有请求到达了。在发送完最后一个响应后，服务器就可以关闭这个连接。概括起来就是：

部分关闭机制可使一些应用协议消除二义性，在这些协议中，对请求的响应可能要传输不定数量的信息。在这些情况下，客户在发送最后一个请求后，可发起部分关闭；服务器在发送完最后一个响应后再关闭这个连接。

6.18 UDP客户的编程

初看起来，编写UDP客户程序看上去是项简单的工作。算法6.2是UDP客户的基本算法，同

TCP 客户的算法（算法 6.1）相似。

算法 6.2

1. 找到期望与之通信的服务器的 IP 地址和协议端口号。
2. 分配套接字。
3. 指明这种通信需要本地机器中的、任意的、未使用的协议端口，并允许 UDP 选择一个这样的端口。
4. 指明报文所要发往的服务器。
5. 使用应用级协议与服务器通信（在此，往往包含发送请求和等待应答）。
6. 关闭连接。

算法 6.2 无连接的客户。发送进程创建了连接的套接字，并使用它循环地发送一个或更多的请求。这个算法忽略了可靠性问题

UDP 客户算法的最初几步与对应的 TCP 客户算法很相像。UDP 客户先获得服务器的地址和协议端口号，然后就分配用于通信的套接字。

6.19 连接的和非连接的 UDP 套接字

客户应用程序可以按两种基本模式之一来使用 UDP：连接的和非连接的。为进入连接模式，客户使用 `connect` 调用指明远程端点地址（即服务器的 IP 地址和协议端口号）。客户一旦指明了远程端点，不用每次重复指明远程地址就可以发送和接收报文。在非连接的模式，客户并不把套接字连接到一个指定远程端点上，而是在每次发送报文时指明远程目的地。连接的 UDP 套接字的主要优点是：对那些一次只与一个服务器进行交互的常规客户软件来说很方便，应用程序只需要一次指明服务器，而不管有多少数据报要发送。非连接套接字的主要优点是其灵活性：客户可以一直等待确定要与哪个服务器联系，直到它有一个请求要发送时为止。此外，客户可以很容易地将每个请求发往不同的服务器。

UDP 套接字可以是连接的，这使得与一个指定的服务器进行交互很方便；也可以是非连接的，这就使应用程序每次发送请求时都必须指明服务器的地址。

6.20 对 UDP 使用 `connect`

尽管客户可以连接 `SOCK_DGRAM` 类型的套接字，但这个 `connect` 调用并不发起任何分组交换，也不检测远程端点地址的合法性。相反，`connect` 调用仅仅在套接字的数据结构中记录下远程端点的信息以备以后使用。因此，当把 `connect` 应用于 `SOCK_DGRAM` 上时，它仅仅保存了一个地址。即使 `connect` 调用成功了，它既不意味着远程端点地址合法，也不意味着服务器是可到达的。

6.21 使用 UDP 与服务器通信

在 UDP 客户调用 `connect` 后，它可以使用 `send` 发送报文或使用 `recv` 接收响应。与 TCP 不同的

是，UDP提供了报文传送。客户每次调用send，UDP便向服务器发送一个报文。这个报文包含了传递给send的全部数据。与之相似，每次调用recv都返回一个完整的报文。假设客户已经指明了足够大的缓存，recv调用就从下一个传入报文返回所有的数据^①。因此，UDP客户不需要为获得单个报文而重复调用recv。

6.22 关闭使用 UDP 的套接字

UDP客户调用close来关闭套接字，并将与之关联的资源释放掉。套接字一旦关闭，UDP软件将拒绝以后到达的报文，这些报文的协议端口是以前分配给该套接字的。然而，发生close的这台机器并没有通知远程端点这个套接字已被关闭。因此，使用无连接传输的应用程序必须使得远程机器知道在关闭套接字之前应把它保留多久。

6.23 对 UDP 的部分关闭

shutdown可以用于连接的UDP套接字，以便在某个给定方向上终止进一步的传输。遗憾的是，不像对TCP连接的部分关闭，当shutdown用于UDP时，它并不向另一方发送任何报文，而是仅仅在本地套接字中标明不期望在指定的方向上传输数据了。因此，如果客户在其套接字上关闭了以后的输出，服务器将收不到任何表明通信业已停止的指示。

6.24 关于 UDP 不可靠性的警告

我们的简单UDP客户算法忽略了UDP的一个基本情况，即它提供不可靠（尽最大努力）的交付语义。尽管简单的UDP算法可以在本地网络（该网络具有低的丢失率、低的时延、并且没有分组失序）中良好地工作，但在复杂的互联网络中，遵循我们这个算法的客户就不能工作了。为了能够在互联网络中工作，客户必须通过超时和重传来实现可靠性。它还必须处理分组重复或分组失序问题。增强可靠性可能是困难的，因为在协议设计中这需要有专门的技术。

使用UDP的客户软件必须利用一些技术（如分组排序、确认、超时与重传等）才能实现可靠性。为一个互联网络环境设计正确的、可靠的和高效的协议需要有相当的专门技术。

6.25 小结

客户程序是最简单的网络程序之一。客户在能够通信之前必须获得服务器的IP地址和协议端口号；为增加灵活性，客户程序常常要求用户在启动客户时指明服务器。接着，客户便将服务器的地址从点分十进制表示法转换为二进制表示法，或者使用域名系统将文本形式的机器名转换为IP地址。

TCP客户算法是简单明了的：TCP客户分配一个套接字，并将其与某个服务器连接。客户使用send向服务器发送请求，并使用recv接收应答。一旦结束使用某个连接，要么由客户，要么由服务

^① 如果用户的缓存不够大，UDP就会丢弃那些装不下的数据。

器调用 `close` 将其终止。

虽然客户必须明确指明它所期望与之通信的服务器的端点地址，但它可以允许 TCP/IP 软件选择一个未使用的协议端口号，并填入正确的本地 IP 地址。这样做避免了在如下情况中会产生的问题：在路由器或多接口主机中，当某个客户不小心选择了一个 IP 地址，但这个地址与传送业务流所要通过的接口的 IP 地址不同。

客户使用 `connect` 为套接字指明远程端点的地址。当使用 TCP 时，`connect` 采取了三次握手方式，以保证通信是可行的。当使用 UDP 时，`connect` 仅仅记录下服务器的端点地址以备后用。

如果客户和服务器都不能确切知道通信将在何时结束，那么关闭 TCP 连接可能会引起困难。为解决这个问题，套接字接口提供了 `shutdown` 原语，该原语引起部分关闭，并让另一方知道不再会有数据到达。客户使用 `shutdown` 关闭到服务器的路径；服务器在此连接上收到 `end-of-file` 信号，它指明客户业已结束。在服务器发送完最后的响应后，就使用 `close` 终止连接。

深入研究

定义了各种协议的 RFC 还针对客户代码建议了算法和实现技术。Steven [1997] 也回顾了客户的实现。

习题

- 6.1 阅读有关 `sendto` 和 `recvfrom` 两个套接字调用的资料。它们工作于使用 TCP 的套接字还是使用 UDP 的套接字？
- 6.2 编写一个程序，确定计算机在没有发送或接收任何分组时，是否使用了网络字节顺序？
- 6.3 当域名系统解析某个机器名时，返回一个或多个 IP 地址，为什么？
- 6.4 试构建一个客户软件，它使用 `gethostbyname` 查询在你的网点中的机器名，并打印所有返回的信息。哪个正式名（如果有）让你感到奇怪？你打算使用正式的机器名还是别名？描述一下别名不能正确工作的情况（如果有）。
- 6.5 测量查询一个机器名（`gethostbyname`）和查询一个服务条目（`getservent`）所要求的时间。就合法名字和非法名字重复这个测试。查询一个非法名字较查询一个合法名字要花费更长的时间吗？解释你所观察到的任何不同之处。
- 6.6 当你使用 `gethostbyname` 查找 IP 地址时，使用一个网络监视仪看看你的计算机所产生的网络通信量。为你知道的每个机器名多次进行这个实验。解释各个查找间的网络通信量的不同。
- 6.7 为测试你的机器的本地字节顺序是否和网络字节顺序一样，编写一个程序，它使用 `getservbyname` 查找针对 UDP 的 ECHO 服务，并打印查找到的协议端口号。若本地字节顺序和网络字节顺序是一样的，该值应为 7。
- 6.8 编写一个程序，它分配一个本地协议端口号，关闭该套接字，延时几秒后，再分配另一个本地端口。在空闲的机器和繁忙的分时系统中分别运行这个程序。在每个系统中，你的程序收到了哪个端口值？若它们不同，解释原因。
- 6.9 在什么环境下，客户程序可以使用 `close` 代替 `shutdown`？
- 6.10 在每次启动时，客户可以使用相同的协议端口号吗？为什么可以或者为什么不可以？

第7章 客户软件举例

7.1 引言

前面一章讨论了客户应用的基本算法，以及用于实现这些算法的特定技术。本章所给出的完整的、可以工作的客户程序的例子详细地说明了这些基本概念。这些例子使用 UDP 和 TCP。本章还将说明程序员如何构建过程库（library of procedures），这些过程隐藏了套接字调用的细节，并使构造可移植且可维护的客户软件变得容易。

7.2 小例子的重要性

TCP/IP 定义了多种服务以及访问这些服务的许多标准应用协议。这些服务从简单（例如，只用于测试协议软件的字符发生器服务）到复杂（例如，提供鉴别和保护功能的文件传送服务）。本章和下面几章集中讨论具有简单服务的客户 - 服务器实现。后面几章将回顾几个具有复杂服务的客户 - 服务器应用。

在例子中所使用的协议可能看上去没有提供什么有意思或者有用的服务，但研究这些例子是很重要的。首先，由于服务本身所需的代码很少，这样，实现这些服务的客户和服务器软件易于理解。更重要的是，小规模的程序突出了基础算法，并清楚地说明了客户和服务器程序是如何使用系统调用的。其次，通过研究这些简单的服务，可以为读者提供一种直觉，这种直觉可以告诉读者服务代码的相对长短及其所提供的服务的数量。在那些讨论设计多重协议或多重服务的必要性的章节，对小服务具有一定的直觉理解尤其重要。

7.3 隐藏细节

大多数程序员都理解将一个复杂的大程序分解成一组过程的好处：一个模块化的程序要比一个等价的单个程序容易理解、排错和修改。如果程序员认真地设计了过程，他们还可以在其他程序中重新使用这些过程。另外，仔细选择过程还可以使程序能容易地移植到新的计算机系统中。

从概念上说，过程通过将细节隐藏起来，提高了程序员所使用的语言的级别。程序员由于使用大多数编程语言中所提供的那些低级设施，都感到这样的编程是单调的，并且容易出错。他们还发现，他们在所写的每个程序中都重复编写了许多基本的程序段代码。使用过程（以其所提供的较高级的操作）可帮助他们避免这种重复。某个特定算法的代码一旦编入某个过程，程序员就可以在许多程序中使用它们，不再需要考虑实现的细节。

在构建客户和服务器程序时，小心地使用过程尤其重要。首先，因为网络软件包含了对某些条目的声明，如端点地址，所以，构建使用网络服务的程序包括了一大堆枯燥的细节，这些细节是在常规程序中所找不到的。使用过程来隐藏这些细节将减少出错的机会。其次，多数代码需要分配一

个套接字、绑定地址并构成网络连接，这些操作在每个客户程序中都要重复出现；将这些代码置于过程中就可以允许程序员重用这些代码而不是再重写一遍。第三，因为 TCP/IP 是为异种机互连而设计的，网络应用程序常常运行于许多不同机器的体系结构上。程序员可以用过程将依赖于操作系统的内容隔离出来，以便容易地将代码移植到新机器中。

7.4 针对客户程序的过程库例子

为理解过程使编程工作变得更容易些的方法，我们来考虑构建客户程序的问题。为与某个服务器建立联系，客户必须选择一个协议（像 UDP 或 TCP）、查找服务器的机器名、查找所期望的服务并将其映射到协议端口号、分配套接字并与之连接。对每个应用来说，为以上每个步骤从头编写代码是浪费时间的。而且，如果程序员需要改变任何一处细节，他们必须修改各个应用程序。为使编程时间尽量减少，程序员可以一次编写代码，将其置于某个过程之中，然后，只是简单地在各个客户程序中调用这个过程就行了。

设计过程库的第一步是抽象：程序员必须想像那些使编写程序更简单的高级操作。例如，某个应用程序员也许会想像两个过程，它们处理分配和连接套接字的工作：

```
socket = connectTCP( machine, service );
```

和

```
socket = connectUDP( machine, service );
```

理解这点是很重要的，即这里所给出的并不是一个关于“正确的”抽象的处方，而是仅仅给出了对这样一个情况的一种可能的处理方式。重要概念是：

过程的抽象允许程序员定义高级操作，在各应用程序之间共享代码，并且减少在微小细节上出错的机会。我们在本书中所使用的这些例子仅仅说明了一种可能的方法；程序员应自由选择他们自己的抽象。

7.5 connectTCP 的实现

由于在我们建议的过程中，无论是 connectTCP 还是 connectUDP，都需要分配套接字并填入基本信息，我们决定将所有的低级代码放置到第三个过程 connectsock 中，这样，实现两个高级操作就成了简单的调用了。文件 connectTCP.c 说明了这一概念：

```
/* connectTCP.c - connectTCP */

int      connectsock(const char *host, const char *service,
                     const char *transport);

/*
 * connectTCP - connect to a specified TCP service on a specified host
 */
```

```
/*
int
connectTCP(const char *host, const char *service )
/*
 * Arguments:
 *      host      - name of host to which connection is desired
 *      service   - service associated with the desired port
 */
{
    return connectsock( host, service, "tcp");
}
```

7.6 connectUDP 的实现

文件 connectUDP.c 说明了如何用 connectsock 建立使用 UDP 协议的一个连接的套接字。

```
/* connectUDP.c - connectUDP */

int      connectsock(const char *host, const char *service,
                     const char *transport);

/*
 * connectUDP - connect to a specified UDP service on a specified host
 *
 */
int
connectUDP(const char *host, const char *service )
/*
 * Arguments:
 *      host      - name of host to which connection is desired
 *      service   - service associated with the desired port
 */
{
    return connectsock(host, service, "udp");
}
```

7.7 构成连接的过程

过程 connectsock 中含有所有需要用来分配套接字和连接该套接字的代码。调用者要指明是创建 UDP 套接字还是创建 TCP 套接字。

```
/* connectsock.c - connectsock */

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

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

#include <netinet/in.h>
#include <arpa/inet.h>

#include <netdb.h>
#include <string.h>
#include <stdlib.h>

#ifndef INADDR_NONE
#define INADDR_NONE          0xffffffff
#endif /* INADDR_NONE */

extern int      errno;

int      errexit(const char *format, ...);

/* -----
 * connectsock - allocate & connect a socket using TCP or UDP
 * -----
 */
int
connectsock(const char *host, const char *service, const char *transport)
/*
 * Arguments:
 *   host      - name of host to which connection is desired
 *   service   - service associated with the desired port
 *   transport - name of transport protocol to use ("tcp" or "udp")
 */
{
    struct hostent    *phe; /* pointer to host information entry */
    struct servent    *pse; /* pointer to service information entry */
    struct protoent   *ppe; /* pointer to protocol information entry */
    struct sockaddr_in sin; /* an Internet endpoint address */
    int      s, type; /* socket descriptor and socket type */

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;

    /* Map service name to port number */
    if ( pse = getservbyname(service, transport) )
        sin.sin_port = pse->s_port;
    else if ( (sin.sin_port = htons((unsigned short)atoi(service))) == 0 )

```

```

        errexit("can't get \"%s\" service entry\n", service);

/* Map host name to IP address, allowing for dotted decimal */
if (phe = gethostbyname(host) )
    memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
else if ( (sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE )
    errexit("can't get \"%s\" host entry\n", host);

/* Map transport protocol name to protocol number */
if ( (ppe = getprotobynumber(transport)) == 0)
    errexit("can't get \"%s\" protocol entry\n", transport);

/* Use protocol to choose a socket type */
if (strcmp(transport, "udp") == 0)
    type = SOCK_DGRAM;
else
    type = SOCK_STREAM;

/* Allocate a socket */
s = socket(PF_INET, type, ppe->p_proto);
if (s < 0)
    errexit("can't create socket: %s\n", strerror(errno));

/* Connect the socket */
if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
    errexit("can't connect to %s.%s: %s\n", host, service,
            strerror(errno));
return s;
}

```

虽然大多数步骤简单明了，但还是有一些细节使代码看上去有点复杂。首先，C语言允许复杂的表达式。其结果是，在许多条件语句的表达式中含有函数调用、赋值、比较，所有这些都在一行中。例如，对getprotobynumber的调用出现在一个表达式中，该表达式将结果赋值给变量ppe，接着将这个结果与0比较。若这个返回值是零（即发生了差错），if语句就执行调用errexit。否则，过程将继续执行。第二，代码使用了两个由ANSI C定义的库过程：memset和memcpy^①。过程memset将给定值的字节（可以是多个字节）放置到一个存储器块中；它是将一块大结构或数组清零的最快途径。过程memcpy将字节块从存储器的一处复制到另一处而不管它的内容如何^②。connectsock使用memset在整个sockaddr_in结构中填入零，然后使用memcpy将服务器IP地址的字节复制到sin_addr中。最后，connectsock调用过程connect以连接该套接字。若有差错发生，就调用errexit。

^① UNIX的早期版本使用的名字是bzero和bcopy。

^② 函数strcpy不能用于复制IP地址，因为IP地址可能包含为零的字节，而strcpy将它解释为字符串结束（end of string）。

```
/* erexit.c - erexit */

#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>

/*
 * erexit - print an error message and exit
 */
int
erexit(const char *format, ...)
{
    va_list args;

    va_start(args, format);
    vfprintf(stderr, format, args);
    va_end(args);
    exit(1);
}
```

erexit 使用可变数量的参数，这些参数传递给 vfprintf 来输出。erexit 的输出格式遵循 printf 的约定。第一个参数指明输出的格式，剩下的参数指明按所给格式准备要打印的参数。

7.8 使用例子库

程序员一旦选用好了抽象并构建了过程库，就可以构造客户应用程序了。若这种抽象选择得好，就会使应用程序编程简单并隐藏许多细节。为说明我们的例子库是如何工作的，我们将用它来构造客户应用程序的例子。因为每个客户都访问标准的 TCP/IP 服务，因此，我们还就此说明一些更简单的应用协议。

7.9 DAYTIME 服务

TCP/IP 标准定义了一个应用协议，该协议允许用户获得当前的日期和时间，输出格式采用用户能读懂的形式。该服务正式命名为 DAYTIME 服务。

为访问 DAYTIME 服务，用户要调用客户应用程序。这个客户联系服务器以获得信息，并将该信息打印出来。尽管标准没有指明其精确的语法，但它建议了一些格式。例如，DAYTIME 可以按如下形式提供日期和时间：

weekday, month day, year, time-timezone

如

Thursday, February 22, 1996 17:37:43-PST

该标准指明，DAYTIME服务既可以针对TCP也可以针对UDP。两者都运行在协议端口13上^①。

DAYTIME的TCP版本利用TCP连接的出现来激活输出：只要一个新的连接到达，服务器就构造包含当前日期和时间的文本字符串，发送这个字符串，然后将连接关闭。这样，客户根本不用发送任何请求。实际上，标准指明，服务器必须丢弃客户发送的任何数据。

DAYTIME的UDP版本要求客户发送请求。请求由任意的UDP数据报构成。服务器只要收到数据报，它就格式化当前的日期和时间，将结果字符串放置到外发数据报中，然后将其发回客户。服务器一旦发送了应答，它便将激活这个响应的数据报丢弃。

7.10 针对DAYTIME的TCP客户实现

文件TCPdaytime.c含有访问DAYTIME服务的TCP客户的代码。

```
/* TCPdaytime.c - TCPdaytime, main */

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

extern int errno;

int TCPdaytime(const char *host, const char *service);
int errexit(const char *format, ...);
int connectTCP(const char *host, const char *service);

#define LINELEN 128

/*
 * main - TCP client for DAYTIME service
 */
int
main(int argc, char *argv[])
{
    char *host = "localhost"; /* host to use if none supplied */
    char *service = "daytime"; /* default service port */

    switch (argc) {
        case 1:
```

① TCP和UDP端口空间不受约束；某个给定的服务不要求使用相同的端口号，且通过这两个协议，并不是所有的服务均可使用。

```

        host = "localhost";
        break;

    case 3:
        service = argv[2];
        /* FALL THROUGH */

    case 2:
        host = argv[1];
        break;

    default:
        fprintf(stderr, "usage: TCPdaytime [ host [ port]]\n");
        exit(1);
    }

    TCPdaytime(host, service);
    exit(0);
}

/*
 * TCPdaytime - invoke Daytime on specified host and print results
 */
TCPdaytime(const char *host, const char *service)
{
    char      buf[LINELEN+1]; /* buffer for one line of text */
    int       s, n;           /* socket, read count */

    s = connectTCP(host, service);

    while((n = read(s, buf, LINELEN)) > 0) {
        buf[n] = '\0'; /* ensure null-terminated */
        (void) fputs(buf, stdout);
    }
}

```

注意使用 `connectTCP` 是如何简化代码的。连接一旦建立，`DAYTIME` 仅仅从这个连接中读取输入并将其打印。它不断重复地读，直到检测出文件结束的条件。

7.11 从 TCP 连接中进行读

例子 `DAYTIME` 说明了一个重要的思想：TCP 提供了一种流服务（stream service），而并不保证保持记录的边界。实际上，流模式意味着 TCP 使发送应用程序和接收应用程序分隔开了。例如，假设发送应用程序在对 `send` 的一次调用中传送了 64 字节，接着在第二次调用时又发送了 64 字节。接收应用可能在对 `read` 的一次调用中就收到了所有这 128 个字节，或者它可能在第一次调用时收到了 10 个字节，在第二次调用时收到了 100 个字节，在第三次调用时收到了 18 个字节。在一次调用中返回的字节数依赖于下层互联网络数据报的大小、可用的缓存空间以及穿越网络所遇到的时延。

因为 TCP 的流服务不能保证按写入时相同的数据块交付数据，从 TCP 连接接收数据的应用程序不能指望所有的数据能够在一次传送中交付完；它必须重复地调用 `recv`（或 `read`），直到获得了所有的数据。

7.12 TIME 服务

TCP/IP 定义了一个服务，它允许一台机器从另外一台机器获得当前的日期和时间。该服务正式命名为 TIME，而且非常简单：在一台机器中运行的某个客户程序向在另一台机器中执行的服务器发送请求。服务器只要收到请求，就从本地操作系统中获得当前的日期和时间，用标准的格式编码该信息，然后在响应中将它发送给客户。

客户和服务器可能处于不同的时区，为避免由此发生的问题，TIME 协议指定，所有时间和日期信息必须用国际标准时间（Universal Coordinated Time）^①表示，简写为 UCT 或 UT。这样，服务器在发送应答前，将其本地时间转换为国际标准时间，客户在应答到达时，又将国际标准时间转换为其本地时间。

DAYTIME 服务意在为人所用，而 TIME 服务不同与此，它意在为那些存储或维护时间的程序所使用。TIME 协议用一个 32 比特的整数指明时间，它表示从某个起始日期所经历的秒数。TIME 协议使用 1900 年 1 月 1 日午夜作为其起始点^②。

用一个整数表示时间允许计算机把时间值迅速地从一台机器传送到另一台机器上，而不必等待着将时间转换为一个文本字符串再将其转换回来。这样，TIME 服务就能使一台计算机用另一系统上的时钟设置其时间。

7.13 访问 TIME 服务

客户可使用 TCP 或 UDP 在协议端口 37 上访问 TIME 服务（在技术上，标准定义了两个单独的服务，一个针对 UDP，而另一个针对 TCP）。为 TCP 构建的 TIME 服务器利用连接的出现来激活输出。这与前面讨论的 DAYTIME 服务极为相似。客户构造到 TIME 服务器的连接，并等待着读取连接的输出。当服务器检测到新的连接时，就把当前时间作为一个整数发送出去，然后就关闭连接。客户不发送任何数据，因为服务器根本就不从连接中读数据。

客户也可以用 UDP 访问 TIME 服务。为此，客户发送仅包含单个数据报的请求。服务器并不处理这个传入数据报，而只是从中取出发送者的地址和协议端口号，以便在应答中使用。服务器将当前时间编码为一个整数，将其放在数据报中，并将此数据报发回给客户。

7.14 精确时间和网络时延

尽管 TIME 服务适用于不同的时区，但它不能处理网络的时延。如果报文从服务器到客户要走 3 秒，客户将收到比服务器慢 3 秒的时间。此外，还有更复杂的协议来处理时钟同步。但是，TIME 服务仍很流行，原因有三。第一，与时钟同步协议比较，TIME 是极其简单的；第二，大多数客户

① 国际标准时间以前称为格林尼治平均时间（Greenwich Mean Time）。

② 建议做一个练习，计算一个 32 比特整数能装下的最大日期。

在同一个局域网中联系服务器，其网络时延总共只有几毫秒；第三，除要使用那些利用时间戳来控制进程的程序外，人们并不关心他们计算机上的时钟是否有很小的误差。

在要求更高精确性的场合，改进 TIME 或使用其他协议也是可能的。提高 TIME 的精确性的最简单途径是计算一下服务器到客户的网络时延近似值，然后将此近似值加到服务器所报告的时间值上。例如，计算网络延时近似值的一种方法是：客户计算从客户到服务器，再从服务器回来这一往返时延。客户假设两个方向有相等的时延，因而取往返时延的一半作为时延的近似值。客户将此时延的近似值加到服务器所返回的时间值上。

7.15 针对 TIME 服务的 UDP 客户

文件 UDPtime.c 包含了实现针对 TIME 服务的 UDP 客户的代码。

```
/* UDPtime.c - main */

#include <sys/types.h>

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define BUFSIZE 64

#define UNIXEPOCH 2208988800UL /* UNIX epoch, in UCT secs */
#define MSG "what time is it?\n"

extern int errno;

int connectUDP(const char *host, const char *service);
int errexit(const char *format, ...);

/*
 * main - UDP client for TIME service that prints the resulting time
 */
int
main(int argc, char *argv[])
{
    char *host = "localhost"; /* host to use if none supplied */
    char *service = "time"; /* default service name */
    time_t now; /* 32-bit integer to hold time */
    int s, n; /* socket descriptor, read count */

    switch (argc) {
```

```
case 1:
    host = "localhost";
    break;
case 3:
    service = argv[2];
    /* FALL THROUGH */
case 2:
    host = argv[1];
    break;
default:
    fprintf(stderr, "usage: UDPtime [host [port]]\n");
    exit(1);
}

s = connectUDP(host, service);

(void) write(s, MSG, strlen(MSG));

/* Read the time */

n = read(s, (char *) &now, sizeof(now));
if (n < 0)
    errexit("read failed: %s\n", strerror(errno));
now = ntohl((unsigned long)now);      /* put in host byte order */
now -= UNIXEPOCH;                  /* convert UCT to UNIX epoch */
printf("%s", ctime(&now));
exit(0);
}
```

该例子代码通过发送数据报联系TIME服务。然后，它调用read等待应答并从应答中取出时间值。UDPtime一旦获得了时间，还必须将该时间转换为适合于本地机器的形式。首先，它使用ntohl将32比特值（C语言的long类型）从网络标准字节序转换为本地主机字节序；其次，UDPtime必须转换为机器的本地表示。该例子代码是为Linux设计的，同Internet协议一样，Linux将时间表示为一个32比特的整数，并将这个整数解释为秒的计数。然而，与Internet不同的是，UNIX假设了起始日期——1970年1月1日。因此，为将TIME协议的起始点转换为Linux的起始点，客户必须扣除1900年1月1日到1970年1月1日之间的秒数。例子代码使用了转换值2208988800。时间一旦转换成与本地机器兼容的表示后，UDPtime就可以调用库过程ctime，该过程将时间值转换为一种人们可读的形式进行输出。

7.16 ECHO 服务

TCP/IP服务为UDP和TCP都指明了一种ECHO服务。初看起来，ECHO服务看起来几乎没有什么用处，因为ECHO服务器仅返回它从客户处收到的所有数据。尽管是这样简单，但ECHO服务是网络管理员测试可达性、调试协议软件以及识别选路问题的重要工具。

TCP ECHO 服务指明，服务器必须接受传入连接请求，从连接中读取数据，然后在该连接上将数据写回，如此进行，直到客户终止传送。而与此同时，客户发送输入数据，然后读取返回的数据。

7.17 针对 ECHO 服务的 TCP 客户

文件 TCPEcho.c 包含了针对 ECHO 服务的简单客户程序。

```
/* TCPecho.c - main, TCPecho */

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

extern int errno;

int TCPecho(const char *host, const char *service);
int errexit(const char *format, ...);
int connectTCP(const char *host, const char *service);

#define LINELEN 128

/*
 * main - TCP client for ECHO service
 */
int
main(int argc, char *argv[])
{
    char *host = "localhost"; /* host to use if none supplied */
    char *service = "echo"; /* default service name */

    switch (argc) {
    case 1:
        host = "localhost";
        break;
    case 3:
        service = argv[2];
        /* FALL THROUGH */
    case 2:
        host = argv[1];
        break;
    default:
        fprintf(stderr, "usage: TCPecho [ host [ port ] ]\n");
    }
}
```

```
        exit(1);
    }
    TCPEcho(host, service);
    exit(0);
}

/*
 * TCPEcho - send input to ECHO service on specified host and print reply
 */
int
TCPEcho(const char *host, const char *service)
{
    char    buf[LINELEN+1];      /* buffer for one line of text      */
    int     s, n;                /* socket descriptor, read count   */
    int     outchars, inchars;   /* characters sent and received   */

    s = connectTCP(host, service);

    while (fgets(buf, sizeof(buf), stdin)) {
        buf[LINELEN] = '\0'; /* insure line null-terminated*/
        outchars = strlen(buf);
        (void) write(s, buf, outchars);

        /* read it back */
        for (inchars = 0; inchars < outchars; inchars+=n) {
            n = read(s, &buf[inchars], outchars - inchars);
            if (n < 0)
                errexit("socket read failed: %s\n",
                        strerror(errno));
        }
        fputs(buf, stdout);
    }
}
```

在打开连接之后，TCPEcho便进入了循环，该循环重复地读取每行输入，通过TCP连接将该行输入发送给ECHO服务器，再读取返回的数据并将其打印出来。在所有行都已发送给服务器、接收到了返回的数据并将其打印出来之后，客户就退出。

7.18 针对ECHO服务的UDP客户

文件UDPEcho.c说明了使用UDP的客户是如何访问ECHO服务的。

```
/* UDPEcho.c - main, UDPEcho */

#include <unistd.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

extern int errno;

int UDPecho(const char *host, const char *service);
int errexit(const char *format, ...);
int connectUDP(const char *host, const char *service);

#define LINELEN 128

/* -----
 * main - UDP client for ECHO service
 * -----
 */
int
main(int argc, char *argv[])
{
    char *host = "localhost";
    char *service = "echo";

    switch (argc) {
    case 1:
        host = "localhost";
        break;
    case 3:
        service = argv[2];
        /* FALL THROUGH */
    case 2:
        host = argv[1];
        break;
    default:
        fprintf(stderr, "usage: UDPecho [ host [ port] ]\n");
        exit(1);
    }
    UDPecho(host, service);
    exit(0);
}

/* -----
 * UDPecho - send input to ECHO service on specified host and print reply
 * -----
 */
int
```

```
UDPEcho(const char *host, const char *service)
{
    char buf[LINELEN+1];           /* buffer for one line of text */
    int s, nchars;                /* socket descriptor, read count */

    s = connectUDP(host, service);

    while (fgets(buf, sizeof(buf), stdin)) {
        buf[LINELEN] = '\0';      /* insure null-terminated */
        nchars = strlen(buf);
        (void) write(s, buf, nchars);

        if (read(s, buf, nchars) < 0)
            errexit("socket read failed: %s\n",
                      strerror(errno));
        fputs(buf, stdout);
    }
}
```

这个UDPECHO客户的例子遵循了和TCP版本一样的--般性算法。它重复地读取输入行，将其发送给服务器，再读取由服务器返回的这些数据，并将其打印出来。UDP与TCP版本之间的最大区别在于它们如何处理从服务器收到的数据。因为 UDP 是面向数据报的，客户将一个输入行作为一个单元，将其放置在单独的数据报中。与此类似，ECHO服务器接收并返回整个数据报。因此，TCP客户把传入数据作为字节流来读取，而 UDP 客户要么收到了由服务器返回的整个行，要么什么都没有收到；除非出现差错，否则每次调用 read 都返回整个行。

7.19 小结

程序员利用过程抽象的方法可使程序灵活，并且易于维护和易于隐藏细节，还可使程序易于移植到新的计算机中。在程序员编写和调试完一个过程后，他或她将该过程放在一个库中，这个库可以容易地在许多程序里重新使用。过程库对使用 TCP/IP 的程序来说特别重要，因为这些程序往往运行于多台计算机上。

本章给出了一个过程库的例子，可以用来创建客户软件。我们的库中有两个主要过程：connectTCP和connectUDP，它们使得为某台指定主机上的某个指定服务分配和连接一个套接字更为简单。

本章给出了几个客户应用的例子。每个例子都包含了实现某个标准应用协议的全部C程序，这些应用协议是：DAYTIME（用于获得当日时间并按人们可读的格式打印）、TIME（用于获得32比特整数形式的时间）以及 ECHO（用于测试网络连通性）。这些例子代码展示了一个过程库怎样隐藏与套接字分配有关的许多细节，还展示了怎样使编写客户软件更加容易。

深入研究

在这里所描述的应用协议都是 TCP/IP 标准的一部分。Postel [RFC 867] 含有 DAYTIME 协议的标准，Postel 和 Harrenstien [RFC 868] 包含了 TIME 协议的标准，Postel [RFC 862] 包含了 ECHO 协议的标准。Mills [RFC 1305] 说明了网络时间协议（Network Time Protocol，NTP）的第三版。

习题

- 7.1 用程序 TCPdaytime 联系多台机器中的服务器。它们各自怎样格式化时间和日期？
- 7.2 Internet 标准用一个 32 比特的整数表示时间，该整数给出自起始点（1900 年 1 月 1 日午夜）所经历的秒数。多数 UNIX 系统也用以秒为单位的一个 32 比特整数表示时间，但把 1970 年 1 月 1 日作为起点。各个系统所能表示的最大时间和日期是什么？
- 7.3 增强 TIME 客户的功能，使它能够检查接收到的数据，看看它是否比 1996 年 1 月 1 日大（或者其他你所知道的最近日期）？
- 7.4 修改 TIME 客户使它计算 E 值，E 是客户发送请求和它接收到响应之间所逝去的时间。把服务器所发送的时间值加上 E 的二分之一。
- 7.5 构建一个 TIME 客户程序，用它联系两个 TIME 服务器，并报告这两个服务器所返回的时间的差异。
- 7.6 如果程序员将 TCPECHO 客户的行的大小改变为任意长度（例如，20 000），解释这样如何会发生死锁。
- 7.7 本章所给出的 ECHO 客户没有验证它们从服务器所收到的文本是否和它们所发送出去的文本一样。修改程序，使它们验证接收到的数据。
- 7.8 本章所给出的 ECHO 客户没有对它所发送和接收的字符进行计数。如果服务器错误地发回了一个客户并没有发送的字符，这时会发生什么？
- 7.9 本章中 ECHO 客户的例子没有使用 shutdown。修改代码，使用 shutdown 关闭连接。
- 7.10 针对上题，解释为什么使用 shutdown 可以提高客户性能？
- 7.11 重写 UDPecho.c 的代码，使它生成一报文，发送该报文并对应答计时，通过这种方法来测试可达性。若应答在 5 秒内未到达，它就声明该目的主机不可达。要保证在互联网络碰巧丢失了一个数据报时，至少能重发一次请求。
- 7.12 重写 UDPecho.c 的代码，使它每秒产生并发送一个新报文。检查应答以保证它们与所发送的相匹配，只报告各个应答往返所用的时间，不必打印报文的内容。
- 7.13 解释在下列情况下 UDPecho 会发生些什么：由客户发往服务器的某个请求重复了；由服务器发往客户的某个响应重复了；由客户发往服务器的某个请求丢失了；由服务器发往客户的某个响应丢失了。修改代码，使其能处理以上各种情况。

第8章 服务器软件设计的算法和问题

8.1 引言

本章考虑服务器软件的设计。在此，我们讨论一些基本问题，包括：无连接的和面向连接的服务器的访问，无状态的和有状态的服务器的应用，以及循环和并发服务器的实现。本章描述了每种方法的优点，还给出了一些适用场合的例子，在这种场合下，某种方法是合适的。后面几章将通过一些完整的服务器的例子来说明这些概念，每个例子都实现一个基本设计想法。

8.2 概念性的服务器算法

从概念上说，各个服务器都遵循一种简单的算法：创建一个套接字，将它绑定到一个熟知的端口上，并期望在这个端口上接收请求，接着便进入无限循环，在该循环中，服务器接受来自客户的下一请求，处理这一请求，构造应答，然后将这个应答发回给客户。

但是，这个并不复杂的、概念性的算法只适用于最简单的服务。为理解其中的道理，我们考虑像文件传送这样的服务，它在处理每一请求时，要求有相当可观的时间。假设联系该服务器的第一个客户要求传送一个巨大的文件（比如有 200 兆字节），而联系到该服务器的第二个客户要求传送一个小文件（比如 20 字节）。若服务器一直等到第一个文件传送完毕才考虑传送第二个文件，那么，第二个客户就将为了一个小文件的传送而等待一段不合理的时间。因为请求很小，第二个用户期望可以得到立即处理。大多数实用的服务器确实能迅速地处理小请求，因为这些服务器一次可以处理多个请求。

8.3 并发服务器和循环服务器

我们用术语循环服务器（iterative server）描述在一个时刻只处理一个请求的一种服务器实现。用术语并发服务器（concurrent server）描述在一个时刻可以处理多个请求的一种服务器。事实上，多数服务器并没有用于同时处理多个请求的冗余设施，而是提供一种表面上的并发性，方法是依靠多个执行线程，每个线程处理一个请求。我们会看到，用其他方法实现并发也是可行的——选择什么方法取决于应用协议。具体讲，如果服务器相对它所执行的 I/O 来说，只执行了一小点计算，那么，用一个单执行线程来实现是可能的，这个单线程使用异步 I/O，以便允许同时使用多个通信通道。从客户的角度看，服务器看上去就像在并发地与多个客户通信。这里的要点就是：

并发服务器这个术语是指服务器是否并发地处理多个请求，而不是指下层的实现是否使用了多个并发执行线程。

一般来说，并发服务器更难设计和构建，其最终的代码也更复杂并且难于修改。然而，大多数

程序员还是选择了并发实现的方法，因为循环服务器会在分布式应用中引起不必要的时延，而且可能会成为影响许多客户应用程序的性能瓶颈。我们概括如下：

使用循环方法实现的服务器易于构建和理解，但这样的结果会使其性能很差，因为这样的服务器要使客户等待服务。相反，以并发方法实现的服务器难于设计和构建，但却有较好的性能。

8.4 面向连接的和无连接的访问

连接性(connectivity)问题是传输协议的中心，而客户使用这个传输协议访问某个服务器。TCP/IP 协议族给应用提供了两种传输协议，TCP 提供了一种面向连接的传输服务，而 UDP 提供了一种无连接的服务。因此，由定义可知，使用 TCP 的服务器是面向连接的服务器，而那些使用 UDP 的服务器是无连接的服务器^①。

尽管我们将这个术语用到了服务器上，但如果我们将其限制在应用协议上则会更准确些，因为，在无连接的实现和面向连接的实现之间进行选择依赖于应用协议。在设计上使用面向连接的传输服务的应用协议，当实际中使用了无连接的传输协议时，也许会不能正确地运行或者是不能有效地运行。概括起来就是：

当我们考虑各种服务器实现策略的优缺点时，设计者必须记住，所使用的应用协议可能会限制某些或者所有的选择方案。

8.5 传输协议的语义

TCP 和 UDP 是 TCP/IP 协议族的两个主要传输协议，它们在很多方面是不同的。我们已讲过，TCP 提供面向连接的服务，UDP 提供无连接的服务。然而，两者最大的不同来自它们提供给应用的语义。

8.5.1 TCP 语义

点到点通信 (Point-To-Point Communication)。我们讲过，TCP 只提供给应用面向连接的接口。TCP 连接是点到点的，因为它只包括两个点——客户应用程序在一端，服务器在另一端。

建立可靠连接 (Reliable Connection Establishment)。TCP 要求客户应用程序在与服务器交换数据前，先要连接服务器，保证连接可靠建立。建立连接测试了网络的连通性，如果有故障发生，阻碍了分组到达远程系统，或者服务器不接受连接，那么，连接企图就会失败，客户就会得到通知。

可靠交付 (Reliable Delivery)。一旦建立连接，TCP 保证数据将按发送时的顺序交付，没有丢失，也没有重复。如果因为故障而不能可靠交付，发送方会得到通知。

具有流控的传输 (Flow-Controlled Transfer)。TCP 控制数据传输的速率，防止发送方传送数据的速率快于接收方的接受速率。因此，TCP 可以用于从快速计算机向慢速计算机传送数据。

双工传输 (Full-Duplex Transfer)。在任何时候，单个 TCP 连接都允许同时双向传送数据，而

^① 尽管套接字接口允许应用程序把某个 UDP 套接字连接（用 `connect` 函数）到远程端点上，但这样做只会影响到传入数据报的分用。因为 UDP 不是面向连接的协议，所以远程站点不会得到关于连接的通知，而且也不会有运输层的连接。

且不会相互影响。因此，客户可以向服务器发送请求，而服务器可以通过同一个连接发送回答。

流模式 (*Stream Paradigm*)。正如我们所见，TCP 从发送方向接收方发送没有报文边界的字节流。

8.5.2 UDP 语义

多对多通信 (*Many-To-Many Communication*)。与 TCP 不同，UDP 在可以进行通信的应用的数量上，具有更大的灵活性。多个应用可以向一个接收方发送报文，一个发送方也可以向多个接收方发送数据。更重要的是，UDP 能让应用使用底层网络的广播或组播设施交付报文。

不可靠服务 (*Unreliable Service*)。UDP 提供不可靠交付语义，即报文可以丢失、重复或者失序。它没有重传设施，如果发生故障，也不会通知发送方。

缺乏流控制 (*Lack Of Flow Control*)。UDP 不提供流控——当数据报到达的速度比接收系统或应用的处理速度快时，只是将其丢弃而不会发出警告或提示。

报文模式 (*Message Paradigm*)。我们已看到，UDP 提供了面向报文的接口，在需要传输数据时，发送方准确指明要发送的数据的字节数，UDP 将这些数据放置在一个外发报文中。在接收机上，UDP 一次交付一个传入报文。因此，当有数据交付时，接收到的数据拥有和发送方应用所指定的一样的报文边界。

8.6 选择传输协议

我们称使用 TCP 的服务器为面向连接的，称使用 UDP 的服务器为无连接的。如果我们把这两个术语限制在称呼应用协议而不是服务器上，可能会更准确，因为这里不仅仅是一个实现细节问题。对传输协议的选择取决于应用协议，如果设计上应使用 TCP 可靠交付语义的应用协议却在 UDP 上发送报文，也许不能正确或有效执行。小结如下：

由于 TCP 和 UDP 的语义极其不同，如果不考虑应用协议所要求的语义，设计者就不能在面向连接和无连接的传输协议间做出选择。

8.7 面向连接的服务器

面向连接的方法的主要优点在于易于编程。特别是，因为传输协议自动处理分组丢失和交付失序问题，服务器就不需要对这些问题操心了。面向连接的服务器只要管理和使用这些连接就行了。服务器接受来自某个客户的传入连接，然后，通过这个连接发送所有的通信数据。它从客户接收请求并发送应答。最后，在完成交互后关闭连接。

当连接保持在打开状态时，TCP 提供了所有需要的可靠性。它重传丢失的数据，验证到达的数据没有传输差错，在必要时，对传入数据进行重新排序。当客户发送请求时，TCP 要么将其可靠地交付，要么通知客户连接已经中断。与此类似，服务器可以依赖 TCP 交付响应或者通知它不能完成交付。

面向连接的服务器也有一些缺点。面向连接的设计要求对每个连接都有一个单独的套接字，而无连接的设计则允许从一个套接字上与多个主机通信。在操作系统中，套接字的分配和最终的连接的管理可能特别重要，这个操作系统必须永远运行下去而不能耗尽资源。对简单的应用来说，用于

建立和终止连接的三次握手过程使 TCP 比起 UDP 来开销要大。最重要的缺点是 TCP 在空闲的连接上根本不发送任何分组。假设客户与某个服务器建立了连接，并与之交换请求和响应，接着便崩溃了。因为客户已经崩溃了，它就不会再发送任何请求了，然而，服务器到目前为止对它收到的所有请求都已进行了响应，它便不会再向客户发送更多的数据了。在这种情况下，问题出在资源的使用上：服务器拥有分配给该连接的数据结构（包括缓存空间），并且这些资源不能被重新分配。应记住，服务器必须设计成始终在运行。如果不断有客户崩溃，服务器就会耗尽资源（比如，套接字、缓存空间、TCP 连接）从而终止运行。

8.8 无连接的服务器

无连接的服务器也有其优缺点。尽管无连接的服务器没有资源耗尽问题的困扰，但它们不能依赖下层传输提供可靠的投递。通信的一方或者双方必须要担当可靠性方面的责任。通常，如果没有响应到达，客户要承担重传请求的责任。如果服务器需要将其响应分为多个数据分组，它可能还需要实现重传机制。

通过超时和重传获得可靠性可能十分困难。实际上，这需要对协议设计具备相当的专业知识。由于 TCP/IP 运行于互联网环境中，其端到端的时延变化很快，因而使用固定的超时值将无法正常工作。许多要设计自己可靠性策略的程序员都体会到了其中的难处。应用程序只是在具有很高可靠性和很小时延变化的局域网上进行过测试，当将其转移到广域互联网（可靠性低且时延变化大）时，简单的重传超时机制就会失败。为适应互联网环境，重传策略必须具有自适应性。因此，为了能在全球因特网上正常工作，使用 UDP 的应用程序必须实现一种与 TCP 所用的一样复杂的重传机制。鉴于此，我们鼓励新程序员使用面向连接的传输。

由于 UDP 不提供可靠交付，无连接传输要求应用协议提供可靠性，并在必要时，使用一种称为自适应重传的复杂技术。为现有的应用程序增加自适应重传比较困难，它需要程序员具有相当的专业知识。

在选择无连接还是面向连接传输时，另一个要考虑的因素取决于该服务是否需要广播或组播通信。由于 TCP 只提供点到点通信，它不允许应用访问广播或组播设施（这种服务要求使用 UDP）。因此，任何一个接受或响应组播通信的服务器必然是无连接的。实际上，大多数网点都试图尽可能避免广播；目前标准的 TCP/IP 应用协议都不需要组播。但是，将来的应用可能会更多地依赖组播。

8.9 故障、可靠性和无状态

如第 2 章所述，服务器所维护的、关于它与客户正进行之交互的状态的信息，称为状态信息（state information）。没有保留任何状态信息的服务器称为无状态服务器（stateless server），而维护状态信息的服务器称为有状态服务器（stateful server）。

无状态的问题源于对确保可靠性的需求，尤其在使用无连接传输时更是如此。应记住在互联网中，信息可能重复、延迟、丢失或失序交付。如果传输协议不能保证可靠交付（UDP 就不保证可靠交付），那么应用协议的设计就必须要保证可靠。此外，实现服务器时要谨慎一些，以免无意间引入了状态依赖性（和低效性）。

8.10 优化无状态服务器

为理解优化过程所涉及的微小细节，考虑一个无连接服务器，它允许客户从存储在服务器机器磁盘上的文件中读取信息。为保持协议无状态，设计者要求每个客户请求都指定一个文件名、文件中的位置以及读取的字节数。大多数简单的服务器实现将独立地处理每个请求：它打开指定的文件，寻址到指定的位置，读取指明数量的字节，将信息发回客户，然后关闭文件。

设计服务器的聪明的程序员将注意到：(1)文件打开和关闭的额外开销较高；(2)使用该服务器的客户每次请求可能只读取十来个字节；(3)客户通常按顺序读取文件数据。此外，程序员根据经验了解到，服务器从内存缓冲区读取数据比从磁盘读数据的速度快几个数量级。因此，为优化服务器性能，程序员决定维护很小的文件信息表，如图 8.1 所示。

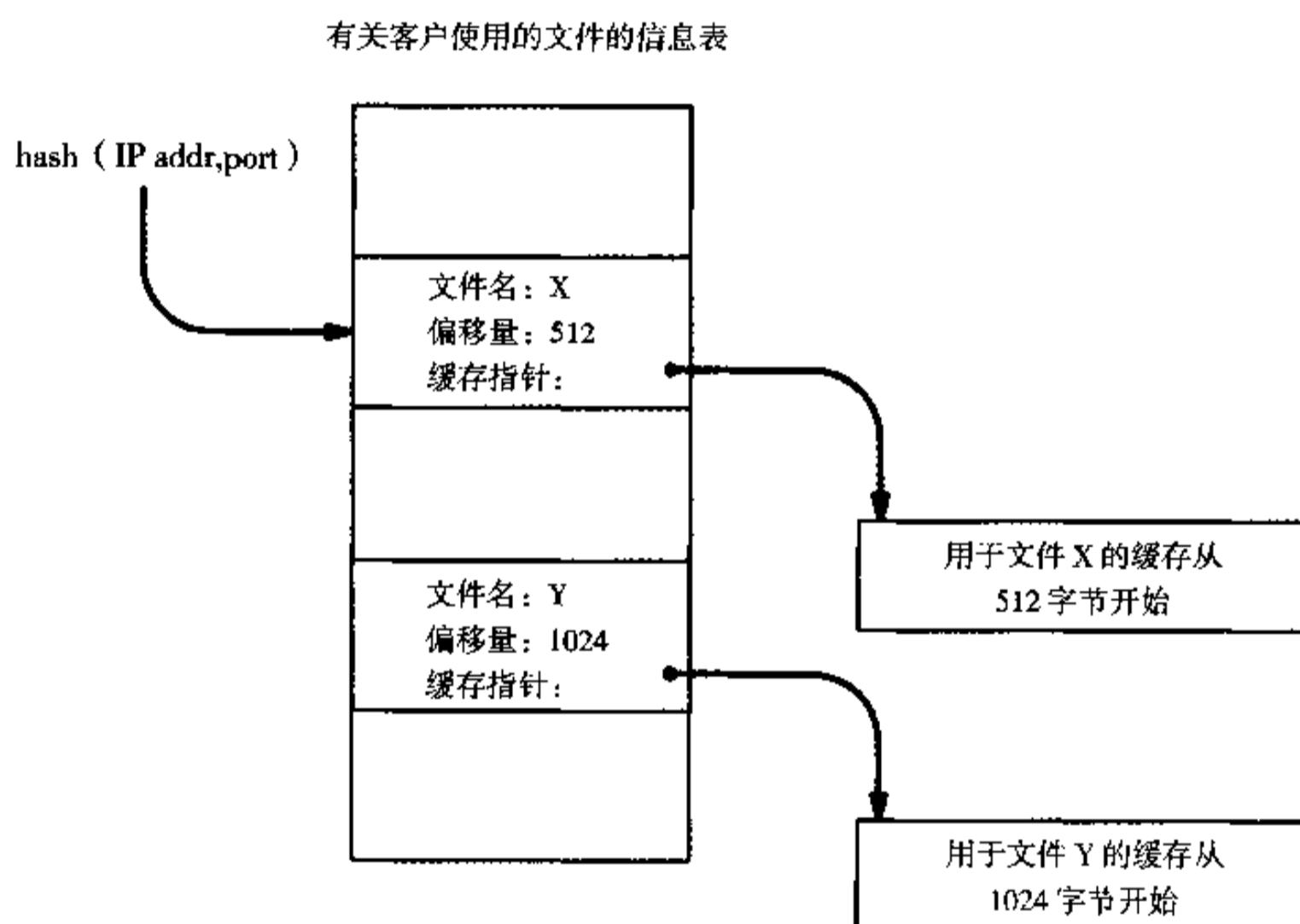


图 8.1 为提高服务器性能而保存的信息表。服务器使用客户的 IP 地址和协议端口号找到某一项。这种优化引入了状态信息

程序员把客户的IP地址和协议端口号作为表的索引来使用，并使表的每个条目包含一个指针，该指针指向大的数据缓存，缓存中的数据来自正在读取的文件。当客户发出它的第一个请求后，服务器在这个表中进行查找，结果发现没有这个客户的记录。于是，它便分配了一块大的缓存以容纳来自文件的数据，还在表中分配新的条目以指向这个缓存，打开指定的文件并把其中的数据读取到缓存中。当要构造应答时，它便从缓存中复制数据。到了下一次，收到了从同一个客户来的请求，服务器在表中发现了匹配的条目，便沿着指针找到缓存，从这个缓存中读取数据而不必打开文件。客户一旦读取了整个文件，服务器便撤消缓存以及在表中的条目，使资源可以被其他客户所使用。

当然，我们聪明的程序员会小心地构建软件，让它进行检查以便确保所请求的数据在缓存中，并且，如有必要，可以再从文件中将数据读取到缓存中。服务器还将在请求中所指明的文件与表中条目里的文件名进行比较，以便验证客户是否还在使用与前一请求相同的文件。

如果客户遵循以上所列的假设，并且程序员是仔细的，那么，在服务器中加入大的文件缓存和简单的表可以显著地提高其性能。此外，在以上给定的假设下，这个服务器优化的版本至少和最初的版本一样快，这是因为，服务器在维护数据结构上所花费的时间比从磁盘读取数据所需的时间要少。这样看来，这种优化可以提高性能而不会有任何坏处。

在服务器中加入所建议的表，就以一种微妙的方法改变了服务器，因为这种方法引入了状态信息。当然，如果选择状态信息欠仔细，就会产生差错，这表现在服务器的响应上。例如，如果服务器使用客户的 IP 地址和协议端口号来寻找缓存，但不检查文件名或请求中的文件偏移地址，重复的或者失序的请求会使服务器返回不正确的数据。但是要记住，我们说过，设计该程序优化版本的程序员是聪明的，并在服务器中编写进了检查每个请求的文件名和文件偏移地址的程序。因此，这样看来，增加状态信息是不会改变服务器响应方式的。事实上，如果程序员是小心谨慎的，协议会保持正确。如果是这样，状态信息还会带来什么坏处呢？

遗憾的是，当机器、客户程序或者网络出故障时，哪怕一点点状态信息也会使服务器表现得很糟糕。为理解其中的原因，我们考虑如下的情况：客户程序中的一个出了故障（即崩溃了），因而必须重启动。在此情况下，出现下列情况的机会是很大的：客户要求一个任意的协议端口号，而 UDP 分配给它一个新的协议端口号，它不同于分配给先前那个请求的协议端口号。当服务器收到来自客户的请求时，它不可能知道该客户业已崩溃并重启动了，所以对该文件分配了新的缓存并在表中分配了新的条目。其结果是，服务器不能知道该客户所使用的那个旧表的条目应被删除。如果服务器没有删除旧的条目，它终究会耗尽表的条目。

只要服务器在它需要新的条目时能够选择一个可删除的条目，那么在表中留有一些没用的条目看来不会引起什么问题。例如，服务器可能会选择删除最近最少使用的 LRU (least recently used) 条目的方法，这非常像用于许多虚存储器系统中的 LRU 页替换策略。然而在网络中如果有多个客户要访问单个服务器，经常性的崩溃可能会使客户支配这个表，它会在表中填上不再使用的条目。在最坏的情况下，到达的每个请求会使服务器删除一个条目并重用之。若某个客户的崩溃和重启动足够经常，它可能会使服务器删除合法客户的条目。因此，与对请求的回答相比，管理这个表和缓存要使服务器费更大的劲^①。

要点如下：

在优化无状态服务器时，程序员必须极其小心，因为，如果客户经常崩溃和重启动，或者下层的网络会使报文重复或迟延，管理少量的状态信息也会消耗资源。

8.11 四种基本类型的服务器

服务器可以是循环的或并发的，可以使用面向连接的传输或无连接的传输。图 8.2 表示这些属性把服务器划分成四种一般的类型。

① 虚存储器系统把这一现象描述为系统颠簸 (thrashing)。

循环的 无连接	循环的 面向连接
并发的 无连接	并发的 面向连接

图 8.2 服务器的四种基本类型，由是否提供并发性以及是否使用面向连接的传输作为标准

8.12 请求处理时间

一般来说，循环方法实现的服务器只够最简单的应用协议使用，因为它们使各个客户按顺序等待。循环方法实现的服务器能否满足要求则取决于所需的响应时间（这可以在本地或全局网进行测量）。

我们定义服务器的请求处理时间（request processing time）为服务器处理单个孤立的请求所花费的时间，我们还定义客户的观测响应时间（observed response time）为客户发送请求至服务器响应之间的全部时延。很明显，客户观测响应时间决不可能小于服务器的请求处理时间。然而，如果服务器有一个队列的请求要处理，观测响应时间就会比请求处理时间大得多。

循环实现的服务器一次处理一个请求。如果服务器正处理一个已经存在的请求时，另一个请求到达了，系统便将这个新的请求排队。服务器一旦处理完一个请求，它便查看队列中是否有新的请求需要处理。若 N 代表请求队列的平均长度，对刚刚到达的请求来说，它的观测响应时间大约是 $N/2 + 1$ 个服务器请求处理时间。因为观测响应时间的增长与 N 成比例，所以，大多数实现中将 N 限制为一个很小的值（比如 5），而对那些小队列不能满足需要的情况，希望程序员使用并发服务器。

一个循环的服务器是否够用？看待这一问题的另一种方式是关注于服务器所必须处理的全部负载。假定一个服务器的设计能力可处理 K 个客户，而每个客户每秒发送 R 个请求，则此服务器的请求处理时间必须小于每请求 $1/KR$ 秒。如果服务器不能以所要求的速率处理完一个请求，那么它的进入等待的请求队列最终将溢出。为使可能具有很长请求处理时间的服务器避免溢出，设计者必须考虑并发实现。

8.13 循环服务器的算法

循环服务器的设计、编程、排错和修改是最容易的。因此，只要循环执行的服务器对预期的负载能提供足够快的响应时间，多数程序员会选择一种循环的设计。循环服务器往往对由无连接的访问协议所访问的简单服务工作得最好。然而，正如下一节所述，在使用循环方法实现的服务器中，使用无连接的和面向连接的传输都是可能的。

8.14 一种循环的、面向连接的服务器的算法

算法8.1给出了通过TCP面向连接的传输访问的循环服务器的算法。下面几节将详细描述各个步骤。

算法 8.1

1. 创建套接字并将其绑定到它所提供的服务的熟知端口上。
2. 将该端口设置为被动模式，使其准备为服务器所用。
3. 从该套接字上接受下一个连接请求，获得该连接的新的套接字。
4. 重复地读取来自客户的请求，构造响应，按照应用协议向客户发回响应。
5. 当与某个特定客户完成交互时，关闭连接，并返回步骤3以接受新的连接。

算法8.1 循环的、面向连接的服务器。单个执行线程一次处理一个来自客户的连接

8.15 用INADDR_ANY绑定熟知端口

服务器需要创建套接字并将其绑定到所提供的服务的熟知端口上。如同客户一样，服务器使用过程getportbyname将服务名映射到相应的熟知端口号上。例如，TCP/IP定义了ECHO服务。实现ECHO的服务器利用getportbyname将字符串“echo”映射到指派的端口7。

当bind为某个套接字指明某个连接端点时，它使用了结构sockaddr_in，该结构中含有IP地址和协议端口号。因此，对一个套接字，bind不能只指明协议端口号而不指明IP地址。但是，选择指明的IP地址，使服务器在此地址上接受连接，会引起些困难。对只有一个网络连接的主机来说，选择地址是很明显的，因为该主机只有一个IP地址。然而，路由器或多接口机拥有多个IP地址。如果服务器在将套接字绑定到某个协议端口号时，若它指明了某个特定的IP地址，这个套接字将不接受客户发到该机器其他IP地址上的通信内容。

为解决这个问题，套接字接口定义了一个特殊的常量——INADDR_ANY，它可以代替IP地址。INADDR_ANY指明了一个通配地址(wildcard address)，它与该主机的任何一个IP地址都匹配。使用INADDR_ANY使得在多接口机上的单个服务器可以接受这样的通信，即传入数据的目的地址是该主机的任一个IP地址。概括起来就是：

当为套接字指明本地端点时，服务器使用INADDR_ANY以取代某个特定的IP地址，这就允许套接字接收发给该机器的任一个IP地址的数据报。

8.16 将套接字置于被动模式

使用TCP的服务器调用listen将套接字置于被动模式。listen还有一个参数用来指明该套接字的内部请求队列的长度。这个请求队列保存着一组TCP传入连接请求，这些连接请求来自客户，每个客户都向这个服务器请求了一个连接。

8.17 接受连接并使用这些连接

TCP服务器调用accept获得下一个传入连接请求（即把它从请求队列中取出）。该调用返回用于新的连接的套接字描述符。服务器一旦接受了新的连接，它就使用read获得来自客户的应用协议请求，并使用write发回应答。最后，服务器一旦结束使用这个连接，便调用close释放该套接字。

8.18 循环的、无连接的服务器的算法

我们还记得，循环服务器对那种具有小的请求处理时间的服务工作得最好。因为像TCP这样的面向连接的传输协议，要比像UDP这样的无连接的传输协议具有更高的额外开销，所以多数循环服务器使用无连接的传输。算法8.2给出了使用UDP的循环服务器的一般算法。

为循环的、无连接的服务器创建套接字，这个过程与面向连接的服务器是一样的。该服务器的套接字将保持无连接的，而且可以接受来自任何客户的传入数据报。

算法8.2

1. 创建套接字并将其绑定到所提供的服务的熟知端口上。
2. 重复地读取来自客户的请求，构造响应，按照应用协议向客户发回响应。

算法8.2 循环的、无连接的服务器。单个线程

一次处理一个来自客户的请求（数据报）

8.19 在无连接的服务器中构造应答

套接字接口提供了两种指明远程端口的方式。第6章和第7章讨论了客户如何使用connect来指明某个服务器的地址。在客户调用connect之后，它可使用send或write发送数据，因为套接字的内部数据结构包含了远程端点地址以及本地端点地址。然而，无连接的服务器不能使用connect，因为这样做会限制套接字，使其只能与一个特定的远程主机和端点通信，服务器也不能再使用该套接字接收来自任意客户的数据报。因此，无连接的服务器使用一个非连接的套接字。它明确地产生应答的地址，并且使用套接字调用sendto，该调用既指明所发送的数据报，又指明它将去的地址。sendto具有如下形式：

```
retcode = sendto( s, message, len, flags, toaddr, toaddrlen );
```

这里s是非连接的套接字，message是缓存的地址，该缓存含有要发送的数据，len指明缓存中的字节数，flags指明排错或者控制选项，toaddr是指向sockaddr_in结构的指针，该结构含有报文将发往的端点的地址，toaddrlen是一个整数，它指明地址结构的长度。

套接字调用为无连接的服务器获得某个客户的地址提供了简单的途径：服务器从收到的请求中的源地址获得应答的地址。实际上，套接字接口提供了一个调用，服务器可以使用该调用从下一个到达的数据报中接收发送者的地址。这个调用就是recvfrom，它有两个指定了两个缓存的参数。系统将到达的数据报放置在一个缓存中，还把发送者的地址放到第二个缓存中。对recvfrom的调用

具有如下形式：

```
retcode = recvfrom( s, buf, len, flags, from, fromlen );
```

这里参数 *s* 指明了所使用的套接字, *buf* 指明了缓存, 系统把收到的下一个数据报放到该缓存中, *len* 指明了缓存中的可用空间, *flags* 控制指定情况的处理 (如, 只向下查看但并不从套接字中提取数据), *from* 指明了第二个缓存, 系统把源地址放置在这里, *fromlen* 指明了一个整数的地址。最初, *fromlen* 所指向的整数说明的是 *from* 缓存的长度, 当该调用返回时, *fromlen* 将包含源地址的长度 (即 *from* 缓存中之数据项的长度)。服务器在请求到达时, 用 *recvfrom* 存储在 *from* 缓存中的地址产生应答。

8.20 并发服务器的算法

将并发引入服务器中的主要原因是需要给多个客户提供快速响应时间。并发性将会缩短响应时间, 如果:

- 构造要求有相当的 I/O 时间的响应,
- 各个请求所要求的处理时间变化很大, 或
- 服务器运行在具有多个处理器的计算机上。

对第一种情况, 允许服务器并发地计算响应意味着, 即使机器只有一个 CPU, 它可以部分重叠地使用处理器和外设。当处理器忙于计算响应时, I/O 设备可以将数据传送到存储器中, 而这可能是其他响应所需要的。对第二种情况, 时间分片允许单个处理器处理那些只要求少量处理的请求, 而不必要等待处理完那些需要很长处理时间的请求。对第三种情况, 服务器在具有多个处理器的计算机上并发执行, 这可以允许一个处理器为一个请求计算响应, 而同时另一个处理器为另一个请求计算响应。实际上, 大多数并发服务器自动适应下层的硬件——硬件资源 (例如, 更多的处理器) 给得越多, 这些服务器的性能就越好。

并发服务器通过使处理和 I/O 部分重叠来达到高性能。这些服务器往往被设计成这样: 如果服务器运行在提供了更多资源的硬件上, 它们的性能会自动提高。

8.21 主线程和从线程

尽管服务器使用一个单执行线程达到某些并发性是可能的, 但大多数并发服务器使用多线程。它们可以划分成两类: 主线程 (master) 最先开始执行, 在熟知端口上打开一个套接字, 等待下一个请求, 并且为处理每个请求创建一个从线程 (可能在一个新进程中)。主线程不与客户直接通信——它将这个任务交给一个从线程, 每个从线程处理与一个客户的通信。在从线程构成响应并将它发送给客户后, 这个从线程便退出。

下面几节将更详细地解释主和从的概念, 展示它们与无连接的和面向连接的服务器的关系, 并介绍其他的实现方法。

8.22 并发的、无连接的服务器的算法

并发无连接服务器的最简单的版本遵循着下面的算法 8.3。

算法 8.3

- 主 1. 创建套接字并将其绑定到所提供的服务的熟知地址上。让该套接字保持为未连接的。
- 主 2. 反复调用 `recvfrom` 接收来自客户的下一个请求，创建一个新的从线程（可能在一个新进程中）来处理响应。
 - 从 1. 从来自主进程的特定请求以及到该套接字的访问开始。
 - 从 2. 根据应用协议构造应答，并用 `sendto` 将该应答发回给客户。
 - 从 3. 退出（即，从线程在处理完一个请求后便终止）。

算法 8.3 并发的、无连接的服务器。主服务器线程接受传入请求（数据报）并为处理每个传入请求而创建一个从线程（可能在一个新进程中）

程序员应记住，尽管创建一个新线程或进程的精确开销依赖于操作系统和下层的体系结构，但这个操作可能还是很昂贵的。在无连接协议的情况下，程序员必须仔细考虑并发性的开销是否会比在速率上的获益大。实际上：

因为创建进程或线程是昂贵的，因此只有很少的无连接服务器采用并发实现。

8.23 并发的、面向连接服务器的算法

面向连接的应用协议使用连接作为其通信的基本模式。它们允许客户同服务器建立连接，在这个连接上进行通信，之后便将此连接丢弃。在大多数场合下，客户和服务器之间的连接将处理不只一个请求：协议允许客户重复地发送请求和接收响应，而不必终止这个连接或创建新的连接。因此，

面向连接的服务器在多个连接之间（而不是在各个请求之间）实现并发性。

算法 8.4 给出了并发服务器使用面向连接协议的步骤。

算法 8.4

- 主 1. 创建套接字并将其绑定到所提供的服务的熟知地址上。让该套接字保持非连接的。
- 主 2. 将该端口设置为被动模式，使其准备为服务器所用。
- 主 3. 反复调用 `accept` 以便接收来自客户的下一个连接请求，并创建新的从线程或进程来处理响应。
 - 从 1. 由主线程传递来的连接请求（即针对连接的套接字）开始。
 - 从 2. 用该连接与客户进行交互：读取消息并发送响应。
 - 从 3. 关闭连接并退出。在处理完来自客户的所有请求后，从线程就退出。

算法 8.4 并发的、面向连接的服务器。主服务器线程接受传入连接，并为每个连接创建一个从线程或进程以便对其进行处理。从线程处理完毕后，它就关闭这个连接

就像在无连接时的情况，主线程从来不同客户直接进行通信。只要新的连接一到达，主线程就创建一个从线程来处理这个连接。在从线程同这个客户进行交互时，主线程在等待着其他的连接。

8.24 服务器并发性的实现

因为 Linux 提供了两种形式的并发性——进程和线程，所以有两种常见的主—从模式实现。一种是服务器创建多个进程，每个进程都有一个执行线程。另一种实现是，服务器在一个进程中创建多个执行线程。图 8.3 说明了这两种形式。

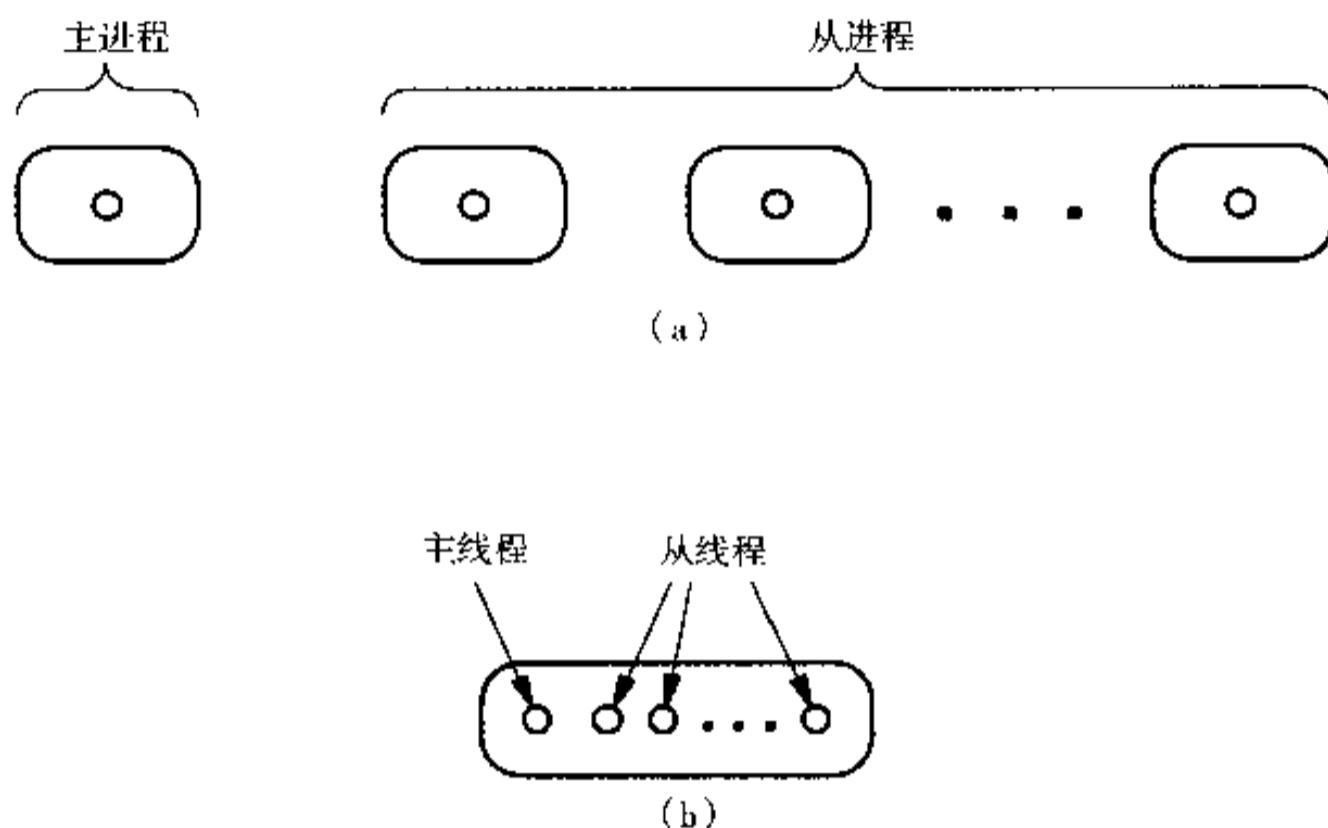


图 8.3 主—从概念的两种实现。(a) 多个单线程进程；(b) 一个进程包含多个执行线程

第 11 章和第 12 章将对两种实现加以说明。这两章都包含一个遵循算法 8.4 的服务器的例子。第 11 章的实现采用了图 8.3 (a) 的方法，主从都由一个单线程的进程实现。第 12 章的实现则按照图 8.3 (b)，主从都由线程实现，但所有的线程都在同一个进程里面。第 12 章对两种实现做了比较，并讨论了各自的优缺点。

8.25 把单独的程序作为从进程来使用

算法 8.4 说明了并发服务器如何为每个连接创建一个新从线程。在单线程的进程实现中，主进程是通过 fork 系统调用做到这点的。对简单的应用协议来说，单个服务器的程序就可以包含主进程和从进程所需的全部代码。在调用 fork 之后，原进程又循环回去接受下一个传入连接，而新进程成为处理这个连接的从进程。然而在有些场合，让从进程执行一个单独编写和编译的程序也许更方便。像 Linux 这样的系统可以容易地处理这种情况，因为它允许从进程在调用 fork 后再调用 execve。Execve 会用新程序的代码覆盖从进程。其一般思想是：

对许多服务来说，单个程序可以既包含主进程也包含从进程的代码。当一个独立的程序使从进程易于编程或理解时，主程序可以在调用 fork 之后包含对 execve 的调用。

8.26 使用单线程获得表面上的并发性

前面几节讨论了用并发线程或进程实现的并发服务器。然而在某些场合，使用单个执行线程来处理客户的请求也是有意义的。特别是，有些操作系统创建线程或进程的开销十分昂贵，以至于服务器无法承担为每个请求或每个连接创建一个新线程或进程的重负。更重要的是，许多应用要求服务器在多个连接中共享信息。

要理解为什么让服务器的一个单线程提供表面上的并发性（apparent concurrency），我们考虑一下X窗口系统。X窗口系统允许多个客户在一些窗口上画出文本或图像，这些窗口出现在一个位映射（bit-mapped）的显示器上。每个客户通过发送更新窗口内容的请求来控制一个窗口。每个客户都独立地操作，它可能要等待许多小时才会更改显示，或者会经常更新显示。例如，一个应用程序通过画一个钟表的图画来显示时间，它也许每分钟更新一次显示。而同时有一个应用程序要显示某个用户的电子邮件状态，它要一直等到有新邮件到来时才会改变显示。

X窗口系统的服务器将它从客户所获得的信息集中到一个单一的、连续的存储器中，该存储器称为显示缓存（display buffer）。来自所有客户的数据都投给了一个单一的共享数据结构，而且，在X设计中，所用的UNIX版本让每个进程运行在各自独立的地址空间中，它们不能共享存储器。然而X服务器确实需要提供并发服务。

尽管可能通过共享存储器的线程达到期望的并发性，但如果出现在服务器中的全部请求没有超过服务器处理它们的能力，那么，获得表面上的并发性也是可能的。为此，服务器作为单个执行线程来运行，使用select系统调用进行异步I/O。算法8.5描述了单线程服务器要处理多个连接所要采取的步骤。

算法8.5

1. 创建套接字并将其绑定到这个服务的熟知端口上。将该套接字加到一个表中，该表中的项是可以进行I/O的描述符。
2. 使用select在已有的套接字上等待I/O。
3. 如果最初的套接字准备就绪，使用accept获得下一个连接，并将这个新的套接字加入到表中，该表中的项是可以进行I/O的描述符。
4. 如果是最初的套接字以外的某些套接字准备就绪，就使用recv或read获得下一个请求，构造响应，用send或write将响应发回给客户。
5. 继续按以上的步骤2进行处理。

算法8.5 由单个执行线程实现的并发的、面向连接的服务器。服务器线程等待下一个准备就绪的描述符，这个新的描述符意味着一个新的连接的到达，或者是某个客户在已有的连接中发送了一个请求^①

8.27 各服务器类型所适用的场合

循环的和并发的：循环的服务器容易设计、实现和维护，但是并发的服务器可以对请求提供更快的响应。如果请求处理时间很短而且循环方案产生的响应时间对应用来说已足够快了，那么就可

① 第13章给出了只用一个执行线程提供并发性的服务器的例子。

以使用一种循环的实现方法。

真正的和表面上的并发性：只有一个线程的服务器依靠异步 I/O 管理多个连接；而多线程的实现（不管是多个单线程的进程，还是一个进程有多个线程）允许操作系统自动提供并发性。如果创建线程或切换环境的开销很大，或者服务器必须在多个连接之间共享或交换数据，那么可使用单线程的方案。如果使用线程的开销不大，而且服务器必须在多个连接之间共享或交换数据，那么可使用多线程的方案。如果每个从进程可以孤立地运行或者为了要获得最大的并发性（比如在多个处理器上），那么可以使用多进程的方案。

面向连接的和无连接的：因为面向连接的访问意味着使用 TCP，所以它暗示着可靠的交付。无连接的传输意味着使用 UDP，所以它暗示着不可靠的交付。只有在应用协议处理可靠性问题（几乎没有这样做的）或每个客户访问它的服务器都是在同一个局域网中进行的（这只有极小的分组丢失率并且没有分组失序），这时才使用无连接的传输。只要客户和服务器被广域网所分隔，就要使用面向连接的传输。在没有检查应用协议是否处理了可靠性问题之前，决不要将无连接的客户和服务器转移到广域网环境中。

8.28 服务器类型小结

循环的、无连接的服务器

这是最常见的无连接服务器的形式，特别适用于要求对每个请求进行少量处理的服务。循环服务器往往是无状态的，这使其易于理解而且不易出错。

循环的、面向连接的服务器

这是一种较常见的服务器类型，它适用于要求对每个请求进行少量处理，但是要求有可靠的传输。因为与建立和终止连接相关的开销可能很高，平均响应时间可能并不短。

并发的、无连接的服务器

这是一种不常见的服务器类型，服务器要为处理每个请求创建一个新线程或进程。在许多系统中，创建线程或进程所增加的开销决定了由并发性所获得的效率。为证明并发性是可取的，要么创建一个新线程或进程所要求的时间必须明显地小于计算响应所需的时间，要么并发的请求必须能够同时使用多个 I/O 设备。

并发的、面向连接的服务器

这是最一般的服务器类型，因为它提供了可靠的传输（即，它可用于跨越广域互联网）以及并发处理多个请求的能力。有两个基本的实现方法：最常见的实现使用了并发进程或并发线程来处理每个连接；还有一个很不常见的实现方法是依赖单线程和异步 I/O 处理多个连接。

在并发进程的实现方法中，主服务器线程为每个连接创建一个从进程以便对其进行处理。使用多进程使如下情况变得容易，即为每个连接执行一个单独编译的程序，而不是将所有代码放在一个单独的、巨大的服务器程序中。

在单线程实现中，一个执行线程管理多个连接。它通过使用异步 I/O 来达到表面上的并发性。服务器反复地在它所打开的连接上等待 I/O，收到请求便进行处理。由于单个线程处理所有的连接，它就可以在多个连接之间共享数据。然而，因为服务器只有一个线程，即使在一个具有多个处理器的计算机上，它处理请求的速度不会比循环服务器更快。应用程序必须共享数据或者对每个请求的

处理时间必须很短，只要在这种情况下这种服务器实现方案才是可取的。

8.29 重要问题——服务器死锁

许多服务器实现都有一个共同的缺陷：服务器可能会被死锁^①所困扰。为理解为什么会出现死锁，考虑一个循环的、面向连接的服务器。假设某个客户应用程序C不能正常工作。在最简单的情况下，假设C同某个服务器建立了一个连接，但从未发送过一个请求。服务器将接受这个新的连接，并且将调用recv或read来取出下一个请求。服务器进程将在该系统调用上被阻塞，它将在这里等待一个永远也不会到来的请求。

如果客户不能正常工作是由于不能处理响应，那么服务器可能会以一种更加微妙的方式产生死锁。例如，假设客户C同某个服务器建立了连接，向它发送了一系列请求，但从未读取响应。服务器不停地接受请求、产生响应、并将响应发回给客户。在服务器里，TCP软件在这个连接上把最初的几个字节发送给了客户。TCP最终会将客户的接收窗口填满并将停止传输数据。如果服务器应用程序继续产生响应，TCP用于为该连接存储外发数据（outgoing data）的本地缓存将被填满，于是服务器将阻塞。

当操作系统不能满足一个系统调用时，会因调用程序的阻塞产生死锁。特别是，如果TCP没有本地缓存（这用来存放已发送的数据），那么对send或write的调用将阻塞调用者；对recv或read的调用也将阻塞调用者，直到TCP接收到数据。对并发的服务器来说，如果某个客户发送请求或者读取响应失败了，只有与这个特定客户相关的一个从线程会阻塞。然而，对一个单执行线程的实现来说，这个中央服务器将阻塞。若这个中央服务器阻塞，它便不能处理其他连接。这里的要点是，任何只有一个线程的服务器可能会被死锁所困扰。

如果服务器使用了与客户通信时可能会阻塞的系统调用，一个不能正常工作的客户可能会引起单线程服务器死锁。在服务器中，死锁是一个严重的问题，因为它意味着一个客户的行为会使服务器不能处理其他客户的请求。

8.30 其他的实现方法

第9章到第13章提供了本章所描述的服务器算法的例子，第14章和第15章扩充了这些概念，讨论了两个这里没有描述的重要的实际实现技术：多协议的和多服务的服务器。尽管这两种技术都为某些应用提供了一些有趣的优点，但这里并没有包含它们，这是因为，最好把它们理解为对单线程服务器算法的简单的一般化，这种单线程服务器算法将在第13章中讨论。

8.31 小结

从概念上说，一个服务器由一个简单的算法构成，它一直循环运行，等待下一个来自某个客户的请求，处理这个请求，发送应答。然而实际上，服务器有多种实现方法来达到可靠性、灵活性和

^① 术语死锁（deadlock）指的是一个或一组程序无法进行下去的状态，因为它们被阻塞并等待着一个永远不会发生的事件。对服务器来说，死锁意味着服务器不能回答请求。

有效性。

对要求很少计算的服务，循环的实现方法工作得很好。当使用面向连接的传输时，循环服务器一次处理一个连接；对于无连接的传输，循环服务器一次处理一个请求。

为达到有效性，服务器往往通过同时处理多个请求来提供并发服务。面向连接的服务器为处理每个新连接创建一个线程/进程，它通过这种方法，在各个连接之间提供了并发性。无连接的服务器通过为处理每个请求而创建一个新线程/进程面提供并发性。

任何服务器，如果它是由一个单线程实现的，而且它使用了像 `recv`、`read`、`send` 或 `write` 这样的同步系统调用，那么它就可能被死锁所困扰。在循环服务器以及使用一个单线程实现的并发服务器中，死锁都有可能发生。服务器的死锁是个特别严重的问题，因为它意味着单个客户的错误行为可能会使服务器不能处理其他客户的请求。

深入研究

Linux 和其他 UNIX 系统所提供的服务器含有许多服务器算法的例子；程序员常常为一些编程技术参考源代码。

习题

- 8.1 计算一下，如果互联网络的吞吐量为每秒 2.3 千字节，那么，一个循环服务器传送一个 200 兆字节的文件需要多长时间？
- 8.2 如果有 20 个客户，各自每秒向服务器发送 2 个请求，服务器可以花费在每个请求上的最大时间是多少？
- 8.3 在你所访问的计算机中，一个并发的、面向连接的服务器接受一个新的连接，并为处理这个连接而创建一个新的进程所需要的时间有多长？如果创建一个新线程又如何？
- 8.4 为一个并发的、无连接的服务器写一个算法，它为每个请求创建一个新的进程。
- 8.5 修改前一题中的算法，使它为每个客户创建一个新进程，而不是为每个请求创建一个新进程。你的算法是如何处理进程终止的？
- 8.6 面向连接的服务器在各个连接之间提供了并发性。为进一步提高并发的、面向连接的服务器的并发性，使从线程为每个连接创建附加的线程，这样做是否有意义？解释原因。
- 8.7 重写 TCPEcho 客户软件，使它用单线程并发地处理来自键盘的输入、来自它的 TCP 连接的输入、以及向它的 TCP 连接的输出。
- 8.8 客户能引起并发服务器出现死锁或中断服务吗？为什么？
- 8.9 仔细查看 `select` 系统调用。单线程服务器怎样使用 `select` 避免死锁？
- 8.10 `select` 调用一个参数，该参数指明它将检查多少个 I/O 描述符。解释一下，这个参数是怎样使单线程服务器程序可以移植到许多 UNIX 系统的。

第9章 循环的、无连接服务器（UDP）

9.1 引言

前一章讨论了多种可能的服务器设计，并比较了各种设计的优缺点。本章将给出一个循环的服务器实现的例子，它使用无连接传输。该例服务器采用算法 8.2^①。后几章将继续通过举例讨论其他服务器算法的实现。

9.2 创建被动套接字

创建被动套接字的步骤，与创建主动套接字类似。这里包括许多细节，并且，为获得熟知的协议端口号，需要程序查找服务名。

为简化服务器代码，程序员应使用一些过程，以便隐藏套接字分配的细节。与客户的例子一样，我们的例子实现使用两个高层的过程，passiveUDP 和 passiveTCP，它们负责分配被动套接字，并将它绑定到服务器的熟知端口上。每个服务器根据服务器是使用无连接传输还是面向连接的传输进行选择，由此决定调用这些过程中的哪一个。本章将研究 passiveUDP；下一章将展示 passiveTCP 的代码。两个过程在许多细节上有共同之处，它们都调用下层的过程 passivesock 来完成工作。

无连接服务器调用函数 passiveUDP，为它所提供的服务创建套接字。如果服务器需要使用为熟知服务保留的某个端口（即编号在低端的端口），服务器进程就必须有特权^②。任意一个应用程序可使用 passiveUDP 为无特权的服务创建套接字。passiveUDP 调用 passivesock 创建无连接的套接字，然后为其调用者返回套接字描述符。

为易于测试客户和服务器软件，passivesock 通过增加全局整数 portbase 的内容，重新分配所有的端口值。本质上，所有端口值都重新映射到更高的范围上。使用 portbase 的重要性将在后几章中弄得更清楚。但是，基本概念还是很容易理解的：

在给定计算机上，两个服务器不能使用相同的协议端口号。为了让程序员能在一台计算机上既测试新版的客户—服务器软件，又让现有的工作版本继续执行，可以临时将所有端口号映射到更高的范围上去。

使用 portbase 的主要优点是其安全性和通用性。首先，由于程序员不需要修改程序中引用端口号的地方，所以，使用 portbase 发生错误（例如，在插入测试代码时不小心漏掉某些地方，或在测试后忘记删除）的可能性就减少了。其次，portbase 是解决该问题的通用方法。除了允许在测试服务器新版本时，继续运行服务器工作版本外，采用 portbase 还允许同时测试多个服务器新版本，为

① 见第 79 页关于算法 8.2 的描述。

② 同多数 UNIX 系统一样，在 Linux 中，应用程序只有作为根（即为超级用户）运行，才能有足够的特权绑定到编号小于 1000 的端口上。

此，程序员只需给每个版本分配一个唯一的、非零的 portbase 值。这样，某个特定版本服务器传递给套接字 API 的端口号就不会与其他版本或运行服务器的端口号相冲突。其要点是：

用全局变量提供端口映射使测试更安全，因为这样可以在不全面修改程序的情况下，同时测试服务器的多个版本。

```
/* passiveUDP.c - passiveUDP */

int      passivesock(const char *service, const char *transport,
                     int qlen);

/*
 * passiveUDP - create a passive socket for use in a UDP server
 */
int
passiveUDP(const char *service)
/*
 * Arguments:
 *      service - service associated with the desired port
 */
{
    return passivesock(service, "udp", 0);
}
```

过程 passivesock 含有分配套接字的细节，包括 portbase 的使用。它带有三个参数。第一个参数指明一个服务名，第二个参数指明协议名，第三个参数（只用于 TCP 套接字）指明连接请求队列所需长度。作为第一个参数的字符串可以是服务的名字，也可以是服务的协议端口号，如果使用的是端口号，必须编码为字符串。passivesock 分配一个数据报或流的套接字，将套接字绑定到服务所用的熟知端口，然后为其调用者返回套接字描述符。

回想在服务器将套接字绑定到一个熟知端口时，它必须使用结构 sockaddr_in 指明地址，该结构包括一个 IP 地址和一个协议端口号。passivesock 使用常量 INADDR_ANY（见第 8 章有关内容）代替特定的本地 IP 地址，这使得它既可在具有单个 IP 地址的主机上运行，也可在具有多个 IP 地址的路由器或多宿主机上运行。注意 passivesock 使用了一个指定的协议端口号——使用 INADDR_ANY 和指定端口号的含义是服务器将在机器的任一 IP 地址上接收发给指定端口的数据。

```
/* passivesock.c - passivesock */

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

#include <netinet/in.h>

#include <stdlib.h>
```

```
#include <string.h>
#include <netdb.h>

extern int errno;

int errexit(const char *format, ...);

unsigned short portbase = 0; /* port base, for non-root servers */

/* -----
 * passivesock - allocate & bind a server socket using TCP or UDP
 * -----
 */
int
passivesock(const char *service, const char *transport, int qlen)
/*
 * Arguments:
 *   service - service associated with the desired port
 *   transport - transport protocol to use ("tcp" or "udp")
 *   qlen     - maximum server request queue length
 */
{
    struct servent *pse; /* pointer to service information entry */
    struct protoent *ppe; /* pointer to protocol information entry*/
    struct sockaddr_in sin; /* an Internet endpoint address */
    int s, type; /* socket descriptor and socket type */

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;

    /* Map service name to port number */
    if ( pse = getservbyname(service, transport) )
        sin.sin_port = htons(ntohs((unsigned short)pse->s_port)
                           + portbase);
    else if ( (sin.sin_port = htons((unsigned short)atoi(service))) == 0 )
        errexit("can't get \"%s\" service entry\n", service);

    /* Map protocol name to protocol number */
    if ( (ppe = getprotobynumber(transport)) == 0)
        errexit("can't get \"%s\" protocol entry\n", transport);

    /* Use protocol to choose a socket type */
    if (strcmp(transport, "udp") == 0)
        type = SOCK_DGRAM;
```

```

else
    type = SOCK_STREAM;

/* Allocate a socket */
s = socket(PF_INET, type, ppe->p_proto);
if (s < 0)
    errexit("can't create socket: %s\n", strerror(errno));

/* Bind the socket */
if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
    errexit("can't bind to %s port: %s\n", service,
            strerror(errno));
if (type == SOCK_STREAM && listen(s, qlen) < 0)
    errexit("can't listen on %s port: %s\n", service,
            strerror(errno));
return s;
}

```

9.3 进程结构

图 9.1 说明了循环的、无连接服务器所用的简单的进程结构。只需要一个执行线程。

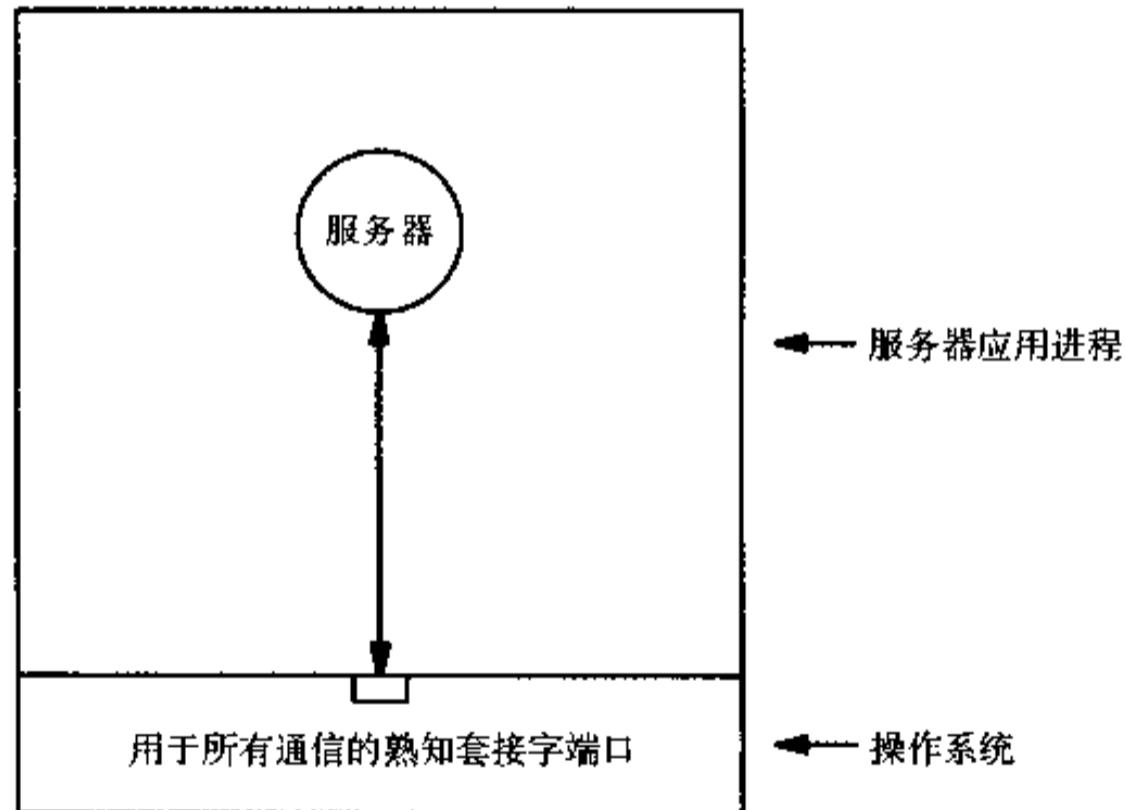


图 9.1 循环的、无连接服务器所用的进程结构。单个执行线程使用一个套接字与多个客户通信

单个服务器线程永远运行着。它使用一个被动的套接字，该套接字已绑定到所提供的服务使用的熟知端口。服务器从套接字获取请求，计算出响应，然后将响应返回给使用相同套接字的客户。服务器把请求中的源地址作为应答中的目的地址。

9.4 TIME服务器举例

下面将举例说明无连接服务器进程如何使用以上描述的套接字分配过程。回忆第7章中，客户使用TIME服务从另一个系统的服务器中获得当前时间。由于TIME几乎不需要计算，循环式服务器实现运行得不错。文件 UDPtimed.c 中含有循环的、无连接 TIME 服务器所用的代码。

```
/* UDPtimed.c - main */

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

#include <stdio.h>
#include <time.h>
#include <string.h>

extern int errno;

int passiveUDP(const char *service);
int errexit(const char *format, ...);

#define UNIXEPOCH 2208988800UL /* UNIX epoch, in UCT secs */

/*
 * main - Iterative UDP server for TIME service
 */
int
main(int argc, char *argv[])
{
    struct sockaddr_in fsin; /* the from address of a client */
    char *service = "time"; /* service name or port number */
    char buf[1]; /* "input" buffer; any size > 0 */
    int sock; /* server socket */
    time_t now; /* current time */
    unsigned int alen; /* from-address length */

    switch (argc) {
    case 1:
        break;
    case 2:
        service = argv[1];
        break;
    default:
```

```

        errexit("usage: UDPtimed [ port]\n");
    }

    sock = passiveUDP(service);

    while (1) {
        alen = sizeof(fsin);
        if (recvfrom(sock, buf, sizeof(buf), 0,
                     (struct sockaddr *)&fsin, &alen) < 0)
            errexit("recvfrom: %s\n", strerror(errno));
        (void) time(&now);
        now = htonl((unsigned long)(now + UNIXEPOCH));
        (void) sendto(sock, (char *)&now, sizeof(now), 0,
                      (struct sockaddr *)&fsin, sizeof(fsin));
    }
}

```

与任何服务器类似，UDPTimed 进程必须永远运行着。因此，代码主体含有一个无限的循环，该循环每次接收一个请求，计算当前的时间，然后给发送请求的客户返回响应。

代码含有几处细节。分析完参数后，UDPTimed 调用 passiveUDP 为 TIME 服务创建一个被动套接字。然后它便进入循环。TIME 协议指明，客户可发送任意一个数据报作为请求。由于服务器不解释数据报的内容，数据报可以是任何长度，并可含有任意值。本例实现使用 recvfrom 读取下一个数据报。recvfrom 将传入数据报放到缓存 buf 中，并将发送数据报的客户的端点地址放到结构 fsin 中。由于它不必查看数据，因此实现只使用了单个字符的缓存。如果数据报含有的数据多于一个字节，recvfrom 就丢弃所有剩余的字节。

UDPTimed 使用系统函数 time 获得当前时间。回忆在第 7 章中，Linux 像多数 UNIX 系统一样，使用 32 比特整数表示时间，时间是从 1970 年 1 月 1 日零时开始计算。从操作系统获得时间后，UDPTimed 必须将它转换为用因特网纪元（epoch）测量的时间值，并用网络字节序存放结果。为完成转换，它增加了一个常量 UNIXEPOCH，该常量值定义为 2208988800，即为因特网计时起始值与 Linux 计时起始值间相差的秒数。然后它调用 sendto 将结果传回客户。sendto 使用结构 fsin 中的端点地址作为目的地址（即它使用了发送数据报的客户的地址）。

9.5 小结

对于简单的服务，服务器为每个请求进行的计算很少，因此循环的实现就可很好地工作。本章给出了用于 TIME 服务的循环服务器例子，它使用 UDP 用于无连接访问。本例说明了过程如何隐藏套接字分配的细节，并使得服务器代码更简单和更容易理解。

深入研究

Harrenstien [RFC 738] 定义了 TIME 协议。Mills [RFC 1305] 描述了网络时间协议 NTP (Network Time Protocol)；Mills [September 1991] 总结了在实际网络中使用 NTP 的有关问题，

Mills [RFC 1361] 还讨论了将 NTP 用于时钟同步。Marzullo 和 Owicki [July 1985] 也讨论了如何在分布式环境中维护时钟。

习题

- 9.1 用仪器测量 UDPtimed 每次处理一个请求要花多长时间。如果你有一个网络分析仪，也用它测量请求和响应的时间间隔。
- 9.2 假定 UDPtimed 在收到请求和发送响应期间不慎破坏了客户的地址(即服务器在调用 sendto 前偶然为 fsin 指派了一个随机值)。这会发生什么？为什么？
- 9.3 做一个试验，判断若 N 个客户同时给 UDPtimed 发送请求，情况会如何？改变发送者的个数 N 和它们发送的数据大小 S。解释服务器为什么不能给所有请求返回响应。(提示：查看 listen 手册页。)
- 9.4 在 UDPtimed.c 的例子代码中，调用 recvfrom 时指明了缓存大小为 1。若指明缓存大小为 0 情况会如何？
- 9.5 计算 Linux 时间起始值与因特网时间起始值间的差距。记住要考虑闰年的情况。你计算的值与 UDPtimed 中定义的常量一致吗？如果不一致，试解释。(提示：阅读有关闰年秒数的内容。)
- 9.6 为安全检查，系统管理员请你修改 UDPtimed，以便使它保存访问该服务的所有客户的书面日志。修改代码，使它在请求到达时打印一行到控制台上。试解释日志记录会如何影响此服务。
- 9.7 如果你可访问一对连接在广域互联网上的机器，使用第 7 章的 UDPtime 客户和本章的 UDPtimed 服务器，查看你的互联网是否会遗漏或复制分组。

第 10 章 循环的、面向连接的服务器（TCP）

10.1 引言

上一章给出了使用 UDP 进行无连接传输的循环服务器的例子。本章将讨论循环服务器如何使用 TCP 进行面向连接的传输。这个服务器的例子采用算法 8.1^①。

10.2 分配被动的 TCP 套接字

第 9 章提到了，面向连接的服务器使用函数 `passiveTCP` 分配一个流套接字，并将该套接字绑定到提供服务的熟知端口上。`passiveTCP` 带有两个参数，第一个参数是字符串，它指明服务的名字或端口号，第二个参数指明传入连接请求队列所需的长度。如果第一个参数含有服务名，该名字就必须与服务数据库中的某一项相匹配，该数据库可以通过系统函数 `getservbyname` 访问。如果第一个参数指明了端口号，它必须将数字表示为文本字符串（如“79”）。

```
/* passiveTCP.c - passiveTCP */

int      passivesock(const char *service, const char *transport,
                     int qlen);

/*
 * passiveTCP - create a passive socket for use in a TCP server
 */
int
passiveTCP(const char *service, int qlen)
/*
 * Arguments:
 *   service - service associated with the desired port
 *   qlen    - maximum server request queue length
 */
{
    return passivesock(service, "tcp", qlen);
}
```

^① 见 78 页中算法 8.1 的描述。

10.3 用于 DAYTIME 服务的服务器

回忆第 7 章，DAYTIME 服务允许某台机器上的用户从另一台机器上获得当前日期和时间。由于 DAYTIME 服务是为人所用的，它规定服务器发送响应时，必须将日期格式化为可读的 ASCII 文本字符串。因此，客户在收到响应时，就可以正确地为用户显示响应结果。

第 7 章说明了客户是如何使用 TCP 与 DAYTIME 服务器联系的，以及如何显示服务器返回的文本。由于获取和格式化日期只需要很少的处理，并且用户对此服务的需求也很少，因此 DAYTIME 服务器不必优化速率。如果在服务器忙于处理某个请求时，其他客户尝试建立连接请求，协议软件就会将这些请求排队。因此，循环实现就足够了。

10.4 进程结构

如图 10.1 所示，循环的、面向连接的服务器使用一个单执行线程。该线程永远循环，使用一个套接字处理入请求，并且用另一个临时的套接字处理与客户的通信。

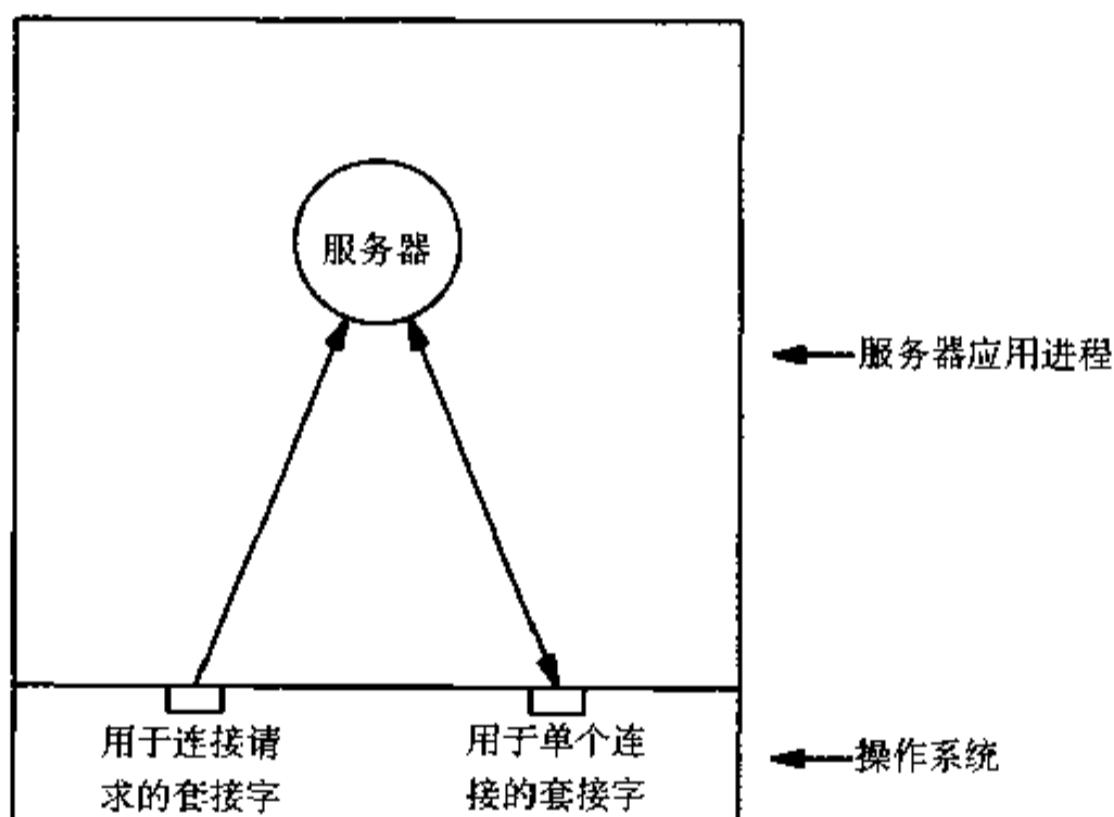


图 10.1 循环的、面向连接的服务器的进程结构。单执行线程在熟知端口上等待连接，然后在此连接上与客户通信

使用面向连接传输的服务器在这些连接上不断循环：它在熟知端口上等待来自某客户的下一个连接、接受连接、处理连接、关闭连接，然后再次等待连接。DAYTIME 服务的实现特别简单，这是因为服务器不必接收来自客户的显式请求——它根据传入连接的出现来触发响应。由于客户不发送显式请求，服务器就不必从连接上读取数据。

10.5 DAYTIME 服务器举例

文件 TCPdaytimed.c 中含有循环的、面向连接的 DAYTIME 服务器的例子代码。

```
/* TCPdaytimed.c - main */

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

#include <unistd.h>
#include <stdio.h>
#include <string.h>

extern int          errno;
int                 errexit(const char *format, ...);
void                TCPdaytimed(int fd);
int                 passiveTCP(const char *service, int qlen);

#define QLEN      32

/*
 * main - Iterative TCP server for DAYTIME service
 */
int
main(int argc, char *argv[])
{
    struct sockaddr_in fsin;           /* the from address of a client */
    char    *service = "daytime";     /* service name or port number */
    int     msock, ssock;             /* master & slave sockets */
    unsigned int    alen;              /* from-address length */

    switch (argc) {
    case 1:
        break;
    case 2:
        service = argv[1];
        break;
    default:
        errexit("usage: TCPdaytimed [port]\n");
    }

    msock = passiveTCP(service, QLEN);

    while (1) {
        alen=sizeof(fsin);
        ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
        if (ssock < 0)
            errexit("accept failed: %s\n", strerror(errno));
    }
}
```

```
TCPdaytimed(ssock);
(void) close(ssock);
}

/*
 * TCPdaytimed - do TCP DAYTIME protocol
 *
 */
void
TCPdaytimed(int fd)
{
    char      *pts; /* pointer to time string */
    time_t    now;   /* current time */
    char      *ctime();

    (void) time(&now);
    pts = ctime(&now);
    (void) write(fd, pts, strlen(pts));
}
```

类似上一章所描述的循环的无连接服务器，这种循环的、面向连接的服务器也必须永远运行。在创建了在熟知端口上监听的套接字后，服务器就进入无限循环，在循环中接受和处理连接。

服务器的代码相当短，这是因为 `passiveTCP` 的调用隐藏了套接字分配和绑定的细节。调用 `passiveTCP` 创建了与 DAYTIME 服务所用熟知端口相关联的主套接字。第二个参数指明主套接字的连接请求队列长度是 QLEN，从而允许系统在忙于回答某个客户的连接请求时，将来自其他 QLEN 个客户的连接请求进行排队。

创建主套接字后，服务器的主程序将进入无限循环。在每次循环中，服务器调用 `accept` 从主套接字获得下一个连接。为防止服务器在等待来自客户的连接时耗费资源，`accept` 调用将一直阻塞，直到一个连接到达。当连接请求到达时，TCP 协议软件为建立连接而忙于进行三次握手。一旦握手完成，并且系统已为传入连接分配了一个新套接字，`accept` 调用将返回新套接字的描述符，并允许服务器继续执行。如果没有连接请求到达，服务器进程将在 `accept` 调用中一直保持阻塞状态。

每次在新的连接到达时，服务器就调用过程 `TCPdaytimed` 对它进行处理。`TCPdaytimed` 中的代码以系统函数 `time` 和 `ctime` 的调用为核心。过程 `time` 返回一个 32 比特整数，以此给出当前时间，它是自 Linux 纪元（计时起始值）以来所经过的秒数。库函数 `ctime` 带有一个整型参数，该参数指明用 Linux 纪元所经过的秒数表示的时间。该函数返回一个含有格式化的时间和日期的 ASCII 字符串的地址，便于人们理解。一旦服务器获得用 ASCII 字符串表示的时间和日期，它就调用 `write` 将字符串通过 TCP 连接发送给客户。

调用 `TCPdaytimed` 一旦返回，主程序就继续执行循环，再次调用 `accept`。而 `accept` 调用在另一个请求到达前将使服务器阻塞。

10.6 关闭连接

过程 TCPdaytimed 写完响应后，调用就返回。一旦调用返回，主程序就明确地关闭该连接到达的套接字。

调用 close 需要从容关闭。具体说就是，TCP 必须保证所有数据都可靠地交付给客户，并且在连接终止前都已被确认。因此，当调用 close 时，程序员不必担心正在被交付的数据。

当然，TCP 从容关闭的定义意味着 close 调用可能不立刻返回——在服务器上的 TCP 从客户的 TCP 收到答复前，调用将阻塞。一旦客户确认它收到了所有数据和终止连接的请求，close 调用即返回。

10.7 连接终止和服务器的脆弱性

应用协议决定了服务器如何管理 TCP 连接。特别是，应用协议通常指定了终止策略的选择。例如，让服务器关闭连接对 DAYTIME 协议来说很合适，因为服务器知道它何时结束数据发送。对那些具有更复杂的客户 - 服务器交互的应用，不能在处理一个请求后立刻关闭连接，因为他们必须等待，以便了解客户是否会发送其他请求报文。例如，考虑一个 ECHO 服务器。由于客户决定了要服务器回送的数据量，因此客户控制了服务器的处理。由于服务器必须处理任意多的数据，它不能在一次读和写之后就关闭连接。因此，客户必须发送信号表示已经完成了，以便让服务器知道应在何时终止连接。

允许客户控制连接的持续时间可能是危险的，因为这就是允许客户控制资源的使用。特别是，误操作的客户可能会导致服务器消耗像套接字和 TCP 连接之类的资源。我们举例的服务器似乎永远不会用光资源，因为它明确地关闭了连接。但我们的简单终止策略可能在误动作的客户面前是脆弱的。要理解其中的原因，回想 TCP 定义了连接关闭后连接超时的时间是最大报文段生命期的两倍（ $2 \times MSL$ ）。在超时期间，TCP 将保存连接的记录，以便它能正确地拒绝任何可能已被延迟的旧分组。因此，如果客户迅速、重复地发出请求，则它们可能会把服务器上的资源用光。虽然程序员可能对协议的控制是很少的，但他们应理解协议是如何使得分布式软件在网络故障下显露出脆弱性的，并且应在设计服务器时竭力避免这种脆弱性。

10.8 小结

循环的、面向连接的服务器每处理一个连接便循环一次。在连接请求从客户到达前，服务器将在 accept 的调用中处于阻塞状态。一旦下层协议软件建立了新的连接，并创建了新的套接字，调用 accept 将返回套接字的描述符，并使服务器继续运行。

回忆在第 7 章中，DAYTIME 协议根据每次连接的出现触发服务器的响应。客户不必发送请求，因为服务器只要检测到新的连接就响应。为形成响应，服务器从操作系统获取当前时间，并将信息格式化为适于人们阅读的字符串，然后发送响应给客户。服务器在发送响应后，将关闭该连接对应的套接字。由于 DAYTIME 服务只允许每个连接发一个响应，因而立刻关闭连接的策略是可行的。但对那些在单个连接上允许有多个请求到达的服务器，就必须等待客户关闭连接。

深入研究

Postel [RFC867] 描述了本章使用的 DAYTIME 协议。

习题

- 10.1 在你的本地系统上,一个进程是否需要有特权才能运行DAYTIME服务器? 它运行DAYTIME客户需要有特权吗?
- 10.2 利用连接的出现来触发服务器响应的主要优点是什么? 主要缺点呢?
- 10.3 一些 DAYTIME 服务器用两个字符的组合来结束文本行: 回车 (CR) 和换行 (LF)。试修改服务器例子,让它在文本行末尾发送CR-LF代替只发送LF。标准规定文本行应如何终止?
- 10.4 在服务器忙时到达的其他连接请求将进行排队,TCP软件通常为此队列分配固定的长度,并允许服务器使用listen改变队列长度。你的本地TCP软件提供多长的队列? 服务器使用listen可让队列有多长?
- 10.5 修改 TCPdaytimed.c 中服务器例子的代码,使它在写完响应后不明显地关闭连接。它仍会正确工作吗? 为什么?
- 10.6 有的面向连接的服务器在发送响应后就明显关闭连接,有的服务器却在关闭连接前允许客户将连接保持一个任意长时间,比较这两种方式。两者的优点和缺点各是什么?
- 10.7 假定 TCP 使用 4 分钟的连接超时(即在连接关闭后保持信息 4 分钟)。如果 DAYTIME 服务器运行在这样的系统上,它只有 100 个时隙用于 TCP 连接信息,服务器处理请求的最大速率应该是多少才能使它不会把这些时隙用光?

第11章 并发的、面向连接的服务器（TCP）

11.1 引言

上一章举例说明了循环服务器如何使用面向连接的传输协议。本章将给出一个使用面向连接传输的并发服务器的例子。所举的服务器例子采用算法 8.4^①，这是程序员在构造并发 TCP 服务器时最常使用的设计。服务器计算响应时，服务器依赖操作系统对并发处理的支持来达到并发性。在系统启动时，系统管理员设法让主服务器进程自动启动。主服务器将永远运行，等待从客户到来的新的连接。主服务器创建一个新的从线程/进程处理每个新连接，并允许各个从线程/进程处理与客户的所有通信。回忆一下第 8 章，我们介绍过算法 8.4 有两种常见的实现方法。本章探讨多个单线程进程的实现方法，下一章将探讨一个进程里有多个线程的实现方法，并对两者进行比较。

11.2 并发 ECHO

考虑第 7 章中描述的 ECHO 服务。客户打开到某个服务器的连接，然后在该连接上重复发送数据，并读取从服务器返回的“回显”(echo)。ECHO 服务器响应每个客户。它接受连接，读取来自该连接的数据，然后向客户发回与该服务器所收到的数据相同的数据。

为允许客户发送任意多的数据，服务器在发送响应前并非读取全部输入。它只是交替地进行读和写。当新的连接到达时，服务器就进入循环。在每一次循环中，服务器首先从该连接中读取数据，然后将数据写回到该连接上。服务器在遇到文件结束(end-of-file)的条件前，将不断循环，遇到该条件后才关闭连接。

11.3 循环与并发实现的比较

ECHO 服务器的循环实现可能表现欠佳，因为它要求某个给定的客户等待此服务器处理在这以前到达的连接请求。如果客户要发送大量的数据（例如若干兆字节），循环服务器在处理完该请求前，会让所有其他的客户延迟。

ECHO 服务器的并发实现避免了长时间的延迟，因为它不允许单个客户占有所有的资源。并发服务器使其可与许多客户同时进行通信。因此，从客户的观点看，并发服务器比循环服务器提供了较短的响应时间。

^① 见 81 页中算法 8.4 的描述。

11.4 进程结构

图 11.1 举例说明了并发的、面向连接的服务器的进程结构，该服务器使用了单线程进程。如图所示，主服务器进程并不与客户直接通信。它只是在熟知端口上等待下一个连接请求。一旦有某一个请求到达，系统就返回用于该连接的新套接字的描述符。主服务器进程创建一个从进程来处理该连接，并允许从进程并发操作。任何时候，服务器都包括一个主进程，以及零个或多个从进程。

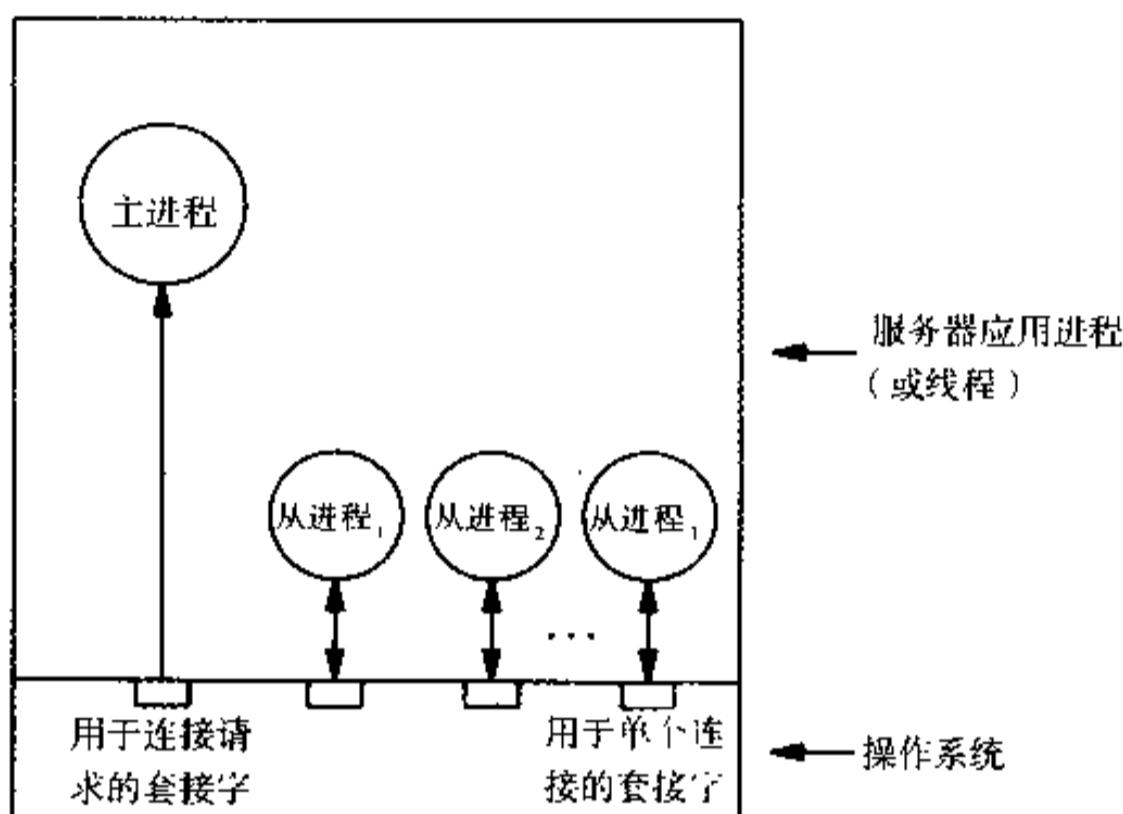


图 11.1 并发的、面向连接的、使用单线程进程的服务器的进程结构。主服务器进程接受每个传入连接，并创建一个从进程来处理它

为避免等待连接时使用 CPU 资源，主服务器使用 accept 的阻塞调用 (blocking call) 从熟知端口上获得下一个连接。因此，类似第 10 章中的循环服务器，并发服务器中的主服务器进程大部分时间都处于 accept 调用的阻塞状态。当连接请求到达时，accept 调用便返回，使得主进程继续运行。主进程创建一个从进程来处理请求，并重新调用 accept。该调用在另一个连接请求到达前将使服务器再次阻塞。

11.5 并发 ECHO 服务器举例

文件 TCPechod.c 中含有 ECHO 服务器的代码，它使用并发进程为多个客户提供并发服务。

```

/* TCPechod.c - main, TCPechod */

#define _USE_BSD
#include <sys/types.h>
#include <sys	signal.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>

```

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

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define QLEN 32      /* maximum connection queue length */
#define BUFSIZE 4096

extern int errno;

void reaper(int);
int TCPechod(int fd);
int errexit(const char *format, ...);
int passiveTCP(const char *service, int qlen);

/*
 * main - Concurrent TCP server for ECHO service
 */
int
main(int argc, char *argv[])
{
    char *service = "echo";           /* service name or port number */
    struct sockaddr_in fsin;         /* the address of a client */
    unsigned int alen;               /* length of client's address */
    int msock;                      /* master server socket */
    int ssock;                      /* slave server socket */
    switch (argc) {
        case 1:
            break;
        case 2:
            service = argv[1];
            break;
        default:
            errexit("usage: TCPechod [ port]\n");
    }

    msock = passiveTCP(service, QLEN);

    (void) signal(SIGCHLD, reaper);

    while (1) {
```

```
    alen = sizeof(fsin);
    ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
    if (ssock < 0) {
        if (errno == EINTR)
            continue;
        erexit("accept: %s\n", strerror(errno));
    }
    switch (fork()) {
    case 0:           /* child */
        (void) close(msock);
        exit(TCPechod(ssock));
    default:          /* parent */
        (void) close(ssock);
        break;
    case -1:
        erexit("fork: %s\n", strerror(errno));
    }
}

/*
 * TCPechod - echo data until end of file
 */
int
TCPechod(int fd)
{
    char      buf[BUFSIZ];
    int       cc;

    while (cc = read(fd, buf, sizeof buf)) {
        if (cc < 0)
            erexit("echo read: %s\n", strerror(errno));
        if (write(fd, buf, cc) < 0)
            erexit("echo write: %s\n", strerror(errno));
    }
    return 0;
}

/*
 * reaper - clean up zombie children
 */
void
```

```

reaper(int sig)
{
    int      status;

    while (wait3(&status, WNOHANG, (struct rusage *)0) >= 0)
        /* empty */;
}

```

如例所示，控制并发的调用只占了代码的一小部分。主服务器进程从 main 开始执行。主服务器检查完参数后，就调用 passiveTCP 为熟知端口创建一个被动套接字，然后便进入无限循环。符号常量 QLEN 作为第二个参数传递给了 passiveTCP，它指定了服务器正忙于处理当前连接时，能被放入队列中的其他传入 TCP 连接的最大数^①。

在每次循环中，主服务器调用 accept 等待客户的连接请求，与在循环服务器中一样，accept 调用在请求到达前将阻塞。在下层协议软件收到连接请求后，系统为新的连接创建一个套接字，然后调用 accept 返回套接字描述符。

accept 返回后，主服务器创建一个从进程处理连接。为此，主进程调用 fork 将自己分成两个进程^②。新创建的子进程中的线程首先关闭主套接字，然后调用过程 TCPechod 处理连接。父进程中的线程关闭那个为处理新连接而创建的套接字，并继续执行无限循环。下一次循环将在 accept 处等待新连接的到达。注意，调用 fork() 后，原进程和新进程都可使用打开的套接字，并且在系统释放它以前，它们都必须要关闭套接字。因此，当主进程中的线程对新连接调用 close 时，该连接所用的套接字只从主进程中消失。类似地，当从进程中的线程对主套接字调用 close 时，该套接字只在从进程中消失。从进程退出前，将继续使用新连接所用的套接字；主服务器也将继续使用熟知端口所对应的套接字。

从进程关闭主套接字后，就调用过程 TCPechod，该过程为一个连接提供 ECHO 服务。过程 TCPechod 包括一个循环，它重复调用 read 以便从连接上获取数据，然后调用 write 通过该连接发回数据。在正常情况下，read 返回读取的字节数（正数）。当出现差错时（如客户与服务器间的网络连接中断了），它就返回一个小于零的值。如果遇到文件结束（end-of-file）的情况（即不能从套接字中提取更多数据），它就返回零。类似地，write 在正常情况下返回所写的字符数，但如果出现差错就返回小于零的值。从进程将检查返回码，并在出现差错时使用 errexit 打印此消息。

TCPechod 如果能正确无误地回送所有数据，它就返回零。当 TCPechod 返回时，主程序使用返回值作为 exit 调用的一个参数。Linux 将 exit 调用解释为终止进程的请求，并使用其参数作为进程退出码。通常约定，进程使用退出码 0 表示正常终止。因此，从进程在完成 ECHO 服务后就正常退出。当从进程退出时，系统自动关闭所有打开的描述符，包括用于 TCP 连接的描述符。

11.6 清除游离（errant）进程

由于使用 fork 的并发服务器动态地生成进程，这就引入了一个潜在的问题，即不完全终止的进程（incompletely terminated process）。Linux 是这样解决这个问题的，只要一个子进程退出，便给父进程发送一个信号（signal）。正在进行退出的进程将保持在死状态（zombie state），直到父进程执

^① Linux 通过常量 SOMAXCONN 限制队列长度的最大值。

^② fork 创建的进程包含一个单执行线程；返回值可以对原来的父进程和新创建的子进程加以区分。

行 `wait3` 的系统调用为止。为完全终止子进程（即消除死进程），我们的 ECHO 服务器例子捕获子进程的终止信号，并执行一个信号处理函数。调用

```
signal(SIGCHLD, reaper);
```

通知操作系统，主服务器进程在收到子进程已退出的信号（信号 `SIGCHLD`）时，就应执行函数 `reaper`。在调用 `signal` 后，服务器每收到一个 `SIGCHLD` 信号时，系统都自动调用 `reaper`。

函数 `reaper` 调用系统函数 `wait3` 完成子进程的终止并退出。`wait3` 在一个或多个子进程退出（不管是什么原因）前将阻塞。它将返回值放在一个状态结构中，通过查看该结构可找到已退出的进程。由于程序在 `SIGCHLD` 信号到达时调用 `wait3`，它在一个子进程已退出后总是被调用的。为确保一个错误的调用不会使服务器死锁，程序使用参数 `WNOHANG` 指明 `wait3` 不要为进程退出而阻塞等待，而应该立刻返回，即使是在没有进程已退出的情况下。

11.7 小结

面向连接的服务器通过允许多个客户与服务器通信而达到并发。本章中的简单实现使用了 `fork` 函数在每次连接到达时创建一个新的从进程。主进程中的线程永远不会与任何客户交互；它只是接受连接，并创建一个从进程处理各个连接。

每个从进程在主程序调用 `fork` 后立即开始执行。主进程关闭新连接所用的描述符的副本，而从进程关闭主描述符的副本。由于操作系统要关闭从进程的套接字副本，在从进程退出后，一个到客户的连接便终止了。

深入研究

Postel [RFC 862] 定义了 TCP 服务器例子中使用的 ECHO 协议。

习题

- 11.1 让 ECHO 服务器保持一个时间日志，记录创建每个从进程的时间，以及从进程终止的时间。在你能够发现在从进程之间发生重叠以前，你必须要启动多少个客户？
- 11.2 在任意一个客户必定会被拒绝服务前，有多少个客户可同时访问本例中的并发服务器？在任意一个客户必定会被拒绝服务前，有多少个客户可同时访问第 10 章中的循环服务器？
- 11.3 构造一个循环实现的 ECHO 服务器。做一个实验，判断人们是否能感觉到并发和循环版本间响应时间的差异。
- 11.4 修改本服务器的例子，让过程 `TCPechod` 在返回前明确地关闭连接。试解释 `close` 的显式调用为何会使代码更易于维护？
- 11.5 构造一个经过修改的 ECHO 服务器，在同一进程中创建一个新的执行线程而不是一个新的进程，测量一下两种服务器执行时间的差异。
- 11.6 在上题所述的经过修改的服务器中，主线程和从线程分别关闭了哪个套接字？为什么？

第 12 章 将线程用于并发 (TCP)

12.1 引言

上一章介绍了面向连接的并发服务器，并采用多进程设计来说明如何实现并发。本章将讨论如何采用线程来实现并发。在介绍线程的一般特征和其优缺点后，我们将举例说明如何在一个进程中创建多个线程来实现一个面向连接的并发服务器。此并发服务器仍采用了算法 8.4^①，基本设计与上一章相同。与采用多进程的并发设计一样，系统管理员应安排在系统启动时就创建一个主线程。主线程将一直处于运行状态，等待来自客户端的新连接请求。一旦收到来自某个客户端的连接请求，主线程将为之创建一个新的从线程，以后与该客户的所有通信将由从线程处理。在某个从线程完成与客户的交互后，该从线程将被终止。

12.2 Linux 线程概述

我们将一次独立的计算抽象为一个执行线程 (a thread of execution)，而一个进程可以包含一个或多个线程。Linux 中的线程符合 POSIX 线程标准，即 1003.1c，该标准也为大多数 UNIX 系统所采用。Linux 线程具有如下特征：

动态创建。调用函数 `pthread_create` 可创建一个新线程。只是操作系统对并发线程数有一个上限限制，就像限制并发进程数一样。

并发执行。所有线程就像在同一时间执行一样。在多处理器机器上，系统可为每个线程分配一个 CPU，多个线程可以并行执行。

抢先。操作系统为线程分配 CPU 资源。如果活动线程数超过可用的 CPU 数（例如单处理器机器），系统将自动在多个线程间调动 CPU 资源，每次只让一个线程执行一小段时间。API 中有一个函数 `sched_yield`，线程一旦调用该函数，在其时间片用完之前就主动放弃对 CPU 的调用。

私有局部变量。每个线程都有自己的私有堆栈。堆栈用于存放分配的局部变量和过程的活动记录（即关于过程调用的信息）。

共享全局变量。一个进程内的所有线程共享一组全局变量。

共享文件描述符。一个进程内的所有线程共享一组文件描述符。

协调和同步函数。线程 API 中包括协调和同步线程执行的函数。

12.3 线程的优点

多线程的进程与单线程的进程相比主要有两个优点：更高的效率和共享的存储器。效率提高源

^① 见第 81 页算法 8.4 的描述。

于上下文交换的额外开销减少。上下文交换是指操作系统将CPU从一个运行线程调度到另一个线程所需执行的指令。在线程间切换时，操作系统必须保存原先线程的状态（如寄存器中的值）并读取新线程的状态。同一个进程中的线程共享的进程状态越多，操作系统需要改变的状态就越少。因此，同一个进程中的两个线程间的切换比不同进程中的两个线程快。尤其是因为同一进程中的线程共享一个存储器地址空间，进程内的线程切换就意味着操作系统不必改变虚拟存储器映射。

线程的第二个优点是共享存储器，这对于程序员而言比效率提高更为重要。并发服务器中多个服务器副本需要相互通信或访问共享的数据，因此利用线程更容易构造并发服务器。另外，利用线程也更容易构造监控系统。尤其是因为它们共享存储器，一个服务器中的线程可将统计数据留在共享存储器中，从而使监控线程可把从线程的活动情况报告给系统管理员。稍后我们将给出一个监控系统的实例。

12.4 线程的缺点

虽然多线程提供的优点是单线程的进程所欠缺的，但它们也有一些缺点。其中最重要的一点是，由于线程间共享存储器和进程状态，一个线程的动作可能会对同一个进程内的其他线程产生影响。例如，当两个线程试图在同一时刻访问同一个变量时，它们之间就会产生相互干扰。

我们知道线程 API 提供了协调线程间动作的函数。但是，许多将指针返回给一个静态数据项的库函数不是线程安全（thread safe）的。也就是说，如果多个线程试图并发调用该库函数，返回结果是不可预知的。我们以 `gethostbyname` 库函数为例加以说明，该函数可用于解析域名并返回相应的 IP 地址。如果两个线程并发调用 `gethostbyname`，后一次解析的结果将覆盖前一次结果。因此，如果多个线程调用某个库函数，线程之间必须加以协调，确保某个时刻只有一个线程调用该库函数。

另一个缺点是缺乏健壮性。在单线程的服务器中，如果某一个并发服务器的副本造成服务器出错（例如一个非法的存储器引用），操作系统只会终止引发故障的进程。但是在多线程的服务器中，如果一个线程使服务器出错，操作系统将终止整个进程。

12.5 描述符、延迟和退出

线程和进程间的关系容易被混淆，特别是那些具有单线程进程编程经验的程序员更易将两者混淆。除了静态资源（例如全局变量），许多动态分配的资源都是与进程相关的（而非单独的线程）。例如，由于文件描述符资源属进程所有，一旦某个线程打开了一个文件，同一个进程中的其他线程也可以使用同一个描述符访问文件。此外，如果某个线程关闭了一个文件描述符，意味着整个进程内的该描述符已被关闭（即该进程中的其他线程不能再访问此描述符）。

类似地，虽然有些操作系统函数只会影响调用它的线程，但有些函数会影响整个进程。例如，如果一个线程进行 I/O 调用（例如调用 `read`）时被阻塞，只有一个线程会被阻塞。但如果某个线程调用了 `exit` 函数，整个进程将立刻退出。也就是说，`exit` 函数是与整个进程而非单个线程相关的。关于以上讨论的要点如下：

虽然一些系统函数只影响调用线程，但有些函数（例如 `exit`）会影响整个进程。

12.6 线程退出

如果一个线程不能调用 `exit` 终止整个进程，它该如何终止自己而不会终止进程内的其他线程呢？有两种方法：一种是在线程的顶级过程（即线程一开始调用的过程）返回时终止该线程；另一种是调用 `pthread_exit` 终止该线程。总结如下：

线程 API 中包括一些只对线程起作用的函数。例如，线程可调用 `pthread_exit`（或从其顶级过程中返回时）终止自己，而不会影响进程中的其他线程。

12.7 线程协调和同步

由于各个线程按照自己的步调并发运行，线程间的同步是必须的。例如，如果一个线程在进行 I/O 操作时被阻塞，延迟时间长度取决于操作系统和下层的硬件；无法预知在该线程的延迟期间其他线程的行为或者它们将执行哪些指令。因此，程序员希望能协调线程的执行，从而引入了一些函数调用。Linux 提供了三种同步机制：互斥（mutex）、信号量（semaphore）和条件变量（condition variable）。

12.7.1 互斥

线程使用互斥可对共享数据项进行排它性访问。通过调用 `pthread_mutex_init` 可动态地初始化一个互斥^①；程序员可为每个需要保护的数据项都安排一个独立的互斥。一个互斥初始化之后，线程在使用数据项前必须调用 `pthread_mutex_lock`，使用完后再调用 `pthread_mutex_unlock`。这样可确保某一个时刻只有一个线程访问数据项。在一个互斥中，第一个调用 `pthread_mutex_lock` 的线程将不受阻挡地继续执行。但在该线程使用数据项的过程期间，后续调用 `pthread_mutex_lock` 的其他线程将被阻塞，直到第一个线程使用完数据项并调用了 `pthread_mutex_unlock` 后，另一个线程才得以使用该数据项。此时，操作系统将使其中一个阻塞等候的线程回到运行状态。总结如下：

互斥是使线程同步的机制之一。每个互斥与一个数据项相关；任何时刻只有一个线程访问受互斥保护的数据。

12.7.2 信号量

信号量（有时被称为计数信号量）是一种同步机制，它用于系统中有 N 个资源可用的情况，是对互斥机制的一种推广。信号量允许 N 个线程同时执行，而不是像互斥一样在某个时刻只允许一个线程执行通过临界区。

类似互斥，信号量可以动态启动。函数 `sem_init` 初始化一个信号量；它带有一个参数 N 表示可用的资源数。初始化一个信号量后，一个线程在使用一个资源前必须调用 `sem_wait`，并在用完后调用 `sem_post` 返回资源。N 个线程都可在调用 `sem_wait` 后不会受到影响——每个线程都可在调用后继续执行。但是，如果再有其他线程调用 `sem_wait`，它们就将被阻塞。这些线程将一直处于阻塞状

① 通过指派常量 `PTHREAD_MUTEX_INITIALIZER` 可静态初始化一个互斥。

态，直到某个运行线程调用了此信号量上的sem_post后，其中一个阻塞的线程才得以继续运行。总结如下：

信号量是一种线程同步机制，是互斥机制的一种推广形式。任一时刻，至多有N个线程能够访问受到信号量（初始化计数器为N）保护的资源。

12.7.3 条件变量

最复杂和难以理解的一种同步机制被称为条件变量。实质上，只有一种情况需要条件变量：

- 一组线程使用互斥对同一个资源提供排它性访问。
- 一旦某个线程获得资源，它需要等待一个特定的条件发生。

如果没有条件变量，面对这种情况，程序必须使用一种忙等待（busy waiting）的形式：每个线程要重复地获得一个互斥，测试条件是否满足，然后释放互斥。条件变量允许线程原子地完成这两个动作，从而令等待更为高效。在被条件变量阻塞前，线程获得了一个互斥。在线程调用pthread_cond_wait以便等待某个条件变量时，线程同时指定了等待的条件变量和所拥有的互斥。操作系统将同时释放线程拥有的互斥并阻塞线程。

线程执行pthread_cond_wait后将处于阻塞状态，直到其他线程给此变量发信号^①时才被唤醒。给条件变量发信号有两种形式，差别在于多个等待线程被处理的方式不同。即使有多个线程等待同一条件变量，函数pthread_cond_signal只允许一个线程继续执行；而函数pthread_cond_broadcast却让所有被阻塞的线程都可继续执行。在操作系统允许某个线程继续执行前，它将在线程被解除阻塞的同时再次获得阻塞前曾经有过的互斥。换句话说，等待条件变量意味着暂时放弃互斥，然后在得到发给该变量的信号时自动地重获互斥。因此，在某个线程因某个条件变量阻塞时，其他线程仍然可以获得互斥——仍然可以在临界区内继续执行。

条件变量是一种与互斥配合使用的线程同步机制，在线程等待一个条件时，它将暂时放弃所拥有的互斥；在条件变量得到信号后线程又将重新获得互斥。

12.8 使用线程的服务器实例

下面将举例说明一个使用线程的服务器。上一章中我们用多个进程实现了ECHO服务，为便于对比，此处我们也选择实现ECHO服务。本章中的多线程服务器采用同一个并发的、面向连接的算法；只是在实现中稍有差别。初始化后，主线程执行主程序进入一个循环，在循环的accept调用处线程将被阻塞，等待一个TCP连接的到来。在连接到达时，主线程继续执行，调用pthread_create创建一个新线程来处理连接。然后主线程继续循环，再次被阻塞而等待新连接的到来。新创建的线程执行TCPechod过程，该过程中包括一个循环，新线程将反复从TCP连接读数据，然后将数据返回发送者。文件TCPmethod.c的代码如下。

```
/* TCPmethod.c - main, TCPechod, prstats */
```

^① 在一个条件变量上等待的线程可能在收到信号后阻塞被解除。因此在实际应用中，程序员会设计一个循环来反复调用pthread_cond_wait，只有当线程等待的条件满足时才能退出循环。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>

#include <sys/types.h>
#include <sys	signal.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
#include <sys/errno.h>
#include <netinet/in.h>

#define      QLEN          32 /* maximum connection queue length */
#define      BUFSIZE        4096

#define      TINTERVAL     5    /* secs */

struct {
    pthread_mutex_t      st_mutex;
    unsigned int         st_concount;
    unsigned int         st_contotal;
    unsigned long        st_contime;
    unsigned long        st_bytecount;
} stats;

void prstats(void);
int TCPechod(int fd);
int errexit(const char *format, ...);
int passiveTCP(const char *service, int qlen);

/* -----
 * main - Concurrent TCP server for ECHO service
 * -----
 */
int
main(int argc, char *argv[])
{
    pthread_t      th;
    pthread_attr_t   ta;
    char      *service = "echo"; /* service name or port number */
    struct sockaddr_in fsin; /* the address of a client */
    unsigned int alen; /* length of client's address */
}
```

```
int      msock;           /* master server socket */  
int      ssock;           /* slave server socket */  
  
switch (argc) {  
case 1:  
    break;  
case 2:  
    service = argv[1];  
    break;  
default:  
    erexit("usage: TCPEchocd [ port]\n");  
}  
  
msock = passiveTCP(service, QLEN);  
  
(void) pthread_attr_init(&ta);  
(void) pthread_attr_setdetachstate(&ta, PTHREAD_CREATE_DETACHED);  
(void) pthread_mutex_init(&stats.st_mutex, 0);  
  
if (pthread_create(&th, &ta, (void * (*) (void *))prstats, 0) < 0)  
    erexit("pthread_create(prstats): %s\n", strerror(errno));  
  
while (1) {  
    alen = sizeof(fsin);  
    ssock = accept(msock, (struct sockaddr *)&fsin, &alen);  
    if (ssock < 0) {  
        if (errno == EINTR)  
            continue;  
        erexit("accept: %s\n", strerror(errno));  
    }  
    if (pthread_create(&th, &ta, (void * (*) (void *))TCPEchocd,  
                      (void *)ssock) < 0)  
        erexit("pthread_create: %s\n", strerror(errno));  
}  
}  
  
/*-----  
 * TCPEchocd - echo data until end of file  
 *-----  
 */  
int  
TCPEchocd(int fd)  
{  
    time_t  start;  
    char     buf[BUFSIZ];
```

```

int      cc;

start = time(0);
(void) pthread_mutex_lock(&stats.st_mutex);
stats.st_concount++;
(void) pthread_mutex_unlock(&stats.st_mutex);
while (cc = read(fd, buf, sizeof buf)) {
    if (cc < 0)
        errexit("echo read: %s\n", strerror(errno));
    if (write(fd, buf, cc) < 0)
        errexit("echo write: %s\n", strerror(errno));
    (void) pthread_mutex_lock(&stats.st_mutex);
    stats.st_bytecto += cc;
    (void) pthread_mutex_unlock(&stats.st_mutex);
}
(void) close(fd);
(void) pthread_mutex_lock(&stats.st_mutex);
stats.st_contime += time(0) - start;
stats.st_concount--;
stats.st_contotal++;
(void) pthread_mutex_unlock(&stats.st_mutex);
return 0;
}

/*
 * prstats - print server statistical data
 */
void
prstats(void)
{
    time_t now;

    while (1) {
        (void) sleep(INTERVAL);

        (void) pthread_mutex_lock(&stats.st_mutex);
        now = time(0);
        (void) printf("--- %s", ctime(&now));
        (void) printf("%-32s: %u\n", "Current connections",
                      stats.st_concount);
        (void) printf("%-32s: %u\n", "Completed connections",
                      stats.st_contotal);
        if (stats.st_contotal) {
            (void) printf("%-32s: %.2f (secs)\n",

```

```
    "Average complete connection time",
    (float)stats.st_contime /
    (float)stats.st_contotal);
(void) printf("%-32s: %.2f\n",
    "Average byte count",
    (float)stats.st_bytectcount /
    (float)(stats.st_contotal +
    stats.st_concount));
}

(void) printf("%-32s: %lu\n\n", "Total byte count",
    stats.st_bytectcount);
(void) pthread_mutex_unlock(&stats.st_mutex);

}
}
```

12.9 监控

此服务器实例中新实现了一个监控机制。虽然本实例中的监控机制并不复杂，但它说明了监控程序如何使用共享程序与从线程交互。监控程序用一个执行 prstats 过程的独立线程实现。本例中，prstats 过程包含一个循环，在每一次循环过程中，监控程序打印连接的相关统计信息，然后睡眠 INTERVAL 秒。统计输出中列出了通信中的连接数、已完成通信的连接数、总连接时间和每个连接所传输的平均字节数。

监控线程和从线程使用一个共享的全局数据结构——stats 相互通信。每个从线程将各自连接的有关信息加入到 stats 结构中，而监控线程每 INTERVAL 秒提取一次信息。为确保任一时刻只有一个线程访问共享结构，服务器使用一个互斥，stats.st_mutex。线程在访问共享结构前等待互斥，并在使用完结构后释放互斥。

在一个实际的服务器中，监控程序可以让管理员以更复杂的形式与服务器交互。例如，监控线程可以不只打印统计信息，而是让管理员用键盘键入命令。因此，监控程序可按需提供信息，并可让管理员控制服务器（例如，动态设置或改变最大并发线程数）。

12.10 小结

并发服务器可在一进程内用若干线程实现。线程的主要优点是它具有较少的上下文切换开销和共享存储器的能力。线程的主要缺点是它增加了编程的复杂性。程序员必须使用同步机制协调线程对全局变量和一些库程序的访问，而且必须记住一些系统函数（如 exit）会影响整个进程而非单个线程。

深入研究

Linux 操作系统的联机手册中描述了线程函数。

习题

- 12.1 比较本章中的多线程并发服务器与上一章中用多个单线程进程实现的服务器。哪个执行快一些？运行时间如何随着并发连接数变化？
- 12.2 上一习题中，如果省略监控线程，结果会改变吗？
- 12.3 阅读 `pthread_attr_init` 函数的有关内容。为什么需要此函数？
- 12.4 如果监控线程连接到一个键盘，并且用户键入了一个 CONTROL-S（即停止输出），服务器会发生什么情况？
- 12.5 在监控程序的实例代码中，监控程序在格式化和打印统计数据时拥有互斥。修改此段代码，使监控程序只在复制 `stats` 结构时拥有互斥。
- 12.6 在监控程序例子里，使用了一个独立的线程定期打印统计信息。使用 Linux alarm 机制实现同样的功能。每种方法的优点和缺点是什么？
- 12.7 修改监控程序，使它从键盘接受命令并响应每个命令。

第 13 章 单线程、并发服务器 (TCP)

13.1 引言

上一章举例说明了大多数并发的、面向连接的服务器是如何运行的。这种服务器使用了操作系统的各种设施为每个连接创建单独的进程或线程，并允许操作系统在线程间用时间分片的方式占用处理器。本章将说明一个有趣的、但并非显而易见的设计思想，即服务器只使用单个控制线程^①，就能为若干个客户提供表面上的并发性。本章首先探讨基本概念，讨论为什么这种方法是可行的，以及在什么时候这要比使用多个线程的实现好。然后，研究单个线程如何使用 Linux 的系统调用来并发地处理多个连接。

13.2 服务器中的数据驱动处理

如果为一个请求准备响应所需的开销中，I/O 占了主导地位，则在此种应用中，服务器可使用异步 I/O 来提供客户间的表面上的并发性。这个想法很简单：让一个服务器执行线程对多个客户打开它们的 TCP 连接，并使服务器在数据到达时处理该连接。因此，服务器使用数据的到达来触发处理。

要理解这种方法为什么是可行的，考虑上一章描述的并发 ECHO 服务器。为达到并发执行，例子代码创建了一个单独的从线程（slave thread）来处理每个新的连接。从理论上讲，服务器依赖操作系统的时间分片机制在多个线程间共享 CPU，并由此在多个连接间共享 CPU。

但在实际中，ECHO 服务器几乎与时间分片无关。如果密切地观察一个并发 ECHO 服务器的执行，你就会发现通常是数据的到达控制了处理的进行。其原因与穿过互联网的数据流有关。由于下层互联网是用离散的分组交付数据的，因而数据是以突发方式（而不是以平稳的数据流方式）到达服务器的。如果客户发送数据块的方式是使每一个最终形成的 TCP 报文段各填入到单个 IP 数据报中，那么客户就加剧了这种突发行为。在服务器上，每个从线程将大部分时间花在 read 调用中，即它被阻塞以等待下一个突发数据的到达。一旦数据到达，read 调用就返回，从线程继续执行。从线程调用 write 发送数据给客户，然后调用 read 再次等待后面的数据。CPU 要能处理许多客户的请求而不减慢处理速率，就必须运行得足够快，以便在另一个从线程收到数据前它就完成了读和写。

当然，如果负荷太重，以致 CPU 不能在另一个请求到达前处理完一个请求，分时机制就将起作用。操作系统在所有有数据要处理的从线程之间切换处理器。对于仅需对每个请求进行很少处理的简单服务，执行由数据到达来驱动的机会是很高的。概括起来就是：

若并发服务器处理每个请求仅需很少的时间，通常它就按顺序方式执行，而执行由数据的到达

^① 读者要仔细区分这里所介绍的单线程实现和第 11 章所介绍的多进程实现之间的区别。在本章介绍的方法中，只有一个进程，而这个进程只有一个线程执行，在第 11 章的方法中，有多个进程，每个进程有一个线程。

驱动。只有在工作量太大，以致 CPU 不能顺序执行时，分时机制才取而代之。

13.3 用单线程进行数据驱动处理

理解并发服务器行为的顺序特征，就可以理解单个执行线程如何完成同样的任务。想像一个单服务器线程，它打开了到许多客户的TCP连接。线程将阻塞以等待数据的到达。一旦任何一个连接上有数据到达，线程就被唤醒，并处理请求和发送响应。然后它再次阻塞，等待另一个连接上更多数据的到达。只要CPU足够快地应付服务器上出现的工作负荷，使用单线程就能像使用多线程那样处理各个请求。实际上，与使用多线程或多进程的实现相比，单线程的实现较少需要在线程或进程上下文之间进行切换，因而可能处理略高些的负荷。

编写一个单线程并发服务器的关键是通过在操作系统原语 select 中使用异步 I/O。一个服务器为它必须管理的每一个连接创建一个套接字，然后调用 select 等待任一连接上数据的到达。实际上，由于 select 可在所有可能的套接字上等待 I/O，它也能同时等待新的连接。算法 8.5^①列举出了单线程服务器所使用的详细步骤。

13.4 单线程服务器的线程结构

图 13.1 给出了单线程、并发服务器的线程和套接字结构。一个执行线程管理所有的套接字。

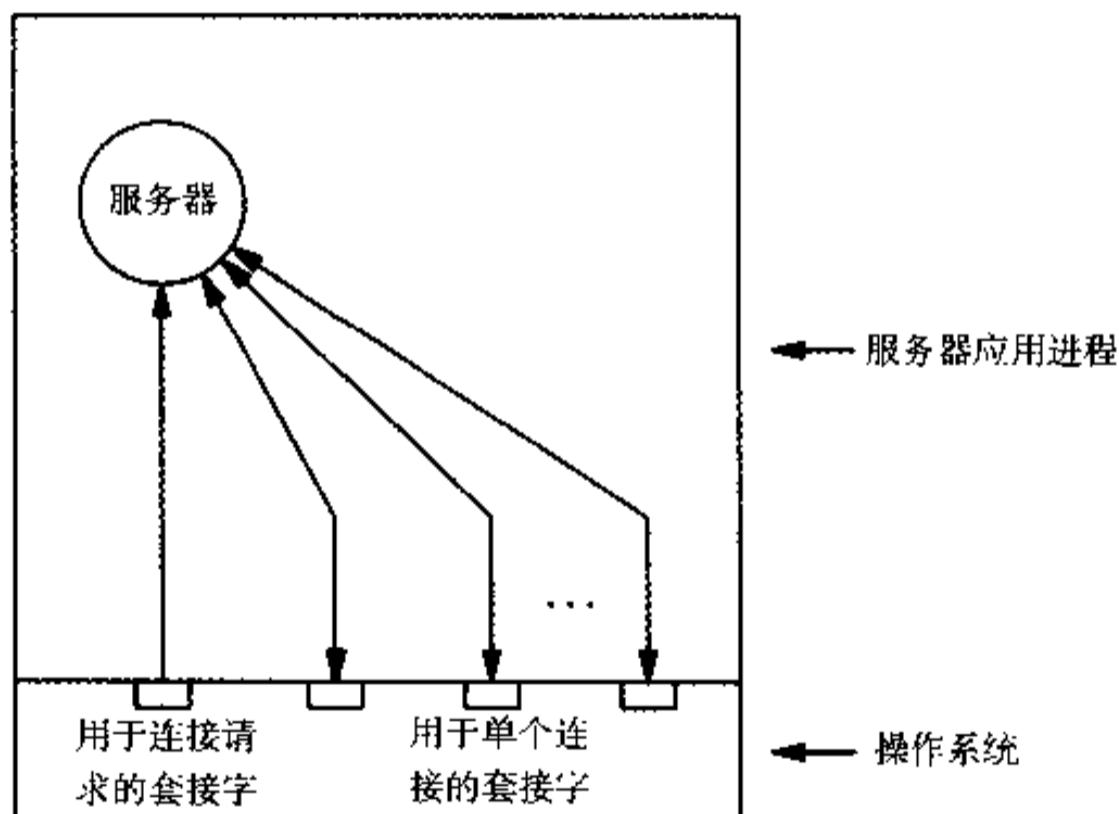


图 13.1 用单个线程完成并发的、面向连接服务器的线程结构。该线程管理多个套接字

实质上，单线程服务器必须完成主线程和从线程双方的职责。它维护一组套接字，组中的一个套接字绑定到主线程将要接受连接的熟知端口上。而组中其他每一个套接字都对应于一个连接，在此连接上一个从线程将处理请求。服务器把这一组套接字描述符作为一个参数传递给 select，并等待任何一个套接字上的活动。当 select 返回时，它返回一个屏蔽位，指明这一组描述符中的哪一个已就绪。服务器按照描述符准备就绪的指示来决定如何继续处理。

^① 算法 8.5 参见 83 页。

单线程服务器使用描述符来区分主线程和从线程的操作。如果主套接字相应的描述符准备就绪，服务器就进行主线程要做的同样的操作：它在套接字上调用 accept 以获得新连接。如果对应于一个从套接字的描述符准备就绪，服务器就进行从线程要做的同样操作：它调用 read 获取一个请求，然后作出回答。

13.5 单线程 ECHO 服务器举例

举一个例子将有助于阐明上述观点，还可以说明一个单线程怎样提供并发性。我们来研究一下文件 TCPmechod.c，它含有一个实现 ECHO 服务的单线程服务器代码：

```
/* TCPmechod.c - main, echo */

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

#include <unistd.h>
#include <string.h>
#include <stdio.h>

#define QLEN 32           /* maximum connection queue length */
#define BUFSIZE 4096

extern int      errno;
int            errexit(const char *format, ...);
int            passiveTCP(const char *service, int qlen);
int            echo(int fd);

/*
 * main - Concurrent TCP server for ECHO service
 */
int
main(int argc, char *argv[])
{
    char      *service = "echo";      /* service name or port number */
    struct sockaddr_in fsin;        /* the from address of a client */
    int        msock;              /* master server socket */
    fd_set    rfds;                /* read file descriptor set */
    fd_set    afds;                /* active file descriptor set */
    int        alen;                /* from-address length */
    int        fd, nfds;
```

```

switch (argc) {
    case 1:
        break;
    case 2:
        service = argv[1];
        break;
    default:
        erexit("usage: TCPmechod [port]\n");
}

msock = passiveTCP(service, QLEN);

nfds = getdtablesize();
FD_ZERO(&afds);
FD_SET(msock, &afds);

while (1) {
    memcpy(&rfds, &afds, sizeof(rfds));

    if (select(nfds, &rfds, (fd_set *)0, (fd_set *)0,
               (struct timeval *)0) < 0)
        erexit("select: %s\n", strerror(errno));
    if (FD_ISSET(msock, &rfds)) {
        int ssock;

        alen = sizeof(fsin);
        ssock = accept(msock, (struct sockaddr *)&fsin,
                      &alen);
        if (ssock < 0)
            erexit("accept: %s\n",
                   strerror(errno));
        FD_SET(ssock, &afds);
    }
    for (fd=0; fd<nfds; ++fd)
        if (fd != msock && FD_ISSET(fd, &rfds))
            if (echo(fd) == 0) {
                (void) close(fd);
                FD_CLR(fd, &afds);
            }
}
/* -----
 * echo - echo one buffer of data, returning byte count
 * -----
 */

```

```

/*
int
echo(int fd)
{
    char      buf[BUFSIZ];
    int       cc;

    cc = read(fd, buf, sizeof buf);
    if (cc < 0)
        errexit("echo read: %s\n", strerror(errno));
    if (cc && write(fd, buf, cc) < 0)
        errexit("echo write: %s\n", strerror(errno));
    return cc;
}

```

类似并发实现中的主服务器线程，单线程服务器一开始就在熟知端口上打开一个被动套接字。它使用系统函数 getdtablesize 来决定描述符的最大个数，然后使用 FD_ZERO 和 FD_SET 创建一个比特向量 (bit vector)，对应于希望测试的套接字描述符。然后服务器进入一个无限循环，在循环中调用 select，等待一个或多个描述符准备就绪^①。

如果主描述符 (master descriptor) 准备就绪，服务器就调用 accept 获取一个新的连接。它将新连接用的描述符加入到它所管理的那些描述符中，并继续等待更多描述符的动作。如果一个从描述符 (slave descriptor) 准备就绪，服务器就调用过程 echo，该过程再调用 read 从这一连接获取数据，并调用 write 将数据发回客户。如果其中某个从描述符报告了文件结束 (end-of-file) 的条件，服务器就关闭该描述符，并使用宏 FD_CLR 从 select 使用的那组描述符中删除它。

13.6 小结

并发服务器中的执行通常是由数据到达驱动的，而不是由下层操作系统中时间分片机制驱动的。在服务仅需很少处理的情况下，单线程的实现可使用异步 I/O 管理到多个客户的连接，这种实现与使用多个线程的实现一样高效。

在单线程实现中，一个执行线程完成了主线程和从线程的职责。当主套接字准备就绪时，服务器接受一个新的连接。当其他任何一个套接字准备就绪时，服务器读取一个请求，并发送响应。一个单线程 ECHO 服务器例子说明了单线程如何获得并发性并展现了编程细节。

深入研究

一个好的协议规约对实现是没有约束的。例如，本章描述的单线程服务器，实现了 Postel [RFC 862] 定义的 ECHO 协议。而第 11 章和第 12 章展示了对同一协议规约的另一种实现方法。这两个实现分别使用了多进程（每个进程有单个执行线程）和多线程进行处理。

^① 因为 select 在返回时，清除了所有没有能进行读或写操作的描述符，所以程序将用到的活动描述符复制到 rds 中，以供每次循环使用。

习题

- 13.1 做一个实验，证明本章的 ECHO 服务器例子可并发处理多个连接。
- 13.2 将本章讨论的实现用于 DAYTIME 服务有意义吗？为什么有或为什么没有？
- 13.3 阅读联机文档，找出传给 select 的描述符列表的确切表示。编写 FD_SET 和 FD_CLR 宏。
- 13.4 在一个具有多个处理器的计算机上，比较单线程和多进程服务器两种实现的性能。在什么情况下，单个线程实现比多进程实现更好（或相同）？
- 13.5 假定大量客户（如 100 个）同时访问本章中的服务器例子。请解释每个客户可观察到的情况。
- 13.6 单个线程服务器在不停地处理来自某个客户的请求时，可曾剥夺另一个客户的服务？多线程服务器或多个单线程进程服务器可曾出现相同情况？请解释。

第 14 章 多协议服务器 (TCP, UDP)

14.1 引言

前面一章描述了如何构造一个单线程服务器，使用异步I/O以便在多个连接上提供表面上的并发性。本章将扩展这个概念，展示一个单执行线程服务器如何可以适用于多个传输协议。本章将给出一个服务器例子，它既可以用TCP也可以用UDP来提供DAYTIME服务。本章通过这个例子来说明前面的概念。尽管这个服务器的例子循环地处理请求，但其基本概念可以直接推广到适合并发处理请求的服务器。

14.2 减少服务器数量的动机

在大多数情况下，一个给定的服务器处理针对一个服务的请求，这些请求是通过一个传输协议发来的。例如，一个提供DAYTIME服务的计算机系统往往要运行两个服务器，一个服务器处理来自UDP的请求，而另一个服务器则处理来自TCP的请求。

为每个协议使用一个单独服务器的主要优点是便于控制：系统管理员可以通过控制系统所运行的服务器来很容易地控制计算机所提供的协议。每种协议使用一个服务器的主要缺点就是重复。由于许多服务器既可以通过UDP也可以通过TCP来访问，因此每种服务都可以需要两个服务器。此外，由于UDP和TCP服务器都使用相同的基本算法来计算响应，它们都要包含执行计算所需要的代码。如果两个程序都含有执行某个给定服务的代码，软件管理和排错就变得冗长乏味了。当改正程序中的差错或者为适应新发布的系统软件而需要改变服务器时，程序员必须要保证两个服务器程序一样。此外，为保证TCP和UDP服务器在任何时间都能够准确地提供相同版本的服务，系统管理员必须小心谨慎地协调它们的执行。为每个协议单独运行服务器的另一个缺点来自对资源的使用：多个服务器进程（或线程）不必要地消耗了进程表的许多项目以及其他系统的资源。你只要回忆一下TCP/IP标准所定义的那几十种服务，问题的严重性就很清楚了。

14.3 多协议服务器的设计

一个多协议服务器由一个单执行线程构成，这个线程既可以在TCP也可以在UDP之上使用异步I/O来处理通信。服务器最初打开两个套接字：一个使用无连接的传输（UDP），一个使用面向连接的传输（TCP）。接着，服务器使用异步I/O等待两个套接字之一就绪。如果TCP套接字就绪，就说明客户请求了一个TCP连接。服务器就使用accept获得新的连接，并在这个连接上与客户通信。如果UDP套接字就绪，就说明客户以UDP数据报的形式发来一个请求。服务器就用recvfrom读取这个请求，并记录此发送者的端点地址。当服务器计算出响应后，服务器用sendto将响应发回给客户。

14.4 进程结构

图 14.1 说明了一个循环的、多协议服务器的进程结构。一个单执行线程接收来自多种传输协议的请求

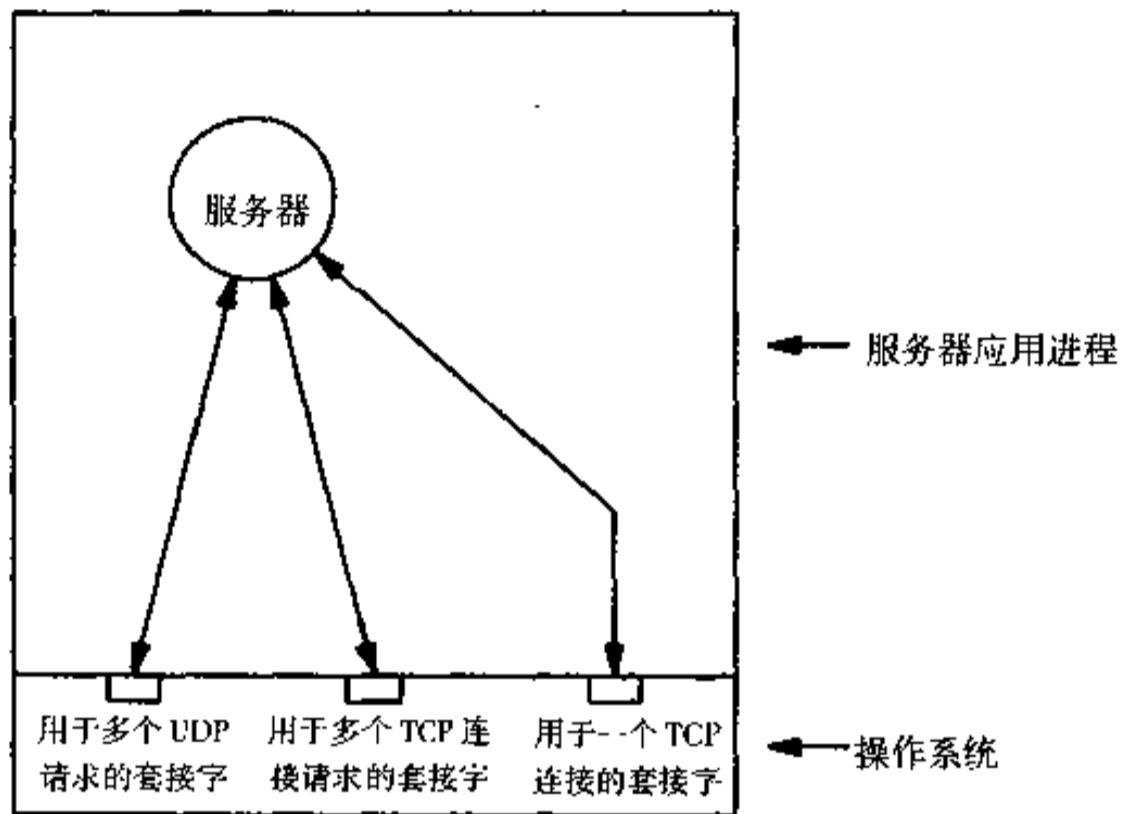


图 14.1 一个循环的、多协议服务器的进程结构。在任何时候，执行线程至多打开三个套接字：一个用于 UDP 请求，一个用于 TCP 请求，还有一个临时的套接字用于一个 TCP 连接

在任何时候，一个循环的、多协议服务器至多打开三个套接字。最初，服务器打开一个 UDP 套接字以接受 UDP 传入数据报，第二个套接字接受传入 TCP 连接请求。当一个数据报到达 UDP 套接字后，服务器计算出响应，并通过同一个套接字将其发回给客户。当一个 TCP 连接请求到达 TCP 套接字后，服务器使用 accept 获得这个新的连接。accept 为这个连接创建第三个套接字，服务器使用这个新的套接字与客户通信。一旦交互结束，服务器将关闭第三个套接字，并等待另两个套接字被激活。

14.5 多协议 DAYTIME 服务器的例子

程序 daytimed.c 说明了一个多协议服务器是如何工作的。它由一个线程构成，这个线程可以同时为 UDP 和 TCP 提供 DAYTIME 服务。

```
/* daytimed.c - main */

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

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

```
#include <string.h>

extern int      _errno;

int      daytime(char buf[]);
int      errexit(const char *format, ...);
int      passiveTCP(const char *service, int qlen);
int      passiveUDP(const char *service);

#define MAX(x, y)          ((x) > (y) ? (x) : (y))

#define QLEN               32
#define LINELEN            128

/* -----
 * main - Iterative server for DAYTIME service
 * -----
 */
int
main(int argc, char *argv[])
{
    char      *service = "daytime"; /* service name or port number */
    char      buf[LINELEN+1];       /* buffer for one line of text */
    struct sockaddr_in fsin;        /* the request from address */
    unsigned int alen;             /* from-address length */
    int       tsock;               /* TCP master socket */
    int       usock;               /* UDP socket */
    int       nfds;
    fd_set   rfds;                /* readable file descriptors */

    switch (argc) {
    case 1:
        break;
    case 2:
        service = argv[1];
        break;
    default:
        errexit("usage: daytimed [port]\n");
    }

    tsock = passiveTCP(service, QLEN);
    usock = passiveUDP(service);
    nfds = MAX(tsock, usock) + 1; /* bit number of max fd */

    FD_ZERO(&rfds);

    while (1) {
```

```
        FD_SET(tsock, &rfds);
        FD_SET(usock, &rfds);

        if (select(nfds, &rfds, (fd_set *)0, (fd_set *)0,
                   (struct timeval *)0) < 0)
            erexit("select error: %s\n", strerror(errno));
        if (FD_ISSET(tsock, &rfds)) {
            int         ssock;           /* TCP slave socket */

            alen = sizeof(fsin);
            ssock = accept(tsock, (struct sockaddr *)&fsin,
                           &alen);
            if (ssock < 0)
                erexit("accept failed: %s\n",
                       strerror(errno));
            daytime(buf);
            (void) write(ssock, buf, strlen(buf));
            (void) close(ssock);
        }
        if (FD_ISSET(usock, &rfds)) {
            alen = sizeof(fsin);
            if (recvfrom(usock, buf, sizeof(buf), 0,
                         (struct sockaddr *)&fsin, &alen) < 0)
                erexit("recvfrom: %s\n",
                       strerror(errno));
            daytime(buf);
            (void) sendto(usock, buf, strlen(buf), 0,
                          (struct sockaddr *)&fsin, sizeof(fsin));
        }
    }

/*
 * daytime - fill the given buffer with the time of day
 */
int
daytime(char buf[])
{
    char      *ctime();
    time_t    now;

    (void) time(&now);
    sprintf(buf, "%s", ctime(&now));
}
```

daytimed有一个可选的参数，它允许用户指明服务名或协议端口号。如果用户没有提供这个参数，daytimed 使用服务 daytime 的端口号。

在分析参数之后，daytimed 调用 passiveTCP 和 passiveUDP 创建两个被动套接字，它们分别使用 UDP 和 TCP。这两个套接字使用相同的服务，并且对大多数服务来说，它们都使用相同的协议端口号。可以认为这两个是主套接字 (master socket)。服务器一直使它们打开着，所有来自客户的最初的联系都要通过这两者之一来进行。对 passiveTCP 的调用要指明系统必须使连接请求排队的长度能够达到 QLEN。

服务器创建主套接字之后，便准备使用 select，以便等待其中之一或两者同时 I/O 准备就绪。首先，它把变量 nfds 设置为两个套接字中较大的那个，以此作为描述符比特屏蔽码中的索引，它还把比特屏蔽码（变量 rfd）清零。接着，服务器进入到一个无限循环之中。在每次循环中，它使用宏 FD_SET 构造比特屏蔽码，其置 1 的比特对应于两个主套接字的描述符。接着便使用 select 等待这二者之一的输入激活。

当 select 调用返回时，就说明主套接字之一或两者都就绪了。服务器使用宏 FD_ISSET 检查 TCP 套接字和 UDP 套接字。服务器必须对两个套接字都进行检查，因为，如果 UDP 数据报恰巧与 TCP 连接请求同时到达，这两个套接字都将就绪。

若 TCP 套接字就绪，就意味着某个客户发起了一个连接请求。服务器使用 accept 建立这个连接。accept 返回一个新的、临时的、只用于新连接的套接字的描述符。服务器调用 daytime 计算响应，使用 write 将这个响应通过新连接发送出去，之后，使用 close 终止连接并释放资源。

若 UDP 准备就绪，就意味着某个客户发送了一个数据报来获取 DAYTIME 响应。服务器调用 recvfrom 读取传入数据报，并记录下客户的端点地址。它也使用过程 daytime 计算响应，之后便调用 sendto 将响应发回给客户。因为对所有的通信，服务器都使用主 UDP 套接字，所以它在发送完 UDP 响应后并不调用 close。

14.6 共享代码的概念

我们的服务器例子说明了一个重要概念：

一个多协议服务器的设计允许设计者创建一个单一的过程，此过程响应某个给定服务的请求，响应该过程的调用，而不必关心这些请求是来自 UDP 还是 TCP。

在我们的 DAYTIME 例子中，这段共享的代码只占用了几行，并被放到一个叫做 daytime 的过程里。然而，在大多数实际的服务器中，计算响应所需的代码可能有几百或者上千行之多，并且往往还要调用许多过程。将代码放置在一个可以共享的地方会使维护更容易，还可以保证两种传输协议所提供的服务是一致的。

14.7 并发多协议服务器

如同先前所展示的单协议 DAYTIME 服务器^①，我们的多协议 DAYTIME 服务器的例子使用了一种循环的方法来处理请求。之所以采用这种循环的方案，其理由也同前面所说的那个服务器的情况

^① 第 10 章包含了一个循环的、面向连接的 DAYTIME 服务器。

一样：对每个请求，DAYTIME 服务所执行的计算是很少的，这样，一种循环的服务器就足够了。

若每个请求要求更多的计算，对这样的服务，一种循环的实现也许就够了。在这种情况下，这种多协议设计可以扩展成并发地处理请求。在最简单的情况下，一个多协议的服务器可以创建一个新的线程或进程，以便并发地处理每个 TCP 连接，同时，它还循环地处理 UDP 的请求。多协议设计也可以扩展成使用第 13 章所述的实现方法，这种实现对各个请求提供了表面上的并发性，而这些请求来自于 TCP 连接或 UDP。

14.8 小结

多协议服务器允许设计者将某个给定服务的所有代码封装到一个程序里，这样就消除了重复，并且也更容易协调各种变化。这种多协议服务器由一个单执行线程构成。这个线程为 UDP 和 TCP 打开主套接字，并且使用 select 等待二者之一或两个套接字就绪。若 TCP 套接字就绪，服务器就接受这个新的连接并处理使用此连接的请求。若 UDP 套接字就绪，服务器就读取请求并响应之。

本章所说明的多协议服务器的设计可以扩展成允许并发处理 TCP 连接，更重要的是，它可以扩展成并发地处理请求，而不管这些请求是来自 TCP 还是 UDP。

多协议服务器使用一个单线程来计算服务的响应，消除了代码的重复。而且多协议服务器对系统资源的要求比多个单独的服务器要少（例如，要求的进程数量少了）。

深入研究

Reynolds 和 Postel [RFC 1700] 列出了一些应用协议以及分配给它们的 UDP 和 TCP 协议端口号。

习题

- 14.1 把本章的服务器的例子扩展为并发地处理请求。研究一些 TCP/IP 所定义的最常用的服务。你能找出这样一个多协议服务器吗？它不能共享计算响应的代码。试解释原因。
- 14.2 本章的例子代码允许用户以参数的形式指明服务名或协议端口号，并且在为服务创建被动套接字时使用了这个参数。有没有一种服务的例子，它对 UDP 和 TCP 使用不同的协议端口号？试修改代码，使它允许用户对每个协议分别指明不同的协议端口号。
- 14.3 我们的服务器的例子不允许系统管理员控制服务器使用哪个协议。试修改服务器，使其包含一个参数，这个参数允许管理员指明是只针对 TCP、UDP，还是对两者都提供服务。
- 14.4 考虑一个网点，它决定通过某种授权机制实现安全性。这个网点为每个服务器提供一个授权了的客户的列表，并立下规则：对那些不在表中的机器所发来的请求，服务器将不予理睬。试在这个多协议服务器的例子中实现这种授权机制。（提示：仔细查看 socket 函数，看看对 TCP 该怎么办。）

第15章 多服务服务器（TCP, UDP）

15.1 引言

第13章描述了如何构造一个单线程服务器，它使用异步I/O在多个连接之间提供表面上的并发性。第14章说明一个多协议服务器如何同时在TCP和UDP传输协议之上提供服务。本章将扩展这些概念，把前面几章所讨论的循环的和并发的服务器的设计方法结合起来。本章还要阐述单个服务器如何才能提供多种服务，并通过一个例子来说明这一思想，这个例子服务器能用一个单控制线程来处理一组服务。

15.2 合并服务器

在大多数情况下，程序员为每个服务设计一个单独的服务器。前面几章中各个服务器的例子说明了单服务的方法——每个服务器在一个熟知端口上等待，并回答与该端口相关联的服务的请求。因此，一台计算机往往为DAYTIME服务运行一个服务器，而为ECHO服务又运行另一个服务器，如此等等。我们在前面一章里讨论了这样的问题：一个使用多协议的服务器如何有助于节约系统资源，并使程序维护更容易。把多个服务合并到一个多服务服务器（multiservice server）中的动机，同设计多协议服务器的动机是一样的，它们具有相同的优点^①。

若为每个服务创建一个服务器，为估计这种方法的开销，你需要检查一下业已标准化了的服务有多少。TCP/IP定义了非常多的简单服务，这些服务是用来帮助对计算机网络进行测试、排错和维护的。前面几章中已有几个例子，如DAYTIME、ECHO以及TIME等，但除此之外还有很多其他服务。一个系统，如果为每个标准化了的服务运行一个服务器，尽管它们中的多数可能根本不会收到请求，但系统中可能有几十个服务器进程。因此，把许多服务结合到一个服务器进程中可以显著地减少正在执行的进程的数量^②。而且，因为许多小的服务可以由一个简单的计算完成，这样，一个服务器中的大多数代码是用来处理接收请求和发送应答的，将许多服务结合进一个服务中可以减少所需要的总的代码数量。

15.3 无连接的、多服务服务器的设计

多服务服务器既可使用无连接的也可使用面向连接的传输协议。图15.1说明了一个无连接的、多服务服务器的一种可能的进程结构。

① 多服务服务器的缺点是可靠性低，因为它引入了单点故障。

② 因为Linux实现限制了单个进程可以打开的套接字的最大数目。单个服务器也许不能提供所有的服务。然而，如果一个进程可以打开N个套接字，使用多服务服务器能以因子N减少所要求的进程数目。

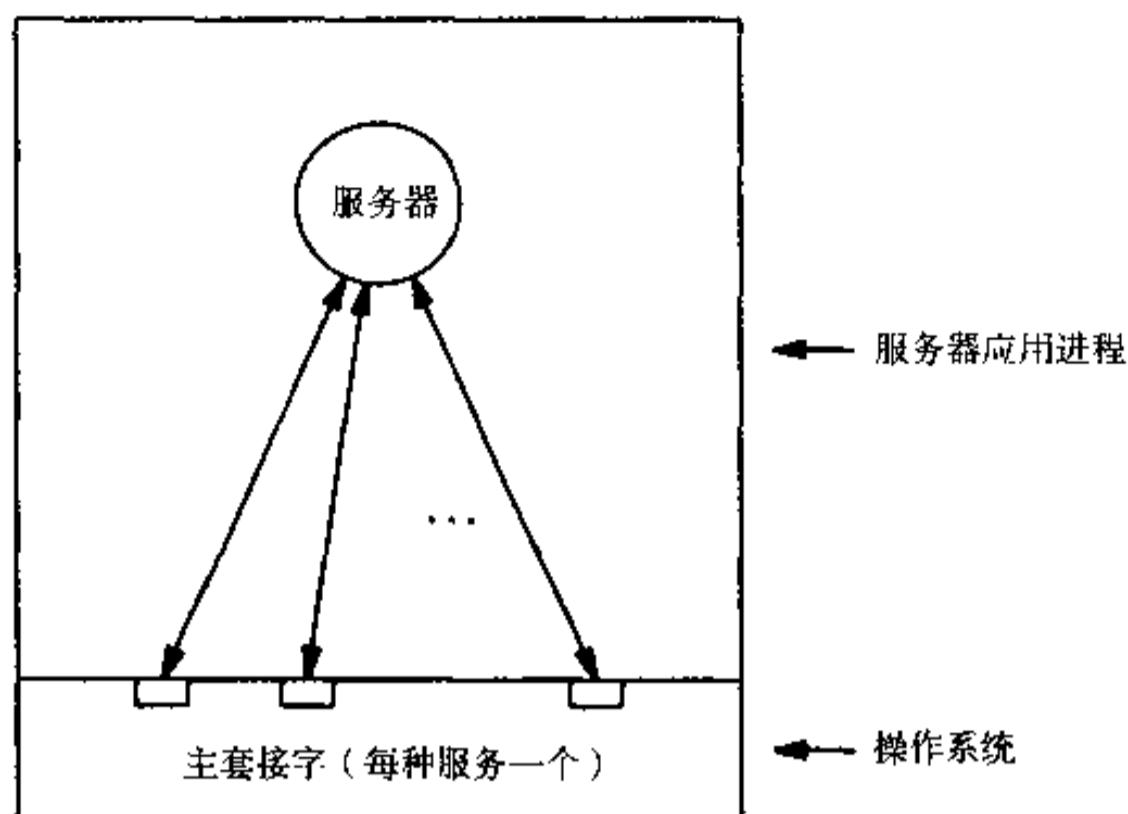


图 15.1 一个循环的、无连接的、多服务服务器。单控制线程在多个套接字上等待数据报，每个套接字都各自对应一种服务

如图 15.1 所示，一个循环的、无连接的、多服务服务器往往由一个控制线程以及提供服务所需要的全部代码所组成。这个服务器打开一组套接字，并将每个套接字与一个熟知端口绑定，每个端口与一个所提供的服务相对应。服务器使用一个很小的表将套接字映射到服务上。对每个套接字描述符，这个表记录了处理这个服务（由这个套接字提供的）的过程的地址。服务器使用 select 系统调用等待任一套接字上的数据报的到来。

当一个数据报到达后，服务器调用合适的过程，计算出响应，并将应答发送出去。由于映射表记录了每个套接字所提供的服务，服务器可以很方便地将套接字描述符映射到处理这个服务的过程上。

15.4 面向连接的、多服务服务器的设计

面向连接的、多服务服务器也可以遵照一种循环的算法。从原理上说，这样一个服务器同一组循环的、面向连接的服务器执行相同任务。更准确地说，就在一个多服务服务器中，单个执行线程取代了一组面向连接的服务器中的多个主服务器线程。在顶级 (top level)，这个多服务服务器使用异步 I/O 处理任务。图 15.2 显示了这种进程结构。

当这个多服务服务器开始执行时，它先为它所提供的每个服务创建一个套接字，并将该套接字绑定在这个服务的熟知端口上，接着，使用 select 等待任一套接字上的传入连接请求。只要这些套接字中有一个就绪，服务器就调用 accept 获得这个刚刚到来的新连接。accept 为这个传入连接创建了一个新的套接字。服务器使用这个新的套接字与客户交互，之后便将其关闭。因此，除了每个服务有一个主套接字外，服务器在任何时候最多只有一个打开的附加套接字。

如同无连接服务器的情况，服务器保持着一个映射表，这样就可决定如何处理每个传入连接。当服务器启动时，它分配了主套接字。对每个主套接字，服务器都在映射表中增加一个条目，这个映射表指明了套接字号以及实现这个（由这个套接字所提供的）服务的过程。在为每个服务分配了一个主套接字后，服务器调用 select 等待连接。一旦连接到达，服务器使用映射表来确定要调用众

多内部过程中的哪一个，由这个过程处理客户所请求的服务。

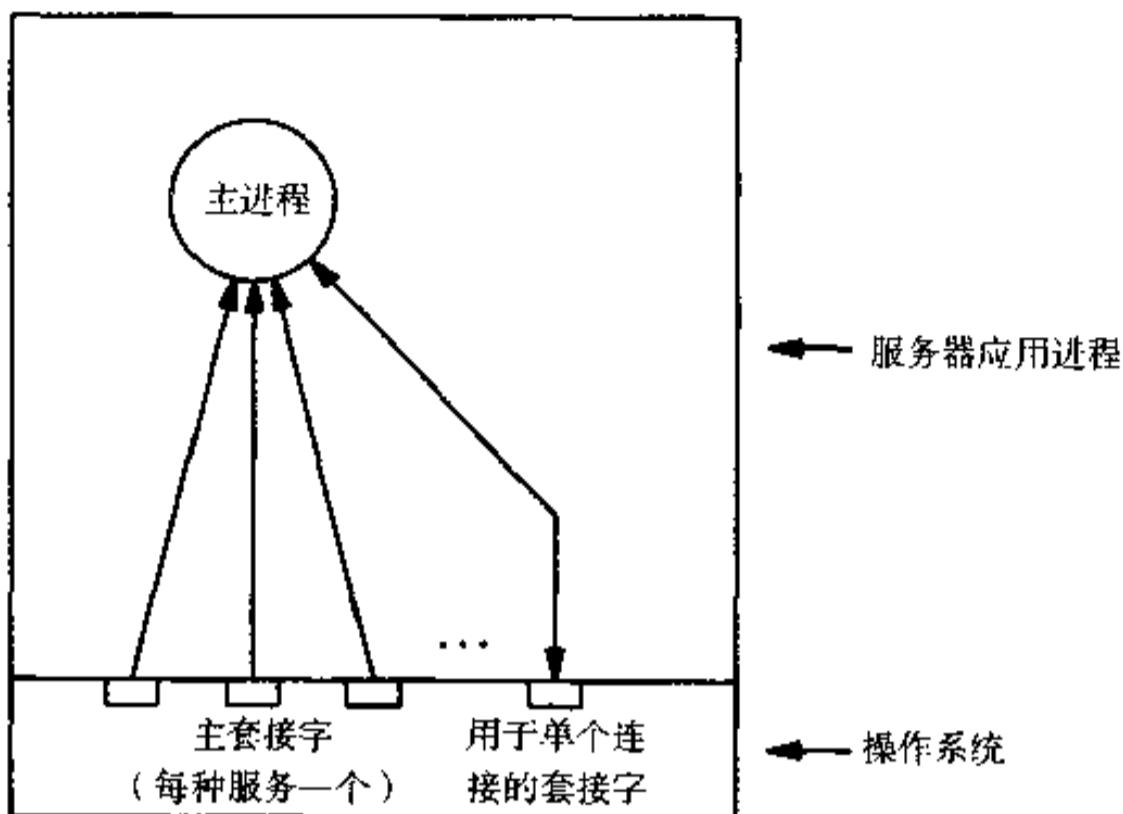


图 15.2 一种循环的、面向连接的多服务服务器的进程结构。在任何时候，对每个服务，单控制线程只有一个打开的套接字，并且最多只有一个附加的套接字用来处理某个特定的连接

15.5 并发的、面向连接的、多服务服务器

当一个连接请求到达时，多服务服务器就调用一个过程，接受并且直接处理（使服务器循环执行）这个新的连接，或者，它也可以创建一个新的从进程（slave process）来处理这个新连接（使服务器并发执行）。实际上，一个多服务服务器程序可以设计成循环地处理某些服务，而对其他的一些服务则并发地处理；程序员并不需要对所有服务都采用单一的方式。如同上一章所示的并发服务器，并发性可以通过多个单线程的进程来实现，也可以通过一个多线程的进程来实现。图 15.3 显示了一种多服务服务器的进程/线程结构，它使用了一种并发的、面向连接的实现方法。

在循环方式的实现里，一旦过程同客户的通信结束，它将关闭这个新连接。而在并发的方式里，主服务器进程一旦创建了从进程就立即关闭这个连接，而在从进程中，这个连接继续保持打开。这个从进程就像一个常规的、面向连接的服务器中的从进程一样工作。它在这个连接之上与客户通信，接受请求并发送应答。当结束交互后，该从进程关闭套接字，终止与客户的通信，然后退出。

15.6 单线程的、多服务服务器的实现

用单个执行线程管理多服务服务器中的所有活动，这种设计尽管并不多见，但却是可能的，它就像第13章所讨论的服务器。不同于为每个传入连接创建一个从线程/线程，服务器把为每个新连接所分配的套接字加入到 select 调用所要使用的套接字集参数中。如果各主套接字中有一个就绪，服务器就调用 accept；如果各从套接字之一就绪，服务器就调用 read 以便从客户那里获得传入请求，接着它便构成响应，并且调用 write 把响应发回给客户。

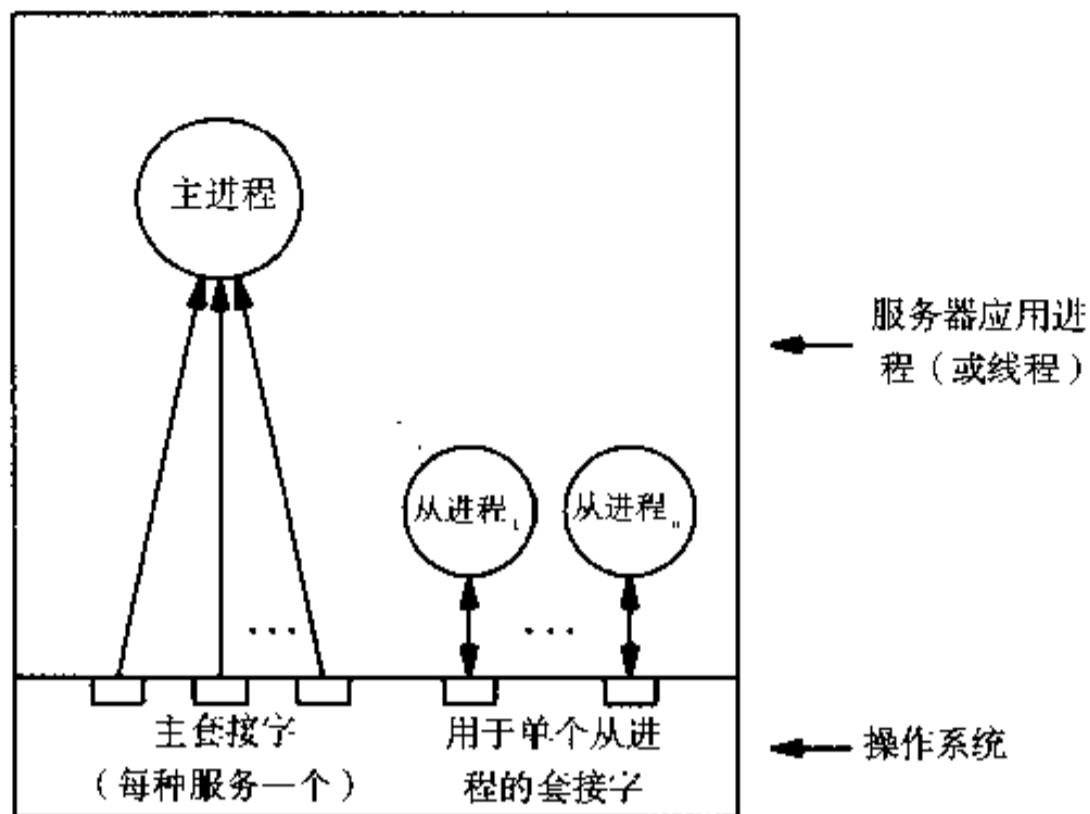


图 15.3 并发的、面向连接的、多服务服务器的进程/线程结构。主进程/线程处理传入连接请求，而从进程/线程处理各个连接

15.7 从多服务服务器调用单独的程序

到目前为止，我们所讨论的多数设计方案的主要缺点是缺乏灵活性：改变任何一个服务的代码都必须要重新编译整个多服务服务器。如果不是考虑让一个服务器处理很多种服务，这个缺点就无关紧要。任何一个小的改变都要求程序员重新编译服务器，并终止服务器的执行，还要用新编译的代码重新启动服务器。

如果一个多服务服务器提供了很多种服务，那么在任何给定的时间里，至少有一个客户要跟服务器通信的机会就更高。因此，终止服务器会在某些客户中引起问题。此外，服务器所提供的服务越多，它需要修改的概率也就越大。

设计者们往往通过使用独立编译的程序，将一个庞大的、完整统一的多服务服务器划分成一个个独立的单元，这些单元处理各个服务。当把这一概念应用到一个并发的、面向连接的设计时，它就是最易理解的。

设想如图 15.3 所示的并发的、面向连接的服务器。主服务器从一组主套接字上等待连接请求。连接请求一旦到达，主进程调用 fork 创建一个从进程以便处理这个连接。主服务器必须将所有服务的代码编译到它的程序中。图 15.4 说明了如何修改设计以便将这种庞大的服务器划分成独立的小片。

如图所示，主服务器使用 fork 创建一个新进程来处理每个连接。然而，与以前的设计不同，从进程以调用 execve 的方式用一个新的程序替代了原来的代码，这个新的程序将处理所有的客户通信。

由于 execve 是从一个文件中获取这个新程序的，上述设计将允许系统管理员在替换文件时，不必再重新编译多服务服务器，然后再终止服务器进程，或再重新启动服务器。从概念上说，使用 execve 就把处理各个服务的程序同设立连接的主服务器代码分开了。

在多服务服务器中，系统调用 `execve` 使得有可能将处理每个服务的代码与管理客户发来的初始请求的代码分割开。

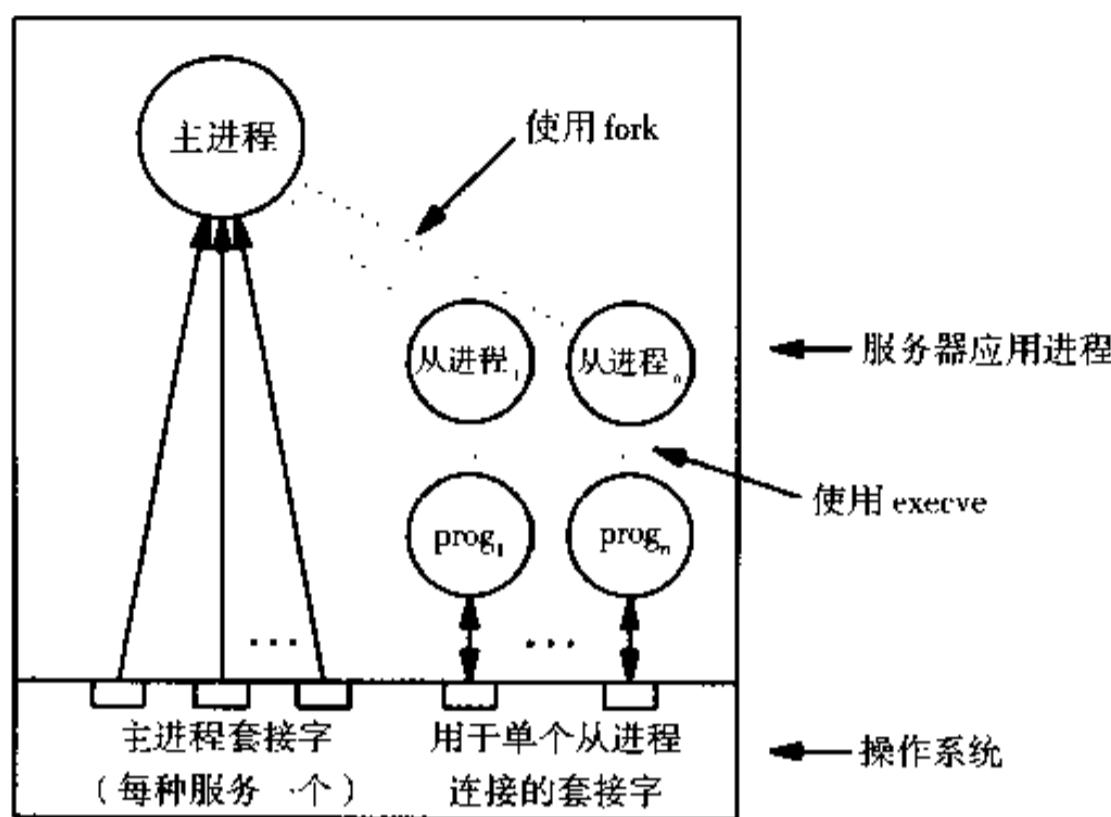


图 15.4 一种面向连接的、多服务服务器的进程结构，它使用 `execve` 执行一个单独的程序来处理每个连接

15.8 多服务、多协议设计

你可能会把一个多服务服务器看作是只单独适应无连接的或面向连接的协议，尽管这看上去很自然，但多协议的设计方案也是可能的。正如第 14 章所述，多协议的设计方案允许单个服务器线程同时管理针对同一个服务的 UDP 套接字和 TCP 套接字。在多服务的情况下，服务器可以为它所提供的一些甚至全部的服务管理 UDP 和 TCP 套接字。

许多网络专家使用术语“超级服务器”(super server)来指一种多服务、多协议的服务器。在原理上，超级服务器的运行很像是一个常规的多服务服务器。在开始时，服务器为它所提供的每个服务打开一个或两个主套接字。对某个给定的服务，它的主套接字对应于无连接的传输(UDP)或者面向连接的传输(TCP)。服务器使用 `select` 等待任一套接字就绪。如果一个 UDP 套接字就绪，服务器调用一个过程，该过程从这个套接字中读取下一个请求(数据报)并计算出响应，然后将应答发送出去。如果是一个 TCP 套接字就绪，服务器也调用一个过程，该过程从这个套接字中接受下一个连接并对其进行处理。服务器可以采用循环的方法直接处理这个连接，也可以创建一个新的进程，使服务器按并发方式来处理这个连接。

15.9 多服务服务器的例子

文件 `superd.c` 中的多服务服务器扩展了第 12 章所述的服务器实现方法。在初始化数据结构并为它所提供的每个服务打开了套接字之后，服务器主程序便进入了无限循环。每一次循环都调用 `select`，以便等待各套接字中的某个准备就绪。当有请求到达时，`select` 就返回。

当 select 返回时，服务器循环扫描每个可能的套接字描述符，使用宏 FD_ISSET 来测试描述符是否就绪。如果发现了就绪的描述符，它就调用一个函数来处理这个请求。为此，服务器首先利用数组 fd2sv 将描述符映射到数组 svent 中的某个条目。

svent 中的每个条目都含有 service 类型的结构，它将套接字描述符映射为服务。service 中的 sv_func 字段含有函数地址，该函数负责处理这个服务。在将描述符映射到 svent 中的某个条目后，程序便调用这个选中的函数。对于 UDP 套接字，服务器直接调用服务句柄（service handler）；而对 TCP 套接字，服务器通过过程 doTCP 间接地调用服务句柄。

TCP 上的服务要求另外的过程，这是因为 TCP 套接字对应于面向连接服务器的主要套接字。当此套接字就绪时，就意味着有一个连接请求已经到达这个套接字。这个服务器需要创建一个新的进程来管理这个连接。因此，过程 doTCP 调用 accept 来接受这个新连接。接着它调用 fork 创建了一个新的从进程。在关闭了那些无关的文件描述符后，doTCP 调用服务句柄函数（sv_func）。当服务函数返回后，从进程便退出。

```
/* superd.c - main */

#define _USE_BSD
#include <sys/types.h>
#include <sys/param.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/errno.h>
#include <sys	signal.h>
#include <sys/wait.h>
#include <netinet/in.h>

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

extern int errno;

#define UDP_SERV 0
#define TCP_SERV 1

#define NOSOCK -1 /* an invalid socket descriptor */

struct service {
    char *sv_name;
    char sv_useTCP;
    int sv_sock;
    int (*sv_func)(int);
};

void TCPechod(int), TCPchargend(int), TCPdaytimed(int), TCPtimed(int);
```

```
int      passiveTCP(const char *service, int qlen);
int      passiveUDP(const char *service);
int      errexit(const char *format, ...);
void     doTCP(struct service *psv);
void     reaper(int sig);

struct service svent[] =
{
    { "echo", TCP_SERV, NOSOCK, TCPechod },
    { "chargen", TCP_SERV, NOSOCK, TCPchargend },
    { "daytime", TCP_SERV, NOSOCK, TCPdaytimed },
    { "time", TCP_SERV, NOSOCK, TCPtimed },
    { 0, 0, 0, 0 },
};

#ifndef MAX
#define MAX(x, y) ((x) > (y) ? (x) : (y))
#endif /* MAX */

#define QLEN          32
#define LINELEN       128

extern unsigned short portbase; /* from passivesock() */ */

/* -----
 * main - Super-server main program
 * -----
 */
int
main(int argc, char *argv[])
{
    struct service     *psv,           /* service table pointer */
                      *fd2sv[NOFILE];   /* map fd to service pointer */
    int      fd, nfds;
    fd_set    afds, rfds;           /* readable file descriptors */

    switch (argc) {
    case 1:
        break;
    case 2:
        portbase = (unsigned short) atoi(argv[1]);
        break;
    default:
        errexit("usage: superd [ portbase]\n");
    }

    nfds = 0;
```

```

FD_ZERO(&afds);
for (psv = &svent[ 0] ; psv->sv_name; ++psv) {
    if (psv->sv_useTCP)
        psv->sv_sock = passiveTCP(psv->sv_name, QLEN);
    else
        psv->sv_sock = passiveUDP(psv->sv_name);
    fd2sv[ psv->sv_sock] = psv;
    nfds = MAX(psv->sv_sock+1, nfds);
    FD_SET(psv->sv_sock, &afds);
}
(void) signal(SIGCHLD, reaper);

while (1) {
    memcpy(&rfds, &afds, sizeof(rfds));
    if (select(nfds, &rfds, (fd_set *)0, (fd_set *)0,
               (struct timeval *)0) < 0) {
        if (errno == EINTR)
            continue;
        erexit("select error: %s\n", strerror(errno));
    }
    for (fd=0; fd<nfds; ++fd)
        if (FD_ISSET(fd, &rfds)) {
            psv = fd2sv[ fd];
            if (psv->sv_useTCP)
                doTCP(psv);
            else
                psv->sv_func(psv->sv_sock);
        }
}
}

/*
 * doTCP - handle a TCP service connection request
 */
void
doTCP(struct service *psv)
{
    struct sockaddr_in fsin;      /* the request from address */
    unsigned int         alen;    /* from-address length */
    int                 fd, ssock;

    alen = sizeof(fsin);
    ssock = accept(psv->sv_sock, (struct sockaddr *)&fsin, &alen);
    if (ssock < 0)
        erexit("accept: %s\n", strerror(errno));
    switch (fork()) {

```

```

    case 0:
        break;
    case -1:
        erexit("fork: %s\n", strerror(errno));
    default:
        (void) close(ssock);
        return;           /* parent */
}
/* child */

for (fd = NOFILE; fd >= 0; --fd)
    if (fd != ssock)
        (void) close(fd);
psv->sv_func(ssock);
exit(0);
}

/*
 * reaper - clean up zombie children
 */
void
reaper(int sig)
{
    int      status;

    while (wait3(&status, WNOHANG, (struct rusage *) 0) >= 0)
        /* empty */;
}

```

我们的超级服务器例子提供了四种服务：ECHO、CHARGEN、DAYTIME 和 TIME，除了 CHARGEN，其他的服务在前面几章中都见到过。程序员使用 CHARGEN 服务测试客户软件。客户一旦同 CHARGEN 服务器构成了一个连接，服务器就生成一个无限的字符序列，并将其发送给客户。文件 sv_funcs.c 包含了处理各个服务的函数的代码。

```

/* sv_funcs.c - TCPEchod, TCPchargend, TCPdaytimed, TCPtimed */

#include <sys/types.h>

#include <unistd.h>
#include <stdio.h>
#include <time.h>
#include <string.h>

#define    BUFSIZE          4096      /* max read buffer size */

extern    int      errno;

```

```
void      TCPechod(int), TCPchargend(int), TCPdaytimed(int), TCPtimed(int);
int       errexit(const char *format, ...);

/*
 * TCPecho - do TCP ECHO on the given socket
 */
void
TCPechod(int fd)
{
    char    buf[BUFFERSIZE];
    int     cc;

    while (cc = read(fd, buf, sizeof buf)) {
        if (cc < 0)
            errexit("echo read: %s\n", strerror(errno));
        if (write(fd, buf, cc) < 0)
            errexit("echo write: %s\n", strerror(errno));
    }
}

#define LINELEN          72

/*
 * TCPchargend - do TCP CHARGEN on the given socket
 */
void
TCPchargend(int fd)
{
    char      c, buf[LINELEN+2]; /* print LINELEN chars + \r\n */
    c = ' ';
    buf[LINELEN] = '\r';
    buf[LINELEN+1] = '\n';
    while (1) {
        int      i;

        for (i=0; i<LINELEN; ++i) {
            buf[i] = c++;
            if (c > '~')
                c = ' ';
        }
        if (write(fd, buf, LINELEN+2) < 0)
            break;
    }
}
```

```
/*
 * TCPdaytimed - do TCP DAYTIME protocol
 */
void
TCPdaytimed(int fd)
{
    char    buf[LINELEN], *ctime();
    time_t  now;

    (void) time(&now);
    sprintf(buf, "%s", ctime(&now));
    (void) write(fd, buf, strlen(buf));
}

#define UNIXEPOCH 2208988800UL           /* UNIX epoch, in UCT secs */

/*
 * TCPTimed - do TCP TIME protocol
 */
void
TCPTimed(int fd)
{
    time_t  now;

    (void) time(&now);
    now = htonl((unsigned long)(now + UNIXEPOCH));
    (void) write(fd, (char *)&now, sizeof(now));
}
```

对各个函数的代码，我们可能觉得大多数都很面熟；它们来自前面各章的各个服务器例子程序。CHARGEN服务的代码可以在过程TCPchargend中找到，它写得简单明了。这个过程含有一个循环，它不停地产生一个填满ASCII字符的缓存，并调用write将这个缓存的内容发送给客户。

15.10 静态的和动态的服务器配置

在实际中，许多系统都提供了一个超级服务器框架，系统管理员可以在它上面增加其他的服。为便于使用，超级服务器通常是可配置的，即不必重新编译源代码就可以改变服务器所能处理的各种服务。可以有两种类型的配置方法：静态的（static）和动态的（dynamic）。静态配置发生在超级服务器开始执行的时候。典型的情况是，配置信息放置在一个服务器启动时可以读取的文件中。配置文件指明服务器可以处理的一组服务以及每个服务所使用的某个可执行的程序。要改变所要处理的服务，系统管理员只需改变配置文件并重新启动服务器。

动态配置发生在超级服务器运行的时候。就像静态配置的服务器那样，动态配置的服务器在开始执行时读取一个配置文件。这个配置文件决定了服务器开始时所能处理的各种服务。与静态配置的服务器不同的是，动态配置的服务器不必重新启动就可以重新定义它所提供的服务。为了改变服务，系统管理员先改变配置文件，然后通知服务器要求重新配置。于是，服务器检查配置文件，按文件所述来改变它的行为。

管理员如何通知服务器需要重新配置呢？答案在操作系统上。在 Linux 系统中，信号（signal）机制被用作进程间通信。管理员向服务器发送一个信号；服务器必须捕获这个信号并把它的到来解释为重配置请求。在没有一种进程间通信机制的操作系统中，动态重配置建立在传统的通信之上——管理员使用 TCP/IP 与服务器通信。为了能够通信，服务器被这样编程，即，它打开了一个用于控制的额外的套接字。当要求重新配置时，管理员就在这个控制连接上与服务器通信。

当管理员迫使服务器动态重新配置时，服务器读取配置文件并更改它所提供的服务。若配置文件中含有一个或多个前一配置中没有的服务，服务器就为新的服务打开套接字以接受服务请求。若有一个或多个服务在配置文件中被删除，服务器就将这些不必再处理的服务所对应的套接字关闭。当然，设计良好的服务器会从容地处理重新配置，即对停止了的服务，尽管服务器不会再接受新的请求，但它并不将那些已在进程中存在的连接异常中止。因此，从客户来的请求要么被拒绝，要么就被完全处理。

使超级服务器成为可动态配置的，将增加相当程度的灵活性。可以不改变超级服务器本身，就能改变处理某个给定服务的可执行程序。更进一步说，不必重新编译服务器的代码，也不必重新启动服务器，就可以改变服务器所提供的那组服务。程序员可以测试新的服务而不必打断正在运行的服务。更重要的是，因为重新配置不要求改变源代码，所以非程序员也能学会配置一个服务器。总之：

可动态重配置的超级服务器是灵活的，这是因为不必重新编译服务器程序或重新启动服务器就可以改变服务器所提供的服务。

15.11 UNIX 超级服务器，inetd

大多数UNIX系统，包括Linux在内，都运行一个能处理许多服务的超级服务器，它被称为inetd，该 UNIX 超级服务器也许是最著名的；许多厂商在其系统中都含有 inetd 的版本。

设计 inetd 的初始动机是这样的：人们期望一种有效的机制可以提供许多服务，但并不过分地使用系统资源。具体说就是，尽管一些 TCP/IP 的服务，如 ECHO 和 CHARGEN，对测试和调试很有用，但在实际工作的系统中却很少使用它们。为每个服务都创建一个服务器要占用系统资源（例如，进程表中的条目和换页空间）。此外，如果各个单独的进程并发执行，它们会竞争使用内存。因此，把各个服务器合并到超级服务器中会减少开销，但并不减少功能。

inetd 是可动态配置的^①，其配置信息保存在一个文本文件中。文件中的每个条目有六个甚至更多的字段，如图 15.5 所示。

当服务器首次启动或者在重新配置之后，它必须为配置文件中的每个新服务创建一个主套接字。为此，服务器将解析配置文件，取出文件中的各个字段。套接字类型（socket type）字段决定主套接字是使用流（stream）还是使用数据报（dgram）类型的套接字。inetd 还必须为每个套接字绑

^① 在收到信号 SIGHUP（即信号的值为 1）后，inetd 就动态地重新配置。

定一个本地协议端口号。为找到一个协议端口号, inetd 取出配置文件中的服务名 (service name) 字段和协议 (protocol) 字段, 用这两个字段向系统的服务数据库查询。该数据库返回这个服务所使用的协议端口号; 如果服务数据库没有包含该服务名字段和协议字段组合构成的条目, inetd 就不能处理这个服务。

字段	含义
服务名 (service name)	所提供的服务的名字 (名字必须出现在系统的服务数据库中)
套接字类型 (socket type)	使用的套接字类型 (必须是一个合法的套接字类型, 如 stream 或 dgram)
协议 (protocol)	服务所使用的协议的名字 (必须是一个合法的协议名, 如 tcp 或 udp)
等待状态 (wait status)	值 wait 指明 inetd 在处理另一请求前应等待服务程序处理完一个请求, 而值 nowait 则允许并发性
用户标识符 (userid)	登录标识符 (login id), 服务程序在其权限下运行。根 (root) 用户拥有绝对的特权
服务器程序 (server program)	要执行的服务程序的名字, 或使用字符串 internal 以指定使用编译到 inetd 中的代码版本
参数 (arguments)	零个或者多个参数, 它们被传递给 inetd 所执行的服务程序

图 15.5 在 inetd 的配置文件中, 一个条目的各个字段。每个条目开始的六个字段是必须要有的, 由一些连续的非空格字符构成; 一行中剩下的字构成了参数

主套接字一旦创建, inetd 便记录下配置文件中剩下的信息, 并等待主套接字上到达的请求。当某个客户联络某个特定的服务时, inetd 利用它记录下来的信息决定如何进行。例如, 等待状态 (waitstatus) 字段决定了 inetd 是否并发地运行服务程序的多个副本。若配置文件指明该字段为 nowait (不等待), inetd 就为每个到达的请求创建一个新的进程, 并允许所有进程并发运行。因为 inetd 要派生一个执行服务程序的子进程, 这样, 每有一个请求到达, 就要创建一个新的进程。inetd 进程一直保持运行, 它不停地在主套接字上等待请求的到来。

从概念上讲, 等待状态字段的值为 wait (等待) 意味着 inetd 将循环地处理服务请求 (即在 inetd 启动另一个进程以便运行某个程序之前, 服务程序应完成对一个请求的处理)。有趣的是, 如果一个请求首次到达, 而它所请求的服务被指明为 wait 状态, inetd 就会派生一个单独的进程来运行服务器程序。为理解其原因, 观察一下 inetd 就会知道, 当等待某个服务时, inetd 不能被阻塞, 这是因为其他服务也许需要继续并发执行。为防止启动多个进程, inetd 只是简单地选择了这样的方式, 即在服务器程序结束前不接受进一步的请求: 在为某个给定服务启动了一个进程后, inetd 使用等待状态字段以决定如何继续下去。若一个服务的等待状态指明为 wait, inetd 从它所监听的套接字集合中把这个服务的主套接字删除。当运行这个服务的进程结束后, inetd 便将这个套接字加入到活动集中, 又开始等待接受对这个服务的请求。

尽管等待状态字段提供了循环执行和并发执行之间的概念上的区别, 但选择 wait 字段还有一个实际的理由。具体说就是, UDP 服务对这样的服务使用 wait, 即这个服务要求客户和服务器交换多个数据报。wait 状态防止 inetd 在服务程序结束前就使用该套接字。这样, 客户可以不受干扰地向服务程序发送数据报。一旦服务程序结束, inetd 就可以重新使用这个套接字了。

无论哪种形式的等待方式, inetd 都使用配置文件中的服务器程序 (server program) 字段来决定执行哪个服务程序。如果该字段指明为内部的 (internal), inetd 就调用一个内部过程来处理这个服

务^①。否则，inetd 将这个字符串看作待执行的服务程序的文件名。在 inetd 调用了一个服务器后，它将参数 (arguments) 字段的内容传递给该服务器程序。

15.12 inetd 服务器的例子

一个简单的例子可以阐明 inetd 的配置以及一些其他的细节。假设程序员希望为 DAYTIME 服务对一个新的服务器进行排错。这个新的服务器能够很容易地加入到 inetd 中。首先，要给这个服务指派一个临时名，并选择一个临时的协议端口号，还要将这个信息加入到系统服务数据库中。例如，若程序员选择了名字 timetest 以及协议端口号 10250，下面的条目将被加到文件 /etc/services 中：

```
timetest      10250/tcp
```

另外，还必须写一个服务器程序。由于 inetd 创建了必要的套接字并接受一个传入连接，服务器程序就不必包含这些细节了；它只需要处理针对一个连接的通信。例如，文件 inetd_daytimed.c 包含了 DAYTIME 服务器的代码，这个服务器可以同 inetd 配合使用。

```
/* inetd_daytimed.c - main */

#include <sys/types.h>

#include <unistd.h>
#include <stdlib.h>
#include <string.h>

/*
 * main - inetd client for DAYTIME service
 */
int
main(int argc, char *argv[])
{
    char    *pts;                      /* pointer to time string */
    time_t   now;                     /* current time */
    char    *ctime();

    (void) time(&now);
    pts = ctime(&now);
    (void) write(0, pts, strlen(pts));
    exit(0);
}
```

正如例子所说明的，这个由 inetd 所调用的、基本的、面向连接的服务器，只需要少量的代码。在 inetd 接受了一个传入连接后，在执行服务器前，它将连接转移到文件描述符 0 上。因此，服务

^① 为提高效率，少数几个简单服务的程序代码被加入到 inetd 中。

器常常使用描述符 0 与客户通信。此外，服务器并没有包含选择使用循环方式还是并发方式的代码，这是因为 inetd 处理了所有这些细节。

服务器代码被编译后，编译的结果是个可执行的程序，它被放置在一个文件当中，inetd 的配置可更改成能引用这个服务器。例如，若上面这个程序的已编译版本放入文件 /pub/inetd_daytimed，下列条目可以加到 /etc/inetd.conf 中：

```
timetest stream tcp nowait root /pub/inetd_daytimed inetd_daytimed
```

这个条目说明了一个叫做 timetest 的服务，它要求一个流 (stream) 套接字和 tcp 协议。服务器并发执行并作为 root 用户运行。最后该服务器的可执行版本可在 /pub/inetd_daytimed 中找到，并且传递给服务器的唯一的命令行参数是它的名字，inetd_daytimed。

15.13 服务器的几种变形清单

本章介绍了服务器实现的各种变化式样。图 15.6 给出了一个简单的小结。

类型	描述
循环式	(不常见) 单服务、多协议 多服务、多协议
单执行线程并发式	(常见) 单服务、多协议 多服务、单协议 多服务、多协议
多进程或线程并发式	典型的多服务、单协议
独立执行程序并发式	超级服务器 往往是多服务、多协议和配置文件

图 15.6 本章所讨论的各种广泛使用的服务器变化式样。在所有多服务设计中，超级服务器（以及面向连接的运输层）是最流行的

15.14 小结

在设计服务器时，程序员可以在无数的实现方案中进行选择。尽管大多数服务器只提供单一的服务，但程序员可以选择一种多服务的实现方法，以减少需要执行的服务器的数量。大多数多服务服务器使用一个传输协议。然而，可以使用多协议以便把无连接的和面向连接的服务结合进一个服务器中。最后，程序员可以用并发进程或线程实现一种并发的、多服务服务器，也可以在单执行线程中使用异步 I/O 以提供表面上的并发性。

本章给出了一个服务器的例子，它说明了一个多服务服务器是如何使用异步 I/O 取代一组主服务器的。该服务器调用操作系统原语 select，等待任一主套接字被激活。

服务器可以静态配置，也可以动态配置。静态配置发生在服务器开始执行时；动态配置发生在服务器运行当中。动态配置允许系统管理员改变所提供的服务集合而不需要重新编译或启动服务器。

Linux 超级服务器 inetd 允许动态重新配置。

深入研究

Linux 程序员手册的第 8 部分描述了 inetd 超级服务器。它还描述了 inetd 配置文件 /etc/inetd.conf 中各条目的语法。

习题

- 15.1 若一个面向连接的、并发的、多服务服务器处理 K 个服务，它将使用的套接字的最大数目是多少？
- 15.2 在你的本地计算机系统中，单个进程可以创建多少个套接字？
- 15.3 考虑一个多服务服务器的单线程实现方案。写出一个算法，该算法说明服务器如何管理连接。
- 15.4 在本章所描述的多服务服务器例子中增加一个 UDP 服务。
- 15.5 阅读 RFC 1288，找出 FINGER 服务。在本章的多服务服务器例子中增加 FINGER 服务。
- 15.6 设计一个超级服务器，允许不必重新编译或重新启动就能增加新服务。
- 15.7 对本章所讨论的每个循环的和并发的多服务服务器的设计，写出一个表达式，计算每个服务器要分配的套接字的最大数。将你的结果表示为所提供的服务数目和要并发处理的请求数目的函数。
- 15.8 一个超级服务器，为处理每个连接要派生一个进程，然后利用 execve 运行一个提供服务的程序，说明这种方法的主要缺点。
- 15.9 查看某台 Linux 机器中的配置文件，查明 inetd 提供哪些服务？
- 15.10 参考描述 inetd 配置文件的手册页，解释参数字段中的 A% 是什么意思。它在什么时候才重要？

第16章 服务器并发性的统一、高效管理

16.1 引言

前几章给出了一些特定的服务器的设计，并展示了每种设计如何使用循环的或并发的处理方式。上一章还研究了如何结合一些设计来创建一个多服务服务器。

本章将以更广的角度研究并发服务器。本章将考察服务器设计所蕴涵的问题以及管理并发性的几种技术，这些技术可以应用于前面的许多设计。它们增加了设计的灵活性，并允许设计人员优化服务器性能。虽然这里提到的两个主要方法似乎相互矛盾，但它们各自在某些情况下可以提高服务器的性能。此外，我们将明白这两种技术出自同一个概念。

16.2 在循环设计和并发设计间选择

至此为止，我们所讨论的服务器设计方案可分为两类：循环地处理请求和并发地处理请求。前几章的讨论表明，设计人员必须在构造服务器前，在两种基本方法之间做出明确的选择。

在循环实现和并发实现之间的选择是很重要的，因为这会影响整个程序的结构，影响觉察到的响应时间以及服务器处理多个请求的能力。如果设计人员在设计过程之初就决策失误，那么改变决策的开销将会很高，很大一部分程序可能需要重写。

程序员如何才能知道能否保证并发性呢？程序员如何知道哪种服务器设计是最佳的？尤其重要的是，程序员如何估计需求或服务时间？由于网络的变化使得这些问题不容易回答：经验表明，网络趋于飞速甚至无法预料地增长。只要用户发现了服务，对服务的访问就会增加。随着已连接用户的数目的扩大，对各个服务器的需求也随之增长。同时，新技术和产品将继续改善通信和处理速度。但是，通信和处理能力正常情况并不以同一速率增长。先是一个增长较快，然后另一个也随之增长。

可能有人会感到疑惑：在不断变化的世界中，设计人员究竟如何才能做出基本的设计抉择。答案通常来源于经验和直觉：设计人员通过观察近期的趋势，以便尽可能做出最佳估计。实际上，设计人员是根据近期历史记录的总结来形成对不久的将来的估计。当然，他们只能提供一个近似值：当技术和用户需求发生变化时，就必须重新评估决策，并且可能要改变设计。其要点如下：

由于用户需求、处理速率和通信能力的迅速变化，在循环的和并发的服务器设计当中进行选择可能很困难。大多数设计人员是根据近期的变化趋势进行推断来做出选择。

16.3 并发等级

让我们考虑并发服务器实现中的一处细节：服务器所允许的并发等级。我们定义服务器的并发

等级 (level of concurrency) 为：在某个给定时刻一个服务器中正在运行着的执行线程总数^①。为处理传入请求，服务器要创建一个线程，在处理完请求后，从线程要退出，因此，服务器的并发等级会随时间发生变化。程序员和系统管理员并不关心跟踪某个给定时刻的并发等级，却关心服务器的整个生命期间，它所展现出的并发等级的最大数值。

到目前为止已提出的各种设计中，只有很少几个需要设计人员为服务器指定最大并发等级。大多数设计允许主服务器创建足够多的从线程，以便处理传入请求。

通常，一个并发的、面向连接的服务器，会为每个来自客户的连接创建一个进程。当然，一个实用的服务器不能处理任意数量的连接。每个TCP的实现可能都限制了活动的连接数，并且每个操作系统都限制可用的线程数（系统必须要限制每个用户可用的线程数，也要限制可用的线程总数）。当服务器达到其中一个限制，系统将拒绝其请求更多的资源。

为了增加灵活性，许多程序员避免为程序的并发等级设置固定的上限。如果服务器代码没有一个预先确定的最大并发等级，那么同一实现可在相当宽广的范围内都适用（即其适用场合可以从不需要很多并发性的环境到有很多并发性需求的环境）。当服务器从一种类型的环境转移到另一种环境下，程序员并不需要改变程序代码或重新编译。但是，在具有重负荷的环境中，不限制并发性的服务器是有危险的。在运行服务器的操作系统中，只要进程没有多得无法招架，并发性还是可以增加的。

16.4 需求驱动的并发

在前面几章所给出的并发服务器设计中，为增加灵活性，大多数利用传入请求来触发并发性的增长。我们将这种方案称为需求驱动 (demand driven)，并且认为并发等级随需求增加 (increase on demand)。

按需求增加并发性的服务器似乎是最佳的，因为它们只在需要时才使用线程或缓存这样的系统资源。因此，需求驱动的服务器只在需要时才使用资源。此外，需求驱动的服务器可处理多个请求而不需等待一个现有请求被处理完毕，这样就能提供明显低的响应时间。

16.5 并发的代价

尽管由需求驱动并发这一方法的一般动机值得推崇，但前面几章提到的实现可能没有产生最佳的效果。要了解其原因，我们必须考虑线程或进程创建和调度的微小细节，以及服务器运行的细节。中心议题是如何度量代价和收益。具体来说就是必须考虑并发的代价和它的收益。

16.6 额外开销和时延

前面几章给出的服务器设计都使用传入请求来度量需求，并且以此来触发并发性的增长。主服务器等待请求，并在请求到达后立刻创建一个新的从线程/进程来处理它。因此，在任一时刻的并发等级反映了服务器已收到的、但还未处理完毕的请求数目。

^① 并发线程可以是同一进程的一部分（即多线程实现），也可以是每个线程只与一个独立的进程相关联（即单线程实现）。我们的度量方法对两种情况都适用。

尽管请求驱动的方案相当简单，但为每个请求创建一个新线程的开销却是昂贵的。不管服务器使用的是无连接的还是面向连接的传输，操作系统都必须通知主服务器有一个报文或是一个连接业已到达。主服务器还必须要请求系统创建一个从线程/进程。

从一个网络接收请求并创建一个从进程/线程可能要花费相当多的时间。除了延迟了对请求的处理，创建一个进程还将消耗系统资源。因此，在传统的单处理器上，当操作系统创建一个新进程或线程并切换环境时，服务器将不会执行。

16.7 小时延可能出麻烦

在创建新进程时，小时延会引起麻烦吗？图 16.1 说明，如果请求所需要的处理不是很多，发生麻烦是可能的。

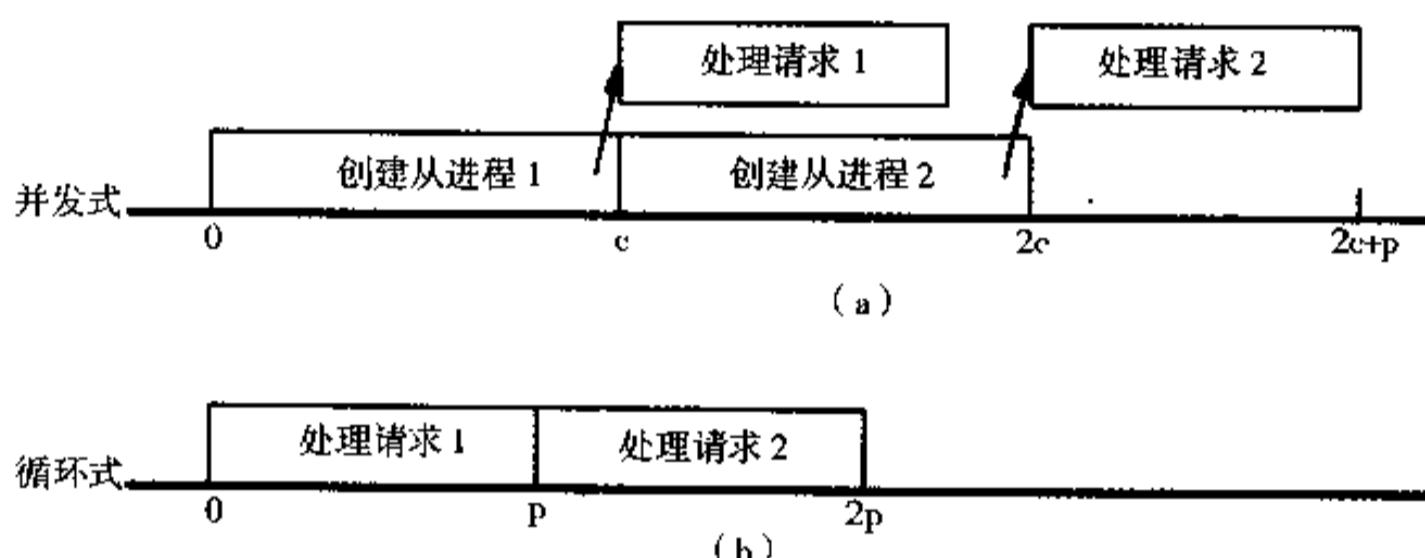


图 16.1 (a) 在并发服务器中处理两个请求所需的时间；(b) 在循环式服务器中处理两个请求所需的时间。处理一个请求所需的时间是 p ， p 小于创建一个进程所需的时间 c ，因而循环版有较低的延迟

在如图所示的例子中，处理一个请求所需的时间小于创建一个新进程所需时间。令 p 表示处理时间，令 c 表示创建一个进程所需的时间。假设两个请求在时间 0 突然到达。并发版本在 $c + p$ 时间单元后完成第一个请求的处理，并在 $2c + p$ 时间单元后完成第二个请求的处理。因此，它处理每个请求平均需要 $3c/2 + p$ 时间单元。一个循环式服务器在时间 p 完成对第一个请求的处理，在时间 $2p$ 完成对第二个请求的处理，处理每个请求平均只需 $3p/2$ 时间单元。因此，循环式服务器设计比并发设计显示出较低的时延。

只考虑几个请求时，少许额外的时延可能不太重要。但是，如果考虑服务器在重负荷下的连续运行，时延就可能很重要了。如果许多请求几乎在同一时刻到达，它们就必须等待服务器创建对其进行处理的进程。如果额外的一些请求到来的速率比服务器的处理速率还快，时延将会累积起来。

从短期看，服务器的小时延只影响可观察到的响应时间，而不会影响整体吞吐量。如果多个请求几乎在同一时刻到达，操作系统的协议软件将把它们放入队列，服务器再从队列取出请求进行处理。例如，如果服务器使用一个面向连接的传输，TCP 将使连接请求排队。如果服务器使用一个无连接的传输，UDP 将使到达的数据报排队。

从长远看，额外的时延使请求丢失。要追其究竟，可以想像一个服务器，它创建一个从进程要花 c 时间单元，但处理一个请求只需 p 时间单元 ($p < c$)。一个并发实现的服务器每个时间单元平均处理 $1/c$ 个请求，而一个循环版本每个时间单元可处理 $1/p$ 个请求。

当请求到达的速率超过 $1/c$ 但小于 $1/p$ 时，会出现问题。一个循环实现能处理这种负荷，但一个并发实现却花了太多时间来创建进程。在并发版本中，协议软件中的队列最终会排满，并将开始拒绝再来的请求。

实际上，几乎没有服务器的运行会接近最大吞吐量。而且，当创建一个从进程的开销超过处理请求的开销时，几乎没有设计人员会使用并发服务器。因此，请求的延迟或丢失在许多应用中都不会发生。但是，要把服务器设计成在重负荷下能提供最佳响应，就必须要考虑按需并发的替代方案。

16.8 从线程/进程预分配

一种直截了当的技术可用于控制延迟、限制最大并发等级，并且当进程创建时间较长时，也能使并发服务器维持高吞吐量。此技术是预分配并发进程或线程（preallocating concurrent processes or threads），这样避免了创建进程/线程所要付出的代价。线程一旦创建，就会持续地运行（即不退出）。

为使用预分配技术，设计人员这样编写主服务器，即使它在开始执行时就创建N个从线程/进程。每个从线程/进程都使用操作系统中所提供的设施以等待请求到达。当请求到达后，一个等待的从线程/进程就开始执行并处理该请求。完成请求的处理后，从线程/进程不退出，而是返回到等待请求的那段代码处。

预分配的主要优点是操作系统的额外开销较低。由于服务器不需要在请求到达时创建从线程/进程，它可更快地处理请求。当请求的处理涉及的 I/O 多于计算时，此技术就显得尤其重要。预分配允许服务器系统在等待与前一个请求相关的 I/O 活动时，切换到另一个从线程/进程，并开始处理下一个请求。概括起来就是：

当使用预分配时，服务器在启动时就创建若干个并发的从线程/进程。预分配避免了在每次请求到达时创建进程的开销，因而降低了服务器的时延，同时允许在处理一个请求时，与另一个请求相关联的 I/O 活动也在重叠进行。

尽管可降低进程创建的开销，但持续性确有一个缺点：程序员必须对资源的使用相当小心。为理解原因，我们来考虑一个在每次到达一个请求时，就分配少量内存的持续运行的从线程/进程。完成该请求的处理后，从线程/进程并不释放内存。尽管问题可能在很长时间后才发生，从线程/进程最终会耗尽地址空间。

16.8.1 Linux 中的预分配

一些支持线程的操作系统，如 Linux，能够使主线程和预分配的从线程之间的交互更容易，因为主线程和从线程可以共享内存。即使当从线程是作为不能共享内存的单独进程来执行时，进程预分配仍是可能的。协调工作依靠于共享的套接字：

当一个 UNIX 进程调用 fork 时，新创建的子进程接收了所有打开的描述符的一个副本，包括各套接字所对应的描述符。

为利用套接字共享，主进程在预分配从进程之前将打开必要的套接字。具体说就是，在主服务器进程启动时，为熟知端口打开一个套接字，请求将到达该端口。主进程然后调用 fork 创建所需的多个进程。由于每个进程从其父进程继承套接字描述符的副本，所有的从进程都可访问对应熟知端口的套接字。下一节将讨论面向连接和无连接的服务器中进程预分配的细节。

16.8.2 面向连接服务器中的预分配

如果并发服务器使用 TCP 通信，那么并发等级与活跃的连接数有关。每个传入连接请求必须用一个单独的进程处理。幸运的是，Linux 为那些试图在同一个套接字接受一个连接的多个进程提供了互斥。每个从进程调用 accept，accept 将阻塞而等待接收一个到熟知端口的传入连接请求。当连接请求到达时，系统只会使一个从进程不再阻塞。在从进程中，当 accept 调用返回时，就提供用于该传入连接的新文件描述符。从进程处理连接，关闭新套接字，然后调用 accept 处理下一个请求。图 16.2 展示了进程的结构。

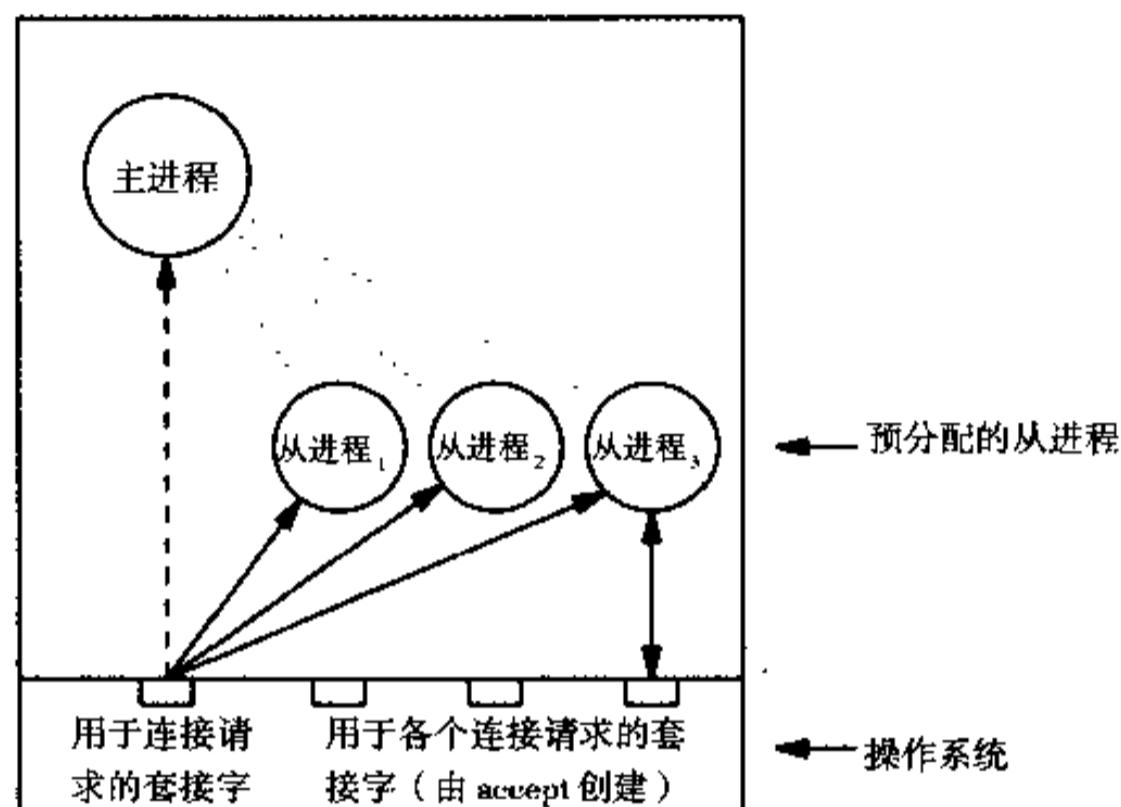


图 16.2 在并发的、面向连接的服务器中的进程结构，该服务器预分配一些从进程。本例显示了三个预分配的从进程，其中一个正在处理连接。主进程为熟知端口打开套接字，但不使用该套接字

如图所示，所有的从进程继承了对熟知端口套接字的访问。当各个从进程调用 accept 返回时，它接受新套接字以用于这个连接。虽然主进程创建了对应熟知端口的套接字，但它并不使用该套接字进行其他操作。图中的虚线表明主进程使用该套接字的方式与从进程的不同。

虽然图 16.2 表明主进程与从进程同时运行，但主进程和从进程之间的区别多少有些模糊。实际上，主进程在预分配从进程后就没有任务了。因此，主进程在从进程启动后就可以简单地退出^①。聪明的程序员甚至可让主进程创建除最后一个从进程外的所有其他从进程。这样主进程就成为最后一个从进程，从而节省了创建额外进程的开销。在 Linux 中，完成此工作的代码很简单。

16.8.3 互斥、文件锁定和 accept 并发调用

在 Linux 中，从进程的预分配是很容易的，因为操作系统能够区分调用 accept（针对某一给定套接字的 accept）的并发进程。更重要的是，Linux 能够有效地处理并发调用。这样，每个预分配的从进程就继承了主进程的套接字描述符，且每个从进程都调用 accept。系统会让所有的从进程保持阻塞状态，一直到有连接到达，然后，只让其中的一个从进程脱离阻塞。

^① 实际上，如果进程进入了一个重复调用系统函数 wait3 的循环，主进程可消除这些死进程（zombie processes）。如果主进程退出了，从进程将由 init 进程掌管。

遗憾的是，并不是所有 UNIX 版本都能提供像 Linux 这样的语义。有些 UNIX 版本不允许并发调用 accept——如果有进程调用了某个套接字上的 accept，针对该套接字的后继 accept 调用将返回错误消息；还有些 UNIX 版本虽然允许并发调用，但处理起来效率不高。具体说来是这样的，当有新连接到达时，有些操作系统会使调用 accept 的所有进程都脱离阻塞状态，第一个执行的线程会获得这个新连接，而后继的其他进程在执行后发现没有要接纳的连接，从而回到阻塞状态。因此，如果有 K 个进程被阻塞了，就会有 K-1 个进程不必要的消耗资源——每个进程都要进行环境切换、都要使用 CPU、都会发现连接已被接纳，然后返回阻塞状态。

预分配方法可以用干这种不能有效处理并发 accept 的系统吗？答案是可以的。采取的办法不是让各个进程并发调用 accept，而是必须使用一个共享的互斥量 mutex 或文件锁定（file locking，即 flock），以便保证任何时候只有一个从线程能够调用 accept。例如，如果使用互斥量，每个从线程在调用 accept 之前，必须调用 pthread_mutex_lock，在调用完 accept 之后，必须再调用 pthread_mutex_unlock。在任何时候，除了某一个从线程之外，所有其他从线程在调用 pthread_mutex_lock 时都会阻塞，没有被阻塞的那个线程得以继续执行，从而调用 accept。一旦从 accept 调用返回，它会调用 pthread_mutex_unlock，从而允许另一个（只一个）从线程得以继续执行。这样，任何时候都只有一个线程调用 accept，系统资源得以有效利用。

16.8.4 无连接服务器中的预分配

如果一个并发服务器使用无连接传输，则并发等级取决于到达的请求数。每个传入请求以一个单独的 UDP 数据报形式到达，并且每个请求必须送给一个单独的执行线程。在并发的、无连接的设计方案中，在请求到达时，通常让主服务器创建单独从线程/进程。

Linux 允许无连接服务器使用面向连接的服务器所采用的预分配策略。图 16.3 显示了进程的结构。

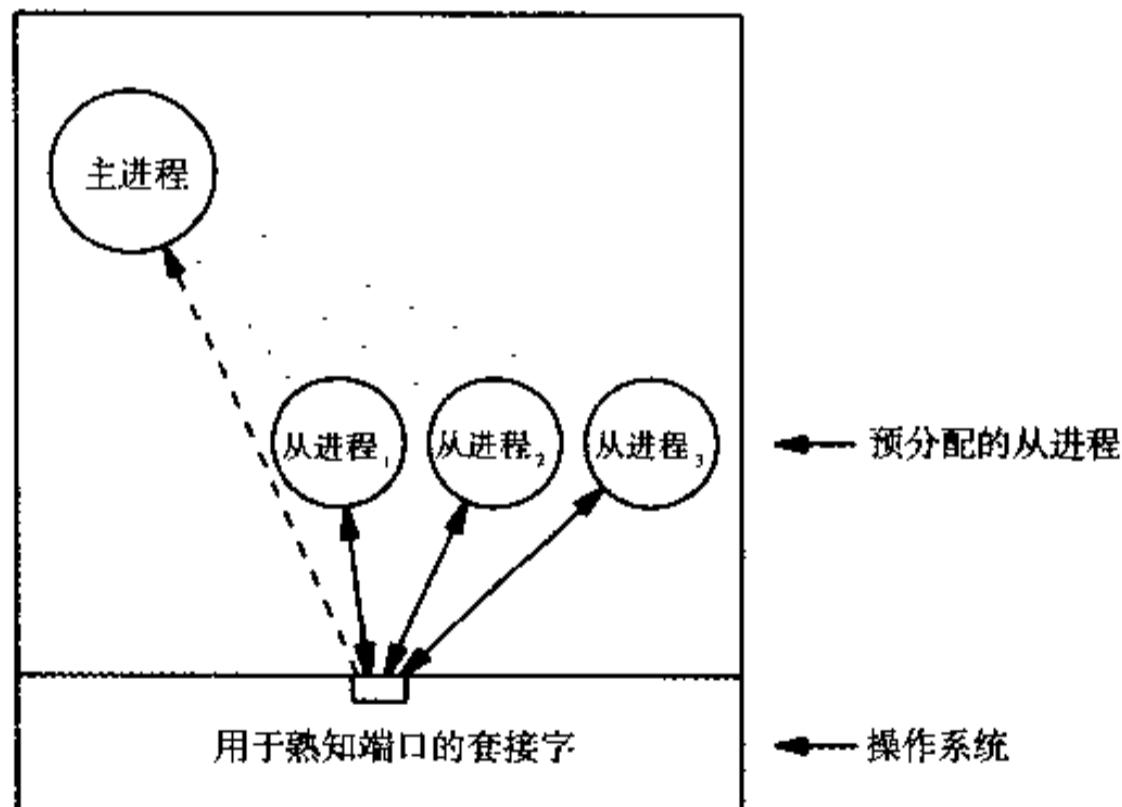


图 16.3 并发的、无连接的服务器的进程结构，该服务器预分配一些从线程/进程。本图显示了三个从线程/进程，他们都从熟知端口对应的套接字上读取数据。只有一个从线程/进程接收传入请求

如图所示，每个从进程继承了熟知端口对应的套接字。由于通信是无连接的，从进程可使用同

一个套接字发送响应以及接收传入请求。一个从进程调用 `recvfrom` 获得发送方的地址和从该发送方发来的数据报；它调用 `sendto` 传输应答。

与使用预分配的面向连接服务器一样，无连接服务器的主进程在为熟知端口打开套接字并预分配从进程后，就几乎不再做工作了。因此，它可以退出，或将其自己作为最后一个从进程，从而省去了创建最后一个进程的开销。

16.8.5 预分配、突发通信量和 NFS

经验表明，由于大多数的 UDP 实现并不为到达的数据报提供很长的队列，因此突然到达的多个传入请求很容易使队列溢出。UDP 只丢弃队列满了以后到达的数据报。因此突发的通信量可能引起丢失。

由于 UDP 软件通常驻留于操作系统中，因此溢出问题特别困难。这样，应用程序员不可能轻易地修改它。但是，应用程序员可预分配从线程/进程。预分配往往足以避免丢失。

许多网络文件系统 NFS（Network File System）的实现使用预分配来避免数据报的丢失。如果检查一下运行 NFS 的系统，通常会发现有一组预分配的服务器进程都从同一个 UDP 套接字上读取数据。实际上，预分配意味着 NFS 的可用实现和不可用实现是有区别的。

16.8.6 多处理器上的预分配

多处理器上的预分配有一个特殊目的。它允许设计人员使服务器的并发等级与硬件性能相关联。如果机器有 K 个处理器，设计人员可预分配 K 个从线程/进程。由于多处理器操作系统给每个线程/进程一个单独的处理器，预分配保证了并发等级与硬件是匹配的。当请求到达时，操作系统将它传给其中一个预分配的从线程/进程，并为该进程指定一个处理器。因为线程/进程已被预分配，几乎不需要时间来启动它。因此，系统将会迅速地分发请求。如果突然到达多个请求，每个处理器将处理一个请求，从而获得尽可能最高的速率。

16.9 延迟的从线程/进程分配

虽然预分配可提高效率，但它不能解决所有问题。在某些情况下，使用一种相反的方法却能提高效率：延迟从线程/进程的分配。

要理解延迟如何能起作用，回想一下从线程/进程创建需要时间和资源。只有当创建额外的从线程/进程能提高系统吞吐量或降低时延时，这样做才是合理的。创建一个从线程/进程不仅花时间，也为管理从线程/进程的操作系统部件带来了额外开销。另外，预分配多个试图接收传入请求从线程/进程可能会给网络代码增添额外开销。

我们讲过，如果创建一个从线程/进程的开销少于一个请求的开销，那么并发会降低时延。如果处理请求的开销较小，那么循环方案工作得最佳。但是，所需的时间可能与请求有关（如搜索一个数据库所需的时间与查询有关），因此，程序员不是总能知道怎样来比较这些开销。

另外，程序员可能不知道是否能迅速找到差错。要理解其原因，我们考虑一下大多数服务软件是如何工作的。当请求到达时，服务器软件就检查此报文，以验证报文各字段的值是否合法，并且验证客户是否被授权发出请求。验证可能花几微妙，或者它可能引发进一步的网络通信，从而多花几个数量级的执行时间。一方面，如果服务器检测到报文中有差错，它将迅速拒绝请求，使得处理

此报文所需的全部时间足可忽略。另一方面，如果服务器收到一个有效的请求，它就可能消耗很可观的处理时间。在处理时间短的情况下，并发处理不会得到认可；一个循环服务器将表现出更低的时延和更高的吞吐量。

若设计人员不知道采用并发处理是否合理，他们如何优化时延和吞吐量呢？答案是采用一种延迟的从线程 / 进程分配（delayed slave allocation）技术。其思想是直截了当的：该方法不是选用一个循环的或并发的设计，而是允许服务器测量处理的开销，然后动态地选用循环处理或并发处理。选择是动态的，因为在处理不同的请求时选择可以不同。

为实现动态的、延迟的分配，服务器经常通过测量逝去的时间来估计处理开销。主服务器接收请求，设置计时器，接着便开始循环地处理请求。如果在计时器到期前，服务器已完成处理请求，服务器将取消计时器。如果在服务器完成处理请求前，计时器就到期了，服务器将创建一个从进程，并让从进程处理请求。总之：

当使用延迟的从线程 / 进程分配时，服务器将开始循环地处理每个请求。仅当处理要花大块时间时，服务器才创建一个并发的从线程 / 进程来处理该请求。这种时延允许主服务器在创建一个进程或切换环境前，先检查有无差错并处理一些短的请求。

在 Linux 中，延迟分配从线程 / 进程不难。由于 Linux 含有一个告警（alarm）机制，主线程可设置一个计时器，并设法在计时器到期时执行一个过程。由于 Linux 的 fork 函数允许一个新创建的进程从父进程处继承打开的套接字以及执行程序和数据的副本，主进程可创建一个从进程，该从进程恰好从主进程超时所执行代码处继续进行处理。

16.10 两种技术统一的基础

预分配从进程和延迟分配从进程似乎没有共同之处。实际上，他们似乎是完全相反的。但是，由于他们基于同一概念上的原理：将请求到达至从线程 / 进程创建之间的间隔扩大，就可能提高某些并发服务器的性能，因而这两种技术又有很多共同的地方。预分配提高了在请求到达前服务器的并发等级；延迟的分配提高了在请求到达后服务器的并发等级。此观点可归纳如下：

预分配和延迟分配基于同一原理：通过把服务器的并发等级从当前活跃的请求数目中分离出来，设计人员可获得灵活性并提高服务器效率。

16.11 技术的结合

延迟分配和预分配的技术可结合使用。服务器一开始可以没有预分配的从线程 / 进程，并可使用延迟分配。它等待请求到达，并且若处理花了很长时间（即计时器到期），就只创建一个从线程 / 进程。但是，一旦创建了从线程 / 进程，该从线程 / 进程不必立刻退出；从线程 / 进程可认为自己是永久分配的，并继续运行。处理完一个请求后，从线程 / 进程可等待下一个请求到达。

结合的系统的最大问题是需要对并发性进行控制。何时应创建新增的从线程 / 进程是很容易知道的，但从线程 / 进程何时应退出而不是继续运行就很难知道了。一种可能的方案是设法让主线程 / 进程在创建一个从线程 / 进程时，指明其最大增长值 M。从线程 / 进程可创建至多 M 个从线程 / 进程，这些从线程 / 进程还可创建零个或多个线程 / 进程。因此，系统开始时只有一个主线程，但最

终会到达固定的并发等级最大值。另一种控制并发的技术是设法让一段时期内不活跃的从线程/进程退出。从线程/进程在等待下一个请求前启动计时器。如果在请求到达前计时器到期，从线程/进程就退出。

如果各个线程处于一个进程中，这些从线程可使用共享内存等设施来协调其活动。它们可存储一个共享的整数用来记录任一时刻的并发等级，并且可使用该整数的值来决定在处理请求后是退出还是继续^①。在某些系统中，若允许应用找出在某个套接字上排队的请求数，从线程/进程也可使用该队列长度来帮助确定并发等级。

16.12 小结

有两种主要的技术能使设计人员提高并发服务器的性能：预分配和延迟分配从线程/进程。

预分配是设法在需要从线程/进程之前就先创建好了一些，从而优化了延迟。主服务器为所要使用的熟知端口打开一个套接字，然后预分配所有的从进程或线程。因为从线程/进程继承了对该套接字的访问，它们都可以等待某个请求的到来。系统将每个收到的请求交给其中的某个从线程/进程。预分配对于并发的无连接服务器很重要，因为处理一个请求的所需时间通常较短，这使得创建线程或进程的开销相对很大。预分配方案也使得在多处理器上的并发、无连接设计方案效率提高。

延迟分配使用一种缓慢的方法处理分配。主服务器一开始循环地处理每个请求，但设置一个计时器。如果主线程/进程完成处理前计时器到期，它就创建一个并发的从线程/进程来处理请求。在各个请求的处理时间不同的情况下，或者当服务器必须检查一个请求是否正确时（即验证客户是否被授权），延迟分配都可以很好地工作。对于短的请求或含有差错的请求，延迟分配消除了额外开销。

虽然这两种优化技术看起来是对立的，但它们都基于同一种基本原理：这两种优化技术减缓了服务器并发等级与挂起请求数之间的严格一致性。这样做可提高服务器的性能。

深入研究

第24和25章描述了网络文件系统（NFS）。NFS的许多实现使用预分配，这有助于避免请求的丢失。

习题

- 16.1 使用预分配来修改前几章的一个服务器例子。其性能会如何变化？
- 16.2 使用延迟分配来修改前几章的一个服务器例子。其性能会如何变化？
- 16.3 测试在多处理器上使用预分配技术的无连接服务器。要设法让客户发送多个突发的请求。使用的并发等级是如何与处理器数相关的？如果这两个数不同，请解释原因。
- 16.4 试编写结合使用延迟分配和预分配的服务器算法。你如何限制最大并发等级？
- 16.5 在前一题中，如果操作系统提供了消息传递机制，你如何使用它来控制并发等级？

^① 当然，一定要使用互斥量防止两个从线程同时访问这个整数值。

- 16.6 把本章所讨论的技术应用于采用单执行线程、提供表面并发性的服务器，可从中获得什么优点？
- 16.7 设计人员如何将本章讨论的技术用于多服务服务器？
- 16.8 构造并测量以下三种采用预分配从进程方法的服务器：第一种允许从进程并发调用 accept；第二种利用共享互斥量保证任何时候只有一个从进程执行 accept；第三种使用 flock。哪种方法最快？当从进程数增加时，结论会改变吗？

第17章 客户进程中的并发

17.1 引言

前几章说明了服务器如何并发处理请求。本章将研究客户软件中的并发问题，讨论客户如何从并发中受益，以及并发的客户如何运行。最后，本章还将展示一个说明并发运行的客户例子。

17.2 并发的优点

服务器使用并发有两个主要的原因：

- 并发可改善观察到的响应时间，从而改善所有客户的总吞吐量。
- 并发可排除潜在的死锁。

另外，并发实现使得设计人员易于创建多协议的或多服务的服务器。最后，使用多进程实现并发非常灵活，因为这样就可在多种硬件平台上很好地运行。把并发实现移植到只有一个处理器的计算机上时，它们可正常运行。当把并发实现移植到具有多个处理器的计算机时，工作的效率会更高，因为它们充分利用了额外的处理能力而不需改变代码。

由于客户通常在一个时刻只进行一种活动，因而它似乎不能从并发中受益。客户一旦向服务器发送了一个请求，在收到响应之前并不能进行其他活动。此外，客户的效率和死锁问题不如服务器那样严重，因为如果一个客户延缓或停止执行，只有一个客户受到影响，而其他客户将继续运行。

尽管表面上如此，客户中的并发确实有优点。第一，并发实现更容易编程，因为功能已被划分为概念上能分开的一些部分。第二，并发实现更易于维护和扩展，因为这使得代码模块化了。第三，并发客户可在同一时刻联系几个服务器，或者比较响应时间，或者合并服务器返回的结果。第四，并发允许用户改变参数、查询客户状态或动态地控制处理。本章将着重讨论客户同时与多个服务器进行交互的概念。

在客户中使用并发的最主要的优点在于异步性。异步性允许客户同时处理多个请求，且不严格规定其执行顺序。

17.3 运用控制的动机

如下情况可能需要使用异步，即需要将控制功能与正常处理分开时。例如，使用一个客户查询大型的人口统计数据。假设用户可能生成了如下式样的查询：

Find all people who live on Elm Street.

如果数据库只含有一个城市的信息，响应可能包括不到 100 个名字。但是，如果数据库包含有美国所有人的信息，响应可能含有数十万个名字。此外，如果数据库系统由分布在很大地理范围内的许多服务器组成，查找可能要花许多分钟。

数据库的例子说明了关于许多客户 - 服务器交互的重要概念：调用客户的用户对于要经过多长时间才能收到响应或响应究竟有多少，可能知之甚少，或者根本不知道。

大多数客户软件仅仅等待响应到达。当然，如果服务器发生故障或发生死锁，客户将阻塞在那里，试图等待一个永不会到达的响应。遗憾的是，由于网络时延很大或服务器超负荷，用户就不可能知道是真发生了死锁，或者只是处理很慢。此外，用户不知道客户是否已从服务器收到了一些报文。

如果用户不耐烦了，或者判断出一个特定的响应需要太多时间，也只有一种选择：放弃客户程序，以后再重试。在这种情况下，并发是很有帮助的，因为一个适当设计的并发客户可使得用户在客户等待响应时，继续与该客户交互。用户可发现是否已收到了一些数据，并选择是发送一个不同的请求，还是从容地终止通信。

我们可以将上面描述的人口统计数据库作为例子来考虑。并发实现可在进行数据库查询的同时，从用户的键盘或鼠标读取和处理命令。因此，用户可打开菜单，选择一个像 status 这样的命令，以判断客户是否已成功打开了到某个服务器的连接，并且是否已发送了请求。用户可选择 abort 停止通信，或者选择 newserver 命令客户终止现有通信，并尝试与另一个服务器通信。

将客户的控制与正常处理分开允许用户与客户交互，即使客户使用文件作为它的正常输入时也能如此。这样，甚至在用户启动了一个客户，让它处理一个很大的输入文件后，他或她仍可与正运行的客户程序交互，以便弄清处理进展得如何。与此类似，并发客户在保持与用户单独交互时，还可将一些响应放入输出文件中。

17.4 与多个服务器的并发联系

并发能使得单个客户同时联系几个服务器，而且只要从任何服务器收到响应就向用户报告。例如，TIME 服务的并发客户可发送请求给多个服务器，并可接受第一个到达的响应或取几个响应的平均值。

考虑使用 ECHO 服务的客户，它负责测量到指定目的地的吞吐量。假定客户建立了到某个 ECHO 服务器的 TCP 连接，发送大量数据，读取返回的响应，计算该任务所需的所有时间，并每秒报告一次表示成字节数的吞吐量。用户可调用这种客户判断当前网络的吞吐量。

现在考虑并发如何能提高这种利用 ECHO 来测量吞吐量的客户的性能。并发客户不是在同一时间只测量一个连接，而是可在同一时刻访问多个目的地。它可并发地给任何一个目的地发请求，并读取返回的响应。由于并发地完成所有测量，它比非并发的客户运行得更快。此外，由于它同时进行所有的测量，CPU 和本地网络上的负载对其影响是相同的。

17.5 实现并发客户

与并发服务器类似，大多数客户实现遵从两个基本方法之一：

- 客户分为两个或多个执行线程，每个处理一个功能。

或

- 客户只含一个线程，它使用 select 异步地处理多个输入和输出事件。

对于像 Linux 这样的系统，因为允许一个进程中的多个线程共享内存，这样就使多线程实现能很好地运行。图 17.1 说明了在这种系统中，如何使用多线程的方法支持面向连接的应用协议。

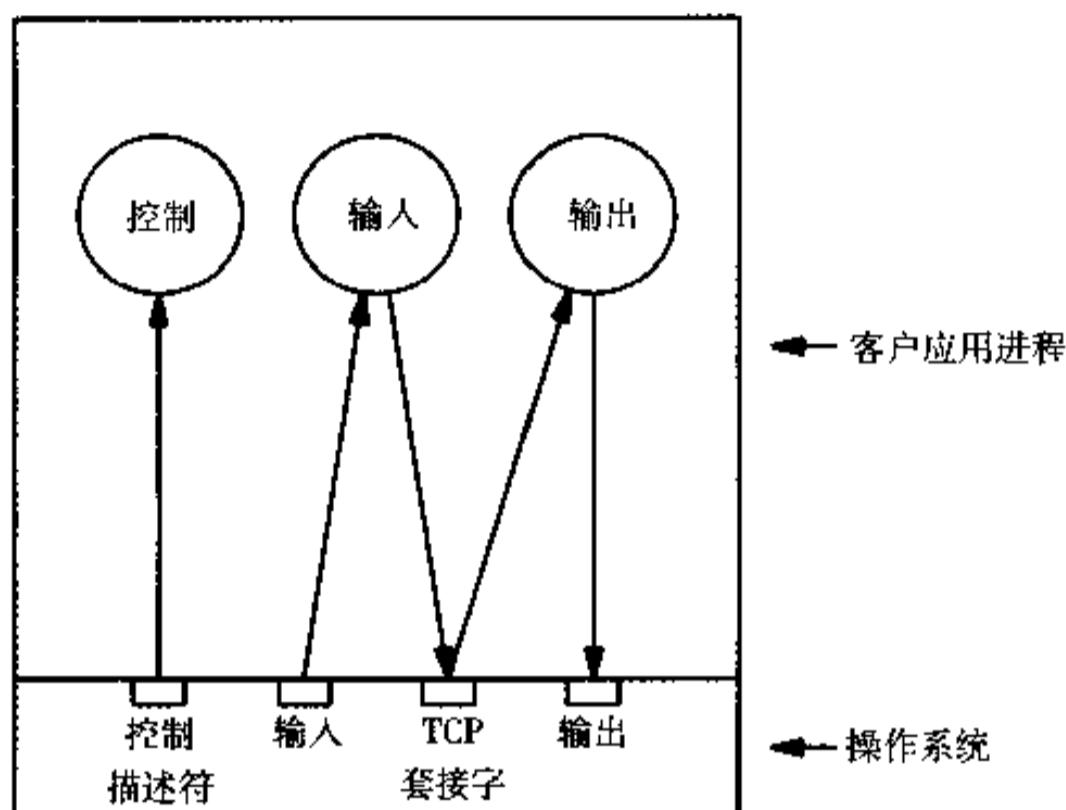


图 17.1 面向连接的客户的一种可能的进程结构，它使用多个线程完成并发处理。
一个线程处理输入并向服务器发送请求，而另一个则取出响应并处理输出

如图 17.1 所示，多线程允许客户把输入和输出处理分开。该图表示线程如何与若干文件描述符及一个套接字描述符交互。一个输入线程（input thread）从标准输入读数据，形成请求，并通过 TCP 连接发送给服务器，而一个独立的输出线程（output thread）从服务器接收响应，并写入到标准输出。同时，第三个控制线程（control thread）从控制处理的用户那里接受命令。

17.6 单线程实现

如果系统不支持共享内存，或者在不希望创建线程的场合，客户可以采用单线程算法实现并发，此算法类似算法 8.5^① 和第 13 章到 15 章中的例子。图 17.2 显示了进程结构。

单线程的客户像单线程的服务器一样，使用异步 I/O。客户为到多个服务器的连接创建套接字描述符。它还可以有一个或多个用于获得键盘或鼠标输入的描述符。客户程序的主体含有一个循环，该循环使用 select 等待其中任何一个描述符准备就绪。如果输入描述符已准备就绪，客户就读取输入，并且可以将输入存储起来以后再用，也可以立刻开始处理输入。如果 TCP 连接输出就绪，客户就在此 TCP 连接上准备和发送请求。如果 TCP 连接输入就绪，客户就读取这个服务器发出的响应并加以处理。

当然，单线程的并发客户与单线程的服务器有许多共同的优点和缺点。客户读取输入或读取来自服务器的响应，是按产生这些数据的速率进行的。即使服务器延迟了一小段时间，本地的处理仍

^① 见 83 页关于算法 8.5 的描述。

继续进行下去。因此，即使服务器出故障不能响应，客户仍会继续读取和执行控制命令。

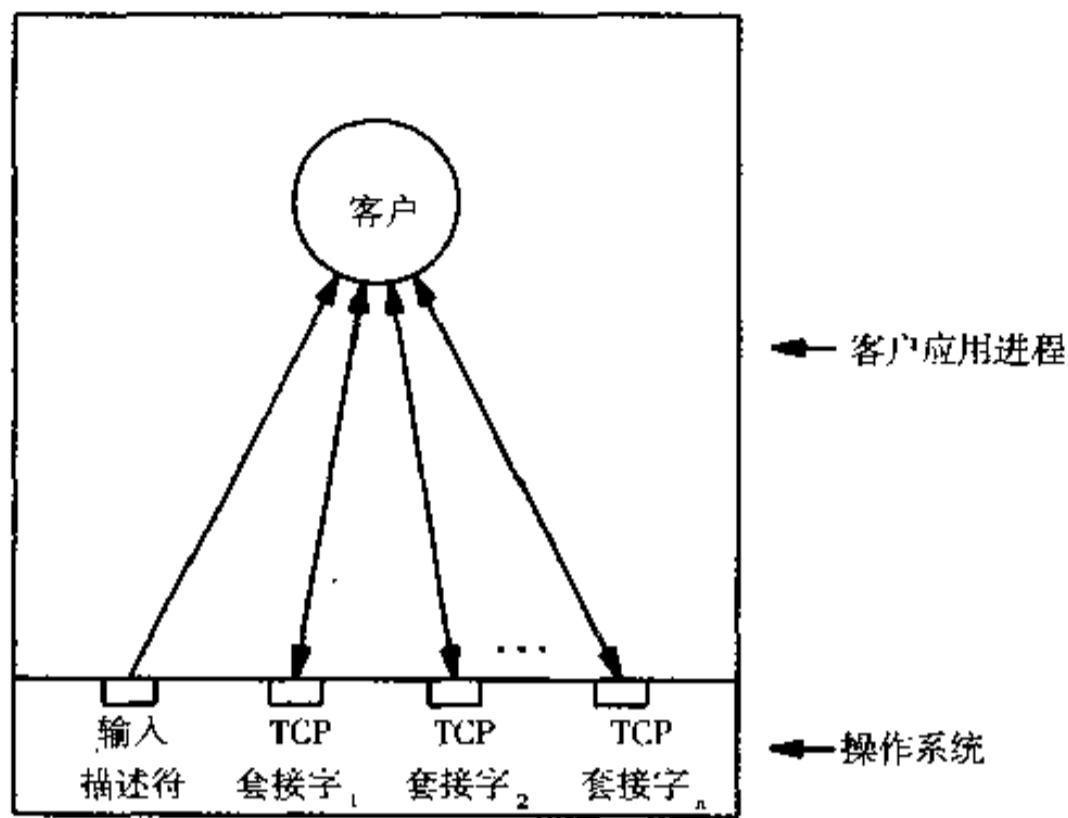


图 17.2 单线程的、面向连接的客户进程结构，能提供表面上的并发。客户使用 select 并发处理多个连接

如果单线程的客户调用会阻塞的系统功能，它就可能转为死锁状态。因此，程序员必须注意确保客户不会无限期地阻塞——在那里等待不会发生的事件。当然，程序员可能选择这样的方法：他们忽略某些情况，并允许用户检测已发生的死锁问题。重要的是，程序员应了解许多微小细节，并为每种情况做出有意识的决策。

17.7 使用 ECHO 的并发客户例子

一个用单线程完成并发的客户例子可阐明上述概念。在如下文件TCPtecho.c中展示了并发客户的例子，它使用第 7 章描述的 ECHO 服务来测量一组机器的网络吞吐量。

```
/* TCPtecho.c - main, TCPtecho, reader, writer, mstime */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/socket.h>

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

extern int errno;
```

```
int      TCPtecho(fd_set *pafds, int nfds, int ccount, int hcount);
int      reader(int fd, fd_set *pfds);
int      writer(int fd, fd_set *pfds);
int      errexit(const char *format, ...);
int      connectTCP(const char *host, const char *service);
long     mstime(unsigned long *);

#define  BUFSIZE          4096           /* write buffer size */
#define  CCOUNT           64*1024        /* default character count */
#define  USAGE            "usage: TCPtecho [ -c count ] host1 host2...\n"

char    *hname[NOFILE];                  /* fd to host name mapping */
int     rc[NOFILE], wc[NOFILE];         /* read/write character counts */
char    buf[BUFSIZE];                  /* read/write data buffer */

/* -----
 * main - concurrent TCP client for ECHO service timing
 * -----
 */
int
main(int argc, char *argv[])
{
    int      ccount = CCOUNT;
    int      i, hcount, maxfd, fd;
    int      one = 1;
    fd_set  afds;

    hcount = 0;
    maxfd = -1;
    for (i=1; i<argc; ++i) {
        if (strcmp(argv[i], "-c") == 0) {
            if (++i < argc && (ccount = atoi(argv[i])))
                continue;
            errexit(USAGE);
        }
        /* else, a host */

        fd = connectTCP(argv[i], "echo");
        if (ioctl(fd, FIONBIO, (char *)&one))
            errexit("can't mark socket nonblocking: %s\n",
                    strerror(errno));
        if (fd > maxfd)
```

```
        maxfd = fd;
        hname[fd] = argv[i];
        ++hcount;
        FD_SET(fd, &afds);
    }
    TCPtecho(&afds, maxfd+1, ccount, hcount);
    exit(0);
}

/*
 * TCPtecho - time TCP ECHO requests to multiple servers
 */
int
TCPtecho(fd_set *pafds, int nfds, int ccount, int hcount)
{
    fd_set      r fds, wfds;      /* read/write fd sets          */
    fd_set      rcfds, wcfds;    /* read/write fd sets (copy)   */
    int         fd, i;

    for (i=0; i<BUFSIZE; ++i) /* echo data                  */
        buf[i] = 'D';
    memcpy(&rcfds, pafds, sizeof(rcfds));
    memcpy(&wcfds, pafds, sizeof(wcfds));
    for (fd=0; fd<nfds; ++fd)
        rc[fd] = wc[fd] = ccount;

    (void) mstime((unsigned long *)0); /* set the epoch      */

    while (hcount) {
        memcpy(&rfds, &rcfds, sizeof(rfd));
        memcpy(&wfds, &wcfds, sizeof(wfd));

        if (select(nfds, &rfds, &wfds, (fd_set *)0,
                   (struct timeval *)0) < 0)
            errexit("select failed: %s\n", strerror(errno));
        for (fd=0; fd<nfds; ++fd) {
            if (FD_ISSET(fd, &rfds))
                if (reader(fd, &rcfds) == 0)
                    hcount--;
            if (FD_ISSET(fd, &wfds))
                writer(fd, &wcfds);
        }
    }
}
```

```
/*
 * reader - handle ECHO reads
 */
int
reader(int fd, fd_set *pfdset)
{
    unsigned long    now;
    int             cc;

    cc = read(fd, buf, sizeof(buf));
    if (cc < 0)
        errexit("read: %s\n", strerror(errno));
    if (cc == 0)
        errexit("read: premature end of file\n");
    rc[fd] -= cc;
    if (rc[fd])
        return 1;
    (void) mstime(&now);
    printf("%s: %d ms\n", hname[fd], now);
    (void) close(fd);
    FD_CLR(fd, pfdset);
    return 0;
}

/*
 * writer - handle ECHO writes
 */
int
writer(int fd, fd_set *pfdset)
{
    int             cc;

    cc = write(fd, buf, MIN((int)sizeof(buf), wc[fd]));
    if (cc < 0)
        errexit("read: %s\n", strerror(errno));
    wc[fd] -= cc;
    if (wc[fd] == 0) {
        (void) shutdown(fd, 1);
        FD_CLR(fd, pfdset);
    }
}
```

```

/*
 * mstime - report the number of milliseconds elapsed
 */
long
mstime(unsigned long *pms)
{
    static struct timeval epoch;
    struct timeval now;

    if (gettimeofday(&now, (struct timezone *)0))
        errexit("gettimeofday: %s\n", strerror(errno));
    if (!pms) {
        epoch = now;
        return 0;
    }
    *pms = (now.tv_sec - epoch.tv_sec) * 1000;
    *pms += (now.tv_usec - epoch.tv_usec + 500) / 1000;
    return *pms;
}

```

17.8 并发客户的执行

TCPecho 接受多台机器名作为参数。对每台机器，它打开一个到该机器上 ECHO 服务器的 TCP 连接，通过该连接发送 ccount 个字符（字节），读取从每个服务器上收到的返回字节，并打印完成任务所需的全部时间。因此，TCPecho 可用于测量到一组机器的当前吞吐量。

TCPecho 开始将字符计数变量初始化为默认值 CCOUNT。然后它分析其参数，查看用户是否键入了 -c 选项。如果键入了，TCPecho 就将该指明的计数变量转换为一个整数，并将它存入 ccount 变量从而取代默认值。

TCPecho 假定除 -c 标志外的所有参数指明了一个机器名。对于每个这样的参数，它调用 connectTCP 形成到使用该名字的机器上的 ECHO 服务器的一个 TCP 连接。TCPecho 在 hname 数组中记录机器名，并调用 FD_SET 宏设置文件描述符掩码中该套接字对应的比特。它还在 maxfd 中记录最大描述符数（调用 select 所需要的）。

一旦为参数指明的每台机器建立了 TCP 连接，主程序就调用过程 TCPecho 处理数据的传输和接收。TCPecho 并发处理所有的连接。它用要发送的数据（字母 D）填充 buf 缓存，然后调用 select 等待任何一个 TCP 连接输入或输出就绪。当 select 调用返回时，TCPecho 遍历所有描述符以查看哪个就绪了。

当一个连接输出就绪时，TCPecho 就调用过程 writer，writer 发送缓存中的数据，只要 TCP 在单个 write 调用中能接受，它便尽量将缓存中的数据发送出去。如果 writer 发现整个缓存都已发送完毕，它就调用 shutdown 关闭这个用于输出的描述符，并从 select 所用的一组输出中删除该描述符。

当一个描述符输入就绪，TCPecho 就调用过程 reader，reader 尽量从连接上接受数据，TCP 能交付多少，它就读多少，并把这些数据放在缓存中。过程 reader 将读取的数据放入缓存，并减少剩

余字符的计数。如果计数减到零（即服务器已收到的数据与被发出的一样多），过程 reader 就计算从开始传送数据以来所逝去的时间，打印一个报文，并关闭连接。它还从 select 所用的一组输入中删除该描述符。因此，每当一个连接完成后，报告数据回显所需时间的消息就出现在输出上。

在一个连接上完成单个输入或输出操作后，过程 reader 和 writer 都将返回，并接着在 TCPtecho 中再次调用 select，继续进行循环。如果 reader 检测到文件结束的条件就返回 0，并关闭连接，反之则返回 1。TCPtecho 使用 reader 的返回码来确定它是否应减少活动连接的数值。当连接数减到零时，TCPtecho 中的循环将终止，TCPtecho 将返回主程序，于是，客户便退出。

图 17.3 显示了 TCPtecho 的三次单独执行产生的输出样本。第一个调用表明 TCPtecho 只需要 311 毫秒就能把数据发送给本地机上的 ECHO 服务器。命令行只有一个参数 localhost。由于第二次调用有三个参数 (ector、arthur 和 merlin)，它使得 TCPtecho 并发地与三台机器交互。第三次调用测量到机器 sage 所需的时间，但命令行指明 TCPtecho 只发送 1000 个字符，而不是默认的 64K 字符。

```
% TCPtecho localhost
localhost: 311 ms

% TCPtecho ector arthur merlin
arthur: 601 ms
merlin: 4921 ms
ector: 11791 ms

% TCPtecho -C 1000 sage
sage: 80 ms
```

图 17.3 在珀杜 (Purdue) 大学的机器上，三次单独执行 TCPtecho 的输出。如果某台机器距离客户更远，或其处理器速率更慢，则该机器将需要更多时间

17.9 例子代码中的并发性

TCPtecho 的一个并发实现从两方面改善了程序。首先，并发实现测量了同一时间间隔内所有连接的吞吐量，因而它可获得对每个连接所需时间更精确的测量。因此，拥塞会同等影响所有的连接。其次，并发实现使得 TCPtecho 对用户更具吸引力。要理解其原因，可以再次观察第二个输出样本中报告的时间。机器 arthur 的报文在半秒多钟后出现，机器 merlin 的报文大约过 5 秒钟后出现，而最后一台机器 ector 的报文大约过 12 秒钟后出现。如果用户必须等待所有测试顺序运行，则整个执行将需要大约 18 秒钟。当测试的机器远在 Internet 上时，各个时间可能需要更长，这就使得并发版本显得快得多。在很多情况下，使用顺序的客户实现测量 N 台机器，可能要比一个并发版本多花将近 N 倍的时间。

17.10 小结

并发执行是一个强有力的工具，可用于客户和服务器中。并发客户实现可提供更快的响应时间，并可避免死锁问题。另外，并发可帮助程序员将控制和状态处理从正常的输入和输出处理中分离出来。

我们研究了面向连接的客户的一个例子，它测量访问一台或多台机器上的ECHO服务器所需的时间。由于客户并发地执行，它通过在同一时间间隔内进行所有的测量工作，避免了网络拥塞引起的不同吞吐量。由于并发实现将多个测量时间相重叠，而不是让用户等待所有测量顺序执行，因此，并发实现对用户也很有吸引力。

习题

- 17.1 注意客户例子是顺序检查就绪的文件描述符。如果许多描述符同时就绪，客户将首先处理最低数字的描述符，然后依次处理其他描述符。处理完所有就绪的描述符后，它将再次调用 `select`，在另外有描述符就绪前将一直等待。考虑在处理一个就绪的描述符与调用 `select` 之间所逝去的时间。对数字大的描述符操作后逝去的时间，比对数字小的描述符操作后逝去时间少。这种区别会导致资源缺乏（starvation）吗？试解释。
- 17.2 修改客户例子，避免上一题中讨论的不公平。
- 17.3 对本章讨论的各个循环和并发客户设计方案，试写出给出使用套接字的最大数量的表达式。

第 18 章 运输层和应用层的隧道技术

18.1 引言

前面几章描述了客户和服务器软件的设计，这些软件用于下面这种情况，即通过 TCP/IP 互联网将所有进行通信的机器互联在一起的时候。我们给出的许多设计都假定客户和服务器运行在功能相当强的计算机上，而且这些机器的操作系统支持并发进程且完全支持 TCP/IP 协议。

本章将开始探讨一些技术，系统管理人员和编程人员用它来开发其他形式的网络拓扑结构。特别是，允许计算机使用高层协议服务来传送 IP 通信量（traffic），以及通过 IP 传送其他协议系统所用的通信量的设计，本章将讨论这些技术。

18.2 多协议环境

在理想环境中，使用 TCP/IP 的编程人员只需要为直接连到 TCP/IP 互联网的计算机构造客户和服务器软件，而这些计算机都提供了对 TCP/IP 协议的完全支持。然而事实上不是所有的机器都提供对 TCP/IP 的完全支持，而且并非所有的机构都仅仅使用 TCP/IP 来互联其计算机。例如，一个机构可能有一些没有足够能力运行服务器软件的嵌入式系统，或者可能有几组机器所连接的网络是使用 DECNET、SNA 或 X.25 协议。实际上，大多数机构中的联网会随时间而扩大，因为机构会增加新的网络来互联已有的计算机组。通常，网络管理人员选择一种硬件技术，并为每个计算机组独立选择一种协议族。他们在选择时考虑的因素包括费用、距离、所期望的速率和厂商可用性等。在能够使用 TCP/IP 协议之前，安装网络的机构可能已经选择了某一种厂商所特有的协议族。网络的这种发展结果使得大多数机构拥有好几个机组，而每个组都使用自己的协议族。上述的要点是：

由于网络是经过多年缓慢发展而来的，而且各厂商纷纷推出自己特有的网络系统，加上 TCP/IP 并非总是可用的，因此一些大的机构通常拥有多组计算机，各使用不同的协议系统进行通信。此外，为最大限度地减少开销，各机构在采用新的技术之前，通常还要继续使用旧的网络系统。例如，图 18.1 展现的是一个使用三个网络的机构，这三个网络位于两个网点，每个网点都有自己的以太网。一个使用 X.25 协议的广域网把两个网点上的主机连接起来^①。如图所示，每个网络都连接了若干台机器。

拥有多个网络系统的最大缺点在于需要付出重复的劳动，并且互操作性也受限制。连接到一个 X.25 广域网的主机必须使用 X.25 协议，而不是 TCP/IP 协议。因此，如果在图 18.1 所示的 X.25 网络所连接的主机上运行客户和服务器，它们必须用 X.25 虚电路进行通信。而运行在以太网上的客户和服务器则使用 TCP 虚电路。

^① X.25 采用面向连接的模式，与帧中继（Frame Relay）或 ATM 所采用的模式相似。

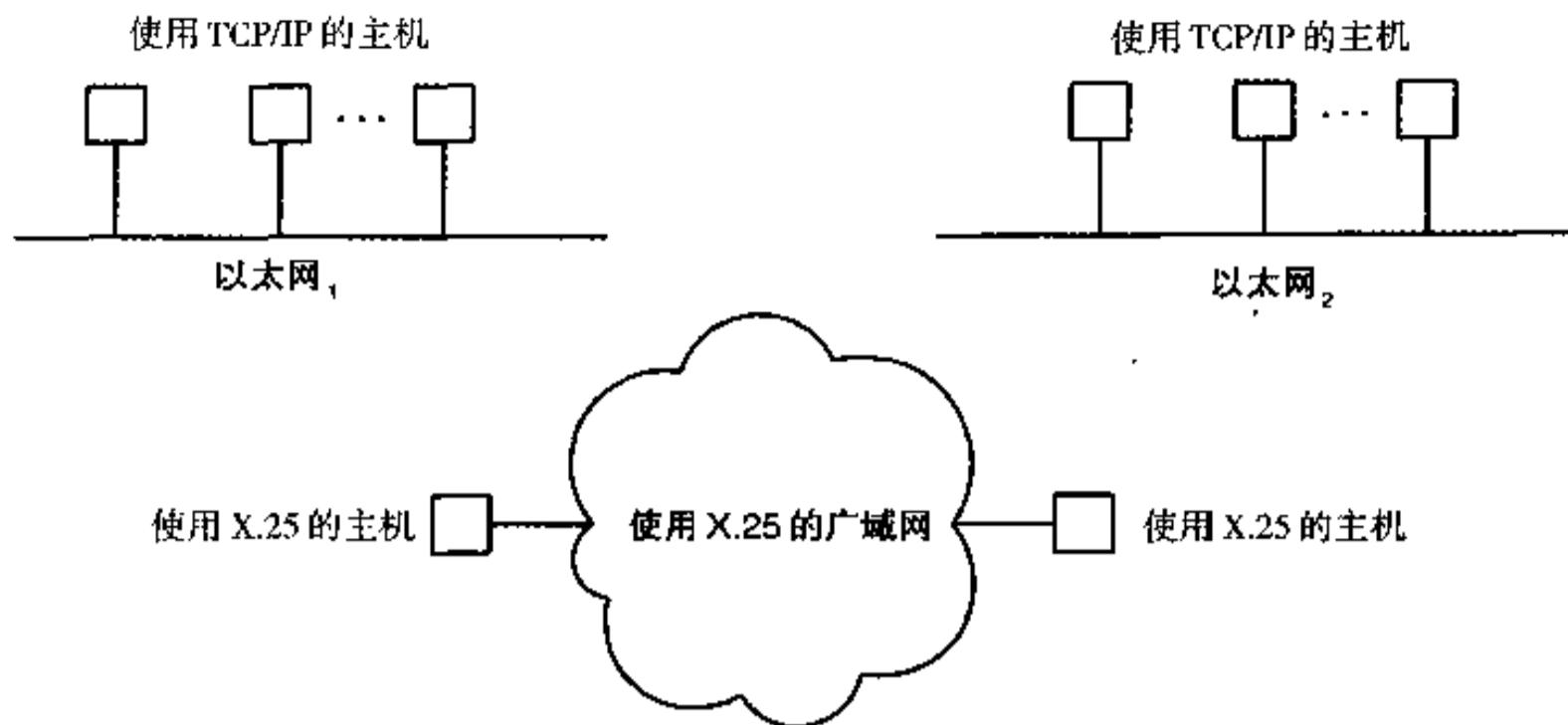


图 18.1 有三个网络的机构示例。连到局域网的所有机器都使用 TCP/IP，而所有连接到广域网的机器都使用 X.25 协议

18.3 混合网络技术

通常，TCP/IP 互联网是由多个主机的集合组成的，而这些主机位于通过路由器（IP 网关）互联的一些物理网络上。互联网上所有主机和路由器都必须使用 TCP/IP 协议。同样，运行其他协议的网络由物理链路和只使用该协议的计算机组成。但是，由于运输层服务能够像分组交换硬件那样很容易地实现点到点的交付分组，因此，对于分组交换系统中的单个物理链路，替换任何一种运输层交换服务应该是可能的。

许多互联网的构造使用运输层交换服务代替物理网络。例如，重新考虑图 18.1 所示的网络。假定该机构决定互连两个以太网，以便形成一个 TCP/IP 互联网，从而使得连接到两个以太网上的主机能够进行通信。最明显的方法是在它们之间安装两个路由器。但是，如果两个以太网相隔太远，连接两个网的专门租用线路的费用可能无法接受。另外，由于该机构已经有 X.25 网络连接两个网点，再增加这种租用线路费用就很不合理了。

图 18.2 说明了图 18.1 所示机构如何利用现有 X.25 网络的连通性来提供两个网点间的 TCP/IP 互联网连接。

该机构在每个网点安装了一个新路由器。每个新路由器连接 X.25 网络和该网点的本地以太网。当路由器启动时，使用 X.25 建立一条常规的运输层虚电路，这条虚电路通过 X.25 广域网连到另一个路由器。每个路由器各自设计其路由表，以便它能够让非本地的通信量通过 X.25 电路。路由器使用 X.25 协议向另一端的路由器发送 IP 数据报。从路由器的角度看，X.25 只提供能发送数据报的链路。从 X.25 网络的角度看，两个路由器上的 IP 软件与其他主机上的应用软件没什么区别。X.25 服务并不知道在虚电路上传送的数据是由 IP 数据报组成的。

有了这两个路由器，任何一台主机上的用户就可以调用标准 TCP/IP 客户软件与任何其他主机的一个服务器联系。客户 - 服务器的交互可以在单个以太网内进行，也可以穿越 X.25 网络。当数据报从一个网点的以太网到达另一个网点的以太网时，用户或应用程序都不需要知道数据报是否穿越了 X.25 网络。两个以太网只是 TCP/IP 互联网的组成部分。此外，广域网上使用 X.25 协议的主机不需要改变。由于这些主机使用的虚电路与路由器之间的新连接无关，因而他们能够继续通信，而

不会受 TCP/IP 通信量的干扰。

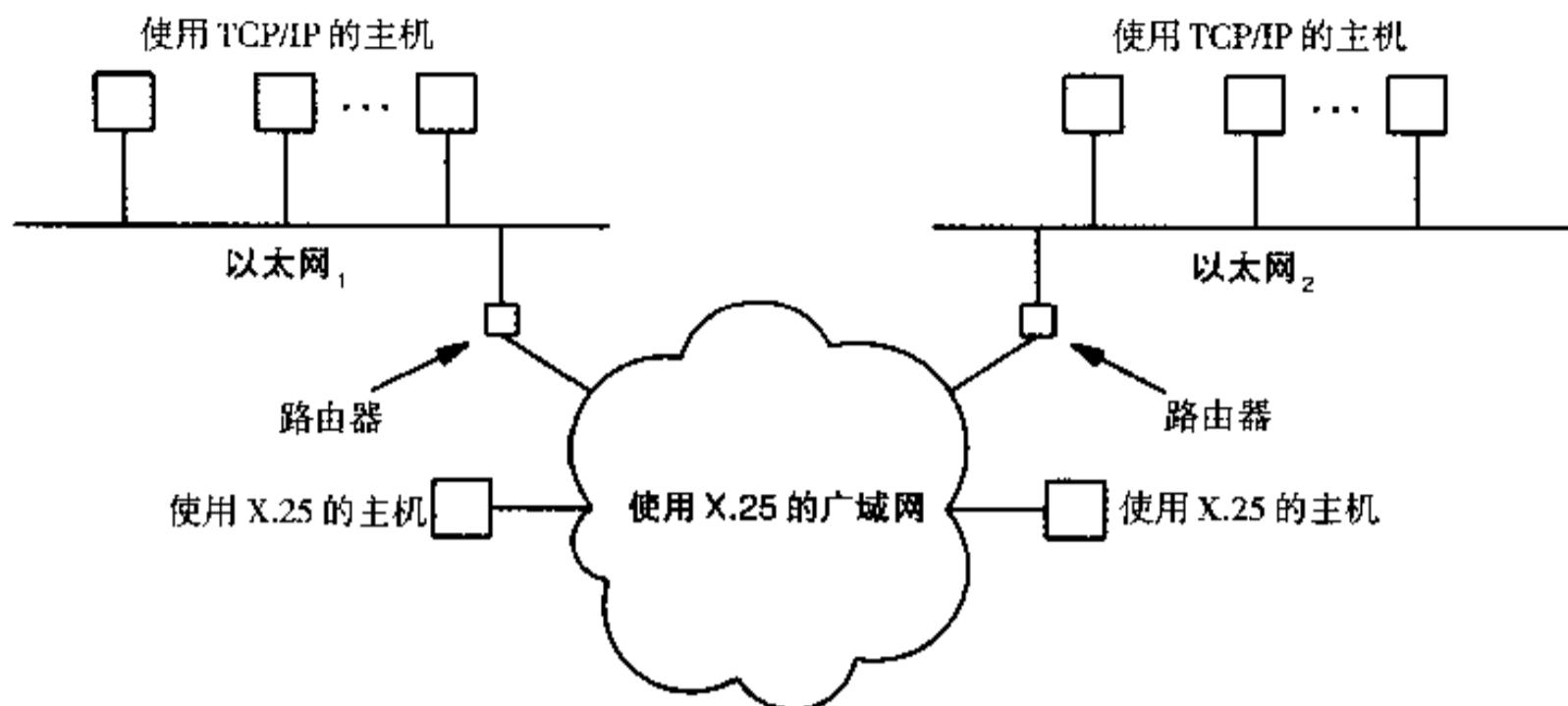


图 18.2 IP 路由器使用 X.25 运输层服务取代物理网络

18.4 动态电路分配

在图 18.2 所示的拓扑结构中，TCP/IP 互连网通信量只需要一条穿过 X.25 网络的 X.25 虚电路，这是因为该机构只有两个网点。如果该机构增加了其他网点，那就要扩大网络拓扑，在新的网点放置路由器，并创建通过 X.25 网的其他虚电路，以便使每个新的路由器能与现有网点中的那些路由器互连。

上述静态电路方案不能扩充到任意数量的网点，这是因为大多数 X.25 网络限制了单台计算机能同时分配的虚电路数。典型情况是，硬件限制一台机器最多可以有 16 或 32 条虚电路。一个具有 N 个网点的机构要互连其所有网点，需要有 $(N*(N-1))/2$ 条虚电路。因此，若机构有 6 个网点，则一个路由器需要有 15 条连接；当机构有 9 个网点时，连接数将超过 32。

当然，也可以增加额外的路由器，以便使每个路由器不需要到所有目的地都各有一条虚电路。然而为了限制开销，大多数使用 X.25 传输数据报的网点采用了另一种方法——按需分配电路，并在不使用时关闭电路。当数据到达一个路由器时，路由器查看该数据报的目的地址以决定它的路由。路由选择查找将产生下一跳地址 (next-hop address)，即应把该数据报发送到的下一个路由器的地址。如果下一跳地址指明的是一个远端网点，路由器将查询其活跃的 (active) X.25 虚电路表。如果到下一跳有一条虚电路，路由器就将通过这条电路转发数据报。如果没有这样的电路，路由器就动态地打开一条新的电路到该目的地。

当路由器需要打开一条新电路时，如果没有可用的电路，它就必须关闭一条现有的电路才行。随之而来的问题是：应关闭哪一条电路呢？通常，路由器采用的策略与按需分页系统使用的相同：它关闭最近最少使用的 (LRU, Least Recently Used) 电路。路由器通过新的电路发送数据报后，将使电路保持打开的状态。通常发出的数据报会使收方回答，因此保持电路打开有助于使时延和开销最小。

通过动态地打开和关闭电路，路由器可限制同时打开所需连接的数目而不会丧失与所有网点通信的能力。路由器只需为正与之通信的每一个网点打开一条电路。

18.5 封装和隧道技术

封装 (encapsulation) 这个术语描述了这样的过程：将一个 IP 数据报放进一个网络分组或帧以便在下层的网络上发送它。封装涉及网络接口如何使用分组交换硬件。例如，在以太网上使用 IP 进行通信的两个主机，要将每个数据报封装进单个分组中以便传输。TCP/IP 所用的封装标准规定：IP 数据报占用以太网分组的数据部分，而且以太网分组类型必须被设置为指定的值，它指明数据是 IP。

隧道技术 (tunneling) 是指使用高层传输网络服务 (high-level transport network service) 运送来自于另一个服务的分组或报文。图 18.2 中的例子说明路由器用隧道通过 X.25 服务发送数据报到其他路由器。隧道技术和封装的主要区别在于 IP 是在硬件分组中传输数据报，还是使用高层传输服务交付这些数据报^①。

当 IP 直接使用硬件发送数据报时，它将每个数据报封装到一个分组中。当 IP 使用一个高层传输服务点对点发送数据报时，便创建一个隧道。

18.6 通过 IP 互联网的隧道技术

在最初定义了 TCP/IP 后，研究人员做了一些实验，研究如何能使 IP 软件用隧道技术通过现有网络（如 X.25），从而交付数据报。动机应该是很明显的：许多机构已有网络在那里了。令人惊讶的是，趋势却与之相反。现在，大多数隧道技术的出现，是由于许多厂商使用 IP 协议交付非 TCP/IP 协议的其他分组。

要理解有关隧道技术的变化，需要理解网络的变化过程。随着 TCP/IP 日益普及，TCP/IP 互联网成为许多集团全球分组交付的机制。实际上，IP 现在为大多数组织的计算机提供了最广泛的连通性。

为了解 IP 的可用性如何影响其他协议，假设一个机构中的两台机器需要使用厂商特有的协议通信。管理人员可将所在机构的 IP 互联网看成一个大网络，并且允许两台机器上的协议软件将报文放进 IP 数据报中进行交换，从而不需要在两台机器之间增加一个额外的物理网络连接。现在有一些商用软件能够用 IP 运送各种高层协议的通信量。

18.7 客户和服务器之间的应用级隧道技术

虽然隧道技术的一般概念是指一个传输级的协议族被另一个使用，但编程人员可将该思想扩展到客户 - 服务器的交互中。编程人员可使用应用级隧道技术 (application-level tunneling)，以此提供客户和服务器间的通信通路。

为了理解应用层隧道技术是如何工作的，让我们考虑两台连接到一个 X.25 网的计算机。假设某位程序员希望在一台机器上运行 UDP 客户应用进程，而在另一台上运行 UDP 服务器应用进程。通常，程序员不能修改操作系统软件，因为他们只允许构建应用程序。因此，如果两台计算机的操作

^① 在像 ATM 和帧中继这种提供端到端通信的技术中，隧道和封装的区别不是很清楚。我们认为在这种网络中，IP 是被封装了，因为它们依赖于面向连接的硬件更甚于依赖传输协议。

作系统不支持 TCP/IP 协议和传输级隧道技术，程序员可能发现使用 UDP 不方便甚至不可能，或者，让 IP 数据报用隧道技术穿越 X.25 网也不方便或不可能。

在这种情况下，应用层隧道可使客户和服务器通过 X.25 网络进行通信。为此，程序员必须构造一个仿真套接字接口的过程库。仿真库（simulation library）必须允许应用程序创建主动的（active）或被动的（passive）UDP 套接字，并能发送或接收 UDP 数据报。套接字仿真库中的过程将对标准套接字例程的调用（如 socket、send 和 recv）加以转换，形成分配和管理本地数据结构的操作，以及通过 X.25 网络传送报文的操作。当客户调用 socket 创建一个套接字，套接字库例程就创建一条到服务器的 X.25 连接。当客户或服务器调用 send 传输一个报文，send 库例程就通过 X.25 连接传送 UDP 数据报。

一旦创建了套接字仿真库，编程人员就能编译任何 UDP 客户或服务器程序，使用仿真库链接编译后的程序，最后运行最终的应用程序。图 18.3 显示了最终的软件结构。

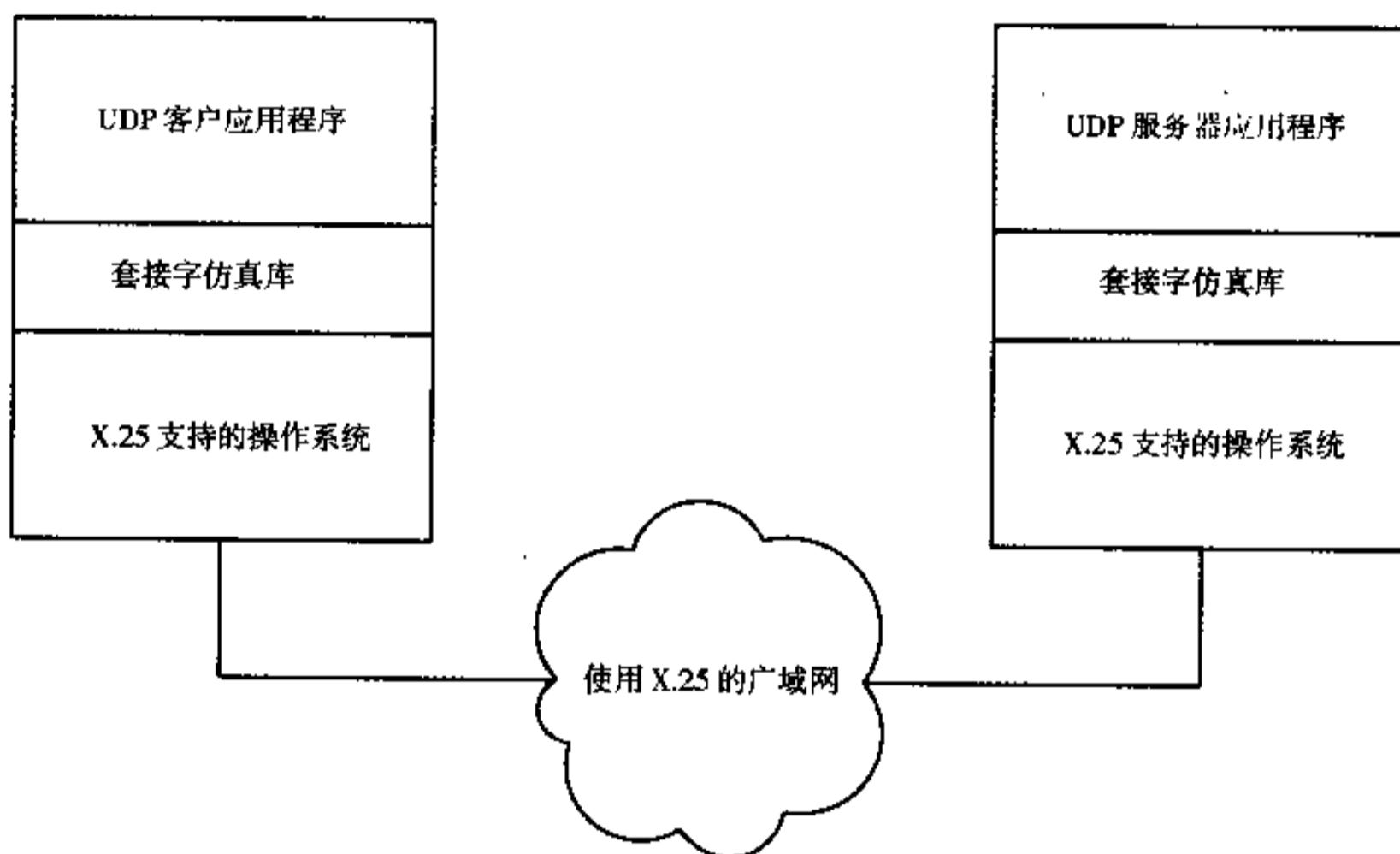


图 18.3 一种客户和服务器的概念性组织结构，使用了应用级隧道技术通过 X.25 网。
套接字仿真库允许客户和服务器通过非 TCP/IP 传输服务交换 UDP 数据报

18.8 隧道技术、封装和电话拨号线

调制解调器可使两台计算机之间通过电话拨号线通信。现在已有一组协议，包括串行线路 IP (Serial Line IP, SLIP) 和点到点协议 PPP (Point-to-Point Protocol)，用于在拨号线上发送 IP。

拨号线上的 IP 传输应该看成是隧道技术或封装的一种形式吗？当然如此，使用拨号与本章讨论的隧道技术的思路类似。电话系统可看成是一个传输系统，在其上 IP 数据报用隧道技术传送。实际上，拨号链接的管理与 X.25 连接很相似。

但是，大多数专家却认为拨号系统不应看成是传输系统，而应看成是一个面向连接的物理网络。因此，诸如 SLIP 和 PPP 的协议定义了一种形式的封装——它们各自定义了一种链路级的组帧 (framing) 格式，规定了如何封装数据报以用于传输。类似专用的串行线路，电话系统可用于连接不同网点中的路由器。下一章将说明如何扩充 SLIP 和 PPP，使它适用于异构寻址环境中的拨号连接。

18.9 小结

隧道技术是使用传输级分组交付系统（而不是直接通过物理网络发送）在计算机之间发送分组。由于某些机构在 TCP/IP 可供使用前已拥有了较大的广域网，这就在早期激发了对穿越已有网络系统的 IP 隧道技术的研究。这些机构不希望为运行 IP 而增加新物理连接的开销。研究人员提出一些方法，允许 IP 使用现有的网络传送分组，而不需改变现有的网络。IP 将传输服务处理为单个物理连接；传输服务将 IP 通信量（traffic）与任何应用程序发出的通信量一样处理。

IP 已成为提供最大互操作性的交付系统。因此，现在对隧道技术的研究集中在如何寻找一些方法，使得 IP 可以作为传送其他网络协议分组的分组交付系统。许多厂商已宣布有了这样的软件，可以使其专有的网络系统通过下层 IP 互联网进行通信。

编程人员可将隧道技术的思想用到应用软件中，方法是通过构造一个仿真套接字接口库，而该库使用一种非 TCP/IP 传输服务交付报文。具体说就是，可以很容易地构造一个套接字仿真库，该库允许客户和服务器使用 UDP 通信，即使在客户和服务器的计算机之间的连接只能由一个传输级交换系统（像 X.25 网）组成时，通信也能进行。

隧道技术可用在电话拨号连接上，也可用在永久连接上。要使用电话系统，两个路由器必须有拨号调制解调器，并且必须就链路级协议达成一致。

深入研究

Comer 和 Korb [1983] 描述了如何使 IP 贯通一个 X.25 网，以及当硬件强制限制了同时打开的连接数时，如何管理 X.25 虚电路。RFC877 还给出了技术编码细节。

习题

- 18.1 阅读 RFC 877。一个路由器如何使用隧道技术将一个目的 IP 地址映射成等价的 X.25 地址以便通过 X.25 网？
- 18.2 许多传输级服务使用其自己的重传策略提供可靠的交付。如果 TCP 和下层网络协议都重传报文，会发生什么情况？
- 18.3 我们说过，路由器使用动态虚电路分配方法通过隧道贯通面向连接的网络，并且通常采用 LRU 以便在需要关闭一个现有的虚电路时能够有一条可供使用。如果路由器的接口允许同时有 K 条电路，并且路由器尝试与 $K+1$ 个其他网点同时通信，试解释在一个路由器中会发生什么情况？
- 18.4 构造一个套接字仿真库，它允许客户和服务器的应用程序在非 TCP/IP 传输级协议上交换 UDP 数据报。试对此进行测试，方法是让一个 UDPECHO 客户与一个 UDPECHO 服务器通信。

第19章 应用级网关

19.1 引言

上一章研究了隧道技术，这种技术允许一种协议族使用其他协议族的传输级服务以代替一个物理网络。从应用程序员的观点看，即使在客户和服务器间只有非 TCP/IP 网络的通路，隧道技术也能使客户和服务器使用 TCP/IP 通信成为可能。

本章继续探索在异构环境中，客户和服务器通信所使用的技术，说明应用程序怎样作为两个不同服务之间的中介，还将说明如何用这种中介扩充可用服务的范围。

19.2 在受约束的环境中的客户和服务器

19.2.1 限制访问的现实

在前面几章所描述的客户-服务器体系结构中，我们假设计算机都连接在一个单一的同构网络里，传输软件提供必要的端到端的连接。这种直接的、统一的访问方式并非总能获得。即使下层的设施能够提供通信能力，但因为经济和政治原因，可能有访问限制等问题。例如，像第 18 章所指出的，许多机构的联网是缓慢发展的，因此，各个下属组可能选定某种协议软件以适应其特定的应用需求。更为重要的是，客户或服务器软件有时来自其他应用的集成套件——为完成某种特定任务的产品可能包含其他客户或服务器服务。

对程序员来说，在异构环境下构建软件更为困难。程序员必须满足各种不兼容系统的需要。除非这个机构提供隧道技术，否则编程人员不能依靠端到端的传输级的连通，因此，他们不能依赖于某一种传输协议为任意一对机器提供通信，也不能提供一种能在任意机器上运行的客户或服务器软件。

有些程序员分别针对每种计算机和每种网络构造和维护一个单独的程序，他们想通过这种方法解决异构问题。例如，如果某个机构里有三种类型的网络，那么为此机构工作的程序员就可能要构造和维护三种单独的电子邮件系统。

19.2.2 有限功能的计算机

编程人员除了要对付多个网络，有时还必须为存储器、CPU 以及协议软件都受限制的小型嵌入式系统编写程序。尽管这类设备的运作看上去和常规计算机一样，但它们不能支持第 8 章讨论的并发服务器算法，或者第 17 章讨论的并发客户算法。

19.2.3 安全性引起的连通性约束

许多机构都制定了安全措施，但安全措施可能也约束了客户和服务器间的通信。一些机构将计

计算机分成安全子集（secure subset）和不安全子集（unsecure subset）。为防止客户和服务器程序的安全受到危害，网络管理人员对连通性设置了约束措施。管理人员将计算机束缚在安全分区内，以便它们之间能够相互通信。但是，这些计算机既不能开始同非安全区的计算机上的服务器联络，也不能接收来自非安全区里计算机上的客户请求。虽然这种措施保证了安全，但会使编程人员很难设计使用客户 - 服务器交互方式的应用程序。特别是，一个分区内的计算机不能直接访问其他分区计算机所提供的服务。

19.3 使用应用网关

需要在受限制的环境中设计客户 - 服务器交互的程序员，通常依靠一种强有力的技术来克服这种连通性约束。这种技术是在一些中间（intermediate）的机器上附加一些应用程序，并允许这些应用程序在客户和它所期望的服务器之间传递信息。提供这种服务的一种中间程序称为应用网关（application gateway）^①。如果中间机器专门用于运行某个应用网关程序，编程人员或网络管理员有时将该机器称为网关机（gateway machine）。例如，如果一台计算机专门用于运行一个在两个电子邮件域间传递电子邮件的程序，则该机器可称为邮件网关（mail gateway）。当然，从技术上讲，应用网关（application gateway）这一术语是指运行着的程序——但程序员将某台机器称为应用网关时，已将这一术语的概念延伸了。

图 19.1 显示了应用网关的一种通常用法，即位于两个电子邮件系统间的一个中介。

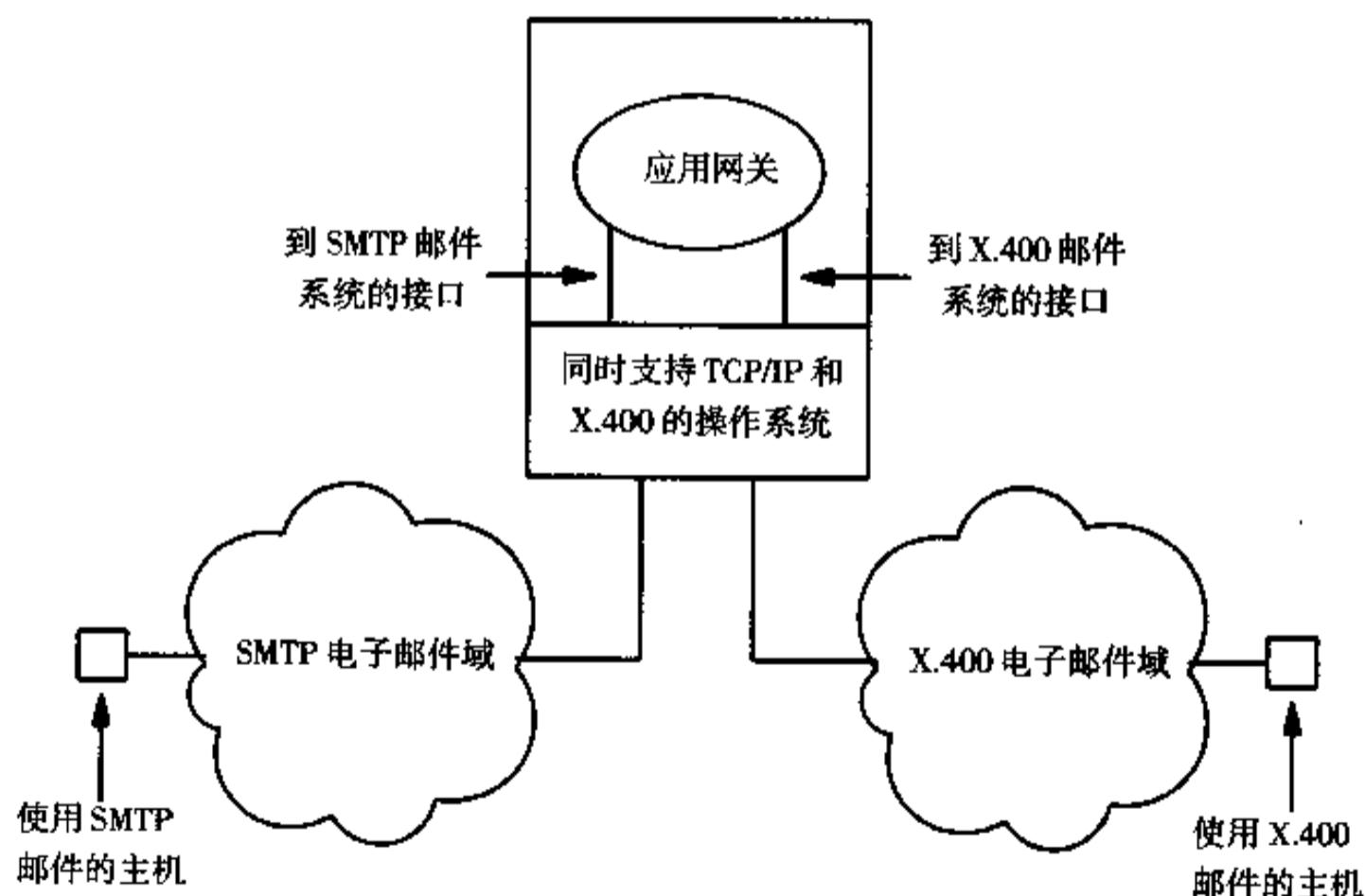


图 19.1 一种应用程序用于在两个不同的邮件域之间传递电子邮件。应用网关知道两个邮件系统的语法和语义，并且在两个系统之间转换邮件

图 19.1 描绘的机构连接到两个电子邮件系统：TCP/IP 因特网所使用的 SMTP 标准和 X.400 标准。每个电子邮件系统各有自己的邮件语法和传输协议。从广义的角度看，两个系统提供同样的服

^① 网关这一名称可能会产生混淆，因为 IP 路由器原来称为 IP 网关。为避免混淆，我们应对两者加以仔细区分。

务。每个系统允许用户编写和发送邮件，或者接收和读取收到的邮件。但是，两个系统不能直接互操作，因为每个系统都有自己的目的地址语法和自己的邮件交付协议。

为能让某一类电子邮件系统上的用户与另一类型系统上的用户交换邮件，这个机构安装了作为邮件网关的应用程序。在本例中，邮件网关程序运行在能访问两类电子邮件的计算机上。邮件网关必须仔细设计，以便能与机构中的任何主机通信。它必须知道如何使用这两种邮件系统来发送邮件，并且还必须具有到两个网络的逻辑连接。

19.4 通过邮件网关互操作

对于图 19.1 中所示的机构，单个邮件网关程序能提供该机构为电子邮件建立互操作性所需的所有设施。通常，整个机构中的每个主机将对外发邮件（outgoing mail）的目的地址进行检查，并对下一跳（next-hop）的机器进行选择。如果发出的邮件是要发给与发送方在同一网络的机器，发送方就使用本地网络上所使用的电子邮件系统交付邮件。但是，如果主机遇到的外发邮件要发往另一个邮件域上的机器，发送方就不能直接交付邮件。它必须将邮件传送给邮件网关程序（mail gateway program）。所有机器都可与邮件网关直接进行通信，因为邮件网关与这两个域都相连，并且可使用两个邮件传输协议中的任一个进行通信。

一旦邮件到达邮件网关，它必须再次转发。邮件网关检查目的邮件地址，决定如何继续向前转发。为做出决定，它可能还要查询目的地数据库。一旦邮件网关知道目的地和交付邮件所要通过的邮件系统，它就选择合适的邮件传输协议。

网关在不同网络之间转发邮件时，可能需要修改邮件的格式，或者改变邮件的首部（header）。具体说来就是，邮件网关往往要修改邮件首部的应答字段，以便接收方的邮件接口能正确构造一个应答地址。修改应答地址可能较简单（如增加一个能标识发送方网络的后缀），也可能较复杂（如增加一些信息，这些信息将邮件网关标识为能回溯到源方的中间机器）。

19.5 邮件网关的实现

理论上说，一个单控制线程足以实现一个邮件网关。但实际上，大多数实现是将邮件网关的功能分成两个线程，每个线程都在一个单独的进程中。一个线程处理传入邮件（incoming mail message），而另一个线程则管理外发邮件（outgoing mail message）。处理传入邮件的线程从不发送邮件。它计算出应答地址，为邮件选择到目的地的路由，然后将出报文存入队列等待发送。处理外发邮件的线程不直接接受传入邮件，它只定期扫描输出队列。对在输出队列中找到的每个邮件，形成一个到目的地址的连接，然后发送该邮件。如果输出线程不能创建到目的地的连接（如由于目的主机崩溃了），它就让该邮件留在队列中，并继续处理队列中的下一个邮件。然后，当输出线程重新扫描队列时，它将再次与目的主机联络并交付该邮件。如果一个邮件留在输出队列中的时间很长（如 3 天），输出线程就向原来发送邮件的用户报告交付出错。

将邮件网关分为输入和输出两部分可允许每个部分独立地进行处理。输出线程可以尝试交付一个邮件，等待查看连接尝试是否成功，然后继续处理下一个邮件，而这并不需要与输入线程相互协调。如果连接尝试成功，输出线程就可以发送一个邮件，而不必考虑邮件的长度。它不需要中断传输去接受传入邮件，因为输入线程可以处理它们。同时，输入线程可继续接受传入邮件，为它们选择路由，然后存储和转发。由于这两部分独立运行，长的输出邮件不会阻塞输入处理过程，而长的输入邮件也不会干扰输出处理过程。

19.6 应用网关与隧道技术的比较

前一章说明了设计人员可选用隧道技术来提供异构环境下的互操作。在隧道技术和应用网关间进行选择是困难的，因为没有哪种技术能很好地解决所有问题，而且每种技术都各在某些情况下显示了一些优点。

使用应用网关而不使用隧道技术的主要优点在于：程序员创建应用网关时，可以不必修改计算机的操作系统。在许多情况下，程序员不能修改操作系统，因为他们不能找到源代码，或者不具备修改系统所需的经验。应用网关可使用传统的编程工具构造；网关不需要改动任何下层协议软件。此外，一旦应用网关安装好了，网点就可使用标准的客户和服务器程序。

使用应用网关有第二个优点：它允许所有现存的网络系统继续运行而不会受到影响。管理员不需要学习新的网络技术，也不需要改变任何物理网络连接。类似地，用户也不需要学习这些服务的新接口；每个用户继续使用现有的与网络相关联的客户软件，而这些正是他们已熟悉的。

应用网关也有一些缺点。应用网关技术需要编程人员为每个服务构造单独的应用网关。邮件网关互连两个独立系统的邮件服务，但不提供远程文件访问或远程登录的功能。每当机构要给其网络系统增加新的服务时，程序员都必须构造新的应用网关，以便连接多个网络之间的新服务。

应用网关可能还需要额外的硬件资源。机构可能需要购买一些新的计算机，或者可能需要为已有计算机增加网络连接。增加新的网络连接可能意味着要求额外的软件以及额外的硬件。由于转发报文时需要转换，或者数据可能较复杂，应用网关可能要使用大量的 CPU 或存储器。因此，机构在增添新的服务时，有必要购买额外的计算机或将已有的机器进行升级以便处理负载。对 CPU 资源的需求也引入了计算的时延，这就增加了客户和服务器之间的时延。如果时延太长，客户可能超时并重发报文。

与应用网关相比，隧道技术在出现新的服务时不需要任何改变。传输级隧道一旦构成，它就成为下层网络结构的一部分。由于应用程序意识不到隧道的存在，隧道可用于任何应用服务。隧道技术还提供了一致性，因为它意味着这个机构可使用单一的传输协议。

隧道技术与应用网关相比也有缺点。为安装提供全部功能的传输级隧道，网点必须修改要连接两个网络系统的网关上所运行的操作系统。令人吃惊的是，这个机构可能还需要修改使用隧道的主机上的软件。要理解为何需求这种修改，让我们设想一个 X.25 网络被配置成可以通过 IP 通信业务的隧道。考虑连接到该网络的主机。在应用程序能使用 IP 隧道之前，它必须可访问 TCP/IP 协议软件，而且 IP 软件必须知道如何让数据报穿越 X.25 网络。因此，主机操作系统必须为应用程序提供 IP 接口（即套接字级的接口），并且必须使通信业务通过这个隧道。如果操作系统不包含 TCP/IP 协议软件，就必须加上。如果现有的 IP 协议软件不知道如何令通信业务通过隧道，那么就应当修改这个软件。

隧道技术也能对用户产生巨大影响。许多机构采用了隧道技术，把它作为在异构网络上提供一致的传输服务的一种方法。一旦机构建立了一条隧道，所有主机便开始使用单一的传输协议，以此用于客户 - 服务器之间的交互。例如，如果机构选用了 IP，并且为提供连通性而创建了贯穿于 X.25 网络的隧道，则整个机构中的计算机都将用 TCP/IP 进行传输连接。随之而来的后果是，所有的计算机都能够支持使用 TCP/IP 的客户 - 服务器之间的交互。

遗憾的是，下层网络协议的改变，通常会导致用户用来进行交互的客户软件的改变。大多数机构购买商用客户软件是用于一些标准应用，例如电子邮件。因此，一个机构不管使用的是什么软件，都必须是在某一给定传输协议下可以使用的软件。如果该机构采用了一种通用的传输协议，那么还

必须购买使用这种新协议族的新客户软件。从用户的观点看，更换成新软件就意味着要学习一种新接口。除非新接口能提供现有接口的所有功能，否则用户可能是不会满意的。

为避免改变用户的环境，许多机构选择应用网关方案。编程人员谨慎地构造应用网关，因此不需要改变任何网络上的用户接口。例如，一个机构为电子邮件增加一个应用网关后，用户可使用原来的客户软件发送和接收邮件。邮件系统可使用目的地址语法，以便区分发给本地机器的和非本地机器的报文。这样做允许用户对本地目的地使用旧地址，而只需要他们去掌握为新的目的地使用的新地址。

19.7 应用网关和有限因特网连接

应用网关可使连接能力有限的计算机所得到的服务范围增加。例如，考虑一个为用户提供拨号因特网访问的ISP，用户没有永久的因特网连接，而是在拨号后才进入在线状态。同样，对某个公司网络的用户，他可能有一台便携式计算机，每天下班后，他会把计算机从公司网络上断开，然后带回家。

具有有限连接能力的计算机可以运行客户软件，因为计算机在执行客户软件之前，可以一直等待建立连接。例如，对电子邮件来说，在用户写好报文后，报文可以存储起来，直到客户程序能够将报文的副本传输到目的地为止。但具有有限连接能力的计算机不能运行电子邮件服务器，因为使计算机断开连接意味着电子邮件服务器将变得不可用。

应用网关为这种主机不能运行服务器的场合提供了解决方案。各个机构创建一个能执行两种功能的应用网关：一是该网关作为一个服务器运行，代表各个主机接受所有的传入信息；二是网关允许主机访问已经到达的信息。

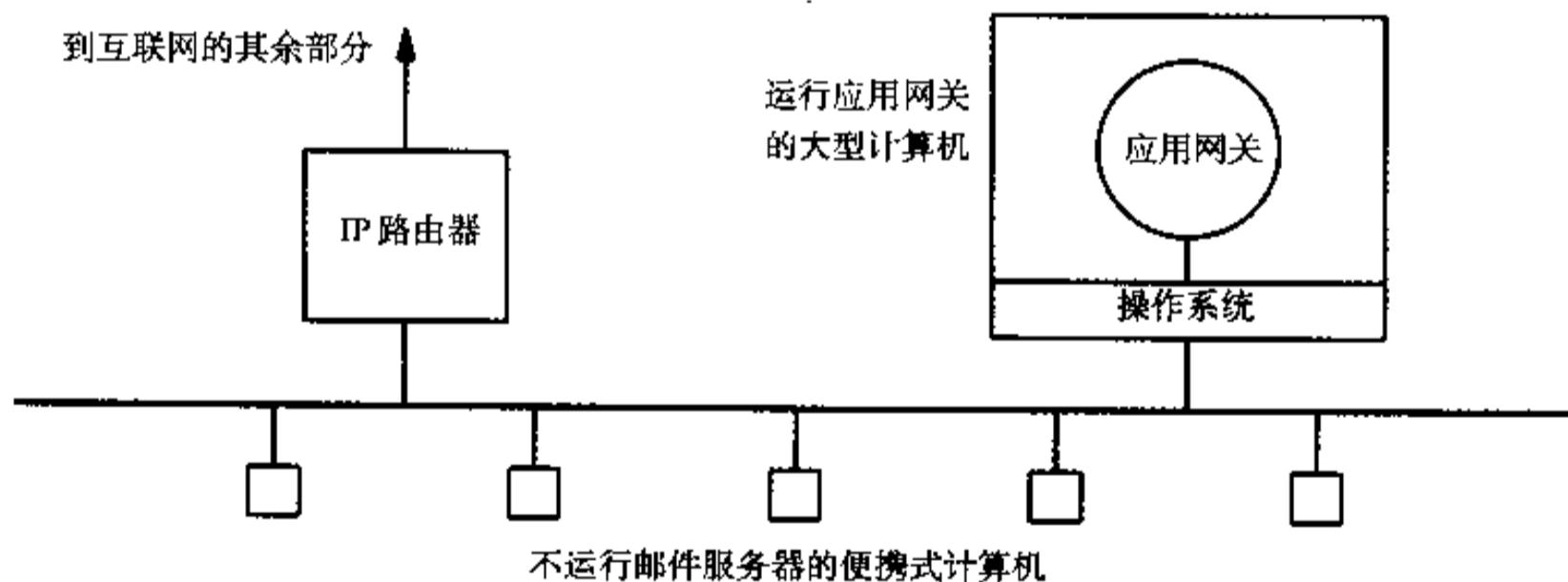


图 19.2 这个网络由便携式计算机以及供这些计算机访问电子邮件的应用网关构成。该网关一直保持能接受电子邮件的状态

如图所示，该机构必须至少拥有一台永久连接的计算机，该计算机运行接受传入电子邮件的软件，服务器可以作为应用网关的一部分实现，也可以采用标准的服务器。无论选择哪种方式，服务器都必须24小时随时准备接受传入邮件。当有邮件到达服务器时，服务器就将它存放在磁盘上的文件中。存储邮件的文件通常称为邮箱文件（mailbox file）或邮箱（mailbox）。系统可能为每个用户保留一个邮箱文件，或者将每个邮件存入各个单独的文件中。通常，为每个邮件使用单独文件的实现是将文件集中收集到若干目录中，而每个目录对应于一个用户。

除了标准的邮件服务器，计算机还必须提供一种应用网关服务，以便允许用户访问他们的邮箱。便携式计算机上的用户为了阅读邮件，要调用一个联络应用网关的客户。在进行身份鉴别之后，用户发出命令，让网关从用户的邮箱中检索邮件并将其发送给客户。此外，用户还可指明，应用网关是否保存或删除有关邮件。

应用网关的主要特征之一是支持异构协议，这对电子邮件传输是很关键的。当因特网上的某个主机发送报文时，它采用的是 SMTP 协议，但当便携式计算机用户访问邮箱时，并不使用 SMTP，而是使用 POP（邮局协议）这样的协议。更重要的是，应用网关可以转换邮件格式，这就允许便携式计算机上的表示方式可以与因特网上其他计算机的表示方式不同。

19.8 为解决安全问题而使用的应用网关

许多机构选择应用网关来解决安全问题。例如，假设一个机构需要限制远程登录。想像该机构针对远程登录将其雇员分为授权的（authorized）或未授权的（unauthorized）两类。图 19.3 说明了该机构如何使用应用网关实现其安全策略。

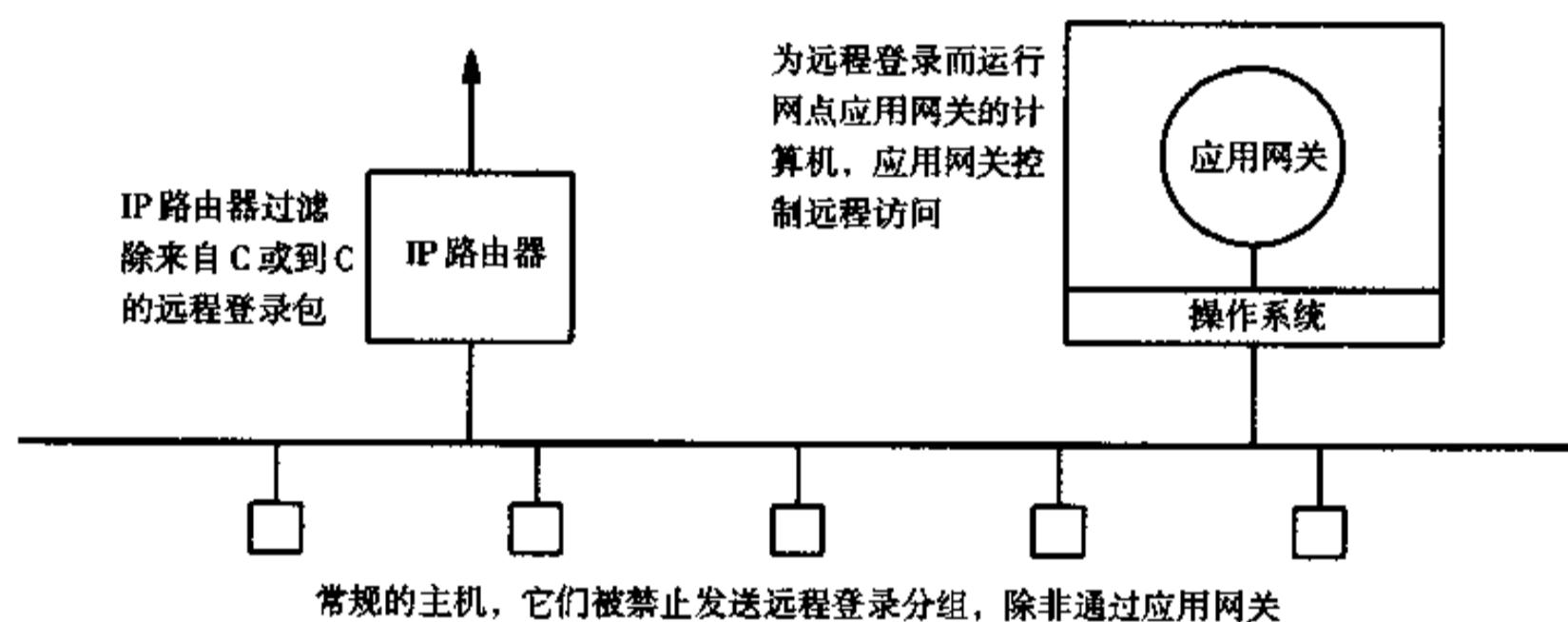


图 19.3 用于实现远程登录安全策略的应用网关。用户必须通过应用网关才能与远程机器通信，应用网关实施授权控制

图中描绘的机构使用了常规 IP 路由器或防火墙进行过滤，除了运行应用网关的主机发出的远程登录请求外，其他含有远程登录请求的数据报都将被阻止。机构中任何用户要形成远程登录请求，都要调用客户，该客户首先连接到应用网关。用户得到授权后，应用网关再将用户连接到所期望的目的地。

19.9 应用网关和额外跳问题

额外跳（extra hop）问题是指数报在到达最终目的地的过程中要两次通过同一个网络。该问题通常是由不正确的路由表引起的。

给现有的网络引入应用网关也能产生一种形式的额外跳问题。要理解其原因，我们考虑图 19.4a 所示的网络拓扑。图中显示的是当主机与远端服务器支持同一种传输协议时，报文从该主机送到远端服务器所经过的路径。现假设现有的主机希望访问这样一种服务，该服务只能通过与主机

所用协议不同的某种协议来获得。通过引入图 19.4b 所示的应用网关就能获得互操作性。应用网关使用一种协议系统来接受请求，而使用另一种协议系统将其送给远端服务器。遗憾的是，每个报文要通过网络两次。该图是符合现实的：由于网络管理员希望避免让现有机器负担过重，他们通常为每个应用网关程序使用一台新的计算机。

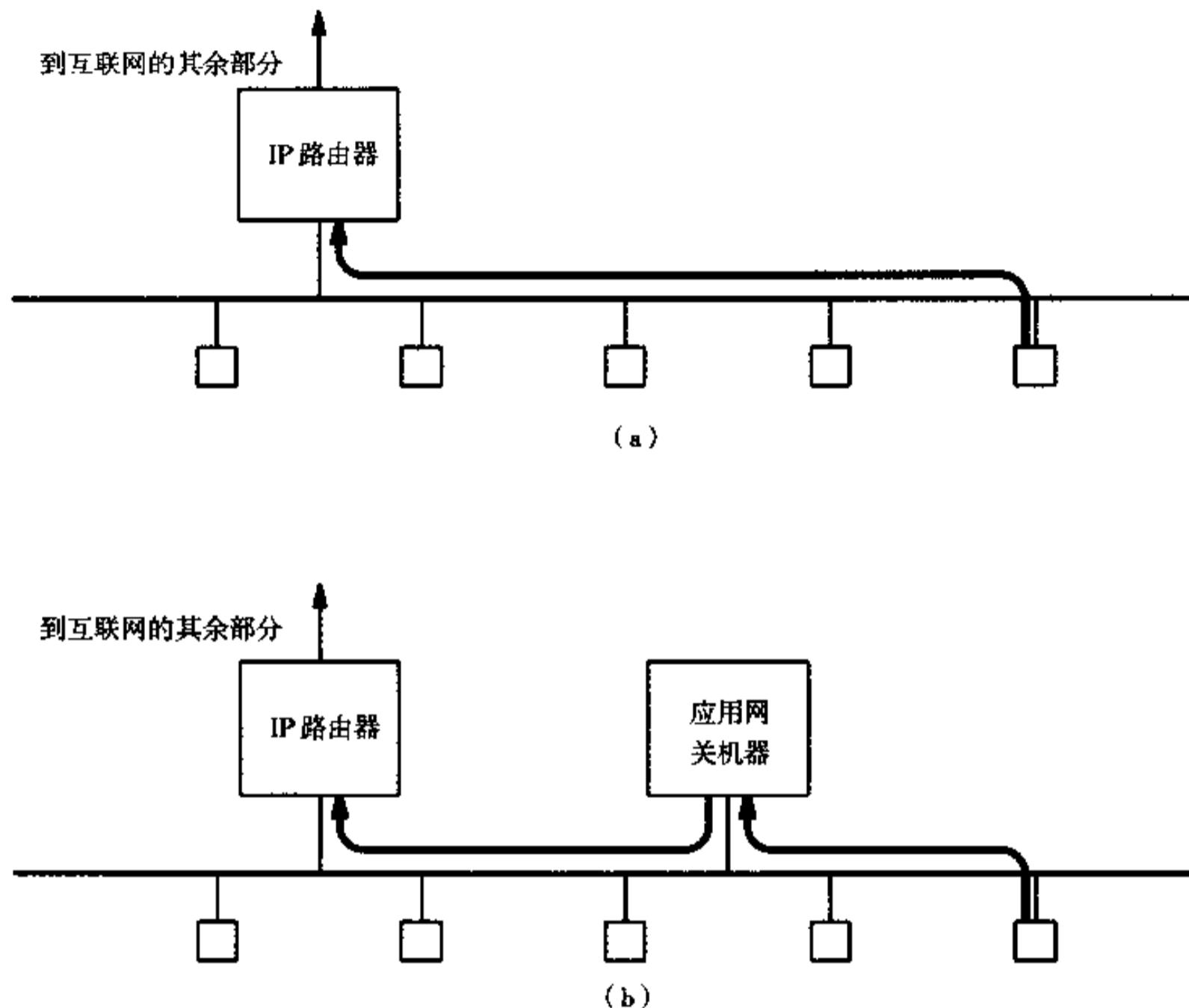


图 19.4 (a) 若干主机和一个网关。粗箭头表示报文从一台主机到远端服务器的路径。
(b) 引入应用网关后报文的路径。新网关将使得每个报文通过网络两次

一旦引入了应用网关，在现有各主机上执行的客户都将使用应用网关来访问它所提供的服务。客户使用一种协议向应用网关发送请求，而应用网关使用另一种协议将请求转发给远端服务器。当服务器给应用网关返回结果时，网关就将响应送回客户。系统看起来是工作得很好。主机上现有的协议软件不需要改变。在安装好应用网关后，运行在任何主机上的客户能够通过网关访问所需服务。

遗憾的是，只要仔细检查下层的网络就可发现图 19.4b 中的配置不能很好利用网络资源。它产生了额外跳的问题。每个请求必须通过本地局域网两次：一次是它从源主机到应用网关机器，而另一次是从应用网关机器到最终的服务器。如果服务器位于 IP 路由器另一边的 TCP/IP 互联网中，当报文从应用网关机器传递到 IP 路由器时，就会发生第二次传输。如果服务器位于本地网上，在报文从应用网关机器传给运行服务器的机器时，就会发生第二次传输。

对于不需要很多网络通信量的服务而言，额外跳可能并不重要。确实有几个厂商构造了使用图 19.4b 所示拓扑的协议。但是，如果网络的负载很重，或者如果服务非常需要网络流量，额外跳就使得这种方案太为昂贵了。因此，设计人员在采用应用网关方法前，需要仔细计算预期的负载。

19.10 应用网关举例

通过给那些并非运行所有协议的客户机器提供访问，应用网关可以扩展服务。例如，考虑一个主机上的一位用户，他可使用电子邮件，但不能使用如 FTP 这的文件传输协议。这种限制可能是由于经济考虑（如 FTP 软件的费用太高）、商业事实（如无法为所考虑的计算机买到 FTP 客户软件）或者安全因素（如网点决定禁止高速数据传送以减少安全威胁）。

假设在一个受限制的机器上的用户需要访问 RFC (Request For Comments) 文档。应用网关技术能够解决这个访问问题，只要使该机构与电子邮件和 FTP 服务互连起来，并同时控制访问和保证授权。

为了从电子邮件提供 RFC 访问，应用网关必须与这两个服务都相连。为使用网关，用户必须向网关发送指明所要求的 RFC 的电子邮件。应用网关验证用户是否被授权访问 RFC，然后完成 FTP 连接，获得 RFC 的一个副本，然后将 RFC 文档放在一个电子邮件中发回给该用户。

例如，想像应用网关的电子邮件地址是机器 somewhere.com 上的 rfc。想像应用网关接受了邮件，该邮件的主题 (subject) 行中注明了 RFC 号，应用网关使用 FTP 取回该 RFC，然后将结果邮回发送者。

为了使用该网关，用户将创建一个发给 rfc@somewhere.com 的邮件：

```
To: rfc@somewhere.com  
From: user@elsewhere.edu  
Subject: 791
```

报文主体 (可能为空)。

这个邮件从应用网关请求 RFC 791。由于网关只查看主题行中的数字，邮件主体中的文本无关紧要；它可能是空的。

19.11 一个应用网关的实现

为了实现以上描述的应用网关例子，编程人员需要三个设施：一台可访问电子邮件和 FTP 的机器，一个充当应用网关的程序，以及一种机制，这种机制要将每个发给指定目的地的邮件送到网关程序。

包括 Linux 在内的大多数 UNIX 系统都允许系统管理员很容易地建立这些必要的部件。管理员可以创建一个特殊的邮件目的地 rfc，要做到这点，他只要在电子邮件的别名文件中增加 rfc，或者，管理员可以为一个名为 rfc 的虚构用户创建一个账号，并且把给该用户的所有邮件转发到一个程序。实际上，大多数 UNIX 系统允许各个用户将收到的电子邮件重定向到一个程序，因此对一个非特权用户来说，构造并测试一个电子邮件应用网关是可能的。

在其他任务中，应用网关需要取回某个 RFC。我们的例子实现使用了一个单独的程序取回 RFC 文档。程序如下所示，它是一个 UNIX 命令解释器脚本 (shell script)。

```
#!/bin/sh  
  
#  
# rfc:          retrieve an RFC document given its number
```

```
#  
# operation:      check the local cache for a copy of the requested  
#                  document and use FTP to obtain a copy from an archive  
#                  if none present.  Store a copy in the local cache when  
#                  retrieving from the archive.  
  
#  
# method:        keep the local cache in directory /usr/tmp/RFC; use in-line  
#                  commands to invoke the ftp command.  Assume any file name that  
#                  ends in .Z contains a compressed file and run zcat to  
#                  decompress it.  
  
#  
SERVER=ftp.isl.edu  
PATH=/bin:/usr/bin  
CACHE=/usr/tmp/RFC  
USER=anonymous  
PASS=guest  
umask 022  
  
if test ! -d $CACHE  
then  
    mkdir $CACHE  
    chmod 777 $CACHE  
fi  
for i  
do  
    if test $i = "index"  
    then  
        i ="-index"  
    fi  
    if test ! -r $CACHE/$i -a ! -r $CACHE/$i".z" -o $i = "-index"  
    then  
        trap "rm -f $CACHE/$i;  
               echo int - $CACHE/$i removed;exit 1" 1 2 3 13 15 24 25  
        ftp -n ${SERVER} >/dev/null <<!  
user $USER $PASS  
binary  
get rfc/rfc$i.txt $CACHE/$i  
quit  
!  
                                trap 1 2 3 13 15 24 25  
fi  
if test -r $CACHE/$i  
then  
    cat $CACHE/$i  
elif test -r $CACHE/$i".z"  
then  
    zcat $CACHE/$i
```

```

        else
                echo Could not retrieve RFC $i
        fi
done
chmod 666 $CACHE/-index >/dev/null 2>&1

```

rfc 脚本带有一个参数，指明是一个 RFC 号或是单词 index（用来表示 RFC 索引）。它创建所需的 FTP 命令，然后调用 ftp 程序。命令指示 ftp 打开一个到网络信息中心的连接，并且取回指明的 RFC。

由于 rfc 脚本含有优化，因而它比必要部分更复杂。代码将它取回的 RFC 存到目录 /usr/tmp/RFC 中作为快速缓存。在 rfc 程序获取一个 RFC 之前，它将查看当前高速缓存（cache）中是否有该 RFC 的副本。如果有，脚本将从高速缓存中取得副本，而避免了不必要的网络流量。

19.12 应用网关的代码

一旦在系统中安装了 rfc 脚本，构造一个访问 RFC 的应用就很简单了。文件 rfcd 含有这种代码。类似以上的 rfc 程序，它也是一个命令解释器脚本。rfcd 使用 rfc 程序读取一个 RFC 文档；它使用常规的电子邮件程序（/usr/ucb/mail）将结果发回给请求该 RFC 的用户。

```

#!/bin/sh
#
# rfcd:      an application gateway between e-mail and FTP
#
# operation: receive an e-mail message, extract an RFC number from the
#             subject line, invoke a program that obtains a copy of the
#             RFC, and send it back to the user that sent the e-mail
#
# method:    use awk to parse the input, extract information, and form
#             a shell program; pipe the program directly into the shell
#             to execute it.
#
PATH=/bin:/usr/bin
RFCPROG=/usr/local/bin/rfc
awk "
BEGIN      { fnd=0 }

/^From:/   { retaddr=substr( \$0, 6);next}

/^Subject:/ { rfcnum=substr( \$0, 9);fnd++ ; next }

/^ *\$/     { if (fnd==0 || rfcnum!~/^[-0-9]*$/) exit
               cmd=\"$RFCPROG \"$rfcnum
               cmd=cmd\" | /bin/mail -s 'RFC \"$rfcnum\" '\"
               cmd=cmd\" '\"retaddr\" '\"'
               print cmd ; exit
} " | /bin/sh

```

对于不熟悉命令解释器脚本和awk实用程序的读者，这些细节似乎较复杂，但是应该明白，创建脚本几乎不需要多少编程工作。

从实质上讲，rfcd脚本读取一个邮件，并提取From:行和Subject:行的内容。它使用一个叫做awk的程序完成大部分工作。awk程序扫描传入邮件中的每一行。它假定Subject:行含有一个RFC号，并且From:行含有发送方的电子邮件地址。发送方的电子邮件地址在rfcd产生响应时将成为目的地址。

一旦rfcd中的awk程序检测到一个空行，它就知道已到达了邮件首部(mail header)的末尾。它停止处理并形成和打印一个命令解释器程序。由于rfcd直接将awk程序的输出送到命令解释器(sh)，因而产生的程序会立刻执行。

如果传入邮件首部中含有一个有效的Subject:行，产生的程序就调用/usr/local/bin/rfc获取指明的RFC，接着调用/usr/ucb/mail将RFC传送给发送请求者。如果邮件不含主题(subject)，或者如果主题行中含有不同于数字和空格的其他内容，awk程序将退出且不产生任何输出。当awk程序过早退出时，rfcd不会试图获取RFC，也不会发送任何邮件(也就是说，这个版本甚至不会返回出错消息)。

19.13 网关交换的例子

用一个例子可阐明上述步骤。假设用户comer决定请求RFC 1094。Comer发送邮件给应用网关地址rfc@somewhere.com，指定Subject:行为1094。该邮件含有：

```
To: rfc@somewhere.com  
From: comer  
Subject: 1094
```

报文主体(可能为空)。

在somewhere.com上的邮件系统已被配置，它将把传入邮件传递给rfcd脚本。

当rfcd运行时，awk代码从邮件的Subject:行中提取1094，并且将1094存放到变量rfcnum中。它从From:行中提取comer，并存入变量retaddr中。当rfcd遇到分隔邮件首部和邮件主体的空行时，它会产生如下的命令行：

```
/usr/local/bin/rfc 1094 | /usr/ucb/mail -s 'RFC 1094' 'comer'
```

Linux用户会认得此行，它是命令解释器bash的有效输入。它指明命令解释器应该运行/usr/local/bin/rfc文件中的程序，且参数为1094，然后使用管道(pipe)机制，将输出连接到/usr/ucb/mail文件中的程序。邮件程序有三个参数：-s、RFC 1094和comer。前两个参数指明主题中含有字符串RFC 1094。第三个参数指明接收方为comer。邮件主体由通过rfc脚本获得的RFC文档组成。

由于rfcd将awk程序的输出直接送给命令解释器，命令解释器将执行以上所示的命令行。它将运行rfc程序，然后将结果用电子邮件发送给用户comer。

19.14 使用rfcd和.forward或.slocal文件

每个用户可使用他们自己的电子邮件账号测试rfcd。为此，用户在各自的主目录中要创建一个

名为.forward 的文件。邮件系统每次收到一个邮件，便从 To: 字段中提取接收方的名字，并检查用户主目录中是否有.forward 文件。如果找到一个，邮件系统将读取.forward 文件，并遵循其中的说明。例如，假设某个用户希望邮件系统将他收到的每个邮件都送给一个全路径名为 /usr/XXX/Q 的程序，用户在他的.forward 文件中放置一行：

```
| "/usr/XXX/Q"
```

邮件系统将这一行解释为：调用程序 /usr/XXX/Q，且将收到的邮件作为标准输入。因此，用户若要调用如上所示的电子邮件应用网关，就必须将 rfc8 脚本放入一个文件，使得文件可执行，并且让 forward 的一项指向它。

除了这种从基础开始构建邮件网关的方法外，还有另一种有趣的方法。Linux 有一个叫做 slocal 的程序，该程序能够处理很多工作。用户可以通过.forward 文件指定每个传入邮件必须传递给 slocal，Slocal 根据一个说明文件（.maildeliver）来确定如何处理邮件。用户可以安排让某些传入邮件进入邮箱，另一些则传递给应用网关。

19.15 通用的应用网关

现在有一种有趣的应用网关，它同电话拨号线一并使用。该网关称为 SLIRP，由于它可接入多种服务，因而显得不同寻常。特别是，SLIRP 允许运行在某台家庭计算机上的应用程序能够接入到 TCP/IP 互联网上的任意应用程序。

为了充分理解 SLIRP 的设计，有必要知道：SLIRP 解决的问题不同于 SLIP 和 PPP。SLIRP 解决了一种重要的 IP 地址问题，而不是在串行线路上定义一种封装协议。当家中计算机上的用户希望拨号接入到 Internet 时，就会出现问题。在传统的 IP 寻址模型中，每台计算机必须被分配一个唯一的 IP 地址。遗憾的是，拨号接入使得验证某个给定 IP 地址的拥有者十分困难，并且，使用该地址实施接入限制也很困难。因此，出于安全考虑，许多计算机公司不将 IP 地址传出给远端计算机，并且他们将拨号限制为常规的终端登录。

限制拨号接入的主要缺点是它限制了功能：虽然拨号终端会话可使用面向字符的协议，如 FTP 和 TELNET，但用户却不能接入到像 WWW 这样的服务，WWW 需要客户具有完整的 IP 地址。SLIRP 克服了这种限制，它使用一种在拨号线上提供 IP 接入的机制，但却不需要为每台计算机分配一个 IP 地址。

19.16 SLIRP 的运行

在既可接入到 Internet 又有电话拨号调制解调器的计算机上，SLIRP 可以当成应用网关来运行。要使用 SLIRP，就必须建立常规的终端会话。也就是说，当家用计算机上的用户第一次拨号到运行 SLIRP 网关的机器 G 时，用户通过发送登录标识符和口令建立终端会话。一旦用户在机器 G 上建立了终端会话，就要很快地连续完成两个步骤。第一步，调用 G 上的 SLIRP。第二步，删除家用计算机上的终端会话，并且允许 PPP^① 使用这个连接。实际上，如下步骤由软件自动完成：拨号，启动 SLIRP，然后启动本地计算机上的 PPP。

当 PPP 运行在拨号连接上时，它处理连接的方式与串行线路类似。家用计算机上的 IP 路由选择将拨号连接作为所有通信量的默认路由。处在连接一端的 PPP 把数据报封装在一个 PPP 帧中，而另一端的 PPP 将还原数据报。

由于 SLIRP 理解 PPP 封装，SLIRP 可在拨号连接上发送或接收数据报。此外，SLIRP 含有处理传入 TCP 报文段和处理 IP 数据报所需的全部代码。例如，SLIRP 能够产生或验证 IP 校验和，并且能发送响应 TCP 报文段的确认。实质上，SLIRP 将整个 TCP/IP 协议栈包含在应用程序中了。

19.17 SLIRP 如何处理连接

由于 SLIRP 是一个应用程序，SLIRP 中的 TCP/IP 代码不直接与网关机器上操作系统的 TCP/IP 协议软件交互。SLIRP 只使用它的 TCP/IP 代码，用它来解释在拨号连接上到达的数据报；SLIRP 使用套接字接口与 Internet 通信。

SLIRP 在拨号连接上发生的事件和 Internet 上发生事件间进行映射。例如，假设家用计算机建立了到目的地 D 的 TCP 连接。当报文段从家用计算机到达时，SLIRP 就产生响应结果。这就是说，当家用计算机发送了 SYN，SLIRP 就像 D 在应答那样发出响应（即 SLIRP 扮演了 D）。同时，SLIRP 创建一个套接字并连接到 D。类似地，SLIRP 在家用计算机和 Internet 间传送数据。例如，如果数据通过 TCP 连接从家用计算机到达，SLIRP 就接收数据，并返回 TCP 确认，然后使用套接字接口发送数据到目的地。

19.18 IP 寻址和 SLIRP

SLIRP 与 PPP 的常规使用有何不同呢？答案就在 IP 的寻址，如图 19.5 所示。无论何时，SLIRP 要与 Internet 上的某个主机交互，它都使用套接字接口。因此，SLIRP 的运行与任何其他运行在网关机器上的应用类似。具体来说，在 TCP/IP 互联网上，所有来自 SLIRP 的通信都使用分配给网关机器的 IP 地址；远端目的地不能区分是在与 SLIRP 通信，还是与网关机上任何其他应用程序通信。

与 SLIRP 在 Internet 上使用的寻址方式相比，SLIRP 在拨号连接上使用的寻址方式是非标准的。由于 SLIRP 将拦截来自家用机器的所有数据报，因此家用机器使用的 IP 地址不必是有效的——任何地址都可以使用，只要该地址没有在其他地方使用^①。因此，SLIRP 的用户通常选择类似 10.0.0.1 这样的 IP 地址，这种地址不会分配给一个 Internet 主机，而且很容易记忆。

① 虽然我们在例子中提到了 PPP，但可使用 SLIP 的 SLIRP 版本也是存在的。

② 如果使用了一个有效的 IP 地址，那么家用计算机就不能与拥有该地址的计算机进行通信。

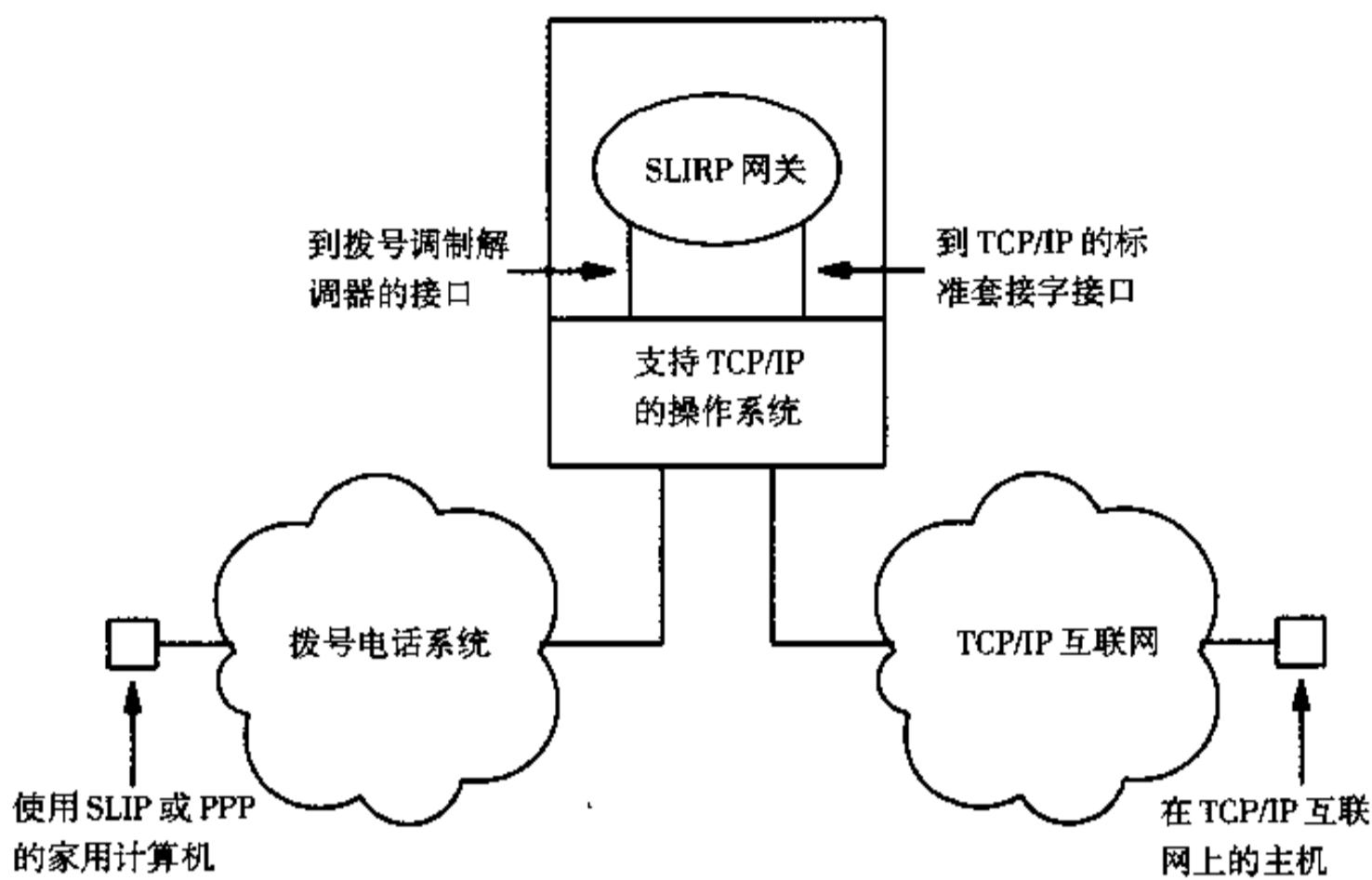


图 19.5 SLIRP 网关的图示。虽然家用计算机可使用任一 IP 地址，但 SLIRP 在与 TCP/IP 互联网上的某个主机通信时，却要使用一个有效的 IP 地址

19.19 小结

虽然隧道技术允许一个协议系统使用另一个高级协议作为传输设备，但它却要求设计人员能访问操作系统代码。应用网关技术是隧道技术的替代方案，它允许应用的编程人员不必改变操作系统而与异构系统互连。

应用网关是一个程序，它使用一个高级协议接受请求，并且使用另一个高层协议处理请求。实质上，每个应用网关是提供某一服务的服务器，也是另一个服务的客户。

我们考察了一个应用网关，它从电子邮件中接受对某个 RFC 的请求，使用 FTP 获取指明的 RFC，并且向使用电子邮件的原请求者返回 RFC。我们的例子说明了应用网关不必是很复杂的，也不必用低级语言编写。我们的例子代码是由命令解释器脚本和 awk 程序组成的。

许多网点使用应用网关实现授权和安全检查。由于网关作为一个常规应用程序运行，要让网关过滤出不受欢迎的接人，或要记录所有的请求，对此，网关的编程几乎不需要费力气。

有一个称为 SLIRP 的网关支持多种服务。SLIRP 是为使用电话拨号线设计的，SLIRP 允许家用计算机接入到 IP 服务，但不需要为家用计算机分配一个有效的 IP 地址。家用计算机运行 PPP 或 SLIP 软件，该软件通过拨号连接发送 IP 数据报。SLIRP 接收到来的数据报，充当前目的地机器来发送确认，并使用常规的 Internet 软件与目的地通信。

深入研究

在操作系统附带的文档中，可以找到更多有关命令解释器和 awk 编程语言的信息。Aho、Kernighan 和 Weinberger [1988] 详细描述了 awk 编程。Bolsky 和 Korn [1989] 描述一个命令解释器

(它是 bash 的先驱), 并举例说明了命令解释器程序。

Linux 程序 sendmail 在各种传输协议上处理电子邮件的传送; 可将它配置成一个应用网关。有关 sendmail 的信息可从联机文档中找到。

邮局协议 POP (Post Office Protocol) 提供了允许个人计算机从服务器访问电子邮件的标准。Myers 和 Rose [RFC 1725] 描述了 POP 协议的第三版 (POP3)。

习题

- 19.1 给本章中应用网关的例子增加授权检查, 从而允许系统管理员限制访问。
- 19.2 给本章中应用网关的例子增加日志机制, 该机制给出请求 RFC 的所有用户的列表。
- 19.3 发送者如何通过本章的应用网关请求一个 RFC 索引 (RFC index) 的副本?
- 19.4 观察发现, 如果用户调用 rfc 脚本请求一个 RFC, 而在同一时刻, 应用网关正使用 rfc 请求同一个 RFC, 那么用户可能获得一个空文件或部分文件。试修改脚本以改正这个缺点。
- 19.5 sendmail 系统如何允许系统管理员建立应用网关?
- 19.6 可以在家用机器上使用 SLIRP 运行客户软件吗? 可使用它运行服务器吗? 为什么能或为什么不能?
- 19.7 用 Perl 脚本语言重写 awk 脚本的例子。哪个程序更小? 哪个更容易阅读?

第 20 章 外部数据表示 (XDR)

20.1 引言

前几章描述了客户和服务器程序的算法、机制和实现技术。本章将着手讨论那些有助于程序员使用客户-服务器模式的概念和技术，以及为这些概念提供编程支持的机制。具体来讲，本章研究了外部数据表示的事实上的标准，以及用于完成数据转换的一组过程库 (library procedure)。

本章描述了外部数据表示 XDR (eXternal Data Representation) 的一般动机，以及一种特殊实现的细节。下一章将说明外部数据表示标准的使用如何会有助于简化客户和服务器的通信，并将说明一个标准如何使下述情况成为可能，即使用单一的、一致的远程访问机制来进行客户-服务器通信。

20.2 数据表示

每种计算机体系结构都有其各自的数据表示定义。即使两种体系结构支持同样的数据类型，但其表示也有可能不同。例如，如果一种体系结构把整数的最高位字节存放在存储器的低端，则称之为大数在前；如果将整数的最低字节存入存储器最低地址，则称之为小数在前；有些则不在两者之列，因为它们不在存储器中连续存储字节。例如，图 20.1 显示了 32 比特整数的大数在前和小数在前两种表示。

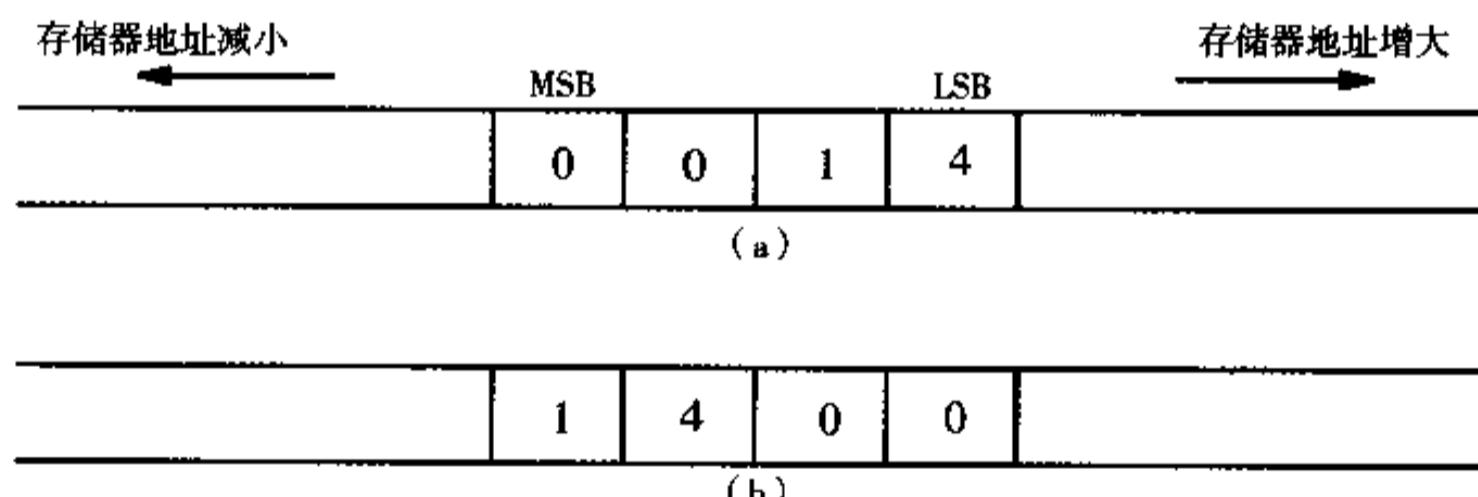


图 20.1 32 比特整数值 260 的两种表示：(a) 最高字节比特在存储器最低地址的小数在前顺序；(b) 最低字节比特在存储器最低地址的大数在前顺序。数字给出每个 8 比特字节的十进制数值

为单台计算机编写程序的程序员不需要考虑数据的表示，因为一台计算机通常只允许有一种表示。程序员声明一个变量为整数（如通过使用 C 的 int 声明）后，当编译程序为整数分配存储空间时，或在它产生读取、存储或比较值的代码时，将使用计算机的本地数据表示。

然而，程序员创建客户和服务器软件时，他们必须考虑数据表示，因为对于在通信双方间通信信道上发送的所有数据，两端必须就这些数据的确切表示达成一致。如果两台机器上的本地数据表

示不同，从一台机器上程序发送给另一端的数据就必须被转换。

20.3 N 平方转换问题

数据表示的中心议题是软件可移植性。一个极端的情况是，程序员可在每一对(pair)客户和服务器中，嵌入计算机体系结构的知识，以便一方将数据转换为适于另一方的表示。如果直接从客户的表示转换成服务器的表示，使用这种模式的设计称为不对称的数据转换，因为转换只是由某一方进行。但遗憾的是，使用不对称的数据转换意味着：对使用的每一对体系结构，程序员都必须为之编写这对客户和服务器的不同版本。

为每对体系结构分别构造单独程序的开销很大，要理解其原因，我们考虑一个N台计算机的集合。如果每台计算机使用不同数据表示存储浮点数，程序员就必须构造总共

$$(N^2-N)/2$$

个版本的客户-服务器程序以交换这些浮点值。我们将这个问题称为N平方转换问题^①，以强调编程劳动量近似与不同数据表示的数目的平方成正比。

看待N平方转换问题的另一种方法是，想像给现有的N台机器集合增加一个新体系所需的劳动。每次增加一台新计算机，在新计算机能与每台现有的计算机进行互操作前，程序员都必须为每对客户-服务器总共构造N个新版本。

即使程序员使用常规的编译器(如C预处理器的ifdef构造)，创建、测试、维护和管理许多程序版本仍很困难。此外，用户调用一个客户时，他可能需要区分不同的版本。总结如下：

如果设计客户-服务器软件时，不对称地在客户的本地数据表示(native data representation)和服务器的本地数据表示之间进行转换，那么软件的版本数将随着体系结构数的平方增长。

20.4 网络标准字节顺序

在维护一个客户-服务器程序的N平方个版本时会出现问题，为避免该固有问题，程序员试图避免不对称的数据转换。他们用某种方式编写客户和服务器软件，以致于每个源程序不需改动就可在不同机器上编译和执行。由于这样做会生成可移植性高的程序，所以使得编程也更容易了。它还使用户访问服务更容易，因为用户只需要记住如何调用客户的一个版本。

如果多种体系结构使用不同的数据表示，单一的一种源程序如何才能在各种不同的体系结构下正确编译和执行呢？更重要的是，如果两台机器使用不同的数据表示，客户或服务器程序如何把数据送到另一端呢？第5章描述了TCP/IP协议如何解决协议首部中的整数表示问题，它是通过使用一组函数来解决这个问题的，这组函数在计算机的本地字节顺序和网络标准字节顺序间进行相互转换。

从本质上讲，在网络中传送的协议首部数据使用了同一种标准数据表示，即协议首部采用了一种标准的、独立于机器的形式来表示整数。我们称这种设计采用的是对称的数据转换(symmetric data conversion)，即不是直接将一种表示转换为另一种，而是每个端点都将其本地表示转换为用于传输的标准表示。对称的数据转换有一个重要的结果就是：只需要一种版本的协议软件。

^① 有些文献将该问题称为n × m问题，以强调客户可以运行在n种体系结构下，而服务器可以运行在m种体系结构下。

大多数程序员在构造客户 - 服务器软件时，都采用对称数据转换技术。客户和服务器都各自完成数据转换，而不是直接从一台机器的表示转换成另一台机器的表示。通过网络发送数据之前，他们将数据从发送方机器的本地表示转换成一个标准的、与机器无关的表示。类似地，在从网络上收到数据后，他们将数据从与机器无关的表示，转换成接收方机器的本地表示。在网络上传送数据所使用的基本表示称为外部数据表示（external data representation）。

使用标准的外部数据标准既有优点，也有缺点。主要优点是灵活性：客户和服务器都不需要理解对方的体系结构。单个客户可与任意机器上的服务器联系，而不必知道该机器的体系结构。总的编程工作量只是近似于机器的体系结构数目，而不是该数目的平方。

对称转换的主要缺点是计算的额外开销。若客户和服务器都在同一体系的计算机上运行，这种开销看来就没有道理了。通信的一端在发送前将所有数据从本地体系的表示转换为外部表示，而另一端在收到后，又将数据从外部表示转换回原来的表示。

即使客户和服务器机器不共享同一体系，使用一种中间形式也将引入额外计算。客户和服务器不是直接将发送方的表示转换成接收方的表示，他们每次都必须耗费 CPU 时间，以便在其本地表示和外部表示之间进行转换。此外，由于外部表示可能需要给数据增加信息，或者要按字对齐数据，这种转换产生的字节流可能会比所需要的字节流更大。

尽管需要额外开销和网络带宽，大多数程序员都认为对称转换的使用是值得的。它简化了编程，减少了错误，并增加了程序间的互操作性。它也使得网络管理和调试更加容易，因为网络管理员不需要知道发送方和接收方机器的体系结构就能解释分组的内容。

20.5 外部数据表示的事实上的标准

Sun Microsystems 公司设计了一种外部数据表示，它规定了在网络上传输数据时如何表示成公共形式的数据。Sun 的外部数据表示 XDR（eXternal Data Representation）已成为大多数客户 - 服务器应用的一个事实上的标准。

XDR 规定了客户和服务器所交换的大多数数据类型的格式。例如，XDR 规定 32 比特二进制整数用小数点前顺序表示（即最高字节比特于存储器最低地址）。

20.6 XDR 数据类型

图 20.2 中的表格列出了 XDR 定义的数据类型的标准表示。

图 20.2 中的类型覆盖了能在应用程序找到的大多数数据结构，这是因为可以允许程序员用其他类型组成复合类型。例如，除了允许整数数组外，XDR 还允许结构数组，且每个结构可以有多个字段，而每个字段可能是一个数组、结构或联合。因此，XDR 可为 C 程序员能指明的大多数结构提供表示。

数据类型	大小	描述
int	32 比特	32 比特二进制带符号整数
unsigned int	32 比特	32 比特二进制无符号整数
bool	32 比特	布尔值 (false 或 true) 用 0 或 1 表示
enum	任意	枚举类型，其值定义成整数 (如 RED=1, WHITE=2, BLUE=3)
hyper	64 比特	64 比特二进制带符号整数
unsigned hyper	64 比特	64 比特二进制无符号整数
float	32 比特	单精度浮点数
double	64 比特	双精度浮点数
opaque	任意	不转换数据 (如用发送方本地表示的数据)
string	任意	ASCII 字符串
fixed array	任意	任何其他数据类型的定长数组
counted array	任意	数组中的类型有一个固定上限，但各个数组的上限大小不同
structure	任意	数据的聚集，像 C 的 struct
discriminated union	任意	一种数据结构，允许在几种形式中选择一种的数据结构，像 C 的 union 或 Pascal 的可变记录
void	0	如果数据项是可选的，它又没有给出数据，就使用该类型
symbolic constant	任意	一个符号常量及相关值
optional data	任意	允许某一项出现零次或一次

图 20.2 由 XDR 定义的外部表示的类型。该标准规定了每个类型的数据项在网络上传送时应如何编码

20.7 隐含类型

对于图 20.2 中列举的各个数据类型，XDR 标准规定了一个数据对象应如何编码。然而，编码只含有数据项，但并没有包含有关数据项类型的信息。例如，XDR 规定 32 比特二进制整数使用小数在前顺序 (在 TCP/IP 协议首部中使用同样的编码)。如果一个应用程序用 XDR 表示对一个 32 比特整数进行编码，结果将恰好占用 32 比特；编码中不含其他比特以识别它是一个整数，或者指明其长度。因此，客户和服务器在使用 XDR 时，必须就其交换的报文的确切格式达成一致。一个程序只有知道确切格式和所有数据字段的类型，才能解释一个用 XDR 编码的报文。

20.8 使用 XDR 的软件支持

程序员若选用 XDR 表示来用于对称数据转换，在向网络发送数据前，必须小心地将每个数据项放入外部形式中。类似地，接收方程序也必须小心处理，以便将每个收到的数据项转换成本地表示。第 5 章说明了一种可用于完成转换的编程方法：在代码中插入一个函数调用，它在发送前将报文中每个数据项转换为外部形式，在接收报文时，还要插入一个函数调用，负责将每个数据项转换成内部形式。

大多数程序员不需花多少工夫就能编写所需的 XDR 转换程序。但是，一些转换需要有相当的经验 (如，从计算机的本地浮点表示转换成 XDR 标准，且不丢失精确度，这可能需要了解基本的数字分析方法)。为减少潜在的转换错误，XDR 的实现通常包含完成必要转换的库例程。

20.9 XDR 库例程

某台机器的 XDR 例程库 (library routine) 可完成数据项的转换，将数据从计算机的本地表示转换成 XDR 标准表示，或者反之。XDR 的大多数实现使用了一种缓存范例 (buffer paradigm)，它允许程序员创建完全是 XDR 形式的报文。

20.10 一次一片地构造报文

使用缓存范例的 XDR 需要程序分配足够大的缓存，以便存放一个报文的外部表示，并且需要程序一次往缓存中添加一个数据项（即字段）。例如，在 Linux 操作系统下给出的 XDR 版本提供了转换例程，每个例程都在存储器缓存的末尾追加外部表示。程序首先调用 `xdrmem_create` 过程，在存储器中分配一个缓存，并且通知 XDR 它将在该缓存中组成外部表示。`xdrmem_create` 初始化存储器缓存，于是，该缓存表示用于编码（转换成标准表示）或解码（转换成本地表示）数据的 XDR 流。该调用通过指定缓存的起始地址为内部指针，将 XDR 流初始化为空（empty）。`xdrmem_create` 返回 XDR 流的指针，而该指针必须被用于对 XDR 例程的后续调用。使用 C 创建 XDR 流的声明和调用是：

```
#include <rpc/xdr.h>
#define BUFSIZE 4000          /* size of memory for encoding */

XDR      *xdrs;           /* pointer to an XDR "stream" */
char     buf[BUFSIZE]     /* memory area to hold XDR data */

xdrmem_create(xdrs, buf, BUFSIZE, XDR_ENCODE);
```

一旦程序创建了 XDR 流，它就可以调用各个 XDR 转换例程，以便将本地数据对象转换成外部形式。每个调用编码一个数据对象，并将编码信息追加到流的末尾（即在缓存的下一个可用比特位置存放编码后的数据，然后更新内部流指针）。例如，`xdr_int` 过程将 32 比特二进制整数从本地表示转换成 XDR 表示，并且将编码信息追加到 XDR 流末尾。程序调用 `xdr_int`，需要给过程传递两个参数：XDR 流的指针和整数指针。

```
int      i;                /* integer in native representation      */
. . .
i = 260;                  /* assume stream initialized for ENCODE */
xdr_int(xdrs, &i);        /* assign integer value to be converted */
                           /* convert integer and append to stream */
```

图 20.3 展示了上例代码中所示的 `xdr_int` 调用的执行情况，它说明该过程如何给 XDR 流增加 4 个字节的数据。

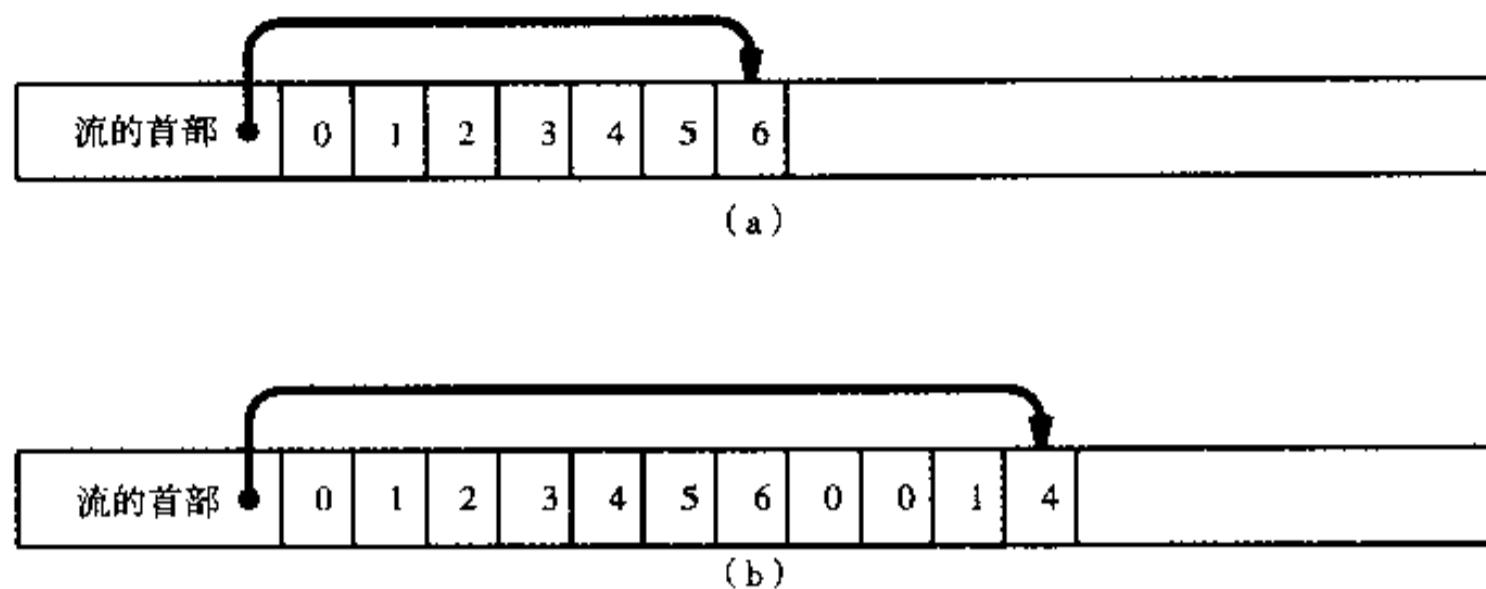


图 20.3 (a) 已被初始化用于编码的 XDR 流，并且已含有 7 个字节的数据；(b) 调用 `xdr_int` 后的同一个 XDR 流，它已被添加了值为 260 的 32 比特整数

20.11 XDR 库中的转换例程

图 20.4 中的表列出了 XDR 转换例程。

过程	参数	被转换的数据类型
<code>xdr_bool</code>	<code>xdrs, ptrbool</code>	布尔型 (C 中的 <code>int</code>)
<code>xdr_bytes</code>	<code>xdrs, ptrstr, strsize, maxsize</code>	计数的字节串
<code>xdr_char</code>	<code>xdrs, ptrchar</code>	字符
<code>xdr_double</code>	<code>xdrs, ptdouble</code>	双精度浮点
<code>xdr_enum</code>	<code>xdrs, ptrint</code>	枚举数据类型的变量 (C 中的一个 <code>int</code>)
<code>xdr_float</code>	<code>xdrs, ptrfloat</code>	单精度浮点
<code>xdr_int</code>	<code>xdrs, ip</code>	32 比特整数
<code>xdr_long</code>	<code>xdrs, ptrlong</code>	64 比特整数
<code>xdr_opaque</code>	<code>xdrs, ptrchar, count</code>	已发送的未转换的字节
<code>xdr_pointer</code>	<code>xdrs, ptrobj, objsize, xdrobj</code>	指针 (用于像表或树这样链接的数据结构)
<code>xdr_short</code>	<code>xdrs, ptrshort</code>	16 比特整数
<code>xdr_string</code>	<code>xdrs, ptrstr, maxsize</code>	一个 C 字符串
<code>xdr_u_char</code>	<code>xdrs, ptruchar</code>	无符号的 8 比特整数
<code>xdr_u_int</code>	<code>xdrs, ptrint</code>	无符号的 32 比特整数
<code>xdr_u_long</code>	<code>xdrs, ptrulong</code>	无符号的 64 比特整数
<code>xdr_u_short</code>	<code>xdrs, ptrushort</code>	无符号的 16 比特整数
<code>xdr_u_union</code>	<code>xdrs, ptrdiscrim, ptrunion, choicefn, default</code>	判别联合
<code>xdr_vector</code>	<code>xdrs, ptrarray, size, elemsize, elemproc</code>	固定长度数组
<code>xdr_void</code>	无	不是一种转换 (用于表示数据结构的空的部分)

图 20.4 在 XDR 库中找到的 XDR 数据转换例程。由于例程的大多数参数都是数据对象的指针而不是数据值，因而例程在两个方向上都进行转换

为形成一个报文，应用程序要为报文中的每个数据项调用 XDR 转换例程。对每个数据项编码并添加到 XDR 流后，应用程序可通过发送最终生成的 XDR 流来发送报文。接收方应用程序必须颠倒整个过程。它调用 `xdrmem_create` 创建一个存放 XDR 流的存储器缓存，并且将传入报文放进缓存。XDR 自己在例程访问的流中记录转换的方向，转换例程可以访问到流中的这个记录。接收方在创建用于输入 XDR 流时，它指明 `XDR_DECODE` 为 `xdrmem_create` 的第三个参数。结果是，只要收方在

输入流上调用了转换例程，该例程就从流中取出一项并转换为本地模式。例如，如果接收方已建立一个 XDR 用于输入的流（即调用指明了 XDR_DECODE），它就能通过调用 `xdr_int`，从输入流中取出一个 32 比特的整数，并将该整数转换为本地表示：

```
int i; /* 使用本地表示的整数 */
...
/* 假定流已被初始化用于 DECODE */
xdr_int(xdrs, &i); /* 从流中提取整数 */
```

因此，不像 Linux 所提供的 `htonl` 和 `ntohl` 转换例程，各个 XDR 转换例程不指定转换的方向。然而，它们需要程序在创建 XDR 流时指明方向。总之：

各个 XDR 转换例程不指明转换的方向。然而，可进行双向转换的单个例程要通过检查所使用的 XDR 流来判断转换的方向。

20.12 XDR 流、I/O 和 TCP

以上代码创建了与存储器缓存相关联的 XDR 流。使用存储器缓冲数据可使程序更有效，因为它允许应用程序在向网络发送数据前，将大量数据转换成外部形式。在各个数据项都被转换成外部形式并放入缓存后，应用程序必须调用像 `write` 这样的 I/O 函数，以便在 TCP/IP 连接上将其发送出去。

可以让 XDR 转换例程在每次把数据项转换为外部形式后，自动地将数据在 TCP 连接上发送。为此，应用程序首先创建一个 TCP 套接字，然后调用 `fdopen` 将标准 I/O 流附于此套接字上。应用程序不是调用 `xdrmem_create`，而是调用 `xdrstdio_create` 创建 XDR 流，并将它连接到现有的 I/O 描述符上。附在 TCP 套接字上的 XDR 流不需要显式调用 `write` 或 `read`。应用程序每次调用 XDR 转换例程，转换例程就使用它所蕴涵的描述符，自动完成带缓存的 `read` 或 `write` 操作。`write` 操作使 TCP 将外发数据发送到套接字；而 `read` 操作将使 TCP 从套接字读取传入数据。应用程序还可以调用 UNIX 标准 I/O 库中的常规函数，以便对 I/O 流进行操作。例如，如果期望输出，应用程序可在仅转换了少数几个字节的数据后，就使用 `fflush` 刷新输出缓存。

20.13 记录、记录边界和数据报 I/O

正如前面所描述的那样，当 XDR 流连接到 TCP 套接字时，XDR 机制可以很好地工作，因为 XDR 和 TCP 的实现都使用流抽象（stream abstraction）。为使 XDR 与 UDP 的合作像它与 TCP 合作一样容易，设计人员增加了第二个接口。这种变通的设计为应用程序提供了面向记录的接口（record-oriented interface）。

为了使用面向记录的接口，程序创建 XDR 流时要调用 `xdrrec_create` 函数。该调用包括两个参数，`inproc` 和 `outproc`，它们分别定义了输入过程和输出过程。在把数据转换成外部形式时，每个转换例程都检查缓存。如果缓存满了，转换例程就调用 `outproc` 发送缓存中已有的内容，为新数据腾出空间。类似地，每次应用程序调用转换例程将外部形式转换成本地表示时，例程将检查缓存是否含有数据。如果缓存是空的，转换例程就调用 `inproc` 获取更多数据。

为使 XDR 结合 UDP 使用，应用程序需创建面向记录的 XDR 流。它让与流相关联的输入和输出过程调用 `read` 和 `write`（或 `recv` 和 `send`）。当应用程序填满缓存时，转换例程就调用 `write` 用单个 UDP 数据报传输缓存内容。类似地，当应用调用转换例程取出数据时，转换例程将调用 `read` 获取下一个传入数据报，并将它放进缓存中。

通过 `xdrrec_create` 创建的 XDR 流，在几个方面不同于其他 XDR 流。面向记录的流允许应用程序标志记录边界。此外，发送方可指明是立刻发送记录，还是等缓存满后再发送。接收方可检测记录边界，跳过输入中固定数目的记录，或查看是否已收到其他记录。

20.14 小结

由于计算机不使用一种公共的数据表示，客户和服务器程序必须设法解决表示问题。为解决这个问题，客户-服务器的交互可以是不对称的或对称的。不对称的转换需要客户或服务器在自己的表示与对方机器的本地表示之间进行转换。对称转换使用一种标准网络表示，并需要客户和服务器在网络标准和本地表示之间进行转换。

不对称交互的主要问题是每个程序需要有多个版本。如果网络支持 N 种体系结构，则不对称交互的编程工作量与 N^2 成正比。而对称设计可能只需要稍多一些的额外计算开销，因为每种体系结构只用一种程序就可提供互操作性。鉴于对称转换所需的编程工作量与 N 成正比，因此大多数设计人员选择对称方案。

Sun Microsystems 公司定义了一种外部数据表示，它已成为事实上的标准。此标准被记为 XDR，它为数据聚合体（data aggregate）（如数组和结构）和基本数据类型（如整数和字符串）提供定义。XDR 库例程提供从计算机的本地数据表示到外部标准的转换，以及反方向的转换。客户和服务器程序可使用 XDR 例程，在发送前将数据转换成外部形式，并在收到数据后将数据转换为内部形式。转换例程可与使用 TCP 或 UDP 的输入或输出相关联。

深入研究

Sun Microsystems 公司 [RFC 1014] 定义了 XDR 编码和标准 XDR 转换例程。Srinivasan [RFC 1832] 含有更新一些的版本，它是一个草案标准。其他信息可在 Linux 操作系统所带的联机文档中找到。

国际标准化组织 ISO [1987a 和 1987b] 定义了另一种外部数据表示，称为抽象语法记法 1 (Abstract Syntax Notation One，即 ASN.1)。虽然 TCP/IP 协议族中一些协议使用 ASN.1 表示，但大多数应用程序还是使用 XDR。Partridge 和 Rose [1989] 说明了 XDR 和 ASN.1 具有相同表示能力。

Padlipsky [1983] 讨论了不对称的问题，并指出它可能需要 $n \times m$ 个可能的转换。

习题

20.1 编写一个 C 程序，确定本地计算机的整数字节序。

20.2 试构造 `ntohs` 的一个版本，并做实验，比较你的程序版本的执行时间与你的系统库或 `include` 文件中的程序版本的执行时间。

- 20.3 XDR 使用缓存范例如何使得编程更容易？
- 20.4 试设计一个外部数据表示，它的每个数据项之前包含一个类型字段。这种方案的主要优点是什么？主要缺点是什么？
- 20.5 阅读 Linux 操作系统所带的联机文档，找出更多有关 XDR 流的格式信息。在 XDR 流的首部中保留了什么信息？
- 20.6 如果 XDR 的设计者这样选择方案，即，为编码和解码分别使用转换例程，而不是在流的首部中记录转换方向，证明这种程序将更容易阅读。分开保留转换例程的缺点是什么？
- 20.7 在什么情况下，程序员需要在客户和服务器间传递 opaque 数据对象？
- 20.8 解释为什么浮点数的数据转换特别困难。

第 21 章 远程过程调用（RPC）的概念

21.1 引言

在前一章，我们开始对一些技术和机制进行讨论，这些技术和机制有助于程序员使用客户 - 服务器范例，我们还讨论了使用对称数据转换的优点，以及 XDR 外部数据表示标准及其相关的库例程是如何提供对称转换的。

本章将继续讨论中间件（即程序员用于构造客户 - 服务器软件的工具和库），引入了远程过程调用 RPC（Remote Procedure Call）的概念，还将描述一个远程过程调用的特定的实现，该实现的数据表示使用了 XDR 标准。本章将说明这一方法如何简化了客户 - 服务器软件设计，并且使最终的程序易于理解。下两章将描述一个工具，该工具可以为使用远程调用的程序生成它所需要的大部分 C 代码，本书将以此来完成对远程过程调用的讨论。在这两章里还有一个完整的可运行的例子，该例子说明，在构建客户和服务器时，如何使用工具减少大部分编写例程的工作量。

21.2 远程过程调用模型

到目前为止，我们在描述客户 - 服务器程序时，只是分别查看了客户和服务器的各个构件（component）的结构。然而，在程序员构建客户 - 服务器软件时，他们不能一次只关注于其中的一个构件。他们必须要考虑整个系统是如何工作的，以及这两个构件将如何交互。

为帮助程序员设计和理解客户 - 服务器的交互，研究人员业已为构建分布式程序设计出一套概念性框架。该框架称为远程过程调用模型（remote procedure call model）或 RPC 模型，它把我们所熟悉的来自常规程序的概念作为设计分布式应用的基础。

21.3 构建分布式程序的两种模式

在设计分布式应用时，程序员可以使用下列两种方法之一：

- **面向通信的设计**

由通信协议开始。设计报文格式和语法，指明对每个传入报文将如何反应以及如何产生每个外发报文，以此来设计客户和服务器各构件。

- **面向应用的设计**

由应用开始。设计常规的应用程序来解决问题。构建并测试可在单台机器上运行的常规程序的工作版本。将这个程序划分成两个或多个程序片，加入通信协议以允许每片程序在单独的计算机上执行。

面向通信的设计有时会带来问题，首先，由于程序员关注于通信协议，他们可能会错过应用中的一些重要的细节，还可能会发现协议并没有提供所有需要的功能。第二，因为很少有程序员具有协议设计方面的经验并精于此道，所以他们往往会展现出笨拙的、不正确的或者是缺乏效率的协议。在协议设计中的微小疏忽可能会导致十分重要的错误，这些错误会隐藏起来，一直到程序运行在某种压力下时才爆发（比如，死锁的可能性）。第三，由于程序员集中精力于通信，这往往使通信成为最终程序的中心部分，这就使程序难于理解和修改。特别是，因为服务器是通过给出一系列报文以及每个报文到达时所要采取的反应来指明的，这样，要理解获得某种特定结果所需要的一系列报文或某种特定报文所蕴涵的动机就很困难了。

远程过程调用模型使用面向应用的方法，它强调的是所要解决的问题而不是所需要的通信。利用远程过程调用，程序员首先设计一个解决问题的常规程序，接着将其划分成若干片，这些程序片运行在两台或更多的计算机上。程序员可遵循良好的设计原则，以便使代码模块化并且可维护。

在理想的情况下，远程过程调用提供的不只是抽象的概念。它允许程序员在将一个程序划分成若干片之前（每个报文运行在各个独立机器之上），先构建、编译和测试一个解决该问题的常规程序的版本，以便确保能够正确解决问题。不但如此，因为 RPC 以过程为边界（即方法调用）划分程序，所以将程序划分为本地部分和远程部分并不会引起程序结构的很大变化。实际上，将某些过程从一个程序转移到远程机器上时，有可能不需要改变，甚至不需要重新编译主程序本身。因此，RPC 把解决问题本身与使这个解决方法运作的任务在分布式环境下分开了。

对编程来说，远程过程调用模型关注于应用。它允许程序员在将一个程序划分成可以运行于多台计算机上的片段之前，把精力集中在设计一个解决问题的常规程序上。

21.4 常规过程调用的概念性模型

远程过程调用模型主要来自于传统编程语言中的过程调用机制。过程调用提供了一个强有力的抽象，它允许程序员将一个程序划分成一些小的、可管理的、易于理解的片段。过程特别有用，因为它具有一个能给出程序执行的概念性模型的简单明了的实现。图 21.1 说明了这个概念。

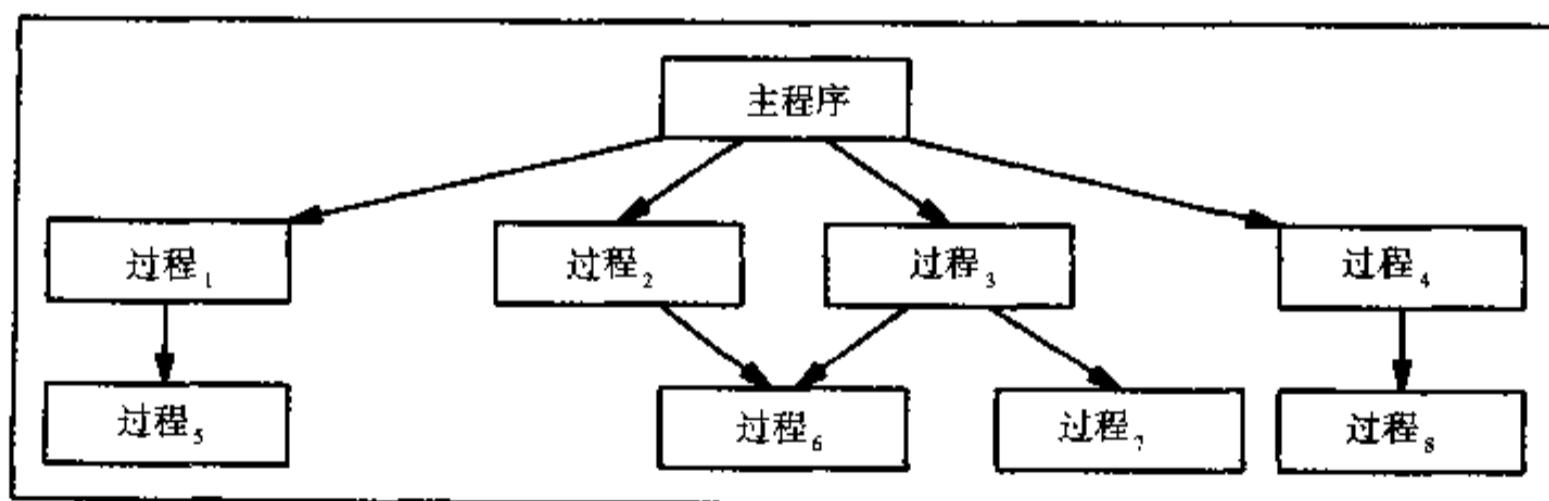


图 21.1 过程概念。一个常规程序由一个或多个过程组成，它们往往按一种调用等级来安排。从过程 n 到过程 m 的箭头代表由从 n 到 m 的调用

21.5 过程模型的扩充

远程过程调用模型使用了和常规程序一样的过程抽象，只是它允许一个过程的边界跨越两台计

计算机。图 21.2 展示了远程过程调用模式如何将一个程序划分成两片，每片在一台单独的计算机上执行。当然，常规的过程调用不能由一台计算机传递到另一台上。在程序可以使用远程过程调用之前，必须加入允许程序与远程过程通信的协议软件。

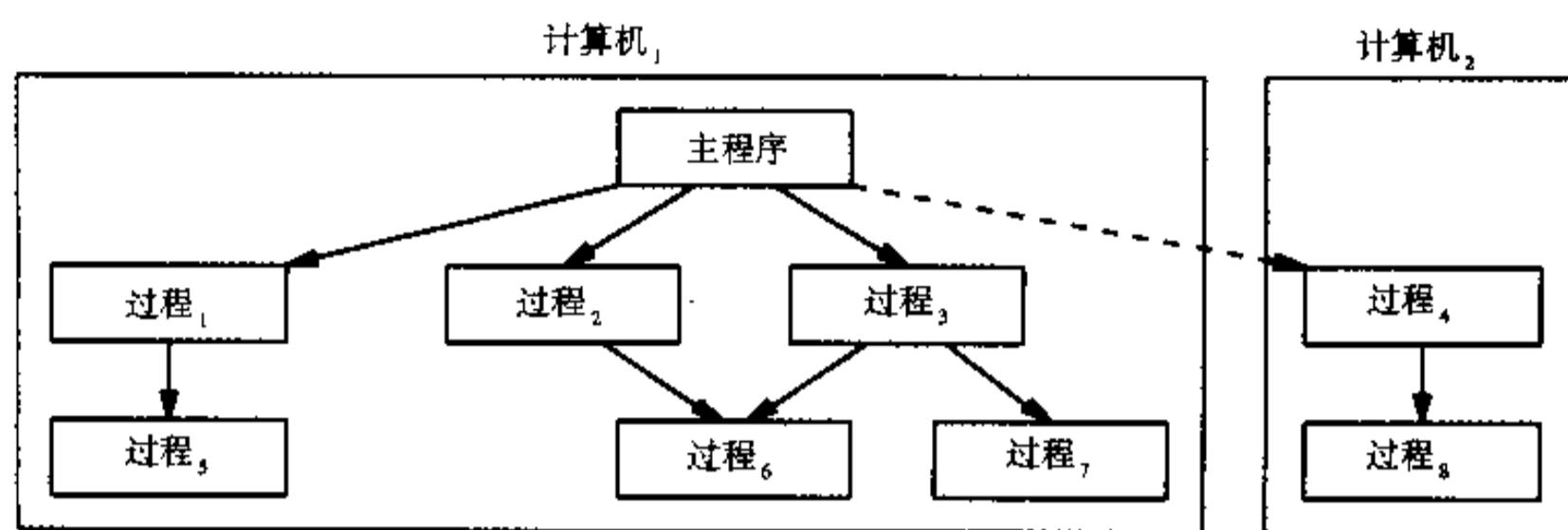


图 21.2 一个分布式的程序，它说明了怎样将图 21.1 所示的程序扩展成使用远程过程调用。划分发生在主程序和过程 4 之间。实现远程过程调用要求有一个通信协议

21.6 常规过程调用的执行和返回

程序的过程模型提供了程序执行的概念性解释，它可以直接扩展到远程过程调用。让我们考虑一下内存中那些已编译好的程序代码的控制流 (control flow) 的关系，这是理解这个概念的最好方法。例如，图 21.3 展示了控制流从主程序传到两个过程，然后返回。

根据过程执行模型，单个控制线索 (thread of control) 或执行线索 (thread of execution) 流经所有过程。计算机从一个主 (main) 程序开始执行，并一直继续下去，直到遇到一个过程调用。这个调用使执行转入到某个指定的过程代码并继续执行。如果遇到了另一个调用，计算机便转入到第二个调用。

程序继续在所调用的过程中执行，直到遇到了一个返回 (return) 语句。这个返回语句使程序恢复到紧挨着最后一个调用之后的那点。例如，图 21.3 执行过程 B 中的返回语句使控制权传递回了过程 A 中紧接着对 B 调用后的那一点。

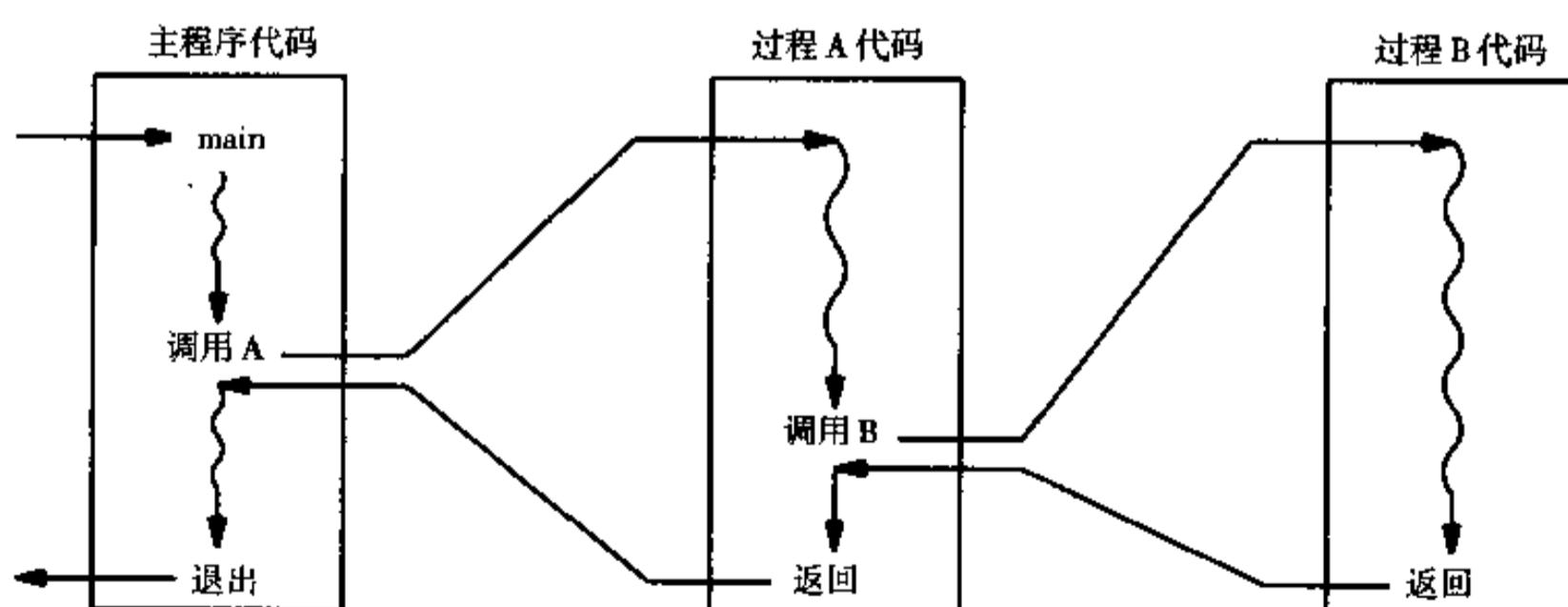


图 21.3 程序执行的概念性模型。它解释了在过程调用和返回时的控制权的流动。单个控制线索由主程序开始，接着传递给过程 A 和 B，最后返回给主程序

在概念模型中，任何时候都只有一个执行线索在继续。因此，在计算机执行对一个过程的调用时，另一个过程必须暂时停止。计算机挂起调用进程，在调用过程中，将这个过程的所有变量值全部冻结。之后，当执行由过程调用返回时，计算机在调用者中恢复执行，而且所有变量的值又可以使用了。被调用过程可以进一步使用过程调用，因为计算机记住了调用的次序，而且总是返回最近执行的调用者。

21.7 分布式系统中的过程模型

通过思考常规的计算机程序，程序员所使用的过程调用的执行模型可以帮助我们理解分布式系统中程序是如何进行的。我们不是考虑客户程序和服务器程序交换消息，而是想像每个服务器实现了一个（远程的）过程，而且，客户和服务器之间的交互对应于过程的调用和返回。由客户发送给服务器的请求对应于对远程过程的调用，而由服务器发回给客户的响应对应于返回指令的执行。图 21.4 展示了这种比喻。

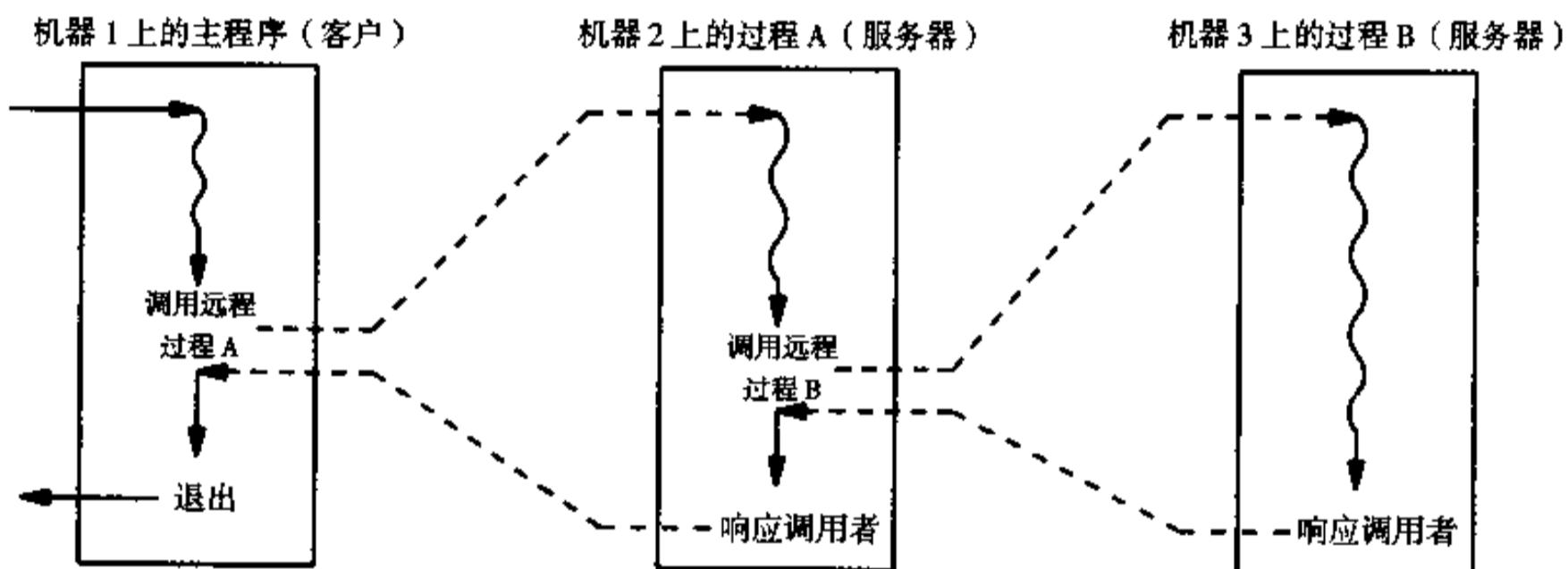


图 21.4 使用远程过程调用的执行模型。单个控制线索在分布式环境下执行。虚线说明了在远程过程调用中，控制权是如何由客户传递给服务器的，以及它又是如何在服务器响应时传递回给客户的。

21.8 客户 - 服务器和 RPC 之间的类比

远程过程调用概念提供了强有力的类比方法，它允许程序员以一种他所熟悉的环境来思考客户和服务器的交互。如同常规的过程调用，远程过程调用把控制权传递给被调用的过程。也像常规的过程调用一样，在调用进行中，系统把调用过程的执行挂起，而只允许被调用过程执行。

当远程程序发出响应时，这对应于在常规过程调用中执行 return。控制权返回给调用者，被调用过程停止执行。嵌套的过程调用的想法也可应用到远程过程调用。远程过程也许要调用另一个远程过程。如图 21.4 所展示的，嵌套的远程过程调用对应于这种情况，即服务器成了另一个服务的客户。

当然，远程过程调用和客户 - 服务器之间交互的这种类比并没有解释所有的细节。例如，我们知道，常规的过程会一直保持在完全不活动 (inactive) 状态，直到控制流传递给了它（即直到它被调用）。与之相反，远程系统中必须要有一个服务器进程，并且在接收到来自某个客户的第一请

求之前, 它一直在那里等待着计算某个响应。更大的不同来自于数据流向远程过程的方式。常规的过程往往接受很少几个参数, 而且仅仅返回很少几个结果。然而, 服务器可以接受或返回任意数量的数据 (即它可以在 TCP 连接上接受和返回任意的流)。

如果本地和远程过程调用的行为是一致的, 这样当然很理想, 但是, 许多实际的约束却使它们不能这样。首先, 网络的延时会使远程过程调用的开销比常规过程调用昂贵几个数量级。第二, 由于被调用的过程与调用过程运行在同一个地址空间上, 所以常规程序可以把指针作为参数来传递。远程过程调用不能把指针作为参数, 因为远程过程与调用者运行在完全不同的地址空间中。第三, 因为远程过程不能共享调用者的环境, 它就不能直接访问调用者的 I/O 描述符或操作系统功能。例如, 远程过程就不能在调用者的标准差错文件中直接写入差错消息。

21.9 作为程序的分布式计算

对远程过程调用进行正确评判的关键在于理解到下面的情况, 即尽管存在着一些实际上的限制, 但该范例有助于程序员更容易地设计分布式程序。为了解其中的原因, 设想每个分布式计算由运行在分布式环境下的程序构成。如果程序在需要访问某个远程服务时, 只是简单地调用一个过程, 而不再是考虑实现通信的客户和服务器软件, 这样, 构建分布式程序将会是多么简单。想像一下这个过程: 程序的执行线索可能会穿过网络传递给远程机器, 执行那台机器上的远程过程, 接着便返回调用者。从程序员的观点看, 对远程服务的访问就如同访问本地的过程或本地的操作系统服务那样容易。简单地说就是, 构造分布式程序将变得同构造传统的程序一样容易, 因为它们可以建筑于程序员的直觉和使用常规过程调用的经验之上。更进一步来说, 熟悉过程参数机制的程序员可以明确地定义客户-服务器的通信, 而不需要任何特殊的记号或语言。概括起来就是:

把分布式计算看作是单个程序, 这个程序穿过网络把控制权传递给一个远程过程, 在其完成后将控制权返回, 这种方式有助于程序员指明客户-服务器的交互; 它使分布式计算的交互同我们所熟悉的过程调用及返回的表示方法联系起来。

21.10 Sun Microsystems 的远程过程调用定义

Sun Microsystems 公司定义了一个特定形式的远程过程调用。它称为 Sun RPC, 开放网络计算 (Open Network Computing, ONC) RPC 或简称为 RPC^①。ONC 远程过程定义业已被业界广泛接受。它已被用做许多应用软件的实现机制, 其中包括网络文件系统 (Network File System, NFS)^②。

ONC RPC 定义了调用者 (客户) 发出的调用服务器中某个远程过程的报文的格式、参数的格式以及被调用过程返回给调用者的结果的格式。它允许调用程序使用 UDP 或 TCP 来装载报文, 它利用 XDR 来表示过程的参数以及 RPC 报文首部中的其他条目。最后, 除了协议说明之外, ONC RPC 还包括一个编译系统, 能帮助程序员自动构建分布式程序。

① 在本书其余的内容中, 除非特别说明, 术语 RPC 将指 ONC RPC。

② 第 24 章将详细讨论 NFS。

21.11 远程程序和过程

ONC RPC 通过定义远程执行环境而扩展了远程过程调用模型。它定义了一个远程程序，把它作为在远程机器上执行的软件的基本单元。每个远程程序对应于我们所设想的服务器，它包括一组（一个或多个）远程过程以及全局数据。在远程程序中的所有过程都能共享它的全局数据。因此，一组密切合作的远程过程可以共享状态信息。例如，要想实现远程数据库，可以构造远程程序，这个程序包含装载共享信息的数据结构，还有三个维护这些信息的远程过程：insert、delete 和 lookup。如图 21.5 所示，远程程序中的所有过程可以共享访问一个数据库。

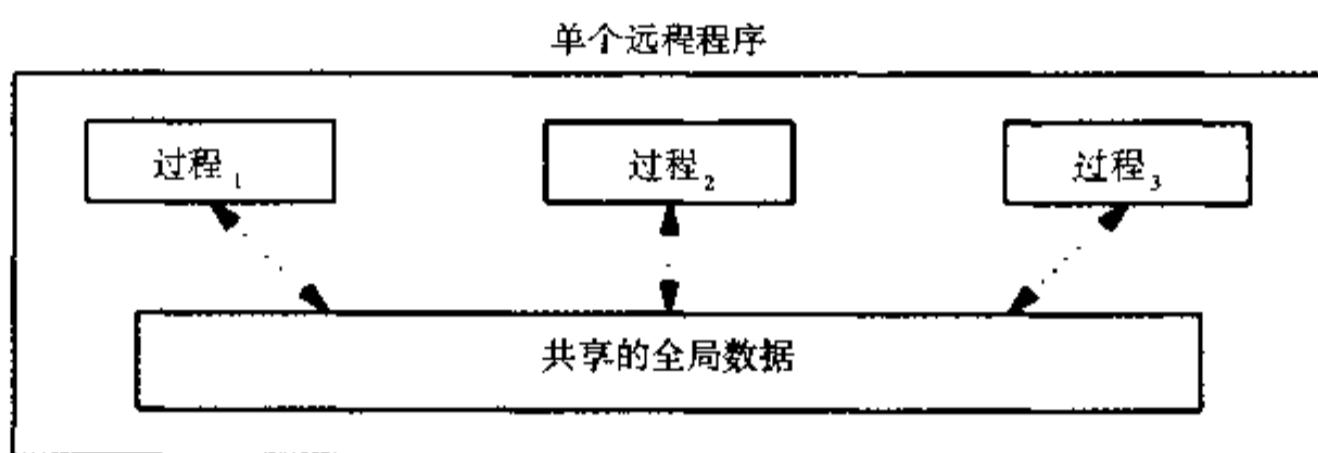


图 21.5 一个远程程序中三个远程过程的概念性组织。所有这三个过程共享访问程序中的全局数据，这就像常规过程共享访问传统程序中的全局数据

21.12 减少参数的数量

因为多数编程语言使用位置记法（positional notation）表示参数，所以含有众多参数的过程调用可能难于阅读。程序员可以把许多参数收集到数据聚合体中（例如，一个 C 结构），并且把结果聚合体作为单个参数来传递，通过这种方法可以解决参数过多的问题。在调用者将结构传递给被调用过程前，它为结构中的每个字段赋予一个值；在调用返回后，调用者再从结构中提取返回值。概括起来就是：

使用结构来代替多个参数可使程序更加可读，因为结构字段的名字可以作为关键字来使用，它告诉读者每个参数如何使用。

因为我们将假设在我们这个讨论的其余部分里，所有使用 RPC 的程序都把它的参数收集到一个结构中，所以每个远程过程将只需要一个参数。

21.13 标识远程程序和过程

ONC RPC 的标准说明，在某台计算机上执行的每个远程程序都必须分配一个惟一的 32 比特整数，调用者使用该整数来标识这个程序。此外，RPC 为一个给定的远程程序中的每个远程过程分配了一个整数标识符。这些过程按序计数：1, 2, …, N^①。从概念上说，在一台给定远程计算机上的某个指定远程过程可以由一个数对来标识：

^① 按照惯例，数目 0 永远保留作为回送过程（echo procedure），用来测试该远程程序是否能被调用。

{ prog, proc }

在此, prog 标识远程程序, proc 标识程序中的某个远程过程。为确保由各个单独的组织所定义的程序号不会冲突, ONC RPC 已将程序号的集合划分成八组, 如图 21.6 所示。

从	至	由谁指派
0x00000000	-	0x1fffffff Sun Microsystems 公司
0x20000000	-	0x3fffffff 在用户网点上的系统管理员
0x40000000	-	0x5fffffff 过渡的(暂时的)
0x60000000	-	0x7fffffff 保留
0x80000000	-	0x9fffffff 保留
0xa0000000	-	0xbfffffff 保留
0xc0000000	-	0xdfffffff 保留
0xe0000000	-	0xfffffff保留

图 21.6 划分成八个组的 32 比特数, 它被 ONC RPC 用来标识远程程序。每个远程程序被分配了一个唯一的号

Sun Microsystems 公司管理第一组标识符, 它允许任何人申请一个标准的 RPC 程序号。因为 Sun 公布了这种分配, 所有使用 RPC 的计算机都使用标准的值。在第一组中, 有 229 个程序号可供使用。Sun 只分配了其中的一小部分。图 21.7 对一些已分配了的号进行了小结。

名字	指派的号	说明
portmap	100000	端口映射器
rstatd	100001	rstat、rup 和 perfmeter
rusersd	100002	远程用户
nfs	100003	网络文件系统
ypserv	100004	yp (现称为 NIS)
mountd	100005	mount, showmount
dbxd	100006	DBXprog (排错程序)
ypbind	100007	NIS 绑定程序
walld	100008	rwall, shutdown
yppasswd	100009	yppasswd
etherstatd	100010	以太网统计
rquotad	100011	rquotaprof, quota, rquota
sprayd	100012	spray
selection_svc	100015	选择服务
dbsessionmgr	100016	unify, netdbms, dbms
rexid	100017	rex, remote_exec
office_auto	100018	alice
lockd	100020	klmprog
lockd	100021	nlmprog
statd	100024	状态监视器
bootparamd	100026	bootstrap
penfsd	150001	对 PC 的 NFS

图 21.7 当前由 Sun Microsystems 公司分配 RPC 程序号的例子

21.14 适应远程程序的多个版本

除了程序号之外，ONC RPC 还包括针对每个远程程序的整数型的版本号（version number）。程序的第一个版本往往分配给版本号 1。以后的每一个版本都收到一个惟一的版本号。

版本号提供了一种能力，它可以使程序不必获得新的程序号就能改变某个远程过程的细节。在实际当中，每个RPC报文通过一个三元组来标识某台给定计算机上的所期望的接收者，这个三元组是：

(prog, vers, proc)

在此，prog 标识远程程序，vers 指明报文所要发往的程序的版本号，proc 指明该远程程序中的某个远程过程。RPC 的规约允许计算机同时运行远程程序的多个版本，这就可以允许改进程序过程时进行从容迁移（graceful migration）。这个想法可概括如下：

因为所有 ONC RPC 报文要标识远程程序、该程序的版本号以及程序中的某个远程过程，这就有可能使程序从某个远程过程的一个版本从容地迁移到另一个版本上去，而且可以在过程的旧版本继续运行的情况下测试服务器的新版本。

21.15 远程程序中的互斥

ONC RPC 机制说明，在给定时刻，至多可以调用远程程序中的一个远程过程。因此，在给定的远程程序里，RPC 自动提供了过程间的互斥。对于那些维护着被多个过程共享的数据的远程程序来说，这种互斥是重要的。例如，如果有一个远程数据库程序，它含有用于 insert 和 delete 操作的远程过程，程序员不需要操心这两个远程过程会相互影响，因为 RPC 的机制只允许一次执行一个调用。系统在当前调用完成之前，会阻塞其他的调用。概括起来就是：

ONC RPC 提供了在单个远程程序中各个远程过程间的互斥；在远程程序中，一次最多可以执行一个远程过程调用。

21.16 通信语义

在选择 ONC RPC 的语义时，设计者必须在两种可能中进行选择。一方面，为尽量使远程过程调用的行为像本地过程调用，RPC 应使用一种像 TCP 这样可靠的传输，而且应该对程序员保证可靠性。远程过程调用机制应当要么把调用传递到远程过程且接收应答，要么报告通信无法进行。另一方面，为允许程序员使用高效率的、无连接的传输协议，远程过程调用机制应当支持用 UDP 这样的数据报协议进行通信。

ONC RFC 并不强迫使用可靠语义。它允许每个应用程序选择用 TCP 还是 UDP 作为传输协议。此外，该标准并没有指明为达到可靠交付所要附加的协议或机制，而是把 RPC 语义作为下层的传输协议语义的一个功能。例如，由于 UDP 允许数据报丢失或重复，RPC 指明，使用 UDP 的远程过程调用也许会丢失或重复。

21.17 至少一次语义

ONC RPC 以一种最简单的方式定义了远程过程调用的语义，它指明程序在任何交互中应当以最坏的可能做基础。例如，当使用 UDP 时，请求或应答报文（调用远程过程或从远程过程返回）可能会丢失或重复，如果远程过程没有返回，调用者不能认为远程过程没有被调用，这是因为即使请求没有丢失，应答也是可能会丢失的。如果远程过程返回了，调用者可以认为远程过程被调用了至少一次。然而，调用过程不能认为远程过程恰好被调用了一次，因为请求可能丢失或重复过，或者应答报文丢失过。

ONC RPC 标准用术语——至少一次语义 (at least once semantics) 来描述 RPC 在执行时其调用者收到了一个应答，此标准还使用零或多次语义 (zero or more semantics) 来描述调用者没有收到应答时一个远程过程调用的行为。

RPC 的零或多次语义意味着程序员的重要责任：

选择 UDP 作为 ONC RPC 应用之传输协议的程序员，他们所构建的程序必须要能容忍零或多次执行语义。

在实际中，零或多次语义往往意味着程序员要使每个远程过程调用是幂等的 (idempotent)^①。例如，考虑一个远程文件访问应用程序，一个远程调用在文件后而加入数据，这个调用就不是幂等的，因为重复执行这个过程会重复地增加数据。另一方面，向一个文件的某个指定位置写入数据的远程过程是幂等的，因为重复执行这个过程还会把数据写到相同的位置上。

21.18 RPC 重传

伴随 ONC RPC 实现一起提供的库软件 (library software) 包含一种简单的超时和重传策略，但它并不保证严格意义上的可靠性。默认的超时机制是以固定的 (非自适应的) 超时时间和固定的重试次数实现的。当 RPC 库软件发送报文时 (对应于进行一次远程调用)，它便启动计时器。如果计时器在响应到达前期满，软件便重发请求。程序员可以为某个给定应用调整超时时间和重试限制，但该软件不能自动适应长的网络延时或这段时间内延时所发生的改变。

当然，简单的重传策略并不能保证可靠性，也不能保证发起调用的应用程序可以得出关于远程过程执行情况的正确结论。例如，如果网络丢失了所有的响应，调用者可能会重传几次请求，而每次请求的结果都是使远程过程执行一次。然而，最后调用者的机器中的库软件将达到它的重试限制，于是宣布远程过程不能执行。更重要的是，应用程序不能将失败解释为远程过程从来没有执行过 (实际上，它可能已执行了多次)。

21.19 将远程程序映射到协议端口

UDP 和 TCP 传输协议使用 16 比特的协议端口号来标识通信端点。前面的几章描述了服务器如何创建被动的套接字，并将这个套接字绑定到某个熟知协议端口上，等待客户与之联系。为使客户和服务器的这种会合成为可能，我们假设每个服务都被指派了一个唯一的协议端口号，并且这些指

^① 这个术语取自数学，在数学中，如果对运算的重复应用产生相同的结果，这个运算就称为幂等的。

派是熟知的。因此，服务器和客户都认可这个协议端口，服务器在这个端口上接受请求，因为它们都参考了公开的端口分配表。

ONC RPC 引出了一个有趣的问题：因为它使用 32 比特数来标识远程程序，RPC 程序的这种表示可能会超出协议端口的范围。因此，不可能将 RPC 程序号直接映射到协议端口号。更重要的是，因为并不是所有的 RPC 程序都能被分配一个惟一的协议端口号，程序员不能使用一种依赖于分配熟知端口的体制。

尽管 RPC 程序的潜在数量超过了分配熟知端口的能力，但 RPC 与其他服务没有什么特别的不同。在任意给定时间，单台计算机仅仅执行少量的远程程序。因此，只要端口分配是临时的，每个 RPC 程序可以获得一个协议端口号，并且使用这个端口号进行通信。

如果 RPC 程序没有使用保留的、熟知的协议端口，客户就不能直接与之联系。为了解其中的原因，让我们考虑一下客户和服务器的一些构件。当服务器（远程程序）开始执行时，它要求操作系统分配一个未使用的协议端口号。服务器使用这个新分配的协议端口进行所有的通信。对服务器的每次启动，系统都有可能选择不同的协议端口（即在每次系统启动时，服务器可能被分配不同的端口）。

客户（发起远程过程调用的程序）知道它所希望与之联系的远程程序的机器地址以及 RPC 程序号。然而，因为 RPC 程序（服务器）只有在它开始执行之后才获得协议端口，客户就不能知道服务器获得了哪个协议端口。因此，客户不能直接联系远程程序。

21.20 动态端口映射

为解决端口标识问题，客户必须能够在它启动时将 RPC 程序号和机器地址映射为目的机器中服务器所获得的协议端口。这种映射必须是动态的，因为如果机器重启或者 RPC 程序再次开始执行，端口可能会改变。

为允许客户联系远程程序，ONC RPC 机制还包含动态映射服务。提供 RPC 程序（即运行服务器）的每台机器维护着一个端口映射数据库，而且还提供了一种允许调用者将 RPC 程序号映射为协议端口的机制。RPC 端口映射机制是这样的，它在每台机器中用一个服务器维护一个小数据库，这个服务器称为 RPC 端口映射器（RPC port mapper）或者有时简称为端口映射器（port mapper）。图 21.8 展示了作为一个单独的服务器进程运行的端口映射器。

21.21 RPC 端口映射器算法

运行在各台机器中的端口映射器使用算法 21.1。即使远程程序动态地分配协议端口，端口映射器也能允许客户到达远程程序。只要一个远程程序（即服务器）开始执行，它便分配一个用于通信的本地协议端口。于是，远程程序便与它本地机器上的端口映射器联系，将一个三元组加到数据库中：

(RPC 程序号, 协议端口号, 版本号)

RPC 程序一旦注册了它自己，其他机器上的调用者就可以通过向端口映射器发送请求来找到它的协议端口。

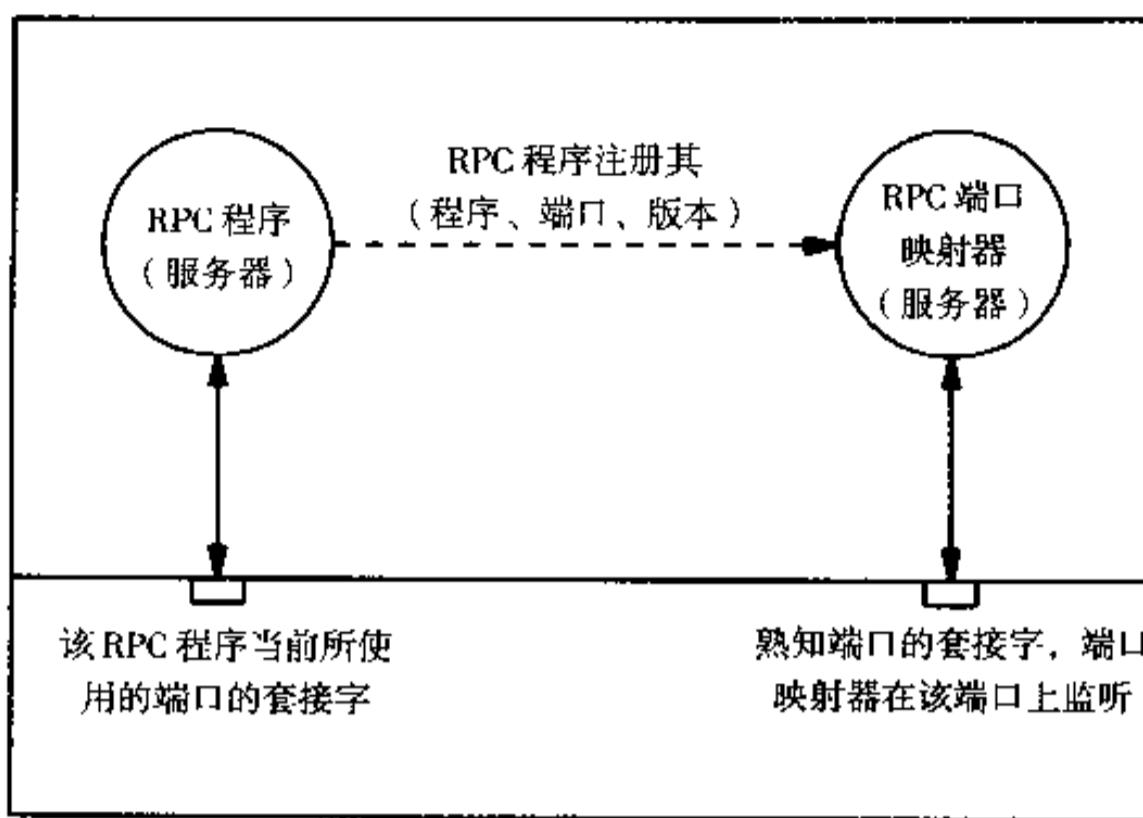


图 21.8 ONC RPC 端口映射器。每个 RPC 程序将其程序号、协议端口号以及版本号注册到本地机器中的端口映射器上。调用者将与机器上的端口映射器进行联系，以便找出这台机器上的某个给定 RPC 程序所使用的协议端口

算法 21.1

1. 创建被动套接字，将其绑定到分配给 ONC RPC 端口映射器服务的熟知端口上 (111)。
2. 重复地接受请求，以注册 RPC 程序号，或是在给定某一 RPC 程序号时查找其协议端口。
注册请求来自与端口映射器处于同一个机器的 RPC 程序。每个注册请求指明了一个三元组，它由 RPC 程序号、版本号以及当前用于到达该程序的协议端口号所组成。当注册请求到达时，端口映射器将这个三元组加到它的映射数据库中。
查找请求来自任意机器。每一个查找请求都指明了远程程序号和版本号，并且请求知道用于达到这个程序的协议端口号。端口映射器在其数据库中查找这个远程程序，并以返回那个程序的相应协议端口作为响应。

算法 21.1 ONC RPC 端口映射器算法。每个实现 RPC 程序之服务器端的机器都要运行端口映射服务器

在一台给定机器中，端口映射器的作用就像是美国电话系统中的号码查询台：调用者可以询问端口映射器如何才能到达某台机器上的特定 RPC 程序。为了与远程程序进行联系，调用者必须要知道执行远程程序的机器的地址、分配给该程序的 RPC 程序号以及版本号。调用者首先联系目的机器中的端口映射器，接着向端口映射器发送 RPC 程序号和版本号。端口映射器返回这个指定程序目前正在使用的协议端口号。调用者总能找到端口映射器，因为端口映射器使用熟知协议端口 (111) 进行通信。调用者一旦知道了目标程序正在使用的协议端口号，就可以直接联系远程程序了。

21.22 ONC RPC 的报文格式

不像许多 TCP/IP 协议那样，ONCRPC 并没有让报文使用固定的格式。协议标准使用一种称为

XDR 的语言定义 RPC 报文的一般格式以及每个字段中的数据条目。因为 XDR 语言与由 C 所声明的数据结构类似，熟悉 C 的程序员往往不需要很多解释就可以阅读和理解这种语言。一般来说，该语言指明应该如何装配构成报文的一系列数据条目。每个条目都用 XDR 表示标准来编码。

RPC 报文首部中的报文类型字段区分两种类型的报文，这两种类型是客户用来发起远程过程调用的报文和远程过程用来进行应答的报文。报文类型字段中所使用的常量可以用 XDR 语言定义。例如，有如下声明：

```
enum msg_type {                                /* RPC message type constants */
    CALL = 0;
    REPLY= 1;
};
```

它声明了符号常量 CALL 和 REPLY 是枚举类型 msg_type 中的值。

XDR 语言中的数据结构可以看作是 XDR 类型的序列，它可以被解释为指令，该指令用于对报文进行装配，装配的方法是用 XDR 构成数据。例如，一旦为符号常量声明了值，XDR 语言可以定义 RPC 报文的格式：

```
struct rpc_msg {                            /* Format of an RPC message */
    unsigned int mesgid;        /* used to match reply to call */
    union switch (msg_type mesgt) {
        case CALL:
            call_body cbody;
        case REPLY:
            reply_body rbody;
    } body;
};
```

这个声明说明了 RPC 报文——rpc_msg，该报文由一个整数型报文标识符 mesgid 和后面跟随着的用 XDR 表示的不同联合（union）组成。使用 XDR 表示，每个联合由一个整数开始，在这里是 mesgt。mesgt 决定了 RPC 报文中剩下部分的格式；它包含一个值，该值定义消息要么是一个 CALL，要么是一个 REPLY。CALL 报文以 call_body 的形式包含了更进一步的信息。REPLY 报文以 reply_body 的形式包含了更进一步的信息。对 call_body 和 reply_body 的声明必须在其他地方给出。例如，RPC 定义的 call_body 具有如下形式：

```
struct call_body {                      /* format of RPC CALL           */
    unsigned int rpcvers;      /* which version of RPC?        */
    unsigned int rprog;        /* number of remote program     */
    unsigned int rprogvers;    /* version number of remote prog */
    unsigned int rproc;        /* number of remote procedure   */
    opaque_auth cred;         /* credentials for called auth. */
    opaque_auth verf;         /* authentication verifier       */
    /* ARGS */                  /* arguments for remote proc.   */
};
```

这个远程过程调用体（body）的前面几行没有什么特别的。调用者必须在字段 rpcvers 中提供 RPC 协

议版本号，以便保证客户和服务器使用相同的报文格式。三个整型字段 rprog、rprogvers 和 rproc 分别指明要调用的远程程序、所期望的程序版本以及在这个程序中的远程过程。字段 cred 和 verf 含有被调用程序用于鉴别调用者身份的信息。

21.23 对远程过程进行参数排序

在 RPC 报文中，鉴别信息之后的一些字段含有远程过程的参数。参数的个数以及每个参数的类型取决于被调用的远程过程。

RPC 必须将它的所有参数表示为一种外部的形式，这种形式允许参数在计算机之间传递。具体来说，如果传递给远程过程的任何参数由一个像链表那样的复杂数据结构组成，那么它必须被编码成一种可通过网络发送的压缩形式。我们使用术语排序 (marshal)、线性化 (linearize) 或串行化 (serialize) 来代表这种对参数编码的工作。我们说，RPC 的客户一端将参数排序到报文中，在服务器一端再将它们反排序。程序员必须记住，尽管 RPC 允许 RPC 调用包含有复杂的数据对象，排序或者反排序大的数据结构可能要求大量的 CPU 时间和网络带宽。因此，多数程序员避免把链接的结构作为参数来传递。

21.24 鉴别

RPC 定义了多个可能的鉴别 (authentication) 形式，包括一个依赖于操作系统提供功能的简单的鉴别体制和一个使用数据加密标准 DES (Data Encryption Standard) 的更复杂的体制，DES 最初由国家标准局 (NBS)^①公布。鉴别信息可以是以下四种类型之一：

```
enum auth_type {          /* possible forms of auth.      */
    AUTH_NULL = 0;        /* no authentication           */
    AUTH_UNIX = 1;         /* Machine name authentic.    */
    AUTH_SHORT= 2;         /* Used for short form auth.in */
                           /* messages after the first   */
    AUTH_DES   = 3;        /* NTST's(NBS's) DES standard */
};


```

对每种情况，RPC 把鉴别信息的格式和解释留给了鉴别子系统。因此，RPC 报文中鉴别结构的声明使用了关键字 opaque，以此指出它出现在报文中，但不要任何解释：

```
struct opaque_auth { /* structure for authent.info. */
    auth_type atype;  /* which type of authentication */
    opaque body<400>; /* data for the type specified */
};


```

当然，每种鉴别方法都使用一种特定的格式来编码数据。例如，多数发送 RPC 报文的网点都使用各种版本的 UNIX 操作系统。UNIX^②鉴别定义其鉴别信息的结构要包含几个字段：

① 国家标准局 NBS 业已改名为国家标准与技术研究院 (National Institute for Standards and Technology, NIST)。

② 尽管这种鉴别机制冠有 UNIX 的名字，但它同样适用于 Linux。

```

struct auth_unix {           /* format of UNIX authentication */
    unsigned int timestamp;   /* integer timestamp */
    string smachine<255>;    /* name of sender's machine */
    unsigned int userid;      /* user id of user making req. */
    unsigned int grpid;       /* group id of user making req. */
    unsigned int grpids<10>; /* other group ids for the user */
};


```

UNIX 鉴别依赖于客户机器在字段 smachine 中提供它的名字，在字段 userid 中给出发出请求的用户的数值标识符 (numeric identifier)。客户还要在字段 timestamp 中指明它的本地时间，该值可以用来对请求排序。最后，客户在字段 grpid 和 grpids 中填入发送用户的主数字组标识符 (main numeric group identifier) 和辅组标识符 (secondary group identifier)。

21.25 RPC 报文表示的例子

XDR 定义了 RPC 报文中每个字段的长度和外部格式。例如，XDR 指明一个整数（可以是有符号的，也可以是无符号的）占用 32 比特，并按照大数在前字节顺序存储。

图 21.9 给出了一个 RPC CALL 报文的例子。每个字段的长度由它的 RPC 定义和 XDR 的长度规约所决定。例如，MESSAGE TYPE 字段被定义为枚举类型，XDR 将其作为一个 32 比特整数存储。

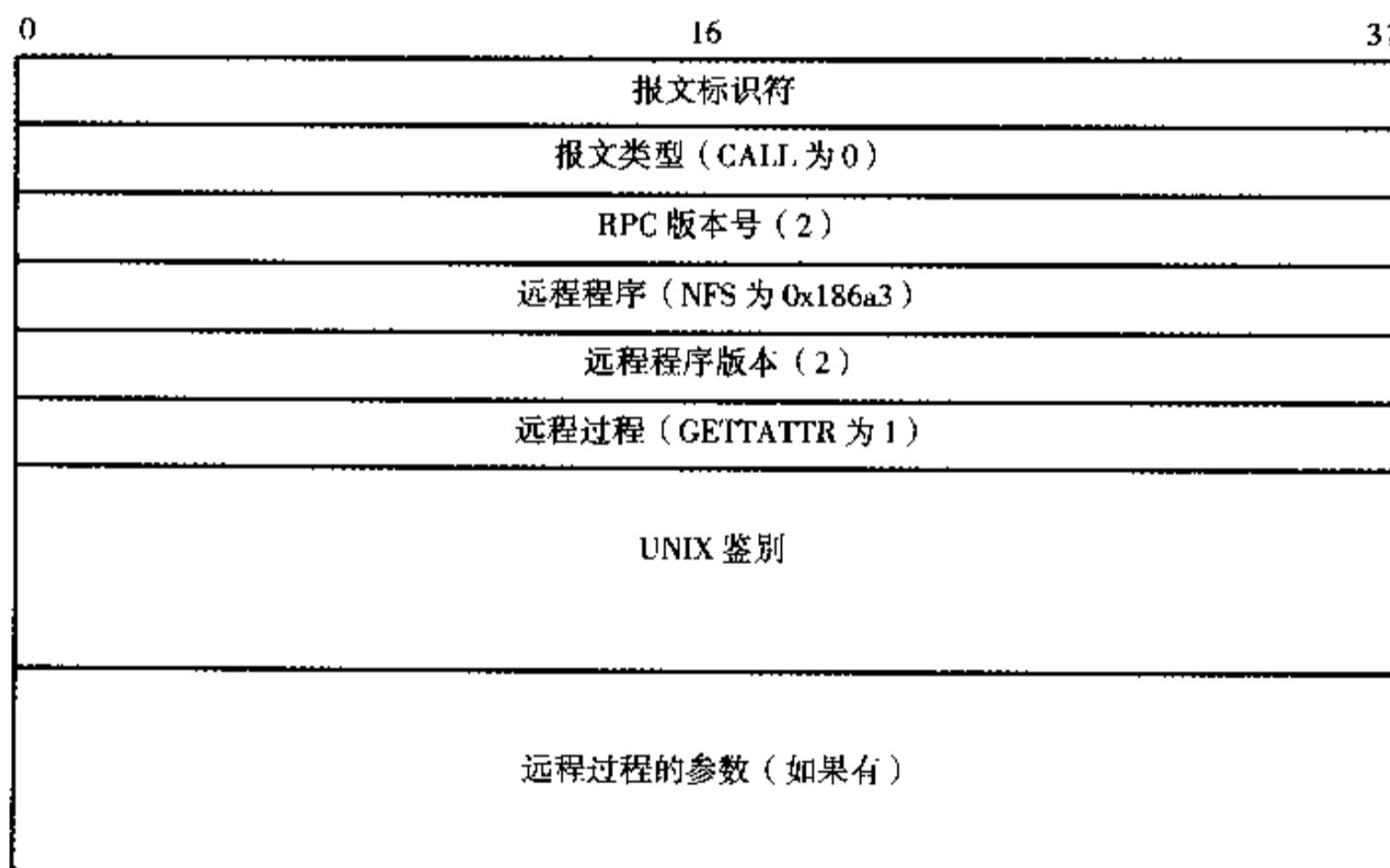


图 21.9 RPC CALL 报文所使用的外部格式的一个例子。该报文的开始几个字段具有固定长度，但以后的一些字段的长度则随其内容而变化

21.26 UNIX 鉴别字段的例子

RPC 报文中鉴别字段的长度取决于其内容。例如，UNIX 鉴别结构中的第二个字段是机器名，

它有可变长度的格式。XDR 将可变长度字符串表示为：开始是 4 字节的整数的长度，接着是字符串本身。图 21.10 展示了 UNIX 鉴别字段的表示。在这个例子中，计算机的名字 merlin.cs.purdue.edu 含有 20 个字符。

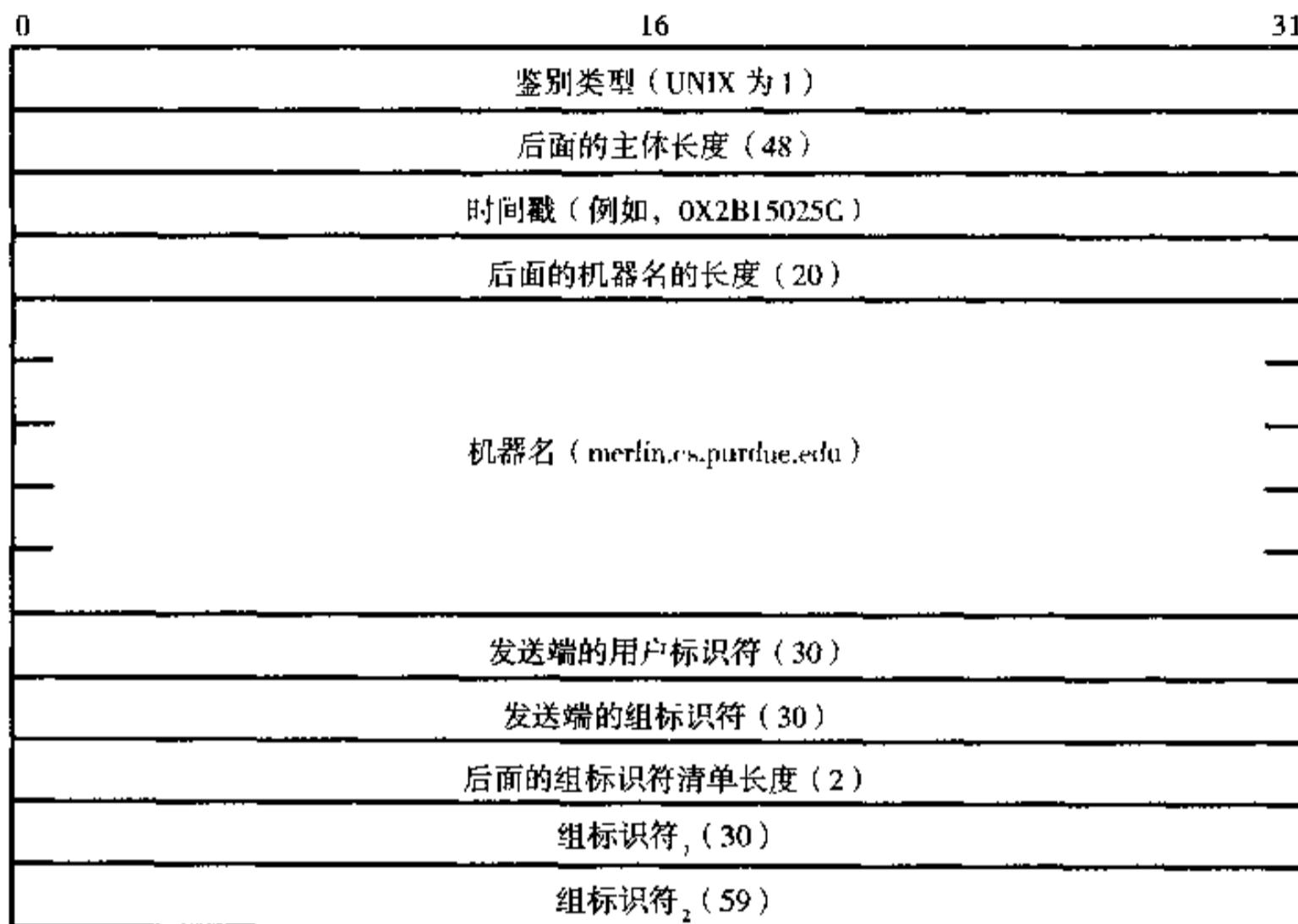


图 21.10 ONC RPC 报文中的 UNIX 鉴别字段的表示。本例所用的值取自某一用户发出的报文，该用户用的机器是 merlin.cs.purdue.edu，具有数值登录标识符 30

21.27 小结

远程过程模型使分布式程序易于设计和理解，这是因为它把客户-服务器之间的通信与传统的过程调用关联起来。远程过程模型把每个服务器看作是一个或多个过程的实现。从客户发给服务器的报文对应于对远程过程的“调用”，而从服务器发给客户的响应对应于过程的返回。

就如同常规的过程，远程过程接受参数并返回一个或多个结果。调用者和被调用的过程之间传递的参数和结果为客户和服务器之间的通信提供了明确的定义。

使用远程过程模型有助于程序员关注于应用而不是通信协议。程序员可以构建和测试解决某个具体问题的常规程序，接着可以将这个程序划分成几个部分，这些部分在两台或多台计算机上执行。

Sun Microsystems 公司定义了远程过程调用的具体形式，现已成为事实上的标准。ONC RPC 为标识远程过程指明了一种体制，还为 RPC 报文格式指明了一个标准。这个标准为使报文表示独立于机器而使用了外部数据表示 XDR。

ONC RPC 程序并不像传统的客户和服务器那样使用了熟知协议端口，取而代之的是使用了一种动态绑定机制，这种机制允许每个 RPC 程序在启动时选择一个任意的、未使用过的协议端口。这种绑定机制还要求每台提供 RPC 程序的计算机在熟知协议端口上运行端口映射服务器（称为 RPC 端口映射器）。每个 RPC 程序在获得了协议端口号之后要向其本地机器中的端口映射器注册。当某

个 RPC 客户想与某个 RPC 程序联系时，它首先要联系目标机器中的端口映射器。端口映射器在响应中告诉客户目标 RPC 程序使用了哪个端口。客户一旦获得了目标 RPC 程序所使用的正确协议端口号，就可以使用这个端口直接与目标 RPC 程序进行联系。

深入研究

Sun Microsystems 公司 [RFC 1057] 为 ONC RPC 定义了标准，并描述了本章所给出的大多数概念。Srinivasan [RFC 1831] 包含了一个新的版本，它是建议的标准。其他信息可以在 Linux 操作系统附带的联机文档中找到。

习题

- 21.1 阅读 ONC RPC 的说明，画一个图，说明典型的回报报文中各个字段的长度。
- 21.2 做一个实验，测量一下使用端口映射器所引入的额外负担。
- 21.3 客户可以通过把协议端口绑定放入高速缓存来避免不必要的额外负担。即在客户联系了端口映射器并获得了目标 RPC 程序的协议端口后，它可以将这个绑定存储在高速缓存中，这样可以避免再次查找。绑定的合法性会保持多久？
- 21.4 端口映射器的概念可以扩展到 RPC 以外的其他服务吗？解释你的答案。
- 21.5 使用端口映射器而不是使用熟知端口的主要优点和缺点是什么？
- 21.6 当 RPC 客户联系某个端口映射器时，它必须或者指明或者试探目标程序是否已经打开了 UDP 端口或 TCP 端口。仔细阅读规约，看看 RPC 客户是如何区分这两者的。
- 21.7 如果你的计算机具有实用程序 `rpcinfo`，阅读手册的有关部分以便了解它的功能。使用 `rpcinfo` 获得你的计算机上可供使用的 RPC 程序及其版本号的清单。
- 21.8 阅读其他厂商设计的 RPC。它们中有哪些概念是 ONC RPC 中所不具备的？
- 21.9 考虑 ONC RPC 中所使用的鉴别体制。在你的机构中使用这种体制完全安全吗？在两个机构之间使用呢？
- 21.10 比较 DCE RPC 和 ONC RPC。它们有什么不同。
- 21.11 阅读 Common Object Request Broker Architecture —— CORBA。这种体系结构为 RPC 提供了哪些新的设施？

第 22 章 分布式程序的生成 (rpcgen 的概念)

22.1 引言

前面一章给出了远程过程调用模型的原理。它描述了远程过程调用概念，还解释了程序员如何使用远程过程调用来构建按照客户 - 服务器范例运行的程序，最后，还描述了 ONC RPC 机制。

本章将继续这一讨论。我们将关注那些使用 RPC 的程序的结构，说明程序如何沿着过程边界进行划分。本章将介绍 stub^①过程概念，还将介绍一个程序生成工具，它可以自动生成与 ONC RPC 相关的大部分代码。另外还将讨论一个过程库，它可以使构建服务器和客户变得容易，这些服务器提供远程过程，而客户调用这些远程过程。

下一章将完成对生成器 (generator) 的讨论，说明一个程序员所要采取的一系列步骤，程序员按照这个步骤创建一个常规的程序，然后将这个程序划分为本地构件和远程构件。它还给出了一个简单的应用例子，我们将遵循构造分布式应用程序的步骤来构建这个例子。下一章所给的例子将补充本章所做的概念性描述，它展示了许多细节并给出了生成器所产生的代码。

22.2 使用远程过程调用

远程过程调用模型是一般性的。程序员可以选择下列方式中的任何一个来使用远程过程模式：

- 只作为一种程序的规约技术。为此，程序员参照 RPC 模型，将客户和服务器间的所有交互要么指明为过程调用，要么指明为返回。过程的参数指明了客户和服务器间所传递的数据。在设计客户和服务器时，程序员可以忽略过程的结构，但是利用过程的规约来验证最终系统的正确性。
- 既作为一种程序的规约又作为程序设计时的一种抽象。为实现这种方法，在设计应用程序和通信协议时，要想到远程过程调用。对所设计的通信协议的每个报文，都紧密地对应于远程过程调用。
- 为概念性设计和实现中的明确性。为在实现中包含 RPC，程序员设计广义的 RPC 报文格式和将控制传递给远程过程的协议。在客户和服务器之间传递数据时，程序员严格地遵照过程的规约。程序使用一种标准的外部数据表示来对参数进行编码，并且严格地按照设计中所给出的数据类型规约来进行。它调用标准的库例程来完成数据的计算机本地表示和数据在网络中穿越时所使用的外部表示之间的转换。
- 通过白手起家构造所有的软件来设计和实现。程序员构建一个解决某个问题的传统应用程序，接着，将这个程序沿着过程边界划分成几片，再将这些程序片转移到各台独立的机器

^① stub procedure 目前无标准译名，故暂用原文 stub。

中。在调用远程过程时，程序使用 ONC RPC 报文格式（包括 XDR 数据表示）和 ONC RPC 程序编号体制。程序员只按照 RPC 的规约来构建程序的实现，使用 ONC RPC 端口映射器将远程程序号绑定到对应的协议端口上。

- 使用标准的库来设计和实现。程序员构建应用程序并且用 ONC RPC 规约将它划分成几片，但是尽可能依赖于现有的 RPC 库例程。例如，程序员使用库例程向端口映射器注册，使用库例程构造和发送远程过程调用，以及使用它们构造应答。
- 为自动实现。程序员完全遵照 ONC RPC 规约，使用程序自动生成工具帮助程序员自动构造客户和服务器代码所必须的一些程序片段，以及调用 RPC 库例程，执行诸如向端口映射器注册、构造一个报文、将调用分派到远程程序中适当的远程过程等项任务。

22.3 支持 RPC 的编程工具

ONC RPC 规约涉及的面既广泛又复杂。构建实现 RPC 的应用而不利用任何现有的程序是乏味和耗时的。多数程序员乐于在各个应用中避免重复劳动。他们依赖于库例程和编程工具来处理多数工作。

ONC RPC 的实现为那些希望避免不必要编程的程序员提供了极大的帮助。帮助源于以下四种形式：

1. XDR 库例程，它们将各自的数据条目从内部形式转换为 XDR 标准外部表示。
2. XDR 库例程，它们将用于定义 RPC 报文的复杂数据聚合体（比如，数组和结构）进行格式化。
3. RPC 运行时间库例程，这些例程允许程序调用远程过程、向端口映射器注册服务、将传入调用分派到远程程序中正确的远程过程。
4. 程序生成工具^①，它产生构建使用 RPC 分布式程序所需要的许多 C 的源文件。

RPC 运行时间库具有一些过程，它们提供了 RPC 所需要的多数功能。例如，过程 call rpc 向服务器发送 RPC 报文。它具有如下形式：

```
callrpc (host, prog, progver, procnum, inproc, in, outproc, out);
```

参数 host 指向一个字符串，该字符串含有机器名，远程过程在这台机器上执行。参数 prog、progver、procnum 分别标识了远程程序号、所使用的程序的版本以及远程过程号。参数 inproc 给出了本地过程的地址，可以调用该过程把参数整理到 RPC 报文中，参数 in 给出了给远程过程的参数的地址。参数 outproc 给出了本地过程的地址，可以调用该过程对结果进行解码，out 参数给出了存储器地址，过程执行结果将放置在这里。

尽管 callrpc 已处理了发送 RPC 报文所要求的许多杂务，ONC RPC 库例程还包含许多其他的过程。例如，客户调用函数：

```
handle = clnt_create(host, prog, vers, proto);
```

^① 程序员往往称此程序生成器为 stub 生成器 (stub generator)。至于为什么普遍采用这个词，待到研究过生成器的工作原理后便清楚了。

它创建了一个整数标识符，可以用于发送 RPC 报文。RPC 将这个整数标识符称为句柄 (handle)；一些 RPC 库过程将句柄作为它们的参数之一。给 clnt_create 的参数指明了远程主机的名字、在该主机上的远程程序的名字、该程序的版本号，还有一个协议 (TCP 或 UDP)。

这个库还包括创建、存储和处理鉴别信息的例程。例如，过程：

```
authunix_create(host, uid, gid, len, aup_gids);
```

它为某个给定主机上的某个给定用户创建了一个鉴别句柄。它的参数指明了远程主机、用户的登录标识符和主组标识符，还有用户所属的一些组——aup_gids。参数 len 指明了在这个组中的条目的数目。

尽管程序员可以编写直接调用 RPC 库例程的应用，但很少有程序员这样做。多数程序员要依靠本章后面要讨论的程序生成器。这个生成器所产生的代码包含了许多库过程的调用。

22.4 将程序划分成本地过程和远程过程

为理解 RPC 编程工具是如何工作的，有必要先理解一下程序是怎样被划分为本地过程和远程过程的。考虑传统应用中的过程调用。图 22.1 说明了这样的调用。

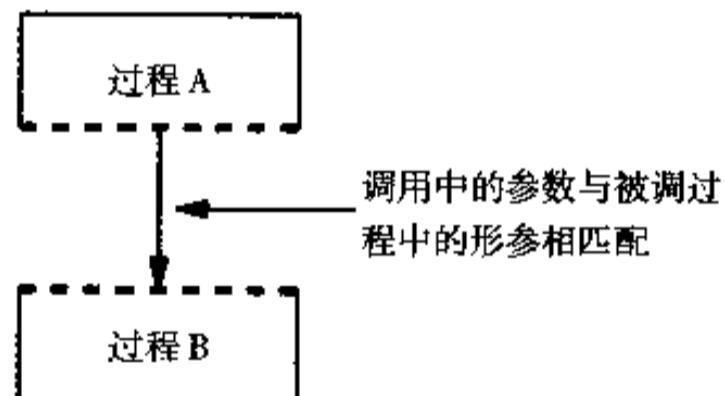


图 22.1 一个过程调用的例子，它说明了调用过程和被调用过程所使用的过
程的接口。虚线表示调用过程和被调用过程中参数之间的一种匹配

每个过程拥有一组形式参数，而每个过程调用要指明一组参数。调用者的所有参数的个数必须同被调用者的所有参数的个数相等，而且每个参数的类型必须与所声明的相应的形式参数的类型相匹配。换句话说，参数定义了调用过程和被调用过程之间的接口。

22.5 为 RPC 增加代码

将一个或多个过程转移到远程机器上要求程序员在过程调用和远程过程之间增加代码。在客户一端，新代码必须要整理参数，将它们转换成独立于机器的表示，创建 RPC 报文，把报文发送给远程程序，接着便等待结果，将最终的结果转换回客户的本地表示。在服务器一端，新代码必须接受传入 RPC 请求，将参数转换成服务器的本地数据表示，将报文分派给合适的过程，还要将结果值转换成独立于机器的数据表示来构成应答报文，再将结果发回给客户。

为保持程序结构原有的交互，而且为把处理 RPC 的代码与处理应用的代码分离开来，RPC 所要求的这段代码可用两个额外过程的形式加入到应用中，它们完全隐藏了通信的细节。这段新过程可在不改变原来的调用和被调用过程之间接口的情况下，加入所要求的功能。保持原来的接口有助

于减少出错的机会，因为它使通信的细节同原来的应用分割开了。

22.6 stub 过程

我们在前面所提到的，为实现 RPC 而加入到程序中的附加过程被称为 stub 过程。理解 stub 过程的最简单方法是设想一个常规的程序，它被分割成两个程序，一个已有的过程被转移到远程机器中。在远程过程（服务器）一端，stub 过程取代了调用者。这两个 stub 实现了远程过程调用所需要的所有通信，它使原来的调用和被调用过程不必更改。图 22.2 展示了 stub 的概念，它说明了 stub 过程怎样允许图 22.1 所示的过程调用被分离成本地部分和远程部分。

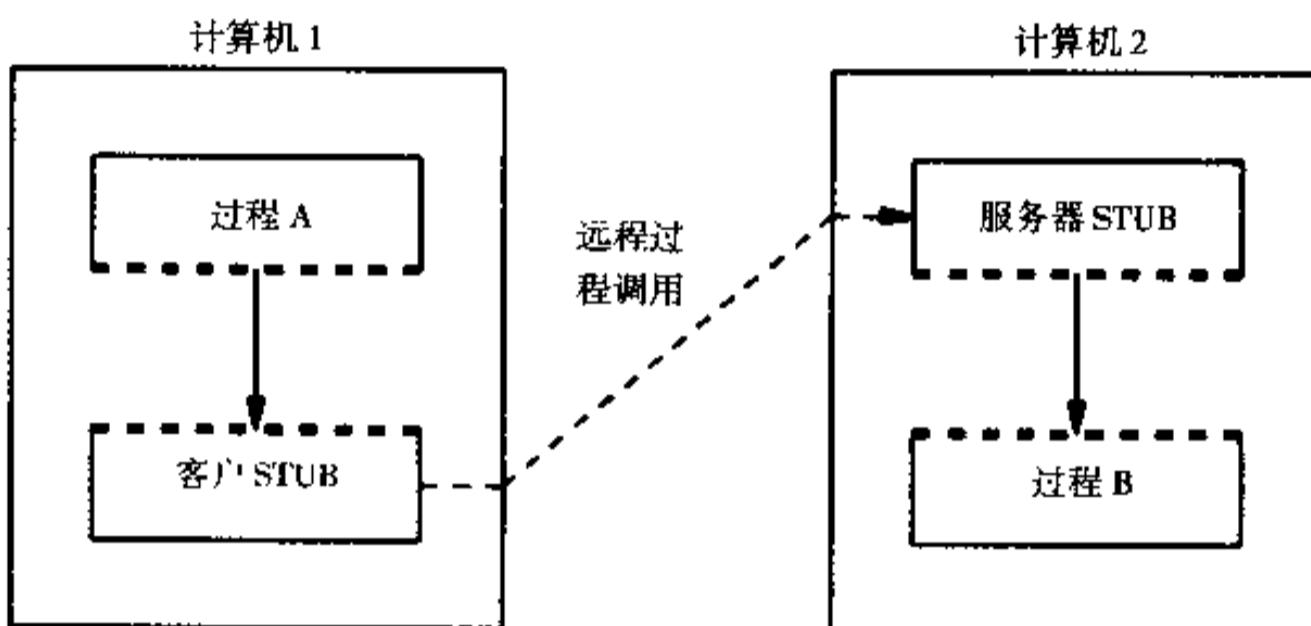


图 22.2 加入到程序中的 stub 过程实现了远程过程调用。因为 stub 与原来的调用使用了一样的接口，所以增加这些 stub 既不要求更改原来的调用过程，也不要更改原来的被调用过程

22.7 多个远程过程和分派

图 22.2 给出了 RPC 的简图，因为它只展示了远程过程调用。在实际中，服务器进程往往在远程程序中包含有多个远程过程。每个 RPC 调用都由一个报文构成，该报文标识某个指明的远程过程。当 RPC 报文到达时，服务器使用报文中的远程过程号将这个调用分派（dispatch）给正确的过程。图 22.3 展示了这个概念。

该图说明了 RPC 是如何与传统的客户 - 服务器实现相关联的。该远程程序由单个服务器进程构成，在任何报文到达之前，它必须已在运行。一个远程过程调用可以来自任何客户，它必须指明运行该服务器的机器的地址、在该机器上的远程程序号以及要调用的远程过程。服务器程序由分派器例程（dispatcher routine）、远程过程以及服务器端的 stub 过程构成。分派器明白远程过程号是怎样与服务器一端的 stub 过程相对应的，它利用这种对应性将每个传入远程过程（incoming remote procedure）转发到合适的 stub 上。

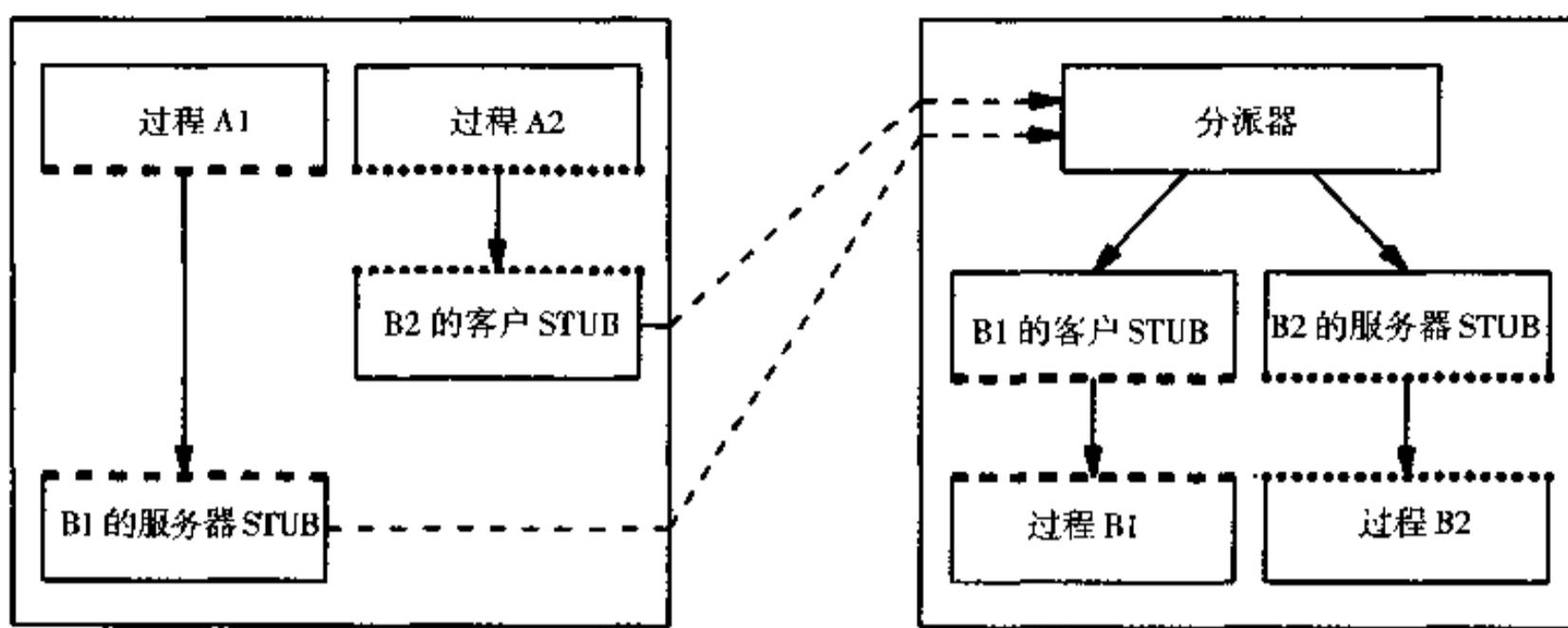


图 22.3 RPC 服务器中的报文分派。客户向一个服务器程序发送 RPC 请求。服务器使用报文中的远程过程号来决定哪个过程将接收这个调用。在本例中，过程 A1 调用过程 B1，过程 A2 调用过程 B2。虚线和点线说明各个过程使用了哪个接口

22.8 客户端的 stub 过程的名字

如果程序员给 stub 过程所取的名字与被调用过程一样，那么将常规的程序转移成分布式的程序就会比较简单。为了理解其中的原因，让我们再次考虑图 22.3 所示的 stub 过程。最初的调用者 A1 含有对过程 B1 的调用。在程序被划分后，A1 成为客户端的一部分，为与远程过程通信，它必须调用 stub 过程。如果程序员把客户一端的 stub 过程命名为最初的那个过程 B1，并在构建该过程时，使其具有同 B1 完全一样的接口，这样，调用过程（A1）就不需要修改了。实际上，甚至不必重新编译过程 A1 就可以完成这种改变。最初编译出的 A1 的二进制代码可以与针对 B1 的新的客户端 stub 过程相链接，以此产生合法的客户。这种方法增加了客户端的 stub 过程，但不必修改最初的调用者，这样就把 RPC 代码从最初的应用程序代码中隔离了出来，它使编程容易，并且减少了引入差错的机会。

当然，把客户的 stub 命名为 B1 使源代码管理更加困难了，因为这样意味着程序的分布式版本将具有两个取名为 B1 的过程：客户端的 stub，以及已成为服务器的一部分的最初那个过程。B1 的这两个版本决不能成为同一个被链接的程序的一部分。在多数情况下，它们不会在同一台计算机上执行。因此，只要程序员在构建目标程序时小心运用，这种 stub 方法就会工作得很好。概括起来就是：

为构建应用程序的分布式版本，程序员必须将一个或多个过程转移到远程机器中。在这样做时，只要客户端的 stub 过程具有同最初被调用的过程一样的名字，那么，增加这个 stub 过程还是可以允许最初的调用和被调用过程保持不变。

22.9 使用 rpcgen 生成分布式程序

很明显，实现 RPC 服务器所需要的多数代码是不变的。例如，如果远程过程号与客户端 stub 过程之间的映射被保存在数据结构中，那么所有的服务器都可以使用相同的分派例程。与此类似，所有的服务器都可以使用相同的代码来向端口映射器注册。

为避免不必要的编程，ONC/RPC 的实现包括一个工具，它自动地生成实现分布式程序所需要的大多数代码。这个工具叫做 `rpcgen`，它读取一个规约文件作为输入，生成 C 的源文件作为输出。规约文件包含常量、全局数据类型、全局数据以及远程过程（包括过程参数和结果类型）的声明。`rpcgen` 产生的代码包含了实现客户和服务器（它提供指明的远程过程调用）程序所需要的大部分源代码。具体地说，`rpcgen` 为客户端和服务器端生成 stub 过程，它包括参数整理、发送 RPC 报文、把传入调用分派到正确的过程、发送应答、在参数和结果的外部表示和本地数据表示之间进行转换。`rpcgen` 的输出在与应用程序和程序员编写的少数文件相结合后，便产生出了完整的客户和服务器程序。

因为 `rpcgen` 将源代码作为输出，程序员可以编辑这个代码（例如，手工优化该代码以提高性能），或者将它与其他文件相结合。在大多数场合，程序员使用 `rpcgen` 来处理尽可能多的细节。他们尽量使生成客户和服务器的整个过程自动化，以便避免手工改变输出。如果程序的规约改变了或者需要新的远程过程调用，程序员可以修改规约，并且再次使用 `rpcgen` 产生新的客户和服务器而不必手工干涉。

22.10 `rpcgen` 输出和接口过程

为了维护的灵活性，并允许自动生成 stub 过程的大部分代码，`rpcgen` 将每个 stub 过程分成了两个部分。一个部分几乎对所有使用 RPC 的应用都是共同的，它提供基本的客户—服务器的通信；另一部分为应用程序提供了接口。`rpcgen` 由远程过程及其参数的描述出发，自动产生了 stub 过程的通信部分。因为 `rpcgen` 为通信 stub 产生了代码，它指明了客户端所要求的参数以及服务器端的调用序列。在使用由 `rpcgen` 所生成的通信 stub 时，程序员必须接受 `rpcgen` 的调用约定。

这种把 stub 分割成通信例程和接口例程的方法所蕴涵的想法是简单的：它允许程序员选择远程过程所使用的调用约定。程序员创建接口 stub，以便在远程过程调用约定与 `rpcgen` 所生成的由通信 stub 过程所提供的约定之间进行映射。图 22.4 展示了所有这些例程是如何交互的。

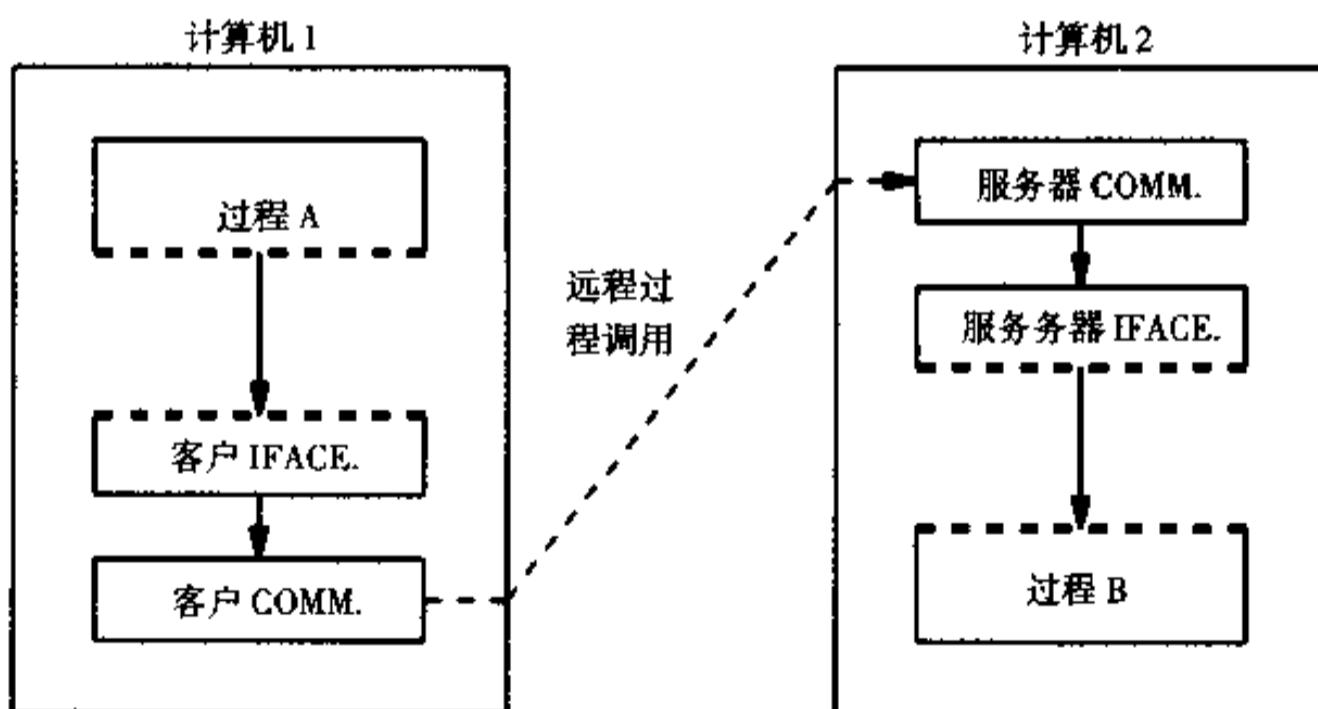


图 22.4 用 `rpcgen` 所创建的分布式程序的形式。`rpcgen` 自动产生基本的通信 stub；程序员提供两个接口过程

如图 22.4 所示，stub 的两个部分都包含两个过程。在客户端，接口过程调用通信过程。在服务器端，通信过程调用接口过程。如果 stub 接口过程定义得仔细，最初的调用者和最初的被调用过程就可以保持不变。

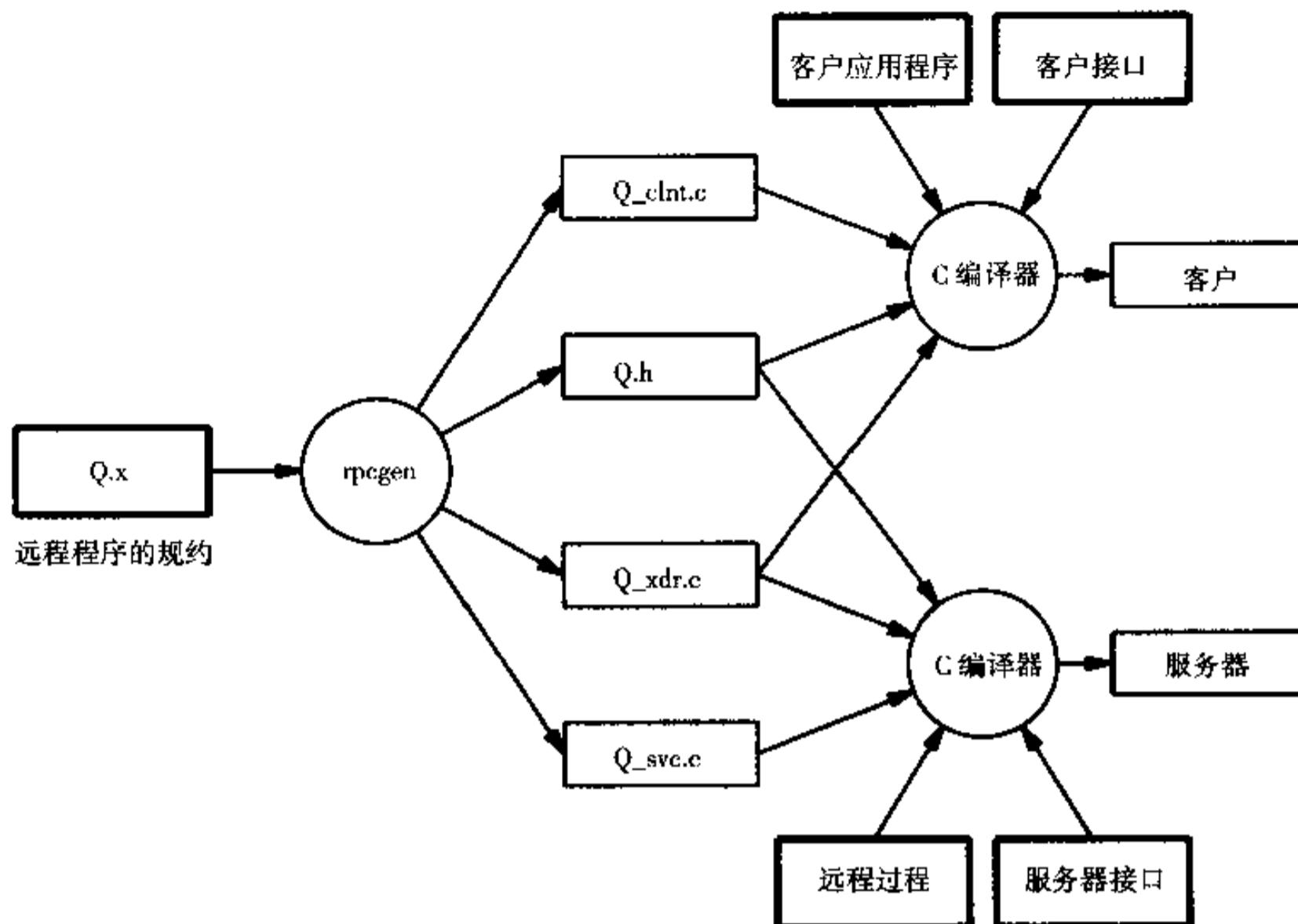


图 22.6 从 rpcgen 的输出构建一个客户和服务器所要求的文件以及处理它们所要求的编译步骤。加粗线的方框表示是程序员所提供的输入

深入研究

关于 rpcgen 的其他信息可以在随该软件所带的文档中找到。Steven [1990] 描述了 RPC 异常处理的细节。

习题

- 22.1 写出当 RPC CALL 报文到达时服务器将采取的步骤的序列。确定数据值何时由外部表示转换为本地表示。
- 22.2 阅读操作系统提供的文档中有关 RPC 库例程的内容。函数 svc_sendreply 的参数是什么？对其中的每个参数，说明为什么需要它。
- 22.3 RPC 的库包含有允许服务器向端口映射器注册的例程。阅读文档，找出过程 pmap_unset 是做什么的。为什么需要它？
- 22.4 如果已经看过针对 RPC 库例程进行编程的源代码，看看它们占用了多少行。将它的大小与针对 rpcgen 程序的源代码的大小进行比较。rpcgen 为什么是这样的？
- 22.5 rpcgen 产生 C 源代码而不是目标代码。随 rpcgen 一起的文档建议说，生成源文件可以允许程序员修改生成的代码。程序员为什么要这样做？

-
- 22.6 参考前一问题，修改 rpcgen 输出的缺点是什么？
 - 22.7 设计一种远程过程调用机制，它通过使用一个附加的参数来决定调用哪个过程（例如，远程过程由 C 的 switch 语句构成，它使用一个新的参数在各种备选动作中作出选择），从而将远程程序中的所有过程结合进单个过程中。这种方法与 ONC RPC 比较，主要优点是什么？主要缺点又是什么？
 - 22.8 如果服务器端的 ONC RPC 程序使用了套接字，它可用来实现互斥的可能方法有哪些（即如何保证在任何时间只有一个远程过程被调用）？每种方法的优缺点是什么？提示：考虑套接字选项和文件的创建。

第 23 章 分布式程序的生成 (rpcgen 的例子)

23.1 引言

前面的几章给出了远程过程调用模型的原理以及 ONC RPC 的机制。我们讨论了远程过程调用的概念，解释了如何沿着过程调用的边界把程序划分开。还描述了 rpcgen 工具及其相关的库例程怎样自动生成使用 ONC RPC 的程序的大部分代码。

本章将完成对 rpcgen 的讨论，给出了程序员所要采取的一系列步骤，程序用这些步骤，首先创建一个常规的程序，接着将这个程序划分为本地构件和远程构件。本章给出一个应用例子，遵照所给的每一步骤进行处理。本章将展示 rpcgen 的输出，还将展示为创建使用 RPC 的分布式程序以及它的客户和服务器所要求的其他代码。

23.2 说明 rpcgen 的例子

我们举个例子来阐明 rpcgen 是如何工作的，并说明大部分细节。这个例子的重点在于解释 rpcgen 是如何工作的，所以我们选择了一个特别简单的应用。当然，在实际当中很少有像我们这个例子这样简单或易于仿效的。因此，读者应把这个例子看作指导教程，并且不必追问这个应用是否是真的构成了分布式解决方案。

23.3 查找字典

作为一个使用 rpcgen 的例子，让我们考虑一个实现简单数据库功能的应用。该数据库提供四个基本的操作：初始化 (initialize)，初始化数据库（即清除以前存储的所有值）；插入 (insert)，插入一项新的条目；删除 (delete)，删除一个条目；查找 (look up)，寻找某个条目。我们假设数据库中的每个条目都是单个的单词 (word)。因此，该数据库的功能可以看作是一个字典。应用程序插入一组合法的单词，接着使用数据库来检查新单词，以便知道每个新单词是否都在字典中。

为使本例简单，我们将假设应用程序的输入是一个文本文件，在文件中，每行包含一个单字母 (one-letter) 的命令，其后跟随一个单词。图 23.1 中的表列出了这些命令，并且给出了每个命令的含义：

单字母命令	参数	含义
I	无	通过删除所有字来初始化数据库
i	word	把 word (单词) 插入到数据库中
d	word	把 word (单词) 从数据库中删除
l	word	在数据库中查找 word (单词)
q	无	退出

图 23.1 数据库应用例子的输入命令及其含义。有些命令之后必须跟随一个单词，这个单词可以看作是这个命令的参数

例如，下列输入含有一系列的命令数据。这些命令的含义是初始化字典、插入计算机厂商的名字、删除一些名字，并查找三个名字。

```

I
i  Navy
i  IBM
i  RCA
i  Encore
i  Digital
d  RCA
d  Navy
l  IBM
d  Encore
l  CDC
l  Encore
q

```

当这个命令文件作为输入提供给这个字典应用程序时，应用程序会在字典中找到IBM，但不会找到Encore或CDC。

23.4 分布式程序的八个步骤

图22.6^①展示了 rpcgen 的输入要求以及它所生成的输出文件。为创建这些分布式程序所要求的文件，并将这些文件与某个客户和服务器相结合，程序员应采取以下八个步骤：

1. 构建解决该问题的常规应用程序。
2. 选择一组过程，以便将这些过程转移到远程机器中，通过这种方法将程序分解。
3. 为远程程序编写 rpcgen 规约，包括远程过程的名字及其编号，还有对其参数的声明。选择远程程序号和版本号（通常为 1）。
4. 运行 rpcgen 检查该规约，如果合法，便生成四个源代码文件，这些文件将在客户和服务器程序中使用。
5. 为客户端和服务器端编写 stub 接口例程。

^① 图 22.6 可在第 216 页找到。

6. 编译并链接客户程序。它由四个主要的文件构成：最初的应用程序（远程过程中被删除了的那个）、客户端的 stub（由 rpcgen 生成）、客户端的接口 stub 以及 XDR 过程（由 rpcgen 生成）。当所有这些文件都被编译和链接到一起后，最终的可执行程序就是客户。
7. 编译并链接服务器程序。它由四个主要的文件构成：由最初的应用程序得来的过程，它们现在构成了远程程序；服务器端的 stub（由 rpcgen 生成）；服务器端的接口 stub 以及 XDR 过程（rpcgen 生成的）。当所有这些文件被编译和链接到一起后，最终的可执行程序就是服务器。
8. 在远程机器上启动服务器，接着在本地机器上启动客户。

下面的章节将更加详细地解释每一步骤，并用这个字典应用程序来说明一些细微之处。

23.5 步骤 1：构建常规应用程序

要构建这个字典应用例子的分布式版本，第一步要求程序员构造解决该问题的常规程序。文件 dict.c 包含了用 C 语言编写的针对字典问题的应用程序。

```
/* dict.c - main, initw, nextin, insertw, deletew, lookupw */

#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define MAXWORD    50          /* maximum length of a command or word */
#define DICTSIZ   100          /* maximum number of entries in dictionary */
char    dict[DICTSIZ][MAXWORD+1]; /* storage for a dictionary of words */
int     nwords = 0;           /* number of words in the dictionary */

int     nextin(char *cmd, char *word), initw(), insertw(const char *word);
int     deletew(const char *word), lookupw(const char *word);

/*
 * main - insert, delete, or look up words in a dictionary as specified
 */
int
main(int argc, char argv[])
{
    char    word[MAXWORD+1]; /* space to hold word from input line */
    char    cmd;
    int     wrdlen;          /* length of input word */
    while (1) {
        wrdlen = nextin(&cmd, word);
```

```
if (wrflen < 0)
    exit(0);
word[wrflen]='\0';
switch (cmd) {
case 'I': /* "initialize" */
    initw();
    printf("Dictionary initialized to empty.\n");
    break;
case 'i': /* "insert" */
    insertw(word);
    printf("%s inserted.\n", word);
    break;
case 'd': /* "delete" */
    if (deletew(word))
        printf("%s deleted.\n", word);
    else
        printf("%s not found.\n", word);
    break;
case 'l': /* "lookup" */
    if (lookupw(word))
        printf("%s was found.\n", word);
    else
        printf("%s was not found.\n", word);
    break;
case 'q': /* quit */
    printf("program quits.\n");
    exit(0);
default: /* illegal input */
    printf("command %c invalid.\n", cmd);
    break;
}
}

/*
 * nextin - read a command and (possibly) a word from the next input line
 */
int
nextin(char *cmd, char *word)
{
    int      i, ch;

    ch = getc(stdin);
    while (isspace(ch))
```

```
        ch = getc(stdin);
    if (ch == EOF)
        return -1;
    *cmd = (char) ch;
    ch = getc(stdin);
    while (isspace(ch))
        ch = getc(stdin);
    if (ch == EOF)
        return -1;
    if (ch == '\n')
        return 0;
    i = 0;
    while (!isspace(ch)) {
        if (++i > MAXWORD) {
            printf("error: word too long.\n");
            exit(1);
        }
        *word++ = ch;
        ch = getc(stdin);
    }
    return i;
}

/*
 * initw - initialize the dictionary to contain no words at all
 */
int
initw()
{
    nwords = 0;
    return 1;
}

/*
 * insertw - insert a word in the dictionary
 */
int
insertw(const char *word)
{
    strcpy(dict[nwords], word);
    nwords++;
    return nwords;
}
```

```
/*
 * deletew - delete a word from the dictionary
 */
int
deletew(const char *word)
{
    int      i;

    for (i=0 ; i<nwords ; i++)
        if (strcmp(word, dict[i]) == 0) {
            nwords--;
            strcpy(dict[i], dict[nwords]);
            return 1;
        }
    return 0;
}

/*
 * lookupw - look up a word in the dictionary
 */
int
lookupw(const char *word)
{
    int      i;

    for (i=0 ; i<nwords ; i++)
        if (strcmp(word, dict[i]) == 0)
            return 1;
    return 0;
}
```

为使应用程序简单并易于理解，文件dict.c中的这个常规程序使用了一个二维数组来存储单词。全局变量nwords记录了任何时刻字典中单词的数量。主程序包含一个循环，在每次循环中读取并处理输入文件中的一行。它调用过程nextin从下一输入行中读取一个命令（可能还有一个“单词”），接着使用一个C语言的switch语句在六种可能的情况下选择其一。这些情况对应于五个有效的命令再加上处理非法输入的默认情况。

主程序对每种情况都调用一个过程来处理细节。例如，对于插入命令i，它调用过程insertw。过程insertw在数组的末尾插入新的单词，并将nwords加1。

其他过程也按照我们所期望的那样进行操作。过程deletew寻找所要删除的单词，如果找到了这个单词，便用字典中的最后一个单词取代它，然后将nwords减1。而lookupw则顺序地查找数组，以便确定字典中是否有该指定的单词。若有则返回1，否则返回0。

为产生该应用程序的可执行的二进制文件，程序员调用 C 编译器。多数 Linux 系统使用命令：

```
cc -o dict dict.c
```

从文件 dict.c 中的源程序产生命名为 dict 的可执行二进制文件。

23.6 步骤 2：将程序划分成两部分

常规程序一旦构建完成并经过测试，就可以将它划分成本地构件和远程构件了。在程序员开始划分程序之前，他们必须具有一个程序过程调用的概念模型。例如，图 23.2 展示了最初这个字典应用的过程组织情况。

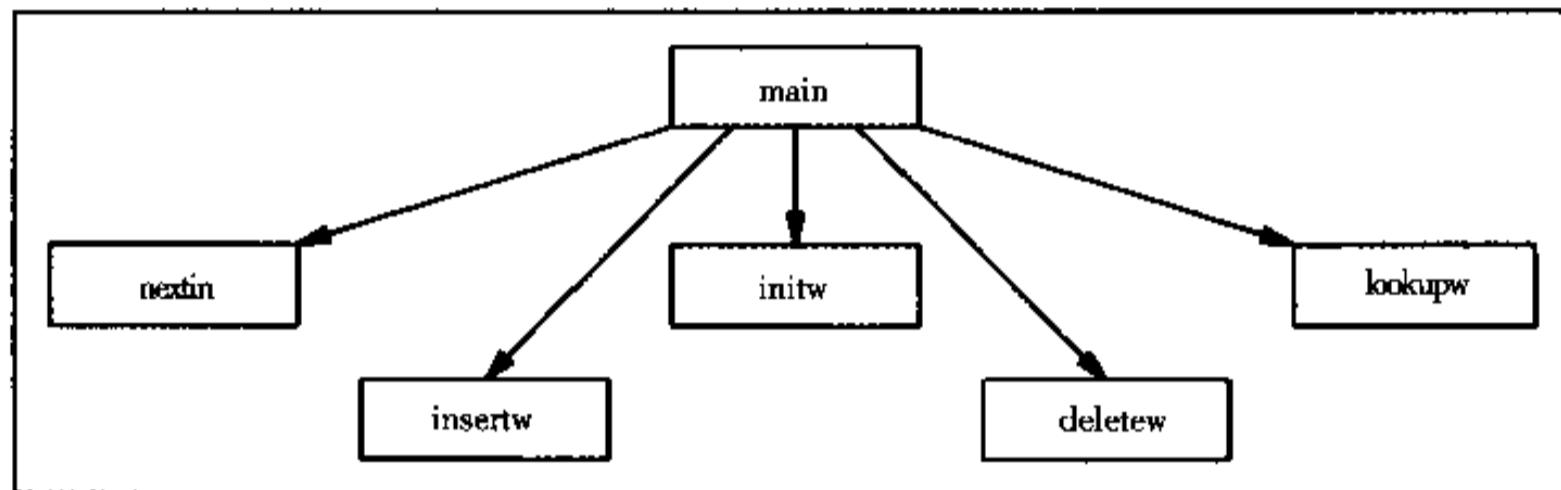


图 23.2 解决字典问题的最初的常规程序的过程调用图。调用图表示程序的过程的组织

在考虑哪个过程可以转移到远程机器上时，程序员必须考虑每个过程所需要的设施。例如，过程 nextin 在每次被调用时要读取下一输入行，并对它进行分析。因为它需要访问程序的标准输入文件，所以 nextin 必须放在主程序中。概括起来就是：

执行 I/O 或者访问文件描述符的过程不能轻易地转移到远程机器中。

程序员还必须考虑每个过程所要访问的数据所处的位置。例如，过程 lookupw 需要访问全部单词数据库。如果执行 lookupw 的机器不同于字典所处的机器，对 lookupw 的 RPC 调用就必须将整个字典作为参数来传递。

将巨大的数据结构作为参数传递给远程过程的效率非常低，因为 RPC 必须为每个远程过程调用对整个数据结构进行读取和编码。一般来说：

执行过程的机器应当与放置过程所要访问数据的机器是同一台。将巨大的数据结构传递给远程过程的效率很低。

在考虑了最初的字典应用程序以及每个过程所访问的数据之后，很明显应当把过程 insertw、deletew、initw、lookupw 和字典本身放到同一台机器中。

假设程序员决定将字典的存储以及相关的过程转移到一台远程机器中。为理解这样做的后果，程序员往往要对创建一个分布式程序和数据结构做到心中有数，甚至要做一个草案。图 23.3 说明了在将数据及对它的访问过程转移到一台远程机器后，这个字典应用程序的新结构。

像图 23.3 这样简单的图可以帮助程序员思考对程序的划分（划分为本地构件和远程构件）。程序员必须要考虑每个过程是否要访问数据以及它所需要的服务，还必须要考虑每个远程过程所要求

的参数以及在网上传送这些信息所带来的开销。另外，该图还能帮助程序员了解网络时延将如何影响程序的性能。

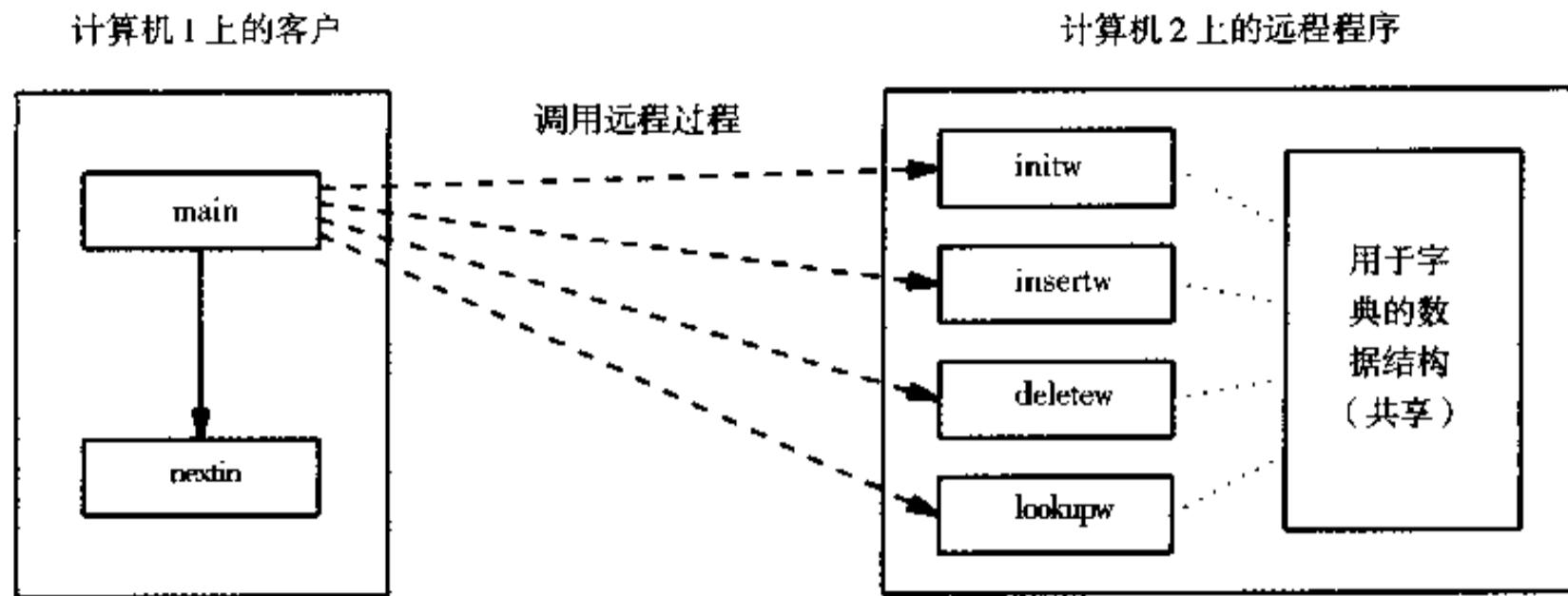


图 23.3 字典应用程序的概念性划分，它被划分成本地构件和远程构件。
远程构件包含字典的数据以及访问、查找这些数据的过程

假设程序员选择了某种概念划分，并决定按此进行，下一步就是将源程序分解为本地和远程两个构件。程序员明确每个构件所要使用的常量和数据结构，将每个构件放置到单独的文件当中。在本字典例子中，划分是简洁明了的，因为最初的源文件可以在过程 nextin 和 initw 之间进行划分。文件 dict1.c 含有主程序和过程 nextin：

```

/* dict1.c - main, nextin */

#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#define MAXWORD 50 /* maximum length of a command or word */

int nextin(char *cmd, char *word), initw(void), insertw(char *)
    deletew(char *), lookupw(char *);

/*
 * main - insert, delete, or look up words in a dictionary as specified
 */
int
main(int argc, char *argv[])
{
    char word[MAXWORD+1]; /* space to hold word from input line */
    char cmd;
    int wrdlen; /* length of input word */
    while (1)
        wrdlen = nextin(&cmd, word);
}

```

```
        if (wrflen < 0)
            exit(0);
        switch (cmd) {
            case 'I': /* "initialize" */
                initw();
                printf("Dictionary initialized to empty.\n");
                break;
            case 'i': /* "insert" */
                insertw(word);
                printf("%s inserted.\n", word);
                break;
            case 'd': /* "delete" */
                if (deletew(word))
                    printf("%s deleted.\n", word);
                else
                    printf("%s not found.\n", word);
                break;
            case 'l': /* "lookup" */
                if (lookupw(word))
                    printf("%s was found.\n", word);
                else
                    printf("%s was not found.\n", word);
                break;
            case 'q': /* quit */
                printf("program quits.\n");
                exit(0);
            default: /* illegal input */
                printf("command %c invalid.\n", cmd);
                break;
        }
    }

/*
 * nextin - read a command and (possibly) a word from the next input line
 */
int
nextin(char *cmd, char *word)
{
    int      i, ch;

    ch = getc(stdin);
    while (isspace(ch))
        ch = getc(stdin);
```

```
if (ch == EOF)
    return -1;
*cmd = (char) ch;
ch = getc(stdin);
while (isspace(ch))
    ch = getc(stdin);
if (ch == EOF)
    return -1;
if (ch == '\n')
    return 0;
i = 0;
while (!isspace(ch)) {
    if (++i > MAXWORD) {
        printf("error: word too long.\n");
        exit(1);
    }
    *word++ = ch;
    ch = getc(stdin);
}
return i;
}
```

文件 dict2.c 包含了来自最初的应用程序中的一些过程，它们将成为远程程序的一部分。另外，它还包含对各个过程要共享的全局数据的声明。在这里，文件并没有包含完整的程序——剩下的代码将在以后加上。

```
/* dict2.c - initw, insertw, deletew, lookupw */

#include <string.h>

#define MAXWORD 50      /* maximum length of a command or word */
#define DICTSIZ 100     /* maximum number of entries in dictionary. */
char dict[DICTSIZ][MAXWORD+1]; /* storage for a dictionary of words */
int nwords = 0;           /* number of words in the dictionary */
/*
 * initw - initialize the dictionary to contain no words at all
 */
int
initw()
{
    nwords = 0;
    return 1;
}
/*
 * insertw - insert a word in the dictionary
 */
```

```

/*
int
insertw(char *word)
char      *word;
{
    strcpy(dict[nwords], word);
    nwords++;
    return nwords;
}

/*
 * deletew - delete a word from the dictionary
 */
int
deletew(char *word)
{
    int      i;

    for (i=0 ; i<nwords ; i++)
        if (strcmp(word, dict[i]) == 0) {
            nwords--;
            strcpy(dict[i], dict[nwords]);
            return 1;
        }
    return 0;
}

/*
 * lookupw - look up a word in the dictionary
 */
int
lookupw(char *word)
{
    int      i;

    for (i=0 ; i<nwords ; i++)
        if (strcmp(word, dict[i]) == 0)
            return 1;
    return 0;
}

```

注意，对符号常量 MAXWORD 的定义在两个构件中都出现了，因为它们都要声明用于存储字的变量。然而，只有在文件 dict2.c 中才含有用于存储字典的数据结构的声明，因为只有远程过程才

包含字典的数据结构。

从实际的观点看,将应用程序分为两个文件使得分别编译客户和服务器成为可能。编译器检查诸如符号常量(双方都要引用的)之类的问题,而链接程序将检查所有数据结构是否同引用它们的过程结合到了一起。在Linux系统中,命令:

```
cc -c dict1.c  
cc -c dict2.c
```

产生两个构件的目标文件(并不是完整的程序)。这些构件必须链接到一起以产生可执行的程序，但是，这并不是编译它们的直接原因：编译器检查这两个文件是否语法正确。

在考虑让编译器检查代码这种方法时，我们要记住，大多数分布式程序要比我们的简单例子复杂得多。一次编译可能会发现一大堆问题，它们会转移程序员的注意力。在插入其他代码之前抓住这些问题可以使修改更为容易。

23.7 步骤 3：创建 rpcgen 规约

程序员一旦为某个分布式程序选择了一种结构，他或她就可以准备 rpc 规约了。从本质上说，`rpcgen` 规约文件包含了对远程程序的声明以及它所使用的数据结构。

该规约文件包含常量、类型定义以及对客户和服务器程序的声明。更明确地说就是，这个规约文件包含：

- 声明在客户或（这更经常）服务器（远程程序）中所使用的常量。
 - 声明所使用的数据类型（特别是对远程过程的参数）。
 - 声明远程程序、每个程序中所包含的过程以及它们的参数的类型。

我们知道 RPC 使用一些数字来标识远程程序以及在这些程序中的远程过程。在规约文件中的程序声明定义了诸如程序的 RPC 号、版本号以及分配给程序中的过程的编号等细节。

所有这些声明必须用RPC编程语言来给出，而不是用C。尽管它们的区别是微小的，但可能会成为障碍。例如，RPC用关键字string代表以null结束的字符串，而C却用char *来表示。因此，即使是一个有经验的程序员，要产生一个正确的规约，可能也需要多次反复。

文件 `rdict.x` 说明了一个 `rpcgen` 规约。它包含了对字典程序之 RPC 版的声明。

```
/* rdict.x */

/* RPC declarations for dictionary program */

const      MAXWORD = 50;           /* maximum length of a command or word */
const      DICTSIZ = 100;          /* number of entries in dictionary */

struct example {                  /* unused structure declared here to */
    int       exfield1;           /* illustrate how rpcgen builds XDR */
    char     exfield2;            /* routines to convert structures. */
};


```

```

/*
 * RDICTPROG - remote program that provides insert, delete, and lookup
 */
program RDICTPROG {                                /* name of remote program (not used) */
    version RDICTVERS {                            /* declaration of version (see below) */
        int INITW(void) = 1;                         /* first procedure in this program */
        int INSERTW(string) = 2;                      /* second procedure in this program */
        int DELETEW(string) = 3;                      /* third procedure in this program */
        int LOOKUPW(string) = 4;                      /* fourth procedure in this program */
    } = 1;                                         /* definition of the program version */
} = 0x30090949;                                    /* remote program number (must be unique)*/

```

一个`rpcgen`规约文件并没有囊括在最初的程序中所能找到的所有声明。它仅仅定义了那些在客户和服务器之间要共享的常量和数据类型，或者是那些需要指明的参数。

这个规约的例子由定义常量`MAXWORD`和`DICTSIZE`开始。在最初的应用中，这两个常量都是用C的预处理语句`define`定义的符号常量。RPC不使用C的符号常量声明，而是要求符号常量用关键字`const`声明，赋值时使用等号(=)。

按照约定，规约文件使用大写的名字来定义过程和程序。正如我们下面将看到的，这些名字会成为可以在C程序中使用的符号常量。在这里，并不绝对要求使用大写，但这样做有助于避免冲突。

23.8 步骤 4：运行 `rpcgen`

在完成了规约后，程序员运行`rpcgen`程序来检查语法错误，并且生成图22.6所示的四个代码文件^①。同大多数UNIX一样，该命令在Linux系统上的语法是：

```
rpcgen rdict.x
```

`rpcgen`在生成四个输出文件时，使用了输入文件的名字。例如，因为输入文件由`rdict`开始，所以输出文件将被命名为：`rdict.h`、`rdict_clnt.c`、`rdict_svc.c`和`rdict_xdr.c`。

23.9 `rpcgen`产生的.h文件

图23.4展示了文件`rdict.h`的内容，它包含了在规约文件中所声明的所有常量和数据类型的C的合法声明。另外，`rpcgen`还增加了对远程过程的定义。在这个例子代码中，`rpcgen`定义大写的`INSERTW`为2，因为规约中声明过程`INSERTW`为远程程序中的第二个过程。

这里需要解释一下的是`rdict.h`中的外部过程声明。这个被声明的过程构成了客户端的stub的接口部分。过程名取自已声明过的过程名，只是将它们映射为小写，并附加上一个下划线和程序的版本号。例如，我们的规约文件的例子声明：远程程序包含有过程`DELETEW`。于是，`rdict.h`含有

^① 如果某个特定的输出文件为空，`rpcgen`将不创建它。因此，有些规约文件产生的文件数少于四个。

对过程 `deletew_1` 的外部声明。为理解为什么 `rpcgen` 要声明这些接口例程，我们可以回顾 `stub` 接口部分的目的：它允许 `rpcgen` 选择自己的调用约定，而又可以让最初的调用过程保持不变。

作为接口 `stub` 命名的例子，让我们考虑过程 `insertw`。最初的过程将成为服务器的一部分，而且将保持不变。这样，服务器将具有名为 `insertw` 的过程，它和最初的应用程序具有相同的参数。为避免命名冲突，服务器必须为接口 `stub` 过程使用不同的名字。`rpcgen` 让服务器端的通信 `stub` 调用名为 `insertw_1` 的接口 `stub` 过程。该调用使用 `rpcgen` 所选择的参数，而且它允许程序员设计 `insertw_1` 以便能用正确的参数调用 `insertw`。

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#ifndef _RDICTIONARY_H_RPCGEN
#define _RDICTIONARY_H_RPCGEN

#include <rpc/rpc.h>

#ifdef __cplusplus
extern "C" {
#endif

#define MAXWORD 50
#define DICTSIZ 100

struct example {
    int exfield1;
    char exfield2;
};

typedef struct example example;

#define RDICTIONARYPROG 0x30090949
#define RDICTIONARYVERS 1

#if defined(__STDC__) || defined(__cplusplus)
#define INITW 1
extern int * initw_1(void *, CLIENT *);
extern int * initw_1_svc(void *, struct svc_req *);
#define INSERTW 2
extern int * insertw_1(char **, CLIENT *);
extern int * insertw_1_svc(char **, struct svc_req *);
#define DELETEW 3
extern int * deletew_1(char **, CLIENT *);
extern int * deletew_1_svc(char **, struct svc_req *);

```

```

#define LOOKUPW 4
extern int * lookupw_1(char **, CLIENT *);
extern int * lookupw_1_svc(char **, struct svc_req *);
extern int rdictprog_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);

#else /* K&R C */
#define INITW 1
extern int * initw_1();
extern int * initw_1_svc();
#define INSERTW 2
extern int * insertw_1();
extern int * insertw_1_svc();
#define DELETEW 3
extern int * deletew_1();
extern int * deletew_1_svc();
#define LOOKUPW 4
extern int * lookupw_1();
extern int * lookupw_1_svc();
extern int rdictprog_1_freeresult ();
#endif /* K&R C */

/* the xdr functions */

#if defined(__STDC__) || defined(__cplusplus)
extern bool_t xdr_example (XDR *, example*);

#else /* K&R C */
extern bool_t xdr_example ();

#endif /* K&R C */

#endif /* __cplusplus
*/
#endif /* !_RDICT_H_RPCGEN */

```

图 23.4 文件 rdict.h，由 rpcgen 所产生的文件的例子^①

23.10 rpcgen 产生的 XDR 转换文件

rpcgen 产生了含有对一些例程的调用的文件，这些例程执行 XDR 转换，而这种调用是针对远

^① 由于本章所示的代码是由 rpcgen 产生的，因而可能并不严格遵从 ANSI 标准，而且，代码开始时也没有标识该文件的注释。

程程序中所声明的所有数据类型的。例如，图23.5所示的文件rdict_xdr.c含有一些对转换例程的调用，这是为了转换字典程序中所声明的那些数据类型。

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "rdict.h"

bool_t
xdr_example(XDR *xdrs, example *objp)
{
    register long *buf;
    if (!xdr_int(xdrs, &objp->exfield1))
        return FALSE;
    if (!xdr_char(xdrs, &objp->exfield2))
        return FALSE;
    return TRUE;
}
```

图23.5 文件rdict_xdr.c，由rpcgen产生的XDR转换例程文件的例子

在我们的例子中，出现在规约文件里的惟一的类型声明被取名为example。它定义了一个结构，该结构具有一个整数字段和一个字符字段。文件rdict_xdr.c含有将结构example在本地数据表示和外部数据表示之间进行转换所需要的代码。这些代码是由rpcgen自动生成的，它为结构中的每个字段调用XDR库中的例程。一旦给出了声明，这个被声明的类型就可以用作给远程过程的参数。如果某个远程过程确实用结构example作为参数，rpcgen将在客户和服务器中生成代码，以便调用过程xdr_example对数据表示进行转换。

23.11 rpcgen产生的客户代码

对我们这个字典应用程序的例子，rpcgen产生了文件rdict_clnt.c，这是源程序，它将成为本程序分布式版中客户端的通信stub。

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <memory.h> /* for memset */
#include "rdict.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };
```

```
int *
initw_1(void *argp, CLIENT *clnt)
{
    static int clnt_res;

    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, INITW,
        (xdrproc_t) xdr_void, (caddr_t) argp,
        (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}

int *
insertw_1(char **argp, CLIENT *clnt)
{
    static int clnt_res;

    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, INSERTW,
        (xdrproc_t) xdr_wrapstring, (caddr_t) argp,
        (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}

int *
deletew_1(char **argp, CLIENT *clnt)
{
    static int clnt_res;

    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, DELETEW,
        (xdrproc_t) xdr_wrapstring, (caddr_t) argp,
        (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}
```

```
int *
lookupw_1(char **argp, CLIENT *clnt)
{
    static int clnt_res;

    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, LOOKUPW,
        (xdrproc_t) xdr_wrapstring, (caddr_t) argp,
        (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}
```

图 23.6 文件 rdict_clnt.c, rpcgen 创建的客户 stub 的例子

该文件为远程程序中的每个过程准备好了通信 stub 过程。像在服务器中一样，选择过程名字时有意避免了冲突。

23.12 rpcgen 产生的服务器代码

在我们这个字典应用的例子中，rpcgen 产生的第四个文件是 rdict_svc.c，它含有服务器所需要的代码。该文件包含服务器在开始时要执行的主程序。它包括获得协议端口号，向端口映射器注册 RPC 程序，接着便等待接收 RPC 调用。它将每个调用分派给合适的服务器端的 stub 接口。当被调用的过程响应时，服务器创建 RPC 应答并将它发回客户。

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "rdict.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif
```

```
static void
rdictprog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
    union {
        char *insertw_1_arg;
        char *deletew_1_arg;
        char *lookupw_1_arg;
    } argument;
    char *result;
    xdrproc_t _xdr_argument, _xdr_result;
    char *(*local)(char *, struct svc_req *);

    switch (rqstp->rq_proc) {
    case NULLPROC:
        (void) svc_sendreply(transp, (xdrproc_t) xdr_void, (char *)NULL);
        return;

    case INITW:
        _xdr_argument = (xdrproc_t) xdr_void;
        _xdr_result = (xdrproc_t) xdr_int;
        local = (char * (*)(char *, struct svc_req *)) initw_1_svc;
        break;

    case INSERTW:
        _xdr_argument = (xdrproc_t) xdr_wrapstring;
        _xdr_result = (xdrproc_t) xdr_int;
        local = (char * (*)(char *, struct svc_req *)) insertw_1_svc;
        break;

    case DELETEW:
        _xdr_argument = (xdrproc_t) xdr_wrapstring;
        _xdr_result = (xdrproc_t) xdr_int;
        local = (char * (*)(char *, struct svc_req *)) deletew_1_svc;
        break;

    case LOOKUPW:
        _xdr_argument = (xdrproc_t) xdr_wrapstring;
        _xdr_result = (xdrproc_t) xdr_int;
        local = (char * (*)(char *, struct svc_req *)) lookupw_1_svc;
        break;

    default:
        svcerr_noproc(transp);
        return;
    }
}
```

```
}

memset ((char *)&argument, 0, sizeof (argument));
if (!svc_getargs (transp, _xdr_argument, (caddr_t) &argument)) {
    svcerr_decode (transp);
    return;
}
result = (* local)((char *)&argument, rqstp);
if (result != NULL && !svc_sendreply(transp, _xdr_result, result)) {
    svcerr_systemerr (transp);
}
if (!svc_freeargs (transp, _xdr_argument, (caddr_t) &argument)) {
    fprintf (stderr, "unable to free arguments");
    exit (1);
}
return;
}

int
main (int argc, char **argv)
{
register SVCXPRT *transp;

pmap_unset (RDICTPROG, RDICTVERS);

transp = svcudp_create(RPC_ANYSOCK);
if (transp == NULL) {
    fprintf (stderr, "cannot create udp service.");
    exit(1);
}
if (!svc_register(transp, RDICTPROG, RDICTVERS, rdictprog_1, IPPROTO_UDP)) {
    fprintf (stderr, "unable to register (RDICTPROG, RDICTVERS, udp).");
    exit(1);
}

transp = svctcp_create(RPC_ANYSOCK, 0, 0);
if (transp == NULL) {
    fprintf (stderr, "cannot create tcp service.");
    exit(1);
}
if (!svc_register(transp, RDICTPROG, RDICTVERS, rdictprog_1, IPPROTO_TCP))
{
    fprintf (stderr, "unable to register (RDICTPROG, RDICTVERS, tcp).");
    exit(1);
}
```

```

    svc_run ();
    fprintf (stderr, "svc_run returned");
    exit (1);
    /* NOTREACHED */
}

```

图 23.7 文件 rdict_svc.c， rpcgen 产生的服务器 stub 的例子

该文件一旦生成，就可被编译并成为目标代码的形式。在 Linux 系统中，编译所有这三个文件（含有 rpcgen 所生成的代码）的命令是：

```

cc -c rdict_clnt.c
cc -c rdict_svc.c
cc -c rdict_xdr.c

```

每个命令用 C 源文件产生相应的目标文件。目标文件名用后缀 “.o” 取代 “.c” 后缀^①。例如，rdict_clnt.c 的编译后版本将放在文件 rdict_clnt.o 中。

23.13 步骤 5：编写 stub 接口过程

23.13.1 客户端接口例程

rpcgen 产生的文件并没有构成完整的程序。它还要求程序员必须编写客户端和服务器端的接口例程。在远程程序中的每个远程过程都必须存在一个接口过程。

在客户端，原来的主应用程序控制着处理的进行。它调用接口过程，而它所使用的过程名和参数类型与最初那个程序（非分布式版）调用原过程所使用的完全一样，在分布式版本中，原来这些过程已成为远程的了。每个接口过程都必须要将它的参数转换为 rpcgen 所使用的形式，还必须接着调用相应的客户端的通信过程。例如，因为最初的程序含有叫做 insertw 的过程，它使用一个指向字符串的指针作为参数，客户端接口必须也含有这样一个过程。该接口过程调用 insertw_1，这是 rpcgen 生成的客户端的通信 stub。

常规的过程参数与通信 stub 所使用的参数间的主要不同在于： rpcgen 生成的所有过程的参数都使用间接方式。例如，如果最初的过程有一个整数参数，那么，通信 stub 中该过程的相应的参数必须是指向整数的指针。在字典程序中，多数过程要求字符串参数，它在 C 中用字符指针（char*）来声明。在相应的通信 stub 中，都要求它们的参数是指向字符指针的指针（char**）。

文件 rdict_cif.c 展示了接口例程如何将参数转换为 rpcgen 产生的代码所期望的形式。对程序中的每个远程过程，文件中都包含客户端的接口过程。

```

/* rdict_cif.c - initw, insertw, deletew, lookupw */

#include <rpc/rpc.h>

#include <stdio.h>

```

^① 程序员有时把目标文件指做 “dot o” 文件。

```
#define RPC_CLNT
#include "rdict.h"

/* Client-side stub interface routines written by programmer */

extern CLIENT      *handle;          /* handle for remote procedure */
static int         *ret;             /* tmp storage for return code */

/*-----
 * initw - client interface routine that calls initw_1
 *-----
 */
int
initw()
{
    ret = initw_1(0, handle);
    return ret==0 ? 0 : *ret;
}

/*-----
 * insertw - client interface routine that calls insertw_1
 *-----
 */
int
insertw(char *word)
{
    char **arg;                      /* pointer to argument */
    arg = &word;
    ret = insertw_1(arg, handle);
    return ret==0 ? 0 : *ret;
}

/*-----
 * deletew - client interface routine that calls deletew_1
 *-----
 */
int
deletew(char *word)
{
    char **arg;                      /* pointer to argument */
    arg = &word;
    ret = deletew_1(arg, handle);
    return ret==0 ? 0 : *ret;
}
```

```

}

/*
 * lookupw - client interface routine that calls lookupw_1
 */
int
lookupw(char *word)
{
    char **arg;                      /* pointer to argument */

    arg = &word;
    ret = lookupw_1(arg, handle);
    return ret==0 ? 0 : *ret;
}

```

23.13.2 服务器端接口例程

在服务器端，接口例程接受来自 rpcgen 所产生的通信 stub 的调用，并将控制权传递给实现这个指定调用的过程。如同客户端那样，服务器端接口例程必须把参数由 rpcgen 所选择的类型转变为被调用过程所使用的类型。在大多数场合，这种差异在于一种间接方式——rpcgen 传递的是指向对象的指针，而不是对象本身。为转换一个参数，接口过程只需要使用 C 的间接运算符 (*)。文件 rdict_sif.c 展示了这个概念，它包含了字典程序的服务器端的接口例程。

```

/* rdict_sif.c - init_1, insert_1, delete_1, lookup_1_svc */

#include <rpc/rpc.h>

#define RPC_SVC
#include "rdict.h"

/* Server-side stub interface routines written by hand */

static int retcode;

int      initw(void), insertw(char *), deletew(char *), lookupw(char *);

/*
 * insertw_1_svc - server side interface to remote procedure insertw
 */
int      *
insertw_1_svc(char **w, struct svc_req *rqstp)
{
    retcode = insertw(* (char **) w);

```

```
        return &retcode;
    }

/*
 * initw_1_svc - server side interface to remote procedure initw
 */
int
initw_1_svc(void *w, struct svc_req *rqstp)
{
    retcode = initw();
    return &retcode;
}

/*
 * deletew_1_svc - server side interface to remote procedure deletew
 */
int
deletew_1_svc(char **w, struct svc_req *rqstp)
{
    retcode = deletew(* (char **) w);
    return &retcode;
}

/*
 * lookupw_1_svc - server side interface to remote procedure lookupw
 */
int
lookupw_1_svc(char **w, struct svc_req *rqstp)
{
    retcode = lookupw(* (char**) w);
    return &retcode;
}
```

23.14 步骤 6：编译和链接客户程序

在客户接口例程编写完成并放入到源文件后，它们就可以被编译了。例如，文件 rdict_cif.c 包含了字典例子中所有的接口例程。在 Linux 系统中，编译出结果所需要的命令是：

```
cc -c rdict_cif.c
```

编译器产生输出文件 rdict_cif.o。为完成客户，程序员需要在最初的主程序中加入一点新的细节，因为新版本使用了 RPC，程序需要有针对 RPC 声明的 C 的 include 文件。它还需要包含文件 rdict.h，这是因为该文件包含了客户和服务器都要使用的常量的定义。

客户程序还需要声明并初始化一个句柄（handle），RPC 通信例程用该句柄和服务器通信。多数客户使用已定义的类型CLIENT 声明这个句柄，并且通过调用 RPC 库例程 clnt_create 来初始化这个句柄。文件 rdict.c 作为例子展示了这些必要的代码。

```
/* rdict.c - main, nextin */

#include <rpc/rpc.h>

#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

#include "rdict.h"

#define MAXWORD 50 /* maximum length of a command or word */
#define RMACHINE "localhost" /* name of remote machine */
CLIENT *handle; /* handle for remote procedure */

int nextin(char *cmd, char *word), initw(), insertw(char *word);
int deletew(char *word), lookupw(char *word);

/*
 * main - insert, delete, or look up words in a dictionary as specified
 */
int
main(int argc, char *argv[])
{
    char word[MAXWORD+1]; /* space to hold word from input line */
    char cmd;
    int wrdlen; /* length of input word */

    /* set up connection for remote procedure call */

    handle = clnt_create(RMACHINE, RDICTPROG, RDICTVERS, "tcp");
    if (handle == 0) {
        printf("Could not contact remote program.\n");
        exit(1);
    }

    while (1) {
        wrdlen = nextin(&cmd, word);
        if (wrdlen < 0)
            exit(0);
    }
}
```

```
word[wrdlen]='\0';
switch (cmd) {
    case 'I':           /* "initialize"      */
        initw();
        printf("Dictionary initialized to empty.\n");
        break;
    case 'i':           /* "insert"          */
        insertw(word);
        printf("%s inserted.\n", word);
        break;
    case 'd':           /* "delete"          */
        if (deletew(word))
            printf("%s deleted.\n", word);
        else
            printf("%s not found.\n", word);
        break;
    case 'l':           /* "lookup"          */
        if (lookupw(word))
            printf("%s was found.\n", word);
        else
            printf("%s was not found.\n", word);
        break;
    case 'q':           /* quit              */
        printf("program quits.\n");
        exit(0);
    default:             /* illegal input     */
        printf("command %c invalid.\n", cmd);
        break;
}
}

/*
 * nextin - read a command and (possibly) a word from the next input line
 */
int
nextin(char *cmd, char *word)
{
    int      i, ch;

    ch = getc(stdin);
    while (isspace(ch))
        ch = getc(stdin);
    if (ch == EOF)
```

```

        return -1;
    *cmd = (char) ch;
    ch = getc(stdin);
    while (isspace(ch))
        ch = getc(stdin);
    if (ch == EOF)
        return -1;
    if (ch == '\n')
        return 0;
    i = 0;
    while (!isspace(ch)) {
        if (++i > MAXWORD) {
            printf("error: word too long.\n");
            exit(1);
        }
        *word++ = ch;
        ch = getc(stdin);
    }
    return i;
}

```

比较 rdict.c 和 dict1.c^①，可以看出，我们只加入了很少的代码。这个例子代码使用符号常量 RMACHINE 来指明远程机器的域名。为使测试容易，RMACHINE 已定义为 localhost，这意味着客户和服务器将在同一台机器中运行。当然，对分布式程序的测试一旦完成，程序员将改变定义，使它指定为服务器的永久性位置。

clnt_create 尝试向某台指定的远程机器建立连接。如果连接的尝试失败，clnt_create 返回值 NULL，这使应用程序向用户报告出现差错。如果 clnt_create 报告了出现差错，我们的代码将退出。在实际中，客户可以重试，或者维护机器表，试着对表中的各台机器进行连接。

就像其他 C 源文件，rdict.c 可以用命令编译：

```
cc -c rdict.c
```

在 rdict.c 的目标程序被编译出来后，构成客户的所有文件可以被链接成可执行的程序。Linux cc 命令为.o 文件调用的链接程序（linker）。在调用时可以加上 -o 选项，以便将输出放到被命名的文件中。例如，下面的命令行将.o 文件链接到一起，并将最终的可执行客户程序放到文件 rdict 中。

```
cc -o rdict rdict.o rdict_clnt.o rdict_xdr.o rdict_cif.o
```

23.15 步骤 7：编译和链接服务器程序

rpcgen 所生成的输出包含了服务器所需要的大多数代码。程序员提供了两个附加的文件：服务器接口例程（我们已将它放在文件 rdict_sif.c 中）和远程过程本身。对字典例子来说，远程过程的

^① 文件 dict1.c 可在第 225 页找到。

最终版本出现在文件 rdict_srp.c 中。这些过程的代码来自最初的应用。

```
/* rdict_srp.c - initw, insertw, deletew, lookupw */

#include <rpc/rpc.h>

#include <string.h>

#include "rdict.h"

/* Server-side remote procedures and the global data they use */

char      dict[DICTSIZ][MAXWORD+1]; /* storage for a dictionary of words */
int       nwords = 0;                /* number of words in the dictionary */

/*-----
 * initw - initialize the dictionary to contain no words at all
 *-----
 */
int
initw()
{
    nwords = 0;
    return 1;
}

/*-----
 * insertw - insert a word in the dictionary
 *-----
 */
int
insertw(char *word)
{
    strcpy(dict[nwords], word);
    nwords++;
    return nwords;
}

/*-----
 * deletew - delete a word from the dictionary
 *-----
 */
int
deletew(char *word)
{
    int      i;

    for (i=0 ; i<nwords ; i++)
```

```

        if (strcmp(word, dict[i]) == 0) {
            nwords--;
            strcpy(dict[i], dict[nwords]);
            return 1;
        }
    return 0;
}

/*
 * lookupw - look up a word in the dictionary
 */
int
lookupw(char *word)
{
    int      i;

    for (i=0 ; i<nwords ; i++)
        if (strcmp(word, dict[i]) == 0)
            return 1;
    return 0;
}

```

远程过程用如下命令编译：

```
cc -c rdict_srp.c
```

一旦编译完成，构成服务器的这些目标程序就可以用如下命令链接成可执行的文件：

```
cc -o rdictd rdict_svc.o rdict_xdr.o rdict_sif.o rdict_srp.o
```

rdict_srp.o 在运行该命令时，它将可执行代码放到文件 rdictd 中^①。

23.16 步骤 8：启动服务器和执行客户

为对整个系统进行第一次测试，要让客户和服务器运行于同一台机器中。在客户试图联系服务器之前，服务器必须开始执行，否则，客户就会显示消息：

```
Could not contact remote program.(无法联系远程程序)
```

并停止执行。在 Linux 系统中，用命令：

```
./rdictd &
```

^① 如同多数 UNIX 系统，Linux 命名服务器时，常常为其加上一个尾巴 "d"，它代表守护进程 daemon，这一技术术语适用于所有运行在后台的程序。

```

${CC} ${CFLAGS} -o dict dict.c
#
# Test division into two parts to see if they compile
#
dict1.o:      dict1.c
              ${CC} ${CFLAGS} -c dict1.c
dict2.o:      dict2.c
              ${CC} ${CFLAGS} -c dict2.c

#
# Dependencies for files generated by rpcgen
#
${GFILES}:    rdict.h rdict.c
rdict.h:       rdict.x
              rpcgen rdict.x

#
# Compile and link client-side files for RPC version
#
${RDDCT_OBJ}:
              ${cc} ${CFLAGS} -c $*.c
rdict:        ${RDICT_OBJ} rdict_xdr.o
              ${CC} ${CFLAGS} -o $0 ${RDICT_OBJ} rdict_xdr.o
              chmod 755 rdict
#
# Compile and link server-side files for RPC version
#
${RDICTD_OBJ}:
              ${cc} ${CFLAGS} -c $*.c
rdictd:       ${RDICTD_OBJ} rdict_xdr.o
              ${CC} ${CFLAGS} -o $0 ${RDICTD_OBJ} rdict_xdr.o

#
# Compile XDR definitions
#
rdict_xdr.o:
              ${CC} ${cflags} -c $*.c

```

Makefile 一旦建立，对客户和服务器的重新编译就完全是自动的了。例如，如果在某个目录下包含了 rpcgen 规约文件、Makefile 以及本章所描述的所有源文件，在该目录下调用 make 便会执行以下命令序列：

```
cc -o dict dict.c
```

```
cc -c dict1.c
cc -c dict2.c
rpcgen rdict.x
cc -c rdict_clnt.c
cc -c rdict_cif.c
cc -c rdict.c
cc -c rdict_xdr.c
cc -o rdict rdict_clnt.o rdict_cif.o rdict.o rdict_xdr.o
chmod 755 rdict
cc -c rdict_svc.c
cc -c rdict_sif.c
cc -c rdict_srp.c
cc -o rdictd rdict_svc.o rdict_sif.o rdict_srp.o rdict_xdr.o
chmod 755 rdictd
```

这样，运行 make 便产生了可执行的客户 rdict 以及可执行的服务器 rdictd。

23.18 小结

用 rpcgen 构造分布式程序由八个步骤组成。程序员由解决该问题的常规应用程序开始，决定如何将程序划分为在本地构件和远程执行构件，将应用划分成两个物理的部分，创建描述远程过程的规约文件，运行 rpcgen 产生所需要的文件。接着程序员编写客户端和服务端的接口例程，并将它们与 rpcgen 所产生的代码相结合。最后，程序员编译并链接客户端的文件和服务端的文件，以便产生可执行的客户和服务程序。

尽管 rpcgen 免除了 RPC 所要求的大部分编写代码的工作，但编写分布式程序还是要认真思考。在考虑如何将程序划分成本地构件和远程构件时，程序员必须检查被每个程序片段所要访问的数据，以便使数据的移动最小。程序员还必须要考虑每个远程过程所引入的延时，以及各个程序片段将如何访问 I/O 设施。

本章所给的字典应用的例子说明将简单的应用转变成分布式的程序要花费很多的精力。更复杂的应用程序要求更加复杂的规约和接口过程。特别是，那些将结构化的数据传递给远程过程的应用，或者那些检查客户授权的应用可能要求相当多的代码。

深入研究

关于 rpcgen 的其他信息可以在与软件附带的文档中找到。Linux 提供的联机文档描述了 make 实用程序的细节。

习题

- 23.1 修改本章所给的例子程序，使客户接口例程保持最近引用的单词的高速缓存，并且在进行远程调用前先查询高速缓存。高速缓存要求多少额外的计算负担？当某个条目可以

在本地找到时，高速缓存可以节省多少时间？

- 23.2 构建一个分布式应用，提供对远程机器上的文件的访问。其中要包含这样的远程过程，它们允许客户在某个指定文件中的某个指定位置读或写数据。
- 23.3 构建一个分布式程序，将链表作为参数传递给远程过程。提示：使用相对指针以取代绝对的存储器地址。
- 23.4 试修改字典程序的例子，使远程过程可以将出错消息写到客户的标准出错文件中。对此，你会遇到什么问题？是如何解决的？
- 23.5 rpcgen 可能已设计成这个样子，即自动为每个远程过程分配一个惟一的号，如 1、2 等等。如果让程序员在规约文件中手工分配远程过程号，以此取代自动分配，这样做的优点是什么？缺点呢？
- 23.6 rpcgen 有哪些限制？提示：考虑试图构建一个服务器，它同时也是另一个服务的客户。
- 23.7 仔细查看 rdict_clnt.c 中的代码。在客户开始执行时，它采取了什么步骤？
- 23.8 试同时构建并测试分布式字典程序的两个版本。如果服务器端使用了熟知端口号，还是否有可能测试新程序版本？解释原因。
- 23.9 字典程序的例子把空格认作输入的分隔字符。修订过程 nextin 使允许有制表符和空格。
- 23.10 是否有可能使字典程序的服务器端成为并发的？为什么？

第 24 章 网络文件系统 (NFS) 的概念

24.1 引言

前面讲述了ONC RPC工具，解释了RPC与客户-服务器之间的关系，还说明了如何用RPC生成应用程序的分布式版本。本章及下一章将集中讨论一种应用程序及一个协议，该协议是用RPC来描述、设计和实现的。本章描述远程文件存取(file access)的一般概念，并对一种特殊的远地文件存取机制的基本概念做了回顾。因为这种机制的许多思想和细节产生于UNIX，所以本章还对Linux的文件系统及文件操作的语义进行简单回顾。本章讨论了层次目录结构和路径名，还讨论了一个远程文件存取机制如何实现分层的操作。我们在下一章中给出有关于该协议的其他细节，并展示该协议规范如何使用RPC来定义远程文件操作。

24.2 远程文件存取和传送

许多早期的网络系统提供了文件传送(file transfer)服务，它允许用户把某文件的一个副本从一台机器移到另一台机器上。而更多近期的网络则提供了文件存取(file access)服务，即允许某个应用程序从一台远程机器上对某个文件进行存取。远程文件存取机制为每个文件保存了一个副本，并允许一个或多个应用程序在需要时对此副本进行存取。

使用远程文件机制对文件进行存取的应用程序，可以同它所要存取的文件驻留在同一台机器上，也可以运行在某台远程机器上。当一个应用程序要存取一个驻留在某台远程机器上的文件时，该程序的操作系统将调用客户软件，这个客户软件与远程机器中的服务器联系，对该文件执行所请求的操作。与文件传输服务不同，该应用的系统并不立刻对整个文件进行检索或存储，而是每次要求传送一小块数据。

为了提供对驻留在某台计算机上的某些或所有文件的远程存取能力，系统管理员必须让该计算机运行一个可以对存取请求进行响应的服务器。这个服务器对每个请求进行检查，以验证该客户是否具有对指定文件进行存取的权利，然后执行所指明的操作，最后向客户返回一个结果。

Sun Microsystems公司定义了一个在整个计算机工业中被广泛接受的远程文件存取机制，它称为Sun的网络文件系统(Network File System)，或者简称为NFS。该机制允许在一台计算机上运行一个服务器，使得对其上的某些或所有文件都可以进行远程存取，还允许其他计算机上的应用程序对这些文件进行存取^①。

^① 本书中描述的概念和许多细节可以应用于所有NFS协议版本。尽管本书出版时，Linux仍然使用版本2，但我们在讨论所有细节时都引用版本3，因为Linux不久将移植到版本3上去。

24.3 对远程文件的操作

NFS对远程文件提供了与本地文件相同的操作。一个应用程序可以打开(*open*)一个远程文件以进行存取，可以从该文件中读取(*read*)数据，向该文件写入(*write*)数据，定位(*seek*)到文件中的某个指定位置(开始、结尾或者其他地方)，最后当使用完毕后关闭(*close*)该文件。

24.4 异构计算机之间的文件存取

提供远程文件存取并不简单。除了基本的读写文件机制以外，文件存取服务还必须提供创建和销毁文件、读取目录、鉴别请求、认可文件保护以及对各种各样计算机表示之间的信息转换。因为远程文件存取服务把两台机器连接进来，所以它必须解决客户和服务器系统的文件命名方式的差异，用目录来表示路径，并且保存文件的有关信息。

NFS的设计是适合异构计算机系统的。从一开始，NFS的协议、它的操作以及语义的选择都是为了方便多种计算机系统的交互。当然，NFS无法在所有操作系统上提供所有文件系统的功能。它只是试图定义一些文件操作来尽可能多地适应不同的系统，同时还要保证它的效率和较低的复杂度。在实践中，这些选择的绝大多数都是可行的。

24.5 无状态服务器

NFS设计在客户端保存状态信息，但允许服务器处于无状态(*stateless*)。因为服务器是无状态的，所以服务的崩溃不会影响到客户。例如，从理论上说，在无状态服务器崩溃并重新启动后，客户仍然能够继续对文件进行存取；而运行于客户端的应用程序对服务器的重新启动却一无所知。而且，因为无状态服务器不必为每个客户分配资源，所以这种无状态设计方案可比有状态设计方案处理更多的客户。

NFS的这种无状态服务器设计方案，既影响了协议又影响了它的实现。最重要的是，不管是在文件还是在目录中，服务器没有任何位置(*position*)的概念。在研究了NFS所提供的操作之后，我们会看到NFS是如何进行无状态设计的。

24.6 NFS 和 UNIX 的文件语义

尽管NFS是设计用来适应异构文件系统的，但是原来的UNIX^①文件系统对它的整体设计、名词以及许多协议细节都产生了很大的影响。NFS的设计者在定义每个操作的意义时采纳了UNIX文件系统的语义。因此，为了理解NFS，我们要从UNIX的文件系统开始。

第3章有一个对UNIX I/O的简单描述；下面将把该描述展开。我们将讨论UNIX下的文件存储和存取，重点讨论与NFS有关的概念和细节。在后面，我们还将说明NFS是如何直接从UNIX文件系统借用了某些概念，它对许多细节做了微小修改后便加以采纳。总而言之：

理解UNIX的文件系统是理解NFS的基础，因为NFS使用了UNIX文件系统的名词和语义。

① 尽管本章引用原来的UNIX系统，但所有的概念和名词都同样适用于Linux。

24.7 UNIX 文件系统的回顾

这一节首先对 UNIX 文件系统中与 NFS 有关的概念、名词和细节进行简单回顾，然后解释 NFS 如何把这些概念具体化。

24.7.1 基本定义

从用户的观点来看，UNIX 定义一个文件 (file) 包含一个字节序列。理论上，UNIX 文件可以任意长；在实际当中，一个文件的长度是要受到物理存储设备空间限制的。UNIX 文件可以动态地增长。文件系统不能对文件大小进行预定义或对文件的空间进行预分配。而且，只要应用程序向文件中写入任何数据，文件都会自动增长。

从概念上说，UNIX 从零开始对文件中的字节进行计数。在任何时候，文件的大小 (size) 都被定义成它所包含的字节数。UNIX 文件系统允许用字节数作为基准对文件进行随机存取。它允许应用程序移到文件中的任意位置并对该位置上的数据进行存取。

24.7.2 无记录界限的字节序列

每个 UNIX 文件都是一个字节序列；除了数据本身以外，系统并不提供任何其他的结构。具体地讲，UNIX 没有记录界限 (record boundary)、记录分块 (record blocking)、索引文件 (indexed file) 或类型文件 (typed file) 的概念。当然，有可能某个应用程序会创建记录文件并在后来对它进行存取。关键是文件系统本身并不理解文件的内容；使用某文件的应用程序必须认同该格式。

24.7.3 文件拥有者和组标识符

UNIX 为多用户提供账号，并为每个用户分配一个数值型的用户标识符 (user identifier)，该标识符用于系统中的记账和鉴别。每个 UNIX 文件都有一个文件拥有者 (owner)，它用创建该文件的用户的数值标识符来表示。所有权信息是与文件存储在一起的（即与目录系统相反）。

除了用户标识符以外，UNIX 通过系统管理员为用户组分配一个数值型的组标识符 (group identifier)，以便在用户组之间能提供文件共享。在任何时候，某用户可以属于一个或多个 UNIX 组。当用户运行一个应用程序时（例如，一个电子表格程序或一个文本编辑器），运行的程序继承了用户的文件拥有者标识符和组标识符。每个 UNIX 文件都属于一个组，并在它内部存有组的数值标识符。系统把存储在文件中的文件拥有者标识符及组标识符与特定进程的用户及组标识符进行比较，以此来决定该进程可对这个文件进行何种操作。

24.7.4 保护和存取

UNIX 存取保护机制允许文件拥有者分别控制文件拥有者、文件组成员以及所有其他用户对文件的存取。保护机制允许文件拥有者指明每个集合中的用户所允许的读、写或执行文件的权限。图 24.1 显示可把 UNIX 文件存取权限看成一个保护比特矩阵 (matrix of protection bits)。

在内部，UNIX 系统把文件存取保护矩阵编码在一个二进制整数的低位比特中，在使用这个对保护位编码的整数时，UNIX 使用了名词文件模式 (file mode) 和文件存取模式 (file access mode)。图 24.2 说明了 UNIX 是如何把文件保护位编码在一个文件模式整数的 9 个低位比特中。除了图 24.2 中所示的保护比特之外，UNIX 还定义了模式整数的附加比特来指明文件的其他属性（例如，模式

比特指明文件是一个正规的文件还是一个目录)。

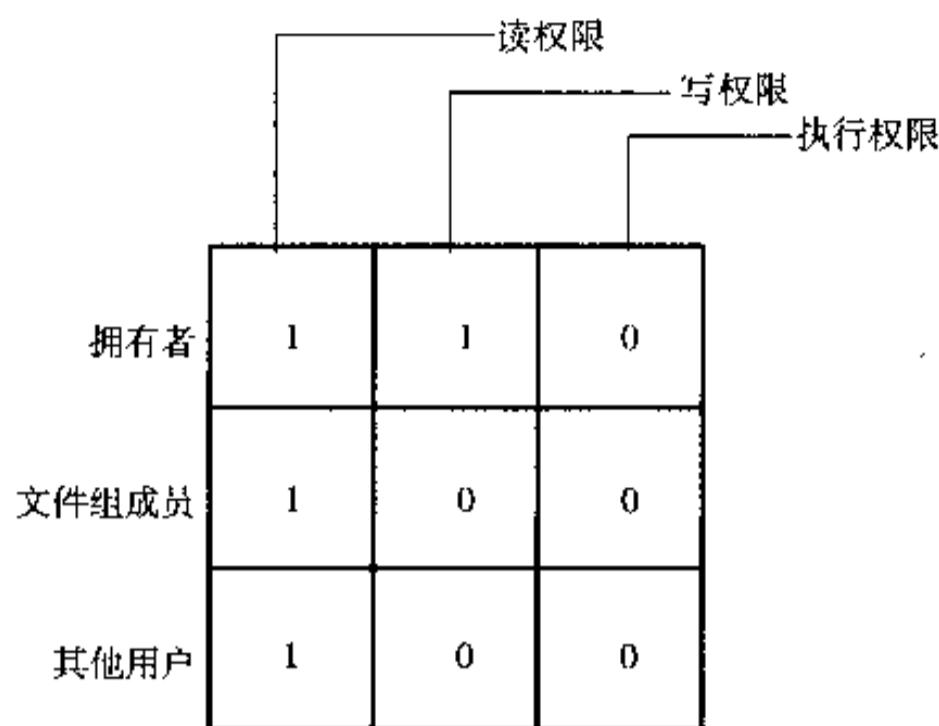


图 24.1 可以把文件存取权限看成是一个保护比特的矩阵

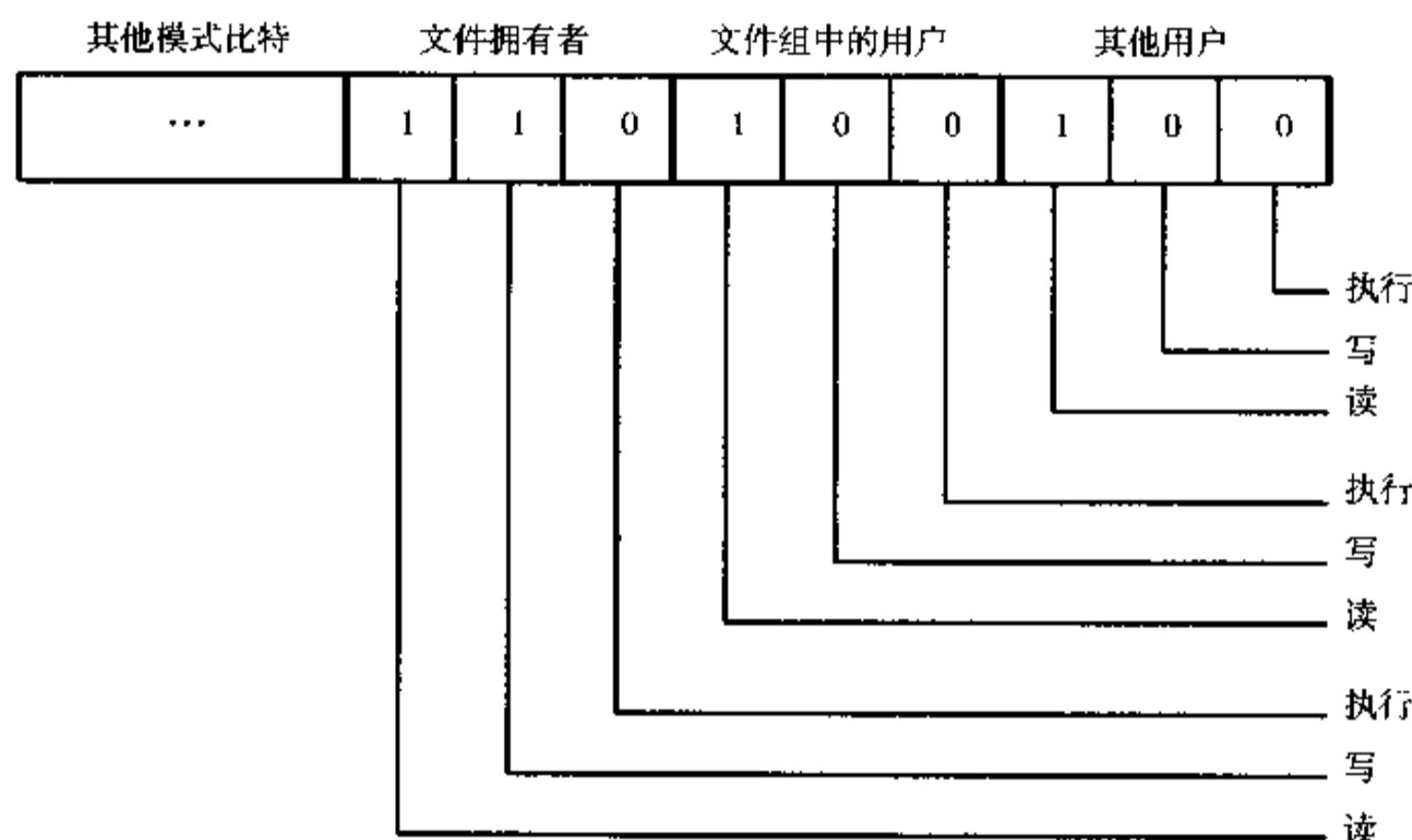


图 24.2 保存在模式整数 9 个低位比特的 UNIX 文件存取权。值 1 是允许；
值 0 是不允许的。如果写成八进制，则所显示的保护的值为 0644

当保护整数的值写成八进制时，最右边三个数字给出了文件拥有者、文件组成员和其他用户的保护位。因此，保护值 0700 指定文件拥有者可以读、写或者执行文件，但其他用户不能对它进行访问。保护模式值 0644 则指定所有用户都可以读该文件，但只有文件拥有者才能对文件进行写操作。

24.7.5 打开 - 读 - 写 - 关闭范例

在 UNIX 中，应用程序使用打开 - 读 - 写 - 关闭 (open-read-write-close) 方法对文件进行存取。为了对某个文件进行存取，应用程序必须调用函数 open，传给它文件的名和一个参数，该参数描述

所要进行的存取。`open` 返回一个整数文件描述符，应用程序用它对该文件进行更进一步的操作。例如下面的代码：

```
fdesc = open("filename", O_CREAT | O_RDWR, 0644);
```

它打开一个名为 `filename` 的文件。第二个参数 `O_CREAT` 的值指明如果文件不存在就创建它，`O_RDWR` 的值指明文件的创建必须是可读可写的。八进制数 `0644` 指明如果文件被创建就把保护模式位赋给它。第二个参数的其他值可用来指明文件是否应被截断或者文件的打开是否可读、可写，或者两者都可以。

24.7.6 数据传送

应用程序用 `read` 把数据从文件传输到内存中，用 `write` 把数据从内存传输到文件。`read` 函数有三个参数：所打开文件的描述符、一个缓冲区地址以及所要读的字节数。例如，下面的代码要求系统从描述符为 `fdesc` 的文件中读取 24 字节的数据：

```
n=read(fdsc, buff, 24);
```

`read` 和 `write` 从文件当前位置开始传送，当这两种操作结束时它们都会对文件位置做了修改。例如，如果一个应用程序打开一个文件，文件位置移到 0，应用程序从文件中读出 10 字节数据，则在 `read` 操作之后，文件位置将是 10。因此，一个程序从位置 0 开始重复调用 `read`，它就可以把一个文件中的所有数据都顺序地提取出来。

如果程序要读的字节数比文件所包含的多，则 `read` 函数把文件中的所有数据读出后返回它所读取的个数。如果在调用 `read` 时文件的位置在文件的末尾，则 `read` 调用返回 0 值以表示文件结束 (`end-of-file`) 的情况。

24.7.7 搜索目录权限

UNIX 用目录 (`directory`)^① 把文件按一种层次结构 (`hierarchy`) 进行组织，目录中有文件和其他目录。系统用同样的 9 位保护模式来描述目录，与一般的文件一样。`read` 权限位决定一个应用程序能否得到某个目录中所有文件列表，`write` 权限位决定一个应用程序能否向该目录中插入或删除文件。每个文件都有各自的权限比特集合，用以决定对文件的内容可进行何种操作。目录权限仅指定了对目录本身可进行的操作。

目录中可以有应用程序。但是它们不包含已编译的代码本身。因此，它们不能像一个应用程序一样执行，所以对目录来说，执行 (`execute`) 权限是没有意义的。UNIX 把目录的执行权限比特解释成搜索 (`search`) 权限。如果某个应用程序具有搜索权限，它就能够引用该目录中的文件；否则，系统不允许对该文件的任何引用。搜索权限可以用来隐藏或打开文件分层结构中的某一整个分支，同时不对该分支中任何一个文件的权限加以修改。

24.7.8 随机存取

当打开一个文件时，可以把文件位置设置到文件的开始（例如，顺序地存取文件）或到文件的

^① 其他系统有时用名词文件夹 (`folder`) 来表示 UNIX 中的目录 (`directory`)。

末尾（例如，向一个现有的文件追加数据）。在打开一个文件后，调用函数 lseek 可以改变文件的位置。lseek 有三个参数：文件描述符、偏移量和偏移度量。第二个参数使程序可以指定偏移量给出的是文件中的一个绝对位置（例如，在字节 512 处），还是一个相对于当前位置的位置（例如，离当前位置 64 的字节处），或是相对于文件末尾的位置（例如，在文件末尾的前 2 两个字节）。例如，常数 L_SET 指明系统应把偏移解释成为绝对值。因此，调用：

```
lseek(fd, 100L, L_SET);
```

指明了当前文件位置的描述符 fd 必须移到字节数 100 处。

24.7.9 搜索超过文件的结束

UNIX 文件系统允许程序移动到文件的任何位置，即使所指定的位置超过当前文件结束的地方。如果应用程序定位到一个确实存在的字节位置并写入新的数据，则新的数据取代该位置上的旧数据。如果程序定位超过文件结束的地方并写入新的数据，则文件系统扩展文件的长度。从用户的观点来看，系统将用空字节（值为零的字节）来填补已有数据与新数据之间的所有空隙。如果程序后来试图从填补空隙的空字节处开始读数据，则文件系统将返回内容值为零的字节。图 24.3 说明了这个概念。

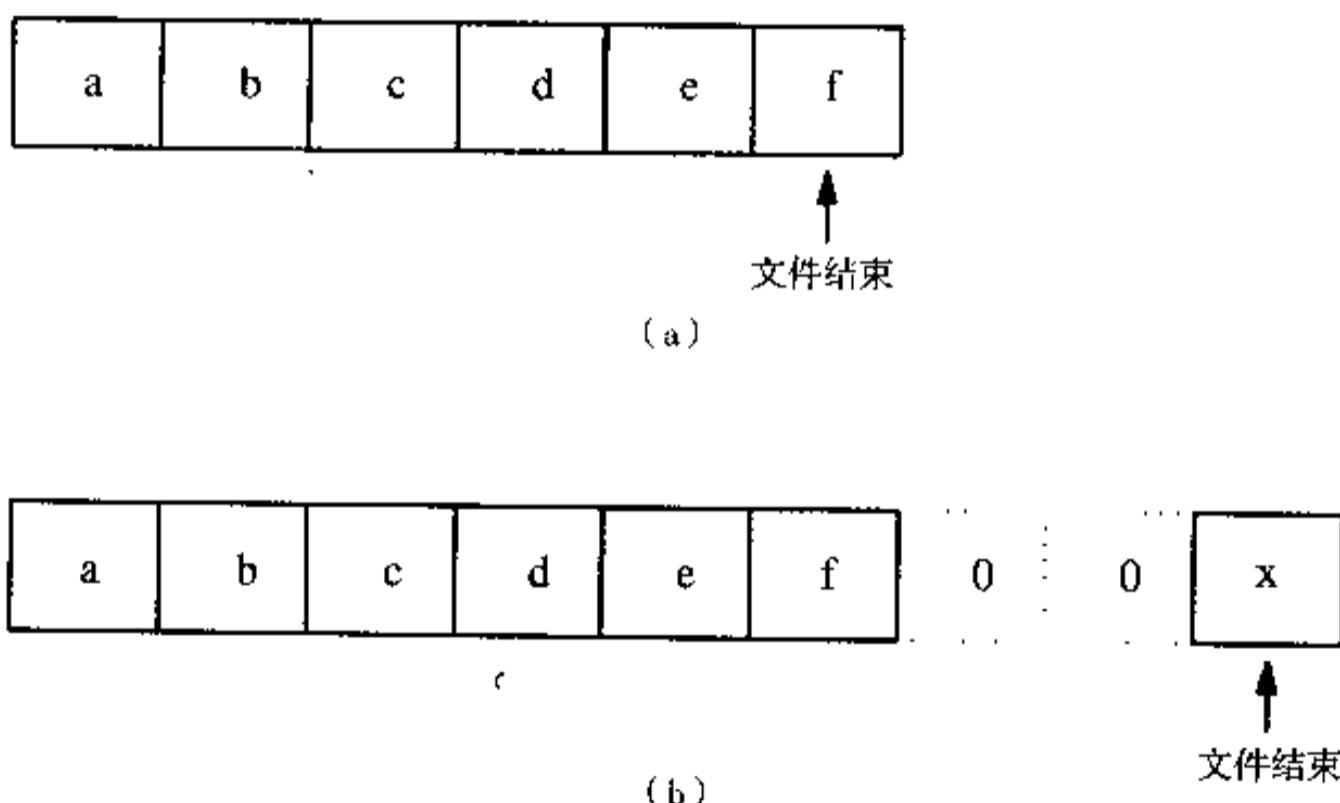


图 24.3 (a) 包含从 a 到 f 共 6 个字节的 UNIX 文件；(b) 程序定位到位置 8，并写入一个内容为字母 x 的字节。未写入的字节的值为零

尽管文件系统看起来用空字节填补了空隙，但是存储结构可以模拟空字节而不在物理媒体上表现出来。因此，文件大小记录了数据被写入的最高字节的位置，而不是已写入的字节的总数。

24.7.10 文件位置和并发存取

UNIX 文件系统允许多个应用程序并发地存取同一文件。每个打开的文件的描述符引用一个记录文件当前位置（current position）的数据结构。当一个进程调用 fork 创建一个新进程时，子进程继承了父进程在调用 fork 时已打开的所有文件的描述符的副本。这些描述符指向一个统一的用来存

取文件的数据结构。因此，如果这两个进程中的任何一个改变了文件的位置，则另一个进程中的文件位置也要发生改变。

每次调用 `open` 都生成一个新的描述符，它的文件位置独立于前面调用 `open` 时所生成的描述符。因此，如果两个应用程序对同一文件都调用了 `open`，那么它们都维护一个各自独立的文件位置。一个程序可以移到文件的末尾，而另一个程序则可以一直在文件的开始处。程序员在设计一个程序时，必须决定是否需要与其他进程共享同一文件位置或有一个独立的文件位置。

当我们研究对远程文件的操作时，理解文件位置与文件的分离性是非常重要的。这个概念可以概括如下：

每次调用 `open` 都产生一个新的文件存取点，它存储了文件偏移量。把当前文件位置从文件本身独立出来，可允许多个应用程序互不干扰地对同一文件进行并发存取。它还允许由 `lseek` 操作在不改变文件本身的情况下修改某个程序的文件位置。

24.7.11 在并发存取时的“写 (write)”语义

当两个程序对某个文件并发写入时，它们可能会产生冲突。例如，假定有两个并发程序，每个都读取某文件的前两个字节，把这两个字节交换一下，并把它们写回到该文件中。如果调度程序选择先运行一个程序然后再运行另一个程序，则前面的程序将交换两个字节并把它们写回到文件中。第二个程序以相反的顺序把这两个字节读出来，把它们交换成为最初的顺序，然后再写回该文件中。但是，如果调度程序同时开始运行两个程序，并允许每个程序可在任一程序写入之前从文件中读取数据，则它们都将以初次序读取这两个字节，并且都把它们以相反次序写入文件。其结果是，文件中字节的顺序取决于系统调度程序如何对这两个程序分配 CPU。

UNIX 没有互斥机制，也没有定义并发存取的语义，它只是指出一个文件总是含有一个最近写入的数据。这样，保证正确性的责任就落在了程序员的身上^①。程序员在设计并发程序时，必须小心设计使得这些程序产生同样的结果。

24.7.12 文件名和路径

UNIX 提供了一个分层的文件名空间。UNIX 文件系统中的每个文件和目录都有自己的用 ASCII 字符串表示的名字。除此之外，每个目录或文件都有一个全路径名 (full path name) 来表示文件在分层结构中的位置。图 24.4 表明了一个很小的 UNIX 分层结构例子中的文件及目录名。

如图所示，该文件系统的顶层目录叫做根 (root)，其全路径名为 `/`。一个文件的全路径名可看成是文件分层结构中从根到该文件的路径上的各标号的拼接，并且用 `/` 作为分隔符。例如，在根目录下名为 `a` 的文件的全路径名为 `/a`。在目录 `b` 下名为 `e` 的文件的全路径名为 `/b/e`，文件 `k` 的全路径名为 `/d/g/k`。

24.7.13 索引节点 (inode): 存储在文件中的信息

除了数据本身以外，UNIX 还在稳定存储设备上存放了有关每个文件的信息。这些信息保存在一个称为文件索引节点 (inode)^② 的结构中。索引节点包含了许多域，包括：文件拥有者和组标识

① 与其他现代 UNIX 版本一样，Linux 也提供了一种推荐的锁机制 (lock mechanism) `flock`。更老的系统使用一种称为 `lockf` 锁机制，或在 `open` 调用中使用参数 `O_EXCL` 提供加锁机制。

② Inode 是索引节点 (index node) 的缩写。

符、模式整数（见第 24.7.4 一节中所述）、最近一次修改的时间、文件长度、文件所驻留的磁盘设备和文件系统、文件的目录项数、文件当前所用的磁盘块数以及基本类型（例如，普通文件或目录）。

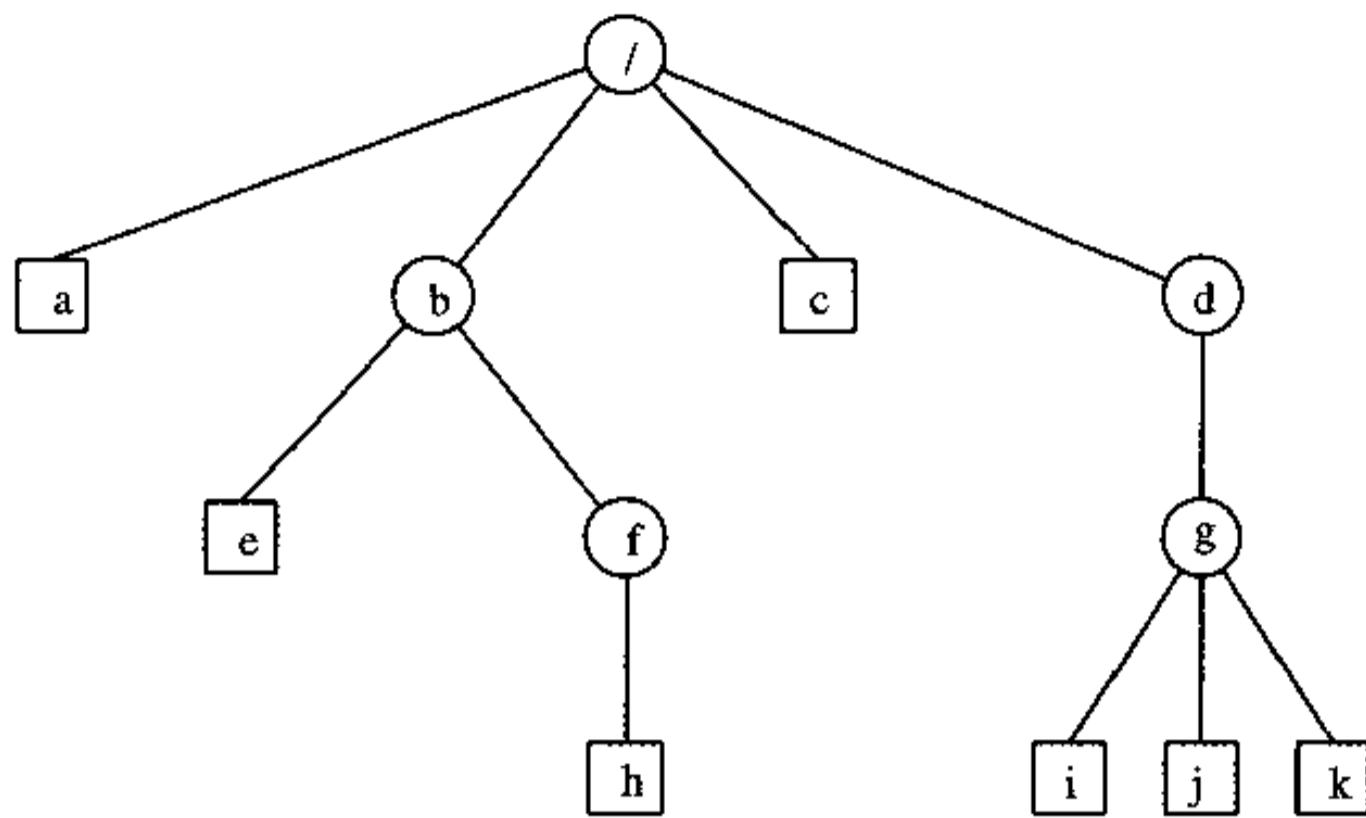


图 24.4 一个分层的文件系统。圆圈表示目录，方框表示文件。在这个例子中，顶层包含两个文件（a 和 c）和两个目录（b 和 d）。实际上，UNIX 文件很少用单字符的名字

索引节点的概念有助于解释几个 UNIX 文件系统的特色，这些特色在 NFS 也使用了。首先，UNIX 把文件拥有者关系和文件保护比特等信息从文件的目录项分离出来。这样就使两个目录项可以指向同一文件。UNIX 用名词链接（link）或硬链接（hard link）来指某个文件的一个目录项。如图 24.5 所示，当一个文件有多个硬连接时，它将出现在多个目录中。

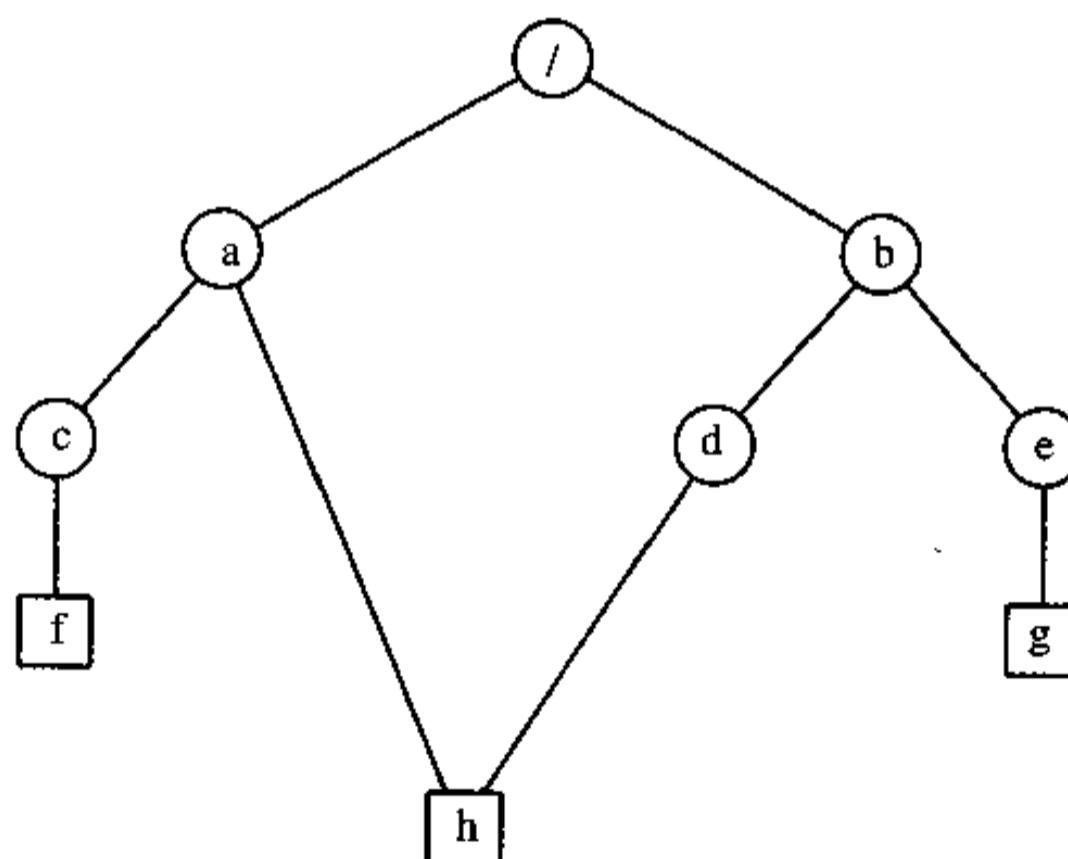


图 24.5 硬链接的示例。文件 h 有两个链接，一个来自目录 a，另一个来自目录 d。文件 h 可用路径名 /a/h 或 /b/d/h 进行存取

有多个硬链接的文件可以通过多个全路径名进行存取。例如，在图 24.5 中，文件 h 可由路径名 /a/h 进行存取，因为它在目录 a 中出现。它也可由全路径名 /b/d/h 进行存取，因为它也出现在目

录 d 中。如例子中所示，有多个链接的文件的路径名长度可以不同。

因为 UNIX 在其索引节点而不是在目录中存放文件的信息，所以不管程序用哪一个路径名来存取文件 h，它的文件拥有者关系及保护信息都能保持一致。如果文件拥有者改变了文件 /a/h 的保护模式，则文件 /b/d/b 的保护模式也将发生改变。

24.7.14 stat 操作

UNIX 系统函数 stat 从它的索引节点中提取文件信息并把它返回给调用者。该调用有两个参数：文件的路径名及存放结果的结构的地址：

```
stat(pathname, &result_struct);
```

第二个参数必须是内存中一块足以存放如下结构的空间的地址：

```
struct stat {                                /* structure returned by stat      */
    dev_t      st_dev;           /* device on which inode resides   */
    ino_t      st_ino;          /* file's inode number             */
    u_short    st_mode;          /* protection bits                 */
    short      st_nlink;         /* total hard links to the file   */
    short      st_uid;           /* user id of file's owner        */
    short      st_gid;           /* group id assigned to files     */
    dev_t      st_rdev;          /* used for devices, not files   */
    long       st_size;          /* total size of file in bytes   */
    time_t     st_atime;         /* time of last file access      */
    int        st_unused1;        /* not used                      */
    time_t     st_mtime;          /* time of last modification     */
    int        st_unused2;        /* not used                      */
    time_t     st_ctime;          /* time of last inode change    */
};
```

任何用户可调用 stat 来得到文件信息，即使该文件是不可读的。但是，调用者必须允许搜索在指定文件的路径上的所有目录，否则 stat 将返回一个错误。

24.7.15 文件命名机制

尽管用户想像所有文件和目录都是某个文件分层的一部分，但是分层是通过一种文件命名机制获得的。命名机制允许系统管理员把几个小的分层综合成一个概念性的层次结构。用户对下层的文件系统结构或多个组件如何形成 UNIX 的分层结构知之甚少，因为命名机制完全隐藏了这个结构。我们将看到，当 NFS 在某个 UNIX 系统，如 Linux 中运行时，它从这种命名机制中大为受益，它把远程文件与本地文件集成到一起。

UNIX 命名机制的最初动机产生于计算机系统有多个物理存储设备（即多个硬磁盘）。为了让用户免于既要识别文件又要识别磁盘，UNIX 设计者产生了一个想法：允许系统管理员把一个磁盘上的层次结构连到另一个磁盘的层次结构上。其结果是一个统一的文件名空间，它允许用户在不知道文件位置的情况下工作。命名机制是这样运行的：

- 由管理员指定某个磁盘上的分层结构为根。
- 管理员在根的分层上创建一个空目录（empty directory）。空目录的全路径名可由 /α 给出。
- 管理员指定命名机制在 /α 目录上覆盖一个新的分层结构（通常是从其他磁盘上）。

一旦管理员接连了新的分层结构，命名机制自动地把新接连的分层结构中的具有路径β的文件或子目录映射到形式为 /α/β 的名字上。这里的重要概念是：

UNIX 命名机制提供给用户和应用程序一个统一的文件分层结构，即使底层的文件跨越了多个物理磁盘。

以上说明了把整个磁盘作为分层结构一部分而接连到系统上，事实上，UNIX 系统比这更具有通用性。它允许系统管理员把一个物理磁盘划分成一个或多个文件系统。每个文件系统都是一个独立的分层，它包括了文件和目录。一个文件系统可以在任何一点上接连到统一的分层结构上去。下一部分的例子将阐明这种命名机制。

24.7.16 文件系统 mount

UNIX 命名机制依赖于 mount 系统调用来构造统一的分层结构。系统管理员用 mount 指明在一个磁盘上的文件系统如何接连到分层结构上。通常，由管理员来安排在系统启动时自动执行必要的安装。图 24.6 所示三个文件系统已经安装成为一个统一的分层结构。

在图中，硬盘 1 上的文件系统 0 已安装到分层结构的根上去了。在同一硬盘上的文件系统 1 已安装为目录 /a，在硬盘 2 上的系统 0 安装为目录 /c。mount 用一个新的文件系统完全地覆盖了最初的目录。通常，系统管理员创建一个用来进行安装的目录。但是，如果将被某个文件系统安装的目录在安装之前已经有文件了，则它们将被完全隐藏起来（也就是说，即使对系统管理员来说也是不可存取的），直到文件系统的安装被撤销为止。

除了很少的一些例外以外，对用户来说，安装完全不可见^①。如果用户列出目录 / 中的所有内容，则系统报告三个子目录：a、b 和 c。如果用户列出目录 /a 中的内容，则系统报告两个子目录：g 和 h。如果用户列出目录 /c/s 中的内容，则系统报告一个文件 u，一个目录 v。

路径名不表示文件系统之间的界限；用户并不知道文件驻留在什么地方。例如，名为 n 的文件（驻留在硬盘 1 上的文件系统 1 上）可用全路径名 /a/g/j/n 对其进行存取。

一旦用 mount 系统函数生成了一个 UNIX 文件系统分层结构，那么各个文件系统之间的接连就成为透明的了。有的文件和目录可以驻留在一个硬盘上，而另一些文件和目录可驻留在其他硬盘上。

用户无法区别它们，因为这种安装在一种统一的命名机制下隐藏了所有的界限。

实际上，如果用户感兴趣，他们会发现如何安装文件系统。系统包含一个可执行名为 mount 的应用程序，它查询系统并显示已安装的文件系统的清单。管理员也可以用 mount 命令生成新的 mount。例如，在一台计算机上运行 mount 命令产生下面的输出：

```
/dev/hda1 on /          type ext2 (rw, noquota)
/dev/hdb1 on /usr        type ext2 (rw, noquota)
/dev/hda2 on /usr/src    type ext2 (rw, noquota)
```

^① 用户不能跨越文件系统界限创建硬链接。

其中搜索子目录 a。一旦它发现 /a，它就打开该目录，并搜索子目录 b。同样，它在目录 b 中搜索目录 c，搜索 c 中是否有一个文件或子目录名为 d。名字解析软件可以每次从全路径名中提取出其中的一部分，因为斜杠把各个部分分隔开来了。尽管 UNIX 文件名解析的细节并不重要，但是它的概念却是很基本的：

UNIX 一次解析一个全路径名的一部分。它从分层结构的根及路径的开始出发，重复地从路径中提取出下一部分，并找出一个具有该名字的文件或子目录。我们将看到 NFS 在解析名字的时候采取了与 UNIX 相同的方法。

24.7.18 符号链接

大多数 UNIX 文件系统允许一种特殊的文件类型，叫做符号链接 (symbolic link)。符号链接是一个特殊的文本文件，它包含了另一个文件的名。例如，我们可以创建一个名为 /a/b/c 的文件，它包含了一个值为 /a/q 的符号链接。如果一个程序打开文件 /a/b/c，则系统发现它包含一个符号链接，并自动地切换到文件 /a/q。

符号链接的主要好处在于它们的通用性：因为一个符号链接可以包含任意一个字符串，它可以对任意文件或目录命名。例如，尽管文件系统禁止创建到某个目录上的硬链接，它却允许用户创建一个到该目录的符号链接。而且，因为符号链接可以指向任意路径，所以它可以用来把长路径名缩写或使分层结构上一个较远的目录显得距离更近一点。

符号链接的主要缺点在于它们缺乏一致性和可靠性。例如，我们可以创建一个到某个文件的符号链接，然后把该文件移走，使符号链接指向一个不存在的对象。事实上，我们也可以创建到一个不存在文件的链接，因为系统在创建符号链接时并不检查它的内容。我们还可以创建一组形成回路的符号链接，或者两个符号链接互相指向对方。在这种链接上调用 open 会产生一个运行时的错误。

24.8 NFS 下的文件

NFS 使用了许多 UNIX 文件系统的定义。它把文件看成一种字节序列，允许文件增长到任意长度，允许用文件中的字节位置作为基准对文件进行随机存取。它也遵循与 UNIX 相同的打开 - 读 - 写 - 关闭 (open-read-write-close) 规范，并提供大多数相同的服务。

正如 UNIX 一样，NFS 也假定了一个分层结构的命名系统。NFS 分层结构使用了 UNIX 的名词；它认为文件层次由目录和文件组成。一个目录可以包含文件或其他目录。

NFS 也采纳了许多 UNIX 文件系统的实现细节，有些没有改变或者只做了很小的修改。下一部分描述了 NFS 的几个特性，并显示了它们与前面所述的 UNIX 文件系统之间的关系。

24.9 NFS 的文件类型

NFS 使用与 UNIX 相同的基本文件类型。它定义了服务器在指定文件类型时可使用的枚举值：

```
enum ftype3 {
    NF3REG  = 1,          /* Regular data file      */
    NF3DIR  = 2,          /* Directory              */
    NF3BLK  = 3,          /* Block-oriented device */
}
```

```

NF3CHR    = 4,          /* Character-oriented device */
NF3LNK    = 5,          /* Symbolic link           */
NF3SOCK   = 6,          /* Socket                  */
NF3FIFO   = 7,          /* Named pipe              */
};


```

类型的集合，包括 NF3BLK 和 NF3CHR，直接来源于 UNIX。具体来说，UNIX 允许系统管理员在文件系统名字空间中对 I/O 设备进行配置，这使得应用程序可以使用传统的打开 - 读 - 写 - 关闭规范来打开一个 I/O 设备，并向它传输或从中读取数据。NFS 还采纳了 UNIX 的方法，把 I/O 设备分成面向块的（例如，硬盘通常每次传输 512 字节的数据）和面向字符的（例如，一个 ASCII 终端设备一次传输一个字符）设备。NFS 有时使用 UNIX 的名词——专有文件 (special file) 来表示设备名。对应于面向块设备的文件名具有块专有 (block-special) 类型，而对应于面向字符设备的文件名具有字符专有 (character-special) 类型。

24.10 NFS 文件模式

像 UNIX 一样，NFS 假定每个文件或目录有一个指明其类型和存取保护的模式 (mode)。图 24.7 列出了 NFS 模式整数的单个比特及其含义。表中使用了八进制数来表示这些比特；定义直接对应于 UNIX 的 stat 函数返回值。

模式比特	意义
0x0800	在执行时设置用户 id
0x0400	在执行时设置组 id
0x0200	保存交换的文本 (POSIX 没有定义)
0x0100	文件拥有者读文件权限
0x0080	文件拥有者写文件权限
0x0040	文件拥有者执行文件的权限或目录拥有者目录查找 (搜索) 的权限
0x0020	组的读权限
0x0010	组的写权限
0x0008	文件所在组执行文件的权限或目录所在组进行目录查找 (搜索) 的权限
0x0004	其他用户的读权限
0x0002	其他用户的写权限
0x0001	其他用户对文件的执行权限或目录中其他用户对文件的查找 (搜索) 权限

图 24.7 NFS 模式整数中各比特的意义。定义直接取自 UNIX

尽管 NFS 为设备定义了文件类型，但它不允许远程设备进行存取（例如，一个客户不能对一个远程设备进行读写）。因此，尽管用户可以得到有关文件名的信息，但不能对设备进行操作，即使保护模式允许。

尽管 NFS 定义了决定一个客户是否可以读写某特定文件的保护比特，但 NFS 拒绝远程机器对所有设备进行存取，即使保护比特指定为允许存取。

24.11 NFS 文件属性

与 UNIX 类似，NFS 有一个获得关于某文件的信息的机制。在谈到文件信息时，NFS 使用了名

词文件属性 (file attribute)。结构 fattr3 描述了 NFS 所提供的文件属性：

```
struct fattr3 {                                /* NFS file attributes      */
    ftype3      type;        /* Type:file, directory, ect   */
    mode3       mode;        /* File's protection bits     */
    uint32      nlink;      /* Total hard links to the file */
    uid3        uid;        /* User id of file's owner    */
    gid3        gid;        /* Group id assigned to file  */
    size3       size;       /* Total size of file in bytes */
    size3       used;       /* Disk space actually used   */
    specdata3   rdev;       /* Description if file is device */
    uint64      fsid;       /* File system ID for file    */
    fileid3     fileid;     /* Unique id for file (inode) */
    nfstime3   atime;      /* Time of last file access   */
    nfstime3   mtime;      /* Time of last modification  */
    nfstime3   ctime;      /* Time of last inode change  */
}
```

正如该结构所示，它的概念及大多数细节都来源于 UNIX 的 stat 函数所返回的信息。

24.12 NFS 客户和服务器

NFS 服务器运行在一台具有本地文件系统的机器上。服务器使远程机器也可以对本地某些文件进行存取。NFS 客户运行在任意机器上，它可以对运行 NFS 服务器的机器上的文件进行存取。一个机构往往指定一台具有很大磁盘空间的计算机来完成服务器的功能。这样的机器常常叫做文件服务器。禁止用户在 NFS 服务器上运行应用程序有利于保持较低的负载，还有利于保证对存取请求的快速响应。指定一台计算机完成文件服务器功能也保证了远程文件存取的通信量不至于减少应用程序所得的 CPU 时间。

大多数 NFS 客户实现把 NFS 文件与计算机的本地文件系统结合起来，对应用程序及用户隐藏文件的位置。例如，考虑 Windows 环境，文件名具有形式 X: α ，其中 X 是一个单字符磁盘标识符， α 表示该磁盘上的一个路径名。Windows 用反斜杠 (\) 分隔路径中的各个部分。因此，文件名 C:\D\E\F 表示在系统的硬盘上的目录 D 中的子目录 E 中的文件 F。如果一个 NFS 客户被加到该系统中，并且被配置成可以对一个远程服务器上的文件进行存取，那么它可以使用 Windows 的命名方案。例如，管理员可能为所有远程文件选用形式为 R: β 的名字，其中 β 表示在远程文件系统中的一条路径。

当程序调用 open 对某个文件进行存取时，操作系统用路径名的语法来选择使用本地的还是远程的文件存取过程。如果路径指向一个远程文件，系统就使用 NFS 客户软件来存取远程文件；如果路径指向一个本地文件，系统就使用计算机的标准文件系统来存取该文件。图 24.8 说明了在作出选择时，操作系统中的各个模块是如何交互的。

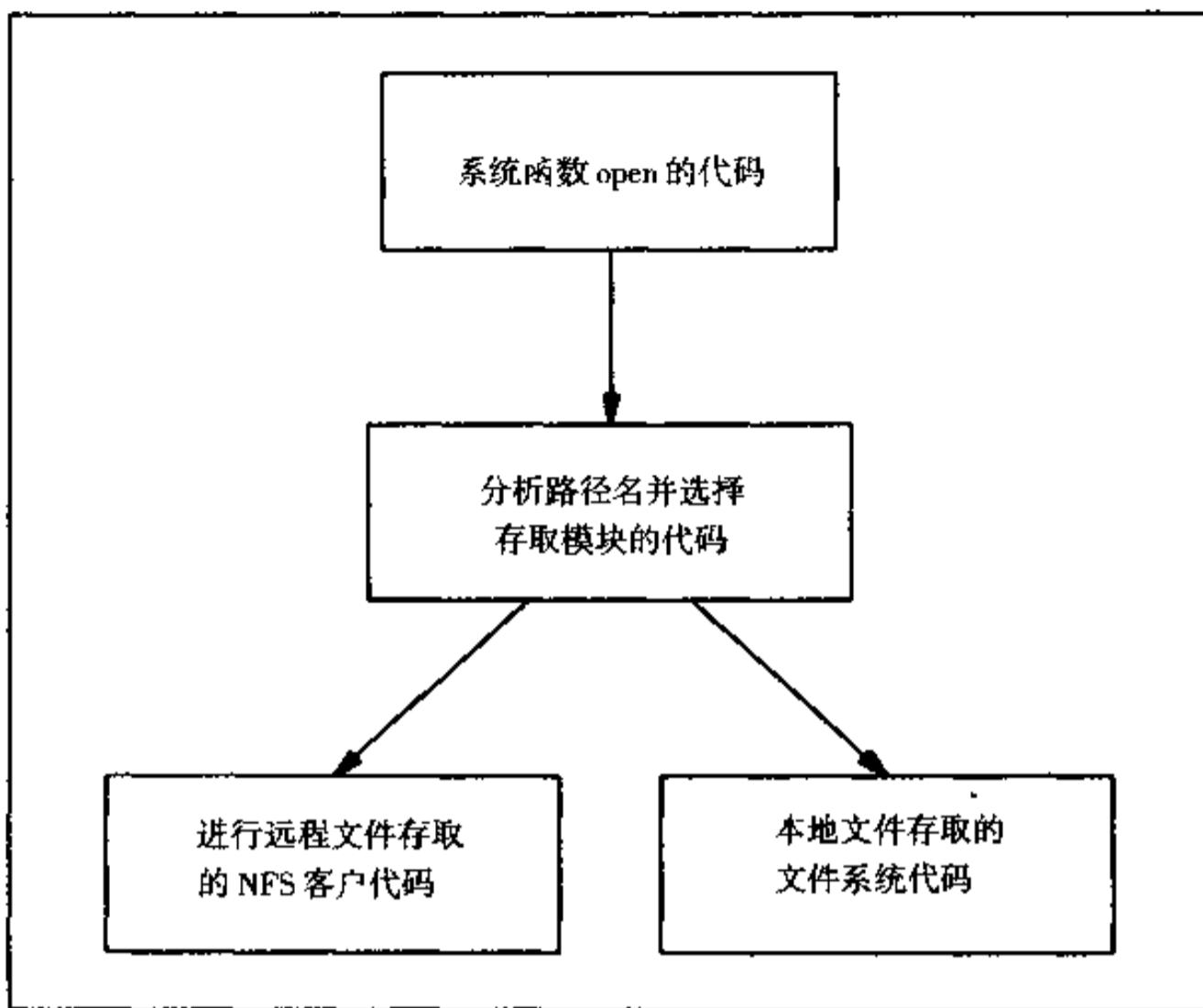


图 24.8 当一个应用程序打开某个文件时，操作系统中被调用的过程。系统使用路径名语法在 NFS 和标准文件系统间做出选择，前者将打开一个远程文件，后者将打开一个本地文件

24.13 NFS 客户操作

NFS是设计用来适应异构计算机系统的。当系统管理员们在操作系统中安装NFS客户代码时，他们想把它集成到系统的文件命名方案中。但是，远程文件系统所使用的路径名语法可能与客户所使用的不同。例如，当NFS客户代码运行在Windows上，这台机器连接一台运行NFS服务器的机器，而这台服务器使用UNIX。客户的系统用反斜杠(\)作为分隔符，而服务器文件系统用斜杠(/)。

为了适应客户与服务器的路径名语法之间可能存在的差异，NFS遵循一个简单的原则：只在客户端解释全路径名。因为在服务器的文件分层目录系统中跟踪一条全路径名，客户每次传送路径名的一个单元。例如，如果某客户用斜杠作为分隔符，需要查找在某台服务器上的路径 /a/b/c，它先从获得服务器的根目录信息开始。接着，它会要求服务器在该目录中查找名 a。服务器发回有关 a 的信息。假定该信息将显示 a 是一个目录。然后客户要求服务器在该目录中查找名 b。当服务器应答时，客户证实 b 是否是一个目录，如果是一个目录，客户会继续要求服务器在其中查找名 c。最后，服务器将发回有关 c 的信息作为响应。

要求客户分析路径名的主要缺点是非常明显的：对路径上的每一个分量，都要在网络上有一次信息交换。而它主要的优点是：在一给定计算机上，应用程序可用与本地文件相同的路径名语法来存取远程文件。最重要的是，应用程序和客户都可以在不知道文件位置或服务器上的命名约定的情况下，编写对远程文件进行存取的程序。因此，当系统管理员对某台服务器进行升级时，任何客户程序都不需要改变，即使新服务器使用了一种不同的操作系统或不同的文件命名方案也是如此。总而言之：

为了保证客户上的程序独立于文件位置和服务器计算机系统，NFS 要求仅由客户来解释全路径名。客户每次向服务器发送路径的一个分量，并接收关于它所命名文件或目录的信息。客户通过这种方法在服务器的文件分层结构中跟踪路径。

24.14 NFS 客户与 UNIX 系统

UNIX 系统，如 Linux，使用安装机制为多个磁盘上的每个文件系统创建一个统一的命名分层结构。NFS 客户代码的 UNIX 实现使用安装机制的扩展版，把远程文件系统集成到了本地文件系统中。从应用程序的观点看，用安装机制的主要优点是一致性：所有的文件名都有相同的形式。一个应用程序从名字语法上无法判断一个文件是本地的还是远程的。如果在调用 open 时指明的路径包括 NFS 安装目录，操作系统最终将遇到远程安装点并把控制传给 NFS 客户端代码。NFS 客户端继续分析路径名并查找路径分量，最终完成打开文件的操作。最后，NFS 客户端到达路径的末尾，并返回远程文件的信息。系统为该远程文件创建描述符，应用程序在后面的 read 和 write 操作中使用该描述符。因此，当应用程序打开一个远程文件时，它得到一个针对该文件的整数描述符，这点正如对本地文件一样。只有操作系统知道与描述符相关的内部信息，该信息说明该文件是通过 NFS 存取的远程文件。

只要应用程序对文件描述符进行操作（比如读），系统就将检查描述符指向的是本地文件还是远程文件。如果文件是本地的，操作系统按照通常的办法处理该操作。如果文件是远程的，操作系统就调用 NFS 客户代码，把该操作转换成一个等价的 NFS 操作，并向服务器发起一个远程过程调用。

24.15 NFS 安装

当管理员把 NFS mount 加到一个 UNIX mount 表上时，他们必须指明一台运行 NFS 服务器的远程机器、在该服务器上的分层结构、一个本地目录（NFS mount 将加到它上面）以及有关该 mount 的细节信息。例如，下面是个由 UNIX 的 mount 命令得到的输出，它显示了珀杜大学的一个 UNIX 系统所使用的某些 NFS mount（非 NFS mount 已被删除）：

```
arthur:/p1 on /p1      type nfs (rw, grpid, intr, bg, noquota)
arthur:/p4 on /p4      type nfs (rw, grpid, intr, bg, noquota)
ector:/u4 on /u4       type nfs (rw, grpid, soft, bg, noquota)
gwen:/    on /gwen     type nfs (rw, grpid, soft, bg, noquota)
gwen:/u5 on /u5       type nfs (rw, grpid, soft, bg, noquota)
```

在这个输出中，每一行都对应着一个 NFS 文件 mount。每一行的第一个字段指明一台运行 NFS 的服务器及该服务器上的分层结构，而第三个字段指明了远程文件系统所安装的本地目录。例如，arthur:/p1 指明 /p1 层在 arthur 机上。它已安装到本地目录 /p1。系统管理员把 arthur 的 /p1 文件系统安装在具有相同名字的本地目录中，这样便于两台机器上的用户用相同的名字对文件进行存取。

例子中所示的所有 mount 都具有类型 nfs，意思是它们指向的是可以通过 NFS 存取到的远程文件系统。除此之外，每行括号中的参数指明了有关 mount 的更进一步的细节。像本地文件系统的 mount 一样，远程文件系统 mount 也能指明文件是否可读写 (rw) 或者是只读 (r)。

NFS 为 UNIX 中的远程安装定义了两个基本的范例。软安装 (soft mount) 指明 NFS 客户应实现超时机制，并且，如果超时则认为服务器脱机。硬安装 (hard mount) 指明 NFS 不应使用超时机制。

系统管理员通常安排在系统启动时自动地创建所有 mount。一旦创建了一个 NFS mount 后，应用程序和用户就无法区别本地和远程文件了。用户可以用常规的程序对远程文件进行操作，这与操作本地文件一样容易。例如，用户可以运行一个标准的文本编辑器来编辑远程文件；编辑器对远程文件的操作方式与对本地文件是一样的。而且，用户可以很容易地进入远程目录或退回到本地目录，为此，只要给出跨越安装点的路径名就可以了。

24.16 文件句柄

一旦某个客户识别并打开了一个文件后，它需要有一种识别该文件的途径以便进行后续的操作（比如读或写）。而且，当客户通过服务器的分层结构来搜索某个目录或文件时，它也需要一种识别它们的方法。为了解决这些问题，NFS 让服务器给每个文件分配一个惟一的文件句柄 (file handle)，以此作为标识符。当客户第一次打开某个文件时，服务器制造一个句柄，并把该句柄发回给客户。当客户再对该文件有操作请求时，就把该句柄发回给服务器。

从客户的观点看，文件句柄是一个用来标识文件的 32 字节的字符串。从服务器的观点看，文件句柄可以是一个惟一地标识某个文件的任意的字节集合。例如，某些信息可以使服务器迅速对句柄解码或能迅速定位文件，一个文件句柄可以对这些信息进行编码。

在 NFS 名词中，文件句柄对客户是不透明的，也就是说，客户不能对句柄进行解码或自行创建一个句柄。只有服务器才可以创建文件句柄，服务器只能识别它们自己创建的文件句柄。而且，NFS 服务器的安全实现使用了一种复杂的编码方式，以防客户猜测某个文件的句柄。具体来说，服务器随机地选择句柄的某些比特以保证用户自己不能创建合法的句柄。

为了提高安全性，服务器也可以限制句柄使用的时间。这样，一个客户就不能永久地拥有一个句柄。为了限制一个句柄的生命期，服务器在句柄中编码引入了时间戳 (timestamp)。如果句柄超时了，服务器就拒绝使用该句柄的对某文件所进行的进一步操作。客户要继续对文件进行存取，必须得到一个新的句柄。在实际当中，NFS 时间戳通常要保持足够长的时间，以便任何合理的存取能够进行。有权使用某个文件的程序通常可以得到一个新的句柄（对为此所必须进行的交易请求，应用程序全然不知）。

24.17 句柄取代路径名

为了理解为什么需要句柄，让我们考虑在一个远程目录分层结构中对文件的命名。为了使客户与服务器的路径名语法隔离开来，并允许异构机存取具有分层结构的文件，NFS 要求客户完成所有对路径名的解释。其结果是，当客户要求对某个文件进行操作时，不能用全路径名指明该文件，而必须要得到一个用于引用该文件的句柄。

让服务器为目录和文件提供句柄可使得客户能够通过服务器的分层结构来跟踪某个路径。为了说明这是如何做到的，让我们考虑图 24.9，图中展示了当客户要在服务器的分层结构中查找一个路径为 /a/b/c 的文件时，客户和服务器之间所进行的信息交换。

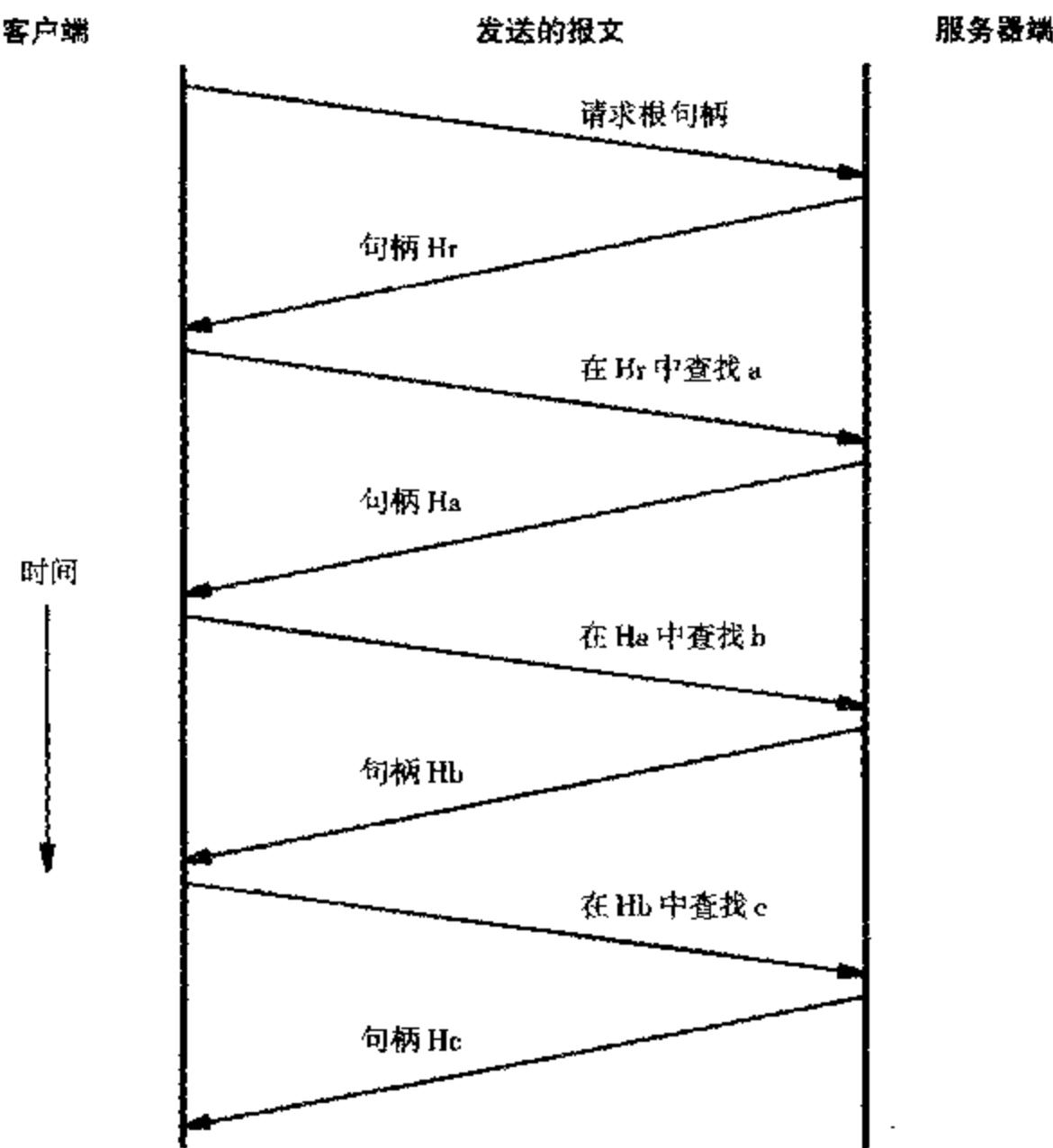


图 24.9 当客户查找全路径名为 /a/b/c 的文件时，客户与服务器之间所进行的报文交换。“/”表示客户的路径名各分量之间的分隔符。名为 x 的文件或目录的句柄用 H_x 表示。实际上，有一个单独的协议提供根句柄

图中显示了服务器为路径上的每个目录返回了一个句柄。客户把各个句柄用在下一个远程过程调用中。例如，当客户得到了目录 a 的句柄后，它把这个句柄发回给服务器并要求服务器在目录 a 中搜索 b。客户不能用 /a/b 来引用该文件，它也不能用 /a 引用该目录，客户不知道服务器的路径名语法。

24.18 无状态服务器的文件定位

因为 NFS 采用了无状态的服务器设计，所以服务器不能为每个使用文件的程序保存文件位置 (file position)，而是由客户来存储所有文件的位置信息，每个发到服务器的请求也必须给出所要使用的文件位置。在客户端存储位置信息还有助于优化那些要改变文件位置的操作。例如，在 Linux 实现中，NFS 用本地文件表存储远程文件的位置，就像在本地文件中存储该位置一样。如果客户调用 lseek，则系统不需要向服务器发信息就把新的文件位置记录在表中。任何后继的存取从该表中提取出文件位置，并把它和存取请求一起发给服务器。因为 lseek 并不在网络上发布任何信息，所以在远程文件中进行定位与在本地文件中进行定位一样简单。

24.19 对目录的操作

从概念上说，NFS 定义目录是由一个对 (pair) 的集合组成的，每个对包含一个文件名和一个指向该命名文件的指针。目录可以为任意长度，没有限制。

NFS 提供的操作包括允许客户：向目录中插入 (insert) 文件；从目录中删除 (delete) 文件；在目录中搜索 (search) 一个名字；读 (read) 目录中的内容。

24.20 无状态地读目录

因为目录的大小可以是任意的，而通信网络对网络上报文的长度有一定限制，所以对目录内容的读取可能需要多次请求。NFS 服务器是无状态的，因而服务器无法记录每个客户在目录中的位置。

NFS 设计者为了克服无状态服务器的这种限制，选用了这样一个办法：当服务器在应答进入某个目录的请求时，返回一个位置标识符 (position identifier)。客户在下一次请求中用该位置标识符来说明它已经收到并还需要哪些目录项。因此，当客户想从一个远程目录中读取某些目录项时，它不断重复请求，每次请求均用上一次请求所返回的位置标识符来指定本次请求的位置，以此逐步获得该目录。

用系统程序员们流行的非正式名词来说，NFS 把它的目录标识符叫做魔块 (magic cookie)。这个词的意思是客户不需要对标识符进行解释，也不能创建标识符。只有服务器才可以生成魔块；客户只能使用服务器提供的魔块。

魔块不保证原子性，也不对目录加锁。因此，两个对同一目录进行操作的程序可能相互干扰。例如，如果一个服务器并发地处理两个以上客户的请求。假设在从某个目录 D 中读出三个目录项后，第一个客户接收到一个魔块，指向第四个目录项之前的位置。接着，假设另一个客户在目录 D 中完成了一系列的插入和删除操作。如果第一个客户试图用旧魔块从 D 中读出其他的目录项，它将无法收到所有修改后的结果。

在实际当中，并发存取目录的潜在问题很少对用户产生影响，因为他们不依赖于文件瞬时的插入或删除。事实上，许多常规的操作系统也有与 NFS 类似的行为；用户很少知道系统如何解释并发目录操作，因为他们几乎不需要知道这些细节。

24.21 NFS 服务器中的多个分层结构

为了使 NFS 在异构机型上能进行互操作，设计者让客户来分析所有的路径名。这样做就使得客户和服务器能使用本地操作系统所具有的文件命名策略，而不需要理解对方的环境。

限制全路径名的使用对大多数的文件操作几乎没有影响。但是，它却引入了一个严重的问题，因为它意味着客户不能用全路径名来标识远程文件系统或目录。

NFS 协议的早期版本假定每个服务器只提供对一个远程目录分层结构的存取。最初的协议包括一个叫做 NFSPROC_ROOT 的函数，客户可以调用它来得到服务器分层结构中根目录的句柄。一旦客户有了根的句柄，它就可以读取其中的目录项并搜寻该分层结构上的任意路径。

后来的 NFS 版本允许单个服务器提供这样的能力，即对位于多个文件分层结构中的文件进行远程存取。在这种协议中，仅仅返回一个根目录句柄是不够的。为了让服务器处理多个分层结构，

NFS 要求额外的机制。这个机制允许客户指明某个分层结构并得到它根目录的句柄。

24.22 安装 (mount) 协议

当前的 NFS 协议用一个单独的协议来处理查找根目录的问题，该协议叫做安装协议 (mount protocol)，它是用 RPC 来定义的。但是，安装协议不是 NFS 远程程序的一部分。尽管它是 NFS 服务器所必需的，但是安装协议是作为一个单独的远程程序来运行的。

安装协议提供了客户在使用 NFS 之前所需要的四种基本服务。首先，它允许客户得到一个可以通过 NFS 访问到的目录分层结构（也就是文件系统）的清单；第二，它接受全路径名，这种全路径名允许客户标识特定的目录分层结构；第三，它鉴别客户的每个请求，从而使客户有权对所请求的分层结构进行存取；第四，它返回客户所指明的分层结构的根目录的文件句柄。客户在发起 NFS 调用时将使用从安装协议中得到的根句柄。

安装协议的名字及其思想来自于 UNIX：当 UNIX 系统在它的名字空间中创建远程文件系统 mount 时，它就使用安装协议。客户系统用安装协议与服务器相联系，并且在把远程 mount 加到它本地分层结构的名字空间以前，验证对远地文件系统的存取。如果安装协议拒绝存取，则客户代码向系统管理员报告一条出错信息；如果安装协议同意存取，则客户代码保存该远程文件系统的根句柄，这样，如果此后某个应用程序要打开该系统中的某文件时，它便可以使用该句柄。

24.23 NFS 的传输协议

原来的 NFS 版本设计是用在局域网环境，因而，设计人员选择使用 UDP 作为运输层协议。UDP 的主要优点是服务器端较低的耗费以及较少地占用操作系统资源；它的主要缺点是可靠性差。为了弥补这一缺点，协议提供了一些基本的超时和请求重发机制。当然在实际中，在局域网上通信时包丢失的情况很少发生。

遗憾的是，NFS 的早期版本中的重传机制在广域网上工作得并不好。重传不是自适应的，没有提供拥塞控制或端到端流控制。虽然该协议有一些缺点，但在 20 世纪 90 年代中期，厂家们开始提供一些修正的 NFS 版本，以便于在因特网上进行应用。大多数修改的版本都依靠 TCP 进行传输。结果，版本 3 的设计就允许使用一种面向连接的传输服务。与版本 2 中定义一个一次操作所能传送的最大数据长度不同，版本 3 放松了在一个 TCP 连接上传送整个文件的限制。结果是，现在可用的 NFS 实现使用 TCP 来达到在连接上的因特网上进行可靠的文件接入。

24.24 小结

Sun Microsystems 公司定义了一个叫做 NFS 的远程文件存取机制，它现在已成为一个工业标准。为了允许多个客户访问同一服务器，并使服务器免受客户崩溃的影响，NFS 采用无状态服务器。

因为 NFS 的设计是用作异构环境的，所以客户无法知道所有服务器上的路径名语法。为了满足异构性，NFS 要求客户对路径名进行分解，并且查找路径上的每一个分量。当客户查找某个特定路径名时，服务器返回一个 32 字节的文件句柄，客户在后继的操作中可将它用作对文件或目录的引用。

大多数的 NFS 定义和文件系统语法都来源于 UNIX。NFS 支持分层的目录系统，把文件看作一个字节的序列，允许文件动态增长，提供对文件的顺序或随机存取，提供的有关文件的信息与 UNIX 的 stat 函数所给出的几乎完全等价。

NFS 所使用的许多概念性文件操作都来源于 UNIX 文件系统所提供的类似操作。NFS 采纳了 UNIX 使用的打开 - 读 - 写 - 关闭的规范，还采纳了 UNIX 的基本文件类型和文件保护模式。

伴随 NFS 的还有另一个协议，安装协议使得单个 NFS 服务器可以提供对多个目录分层结构的存取。安装协议实现了存取鉴别，允许客户得到某个特定目录分层的根目录的句柄。一旦客户得到了根的句柄，它就可以用 NFS 协议对该分层结构中的目录和文件进行存取。当系统管理员在 UNIX 的分层名字空间中安装远程 mount 时，系统将使用安装协议。

深入研究

Ritchie 和 Thompson [1974] 解释了最初的 UNIX 文件系统，还讨论了用索引节点和描述符实现文件的问题。Bach [1986] 讨论了 UNIX 文件系统语义。McKusick 等人的 [August 1984] 描述了后来成为 Linux 基础的快速文件系统。Callaghan 等人的 [RFC 1813] 描述了 NFS 协议以及安装协议的第 3 版，其中提到了本章出现的大多数概念。Shepler [RFC 2624] 对 NFS 协议的第 4 版本进行了初步讨论，[RFC 2339] 给出了相关合法约定。Callaghan 等 [RFC 2054, 2055 和 2224] 讨论了 NFS 和广域网的集成问题。

习题

- 24.1 编写一个从 NFS 服务器得到魔块的程序。用这个程序获得多个服务器上的多个目录的魔块，把它们中的内容用十六进制打印出来。猜一下魔块包含了什么信息以及服务器是如何对它们进行编码的。
- 24.2 用网络分析仪观察 NFS 服务器和客户之间所交换的报文。在下面的操作中，共交换了多少个分组：打开一个不在顶层目录中的文件，从该文件中读 10 个字节，然后关闭该文件？
- 24.3 假定某个连接有一个 NFS 客户和一个 NFS 服务器的网络，在它上面交付的信息分组失序了。这些重排序的 NFS 操作会产生什么错误结果？
- 24.4 假定连接有一个 NFS 服务器和 NFS 客户的网络会产生重复的分组并会发生交付失序。重复和重新排列的 NFS 操作将会产生什么错误结果？把你的答案与你的上一个练习的结果进行比较，分组重复是否引入了其他的错误情况？为什么？
- 24.5 检查 NFS 的第 3 版和第 4 版的规约。它们的主要差异是什么？版本 4 是否做了一些对程序员来说明显或重要的修改？
- 24.6 尽管 NFS 的设计是无状态的，但是文件句柄机制却为该协议增加了状态。文件句柄引入了什么问题（如果真的引入了问题）？怎样克服这样的问题？（提示：许多 NFS 的实现要求所有活跃的客户在服务器崩溃或重新启动后也重新启动。）

第 25 章 网络文件系统协议 (NFS, Mount)

25.1 引言

前一章描述了 Sun 公司的网络文件系统 (NFS) 中所蕴涵的概念，还展示了来自 UNIX 文件系统的许多名词和细节。本章将通过描述 NFS 协议继续对它进行讨论^①。本章将说明如何用 ONC RPC 把 NFS 定义成远程程序，还将说明对文件的每个操作如何与远程过程调用相对应。

25.2 用 RPC 定义协议

第 21 章到第 23 章介绍了如何用 RPC 把一个程序划分成运行在不同机器上的一些构件。大多数程序员都用那些章节中所描述的方法来使用 RPC：他们首先写一个常规的程序，然后用 RPC 来形成一个分布式的版本。本章将采取另一种办法。这种办法说明如何用 RPC 定义一个协议，而同时不把它与任何特定程序相联系。

可以用 RPC 构造一个程序的分布式版本，也可用它来构造一个一般目的的协议，它们的主要差异在于设计者考虑问题的方式。当构建一个程序的分布式版本时，程序员从一个现有的程序开始，该程序包括了过程及数据结构。而在设计一个协议时，程序员从一系列所需的服务开始，还要设计支持它们的抽象过程。

在设计协议时，要考虑如何使用服务，还要考虑程序员如何实现提供这些服务的程序。设计者必须选用一种协议的定义，它既要有精确性又要有灵活性。协议必须足够精确，以保证在其上的程序之间的互操作性，但是它又要具有一般性以允许多种实现。

靠凭空想像设计协议是行不通的。成功的协议规约要求设计者既要精通通信技术的细节又要有关设计的直觉。这个直觉通常产生于对计算机系统、应用程序以及其他通信协议的广泛经验。因此，协议的设计要比看起来复杂得多。具体来说，像 NFS 这样的协议，人们不能被其表面的简单所迷惑，误以为在设计时可以不必考虑很多。这里的关键是：

尽管 RPC 提供了一个很方便的方法来说明协议，但是它并没有使协议设计变得简单，也不能保证效率。

25.3 用数据结构和过程定义协议

为了指明使用 RPC 的协议，我们必须：

- 给出作为过程参数或函数返回值的常数、类型和数据结构

^① 本章的细节指的是协议的第 3 个版本。

- 给出每个远程过程的声明，它指明过程的参数、结果以及它所完成的动作的语义
- 最后通过说明过程如何处理其参数，如何计算出它的返回值来定义每个过程的语义

从概念上讲，使用 RPC 的协议把服务器定义成一个远程程序。从客户传送给服务器的一个操作对应着一个远程过程调用，而从服务器返回给客户的报文则对应着一个过程返回。因此，在任何由 RPC 定义的协议中，必须由客户发起所有的操作；服务器只能对每个客户的请求作出响应。

对 NFS 来说，因为文件存取协议可以设计成由客户来驱动，所以，要求由客户发起每次操作。服务器提供一些过程，允许客户做如下操作：创建 (create) 或删除 (delete) 文件、目录及符号链接；读 (read) 写 (write) 数据；在目录中搜索 (search) 某个命名文件；得到关于整个远程文件系统状态信息 (status)，或者获得某个文件或目录的信息。尽管其他应用协议可能不像 NFS 那样直接把自己引入 RPC 规范，但经验表明，大多数客户-服务器的交互都可以通过一定努力纳入到远程过程调用的范例中。

NFS 提供了一个 RPC 规约的有趣示例，因为这个协议相当复杂，它要求具有几个远程过程和数据类型。但在直觉和概念上，它又非常简单和便于理解。下一节描述了在 NFS 协议中所用的基本常数和类型声明的例子。后面的章节将展示那些过程声明如何用这些常数和类型来说明参数和结果。

25.4 NFS 常数、类型和数据声明

NFS 协议标准定义了常数、类型名和数据结构，在整个过程声明中都要使用它们。所有的声明都是用 RPC 声明语法给出的。

25.4.1 NFS 常数

NFS 定义了六种基本常数来指明协议使用的值的大小，这些声明是用 RPC 语言给出的。例如，下面的声明指明符号常数 FSF3_SYMLINK 的值为 2。

```
const FSF3_SYMLINK = 0x0002;
```

除了基本常数外，协议还定义了一个常量枚举集合，它用于报告差错状态。每个远程过程调用都返回其中的一个值。该信息集合的命名为 nfsstat3，定义为：

```
enum nfsstat3 {
    NFS3_OK                = 0,          /* Successful call           */
    NFS3ERR_PERM            = 1,          /* Ownership mismatch or error */
    NFS3ERR_NOENT            = 2,          /* File does not exist      */
    NFS3ERR_IO                = 5,          /* I/O device error occurred */
    NFS3ERR_NXIO              = 6,          /* Device or address does not exist */
    NFS3ERR_ACSES              = 13,         /* Permission to access was denied */
    NFS3ERR_EXIST              = 17,         /* Specified file already exists */
    NFS3ERR_XDEV                = 18,         /* Attempt to link across devices */
    NFS3ERR_NODEV              = 19,         /* Specified device does not exist */
    NFS3ERR_NOTDIR              = 20,         /* Specified item not a directory */
    NFS3ERR_ISDIR                = 21,         /* Specified item is a directory */
    NFS3ERR_INVAL              = 22,         /* Invalid argument           */
};
```

```

NFS3ERR_FBIG          = 27,           /* File is too large for server      */
NFS3ERR_NOSPC         = 28,           /* No space left on device (disk)    */
NFS3ERR_ROFS          = 30,           /* Write to read-only file system    */
NFS3ERR_MLINK         = 31,           /* Too many hard links               */
NFS3ERR_NAMETOOLONG   = 63,           /* File name was too long            */
NFS3ERR_NOTEMPTY       = 66,           /* Directory not empty              */
NFS3ERR_DQUOT          = 69,           /* Disk quota exceeded              */
NFS3ERR_STALE          = 70,           /* File handle is stale             */
NFS3ERR_REMOTE         = 71,           /* Too many levels in remote path   */
NFS3ERR_BADHANDLE      = 10001,        /* File handle is corrupted/illegal */
NFS3ERR_NOT_SYNC        = 10002,        /* Setattr synchronization mismatch */
NFS3ERR_BAD_COOKIE      = 10003,        /* Cookie is stale                 */
NFS3ERR_NOTSUPP         = 10004,        /* Operation not supported         */
NFS3ERR_TOOSMALL        = 10005,        /* Buffer or request is too small  */
NFS3ERR_SERVERFAULT     = 10006,        /* Server error other than above   */
NFS3ERR_BADTYPE         = 10007,        /* Requested type not supported   */
NFS3ERR_JUKEBOX         = 10008,        /* Client should reissue request  */
};


```

这些差错值只有在调用的具体环境下才有意义。例如，如果某客户试图对一个普通文件完成目录操作，服务器返回差错代码 NFS3ERR_NOTDIR，如果客户试图对一个目录完成普通文件的操作，服务器则返回差错代码 NFS3ERR_ISDIR。

25.4.2 NFS 的 `typedef` 声明

为了使结构声明更为清楚，NFS 协议标准为那些用于多个结构中的类型定义了名字。例如，filename3 被定义为一个字符数组，其长度足以包含一个路径分量名。在 RPC 的语法中，必须用关键字 `string` 声明一个字符数组。因此，上述声明为：

```
typedef string dirpath<MNTPATHLEN>;
```

声明 `dirpath` 是一个类型名，可用来声明值为目录路径的变量。同样，标准把 `fhandle3` 类型定义成一个 64 字节的数组，由它来装载文件句柄。该类型之所以被声明成 `opaque`，是因为客户不知道它的内部结构：

```
typedef opaque fhandle3<FHSIZE3>;
```

25.4.3 NFS 数据结构

有了常数和类型定义后，协议设计者就可以定义所有要使用的数据结构类型了。NFS 遵循了远程过程调用的习惯，把所有参数都组合到一个结构中。因此，该标准为每个远程过程定义了一个参数结构，每个过程的结果也有一个单独的结构。除此之外，该标准还定义了几个由一些过程共享的结构。如 24 章中所描述的 NFS `fattr3` 结构，NFS 用它来规定文件属性。`fattr3` 来源于 `stat` 结构所返回的数据。

fattr3 结构中的某些字段记录了文件最近一次被修改或存取的时间。某些字段被声明为类型 nfstime3，结构：

```
struct nfstime3 {           /* Date and time used by NFS      */
    uint32 seconds;          /* Seconds past epoch (1/1/70)      */
    uint32 nseconds;         /* Additional nanoseconds           */
};
```

声明指明 NFS 把时间值存储在一个 32 比特整数中。第一个整数记录离纪元时间有多少秒，第二个整数记录了微秒数（允许更精确）。NFS 用 1970 年 1 月 1 日^①作为纪元时间来度量时间值。

其他的多数声明定义了传给远程过程的参数或过程返回的结果的类型。例如，当调用一个对目录进行操作的过程时（例如删除某个文件），客户必须给出该文件的名字。NFS 声明参数类型是一个如下结构， diropargs3 (directory operation arguments for version 3，版本 3 的目录操作参数)：

```
struct diropargs3 {          /* Directory operation arguments   */
    nfs_fh3 dir;             /* Handle for directory file is in */
    filename3 name;           /* Name of file in that directory */
};
```

这个结构显示，参数是由该目录的文件句柄和目录中的某个文件名组成的。为了理解为什么需要 diropargs3，回忆一下上一章所讲的，所有的路径名是由 NFS 客户来分析的。因此，客户不能用全路径名来识别某个文件，NFS 要求给出该文件所驻留目录的句柄和该文件的名字，所有对文件的操作都要通过它们来指明。

除了声明参数类型以外，标准还定义了远程过程所返回的结果的类型。从目录中读取的操作的声明是最复杂的，因为它返回一个人口的列表，每个人口分别对应一个文件。表中的每个人口都具有以下形式：

```
struct entry3 {
    fileid3     fileid;          /* Identifier for this file        */
    filename3   name;            /* Name of the file                */
    cookie3     cookie;          /* Magic cookie for the entry      */
    entry3      *nextentry;       /* Pointer to next entry           */
};
```

以下声明指明 dirlist3 是人口列表的头部。布尔值 eof 用来指明该表中是否包括目录的最后一项（也就是说，没有其他项要读）。

```
struct dirlist3 {             /* A list of directory entries    */
    entry3      *entries;        /* Pointer to next entry          */
    bool        eof;            /* True if list is directory tail */
};
```

一旦声明了一个人口的列表，我们就可以使用该声明来指明 READDIR 操作成功或不成功后返回的值的类型：

^① 设计人员使 NFS 选择与 UNIX 系统相同的时间纪元。

```

struct REaddir3resok {           /* Value returned for success */
    post_op_attr dir_attributes; /* Attributes of directory */
    cookieverf3 cookieverf;     /* Value from previous call */
    dirlist3 reply;             /* List of directory entries */
};

struct REaddir3resfail {         /* Value returned for failure */
    post_op_attr dir_attributes; /* Attributes of directory */
};


```

最后，可以用 discriminated union 来声明一个类型，使其中包含成功或不成功的结果。

```
union REaddir3res switch (nfsstat3 status) { /* Value returned */
    case NFS3_OK:                                /*Used if status == NFS3_OK */
        REaddir3resok resok;
    default:                                     /*Used if status != NFS3_OK */
        REaddir3resfail resfail;
};
```

除了所有过程结果的声明外，标准还包括对参数的声明。例如，`read` 过程要求三个参数：文件的句柄、要读文件内的位移以及读到的 8 元组的计数器：

```
struct READ3args {           /* Argument to READ
    nfs_fh3 file;          /* Handle for the file to read
    offset3 offset;         /* Offset into the file
    count3   count;         /* Count of characters to read
};
```

类似地，结构 `WRITE3args` 指明在调用 `write` 向文件中写数据时的参数：

```
struct WRITE3args {           /* Argument to WRITE
    nfs_fh3 file;          /* Handle for the file to write
    offset3 offset;         /* Offset in file
    count3   count;         /* Count of characters to write
    stable_low stable;      /* Commit to disk before return?
    opaque    data<>;       /* Data to be written
};
```

过程 READLINK 允许客户读取一个符号链接（快捷方式）的内容。它的结果是用结构 READLINK3resok 定义的：

```
struct READLINK3resok {
    post_op_attr symlink_attributes; /* Attributes of link */  
    nfspath3 data; /* Path name found in link */  
};
```

25.5 NFS 过程

一旦声明了常数和数据类型后，我们余下的工作就是指明实现协议的远程过程了。NFS服务器提供一个实现 18 个过程的远程程序。该程序可用 RPC 语言声明如下：

```

program NFS_PROGRAM {
    version NFS_VERSION {
        void          NFSPROC3_NULL (void)           = 0;
        GETATTR3res   NFSPROC3_GETATTR (GETATTR3args ) = 1;
        SETATTR3res   NFSPROC3_SETATTR (SETATTR3args ) = 2;
        LOOKUP3res    NFSPRDC3_LOOKUP (LOOKUP3args)  = 3;
        ACCESS3res    NFSPROC3_ACCESS (ACCESS3args)  = 4;
        READLINK3res  NFSPROC3_READLINK (READLINK3args) = 5;
        READ3res      NFSPROC3_READ (READ3args)       = 6;
        WRITE3res     NFSPROC3_WRITE (WRITE3args)     = 7;
        CREATE3res    NFSPROC3_CREATE (CREATE3args)   = 8;
        MKDIR3res     NFSPROC3_MKDIR (MKDIR3args)    = 9;
        SYMLINK3res   NFSPROC3_SYMLINK (SYMLINK3args)= 10;
        MKNOD3res     NFSPRDC3_MKNOD (MKNOD3args)   = 11;
        REMOVE3res    NFSPROC3_REMOVE (REMOVE3args)  = 12;
        RMDIR3res    NFSPROC3_RMDIR (RMDIR3args)   = 13;
        RENAME3res    NFSPROC3_RENAME (RENAME3args) = 14;
        LINK3res      NFSPROC3_LINK (LINK3args)      = 15;
        REaddir3res   NFSPROC3_READDIR (REaddir3args) = 16;
        REaddirplus3res NFSPROC3_READDIRPLUS (REaddirplus3args) = 17;
        FSSTAT3res   NFSPROC3_FSSTAT (FSSTAT3args) = 18;
        FSINFO3res   NFSPROC3_FSINFO (FSINFO3 args) = 19;
        PATHCONF3res  NFSPROC3_PATHCONF (PATHCONF3args) = 20;
        COMMIT3res   NFSPROC3_COMMIT (COMMIT3args) = 21;
    } = 3;      /* current version of NFS protocol */
} = 100003; /* RPC program number assigned to NFS */

```

25.6 NFS 操作的语义

大多数 NFS 操作的语义遵循像 Linux 这样的系统的文件操作语义。下面分别描述 NFS 各个远程过程是如何操作的。

25.6.1 NFSPROC3_NULL (过程 0)

按照习惯，在任何 RPC 程序中过程 0 被称为空 (null)，因为它没有任何动作。应用程序可以调用它来测试某个服务器是否响应。

25.6.2 NFSPROC3_GETATTR (过程 1)

客户调用过程1来得到某个文件的属性，包括保护模式、文件拥有者、大小及最近存取时间等项。

25.6.3 NFSPROC3_SETATTR (过程 2)

过程2允许客户设置文件的某些属性。客户不能设置所有属性（例如，除非向文件中写入字节或截断文件，否则，它不能改变已记录的文件的大小）。如果调用成功，返回的结果包含了改变后文件的属性。

25.6.4 NFSPROC3_LOOKUP (过程 3)

客户调用过程3在一个目录中搜索某个文件。如果成功，则返回的值由该文件的属性及其句柄组成。

25.6.5 NFSPROC3_ACCESS (过程 4)

客户调用过程4来测试是否可以不传送整个项就访问该项。

25.6.6 NFSPROC3_READLINK (过程 5)

过程5允许客户从某个符号链接（快捷方式）中读值。

25.6.7 NFSPROC3_READ (过程 6)

过程6提供了最重要的功能之一，因为它允许客户从某个文件中读出数据。服务器返回的结果是一个联合。如果操作成功，结果包含了所要的数据及该文件的属性；如果操作失败，则状态值包含了一个差错代码。

25.6.8 NFSPROC3_WRITE (过程 7)

过程7提供了另一种基本功能：它允许客户向一个远程文件中写入数据。调用返回一个联合，若操作失败则包含一个差错代码，若操作成功则包含文件的属性。

25.6.9 NFSPROC3_CREATE (过程 8)

客户调用过程8在指定目录中创建一个文件。该文件不能存在，否则该调用将返回差错。调用返回一个联合，其中或者包含一个差错状态或者是新文件的句柄及其属性。

25.6.10 NFSPROC3_MKDIR (过程 9)

客户调用过程9来创建目录。如果调用成功，则服务器返回新目录的句柄及其属性的清单。如果调用失败，则返回的状态值表示失败的理由。

25.6.11 NFSPROC3_SYMLINK (过程 10)

过程 10 创建一个符号链接(快捷方式)。参数指明了一个目录句柄、要创建的文件名以及作为该符号链接内容的字符串。服务器创建该符号链接，然后返回一个状态值，或者表明成功或者给出失败的理由。

25.6.12 NFSPROC3_MKNOD (过程 11)

客户调用过程 11 来在服务器端创建一个特殊文件。参数指明了该文件的类型(例如，一个套接字)，过程返回新文件的句柄。

25.6.13 NFSPROC3_REMOVE (过程 12)

客户调用过程 12 来删除一个已存在文件。该调用返回一个状态值。该状态值或者指示操作成功或者给出一个差错代码说明调用为什么失败。

25.6.14 NFSPROC3_RMDIR (过程 13)

客户可用过程 13 删除目录。目录在被删除以前必须是空的。因此，为了移走整个目录分支，客户必须遍历整个分支，删除所有文件，然后把剩下的空目录删除。通常对目录树进行一遍后序遍历就可以删除其上的所有文件和目录。

25.6.15 NFSPROC3_RENAME (过程 14)

过程 14 允许客户为文件改名。因为参数使客户可以指定文件的新名字和新目录，所以 rename 操作对应着 UNIX 的 mv (move) 命令。NFS 保证 rename 在服务器上是原子操作(也就是说，它的执行不能被中断)。对原子性的保证非常重要，因为它意味着直到安装好文件的新名后才能把旧名删除。因此，在 rename 操作过程中，文件不会看上去像丢失了一样。

25.6.16 NFSPROC3_LINK (过程 15)

客户调用过程 15 来形成到一个已存在文件的硬链接。NFS 保证，如果一个文件有多个链接，那么无论用哪条链接对该文件进行存取，文件的可视属性都将是一致的。

25.6.17 NFSPROC3_READDIR (过程 16)

客户调用过程 16 从一个目录中读取其中的目录项。参数为将要读取的目录、魔块和要读取的最大字符数指定一个句柄。在最初的调用中，客户指明一个含零的魔块，让服务器从目录的最开始读起。返回的值包含零个或多个目录项的链表和一个布尔值，这个布尔值表示最后返回的目录项是否在目录的最后。

在成功返回之后，链表上的每个目录项都包含一个文件名、该文件的惟一标识符、一个给出该文件在目录中的位置的魔块以及一个指向链表中下一项的指针。

为了读取目录中的目录项序列，客户必须从调用 NFSPROC3_READDIR 开始，传给它一个参数为零的魔块和一个等于它的内部缓存大小的字符计数。服务器返回的目录项的个数将尽可能占满该缓存所能存放的数量。客户循环扫描链表中的每一项，分别处理每个文件名。如果返回值表明客户

已到达目录的末尾，它便停止处理。否则，客户用最后一项的魔块再一次向服务器发起一次调用，从而得到更多的目录项。客户继续读取目录项组直到达到该目录的结尾。

25.6.18 NFSPROC3_READDIRPLUS (过程 17)

过程 17 的工作与过程 16 类似，不同的是，它除了返回每个文件的名字以及句柄外，NFSPROC3_READDIRPLUS 还返回完整的信息。如果客户需要目录中所有文件的全部细节（例如文件的属性等），NFSPROC3_READDIRPLUS 比 NFSPROC3_READDIR 和 NFSPROC3_GETATTR 都快。

25.6.19 NFSPROC3_FSSTAT (过程 18)

过程 18 允许客户得到驻留有某个文件的文件系统的信息。返回结果中含有指明文件系统整个大小以及最大文件个数的字段。此外，过程还报告剩余空间大小以及文件槽的数量。

25.6.20 NFSPROC3_FSIONO (过程 19)

客户用过程 19 返回的信息对传送进行优化。例如，结果指明 READ、WRITE 和 READDIR 请求的最大和最优大小。另外，服务器指明它是否支持符号链接等特性。

25.6.21 NFSPROC3_PATHCONF (过程 20)

过程 20 允许客户从服务器得到 pathconf 信息。pathconf 信息包括诸如文件名中最大分量长度、名字是否区分大小写以及系统是否切断超过最大长度的文件名等信息。

25.6.22 NFSPROC3_COMMIT (过程 21)

过程 21 强迫服务器刷新缓存。也就是说，在该操作完成之前，所有以前写到文件中的数据都必须保存到稳定的存储器中去。结果，一旦 COMMIT 返回，客户就可以确定，即使服务器重新启动，数据也已经得到保护。

25.7 安装协议

第 24 章中描述的安装协议也是用 RPC 定义的。尽管 NFS 服务器必须伴有一个安装服务器，但是这两者被分别定义成两个独立的远程程序。因此，安装的协议标准也说明了一些常数、类型和一组组成服务器的远程过程。

25.7.1 安装协议的常数定义

尽管这两个协议是被分别定义的，但是安装协议的许多常数的值和类型都来源于 NFS 协议中相对应的常数。事实上，除非这两个协议的对象有相同的表示方法（如文件句柄），否则它们无法一起工作。例如，mount 定义了文件名、路径名和文件句柄大小，如下所示：

```
const MNTPATHLEN = 1024; /* Maximum characters in a path name */
const MNTNAMLEN = 255; /* Maximum characters in a name */
const FHSIZE3 = 64; /* Octets in an NFS file handle */
```

该声明使用 RPC 语法；它们每个都用长度来表示占用字节的数量。

25.7.2 安装协议的类型定义

安装协议还规定了类型定义，这些定义与它们在 NFS 中所对应的部分相一致。例如，安装协议规定文件句柄的类型为：

```
typedef opaque fhandle3<FHSIZE3>;
```

同样，安装协议还规定由一个字符数组组成的路径名：

```
typedef string dirpath<MNTPATHLEN>;
```

25.7.3 安装数据结构

因为安装协议已用 RPC 定义，所以它也遵循这样的习惯，即为每个远程过程的参数和结果声明一个数据结构。例如，协议中一个基本过程是返回一个在命名分层结构中的根目录的文件句柄。返回的值由一个联合结构 mountres3 组成，它被声明为：

```
union mountres3 switch (mountstat3 fhs_status) {
    case MNT_OK:                      /* If successful          */
        mountres3_ok mountinfo;         /* information for specified root */
    default:                           /* Otherwise               */
        void;                            /* nothing                */
};
```

正像在 NFS 协议中一样，安装协议中的每个远程过程都返回一个状态值及其他信息。如果操作失败，状态值指出失败原因。

除了一个返回文件句柄的过程以外，安装协议还提供了这样一个过程，该过程允许客户决定哪个文件系统可以存取。该过程把返回的结果放在一个名为输出表 (export list)^① 的链表中。输出表中节点的类型是用结构 exportnode 来声明的，同时还定义了指针类型的结构 exports：

```
typedef struct exportnode *exports;

struct exportnode {           /* List of available hierarchies */
    dirpath ex_dir;           /* Path name for this hierarchy */
    groups ex_groups;         /* Groups allowed to access it */
    exports ex_next;           /* Pointer to next item in list */
};
```

exportnode 节点的 ex_groups 字段中包含了一个指向链表的指针，该链表指明了哪个保护组有权对该命名的分层结构进行存取。链表中的节点被定义成类型为 groupnode 的结构，groups 是一个指向 groupnode 的指针：

```
typedef struct groupnode *groups;
```

^① 术语输出 (export) 指的是这样一种概念：一台服务器把它的某些文件输出给其他机器。

```

struct groupnode {           /* List of group names          */
    name   gr_name;        /* Name of one group          .      */
    groups gr_next;        /* Pointer to next item on list */
};


```

安装协议还允许客户决定某台给定机器可以对哪个远程文件系统进行存取。因此，可以为一组机器建立一个NFS交叉引用表。为此，可以询问这组机器中的每台机器，从而得到一个该机器正在进行存取的所有远程文件系统的清单。注意，正被某台给定机器存取的远程文件系统的集合，与该机器提供给其他机器存取的本地文件系统的集合是不相交的。

安装协议对远程存取列表的应答是一个各节点类型为mountbody的链表：

```

typedef struct mountbody *mountlist;

struct *mountbody {           /* List of remote mounts          */
    name     ml_hostname; /* Machine on which files reside */
    dirpath  ml_directory; /* Path name of hierarchy         */
    mountlist ml_nextentry; /* Pointer to next item on the list */
};


```

25.8 安装协议中的过程

像 NFS一样，安装协议定义了远程程序中的所有操作。安装协议程序的 RPC 声明如下：

```

program MOUNT_PROGRAM {
    version MOUNT_V3 {
        void          MOUNTPROC3_NULL(void)          = 0;
        mountres3    MOUNTPROC3_MNT(dirpath)        = 1;
        mountlist    MOUNTPROC3_DUMP(void)          = 2;
        void          MOUNTPROC3_UMNT(dirpath)        = 3;
        void          MOUNTPROC3_UMNTALL(void)       = 4;
        exports      MOUNTPROC3_EXPORT(void)        = 5;
    } = 3;                  /* Mount version 3 matches NFS vers. 3 */
} = 100005;              /* RPC program number assigned to mount */


```

25.9 安装操作的语义

安装协议定义了上面列出的各个操作的语义。下面将给出他们的简要说明。

25.9.1 MOUNTPROC3_NULL (过程 0)

遵循 RPC 的约定，过程 0 没有任何动作。

25.9.2 MOUNTPROC3_MNT (过程 1)

客户调用过程 1 来获得一个特定分层结构的句柄。参数包含有一个路径名，由于服务器可以输

出多个分层结构供客户存取，该参数用于对这些分层结构做出区分；结果的类型为 `fhstatus`。某台给定服务器上可供存取的分层结构的名字可以通过调用 `MOUNTPROC3_EXPORT` (见下所述) 得到。

25.9.3 MOUNTPROC3_DUMP (过程 2)

过程 2 使客户可得到某台特定机器正在使用的远程文件系统的清单。由 `MOUNTPROC3_DUMP` 所提供的信息对一般程序没有意义；它只对系统管理员有用。

25.9.4 MOUNTPROC3_UMNT (过程 3)

客户用过程 3 来提醒其他机器它将终止服务。例如，如果机器 A 已经从服务器 B 上安装了一个以上的文件系统，则机器 B 可用 `MOUNTPROC3_UMNT` 来通知 A 有某个文件系统将不再服务（例如，为了磁盘维护）。这样做使得在文件脱机时 A 可以不再向 B 发其他请求了。

25.9.5 MOUNTPROC3_UMNTALL (过程 4)

过程 4 允许一台机器告知另一台机器，它的所有 NFS 文件系统将不能使用了。例如，一个服务器可以告诉客户，在服务器重启动前，客户要把它所有的文件系统卸载（`umount`）。

25.9.6 MOUNTPROC3_EXPORT (过程 5)

过程 5 提供了一个重要的服务：它允许客户获得在某台给定服务器上可供存取的所有文件系统的清单。该调用返回一个链表，每个可供存取的文件系统在表中都有一个类型为 `exportlist` 的节点。客户在调用 `MOUNTPROC3_MNT` (过程 1) 的时候，必须使用在输出表中找到的某个目录路径的名字。

25.10 NFS 和安装鉴别

NFS 依赖安装来提供鉴别。安装协议鉴别某个客户对根目录句柄的请求是否合法，但是 NFS 并不对客户的每个请求都进行鉴别。

令人惊讶的是，安装协议并不提供更多的保护。它用 RPC 的 `AUTH_NONE` 来鉴别客户。一旦客户获得了根目录的句柄，对各个文件进行保护将变得没什么意义。例如，如果程序员得到在他们专用工作站的优先权，他们就可以对服务器及他们本地机器上的任意文件进行存取。而且，如果程序员可以猜出一个不透明文件句柄的内容，他便可以创建任意目录的句柄。

为使句柄更难以被猜测，大多数 NFS 都在句柄中对文件或目录信息进行编码。这些信息放在单独的字段中，避免目录句柄以及该目录中文件句柄之间有任何明显关系。早期版本以固定长度字段来提高句柄分析的效率。例如，图 25.1 说明了一个实现如何把 32 字节句柄划分成十个字段：

文件句柄使用固定格式的危险在于：NFS 对未授权存取只有很少的保护。某个想要存取文件的客户可以对任一 UNIX 文件创建句柄，从而避免运行安装协议。它也可以得到有关该文件的任意信息，包括文件拥有者及保护比特。因此，即使未经安装协议授权，它也可以存取任意可读的文件。而且，如果客户对本地机器具有超级用户特权，它也可以向服务器发出请求，而请求中含有任意的用户标识符。因此，具有超级用户特权的客户首先可以找出谁拥有某个特定文件，然后声明以该用户的名义发送请求。

字段	大小	内容
Fileid	4	该文件的主要和次要设备号
one	1	总为 1
length ₁	2	后面三个字段的总长度
zero ₁	2	总为 0
inode	4	该文件的索引节点号
igener	4	文件索引节点的生成号（为安全起见随机产生）
length ₂	2	后面三个字段的总长度
zero ₂	2	总为 0
inode	4	文件系统的根的索引节点号
rigener	4	根索引节点的生成号（为安全起见随机产生）

图 25.1 NFS 文件句柄的各字段的内容。这个特定格式来自在伯克利 UNIX 操作系统下运行的 NFS 服务器；并非所有服务器都用同样的方法创建文件句柄

为了使文件句柄更难以猜测，许多管理员采用了一种实用程序，该程序随机产生索引节点生成号。这种程序扫描磁盘上的所有索引节点，并在索引节点的生成字段中存放一个随机数。当安装协议传递文件句柄时，它将使用这个生成号，而 NFS 则检查句柄中的值是否与索引节点中的生成号匹配。

随机产生生成号增加了安全性，但它并没有增加创建一个句柄所需要的计算负载，因为随机化是在安装协议运行之间完成的。随机化也使文件句柄变得难以猜测。一个想要进行未经授权存取的客户必须在 232 种可能的值中选出生成号。因为猜测出有效句柄的可能性非常小，所以获得对某文件的未经授权存取的可能性也非常小。

25.11 文件加锁

由于版本 2 以上的 NFS 是无状态的，所以使用附加协议给文件加锁。这就是网络锁管理器 NLM (Network Lock Manager)，该协议允许客户以独占方式使用 NFS 安装的文件（也就是给该文件加锁），然后打开锁。尽管从 NFS 的版本 2 以后文件加锁的基本概念没有变化，但许多细节都有了变化。特别是，每个 NLM 过程都被扩充加入了一个协议版本号，避免新旧版本之间的模糊。

遗憾的是，NLM 版本号并不与 NFS 的版本号匹配。NLM 的版本 1 或 3 可以用在 NFS 的版本 2 中；NFS 的版本 3 要求使用 NLM 的版本 4。大概 NLM 版本 4 中最大的变化是把文件长度和偏移字段从 32 比特扩大到 64 比特。

25.12 NFS 第 3 版与第 4 版之间的变化

尽管在 NFS 第 3 版与第 4 版之间有许多差异，但是大多数的变化都可以划归到一个种类：安全。第 4 版完成时^①，将更好地对通信进行鉴别，确保没有未经许可的接入，同时对传送进行加密，确保数据的私有性。

① 在本书即将出版时，NFS 的第 4 版已经定义完成，只是还没有开始使用。

25.13 小结

当我们用 RPC 来定义协议时，必须提供规约中所用的常数和数据类型的定义、服务器提供的过程的定义、所有过程参数及其结果的类型定义以及每个过程的语义。定义协议与用 RPC 形成某个分布式版程序的区别在于：它要求设计者考虑抽象概念而不是一个已有的程序。

NFS 已用 RPC 定义。该协议标准规定由 21 个过程来组成一个服务器。除了允许客户读写文件的操作外，协议还定义其他的数据结构和操作，它们允许客户从目录中读取目录项、创建文件、删除文件、为文件改名或得到文件的信息等。

与 NFS 相伴的安装协议，它提供给客户鉴别，允许客户找到某个分层结构的根的句柄。安装协议允许某个给定服务器输出多个分层结构，它允许客户用全路径名来指定某个特定的分层结构。

NFS 的安全性依赖于安装协议。它假定一旦任何客户得到根目录的句柄，便可以发出存取请求。大多数 NFS 实现通过把有关该文件的信息编码来创建一个句柄。运行在 UNIX 系统下（如 Linux）的 NFS 服务器通常把文件的索引生成号编码在它的句柄中。为了避免客户在猜测出文件句柄后用它来获得未授权的存取，许多系统管理员使用生成随机索引生成的工具。随机性使客户很难猜测到一个有效的文件句柄。

NFS 的版本 4 有许多提高安全性的改变。尽管版本 4 已经定义了一个标准，但是许多实现仍然使用版本 2 或版本 3。

深入研究

Callaghan 等人的 [RFC 1813] 描述了 NFS 协议和使用 RPC 的安装协议的第 3 版。它声明了组成协议的常数、类型和过程，并且描述了语义以及实现的提示。Shepler [RFC 2624] 概述了第 4 版的一些设计思路。

习题

- 25.1 编写一个程序，用安装协议得到文件句柄。你的系统是否使用 UNIX 鉴别？
- 25.2 用上面的程序检查在一台给定的 Linux 服务器上的几个文件。看看管理员是否把服务器上的文件系统的索引节点生成号随机化了。
- 25.3 参考第 2 版的协议标准，找出 WRITECACHE 操作。该操作的目的是什么？
- 25.4 NFS 协议规约提到了几个不幂等（non-idempotent）操作。找出这些操作并解释它们为什么不幂等。
- 25.5 NFS 语义保证，在数据被存储到一个稳定存储设备（例如磁盘）之前，write 请求不会结束。如果协议允许服务器把数据复制到一个输出缓存，并允许过程调用不必等待缓存中的内容全部写入磁盘就可以返回。估计一下，在你的本地服务器执行一个 write 操作能快多少。
- 25.6 NFS 是设计用来允许客户和服务器在异构环境下操作的。解释一下符号链接会产生什么问题。（提示：仔细考虑协议，看看客户或服务器是否会中断符号链接。）
- 25.7 阅读协议规约，说出客户能够设置哪些文件属性。

- 25.8 如果 NFS 在 UDP 上使用 RPC，则一个远程过程调用可以被复制、延迟或失序交付。解释一下，如何用一组在客户端出现的 NFS 调用来创建文件、往其中写数据，但其结果却是一个长度为零的文件。
- 25.9 假定一个客户调用 NFSPROC3_FFSTAT 来查找合适的数据传送长度。什么限制可能会造成用该长度传输不是最优的？
- 25.10 阅读协议规约，找到有关作废的文件句柄的内容。为什么服务器会声明某个文件句柄已作废？使句柄作废如何能提高安全性？
- 25.11 编写一个程序，它调用服务器 S 上的 MOUNTPROC3_UMNTALL 过程。该调用对后继的对 S 的调用会产生影响吗？为什么？
- 25.12 编写一个程序，该程序调用 MOUNTPROC3_EXPORT。运行该程序并把输出文件系统的清单打印出来。

第 26 章 TELNET 客户（程序结构）

26.1 引言

前面，我们用简单的例子说明了许多客户-服务器软件中所使用的概念和技术。本章和下一章将探讨如何把客户-服务器范例应用在复杂的应用协议上。我们用的示例协议是TELNET，它是TCP/IP 协议族中应用最为广泛的协议。

本章侧重于整体的程序结构。我们假定读者熟悉TELNET的基本知识，因而本章的重点在于解释它的一个实现。本章讨论了客户软件的设计、过程/线程结构以及用有限状态机来控制处理。本章还回顾了 UNIX 终端 I/O，并说明客户如何把 TELNET 通信映射为 UNIX 终端。

下一章将完成整个描述。它将侧重于过程的细节，对这些过程的调用将完成一些语义动作，这些语义是与有限状态机的状态转移相关联的。这两章中的示例代码很清楚地表明，编程细节决定了程序的代码，它还表明这些细节使实现复杂化了。

26.2 概述

26.2.1 用户终端

TELNET协议定义了一个交互通信的工具，它允许用户与远程机器上的服务器进行通信。在多数例子中，用户用 TELNET 与远程注册服务（remote login service）进行通信。如第 1 章中的例子所示，设计良好的 TELNET 客户还允许用户与其他服务相联系。

TELNET协议定义了一种交互的、面向字符的通信。协议说明了由键盘和显示屏组成的网络虚拟终端 NVT（network virtual terminal）。它为虚拟终端定义了字符集。有几个键对应一些概念性操作，而不是数据值。例如有一个键可引起中断（interrupt）或异常中止（abort）。使用网络虚拟终端的主要优点是它允许客户从各种计算机上与某个服务相连接。正如第 20 章中所描述的 XDR 标准一样，TELNET 使用了一种对称的数据表示。当每个客户发送数据时，把它的本地终端的字符表示映射到 NVT 的字符表示上，当接收数据时，又把 NVT 的表示映射到本地字符集合上。总而言之：

TELNET 是面向字符的协议，在传送数据时使用标准编码。

26.2.2 命令和控制信息

除了字符数据外，TELNET 还允许客户同服务器交换命令（command）或控制（control）信息。因为客户和服务器之间的所有通信都是通过一个TCP连接进行的，所以协议特别安排了对命令或控制信息的编码，使接收方可以把它们从正常的数据中区分出来。因此，协议的许多地方都侧重于定义发送方如何对命令编码，以及接收方如何识别它们。

26.2.3 终端、窗口和文件

TELNET 定义了在用户终端和远程服务之间的通信。该协议规约假定终端是由键盘和显示屏组成的，其中，用户可用键盘来输入字符，用显示屏来显示多行文本。

在实际应用中，用户可以决定是否调用客户，它可用输入文件取代键盘或用输出文件取代显示屏。另外，用户可以从位图显示器的一个窗口（window）内调用客户程序。尽管这种方法又引入了少量代码，但是，如果我们想像调用客户的每个用户有一个传统的终端，这对理解协议及其实现来说，是最简单的方法了。总而言之：

TELNET 客户软件是设计用来处理与用户终端的交互通信的。

26.2.4 对并发性的需要

从概念上说，TELNET 客户在用户终端和远程服务之间传送字符。一个方面，当它与用户终端交互时使用本地操作系统函数；另一方面，当它与远程服务通信时使用一个 TCP 连接。图 26.1 说明了这个概念：

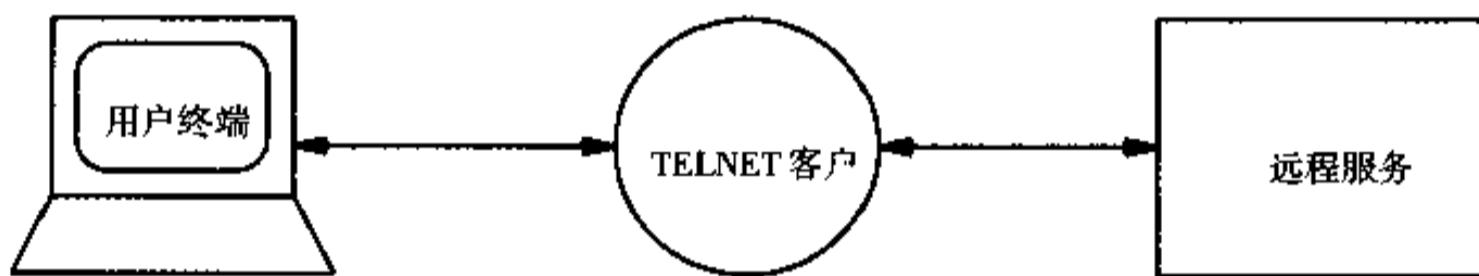


图 26.1 TELNET 客户的概念性作用。该客户必须把字符从用户的键盘上传送到远程服务上，它还必须把字符从远程服务传送到用户的显示屏上

为了在用户终端和远程服务之间提供全双工连接，TELNET 客户必须同时完成两个任务：

- 客户必须读取用户在键盘上键入的字符，并在 TCP 连接上把它们发送到远程服务上去。
- 客户必须读取从 TCP 连接上到达的字符，并把它们显示在用户的终端屏幕上。

因为远程服务可以在任何时刻发出输出，而且用户也可以在任何时刻键入，所以客户不知道哪个数据源的数据会首先到达。因此，它不能无限地等待从其中的某个输入源中输入数据而不检测另一个输入源的输入。简而言之，客户必须并发地向两个方向传送数据。

26.2.5 TELNET 客户的过程模型

为了适应并发的数据传送，客户必须由多个并发执行的过程组成，或者在一个过程中实现并发 I/O。我们的示例代码采用了后面的策略：客户由一个过程组成，它一直阻塞，直到某个数据源准备好为止。图 26.2 说明了我们所选用的过程结构：

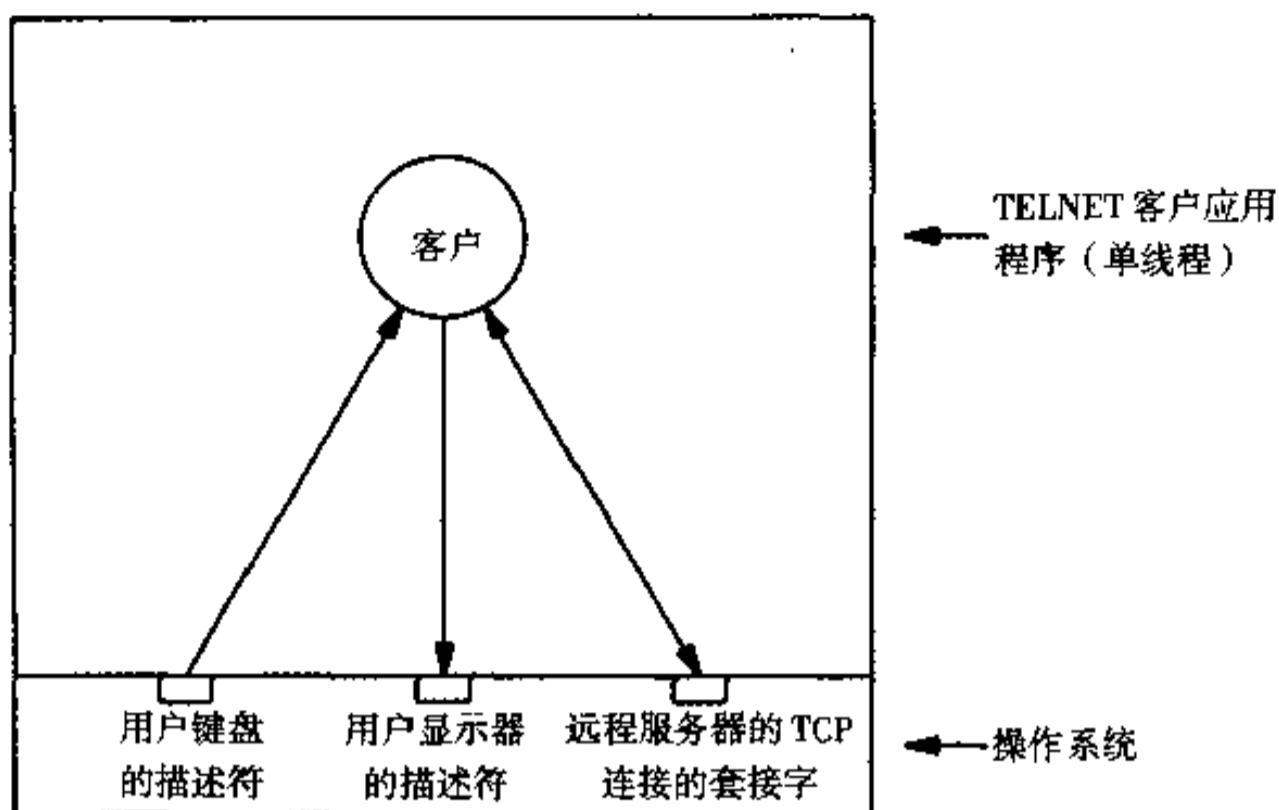


图 26.2 TELNET 客户过程结构示例。单个过程从用户的键盘读取字符，并通过 TCP 连接把它们发送到远程服务上去。该过程还从远程服务读取字符并把它们发给用户的显示器

26.3 TELNET 客户算法

算法 26.1 说明了单进程的 TELNET 客户是如何运行的。与 13 章描述的单进程、并发服务器设计一样，单进程、并发客户程序的 Linux 实现使用 select 系统函数来实现算法 26.1 中的步骤 3。当客户调用 select 时，它指明该线程必须阻塞，直到有输入到达 select 所指定的描述符，这些描述符或者与用户键盘相对应、或者与 TCP 连接的套接字描述符相对应。当这两者中的任何一个准备就绪时，select 调用将返回，客户便可从任一准备好的描述符中读取数据。

算法 26.1

1. 分析参数并对数据结构初始化。
2. 打开到指定远程主机上的指定端口的 TCP 连接。
3. 阻塞，直到用户键入或数据到达该 TCP 连接。
4. 如果数据从键盘到达，便读取数据并对其进行处理，把它翻译成 NVT 表示，并在 TCP 连接上将其传送出去。否则，从 TCP 连接上读取数据并进行处理，把它翻译成本地字符表示，并把这些数据传送到用户的显示器上。
5. 返回第 3 步。

算法 26.1 TELNET 客户。执行的单进程在两个方向上传送数据；
它一直阻塞，直到从键盘或套接字中可得到数据为止

26.4 Linux 中的终端 I/O

算法 26.1 看起来可能太简单了。但是，TELNET 协议的细节以及终端 I/O 却使代码非常复杂。为了理解 TELNET 客户软件是如何与用户终端进行交互的，我们必须理解客户的操作系统是何处处

理终端 I/O 的。本节将描述 Linux 是如何处理终端 I/O 的，并将给出控制用户终端的代码的例子。

在 Linux 中，对终端设备的 I/O 遵循着与对文件或套接字 I/O 相同的打开 - 读 - 写 - 关闭范例。应用程序调用 open 得到一个终端的键盘或显示器的 I/O 描述符，调用 read 来接收用户从键盘键入的数据，或者调用 write 把数据传送到终端的显示器上。

在实践中，大多数应用程序并不用 open 来创建用户终端的描述符，因为命令解释器自动提供打开的描述符。命令解释器通常把键盘与描述符 0 联系起来（标准输入），而把终端显示器与描述符 1 联系起来（标准输出）。

尽管终端 I/O 遵循与文件 I/O 相同的基本范例，但是大量的细节问题却使它很复杂。许多细节是由于终端硬件很原始。例如，尽管大多数用户把键盘和显示器看成一个单独的单元，但是硬件把键盘输入与显示器输出区别对待，把它们看成两台独立的设备。因此，硬件并不自动地显示用户键入的每个字符。相反，硬件仅把键盘输入发送出去，并要求计算机在显示器上写出每个字符的副本。

大多数应用程序希望用户在显示器上看到他们所键入的每个字符的副本，但有的应用需要禁止打印输入。具体来说，为了避免口令被偷看，需要输入口令的程序通常在用户输入口令时停止显示输入字符。

让每个应用程序处理所有终端 I/O 和字符显示的细节会很不便，因此，Linux 操作系统提供了自动处理这些内容的软件。这种软件称为终端设备驱动器（terminal device driver），该软件驻留在如图 26.3 所示的操作系统内核中。

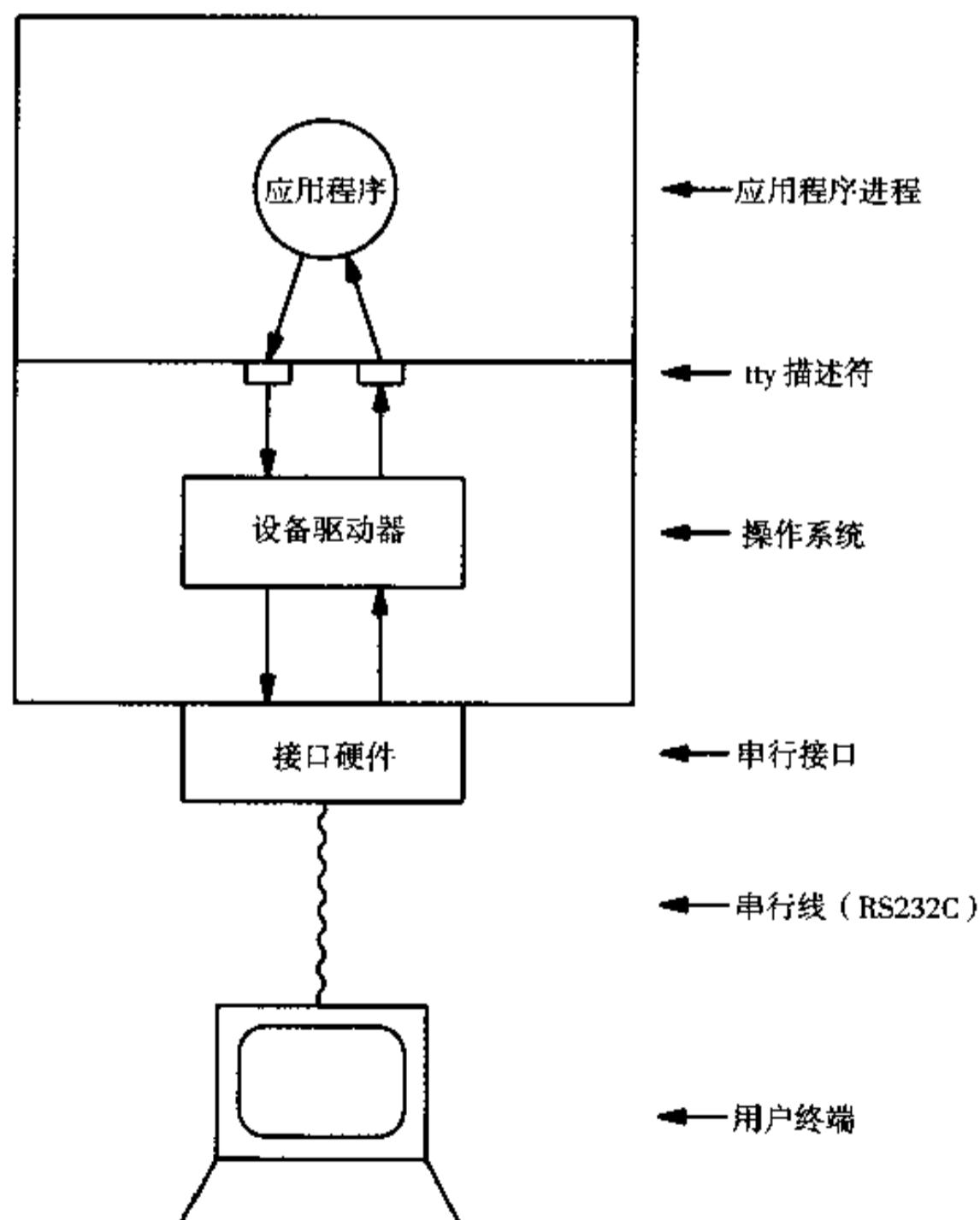


图 26.3 Linux 中的终端设备驱动器以及数据流的路径，数据从键盘流入系统或从系统流出到终端显示器。所有终端 I/O 通过设备驱动器传送

设备驱动器把键盘输入与对应的终端显示器联系起来。它可以被命令回显每个输入字符(即显示用户键入的每个字符),或者禁止回显(即关闭显示输入字符)。它允许用户用退格键或删除键来编辑错误,或者直接把所有字符(包括 delete 和 backspace)传送给应用程序。当程序发送一个行结束符时大多数终端要求操作系统发送一个 linefeed(换行)以及一个 carriage return(回车)字符来把光标移到下一行的开始。linefeed 把显示屏垂直向下移动, carriage return 移到当前行的开始。, 驱动器也可以生成一个特殊的字符序列, 把光标放在新行的开始。最后, 设备驱动器能识别一个(或多个)特殊字符, 它们使系统中断(interrupt)或异常中止(abort)当前进程。当用户键入该特殊字符时, 驱动器把它转换成一个信号(signal), 该信号把当前应用程序异常中止。

26.4.1 控制设备驱动器

当一个 TELNET 客户开始执行的时候, 它需要改变终端的行为(例如, 这样做可以使它能把 abort 键读成数据)。客户将调用操作系统函数 tcgetattr 来得到所有驱动器参数的副本; 然后便调用系统函数 tcsetattr 来指定新的参数; 最后, 在客户程序退出或挂起之前, 它调用 tcsetattr 来恢复原来的参数值。因此, 从用户的观点来看, 尽管在客户操作时, 终端参数和某些键的意义改变了, 但当客户退出或挂起后, 它们仍然返回它们的初始值。

过程 tcgetattr 从与键盘相关的驱动器(设备 0)中提取当前的参数值, 并把它们存放在一个类型为 termios 的结构中。过程 tcsetattr 把事前存放在 termios 结构中的参数值传送给驱动器。这里, 每个过程都要有一些参数, 这些参数将指明设备、要完成的操作以及所使用的结构的地址。

26.5 建立终端模式

当客户开始启动时, 它调用过程 ttysetup 为终端设备驱动器建立参数。文件 ttysetup.c 中包含了代码:

```
/* ttysetup.c - ttysetup */

#include <unistd.h>
#include <termios.h>
#include <stdio.h>
#include <string.h>

#include "local.h"

/*
 * ttysetup - set up tty
 */
int
ttysetup(void)
{
    extern struct termios          tntty;

    if (tcgetattr(0, &oldtty) < 0)      /* save original tty state */
        /* ... */
}
```

```

        errexit("can't get tty modes: %s\n", strerror(errno));

    sg_erase = oldtty.c_cc[VERASE];
    sg_kill = oldtty.c_cc[VKILL];
    t_intrc = oldtty.c_cc[VINTR];
    t_quitc = oldtty.c_cc[VQUIT];
    t_flushc = oldtty.c_cc[VDISCARD];

    tntty = oldtty;           /* make a copy to change */

    /* disable some special characters */
    tntty.c_cc[VINTR] = _POSIX_VDISABLE;
    tntty.c_cc[VQUIT] = _POSIX_VDISABLE;
    tntty.c_cc[VSUSP] = _POSIX_VDISABLE;
#ifdef VDSUSP
    tntty.c_cc[VDSUSP] = _POSIX_VDISABLE;
#endif

    if (tcsetattr(0, TCSADRAIN, &tntty) < 0)
        errexit("can't set tty modes: %s\n", strerror(errno));
}

```

ttysetup 调用 tcgetattr，在全局变量 oldtty 中记录原来的终端模式。因为 tcgetattr 只允许程序一次性提取或装载所有参数，所以客户不能分别设置各个参数。为了改变某个或某几个参数，客户必须把原来的参数复制到本地变量 tntty 中，然后，对副本进行修改以改变其中一个或几个参数，接着便调用 tcsetattr 把新的参数存放在终端设备驱动器上。ttysetup 中的这些修改使引起中断 (interrupt)、退出 (quit)、挂起 (suspend) 和延迟挂起 (delayed suspend) 的字符失效。从本质上说，把指明的值设成 _POSIX_VDISABLE 避免本地设备驱动器中断或挂起客户程序。因为 ttysetup 只是在终端原有参数的副本上进行的修改，所以它并不改变终端的其他特性。例如，如果用户已经定义可以像读其他字符一样读 control-C。但是，因为 ttysetup 是从参数的副本开始进行的，所以，使对 control-C 的解释失效并不影响终端的其他特性（例如回显）。我们将看到，客户使用保存的终端模式来确定用户已经定义哪个字符作为中断信号，当用户键入该字符时，它便向服务器端发送中断命令。

26.6 用于保存状态的全局变量

文件 local.h 包含了对全局变量 oldtty 以及其他用于本地终端的全局变量的声明。

```

/* local.h */

#include <termios.h>

extern FILE      *scrfp;
extern char      scrname[];
extern struct termios   oldtty;

```

```

extern char          t_flushc, t_intrc, t_quitc, sg_erase, sg_kill;

extern int           errno;

int     errexit(const char *format, ...), cerrexit(const char *format, ...);
int     ttwrite(FILE *sfp, FILE *tfp, unsigned char *buf, int cc);
int     sowrite(FILE *sfp, FILE *tfp, unsigned char *buf, int cc);
int     fsmbuild(void);

```

代码把变量 oldtty 声明成一个 termios 结构，系统为该结构定义了六个字段：

```

struct termios {
    tcflag_t   c_iflag;      /* terminal input modes */
    tcflag_t   c_oflag;      /* terminal output modes */
    tcflag_t   c_cflag;      /* terminal control modes */
    tcflag_t   c_lflag;      /* line discipline modes */
    char       c_line;       /* line discipline */
    cc_t       c_cc[17];     /* control characters */
};

```

26.7 在退出之前恢复终端模式

当客户开始执行时，它在变量 oldtty 中保存了原来的终端模式。当它退出时，客户进程调用 ttyrestore 来恢复保存的终端模式。例如，过程 dcon 用于切断与某个服务器的 TCP 连接；当用户要求终止连接时，客户便调用 dcon。dcon 打印一个报文、恢复终端模式，接着便正常退出（使用退出代码 0）。文件 dcon.c 包含了这些代码：

```

/* dcon.c - dcon */

#include <stdlib.h>
#include <stdio.h>
#include <termios.h>

#include "local.h"

/*
 * dcon - disconnect from remote
 */
int
dcon(FILE *sfp, FILE *tfp, int c)
{
    fprintf(tfp, "disconnecting.\n");
    (void) tcsetattr(0, TCSADRAIN, &oldtty);
}

```

```

    exit(0);
}

```

在整个代码中，如果出现严重错误，将存在客户。在它存在之前，客户必须恢复终端模式，否则用户的终端可能无法正常工作。文件 cerrexit.c 包含该代码。

```

/* cerrexit.c - cerrexit */

#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>

#include "local.h"

/*
 * cerrexit - cleanup and exit with an error message
 */
int
cerrexit(const char *format, ...)
{
    va_list args;

    va_start(args, format);
    vfprintf(stderr, format, args);
    (void) tcsetattr(0, TCSADRAIN, &oldtty);
    va_end(args);
    exit(1);
}

```

像过程 dcon一样，cerrexit 打印一个报文，并在退出之前恢复终端模式。与 dcon 不同的是，cerrexit 用退出码 1 表示一个异常退出。

26.8 客户挂起与恢复

Linux 允许用户暂时挂起一个正在运行的程序。在挂起期间，对用户终端的控制将转给另一个进程，这个进程通常是一个命令解释器。当 TELNET 客户接收到一个引起进程挂起的信号时，在挂起之前，它必须恢复原来的终端参数。在该客户进程又恢复运行时，必须要把参数重新设置为客户所使用的值。文件 suspend.c 中包含了函数 suspend 的代码，该函数在挂起客户进程之前恢复原来的终端模式。当客户恢复执行时，suspend 重新得到控制权，重新设置终端模式并返回到它的调用者。

```

/* suspend.c - */

#include <sys/types.h>

```

```
#include <sys	signal.h>

#include <stdio.h>
#include <string.h>

#include "local.h"

extern struct termios          tntty;

/*
 * suspend - suspend execution temporarily
 */
int
suspend(FILE *sfp, FILE *tfp, int c)
{
    if (tcgetattr(0, &tntty) < 0) /* save current tty state */
        erexit("can't get tty modes: %s\n", strerror(errno));
    if (tcsetattr(0, TCSADRAIN, &oldtty) < 0) /* restore old state */
        erexit("can't set tty modes: %s\n", strerror(errno));

    (void) kill(0, SIGTSTP);

    if (tcgetattr(0, &oldtty) < 0)      /* may have changed */
        erexit("can't get tty modes: %s\n", strerror(errno));
    if (tcsetattr(0, TCSADRAIN, &tntty) < 0) /* back to telne modes */
        erexit("can't set tty modes: %s\n", strerror(errno));
    return 0;
}
```

26.9 有限状态机的规约

TELNET协议规定了客户如何把字符传给远程服务,以及客户如何显示该远程服务所返回的数据。在连接上所进行的大多数通信业务都是由各个数据字符组所组成的。在客户端,这些数据字符来自用户从键盘键入的数据;在服务器端,它们来自远程会话所产生的输出。除了数据字符以外,TELNET还允许客户和服务器交换控制信息。具体来说就是,客户可以向服务器发送字符序列,该字符序列构成了控制远程服务执行的命令。例如,客户可以发送一个中断远程应用程序的命令序列。

大多数TELNET的实现用有限状态机(finite state machine, FSM)来说明命令序列的确切语法以及对命名序列的解释。作为规约工具,有限状态机提供了对协议的精确描述。它确切地说明了发送方如何在数据流中嵌入命令序列,同时,它也确切说明了接收方如何解释这种序列。更为重要的是,有限状态机可以直接转换成遵从该协议的程序。因此,可以验证最终的程序是否符合协议规约。总而言之:

因为TELNET是一个面向字符的协议，它把命令序列嵌入到客户与服务器之间的数据流中，大多数实现使用有限状态机来定义正确的行为。

26.10 在TELNET数据流中嵌入命令

TELNET所蕴涵的思想是简单的：无论何时，只要客户或服务器要发送命令序列而非正常的数据，它就在数据流中插入一个特殊的保留字符。该保留字符称为“解释为命令”(IAC, Interpret As Command)字符。当接收方在一个传入数据流中发现IAC字符时，它就把后继的字节处理为命令序列。当要把IAC作为数据发送时，发送者将在它前加额外的IAC字符。

一个单独的命令序列可包含可选请求(option request)或可选回答(option reply)。请求询问接收方兑现(或不兑现)某个特殊的TELNET选项；回答则对请求进行确认，并指明接收方是否兑现该请求。

协议定义了两个发送方用来形成请求的动词：DO和DONT。像大多数TELNET定义的项一样，协议标准规定，每个动词及每个选项都必须用单个字符编码，因此，出现在数据流中的请求通常是由三个字符组成的：

IAC verb option

其中verb表示DO或DONT的编码字符，option表示某个TELNET选项的编码字符。

TELNET的echo option(回显选项)提供了很好的例子。正常情况下，服务器回显它所接收到的每个字符(也就是说，把一份副本发回到用户的显示器上)。为了关闭远程字符回显，客户发出三个与下面对应的编码字符：

IAC DONT ECHO

26.11 选项协商

一般来说，接收方要用动词WILL或WONT来响应请求。接收方发送WILL以指明它将兑现所要求的选项，或者用WONT指明它不兑现该选项。

这种对请求的响应为请求的发送方提供了确认，它告诉发送方接收方是否同意兑现该请求。例如，启动时，客户和服务协商决定哪一方回显用户键入的字符。通常情况是，客户向服务器发送字符而服务器将其回显到用户的终端上。但是，如果网络的时延会引起一些麻烦时，用户可能更愿意让本地系统回显字符。在客户允许本地系统回显前，它要向服务器发送以下序列：

IAC DONT ECHO

服务器收到请求后，发出3个字符的响应：

IAC WONT ECHO

注意动词WONT指的是选项；它不一定意味着服务器拒绝该请求。例如，在这种情况下，服务器已经按请求同意关闭回显。

26.12 请求 / 提供的对称性

有趣的是，TELNET 允许连接的一方在另一方提出请求之前，就提供某个特定的选项。为了做到这一点，提供执行（或不执行）这种选项的一方要发送一个包含动词 WILL 或 WONT 的报文。因此，WILL 或 WONT 要么是对前面的请求的确认，要么是执行该选项的供给。例如，像文本编辑器这样的应用程序，往往要发送特殊的控制序列来定位光标。他们不能用网络虚拟终端来编码，因为它不支持所有可能的 8 比特字符。因此，大多数系统的 TELNET 服务器会自动地发送传输二进制（transmit binary）选项的 WILL，以便能用 8 比特二进制（未编码）字符传送而并非用 NVT 编码。客户必须发出一个指定 DO transmit binary 或 DONT transmit binary 的命令序列来进行响应。

26.13 TELNET 字符定义

文件 telnet.h 中包含了协议中所用常数的定义：

```
/* telnet.h */
typedef unsigned char u_char;

/* TELNET Command Codes: */
#define TCSB          (u_char)250    /* Start Subnegotiation      */
#define TCSE          (u_char)240    /* End Of Subnegotiation     */
#define TCNOP          (u_char)241    /* No Operation               */
#define TCDM          (u_char)242    /* Data Mark (for Sync)      */
#define TCBRK          (u_char)243    /* NVT Character BRK         */
#define TCIP          (u_char)244    /* Interrupt Process          */
#define TCAO          (u_char)245    /* Abort Output               */
#define TCAYT          (u_char)246    /* "Are You There?" Function */
#define TCEC          (u_char)247    /* Erase Character            */
#define TCEL          (u_char)248    /* Erase Line                 */
#define TCGA          (u_char)249    /* "Go Ahead" Function        */
#define TCWILL         (u_char)251    /* Desire/Confirm Will Do Option */
#define TCWONT         (u_char)252    /* Refusal To Do Option       */
#define TCDO           (u_char)253    /* Request To Do Option       */
#define TCDONT         (u_char)254    /* Request NOT To Do Option  */
#define TCIAC          (u_char)255    /* Interpret As Command Escape */

/* Telnet Option Codes: */
#define TOTXBINARY    (u_char) 0    /* TRANSMIT-BINARY option     */
#define TOECHO         (u_char) 1    /* ECHO Option                */
#define TONOGA          (u_char) 3    /* Suppress Go-Ahead Option   */
#define TOTERMTYPE     (u_char) 24   /* Terminal-Type Option        */

/* Network Virtual Printer Special Characters: */
#define VPLF           '\n'    /* Line Feed                  */
```

```

#define VPCR          '\r' /* Carriage Return      */
#define VPBEL         '\a' /* Bell (attention signal) */
#define VPBS          '\b' /* Back Space           */
#define VPHT          '\t' /* Horizontal Tab       */
#define VPVT          '\v' /* Vertical Tab          */
#define VPFF          '\f' /* Form Feed             */

/* Keyboard Command Characters: */
#define KCESCAPE      035 /* Local escape character ('^]') */
#define KCDCON        '.' /* Disconnect escape command */
#define KCSUSP        032 /* Suspend session escape command('^Z') */
#define KCSCRIPT       's' /* Begin scripting escape command */
#define KCUNSCRIPT    'u' /* End scripting escape command */
#define KCSTATUS       024 /* Print status escape command ('^T') */
#define KCNL           '\n' /* Newline character      */

#define KCANY          (NCHRS+1)

/* Option Subnegotiation Constants: */
#define TT_IS          0      /* TERMINAL_TYPE option "IS" command */
#define TT_SEND         1      /* TERMINAL_TYPE option "SEND" command */

/* Boolean Option and State variables */
extern char        synching, doecho, sndbinary, rcvbinary;

```

注意，文件为TELNET所使用的每个字符都定义了符号名，包括像WILL和WONT这样的动词以及选项代码。

26.14 针对来自服务器数据的有限状态机

图26.4展示了一个说明TELNET协议的理论有限状态机，包括与上面所述的选项协商相对应的状态。我们可以把该状态机看作是指明了客户如何处理从服务器收到的字符序列。

有限状态机的状态转移图使用了传统的表示方法。从一个状态转换到另一状态都有一个形如 α/β 的标志，其中 α 表示引起该转移的特定输入字符， β 表示发生该转移所完成的动作。从状态X转移到状态Y上如果有标志 α/β ，表示如果在状态X时字符 α 到达，执行动作 β ，然后转换到状态Y。状态名及图中的字符都来自软件。例如，文件telnet.h定义常数TCIAC与TELNET中的IAC字符相对应。为简单起见，TCANY表示那些已明确列出的转移以外的任意其他字符。

为了理解有限状态机是如何工作的，让我们想像一下这样的情况：当来自服务器的数据从TCP连接上到达时，客户是怎样使用这种有限状态机的呢？当字符从服务器到达时，客户就按照有限状态机的说明进行状态转移。有的转移使状态机保持同一状态，而另一些转移则使它转变到一个新的状态。

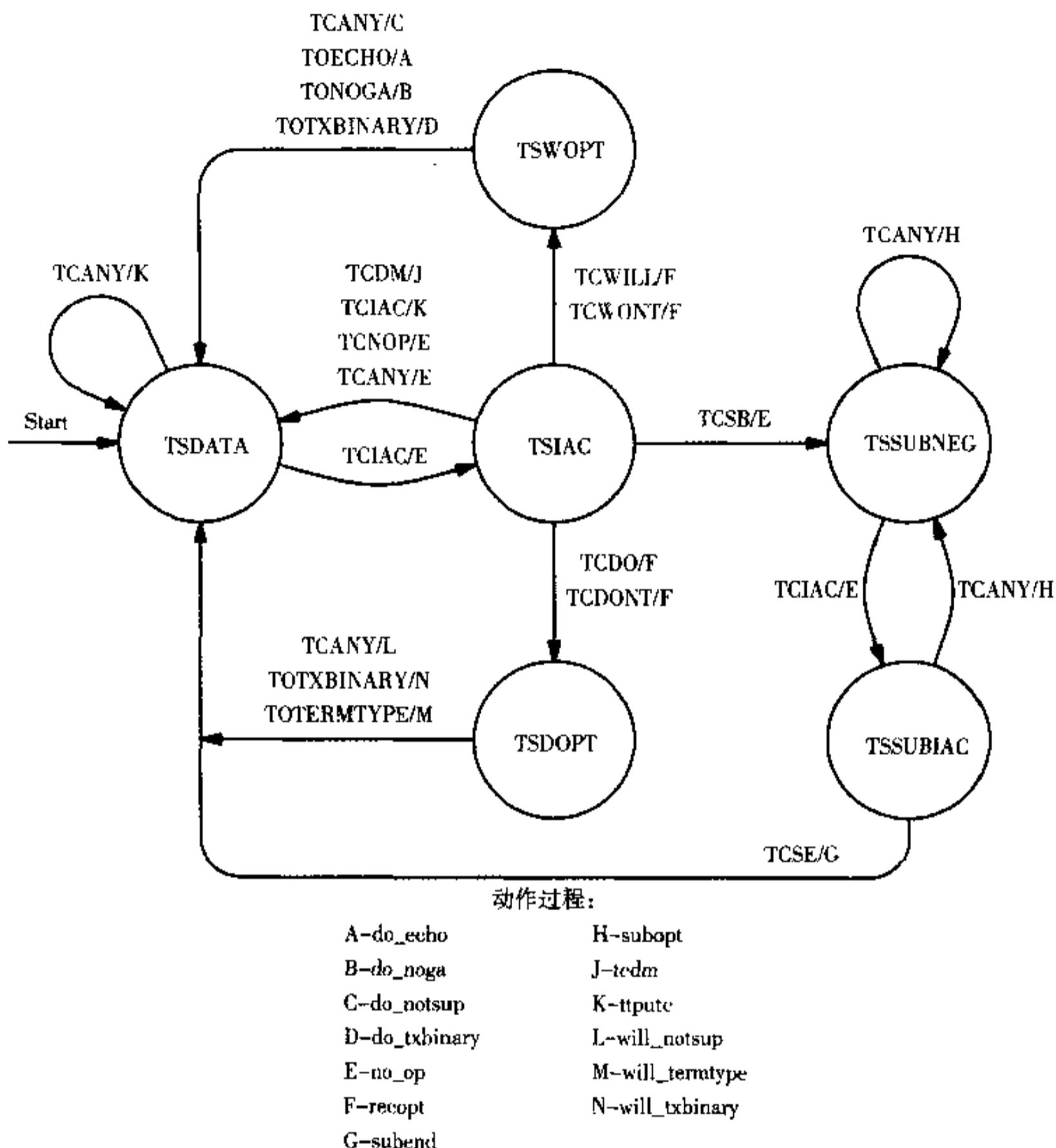


图 26.4 有限状态机，它描述 TELNET 如何在数据中编码命令序列。状态和字符名直接来自软件。TCANY 代表“除了明确说明以外的其他任意字符”

26.15 在各种状态之间转移

当客户开始执行的时候，它从标有 TSDATA 的状态开始。状态 TSDATA 对应着一种情况：客户希望接收正常的字符并把它们发送到用户的显示器上（即客户还没有开始读命令序列）。例如，如果字符 q 到达，则客户保持在状态 TSDATA，并执行动作 K（即客户调用过程 ttpute，把字符显示在用户的终端屏幕上，然后沿着回路返回到同一状态）。

如果在有限状态机处于状态 TSDATA 时字符 TCIAC 到达，则客户转移到状态 TSIAC，并执行图中所标的动作 E。该说明指出动作 E 对应着“无操作”。一旦客户转到状态 TSIAC，它便开始解释命令序列了。如果跟随 TCIAC 之后的字符是一个动词（如 TCDO），客户将转移到某个处理选项状态。

TELNET 的有限状态机只需要六个状态，因为协议的解释只依赖于字符到达的那段简短历史。

例如，在 TCIAC 字符之后，服务器可能发送某个选项请求或响应：TCDO、TCDONT、TCWILL、TCWONT，或者可能发送选项子协商请求（option subnegotiation request）。选项子协商允许发送方在选项中包含可变长度的字符串（例如，客户用于向服务器传送终端类型的选项可用于协商。这样，它可以发送含有该终端名的编码的字符串）。尽管子协商允许可变长度的命令序列，但是有限状态机仅需要 2 个状态来处理它，因为一个 2 字符的序列就终止子协商了。当客户先遇到子协商请求时，它便进入状态 TSSUBNEG；当接收到字符 TCIAC 时，它便进入状态 TSSUBIAC；如果字符 TCSE 紧跟其后，它便退出子协商。如果有任何其他的 2 字符序列出现，有限状态机仍停留在状态 TSSUBNEG。

26.16 有限状态机的实现

由于构造有限状态机的有效实现是可能的，也由于这种机器能够很容易地描述面向字符的协议，所以我们的示例代码用了三个独立的有限状态机。一个状态机控制客户如何对来自键盘的字符进行响应，另一个控制客户如何机处理在 TCP 连接上到达的来自服务器的字符，第三个状态机处理选项子协商的细节。所有这三个有限状态机都使用同样的数据结构，这使操作这些数据结构的某些过程可以被共享。

为了能高效地处理，我们的实现把有限状态机的转移编码成一个如图 26.5 所示的转移矩阵（transition matrix）。

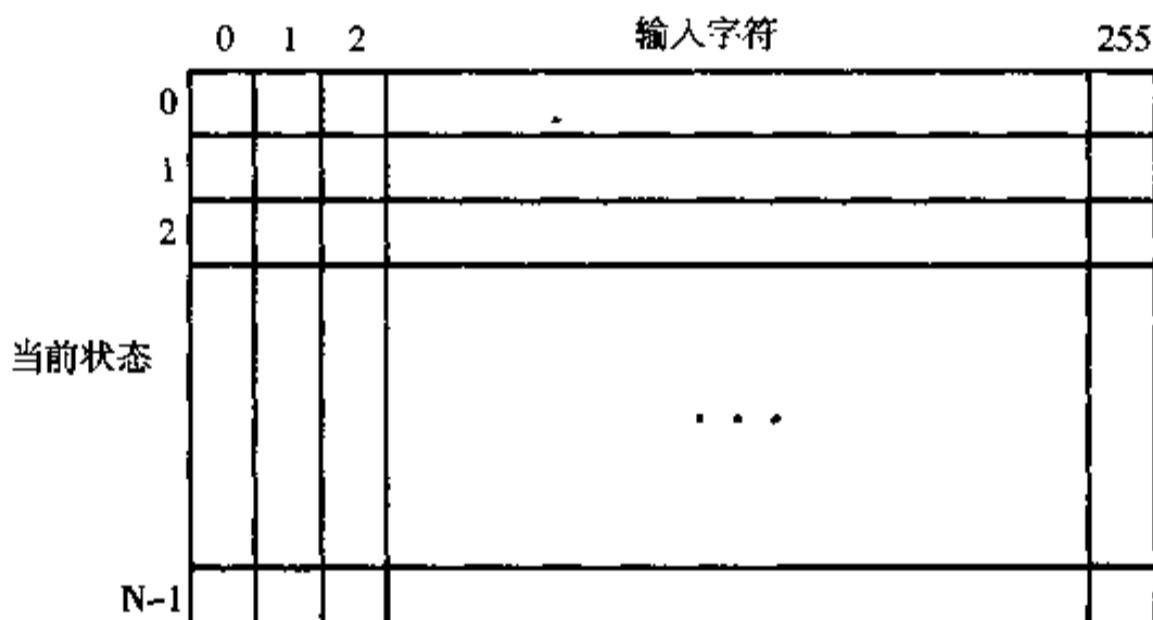


图 26.5 由转移矩阵对应的有限状态机。每行对应一个状态而每列对应一个可能的输入字符

在运行时，客户维护记录当前状态的变量。当有一个字符到达时，客户用当前状态变量及该字符的数字值在转移矩阵中进行索引。

26.17 压缩的有限状态机表示

用 C 代码来初始化一个大矩阵将是非常繁琐的。而且，如果转移矩阵的每个元素都要包含关于要采取的动作以及下一状态的全部信息，该矩阵将消耗很大的内存空间。为了使转移矩阵更小也为了使初始化的工作更为容易，我们的代码使用了有限状态机的压缩表示。

就本质来讲,所选的数据结构允许程序员构造一种压缩的数据结构,用它来表示有限状态机,并且让程序在运行时构造相关的转移矩阵。文件tnfsm.h中包含了在压缩表示中所用的结构fsm_trans的声明:

```
/* tnfsm.h */

/* Telnet Socket-Input FSM States: */
#define TSDATA 0 /* normal data processing */ 
#define TSIAC 1 /* have seen IAC */ 
#define TSWOPT 2 /* have seen IAC-{ WILL/WONT } */ 
#define TSDOPT 3 /* have seen IAC-{ DO/DONT } */ 
#define TSSUBNEG 4 /* have seen IAC-SB */ 
#define TSSUBIAC 5 /* have seen IAC-SB-...-IAC */ 

#define NTSTATES 6 /* # of TS* states */ 

/* Telnet Keyboard-Input FSM States: */
#define KSREMOTE 0 /* input goes to the socket */ 
#define KSLOCAL 1 /* input goes to a local func. */ 
#define KSCOLLECT 2 /* input is scripting-file name */ 

#define NKSTATES 3 /* # of KS* states */ 

/* Telnet Option Subnegotiation FSM States: */
#define SS_START 0 /* initial state */ 
#define SS_TERMTYPE 1 /* TERMINAL_TYPE option subnegotiation */ 
#define SS_END 2 /* state after all legal input */ 

#define NSSTATES 3 /* # of SS_* states */ 

#define FSINVALID 0xff /* an invalid state number */ 

#define NCHRS 256 /* number of valid characters */ 
#define TCANY (NCHRS+1) /* match any character */ 

struct fsm_trans {
    unsigned char ft_state; /* current state */ 
    short ft_char; /* input character */ 
    unsigned char ft_next; /* next state */ 
    int (*ft_action)(FILE *sfp, FILE *tfp, int.c); /* action to take */ 
};
```

压缩的有限状态机表示由1维的fsm_trans结构数组组成。每个元素指定了一种转移。ft_state字段指明转移开始时的有限状态机状态。ft_char字段指定了引起转移的字符(或者用TC_ANY来表示没有明确声明的其他所有字符)。字段ft_char指定了转移终端的状态,而ft_action字段给出了要调

用的过程的地址，用来完成与状态转移相关的动作。

26.18 在运行时维持压缩表示

本客户的例子并没有把压缩表示中的所有信息都复制到转移矩阵中，而是采用了这样的方法，它不改变压缩表示的内容并用它来维护转移信息。为此，软件在转移矩阵的每个元素中存放一个整数。该整数是压缩表示中某一项的索引，而该项与所要进行的转移相对应。图 26.6 展示了该数据结构：

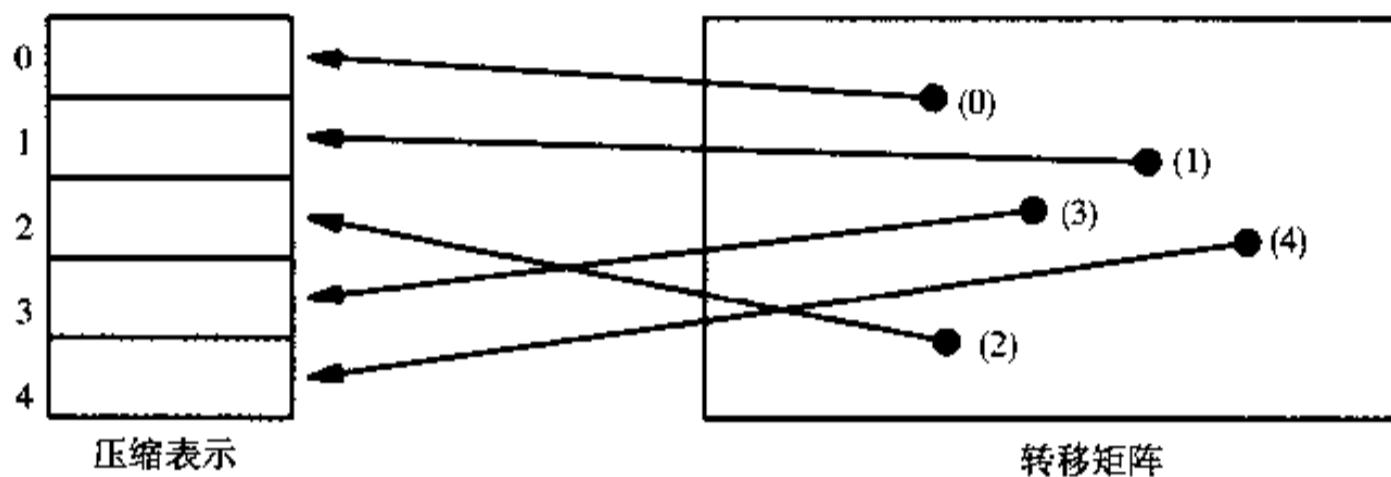


图 26.6 运行时有限状态机数据结构。转移矩阵中的每一项都包含一个指向压缩表示中某元素的索引

26.19 压缩表示的实现

文件 ttfsm.c 包含了压缩有限状态机表示的例子，它表示的是图 26.4 中的理论有限状态机：

```
/* ttfsm.c */

#include <sys/types.h>
#include <stdio.h>
#include "telnet.h"
#include "tnfsm.h"
#include "local.h"

extern int do_echo(FILE *, FILE *, int), do_noga(FILE *, FILE *, int),
    do_notsup(FILE *, FILE *, int), do_status(FILE *, FILE *, int),
    no_op(FILE *, FILE *, int), recopt(FILE *, FILE *, int),
    subend(FILE *, FILE *, int), subopt(FILE *, FILE *, int),
    tcdm(FILE *, FILE *, int), ttputc(FILE *, FILE *, int),
    will_notsup(FILE *, FILE *, int), will_termtype(FILE *, FILE *, int),
    will_txbinary(FILE *, FILE *, int), tnabort(FILE *, FILE *, int),
    do_txbinary(FILE *, FILE *, int);

struct fsm_trans ttstab[] = {
    /* State      Input      Next State      Action */
    {
```

```

/* ---- ----- ----- */
{ TSDATA,      TCIAC,      TSIAC,      no_op      },
{ TSDATA,      TCANY,      TSDATA,      ttputc     },
{ TSIAC,       TCIAC,      TSDATA,      ttputc     },
{ TSIAC,       TCSB,       TSSUBNEG,   no_op      },
/* Telnet Commands */
{ TSIAC,       TCNOP,      TSDATA,      no_op      },
{ TSIAC,       TCDM,       TSDATA,      tcdm      },
/* Option Negotiation */
{ TSIAC,       TCWILL,     TSWOPT,     recopt     },
{ TSIAC,       TCWONT,     TSWOPT,     recopt     },
{ TSIAC,       TCDO,       TSDOPT,     recopt     },
{ TSIAC,       TCDONT,    TSDOPT,     recopt     },
{ TSIAC,       TCANY,      TSDATA,      no_op      },
/* Option Subnegotiation */
{ TSSUBNEG,    TCIAC,     TSSUBIAC,   no_op      },
{ TSSUBNEG,    TCANY,      TSSUBNEG,   subopt     },
{ TSSUBIAC,   TCSE,       TSDATA,     subend     },
{ TSSUBIAC,   TCANY,      TSSUBNEG,   subopt     },
{ TSWOPT,      TOECHO,     TSDATA,     do_echo    },
{ TSWOPT,      TONOGA,     TSDATA,     do_noga    },
{ TSWOPT,      TOTXBINAR, TSDATA,     do_txbinary },
{ TSWOPT,      TCANY,      TSDATA,     do_notsup  },
{ TSDOPT,      TOTERMTYPE, TSDATA,     will_termtype },
{ TSDOPT,      TOTXBINAR, TSDATA,     will_txbinary },
{ TSDOPT,      TCANY,      TSDATA,     will_notsup },
{ FSINVALID,   TCANY,      FSINVALID,  tnabort    },
};

#define NTRANS (sizeof(ttstab)/sizeof(ttstab[0]))

int ttstate;
u_char ttfsm[NTSTATES][NCHRS];

```

ttstab 数组中包含了 22 个有效项，每一项都对应着图 26.4 所示的某个转移（加上标志数组结束的额外项）。数组中的每一项都由指明某个转移的 fsm_trans 结构组成。注意 ttstab 是压缩的并且很容易定义。它是压缩的，因为它并不包含任何空项；它也很容易定义，因为每一项都直接与有限状态机中的某个转移相对应。

26.20 构造有限状态机转移矩阵

只有了解了如何用压缩表示来生成转移矩阵，我们才会了解这种压缩表示的用处。文件 fsminit.c 中包含了该代码：

```
/* fsminit.c - fsminit */

#include <sys/types.h>

#include <stdio.h>

#include "tnfsm.h"

#define TINVALID 0xff      /* an invalid transition index */

/*
 * fsminit - Finite State Machine initializer
 */
int
fsminit(unsigned char fsm[][NCHRS], struct fsm_trans ttab[], int nstates)
{
    struct fsm_trans *pt;
    int sn, ti, cn;

    for (cn=0; cn<NCHRS; ++cn)
        for (ti=0; ti<nstates; ++ti)
            fsm[ti][cn] = TINVALID;

    for (ti=0; ttab[ti].ft_state != FSINVALID; ++ti) {
        pt = &ttab[ti];
        sn = pt->ft_state;
        if (pt->ft_char == TCANY) {
            for (cn=0; cn<NCHRS; ++cn)
                if (fsm[sn][cn] == TINVALID)
                    fsm[sn][cn] = ti;
        } else
            fsm[sn][pt->ft_char] = ti;
    }
    /* set all uninitialized indices to an invalid transition */
    for (cn=0; cn<NCHRS; ++cn)
        for (sn=0; sn<nstates; ++sn)
            if (fsm[sn][cn] == TINVALID)
                fsm[ti][cn] = ti;
}
```

过程 fsminit 要求三个参数。参数 fsm 指明必须被初始化的转移矩阵；参数 ttab 给出压缩有限状态机表示的地址，而参数 nstates 则指明最终有限状态机的状态数。

fsminit 首先把转移矩阵的全部项初始化成 TINVALID。然后循环扫描压缩表示的每个元素，并把由该元素所指定的状态转移加到转移矩阵中去。最后，它再次扫描整个转移矩阵，修改还没有填充的转移，使它们指向压缩表示末尾的非法状态转移项。

fsminit 中的大多数代码都非常简单。但是，当增加状态转移时，fsminit 必须对某种明确的转移和某种缩写做出区分。为了理解这段代码，我们记得压缩表示用字符 TCANY 表示没有明确声明的所有其他字符。因此，当 fsminit 检查各个转移时，它将检查引起该转移的字符。如果该项指定了字符 TCANY，则 fsminit 循环处理所有可能的字符，把该转移加到还没有初始化的字符上去。如果该项指明了除 TCANY 以外的任何字符，则 fsminit 用该字符填在转移矩阵中。

26.21 套接字输出有限状态机

图 26.4 给出的有限状态机定义了客户对每个来自服务器的字符所做的动作。一个单独的、更简单的有限状态机描述了客户如何处理来自键盘的字符。我们把与键盘输入相关的 FSM 称为套接字输出 FSM (socket output FSM)。这个名字看起来有点不同寻常。图 26.7 展示了客户软件的组织是如何支持这种名字的。

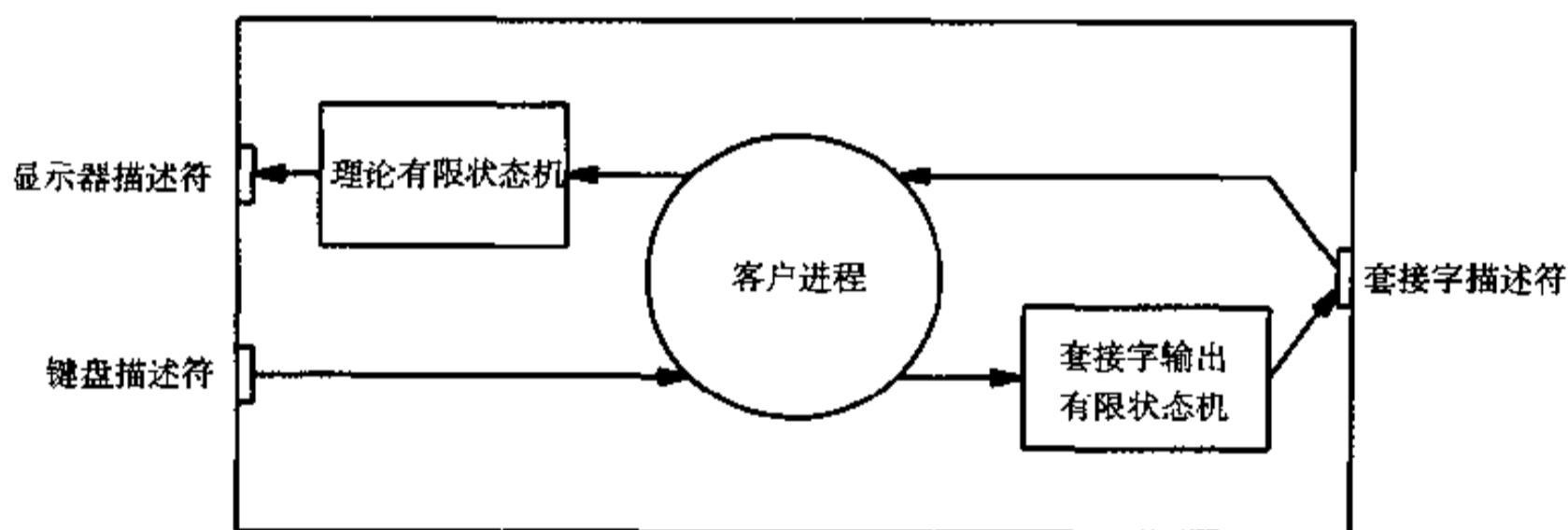


图 26.7 客户软件组织。中心过程从套接字或键盘中读取数据并调用某个 FSM 过程并对其进行处理。处理键盘数据的 FSM 是与套接字输出相关的

图 26.8 显示了套接字输出有限状态机。替代为每个操作定义状态，该机器仅有三种状态。多数操作是在单个转移上处理，该转移将客户从表示本地处理的状态带到表示（普通）远程系统交互的状态。

套接字输出有限状态机从状态 KSREMOTE 开始，它使客户把从键盘来的每个字符都发送到远程服务器上去。当用户键入键盘退出（keyboard escape）键时，客户进入状态 KSLOCAL，并等待一个击键动作。紧跟 escape 的大多数击键都没有意义，但也有几个会引起客户动作，接着便返回到状态 KSREMOTE。只有一个 KCSCRIPT，会使客户进入状态 KSCOLLECT，它为脚本搜索某个文件名。

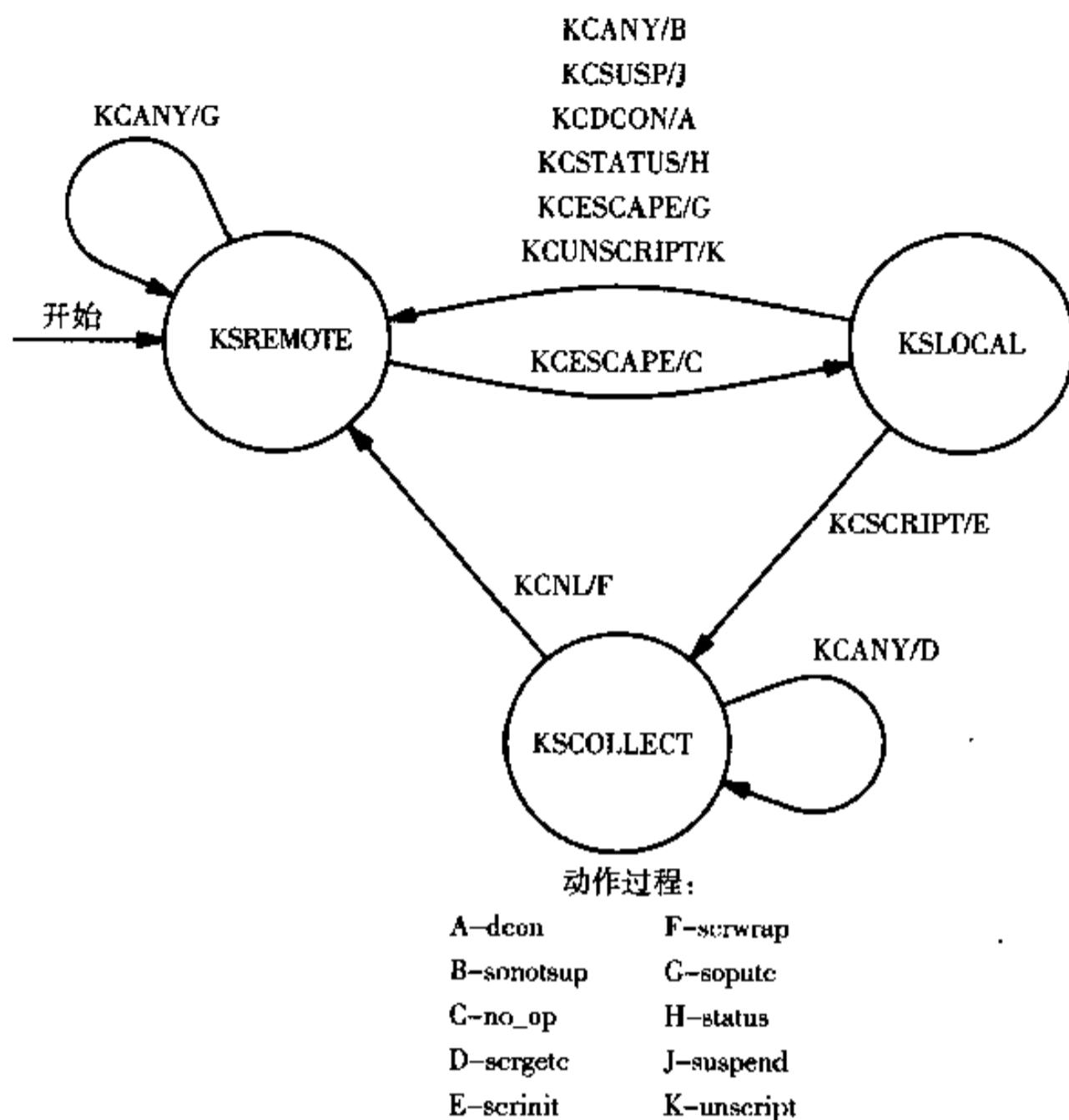


图 26.8 套接字输出有限状态机，用于定义对每个用户输入字符所做的动作。大多数字符被客户发送到远程服务器上去。但是，该设计允许用户从数据连接中退出（escape），然后与本地客户程序通信

26.22 套接字输出有限状态机的相关定义

文件 sofsm.c 定义了套接字输出有限状态机的压缩表示：

```
/* sofsm.c */

#include <sys/types.h>

#include <stdio.h>
#include "telnet.h"
#include "tnfsm.h"

/* Special chars: */
char      t_flushc, t_intrc, t_quitc, sg_erase, sg_kill;

extern int      soputc(FILE *, FILE *, int), scrinit(FILE *, FILE *, int),
```

```

scrgetc(FILE *, FILE *, int), scrwrap(FILE *, FILE *, int),
unscript(FILE *, FILE *, int), dcon(FILE *, FILE *, int),
suspend(FILE *, FILE *, int), status(FILE *, FILE *, int),
sonotsup(FILE *, FILE *, int), no_op(FILE *, FILE *, int),
tnabort(FILE *, FILE *, int);

struct fsm_trans sostab[] = {
    /* State           Input      Next State     Action   */
    /* -----       -----      -----       -----   */
/* Data Input */
    { KSREMOTE,      KCESCAPE,   KSLOCAL,      no_op    },
    { KSREMOTE,      KCANY,      KSREMOTE,     soputc   },
/* Local Escape Commands */
    { KSLOCAL,       KCSCRIPT,   KSCOLLECT,   scrinit  },
    { KSLOCAL,       KCUNSCRIPT, KSREMOTE,   unscript  },
    { KSLOCAL,       KCESCAPE,   KSREMOTE,   soputc   },
    { KSLOCAL,       KCDCON,    KSREMOTE,   dcon     },
    { KSLOCAL,       KCSUSP,    KSREMOTE,   suspend  },
    { KSLOCAL,       KCSTATUS,   KSREMOTE,   status   },
    { KSLOCAL,       KCANY,     KSREMOTE,   sonotsup },
/* Script Filename Gathering */
    { KSCOLLECT,    KCNL,       KSREMOTE,   scrwrap  },
    { KSCOLLECT,    KCANY,     KSCOLLECT,  scrgetc  },
    { FSINVALID,    KCANY,     FSINVALID, tnabort  },
};

#define NTRANS (sizeof(sostab)/sizeof(sostab[0]))

int      sostate;
u_char   sofsm[NKSTATES][NCHRS];

```

数组 sostab 中是有限状态机的压缩表示，变量 sostate 中含有一个整数，它给出了套接字输出有限状态机的当前状态。

26.23 选项子协商有限状态机

图 26.9 显示了客户使用的第三种有限状态机。它处理在选项子协商 (option subnegotiation) 过程中到达的字符序列。因为它只识别一种可能的选项子协商 (终端类型)，所以该有限状态机只需要三个状态。

理解子协商的最简单的方法就是想像它描述了理论有限状态机中状态 TSSUBNEG 的内部结构。当主有限状态机运行在状态 TSSUBNEG 时，它调用过程 subopt 处理每个传入字符。subopt 读行子协商有限状态机。如图 26.9 所示，子协商有限状态机依赖于选项立即做出决定。如果它发现终端类型的子协商，则机器移到状态 SS_TERMTYPE。否则，它直接变到状态 SS_END，并忽略剩下的子协商字符串。

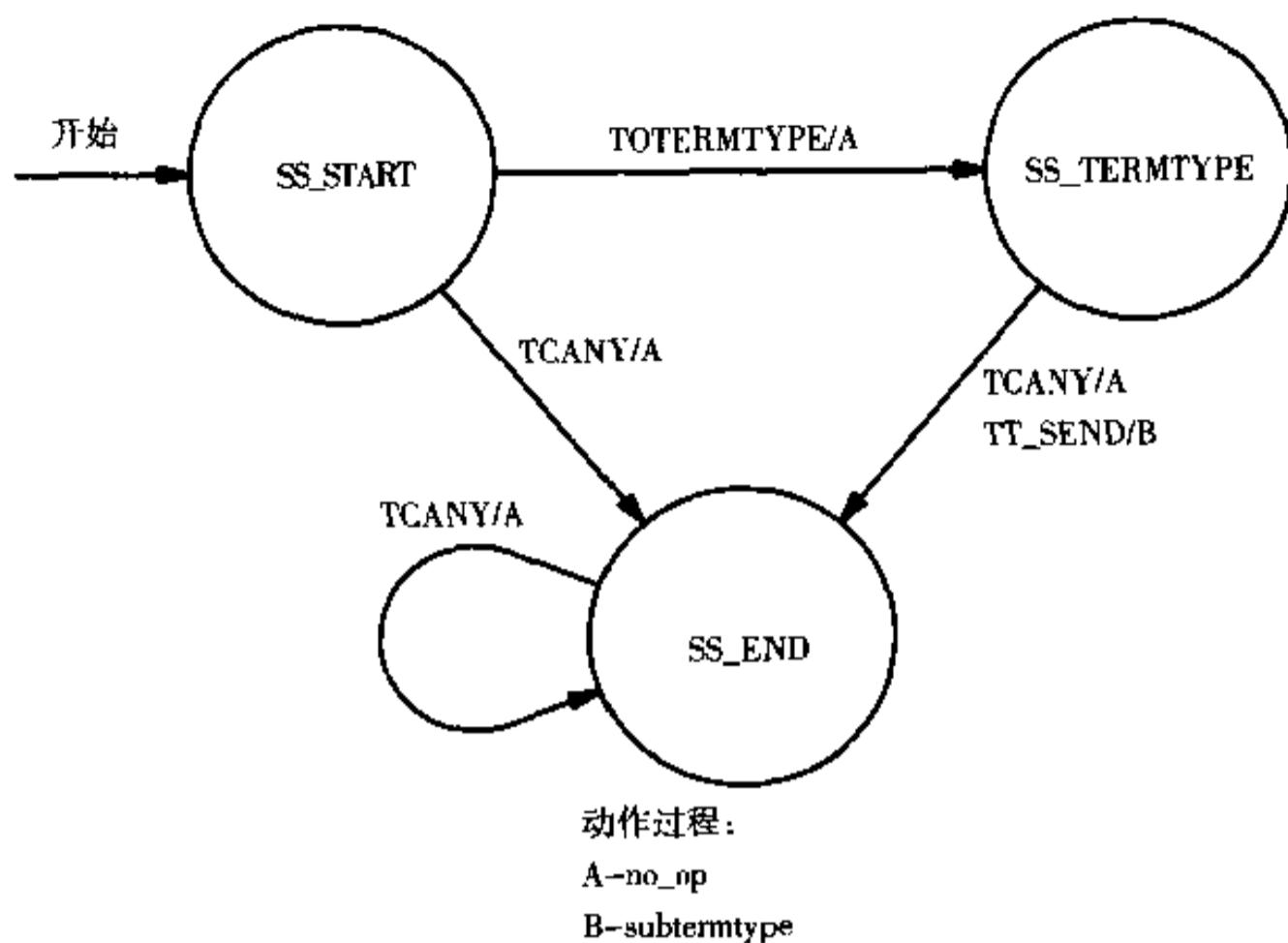


图 26.9 选项子协商所用的简单有限状态机。每当客户完成一次选项子协商后，都要把该状态机重新初始化

一旦有限状态机到达状态 SS_TERMTYPE，它就检测子协商动词。如果动词是 TT_SEND，它便调用 subtermtype 并忽略其他子协商。在我们看到客户如何响应终端类型选项之后，子协商有限状态机的意图和运行方式就很清楚了。

26.24 选项子协商有限状态机的相关定义

文件 subfsm.c 中是子协商有限状态机的 C 语言的声明：

```
/* subfsm.c */

#include <sys/types.h>

#include <stdio.h>

#include "telnet.h"
#include "tnfsm.h"

extern int          no_op(FILE *, FILE *, int),
                    subtermtype(FILE *, FILE *, int),
                    tnabort(FILE *, FILE *, int);

struct fsm_trans substab[] = {
    /* State           Input           Next State        Action      */
    /* ----           -----           -----           -----      */
    {
```

```

    { SS_START,           TOTERMTYPE,      SS_TERMTYPE,      no_op        },
    { SS_START,           TCANY,          SS_END,          no_op        },
    { SS_TERMTYPE,        TT_SEND,         SS_END,          subtermtype },
    { SS_TERMTYPE,        TCANY,          SS_END,          no_op        },
    { SS_END,             TCANY,          SS_END,          no_op        },
    { FSINVALID,          TCANY,          FSINVALID,       tnabort     },
};

int      substate;
u_char   subfsm[NSSTATES][NCHRS];

```

26.25 有限状态机初始化

启动时，客户调用过程fsmbuild来初始化所有有限状态机。如文件ttinit.c中代码所示，fsmbuild调用 fsminit 来为每台状态机构造所需数据结构，并给每台状态机的状态变量赋予初始状态。

```

/* ttinit.c - ttinit */

#include <sys/types.h>

#include <stdio.h>

#include "tnfsm.h"

extern struct fsm_trans      ttstab[], sostab[], substab[];
extern unsigned char          ttfsm[][NCHRS], sofsm[][NCHRS], subfsm[][NCHRS];
extern int                     ttstate, sostate, substate;

int
fsminit(unsigned char fsm[][NCHRS], struct fsm_trans ttab[], int nstates);

/*
 * fsmbuild - build the Finite State Machine data structures
 */
int
fsmbuild()
{
    fsminit(ttfsm, ttstab, NTSTATES);
    ttstate = TSDATA;

    fsminit(sofsm, sostab, NKSTATES);
    sostate = KSREMOTE;

    fsminit(subfsm, substab, NSSTATES);
}

```

```

    substate = SS_START;
}

```

26.26 TELNET 客户的参数

文件 tclient.c 中是主程序的代码，当用户调用 TELNET 客户时，执行该程序：

```

/* tclient.c - main */

#include <stdlib.h>

char      *host = "localhost";           /* host to use if none supplied */

int      erexit(const char *format, ...);
int      telnet(const char *host, const char *service);

/*
 * main - TCP client for TELNET service
 */
int
main(int argc, char *argv[])
{
    char      *service = "telnet";          /* default service name */

    switch (argc) {
    case 1:   break;
    case 3:
        service = argv[2];
        /* FALL THROUGH */
    case 2:
        host = argv[1];
        break;
    default:
        erexit("usage: telnet [host [port]]\n");
    }
    telnet(host, service);
    exit(0);
}

```

用户可给出 0 个、1 个或 2 个命令行参数让程序分析。如果没有参数 (argc=1)，客户将与本地主机上的某个服务器联系并使用 telnet 服务。如果有一个参数 (argc=2)，则客户把该参数作为运行服务器的远程主机的名字。最后，如果有两个参数，则客户把第二个参数看成远程机上的某个服务的名字，而把第一个参数作为远程主机的名字。在分解完参数后，主程序调用函数 telnet。

26.27 TELNET 客户的核心

在被激活时，客户完成一些必要的初始化工作。一旦完成初始化工作后，客户进入无限循环，用 select 来等待 I/O。文件 telnet.c 包含客户代码。

```
/* telnet.c - telnet */

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys	signal.h>
#include <sys/errno.h>

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#include "local.h"

void      rcvurg(int);

#define    BUFSIZE          2048      /* read buffer size */

struct termios      oldtty;

int      connectTCP(const char *host, const char *service);
int      ttyssetup(void);

/*
 * telnet - do the TELNET protocol to the given host and port
 */
int
telnet(const char *host, const char *service)
{
    unsigned char    buf[BUFSIZE];
    int      s, nfds; /* socket and # file descriptors */
    int      cc;
    int      on = 1;
    fd_set    arfds, awfds, rfds, wfds;
    FILE     *sfp;

    s = connectTCP(host, service);

    ttyssetup();

    fsmbuild();           /* set up FSM's */
}
```

```

(void) signal(SIGURG, rcvurg);
(void) setsockopt(s, SOL_SOCKET, SO_OOBINLINE, (char *) &on,
                  sizeof(on));

nfds = getdtablesize();
FD_ZERO(&rfds);
FD_ZERO(&wfds);
FD_SET(s, &rfds);           /* the socket */
FD_SET(0, &rfds);          /* standard input */
sfp = fdopen(s, "w");       /* to get buffered output */

while (1) {
    memcpy(&rfds, &arfds, sizeof(rfds));
    memcpy(&wfds, &awfds, sizeof(rfds));

    if (select(nfds, &rfds, &wfds, (fd_set *) 0,
               (struct timeval *) 0) < 0) {
        if (errno == EINTR)
            continue; /* just a signal */
        cerrexit("select: %s\n", strerror(errno));
    }

    if (FD_ISSET(s, &rfds)) {
        cc = read(s, (char *) buf, sizeof(buf));
        if (cc < 0)
            cerrexit("socket read: %s\n",
                      strerror(errno));
        else if (cc == 0) {
            printf("\nconnection closed.\n");
            if (tcsetattr(0, TCSADRAIN, &oldtty) < 0)
                errexit("tcsetattr: %s\n",
                        strerror(errno));
            exit(0);
        } else
            ttwrite(sfp, stdout, buf, cc);
    }

    if (FD_ISSET(0, &rfds)) {
        cc = read(0, (char *) buf, sizeof(buf));
        if (cc < 0)
            cerrexit("tty read: %s\n",
                      strerror(errno));
        else if (cc == 0) {
            FD_CLR(0, &arfds);
            (void) shutdown(s, 1);
        } else
            sowrite(sfp, stdout, buf, cc);
    }
}

```

```
    }
    (void) fflush(sfp);
    (void) fflush(stdout);
}
}
```

过程 telnet 要求有两个参数，分别指明远程机器及该机器上某个服务的名字。程序从调用 connectTCP 开始，分配一个套接字，并形成到该服务器的 TCP 连接。接着，调用 ttysetup 对本地终端参数初始化，调用 fsmbuild 对三个有限状态机初始化。然后，调用系统函数 signal 为紧急数据建立句柄。如果在调用 signal 后，TCP 收到了从服务器来的紧急数据，系统将启动过程 rcvurg 来对其进行处理。

为了使代码可移植，我们的实现并没有定义文件描述符的最大值。相反，telnet 在对文件描述符进行初始化之前调用 getdtablesize 来确定系统可得到的文件描述符的最大值。然后它把结果记录在变量 nfd 中。在主循环的每次循环中，telnet 把该值作为第一个参数传给 select。

因为它们都非常短，所以没有把 tcdm 和 rcvurg 放在单独的文件中，而是在文件 sync.c 中。

```
/* sync.c - tcdm, rcvurg */

#include <stdio.h>

char      synching; /* non-zero, if we are doing telnet SYNCH      */

/*
 * tcdm - handle the telnet "DATA MARK" command (marks end of SYNCH),
 */
int
tcdm(FILE *sfp, FILE *tfp, int c)
{
    if (synching > 0)
        synching--;
    return 0;
}

/*
 * rcvurg - receive urgent data input (indicates a telnet SYNCH)
 */
int
rcvurg(int sig)
{
    synching++;
}
```

TELNET协议指明，当紧急数据到达时，客户必须与服务器同步（ synchronize）。为了同步，客户从服务器到达的数据流中向前搜索，直到它遇到DATAMARK字符。因此，当紧急数据信号发生时，revurg仅仅使全局变量 synching 加 1，从而把客户置于同步模式。当 synching 设置好后，有限状态机动作过程（action procedure）将丢弃传入数据，并不把它显示在用户终端。稍后，当DATAMARK字符到达时，客户调用过程 tcdm，把全局变量 synching 重新设成零，并使客户返回到正常的处理中。

一旦客户完成了初始化，它便进入无限循环。每次重复循环时，它用 select 来等待从套接字或键盘来的 I/O（也就是说，与描述符 0 相关的标准输入）。如果在进程阻塞时，程序接收到信号，select 调用便返回值 EINTR。如果这样，则客户继续下一次循环。如果 select 调用返回其他任何差错代码，则客户给用户打印一个消息并退出。

如果 select 正常返回，则在键盘、套接字或两者兼而有之会有数据出现。telnet 首先检查套接字，看一下数据是否从服务器到达。如果有数据到达，则调用 read 读取该数据并调用 ttwrite 把数据写到用户的显示器上。我们将在下面看到，ttwrite 实现了解释传入数据流的理论有限状态机，它还对退出（escape）及嵌入的命令序列进行处理。

在 telnet 检查了套接字的数据后，它又检查键盘的描述符。如果数据从键盘到达，则 telnet 调用 read 读取数据并调用 sowrite 把它写到套接字中。sowrite 中的代码负责执行处理本地 escape 的有限状态机。作为特例，客户把文件结束（end-of-file）解释成中断连接的请求。如果它接收到 end-of-file 字符（也就是说 read 返回 0），则 telnet 调用 shutdown 向服务器发送 end-of-file 条件。在任何情况下，telnet 在每次循环中都调用 fflush，以保证输出例程不会只把已写入的数据放入缓存（没有真正地写）。fflush 强迫一个到 write 系统过程的调用。

26.28 主有限状态机的实现

过程 ttwrite 实现了图 26.4 中的有限状态机，该有限状态机在数据从服务器到达后对其进行解释。程序在文件 ttwrite.c 中：

```
/* ttwrite.c - ttwrite */

#include <sys/types.h>

#include <stdio.h>

#include "tnfsm.h"

extern struct fsm_trans          ttstab[];
extern unsigned char             ttfsm[][NCHRS];
extern int                        ttstate;

/*
 * ttwrite - do output processing for (local) network virtual printer
 */
int
```

```
ttwrite(FILE *sfp, FILE *tfp, unsigned char *buf, int cc)
{
    struct fsm_trans    *pt;
    int                  i, ti;

    for (i=0; i<cc; ++i) {
        int      c = buf[i];

        ti = ttfsm[ttstate][c];
        pt = &ttstab[ti];
        (pt->ft_action) (sfp, tfp, c);
        ttstate = pt->ft_next;
    }
}
```

ttwrite 从缓存 buf 中每次取出 cc 个字符。ttwrite 每次取出字符后，便用该字符及当前状态（变量 ttstate）作为索引在转移矩阵中进行查找。转移矩阵返回 ti——压缩表示（ttstab）中某个转移的索引。ttwrite 调用与该转移相关联的过程（字段 ft_action），并把当前状态变量设置成相应的下一状态（字段 ft_next）。

26.29 小结

TELNET 是 TCP/IP 协议族中最为流行的应用协议。该协议为客户及服务器之间提供了交互式的字符传输。通常，客户通过 TCP 连接把用户终端与服务器连起来。我们所示例的客户软件由一个进程组成，它用操作系统函数 select 来进行并发传送。

TELNET 用 escape 序列在数据流中嵌入命令及控制信息。为了简化代码，我们的客户实现例子用了三个有限状态机来解释字符序列。一个处理来自服务器的数据，另一个处理来自用户键盘的数据，第三个处理选项子协商。

本章的示例代码显示了客户进程的核心实现以及用来实现有限状态机的数据结构。下一章探讨了一些过程的细节，这些过程完成与有限状态机状态转移相关的动作。

深入研究

Postel [RFC 854] 包含了基本的 TELNET 协议标准，包括网络虚拟终端编码。Postel 和 Reynolds [RFC 855] 说明了选项协商的细节以及选项子协商。各个选项的细节可以在其他 RFC 中找到。VanBokkelen [RFC 1091] 中说明了终端类型选项。Postel 和 Reynolds [RFC 857] 描述了 echo 选项，而 Postel 和 Reynolds [RFC 856] 描述了二进制传输选项。

Stevens [1998] 描述了 rlogin 协议，这是在许多 UNIX 系统（包括 Linux）上可得到的另一种 TELNET，其中还展示了一个简单的客户和服务器的例子。Leffler 等人 [1996] 讨论了更多的有关终端 I/O 的细节。

习题

- 26.1 将在单进程中使用多线程执行的 TELNET 客户实现与本章给出的设计进行比较。它们各自的优点和缺点是什么？
- 26.2 阅读联机文档，找出有关终端设备和设备驱动器的内容。什么是 cooked 模式、 cbreak 模式和 raw 模式？
- 26.3 编写一个程序，该程序禁止在用户终端上回显字符。程序退出时回显参数会怎样？
- 26.4 阅读 Linux 系统文档，找出 stty 命令。如果把输出从 stty 重定向到文件，将发生什么？为什么？
- 26.5 在 suspend.c 中的代码如何挂起客户进程？
- 26.6 重写理论有限状态机，把选项子协商包括进去。
- 26.7 有限状态机例子的实现中用压缩表示来节省空间。试估计一下用传统方法来表示有限状态机时所需的空间，并与压缩表示的空间估计做比较。
- 26.8 在什么条件下 read 从终端返回 0 值？
- 26.9 阅读 TELNET 协议规约，找到同步的明确规则。发送者何时发送紧急数据？为什么需要同步？
- 26.10 在客户发出 end-of-file 后，服务器还可以再发送数据。客户如何知道何时终止与服务器的连接呢？
- 26.11 不用有限状态机重写 twrite。两种实现的优点和缺点是什么？

第 27 章 TELNET 客户（实现细节）

27.1 引言

前一章讨论了TELNET客户的结构，并说明了它如何使用有限状态机来控制处理。本章将说明语义动作过程如何实现字符处理的细节，以此完成关于TELNET的讨论。

27.2 有限状态机动作过程

有限状态机实现了TELNET协议的大多数细节。这些状态机控制处理，协调响应与请求，并将传入的命令序列映射成动作。客户每进行一次有限状态机中的转移，它就调用某个过程，完成与该转移相关的动作。图 26.4^①展现了一些与有限状态机中的转移相关联的过程名，该有限状态机处理来自服务器的字符。

一个动作可能简单（如丢弃入字符），也可能复杂（如通过发送识别本地终端的字符串来做出响应）。将每个动作封装在过程中有助于使机器规格说明保持统一，并简化顶层的代码。但是，将软件划分成多个过程，每个过程处理一个动作，这意味着要理解这些过程间的关系只能通过参考互连它们的有限状态机。

下面几节将各自描述与有限状态机转移相关联的动作过程。

27.3 记录选项请求的类型

在状态TSIAC（即跟随TCIAC字符的到达）下，TCWILL或TCWONT的到达将使有限状态机转移到状态TSWOPT。该有限状态机规定，此时应调用recopt过程。recopt将记录引起转移的字符，以被后用。类似地，在到状态TSDOPT的转移中，有限状态机使用recopt记录字符TCDO或TCDONT。文件recopt.c含有代码：

```
/* recopt.c - recopt, no_op */

#include <sys/types.h>

#include <stdio.h>

unsigned char    option_cmd;      /* has value WILL, WONT, DO, or DONT */
```

^① 本图出现在第 299 页。

```

/*
 * recpt - record option type
 */
int
recpt(FILE *sfp, FILE *tfp, int c)
{
    option_cmd = c;
    return 0;
}

/*
 * no_op - do nothing
 */
int
no_op(FILE *sfp, FILE *tfp, int c)
{
    return 0;
}

```

27.4 完成空操作

文件 recpt.c 含有过程 no_op 的代码。由于有限状态机必须使状态和输入字符的所有可能组合都有一个动作，这样，过程 no_op 可用于不需要任何动作的那些转移。例如，从状态 TSDATA 到 TSIAC 的转移不需要任何动作。因此，有限状态机定义了空操作 no_op 的调用。

27.5 对回显选项的 WILL/WONT 做出响应

跟在回显选项后，服务器会发送 WILL 或 WONT，用以通知客户它愿意回显字符，或它希望停止字符的回应。该有限状态机规定，当这个报文到达时，应调用过程 do_echo。

```

/* do_echo.c - do_echo */

#include <sys/types.h>

#include <termios.h>
#include <stdio.h>

#include "telnet.h"

char          doecho;           /* nonzero, if remote ECHO      */
extern u_char    option_cmd;

```

```
/*
 * do_echo - handle TELNET WILL/WON'T ECHO option
 */
int
do_echo(FILE *rfp, FILE *tfp, int c)
{
    struct termios      tio;
    static char        savec[2];
    int                 ok, tfd = fileno(tfp);

    if (doecho) {
        if (option_cmd == TCWILL)
            return 0; /* already doing ECHO */ */
    } else if (option_cmd == TCWONT)
        return 0; /* already NOT doing ECHO */ */

    if (ok = tcgetattr(tfd, &tio) == 0) {
        if (option_cmd == TCWILL) {
            tio.c_lflag &= ~ (ECHO | ICANON);
            /* VMIN & VTIME are overloaded with other chars,
             * so save and restore them later.
             */
            savec[0] = tio.c_cc[VMIN];
            savec[1] = tio.c_cc[VTIME];
            tio.c_cc[VMIN] = 1;
            tio.c_cc[VTIME] = 0;
        } else {
            tio.c_lflag |= (ECHO | ICANON);
            tio.c_cc[VMIN] = savec[0];
            tio.c_cc[VTIME] = savec[1];
        }
        ok &= tcsetattr(tfd, TCSADRAIN, &tio) == 0;
    }
    if (ok)
        doecho = !doecho;
    (void) putc(TCIAC, rfp);
    if (doecho)
        (void) putc(TCDO, rfp);
    else
        (void) putc(TCDONT, rfp);
    (void) putc((char)c, rfp);
    return 0;
}
```

TELNET 协议规定，服务器可发送 WILL 或 WONT，宣告它对于执行某个选项的意愿，或者可在给客户的某个请求的响应中发送这样一个报文。因此，如果客户已发送了含有 DO 或 DONT 的请求，来自服务器的报文就构成一个响应；否则，它就构成一个通告（advertisement）。

客户使用当前条件决定如何响应。如果当前客户期望服务器完成字符回显，全局变量 doecho 应含有一个非零值。如果服务器发送了 WILL 且客户已允许远程回显，客户不用答复。类似地，如果服务器发送了 WONT 且客户也不允许远程回显，客户也不用答复。但是，如果服务器发送了 WILL，但客户当前已禁止远程回显，客户就使用 tcsetattr 禁止本地字符回显。如果客户在 WONT 到达时禁止了本地回显，客户就假定服务器已禁止了远程回显。因此，客户便调用 tcsetattr 允许本地回显。如果客户改变了回显模式，它只发送 DO 或 DONT 响应。

27.6 对未被支持的选项的 WILL/WONT 做出响应

当客户收到它所不理解的选项的 WILL 或 WONT 请求，便调用过程 do_notsup 应答 DONT。

```
/* do_notsup.c - do_notsup */

#include <sys/types.h>

#include <stdio.h>

#include "telnet.h"

extern u_char      option_cmd;

/* -----
 * do_notsup - handle an unsupported telnet "will/won't" option
 * -----
 */
int
do_notsup(FILE *sfp, FILE *tfp, int c)
{
    (void) putc(TCIAC, sfp);
    (void) putc(TCDONT, sfp);
    (void) putc((char)c, sfp);
    return 0;
}
```

27.7 对 no go-ahead 选项的 WILL/WONT 做出响应

当服务器发送对 no go_ahead 选项的 WILL 或 WONT 请求时，客户用过程 do_noga 做出响应。

```
/* do_noga.c - do_noga */
```

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

#include <stdio.h>

#include "telnet.h"

extern u_char      option_cmd;

/* -----
 * do_noga - don't do telnet Go-Ahead's
 * -----
 */
int
do_noga(FILE *sfp, FILE *tfp, int c)
{
    static noga;

    if (noga) {
        if (option_cmd == TCWILL)
            return 0;
    } else if (option_cmd == TCWONT)
        return 0;
    noga = !noga;
    (void) putc(TCIAC, sfp);
    if (noga)
        (void) putc(TCDO, sfp);
    else
        (void) putc(TCDONT, sfp);
    (void) putc((char)c, sfp);
    return 0;
}
```

正如其他选项一样，如果客户对选项的当前设置与服务器的请求一致，客户就不响应。如果服务器请求客户改变设置，客户就通过对全局整数noga求反颠倒当前设置，并且发送一个DO或DONT响应。

27.8 生成用于二进制传输的 DO/DONT

服务器发送给客户的字符可用网络虚拟终端编码，或不编码而直接使用8比特二进制值。全局变量revbinary控制客户是期望收到二进制字符数据，还是收到NVT编码数据。当服务器发送用于二进制选项的 WILL 或 WONT 时，客户就调用过程 do_txbinary 作出响应。类似与其他处理选项的过程，do_txbinary 调用被这样设计，在请求服务器发送二进制时可以调用该过程，或者用它来响应来自服务器的一个通告。它使用全局变量option_cmd 来决定如何处理，并且假定该变量含有一个传入

请求。过程 do_txbinary 中的测试是为了查看客户是否期望服务器以二进制发送，它根据传入请求设置 rcvbinary，并且，若 rcvbinary 有变化便对服务器做出响应。

```
/* do_txbinary.c - do_txbinary */

#include <sys/types.h>

#include <stdio.h>

#include "telnet.h"

char          rcvbinary;           /* non-zero if remote TRANSMIT-BINARY */
extern u_char   option_cmd;

/*
 * do_txbinary - handle telnet "will/won't" TRANSMIT-BINARY option
 */
int
do_txbinary(FILE *sfp, FILE *tfp, int c)
{
    if (rcvbinary) {
        if (option_cmd == TCWILL)
            return 0;
    } else if (option_cmd == TCWONT)
        return 0;
    rcvbinary = !rcvbinary;
    (void) putc(TCIAC, sfp);
    if (rcvbinary)
        (void) putc(TCD0, sfp);
    else
        (void) putc(TCDONT, sfp);
    (void) putc((char)c, sfp);
    return 0;
}
```

27.9 对未被支持的选项的 DO/DONT 做出响应

服务器发送 DO 或 DONT 报文，告诉客户它应该允许或禁止某个指定的选项。如果客户同意承诺 (honor) 该选项，它就通过发送 WILL 作出响应；如果它不同意承诺该选项，便发送 WONT。如图 26.4 所示的有限状态机，当客户不支持某个特定选项时，它就调用过程 will_notsup。过程 will_notsup 发送 WONT 告诉服务器它不支持该选项。

```
/* will_notsup.c - will_notsup */
```

```
#include <sys/types.h>
#include <stdio.h>
#include "telnet.h"

/*
 * will_notsup - handle an unsupported telnet "do/don't" option
 */
int
will_notsup(FILE *sfp, FILE *tfp, int c)
{
    (void) putc(TCIAC, sfp);
    (void) putc(TCWONT, sfp);
    (void) putc((char)c, sfp);
    return 0;
}
```

27.10 对传输二进制选项的 DO/DONT 做出响应

当客户启动时，所有发给服务器的数据都使用了网络虚拟终端编码。虽然 NVT 编码包括了大多数可打印字符，但它没有为所有控制字符提供编码。如果服务器运行在支持面向屏幕应用的系统上，它往往需要传输任意字符数据的能力。因此，这种服务器往往要告知其希望传输任意数据的意愿，并且请求客户也传输任意数据。

服务器发送用于传输二进制选项的 DO，以便请求客户开始使用 8 比特无编码传输。客户收到请求后就调用过程 will_txbinary。

```
/* will_txbinary.c - will_txbinary */

#include <sys/types.h>
#include <stdio.h>
#include "telnet.h"

char                sndbinary;           /* non-zero if TRANSMIT-BINARY */
extern u_char        option_cmd;

/*
 * will_txbinary - handle telnet "do/don't" TRANSMIT-BINARY option
 */

```

```

int
will_txbinary(FILE *sfp, FILE *tfp, int c)
{
    if (sndbinary) {
        if (option_cmd == TCDO)
            return 0;
    } else if (option_cmd == TCDONT)
        return 0;
    sndbinary = !sndbinary;
    (void) putc(TCIAC, sfp);
    if (sndbinary)
        (void) putc(TCWILL, sfp);
    else
        (void) putc(TCWONT, sfp);
    (void) putc((char)c, sfp);
    return 0;
}

```

客户使用全局变量 `sndbinary` 控制其传输模式。如果请求会引起状态的改变，客户就通过发送 `WILL` 或 `WONT` 确认请求。

27.11 对终端类型选项的 DO/DONT 做出响应

从客户到服务器沟通终端类型需要两个步骤。首先，服务器询问客户它是否承诺 `termtype` 选项。其次，如果客户同意承诺终端类型选项，服务器就使用选项子协商，请求一个识别用户终端类型的字符串。

在 Linux 中，客户通过查看进程环境中的 `TERM` 变量，找到用户的终端类型。为此，它要调用库函数 `getenv`，`getenv` 找到某个指明的变量，并返回其字符串值的指针。

当用于终端类型选项的请求到达时，客户就调用过程 `will_termtype`。文件 `will_termtype.c` 含有代码：

```

/* will_termtype.c - will_termtype */

#include <sys/types.h>

#include <stdlib.h>
#include <stdio.h>

#include "telnet.h"

char          termtype;      /* non-zero if received "DO TERMTYPE" */
char          *term;         /* terminal name */
extern u_char  option_cmd;

int   do_txbinary(FILE *, FILE *, int), will_txbinary(FILE *, FILE *, int);

```

```

/*
 * will_termtype - handle telnet "do/don't" TERMINAL-TYPE option
 */
int
will_termtype(FILE *sfp, FILE *tfp, int c)
{
    if (termtype) {
        if (option_cmd == TCDO)
            return 0;
    } else if (option_cmd == TCDONT)
        return 0;
    termtype = !termtype;
    if (termtype)
        if (!term && !(term = getenv("TERM")))
            termtype = !termtype; /* can't do it... */
        (void) putc(TCIAC, sfp);
    if (termtype)
        (void) putc(TCWILL, sfp);
    else
        (void) putc(TCWONT, sfp);
    (void) putc((char)c, sfp);
    if (termtype) /* set up binary data path; send WILL, DO */
        option_cmd = TCWILL;
        (void) do_txbinary(sfp, tfp, TOTXBINARY);
        option_cmd = TCDO;
        (void) will_txbinary(sfp, tfp, TOTXBINARY);
    }
    return 0;
}

```

过程 will_termtype 的行为与其他选项处理过程很像。它使用全局变量 termtype 记录服务器以前是否请求过终端类型选项，并且查看当前请求是否改变了状态。如果改变了，它就调用 getenv 获取与环境变量 TERM 相关联的值。如果不存在这个变量，客户发出不愿承诺请求的响应。

服务器请求终端类型信息，以便应用程序可专门为用户终端准备输出。例如，文本编辑器在产生清空屏幕的字符序列时，移动光标时，或加亮文本时，都要使用终端类型。因此，客户期望远程应用一旦收到终端类型信息便发送终端控制序列。由于这种序列不能使用 NVT 编码发送，will_termtype 发送 WILL 报文，宣告客户要使用二进制传输的意愿，并发送 DO 报文请求服务器使用二进制传输。由于函数 do_txbinary 和 will_txbinary 可在有限状态机选项处理代码中被调用，它们使用了全局变量 option_cmd 来控制处理。当直接调用这两个函数时，其他过程必须明确地初始化 option_cmd，就好像客户在调用该函数前，已从服务器收到了相应的 WILL 或 DO 报文。

27.12 选项子协商

一旦客户同意处理终端类型选项，服务器就使用选项子协商来请求终端名。与具有固定长度的正常选项不同，子协商允许发送方在数据流中插入任意长的字符串。为此，发送方通过发送子协商首部（subnegotiation header）、某个特定选项的子协商的数据，以及表示子协商结束的尾部，用来把一个字符串分开。

在上有限状态机（图 26.4）遇到子协商命令序列后，它便进入状态 TSSUBNEG。一旦处于状态 TSSUBNEG，客户每次收到一个字符便调用过程 subopt。正如文件 subopt.c 中代码所示，subopt 运行选项子协商有限状态机来处理子协商。

```
/* subopt.c - subopt */

#include <sys/types.h>

#include <stdio.h>

#include "telnet.h"
#include "tnfsm.h"

extern struct fsm_trans          substab[];
extern int                         substate;
extern u_char                      subfsm[][NCHRS];

/*
 * subopt - do option subnegotiation FSM transitions
 */
int
subopt(FILE *sfp, FILE *tfp, int c)
{
    struct   fsm_trans *pt;
    int      ti;

    ti = subfsm[substate][c];
    pt = &substab[ti];
    (pt->ft_action)(sfp, tfp, c);
    substate = pt->ft_next;
    return 0;
}
```

27.13 发送终端类型信息

选项子协商有限状态机 FSM^① 调用过程 subtermtype 来对终端类型请求做出应答。服务器发送如

^① 见 26.23 节对选项子协商的有限状态机描述。

下序列：

IAC SUBNEG TERMTYPE SEND IAC SUBEND

服务器以此来请求一个终端类型。客户通过发送如下序列做出应答：

IAC SUBNEG TERMTYPE IS term_type_string IAC SUBEND

文件 subtermtype.c 中含有如下代码：

```
/* subtermtype.c - subtermtype */

#include <sys/types.h>

#include <stdio.h>

#include "telnet.h"

extern char      *term;           /* terminal name, from initialization */

/*
 * subtermtype - do terminal type option subnegotiation
 */
int
subtermtype(FILE *sfp, FILE *tfp, int c)
{
    /* have received IAC.SB.TERMTYPE.SEND */

    (void) putc(TCIAC, sfp);
    (void) putc(TCSB, sfp);
    (void) putc(TOTERMTYPE, sfp);
    (void) putc(TT_IS, sfp);
    fputs(term, sfp);
    (void) putc(TCIAC, sfp);
    (void) putc(TCSE, sfp);
    return 0;
}
```

选项子协商有限状态机在收到SEND请求后便调用 subtermtype。在此之前，客户必须已向服务器的请求做出肯定答复，将承诺终端类型选项。因此，全局变量 term 必须已指向一个含有终端类型的字符串。subtermtype 是这样发送应答的，它调用 putc 发送各个控制字符，并调用 fputs 发送含有终端类型信息的字符串。

27.14 终止子协商

当图 26.4 所示的主有限状态机遇到子协商的结束符后，将回到状态 TSDATA。无论在什么时候，只要它要这样做，都将调用过程 subend。subend 简单地把选项子协商有限状态机重新设置到初始状态，以便使它准备处理下一个子协商。文件 subend.c 中含有如下代码：

```
/* subend.c - subend */

#include <sys/types.h>

#include <stdio.h>

#include "tnfsm.h"

extern int substate;

/*
 * subend - end of an option subnegotiation; reset FSM
 */
int
subend(FILE *sfp, FILE *tfp, int c)
{
    substate = SS_START;
    return 0;
}
```

27.15 向服务器发送字符

客户调用过程 soputc 将输出字符转换成网络虚拟终端编码，并通过 TCP 套接字将编码发送给服务器。文件 soputc.c 中含有如下代码：

```
/* soputc.c - soputc */

#include <sys/types.h>

#include <stdio.h>

#include "telnet.h"
#include "local.h"

/*
 * soputc - move a character from the keyboard to the socket
 */
```

```

/*
int
soputc(FILE *sfp, FILE *tfp, int c)
{
    if (sndbinary) {
        if (c == TCIAC)
            (void) putc(TCIAC, sfp); /* byte-stuff IAC */
        (void) putc(c, sfp);
        return 0;
    }
    c &= 0x7f;                  /* 7-bit ASCII only */
    if (c == t_intrc || c == t_quitc) {           /* Interrupt */
        (void) putc(TCIAC, sfp);
        (void) putc(TCIP, sfp);
    } else if (c == sg_erase) {                   /* Erase Char */
        (void) putc(TCIAC, sfp);
        (void) putc(TCEC, sfp);
    } else if (c == sg_kill) {                    /* Erase Line */
        (void) putc(TCIAC, sfp);
        (void) putc(TCEL, sfp);
    } else if (c == t_flushc) {                  /* Abort Output */
        (void) putc(TCIAC, sfp);
        (void) putc(TCAO, sfp);
    } else
        (void) putc(c, sfp);
    return 0;
}

```

当用二进制模式传输时，只有IAC字符需要字符填充。也就是说，soputc必须将每个IAC字符替换成两个IAC字符。而对于其他任何字符，soputc只需调用putc将其发送。

当用正常模式传输时，soputc必须将本地字符集转换成网络虚拟终端字符集。例如，如果到达的字符对应interrupt（中断）字符或quit（退出）字符，soputc将发送一个两字符序列：

IAC IP

程序显式地检查NVT定义的各个特殊字符。soputc还必须处理那些不存在NVT编码的字符。但是，NVT协议规定，如果服务器不请求客户使用二进制传输，服务器将丢弃用户键入的大多数控制字符。

27.16 显示在用户终端上出现的传入数据

通过TCP连接到达的来自服务器的数据可能是未编码的（如果服务器已同意传输二进制数据），或者可能包含有一些按照某个NVT规则编码的字符。客户调用过程ttputc在用户终端上显示传入字符。

```
/* ttputc.c - ttputc */

#include <sys/types.h>

#include <stdio.h>

#include "telnet.h"

int      tcout(char *cap, FILE *tfp);
int      xputc(char ch, FILE *fp);

/*
 * ttputc - print a single character on a Network Virtual Terminal
 */
int
ttputc(FILE *sfp, FILE *tfp, int c)
{
    static    last_char;
    int       tc;

    if (recvbinary) {
        (void) xputc(c, tfp); /* print uninterpreted */
        return 0;
    }
    if (synching)           /* no data, if in SYNCH */
        return 0;

    if ((last_char == VPCR && c == VPLF) ||
        (last_char == VPLF && c == VPCR)) {
        (void) xputc(VPLF, tfp);
        last_char = 0;
        return 0;
    }
    if (last_char == VPCR)
        (void) tcout("cr", tfp);
    else if (last_char == VPLF)
        (void) tcout("do", tfp);
    if (c >= ' ' && c < TCIAC)          /* printable ASCII */
        (void) xputc(c, tfp);
    else {                                /* NVT special */
        switch (c) {
            case VPLF:                  /* see if CR follows */
            case VPCR:      tc = 1;     /* see if LF follows */
                break;
        }
    }
}
```

```
        case VPBEL: tc = tcout("bl", tfp);
                      break;
        case VPBS: tc = tcout("bc", tfp);
                      break;
        case VPHT: tc = tcout("ta", tfp);
                      break;
        case VPVT: tc = tcout("do", tfp);
                      break;
        case VPFF: tc = tcout("cl", tfp);
                      break;
        default:
                      tc = 1;
                      break; /* no action */
      }
      if (!tc)           /* if no termcap, assume ASCII */
                     (void) xputc(c, tfp);
    }
    last_char = c;
    return 0;
}
```

如果服务器已同意发送二进制数据, ttputc 就调用 xputc 显示这些数据。文件 xput.c 中含有如下代码:

```
/* xput.c - xputc, xfputs */

#include <stdio.h>

extern FILE      *scrfp;

/*
 * xputc - putc with optional file scripting
 */
int
xputc(char ch, FILE *fp)
{
    if (scrfp)
        (void) putc(ch, scrfp);
    return putc(ch, fp);
}

/*
 * xfputs - fputs with optional file scripting
 */

```

```

int
xfputs(char *str, FILE *fp)
{
    if (scrfp)
        fputs(str, scrfp);
    fputs(str, fp);
}

```

由于客户提供了一种脚本 (scripting) 设施，`xputc` 不同于传统的 `putc`。如果允许脚本，`xputc` 就将输出字符写入终端和脚本文件中。否则，它只将其写入终端。

如果服务器不发送二进制数据，`tputc` 必须将 NVT 编码转换成用户终端上相应的字符序列。这会发生两种情况：客户可能处于正常模式，或者处于同步模式。客户收到一个 TELNET SYNCH 命令后就进入同步模式。在处于同步模式时，客户将读取并丢弃所有的数据。客户在遇到 TELNET DATA MARK 后会回到正常模式。

服务器将 SYNCH 命令作为紧急数据发送。当紧急数据到达客户时，操作系统向客户发送信号 SIGURG，使得客户执行信号处理过程 `revurg`。`revurg` 设置全局变量 `synching`，使客户进入同步模式，并在数据流中搜索下一个 DATA MARK 字符。客户在同步模式下将抛弃所有输入，而且不把这些字符显示出来。因此在程序中，`tputc` 检查变量 `synching`，并且若 `synching` 非零，则丢弃输出字符。

在 `tputc` 检测过同步模式后，它就必须使用 NVT 编码解释剩余的字符。由于一些 NVT 编码是由两字符序列构成的，因此，`tputc` 在全局变量 `last_char` 中保持前一个字符的副本。

首先，`tputc` 要处理回车 (CR) 和换行 (LF)。它将两字符序列 CR-LF 或 LF-CR 都识别成行结束，并将它们转换成 Linux 所使用的单个字符 LF (行结束)。当然，若回车或换行符单独出现，`tputc` 将完成与该字符相关联的动作。为此，它将调用过程 `tcout`，指明一个光标移动操作作为其第一个参数。例如，如果它遇到了一个换行符，`tputc` 将调用 `tcout`，`tcout` 所带的字符串参数为 `do`，`do` 代表 `down`。

`tputc` 调用 `xputc` 直接打印任何一个可打印的 ASCII 字符。否则，它便对特殊字符进行处理。例如，如果要显示的字符是一个 NVT BEL 字符 (代码中的 VPBEL)，`tputc` 以代码 `bl` 为参数调用 `tcout`。

27.17 使用 termcap 控制用户终端

过程 `tcout` 把标准的 termcap^① 终端兼容名和终端的输出文件指针作为参数。它使用函数 `getenv` 从环境变量 `TERM` 中提取终端类型，然后调用过程 `tgetstr` 查找为在用户终端上获得指明的效果所需的字符序列。最后，它调用 `xfputs` 将所生成的字符序列写到终端上。

```

/* tcout.c - tcout */

#ifndef BSD
#include <newcurses.h>
#else
#include <curses.h>
#endif

```

① 在一些 UNIX 版本中，该库称为 `termlib`。

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

#define TBUFSIZE 2048

int xfputs(char *str, FILE *fp);

/*
 * tcout - print the indicated terminal capability on the given stream
 */
int
tcout(char *cap, FILE *tfp)
{
    static init;
    static char      *term;
    static char      tbuf[TBUFSIZE], buf[TBUFSIZE], *bp = buf;
    char            *sv;

    if (!init) {
        init = 1;
        term = getenv("TERM");
    }
    if (term == 0 || tgetent(&tbuf[0], term) != 1)
        return 0;
    if (sv = tgetstr(cap, &bp)) {
        xfputs(sv, tfp);
        return 1;
    }
    return 0;
}
```

27.18 将数据块写到服务器

telnet 调用过程 sowrite 将数据块写到服务器上。

```
/* sowrite.c - sowrite */

#include <sys/types.h>

#include <stdio.h>

#include "tnfsm.h"

extern struct fsm_trans      sostab[];
```

```

extern int                      sostate;
extern unsigned char           sofsm[][][NCHRS];

/*
 * sowrite - do output processing to the socket
 */
int
sowrite(FILE *sfp, FILE *tfp, unsigned char *buf, int cc)
{
    struct fsm_trans   *pt;
    int             i, ki;

    for (i=0; i<cc; ++i) {
        int         c = buf[i];

        ki = sofsm[sostate][c];
        pt = &sostab[ki];

        if ((pt->ft_action)(sfp, tfp, c) < 0)
            sostate = KSREMOTE; /* an error occurred */
        else
            sostate = pt->ft_next;
    }
}

```

sowrite 遍历指定块中的每个字符，并执行有限状态机 sofsm 处理每个字符。

27.19 与客户进程交互

与大多数TELNET客户程序相类似，我们的实现允许用户与客户进程交互。为此，用户要在每个命令前键入键盘转义字符（keyboard escape character）。图 27.1 中的表格列举了可能的命令及其含义，每个命令都要跟在一个转义字符后。

符号名	键入的字符	含义
KCSUSP	↑ Z	临时挂起客户进程
KCDCON	.	终止 TCP 到服务器的连接
KCSTATUS	↑ T	打印当前连接的状态信息
KCESCAPE	↑ I	将 escape 字符作为数据发送给服务器
KCSCRIPT	s	开始给指明的文件编写脚本
KCUNSCRIPT	u	终止编写脚本

图 27.1 当键盘输入的字符跟在 KCESCAPE 后面，它就被 TELNET 客户解释为命令。记法 ↑ X 是指按住 CONTROL 键并键入 X 所产生的字符

文件 telnet.h^①含有每个键盘命令字符的符号定义。例如，它将键盘转义字符 KCESCAPE 定义成↑，即八进制码为 035 的字符。

当客户遇到键盘转义字符时，它将套接字输出有限状态机的状态从 KSREMOTE 变成 KSLOCAL，并将后继字符解释为一个命令套接字输出有限状态机^②。由于大多数命令由单个字符组成，套接字输出有限状态机通常返回到 KSREMOTE 状态，并执行与命令相关联的动作过程。例如，如果有限状态机遇到 KCESCAPE 之后的字符是 KCDCON，它将调用过程 dcon。

27.20 对非法命令做出响应

如果在一个键盘转义后，用户键入了一个无法识别的字符，套接字输出有限状态机将调用动作过程 sonotsup 打印一个差错消息。文件 sonotsup.c 中含有如下代码：

```
/* sonotsup.c - sonotsup */

#include <stdio.h>

/*
 * sonotsup - an unsupported escape command
 */
int
sonotsup(FILE *sfp, FILE *tfp, int c)
{
    fprintf(tfp, "\nunsupported escape: %c.\n", c);
    fprintf(tfp, "s - turn on scripting\t\t");
    fprintf(tfp, "u - turn off scripting\n");
    fprintf(tfp, ". - disconnect\t\t\t");
    fprintf(tfp, "^Z - suspend\n");
    fprintf(tfp, "^T - print status\n");
    return 0;
}
```

27.21 脚本描述文件

我们的TELNET客户例子具有一个非凡的特性，该特性是无法在其他大多数客户中找到的：它允许用户动态创建脚本文件，该脚本含有被发送到用户终端的所有数据的副本。脚本描述的基本想法是用户可能需要保持 TELNET 会话的所有或者部分记录。

脚本描述是动态的 (dynamic)，这是由于用户可在任何时候启动或终止它。此外，用户可改变该文件（客户把脚本写入该文件）。因此，为了捕捉远程命令的输出，用户在登录到远程系统时脚

① 文件 telnet.h 见 26.13 节。

② 套接字输出有限状态机见图 26.8。

本描述可被禁止，然后再允许脚本描述，接着便发出一个或多个命令，这些命令的输出是要保存的，最后，再禁止脚本描述。这样，该脚本文件将含有在脚本记录有效期间客户在用户终端上所显示的所有信息。

27.22 脚本描述的实现

套接字输出有限状态机如图26.8所示，它定义了客户如何进行脚本描述。如果用户键入↑]s(即KCSCRIPT字符跟在KCESCAPE字符后)，套接字输出有限状态机将调用动作过程scr init并进入KSCOLLECT状态。在用户键入一个行结束符(即KCNL)前，有限状态机将一直处于KSCOLLECT状态，并将调用过程sergetc收集形成脚本文件名的字符串。一旦用户终止了行的输入，有限状态机调用scrwarp过程打开脚本文件，并返回到KSREMOTE状态。以下几节将逐个介绍与脚本描述相关联的动作过程。

27.23 初始化脚本描述

当套接字输出有限状态机首次遇到开始进行脚本描述的请求时，它将调用动作过程scrinit。

```
/* scrinit.c - scrinit */

#include <unistd.h>
#include <termios.h>
#include <string.h>
#include <stdio.h>

#include "telnet.h"
#include "local.h"

extern int          scrindex;
extern struct termios tntty;

/*
 * scrinit - initialize tty modes for script file collection
 */
int
scrinit(FILE *sfp, FILE *tfp, int c)
{
    struct termios      newtty;

    if (!doecho) {
        fprintf(tfp, "\nscripting requires remote ECHO.\n");
        return -1;
    }
    if (scrfp) {
```

```

        fprintf(tfp, "\nalready scripting to \"%s\".\n", scrname);
        return -1;
    }
    scrindex = 0;

    if (tcgetattr(0, &tntty))      /* save current tty settings */
        erexit("can't get tty modes: %s\n", strerror(errno));
    newtty = oldtty;
    newtty.c_cc[VINTR] = _POSIX_VDISABLE;      /* disable interrupt */
    newtty.c_cc[VQUIT] = _POSIX_VDISABLE;      /* disable interrupt */
    newtty.c_cc[VSUSP] = _POSIX_VDISABLE;      /* disable suspend */
#ifndef UDSUSP
    newtty.c_cc[VDSUSP] = _POSIX_VDISABLE;      /* disable suspend */
#endif

    if (tcsetattr(0, TCSADRAIN, &newtty))
        erexit("can't set tty modes: %s\n", strerror(errno));

    fprintf(tfp, "\nscript file: ");
    (void) fflush(tfp);
    return 0;
}

```

scrinit首先验证客户是否正使用远程回显(即所有显示的字符是来自服务器,还是来自本地设备驱动程序)。它还检验用户是否还没有允许脚本描述。scrinit把全局变量scrindex设置为零。另一个过程在读取脚本文件名时,将用scrindex对字符数进行计数。最后,在它打印出提示前,scrinit会改变用户终端的模式,以便本地终端驱动程序在用户键入文件名时能把它打印出来。

27.24 收集脚本文件名的字符

套接字输出有限状态机用动作过程scrgetc来读取将用作脚本文件名的字符序列。文件scrgetc.c中含有代码:

```

/* scrgetc.c - scrgetc */

#include <termios.h>
#include <string.h>
#include <stdio.h>

#include "local.h"

#define SFBUFSZ          2048      /* script filename buffer size */

struct termios   tntty;
FILE            *scrfp;
char           scrname[SFBUFSZ];
int            scrindex;

```

```

/*
 * scrgetc - begin session scripting
 */
int
scrgetc(FILE *sfp, FILE *tfp, int c)
{
    scrname[scrindex++] = c;
    if (scrindex >= SFBUFSZ) { /* too far */
        fprintf(tfp, "\nname too long\n");
        if (tcsetattr(0, TCSADRAIN, &oldtty) < 0)
            erexit("tcsetattr: %s\n", strerror(errno));
        return -1;
    }
    return 0;
}

```

每次到达一个字符，客户就调用 scrgetc，它将字符追加到字符串 scrname 后。

27.25 打开脚本文件

当客户遇到行结束符时，它便调用过程 scrwrap 打开脚本文件。

```

/* scrwrap.c - scrwrap */

#include <sys/file.h>

#include <termios.h>
#include <stdio.h>
#include <string.h>

#include "local.h"

extern struct termios      tntty;
extern char                scrname[];
extern int                 scrindex, errno;

/*
 * scrwrap - wrap-up script filename collection
 */
int
scrwrap(FILE *sfp, FILE *tfp, int c)
{

```

```

int      fd;

if (scrindex) {
    scrname[scrindex] = '\0';
    scrindex = 0;
    fd = open(scrname, O_WRONLY|O_CREAT|O_TRUNC, 0644);

    if (fd < 0)
        fprintf(tfp, "\ncan't write \"%s\": %s\n",
                scrname, strerror(errno));
    else
        scrfp = fdopen(fd, "w");
}
if (tcsetattr(0, TCSADRAIN, &tntty) < 0)
    errexit("tcsetattr: %s\n", strerror(errno));
return 0;
}

```

scrwrap 给收集到的字符串增加一个空终止符 (null terminator)，重新设置全局变量 scrindex 以便它可被再次使用，接着便调用 open 打开脚本文件。如果它成功地获得新的脚本文件描述符，scrwrap 就调用 fdopen 为脚本文件创建标准的 I/O 文件指针，并将该指针放到全局变量 scrfp 中。在它返回前，scrwrap 调用 tcsetattr 将终端模式重新设置为被 scrinit 改变前的值。

27.26 终止脚本描述

当用户决定禁止脚本描述时，套接字输出有限状态机就调用动作过程 unscript。

```

/* unscript.c - unscript */

#include <sys/types.h>
#include <sys/stat.h>

#include <stdio.h>

#include "local.h"

/*
 * unscript - end session scripting
 */
int
unscript(FILE *sfp, FILE *tfp, int c)
{
    struct stat          statb;

    if (scrfp == 0) {

```

```

        fprintf(tfp, "\nNot scripting.\n");
        return 0;
    }
    (void) fflush(scrfp);
    if (fstat(fileno(scrfp), &statb) == 0)
        fprintf(tfp, "\n\"%s\": %d bytes.\n", scrname,
                statb.st_size);
    (void) fclose(scrfp);
    scrfp = 0;
    return 0;
}

```

`unscript` 打印信息报文，告诉用户客户已停止了脚本描述。它使用 UNIX 系统函数 `fstat` 获得有关脚本文件的信息，并打印报文，给出脚本文件的大小。最后，`unscript` 关闭脚本文件，并清空全局文件指针 `scrfp`。

27.27 打印状态信息

用户通过一个跟在键盘转义字符后的 KCSTATUS 命令，可获得有关当前连接的状态信息。套接字输出有限状态机调用动作过程 `status` 打印连接状态。

```

/* status.c - status */

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

#include <stdio.h>

extern char      doecho, sndbinary, rcvbinary; /* telnet options */
extern char      *host, scrname[];
extern FILE      *scrfp;

/*
 * status - print connection status information
 */
int
status(FILE *sfp, FILE *tfp, int c)
{
    struct sockaddr_in      sin;
    int                      sinlen;

    fprintf(tfp, "\nconnected to \"%s\" ", host);
}

```

```
sinlen = sizeof(sin);
if (getsockname(fileno(sfp), (struct sockaddr *)&sin,
                 &sinlen) == 0)
    fprintf(tfp, "local port %d ", ntohs(sin.sin_port));
sinlen = sizeof(sin);
if (getpeername(fileno(sfp), (struct sockaddr *)&sin,
                 &sinlen) == 0)
    fprintf(tfp, "remote port %d ", ntohs(sin.sin_port));
(void) putc('\n', tfp);
if (doecho || sndbinary || rcvbinary) {
    printf("options in effect: ");
    if (doecho)
        fprintf(tfp, "remote_echo ");
    if (sndbinary)
        fprintf(tfp, "send_binary ");
    if (rcvbinary)
        fprintf(tfp, "receive_binary ");
    (void) putc('\n', tfp);
}
if (scrfp)
    fprintf(tfp, "scripting to file \'%s\'\n", scrname);
return 0;
}
```

过程status打印的信息有：远程主机名、连接使用的本地和远端TCP协议端口，以及有效的选项清单。

27.28 小结

我们的TELNET例子使用了三个有限状态机，用以解释从用户键盘或服务器到达的字符序列。每个传入字符将引起有限状态机的一次转移。当客户执行转移时，它便调用实现该转移相关动作的过程。

本章描述了客户例子中包含的三个有限状态机所使用的动作过程。一些动作简单，而另一些却很复杂。将客户软件组织成有限状态机的动作过程，这样做的主要缺点在于其可读性不强。由于我们只能参考有限状态机才能确定过程间的关系，产生的代码可能难于理解。

深入研究

一系列 RFC 记录了TELNET选项的细节，并含有本例子代码所处理的各个选项的协议标准。Postel 和 Reynolds [RFC 858] 讨论了 go-ahead 选项，而 Postel 和 Reynolds [RFC 857] 讨论了字符回显。Postel 和 Reynolds [RFC 856] 描述了控制8比特二进制传输的选项。最后，VanBokkelen [RFC 1091] 讨论了终端类型选项和相关的选项子协商。

习题

- 27.1 一些终端类型支持多个仿真模式，这使得一个终端可能有一组终端类型名。阅读 RFC 1091。客户在与服务器协商终端类型时，如何使用终端名的清单？
- 27.2 阅读协议标准以便准确地找出服务器在什么时候必须从发送网络虚拟终端编码数据切换到发送 8 比特二进制数据？具体来说，服务器自愿传输二进制数据，但还没有收到客户的确认时，它如何处理传输？
- 27.3 在客户请求服务器执行指定选项时，客户发送 WILL 或 DO 吗？服务器请求客户执行一个选项时，它发送什么？
- 27.4 过程 `scrwrap` 中对 `open` 的调用，`open` 的模式参数 `O_WRONLY|O_CREAT|O_TRUNC` 是什么意思？
- 27.5 命令客户收到一个选项请求时打印一个报文。用这个修改后的客户联系多种服务器。这些服务器都自动发送了哪些选项请求？
- 27.6 如果客户发送了 DO ECHO，而同时服务器发送了 WILL ECHO，会发生什么？
- 27.7 如果客户给已经允许 ECHO 的服务器发送 DO ECHO，会发生什么？

第28章 流式音频和视频传输（RTP概念和设计）

28.1 引言

在前面两章中，我们考察了客户-服务器交互的各种细节，它们被常规的面向字符的应用程序所使用。本章和下章将集中讨论发送和交付音频和视频信息的各种应用。本章将解释流式传输服务的基本概念，给出一个用于流式传输的协议，并讨论实时库的组织。下章将讨论一个例子，它是一个可以接收和播放实时音频流的软件实现。

28.2 流式传输服务

流式传输服务（streaming service）与那种在每个请求和响应中只传送固定长度的服务不同，它能提供一种连续的、任意长度的数据流。数据流在长度上没有限制，在传送时间上也没有限制。对大多数情况来说，流在理论上是无限的——发送者可以没完没了地提供数据。还有的情况是，持续时间虽然是有限的，但流可以保持很长时间（比如几个小时）。

流式传输被典型地用于传送音频和视频实况。在发送端，由摄像机或话筒提供连续信号，这些信号经数字化后发送到因特网。在接收端，数字流被转换回模拟信号并加以播放。由于数据是连续产生的，只要摄像机或话筒还在工作，流就不会结束。

28.3 实时交付

在一个应用中，如果数据的交付必须与数据的产生保持精确的时间关系，我们把这种应用叫做实时应用。流式传输总是和实时交付相关联，因为像音频和视频这种适合流式传输的应用总是要求实时交付。例如，音频实况是实时的，因为接收方必须精确地按与原来的声音相同的时间序列播放。

实时交付协议软件要求处理两方面的问题：

顺序。这点很明显，进来的数据必须精确地按其产生时的顺序播放。

时间。为了精确地再现输入信号，接收方必须知道产生每个分组时的精确时间。接收方把数据转换回模拟形式，并把这一片片模拟信号精确地按照其输入时间播放出来。

28.4 抖动的协议补偿

但是，下层的网络对顺序和时间都可能产生破坏。例如，当数据流在因特网中穿越时，各个相继分组的时延可能有相当大的不同。事实上，网际协议（Internet Protocol）的交付语义允许数据报与其前驱和后继分组有所交错，因而数据报的到达就可能失序。

时延的变化在技术上称为抖动。抖动意味着应用软件不能依赖下层的网络来提供合乎要求的重放。例如对音频传输，抖动意味着尽管发送方每 20 ms 传送一个数据报，但数据报并不能精确地按照每 20 ms 一个的顺序到达接收方。

为了保证实时数据的正确回放（playback），协议软件必须对失序分组，并对穿越了具有较小抖动的网络而到达接收方的分组提供补偿。协议在每个分组中设置了两个字段来处理这两个问题。发送方在每个分组上设置一个序号；接收方使用序号信息来保证按照与发送相同的循序来处理分组。发送方还在每个分组上设置一个时间戳，该时间戳记录分组中数据的录制时间；接收方使用时间戳来保证当前数据回放的时间与前一分组的回放具有正确的时间关系。

在实时协议中，把序列信息与时间戳信息分离开是很关键的，因为这样就可以把处理分组与数据回放定时分离开。例如，考虑一个音频编码应用，它通过交替发送左右声道的采样信号来发送立体声信号，在这种场合下，两个相继的分组可以具有相同的时间戳，对此，把序号与时间戳分开设置可以让接收方知道正确的顺序。同样，当信号输入空闲而没有数据要传送时，把序号与时间戳分开设置也发挥了重要作用。为了理解其中的原因，我们可以设想一下，当出现一段时间的静默而使传输挂起时，尽管没有分组要传送，但用于时间戳的时间还是在流逝，因此，当负责发送的应用经过 N 个时间单元的静默又开始传送数据时，时间戳将比上一个分组的时间戳大 N。从接收者的观点看，时间戳看上去是在两个相继分组间有一个 N 单位的跳跃，虽然时间戳向前跳跃了，可是接收方知道分组并没有丢失，因为序号只增加了 1。

28.5 重传、丢失和恢复

实时协议虽然能够处理时延和重排序问题，但是它并没有采用确认（acknowledgement）和重传机制来处理分组丢失问题。为了理解此中原因，我们不妨重申一下，实时应用使用时间-序列的（time-sequenced）回放——数据以一个不变的速率到达，回放也必须以一个不变的速率进行。例如，考虑音频重传的影响，如果一小段音频数据没有到达，而回放刚好播放到这段丢失的数据，输出只好临时挂起。如果协议延迟回放并等待重传，新数据还会继续到达，未播放数据队列就会增长。一旦重传数据达到，则继续重放，但队列还会保持在增长后的长度上，这是因为新数据到达的速率与播放的速率一样。因此，尽管队列增长只是在每次重传时才发生，但它再也不会缩短了。事实上，当重传不断发生时，队列会无限增长。

为了避免出现任意长的队列长度，实时协议没有使用重传机制，而是使用了前向纠错（forward error correction）方法或者跳回升放（skipped playback）。前向纠错方法是在每个分组里增加额外的信息，接收方利用到达分组里的这些信息，通过数学方法重新生成丢失分组中的数据。跳回升放则是用一段与丢失数据所占时间相同的数据替代输出。例如，当音频数据在传输过程中丢失，接收方可以停止输出，重复刚刚收到的上一片声音，或者产生白噪声^①（静态背景，background static）。

28.6 实时传输协议

在因特网中，实时传输协议用来处理顺序和时间问题。正如所料，每个 RTP 分组包含有一个序号和一个时间戳，接收方用此信息来控制回放。为了保持充分的一般性以便支持各式各样的应用，

^① 很少完全停止输出，因为经验表明，与绝对安静相比，人们更喜爱白噪声。

RTP 并没有详细说明时间戳的精确解释、数据采样率以及编码，而是定义了一个很小的、固定的头部，在每个分组的开始，必须给出这个头部，接着就可以让应用来指定分组余下部分的细节、数据编码以及语义。图 28.1 展示了 RTP 固定头部的各个字段。

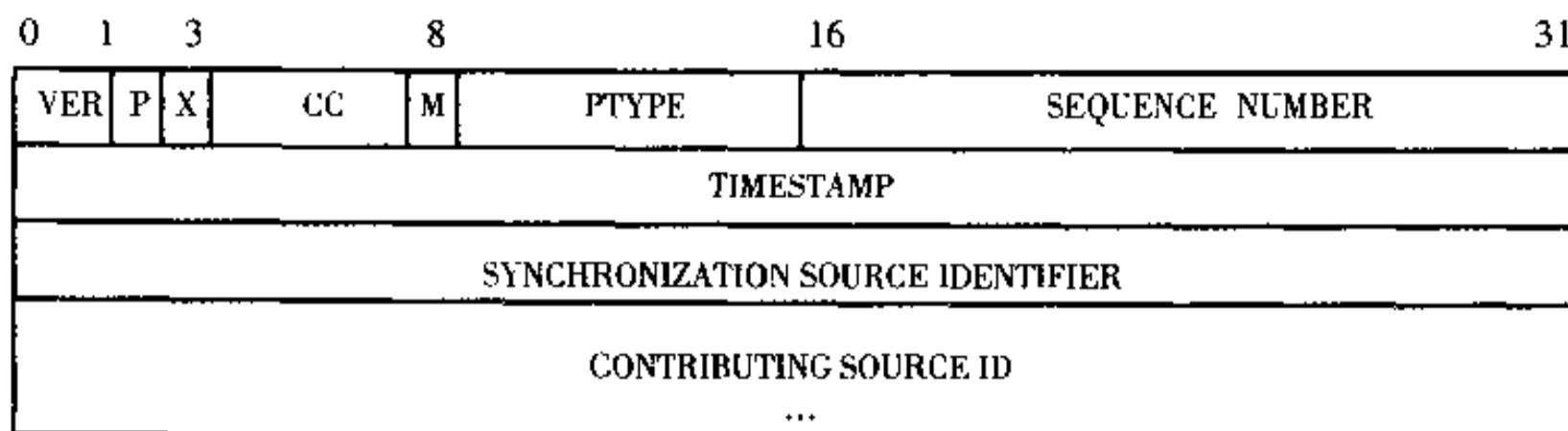


图 28.1 出现在每个 RTP 分组起始的固定头部

RTP 为该头部中的两个字段指派了含义，这些字段用于处理各个传入的分组。两比特的 VER 字段指定协议的版本；当前的版本是 2。16 比特的 SEQUENCE NUMBER 字段可以让 RTP 按序放置各分组。一次会话的初始序号是随机选择的。这有两个原因，一是使用随机的序号可以避免重放，当某次会话的分组在网络上延迟后，却被稍后的另一次会话所接受，这就会遇到重放问题；另一个原因是使用随机的起始序号可以提高安全性，使攻击者猜测序号信息更为困难。

对 RTP 首部中其余大多数字段的解释取决于 7 比特的 PTTYPE 字段，它指定了负载的类型。例如，一种音频编码（PCM）的负载类型是 0。其他负载类型要求数据按长度固定的块来发送。如果分组是加密的，某些加密方案也要求长度固定的数据块。在这些情况下，P 比特就被用来说明是否在分组后增补了零填充（zero padding），从而使分组达到要求的长度；填充的最后字节指明了填充的长度。其他负载类型使用 M（标记）比特标记分组。例如，视频传输使用 M 比特来标记帧的开始。

32 比特的 TIMESTAMP 字段用于确定分组中数据的正确回放时间。时间戳的值给出了分组中数据的第一个字节的采样时间。更重要的是，要求发送方时间戳的时钟是连续增长的，即使在没有检测到输入信号或者没有数据要发送时也是如此。这样，在静默时，发送方不必生成一些无用的数据，接收方通过时间戳就可以确定要在输出间保持多大的间隔。

RTP 规定，一次会话的初始 TIMESTAMP 必须随机选择；但协议并没有规定 TIMESTAMP 的单位，也没有规定关于该值的精确解释，而是由负载类型来确定时钟的粒度。这样，各种应用类型就可以根据需要选择合适的输出计时精度。例如，在 RTP 上传输音频数据时，可以选定逻辑时间戳速率与采样速率相同，这样做是合理的。但是，当传输视频数据时，必须使时间戳速率大于每帧一个滴答，这样才能使图像回放更为平滑。最后，如果数据是在同一时刻采样的，协议标准还允许两个分组具有相同的 TIMESTAMP 值。例如，如果一个大帧被分为多个小分组，这时， TIMESTAMP 值就可能出现重复，因为该帧是在同一时刻录制的，所以每个分组中的 TIMESTAMP 值是相同的。

28.7 流的转换和混合

RTP 允许两个或多个数据流混合（即组合为一个流），还允许对流进行转换（即改变编码）。为了便于接收方对流入的分组进行多路分用，每个数据源选定一个 32 比特的整数作为数据源标识符。这个数据源标识符放在字段 SYNCHRONIZATION SOURCE IDENTIFIER（同步源标识符）中。如

果某个中介系统将两个流合并为一个新流，那么该中介系统将成为这个新流的源。RTP 允许混合器包含一个或多个 CONTRIBUTING SOURCE ID（贡献源标识符）字段来指明最初的发送者。如果混合了多个源，就用 4 比特的 CS 字段指明贡献源（contributing source）的个数。

28.8 迟延回放和抖动缓存

由于实时数据是连续的，抖动问题就成了回放软件的另一个约束：当数据刚一到达时，回放并不能开始。为了理解这点，我们可以考虑一下，在进行音频传输时，如果立即开始回放会发生什么情况。假设回放软件正在播放当前分组中的数据，如果下一分组刚好在播放完数据后到达，这时回放不会中断，能继续播放。但如果下一分组延迟了，回放必须暂停，直到下一分组到达才能继续。更重要的是，当回放重新开始时，不能从这一新到达的分组开始（否则未播放数据队列就会开始增长），而必须从该分组中与当前时刻相对应的那一点开始播放。因此，听众就会错过一小段声音（这段时间等于第二个分组的延迟时间），然后再向前跳跃到如果分组是按时到达时所应到的回放点上。

由于抖动是经常发生的，因此分组到达时间出现少量时延很常见。为了避免输出在每段时延上都产生空隙，回放软件使用了一种具有探索性的技术，称为迟延回放（delayed playback）。这项技术是这样的，接收方在收到第一个数据时并不马上开始回放，而是将传入数据放入缓存中，该缓存称为抖动缓存（jitter buffer）或回放缓存（playback buffer），该区按照先进-先出队列进行操作。接收方设置一个门限值 K，该值比预计的最长的抖动时间要大。开始时，回放等待数据在缓存中积累，直到缓存中的数据足够播放 K 个时间单元。缓存大小一旦达到 K 个单元，回放便开始，数据被以不变的速率从缓存中提取出来。当新分组达到时，分组中的数据被加到缓存中，只要没有分组的时延超过 K 个时间单元，缓存中就有足够的数据用于连续、不中断地回放。图 28.2 是这一概念的图示。

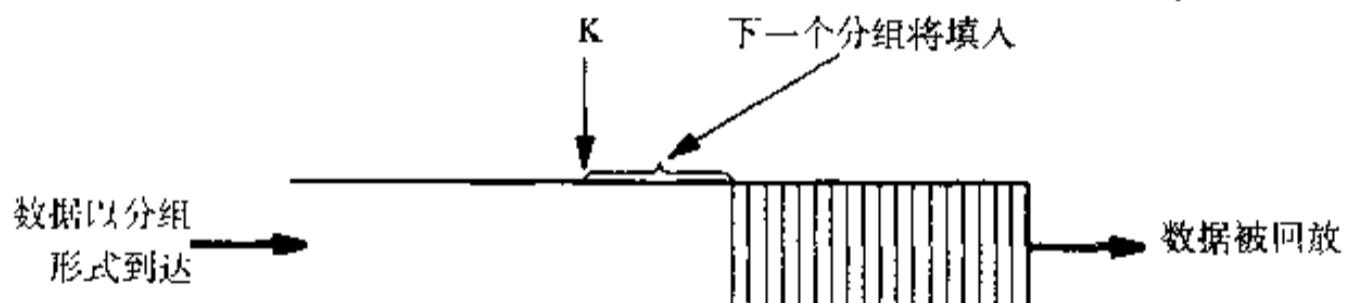


图 28.2 抖动缓存的一个图示。图中所展示的时刻是缓存正期待接收分组。

通过以不变的速率从缓存中提取数据达到连续不中断地回放

图中所展示的时刻是缓存正期待接收分组。在回放开始时，缓存包含 K 个时间单元的数据。因此，如果分组到达的速率与提取数据的速率相同，每个新到达的分组将使缓存重新填回到位置 K 上。但如果某个分组延迟了，回放还能继续从缓存中提取数据，直到新分组到达。如果某个分组提前到达了，在短时间内，缓存中所包含的数据就会超过 K 个时间单元。

28.9 RTP 控制协议 (RTCP)

尽管 RTP 提供的信息可以使接收方重新生成实时输出，但其协议头部中没有包含能让通信端点控制元信息（meta-information）传输或通信的字段。这些元信息特别重要，因为某些实时编码是

自适应的 (adaptive)，这意味着当下层网络的条件有所改变时，编码也随着改变。例如当下层网络分组丢失率很高时，可以采用另一种编码。

为处理所有与会话相关的通信，RTP 首部中没有为此预留空间，而是使用了另一个新协议与 RTP 配合。这一单独的协议称为 RTP 控制协议 (RTCP)。使用 RTCP 时，接收方监视下层网络的性能，并将有关信息通知回发送方。此外，RTCP 还可以让发送方提供有关每次会话的信息。更重要的是，RTCP 可以传输复合分组，即多个 RTCP 报文组合为一个分组。

每个 RTCP 分组以一个固定的首部开始，如图 28.3 所示；分组中其他字段的格式取决于分组首部中的类型字段。

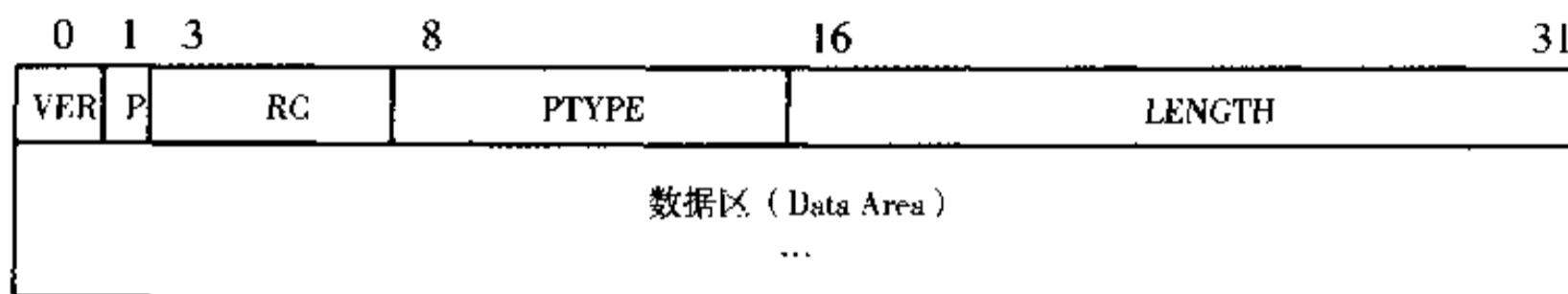


图 28.3 在 RTCP 分组起始 4 字节中的各个固定字段。

PTYPE 字段决定数据区 (Data Area) 的格式

如图所示，RTCP 首部中的前两个字段与 RTP 首部一样。VER 是两比特的版本号，RTP 的当前版本是 2；同 RTP 一样，RTCP 也允许在需要时对分组进行填充。一比特的 P 字段指明是否进行了填充，填充数据的最后一个字节表示填充数据的计数。

分组的数据区包含一系列报告记录，5 比特的 RC 字段含有报告计数 (Report Count)，它表明首部后面的报告数。16 比特的 LENGTH 字段用于说明分组的全部长度，其中包括首部长度。长度以 32 比特字为单位度量，首部中的值定义为该长度减一。

8 比特的 PTTYPE 字段用于说明分组类型。接收方根据分组类型确定跟随在首部后的报告格式和内容。图 28.4 列出了五种报告类型。

名称	类型	意义
发送方报告	200	每一个同步源的时间信息和已发送的数据八位组的计数
接收方报告	201	分组丢失和抖动的报告，以及定时和往返时间估计的信息
源描述	202	拥有信源的用户的描述
再见	203	接收方离开会话
应用	204	特定应用的报告

图 28.4 五种 RTCP 报告类型及其用途。这些类型决定了 RTCP 分组中报告的格式

28.10 多种流同步

RTCP 的一个关键作用是能让接收方同步多个 RTP 流。例如，考虑使音频伴随视频一起传输。由于传输音频和视频的编码不同，RTP 使用了两个流分别进行传输，这两个流的时间戳时钟以不同速率运行。如果接收方将两个流独立播放，结果将毫无意义（即声音和影像不匹配）。因此，接收方必须同步回放两个流，以保证声音与影像完全协调一致。但是，无论 SYNCHRONIZATION SOURCE IDENTIFIER (同步源标识符) 字段还是 TIMESTAMP 字段，都没有为此目的提供充分的信息，这一信息将来自 RTCP。

为了能进行流同步，RTCP 要求每个发送方要为每个活动流（active stream）传送信息。发送方要传送源描述（source description）报文并周期性^①地产生发送方报告（sender report）报文。源描述报文含有一个惟一标识数据源（即发送数据的应用程序）的规范名（canonical name）。尽管由给定数据源所发送的每个流都有一个不同的 SYNCHRONIZATION SOURCE IDENTIFIER，但它们都具有同一个规范名。

发送方报告报文所包含的信息可被接收方用于协调两个流中的时间戳值。图 28.5 展示了有关格式。如图所示，发送方报告开始是一段 28 字节的信息，该信息含有接收方用于同步流的时钟信息。开始，报告含有一个以网络时间协议 NTP（Network Time Protocol）格式表示的绝对时间值。接着，RTCP 报告给出一个 RTP 时间戳值，产生该值的时钟就是产生 RTP 分组中 TIMESTAMP 字段值的那个时钟。由于发送方发出的所有流和发送方报告都使用同一个绝对时钟，接收方就可以比较来自数据源两个流的绝对时间，从而确定如何将一个流中的 TIMESTAMP 值映射为另一个流中的 TIMESTAMP 值。

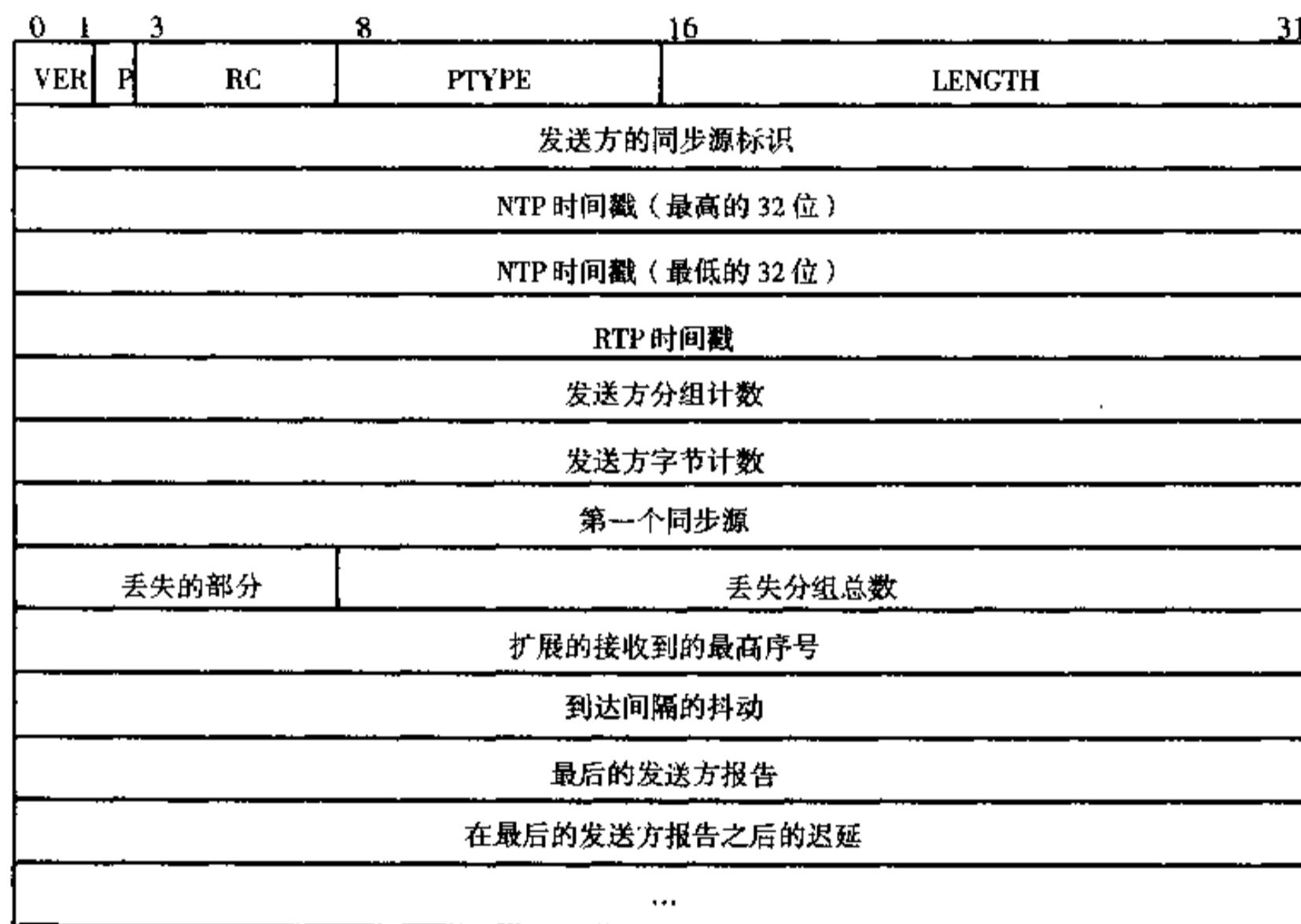


图 28.5 含有发送方报告的 RTCP 分组。分组的最后部分包含来自接收方的一系列报告块。

28.11 RTP 传输和多对多传输

RTP 被设计成运行在无连接传输协议之上，在大多数情况下，使用的是 UDP 协议。发送方将每个 RTP 分组封装在 UDP 报文中传送，接收方再从传入的报文中将分组提取出来。RTP 选用无连接协议是有多个原因的。首先，RTP 不期望重传，这点我们已经讨论过。其次，由于 RTP 自己能处

^① 协议还要求每个接收方周期性地产生接收方报告；接收方报告告知发送方网络状况。

理流控制和顺序化，所以不需要传输协议进行同样的工作（这样效率不高）。第三，RTP 是面向分组的（TCP 不是）。第四，由于很多流式传输的应用要使用组播（multicast）机制，RTP 要被设计为能运行在组播环境中，因此，要使用 UDP 这样的无连接的运输层协议，因为面向连接的运输层协议不适合于组播。

UDP 的一个主要优点在于，它能够在实时环境中提供任意方式的通信。它不但能让一对应用之间进行通信，还可以让单个源向多个接收方进行组播。与此类似，它还可以让任意一组应用向某一个接收方发送数据。最后，它甚至能允许任意一组应用向任意一组接收方发送数据。用数学术语来说就是，UDP 支持一对一、多对一、一对多以及多对多的通信方式。

多个参与者（participant）之间要进行通信，其通信方式要求对典型的客户—服务器模式进行一些修改。在进行组播时，我们要指定一个组，而不是指定某台计算机或这台计算机上的协议端口作为通信的端点。而且，组播组的成员还是动态变化的——各个成员可以在任何时候加入或离开组。其结果是使 RTP 很复杂。我们可将要点总结如下：

由于 RTP 能以组播的方式到达，所以接收计算机上的软件比传统客户—服务器软件要复杂得多。

28.12 会话、流、协议端口和分用

在我们考虑实时协议软件是如何组织的这一问题之前，还有一些基本概念需要解释。第一个概念是，我们需要指定一组参与者。我们这样定义 RTP 会话（RTP session）：一个 RTP 会话包括传给某个指定目的地对（destination pair）的所有通信量，而发送方可能有多个：

(IP 地址, 协议端口号)

注意，如果 IP 地址是一个单播地址，RTP 会话的接收方只有一个；如果 IP 地址是一个组播地址，其接收方就会有多个。组播组尤其重要，因为它允许成员之间任意交换数据。如果不使用组播组，一个 RTP 会话就只有一个接收方。

第二，我们将对传送相关数据的分组序列加以定义。我们将从一个同步源发出的 RTP 分组序列定义为流（stream）。一个 RTP 会话可能包含一个或多个独立的 RTP 流，这些流都被指定送到同一个目的地址和协议端口。例如，在一个多路音频会议中，每个与会者都可以发送独立的音频流。

虽然大多数应用都是让每个源一次只发送一个流，但在必要时一个源也可以发送多个流。例如，一个源可能需要传输多种类型的数据（例如音频和视频）。另外，一个源也可选择用多种编码发送一组数据（例如，用高、低两种分辨率发送同一个视频流）。

接收端的计算机为保证收到的每个 RTP 报文都是与当前流相关联的，必须对收到的 RTP 报文进行分用。有两个级别的分用。第一级分用——会话分用——发生在运输层。一个希望接收 RTP 数据的应用必须创建一个 UDP 会话。类似常规的服务器，应用还应指定一个特殊的协议端口号。与此类似，一个等待接收单播数据的音频应用可以使用 IP 地址 INADDR_ANY。但是，如果音频数据通过组播方式送到，服务器必须给出一个特定的组播 IP 地址。不管什么情况，传输软件都使用地址进行会话级分用。也就是说由 UDP 检查每个分组的（IP 地址，协议端口号），从而判断出该计算机上的哪个应用该接收此会话。

第二级分用——流分用——发生在分组已交给 RTP 软件后。RTP 使用同步源标识符和分组类型把同一个流中的分组合起来。只有在分组与一个流建立关联后，才能用 SEQUENCE NUMBER

(序号) 和 TIMESTAMP (时间戳) 字段对分组进行排序，并决定回放的时间。

28.13 编码的基本方法

应用程序使用 RTP 的方式是多种多样的，这是 RTP 软件之所以复杂的重要原因。一个应用程序与另一个应用程序相比，即使在基本模式上也有可能不同。例如，数据的编码就有两种基本方法。

采样式样编码 (Sample Style Encoding)。发送音频的应用程序通常使用字节流编码，它往往采用脉码调制 PCM 的形式来表示数字化的值。在 PCM 流中，每项 (item) 数据代表一个采样。例如，根据电话系统所使用的 G.711 标准，发送方每秒对输入信号进行 8000 次采样，并把每次采样值编码为 0 到 255 间的某个值。因此，PCM 编码的输出由字节流组成，它没有内部结构——接收方可以在流的任意一点开始回放。高品质话音也同样使用采样式样编码，只不过采样的编码更大（例如，每个采样 16 比特）或者使用了更高的采样速率。

帧式样编码 (Frame Style Encoding)。这种编码方式与采样式样编码不同，它要进行组帧 (framing)，即发送方将数据分割为一段段称为帧的离散的单元。关于组帧的最好示例是视频传输，在视频传输中，要把每一扫描（即，每一屏）放置在帧里。为了节约带宽，许多视频编码技术使用了不同的组帧方案，这种方案是发送方只周期性地传送一个满屏的视频帧，此满屏帧后的帧只需把相应的更新（上一个满屏帧与当前帧的差异）发送出去。每一帧要有一个帧首部（这样发送方就可以指明该帧包含的是一个满屏还是一个更新）。

由于缺乏相应的标准，帧编码更为复杂——每种编码要指明帧化的精确的细节，包括帧格式以及对帧首部的解释。有些编码方法使用固定长度的帧；有些却允许帧长度可变。更重要的是，没有为如何封装达成一致。有些编码方法用一个单独的 RTP 分组发送一个帧；有些使用短帧编码，往往允许多个帧占用一个 RTP 分组；而有些使用长帧的编码方法，则允许一个帧跨越几个 RTP 分组。

28.14 RTP 软件的概念性组织

许多使用实时传输的应用都把 RTP 代码集成进了应用程序本身，RTP 处理部分与应用程序所要进行的解码和回放等功能没有明显的区别。更重要的是，各个应用的 RTP 代码似乎都是各自从头做起的。尽管由应用各自定制代码可能使程序的效率最高，但每个应用各自重新编写协议毕竟是重复劳动。为什么程序员偏要从头实现 RTP 而不是构建一个可以重用的库呢？如果创建一个支持实时软件的库，它又应该为应用提供哪些功能呢？这个库又该如何组织呢？与把对实时功能的支持直接集成进应用程序相比，这种通用的库增加了多少复杂性呢？为了回答这些问题，我们将考虑这种库所需要提供的功能，并将其与一个特定应用所需的功能相比较。本章描述的是一般情况，下一章将给出一个特定应用的实现。

图 28.6 说明了构成支持实时数据流的软件的四个概念性的层次。

如图所示，实时软件位于应用层和传输协议之间。实时软件又分为两个概念性的片，下层处理实时传输，上层提供同步。下层与 UDP 交互（通过套接字），它接受会话中所有传入的分组，将其分用成流。它还收集 RTCP 所需的各种信息，以便针对每个流发送接收方报告。上层提供应用程序所使用的接口，此外，上层还要处理时钟同步问题——它利用发送方报告，对来自同一个规范源 (canonical source) 的各个流的 TIMESTAMP 字段值进行映射。其结果是，读取同步流的应用可以接收到同时产生的数据。图 28.7 给出了 SYNCHRONIZATION 层的进一步的细节。

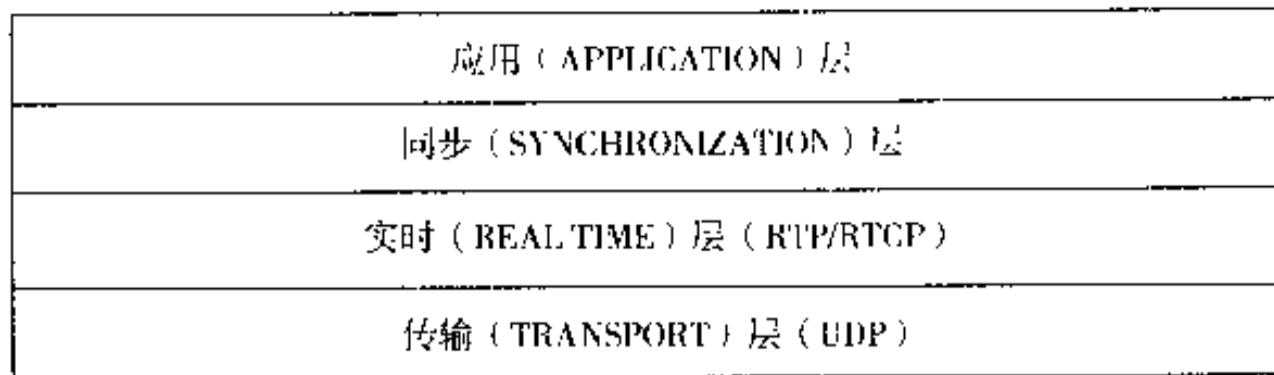


图 28.6 实时数据传送软件的概念性组织结构。SYNCHRONIZATION (同步) 层并不实现某个特定的协议。



图 28.7 SYNCHRONIZATION 层的结构。该层为应用提供接口，除了进行数据传送外，应用还需要控制会话和流。

如图所示，SYNCHRONIZATION 层提供了两个概念性的接口功能。首先，软件提供了一个让应用执行管理工作的控制接口（例如，获得会话中的流的有关信息）。第二，软件提供一个供应用接收数据的数据传送接口。尽管应用可能只注意到单一的数据传送接口，但从内部看（即对应用是不可见的），软件可能使用三种机制之一接收数据。如果流使用帧式样编码，软件就提取并交付帧。如果流使用采样式样编码，软件就提取并交付字节。最后，如果流没有使用这两种常规式样，软件便允许应用访问原始 RTP 分组。

28.15 进程 / 线程结构

RTP 软件必须并发管理多项任务。例如，每个 RTP 数据分组到来时，必须检查其序号，并将数据内容提取出来放置到抖动缓存中。此外，软件还必须处理传入的 RTCP 报文，同时，软件也必须周期性地产生并发送 RTCP 接收方报告。图 28.7 只是实时软件的静态分层结构，实际的实现需要多个线程来执行以便并发地处理多项任务。图 28.8 显示了一种可能的线程结构。

如图所示，除了执行应用程序的线程之外，还有四个线程执行实时软件。RTP 线程在两层之间完成数据传送功能，它从传入的分组中提取数据并管理抖动缓存。单独的 RTCP 线程处理传入的 RTCP 分组。TIMER（计时器）线程周期性地产生外发的 RTCP 接收方报告。最后，EVENT（事件）处理正常数据流之外的其他异步事件。

在我们的模型中，当要求时间同步时，各线程间要发生交互。尽管每个应用都有一个实时“栈”的副本，线程间必须进行交互以保证同步回放。如图 28.9 所示，为了达到同步，两个应用通过传递报文进行通信。

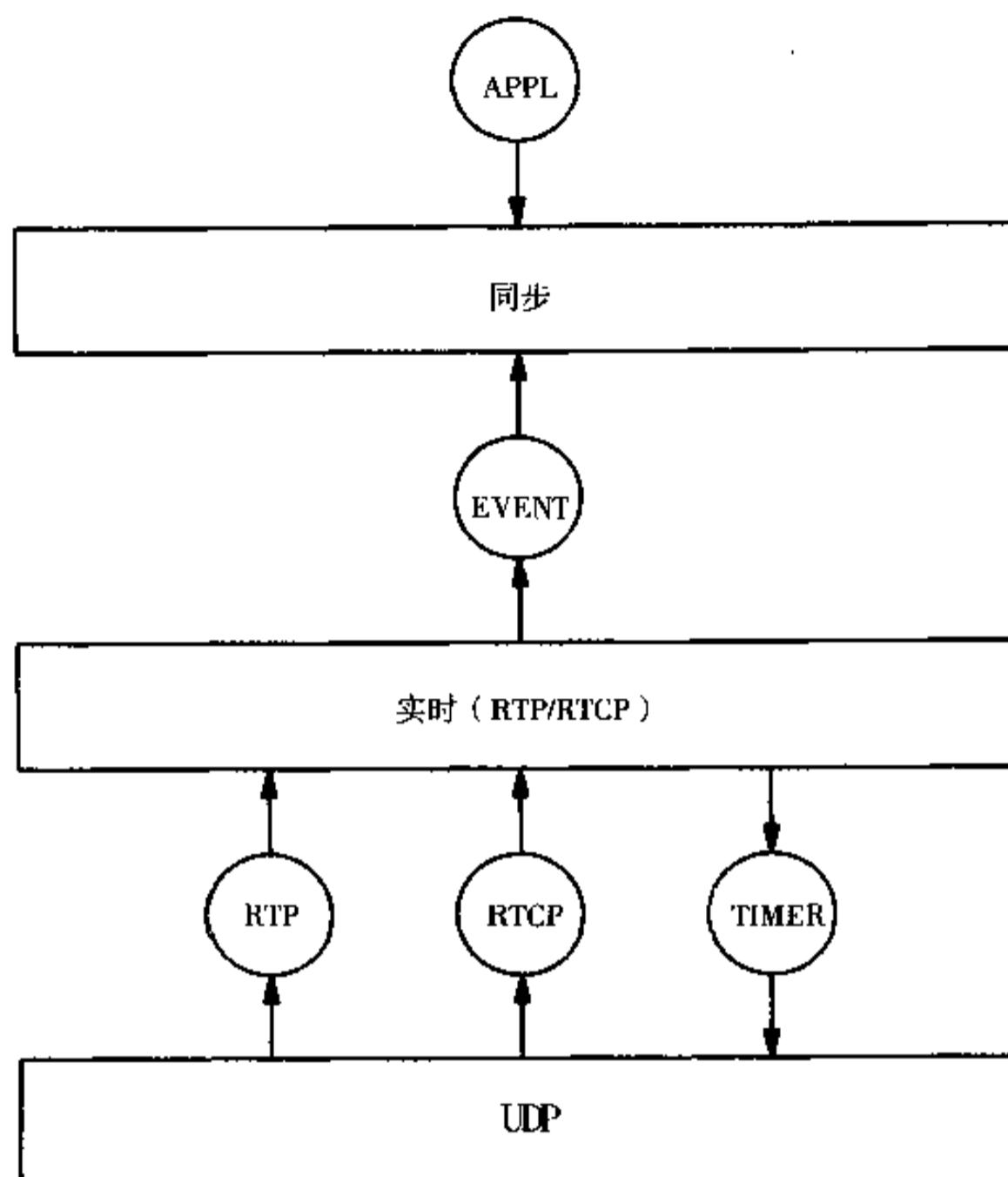


图 28.8 实时软件的线程结构。层与层之间的数据传送与控制和报告事件是分离的

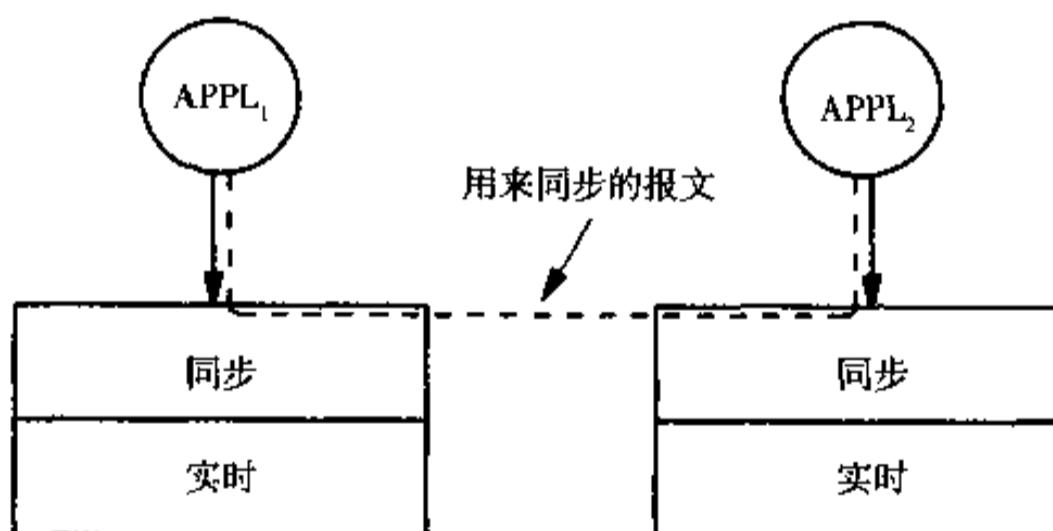


图 28.9 两个应用线程之间的通信。这两个应用线程各自处理一个实时流。如果两个流要求同步，可以使用这种协调方式

28.16 API 的语义

应用接口应给访问实时数据的程序提供什么样的语义呢？我们已经讨论过面向流的和面向帧的编码的差异，然而，这又有一个问题：时间如何表示？对此，有两种选择。

显式时间管理（Explicit Time Management）。 接口往往返回伴随数据的时间戳。应用程序决定何时播放数据（即应用在发送方的时间戳和本地计算机时钟之间做出映射）。

隐式时间管理（Implicit Time Management）。当应用程序首次说明要接收某个特定的流时，实时软件要计算出本地计算机时钟和流的媒体时钟之间的映射关系。只要应用程序试图读取更多数据，实时软件便将来自本地计算机时钟的当前时间映射为流的媒体时钟，并返回该时间点的数据。

使用显式时间的主要优点在于能使应用程序对映射和回放保持绝对控制。如果在一小段空闲后，又开始读取数据，应用程序会接收到空闲前最后读到的数据之后的所有数据，因此，跳跃播放可能效率不高，因为，这会使应用程序读取了数据但又丢弃了它。

使用隐式时间的主要优点在于易于编程——应用程序打开流，读取数据，然后进行播放。应用程序不需要计算时间，因为，下层系统总是返回对应当前时间的数据。如果应用程序在几秒钟的空闲后又开始读取数据，下层系统会将缓存中的一些数据跳跃过去，这些数据对应那段流逝去的时间。

隐式时间管理的主要缺点之一是不能提供绝对控制——这尤其体现在发生错误或异常情况下。例如，假设某个使用采样式样编码的应用程序，企图在对应于当前时间的数据到达之前就进行读取。实时软件有两种选择，它可以阻塞应用程序，直到数据到达，也可以返回一个错误报文。又例如，假设某个使用帧式样编码的应用程序在某个时间T请求读取数据，进一步假设对应于时间T的数据在帧的中间。此时，接口是阻塞应用程序，直到可以返回下一帧为止呢？还是仅仅返回一个差错报文，而让应用自己过后再试读呢？显然，某些应用希望被阻塞，而还有些应用希望收到差错报文。

28.17 抖动缓存的设计和重新缓存

抖动缓存的实现也使实时软件变得复杂。应该使用什么样的数据结构呢？对面向流的编码方法，环形缓存可能是一个理想的数据结构，因为到达的数据可以插进缓存尾部，同时应用可以从缓存首部提取数据。而且，对面向采样的编码来说，此种结构使缓存位置和时间之间的转换也变得简单，因为缓存的位置与其对应的时间是一个线性函数。因此，环形缓存适合于隐式时间管理。当然，为了能适应抖动，在缓存填充到最小门限值之前，不能开始回放。

环形缓存也有一些缺点。首先是增加了开销，每当有分组到达，数据必须复制到缓存中，而当数据交付给应用时，又必须再一次复制到缓存中。此外，对许多面向帧的编码方式，环形缓存并不是很适合。面向帧的编码往往要在数据前增加一个首部，如果将这个数据首部与数据分开存放，环形缓存的许多优点就没有了。但如果将数据首部也存放在环形缓存中，要找到对应于某个给定时间的数据就要困难一些了。因此，许多实现保存一个分组表而不用环形缓存。

围绕着缓存管理还有很多复杂问题，这些问题的起因是 RTP 允许发送方动态地改变编码方式或分组长度。例如，如果语音不是很清楚，发送方可以切换到另一种编码方式，使每个采样的比特数更高。如果发生了这种切换，接收方必须改变缓存的长度。如果发送方和接收方计算机的时钟不能精确地以相同速率工作（即时钟漂移了），接收方也必须改变缓存。如果接收方的时钟比发送方的时钟稍快，时钟漂移的最终结果是使到达的分组看上去好像过期了，即该分组应播放的那个时刻好像已经过去了。

尽管改变编码方式、时钟漂移等现象很少发生，但它们使通用库的代码复杂了。为了处理这些问题，接收方必须准备重新缓存（rebuffer）数据，即接收方必须重新计算本地时钟与 TIMESTAMP 值间的对应关系，还必须调整回放缓存指针以反映这种新的对应关系。重新缓存数据可能会丢弃缓存中的某些数据，阻塞正在接收数据的应用直到有数据到达为止，或者不改变缓存中的数据，只对回放指针做很小的改动。

28.18 事件处理

除了数据传送功能外，实时软件必须处理一系列异步发生的事件。例如，在任何时刻都有可能有新听众加入到某个会话中，或者原来的参与者离开会话。发送方可能改变编码方式。更重要的是，并不是所有的应用都需要知道所有的事件。

异步事件的一种可能实现是采用所谓的上调 (upcall) 机制，即每个应用线程提供一个供实时软件在遇到各个事件时调用的函数。另一种实现采用事件队列 (event queue)，为了使用事件队列机制，应用程序或同步层软件要动态地创建一个队列，并指定其希望接收的一系列事件。当 RTP 层有类型为 T 的事件发生时，软件便检查各个事件队列，并将该事件的副本排列进那些请求事件类型 T 的队列中。

事件队列实现不像上调实现那样危险，因为实时软件并不直接执行应用程序代码。然而，问题还是有可能发生的，具体来说，如果应用程序创建了事件队列但却从来不去提取事件，系统迟早会耗尽内存。如果许多线程都请求某个给定的事件类型，在处理过程中就会出现时延，因为下层软件必须把事件的副本放置到每个队列中。

28.19 回放异常及时间戳的复杂性

还有其他一些细节使实现设计困难，并增加了最终软件的复杂性。传送数据的长度、下层缓存的长度以及设备行为之间的交互作用可能产生不可预计的异常：每次数据传送过程中，增加写到设备上的数据的数量可能会使回放产生间隙。为了理解此中原因，我们可以考虑以下情况，某个音频设备可以缓存 16 000 个 PCM 音频样本（即 2 秒钟声音），如果应用程序写入了一个包含 12 000 个样本的数据块，驱动程序将数据副本到设备的缓存中，并在设备播放时，让应用程序重新执行。这时，输出会连续平滑进行下去，因为在驱动程序接受了 N 块数据后，应用程序有一秒半的时间计算块 N+1。

然而，如果应用企图写入 16 001 个样本，驱动程序将前面 16 000 个数据副本到缓存中，接着将线程阻塞，并开始回放。当缓存接近空了的时候（例如，还剩不到一打数据的时候），设备产生中断。在中断期间，驱动程序把最后一个样本副本到缓存中，重新启动应用线程，使设备继续回放。但是，此时应用程序只有很少的时间（小于播放 12 个样本的时间）来计算下一块数据。如果线程在此花费的时间比回放要长，设备播放完数据后就没声音了，这样，输出就产生了间隙。

时间戳的比较尤其麻烦——一个特别长的 RTP 流可能使时间戳值出现重叠。例如，采样率为 90kHz 的 MPEG 编码可能使 32 比特的时钟在 13.27 小时内出现重叠。当分组到达时，为了检验该分组是否有效，接收方必须确定时间戳值与当前时间是否足够接近。这种检查并不简单，因为，接收方不仅要将发送方媒体时间转换为本地时间，还必须要处理时钟重叠问题。为了简化计算，一般程序都假设，在任意时刻，所有要处理的时间戳都在各自地址空间的一半以内（即 2^{31} 以内）。

28.20 实时库例子的大小

如果一个实时软件库既能处理帧式样编码，也能处理采样式样编码，这样库要有多大呢？为了回答这个问题，我们设计了一个库，这个库支持异步事件处理，也支持流同步，还包括有关发送方和接收方 RTCP 报文的代码，并允许进行重新缓存。由于我们的目的是比较库与应用集成实现两种

方法，我们在两个方面限制库的功能。首先，库只支持隐式时间管理；其 API 和数据结构都不包含显式管理。其次，库代码中并不包含 MPEG 等特定的编码。库的源代码可以在线提供，地址如下：

<ftp://ftp.cs.purdue.edu/pub/comer/rplib.tar>

这个库的最终实现大约在 6000 行以上，不包括 160 行的 makefile 文件及应用程序。图 28.10 给出了该库主要部分的大约行数。

行	百分数	描述
1293	20.8	RTP
1063	17.1	RTCP
990	15.9	流同步
969	15.6	杂项实用例程
893	14.3	头文件说明和常量
578	9.3	帧样式处理
253	4.1	采样式样编码
189	3.0	分组读
6228		总数

图 28.10 实时库例子代码的主要部分以及各部分的大小。百分比经过四舍五入

28.21 MP3 播放器的例子

一种称做 MP3^① 的音频格式在因特网上很是流行。为了对我们的 RTP 库进行实验，我们从以下地址获得了一个 MP3 播放器的实现：

<http://www.mp3-tech.org/programmer/sources/dist10.tgz>

我们将其加入到我们所实现的库中。其结果是一个可以接受并播放 MP3 音频流的应用程序。有趣的是，MP3 解码器有 12 144 行代码，大约占全部代码量的三分之二。除了程序本身外，MP3 解码还依赖于一些动态装载的表里的值。我们在计算源代码时，加入了这些表。

28.22 小结

发送音频和视频数据是一种流式服务，它要使用其他的协议支持才能进行有意义的实时数据回放。这种协议要发送分组序号信息以便接收方能检测出到达的分组是否失序，还要发送时间戳信息以便接收方重新组装数据，从而重新播放。

因特网中用于流式服务的主要协议是实时协议 RTP (Real Time Protocol)。除了支持实时数据外，RTP 还伴随有另一个协议——RTCP，用于控制和获得关于 RTP 会话的信息。RTP 允许任意数据编码，允许单个会话发送多个、独立的流。当回放来自同一源的两个流时，可以对两个流进行同步。同步对那些需要发送视频及伴音的应用是非常重要的，因为这两类信号是在不同流中发送的。

支持实时软件的通用库必须处理所有的情况和使用方式。库必须既允许采样式样的编码，也允

^① MP3 是 MPEG layer 3 (MPEG 层 3) 的简写；MPEG 本身是 Motion Picture Experts Group (动态图像专家组) 标准的缩写。

许帧式样的编码，还要允许独立的或同步的流。此外，这种库还必须能被多个应用线程协同访问。

深入研究

RTP 和 RTCP 由 Schulzrinne 等 [RFC 1889] 制定，该文献讨论了各种决策的动机。关于 MPEG 的信息可在以下地址找到：

<http://drogo.cselt.stet.it/mpeg/>

习题

- 28.1 RTP 是运输层协议吗？从各方面讨论此问题。
- 28.2 进一步阅读 G.711 中用于语音电话系统的脉码调制的形式。ULAW 和 ALAW 格式中的值是如何确定的？
- 28.3 阅读 RTP 标准，说明两个流之间是如何同步的。编写程序，在以下时间间进行转换：某个流的媒体时钟时间、另一个流的媒体时钟时间、用网络时间协议（NTP）编码的绝对时间。
- 28.4 阅读 RTCP 协议规范。协议是如何避免使接收方报告报文过分占用带宽的？
- 28.5 查看 MPEG-2 标准。其编码是面向采样的还是面向帧的？
- 28.6 加密属于哪一层（在 RTP 之上还是之下）？为什么？

第 29 章 流式音频和视频传输（RTP 实现示例）

29.1 引言

前一章介绍了流式的概念，讨论了用于支持交付实时数据的协议，还探讨了 RTP 库的组织及其大小等问题。本章将继续讨论有关问题，采用的方法是将通用的、实时支持库的复杂性与流式应用例子的复杂性进行比较。

29.2 集成实现

为了比较前一章所描述的通用库与集成代码两者的复杂性，我们构造了一个最小要求的集成实现。我们的程序由一个音频播放器构成，该播放器用采样式样编码 μ 律 PCM，这可以为许多 Linux 系统上的硬件所支持。应用程序拥有必需的内置 RTP 和 RTCP 支持，但只包含基本功能，只要不是一个会话、一个音频流、一种编码所必需的，我们就全部忽略。在总体介绍程序结构后，我们再考察程序的代码。

29.3 程序结构

尽管例子程序很小，但其 RTP 实现是符合标准的。具体来说就是，除了接受和处理 RTP 数据分组的代码外，我们的例子程序还包括发送和接收 RTCP 最小报文集的程序。我们会看到，这个程序实现所包括的细节数量令人惊讶。为了尽量减少复杂性，程序围绕四个线程构成，如图 29.1 所示。

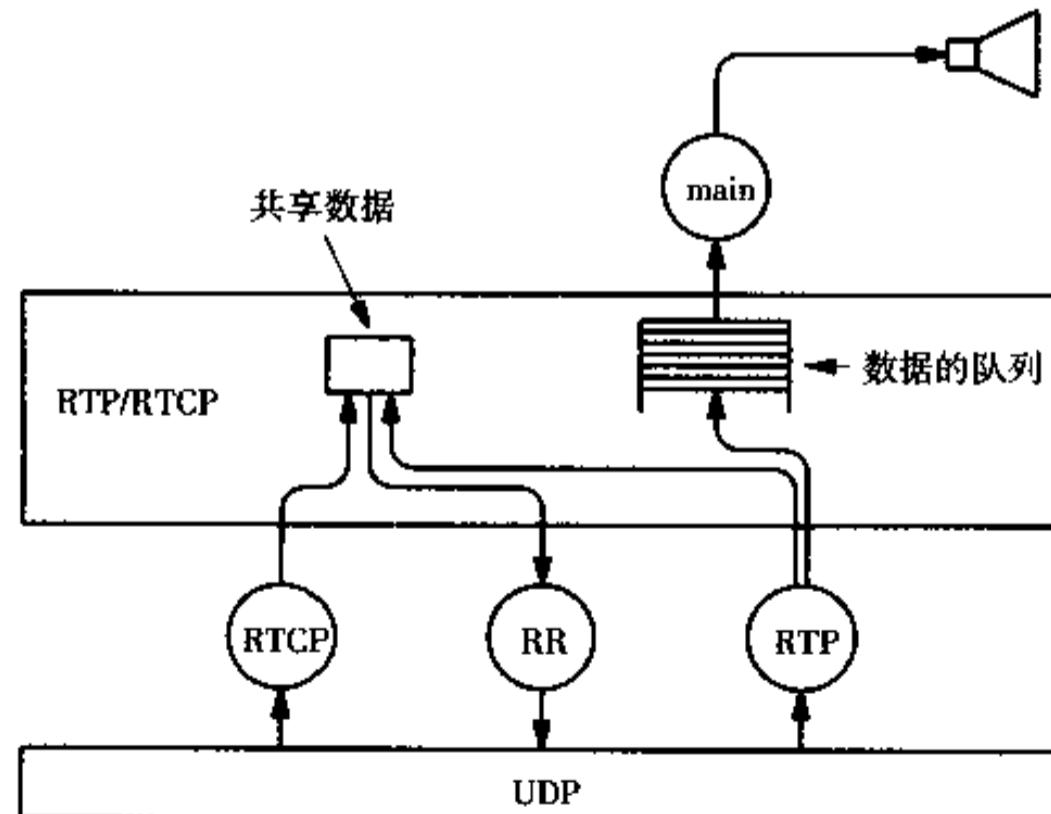


图 29.1 本章所描述的流式音频播放器的线程结构

除了用于提取和播放音频数据的主线程外，该程序还包括三个支持线程。一个线程处理传入的 RTCP 分组，另一个处理传递 RTCP 报文，第三个线程处理 RTP 分组的接收。

29.4 RTP 定义

我们这个 RTP 代码可以大致分成三类：处理传入 RTP 数据分组的代码、处理接收和传输 RTCP 报文的代码以及实用工具的代码。在下面所列的文件 rtp.h 中，包含有与 RTP 相关的常量和数据结构定义。除了这些常量和定义 RTP 分组格式^①的 rtp 结构外，该文件还包括结构 stream 和 session，它们包含有正在接收的会话的有关信息以及与该会话相联的流的信息。stream 结构中有两个字段用于连接传入分组表，stm-qhead 字段指向队列中的第一个分组，stm-qtail 指向队列中的最后一个分组，第三个字段 stm-qmutex 包含一个互斥机制^②用以保证任何时候只有一个线程访问队列。最后，结构 rtpln 用于描述链接在一起的 RTP 分组表中的一项。

```
/* rtp.h - rtptsigt, rtptsmax, rtphdrflen, rtpdata */

#include <common.h>
#include <util.h>

typedef unsigned int ssrc_t;
typedef unsigned int mediatime_t;
typedef unsigned short seq_t;

#define RTP_TSWINDOW      (1 << 31)
#define RTP_CURRVERS     2      /* RTP current version          */
#define RTP_MINHDRLEN    12     /* RTP minimum header length   */
#define RTP_MINSEQUENTIAL 2     /* sequential packets required */
#define RTP_JITTHRESH    8000   /* jitter buffer 'K' (in 1/8000 sec) */
#define RTP_BPBUFSZ       2048   /* buffer pool buffer size    */
#define RTP_BPBUFCNT      64     /* buffers in buffer pool     */
#define RTP_LEEWAY         400    /* playback leeway (in 1/8000 sec) */
#define RTP_INACTTHRESH   5      /* number of cycles until timeout */
#define PCMMU_ID           0     /* RTP ID for PCM mu-law      */
#define PCMMU_CLKRT        8000   /* PCM clockrate               */
#define PCMMU_BW            64000  /* PCM bandwidth (bits/sec)    */
#define rtptsigt (x, y)    (y>x? (x-y<RTP_TSWINDOW? TRUE: FALSE):
                           (y-x<RTP_TSWINDOW? FALSE: TRUE ))
#define rtptsmax(x, y)    (rtptsigt(x, y) ? x : y)
#define rtphdrflen(prtp)  (RTP_MINHDRLEN + (prtp->rtp_cc * 4))
#define rtpdata(prtp)     ((char *) prtp + rtphdrflen(prtp))
```

① 分组中各比特字段的定义以小数在前的体系结构给出。

② 如 Linux 这种遵循 POSIX 标准的系统，对 semaphore 和 mutex 具有分别的抽象。前者可以有一个计数，而后者只能用于互斥。

```

    *
struct rtp{
    unsigned int rtp_cc:4;          /* source count */          */
    unsigned int rtp_ext:1;         /* extension flag */        */
    unsigned int rtp_pad:1;         /* padding flag */         */
    unsigned int rtp_ver:2;         /* version */             */
    unsigned int rtp_payload:7;     /* payload type */        */
    unsigned int rtp_mark:1;        /* marker flag */         */
    seq_t       rtp_seq;           /* sequence number */      */
    mediatime_t rtp_time;          /* timestamp */           */
    ssrc_t      rtp_ssrc;          /* synchronization source identifier */
    char        rtp_data[1];        /* beginning of RTP data */ */
};

struct rtpln {
    int          rln_len;          /* total length of packet */ */
    unsigned int rln_seq;          /* extended sequence number */ */
    struct rtpln *rln_next;        /* pkt with next lower sequence number */
    struct rtpln *rln_prev;        /* pkt with next greater seq number */
    struct rtp   rln_rtp;          /* RTP packet */          */
};

struct stream {
    ssrc_t      stm_ssrc;          /* synchronization source identifier */
    struct rtpln *stm_qhead;        /* pointer to pkt with lowest seq num */
    struct rtpln *stm_qtail;        /* pointer to pkt with highest seq num */
    pthread_mutex_t stm_qmutex;    /* mutex to lock queue */
    pthread_cond_t  stm_rcond;      /* cond var for blocking read */
    pthread_mutex_t stm_rmutex;    /* mutex associated read cond var */
    pthread_mutex_t stm_smutex;    /* mutex for locking stream structure */
    bool         stm_buffering;    /* TRUE when stream is still buffering */
    struct timeval stm_clkx;       /* local time stamp */
    mediatime_t  stm_clky;         /* equivalent media time stamp */
    double       stm_jitter;        /* jitter measure */
    int          stm_inactive;      /* number of inactive cycles or status */
    int          stm_packets;       /* cumulative packets received */
    int          stm_probation;     /* sequential pkts to validate stream */
    seq_t        stm_firstseq;      /* first sequence number received */
    seq_t        stm_hiseq;         /* greatest seq number recently recv'd */
    int          stm_badseq;        /* bad sequence number */
    int          stm_roll;          /* sequence space ' ' roll-overs' */
    int          stm_recprior;      /* packets recv'd in last RTCP cycle */
    int          stm_expprior;      /* packets expected in last RTCP cycle */
    mediatime_t  stm_lastts;        /* last ts for jitter computation */
    struct timeval stm_lastrec;    /* time last pkt rcvd for jitter comp */
    unsigned int  stm_ntp[2];        /* NTP timestamp of last SR (frac, int */
};

```

```

    struct timeval lastsr;      /* local time last SR received */ *
};

struct session {
    unsigned int sn_ssrc;        /* local SSRC */ *
    int         sn_rtpfd;        /* file descriptor for RTP */ *
    int         sn_rtcpfd;       /* file descriptor for RTCP */ *
    struct sockaddr_in sn_rtcp_to; /* destination for RTCP packets */ *
    struct stream sn_stream;     /* single stream being received */ *
    bool        sn_detected;     /* TRUE once stream detected */ *
    int         sn_avgrtcp;      /* RTCP len for interval computation */ *
};

void          rtpinit(unsigned int, int);
void          rtprecv(void);
int           rtpupdate(struct rtp *);
void          rtpinitseq(seq_t);
int           rtpupdateseq(seq_t);
void          rtpnewdata(void);
bool          rtpqinsert(struct rtqln *);
struct rtqln *rtpqdequeue(void);
void          playaudio(void);

extern struct session sn;
extern struct stream stm;

```

文件 rtp.h 包含两个与 RTP 有关的文件 common.h 和 util.h。文件 common.h 含有标准 Linux 文件的一系列 include 以及有关常量和类型定义 typedef。

```

/* common.h - min, max */

#ifndef COMMON_H
#define COMMON_H
#include <arpa/inet.h>
#include <fcntl.h>
#include <linux/soundcard.h>
#include <math.h>
#include <netinet/in.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <strings.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/types.h>

```

```
#include <time.h>
#include <unistd.h>

typedef char          bool;
#define TRUE           1
#define FALSE          0
#define OK             0
#define ERROR          -1
#define min(a, b)      (a < b ? a : b)
#define max(a, b)      (a > b ? a : b)
#endif
```

29.5 时间值的处理

文件util.h包含了缓存池函数以及转换时间值的宏的声明。这个宏名为tv2lin，之所以需要它是因为Linux用两个值存储时间，这两个值分别代表秒和毫秒，而在RTP及很多计算场合看来，都把时间作为单个值。

宏tv2lin有两个参数：Linux时间值和时钟速率（以Hz为单位）。该宏返回一个值，该时间值为与秒参数相等的时钟率（clock rate）。因此，这样调用宏后，

```
tv2lin(tim, c)
```

会产生单个时间值，它是将tim结构中的时间值转换为时钟率为c的时间值。

```
/* util.h - tv2lin */

#include <common.h>
#define tv2lin(t, clkrt) ((t.tv_sec * clkrt) + ((int) ((double) \
                           t.tv_usec * .000001 * (double) clkrt)))

struct bufpool {
    char     *bp_next;                      /* pointer to next free buffer */
    sem_t    bp_sem;                        /* counting semaphore */
    pthread_mutex_t bp_mutex;               /* lock for list manipulation */
};

void        bpinit (void);
char       *bpget (void);
void       bpfree (void *);

extern struct bufpool  bp;
```

29.6 RTP 序列空间的处理

RTP 必须检查每个传入分组的序列号。当流首次开始时，RTP 强制执行一个检查期（probation period），在此期间内，必须按序到达的分组有一个最小限度（RTP_MINSEQUENTIAL）。在例子程序中，过程 rtpupdateseq 使用变量 stm.stm_probation 来对顺序到达的分组计数。如果有分组失序，则此计数器清零，检查期重新开始。

在通过了初始的检查期后，RTP 接收起数据来就宽松得多了，只会拒绝那些距期望序列超前或落后太多的分组。但是，由于序列号空间的限制，比较两个序列号还是复杂的——因为代码要处理序列号回绕这种情况。如果发现序列号回绕，RTP 要检查确定传入分组中的序列号距期望值是否在一个固定的距离之内（本程序中，此距离由常量 RTP_MAXMISORDER 定义）。文件 rtpseq.c 包含了有关的代码。

```
/* rtpseq.c - rtpinitseq, rtpupdateseq */

#include <rtp.h>

#define RTP_MAXDROPOUT          3000
#define RTP_MAXMISORDER          100
#define RTP_SEQMOD                (1 << 16)
/* -----
 * rtpinitseq - init vars related to seq numbers (adapted from RFC 1889)
 * -----
 */
void
rtpinitseq(seq_t seq)
{
    stm.stm_firstseq = seq;
    stm.stm_hiseq = seq;
    stm.stm_roll = 0;
    stm.stm_packets = 0;
    stm.stm_recprior = 0;
    stm.stm_expprior = 0;
}

/* -----
 * rtpupdateseq - check sequence number of packet, update counters
 *                 (adapted from RFC 1889)
 * -----
 */
int
rtpupdateseq (seq_t seq)
{
    seq_t udelta = seq - stm.stm_hiseq;
```

```
if (stm.stm_probation) {
    if (seq == stm.stm_hiseq + 1) {
        stm.stm_probation--;
        stm.stm_hiseq = seq;
        if (stm.stm_probation == 0) {
            rtpinitseq (seq);
            return OK;
        }
    } else {
        stm.stm_probation = RTP_MINSEQUENTIAL - 1;
        stm.stm_hiseq = seq;
    }
    return ERROR;
}
else if (udelta < RTP_MAXDROPOUT) {
    if (seq < stm.stm_hiseq)
        stm.stm_roll++;
    stm.stm_hiseq = seq;
}
else if (udelta <= RTP_SEQMOD - RTP_MAXMISORDER) {
    if (seq == stm.stm_badseq)
        rtpinitseq (seq);
} else {
    stm.stm_badseq = (seq + 1) & (RTP_SEQMOD - 1);
    return ERROR;
}
return OK;
}
```

29.7 RTP分组队列的处理

在我们的RTP程序中，核心的数据结构是一个按序号排序的分组队列。尽管操纵队列很简单，但因为多个线程可能同时操纵队列，这使我们的程序复杂化了。为了保证线程间不互相干扰，此队列由互斥量`stm.stm_qmutex`所保护。在使用队列前，线程要调用函数`pthread_mutex_lock`，该函数能锁住互斥量并防止其他线程获得锁，因此，`pthread_mutex_lock`保证了每次只有一个线程访问队列，而其他试图同时访问队列的线程被阻塞了。线程在访问完队列后，要调用`pthread_mutex_unlock`。如果没有其他线程被阻塞在那里等待访问队列，该调用仅仅将互斥量重新设置回未锁状态，如果有线程被阻塞，该调用会使某一个线程脱离阻塞状态，得以继续进行下去。

从队列中删除分组比较简单，因为分组总是从队列头删除出去。在互斥上锁后，将队列头指向下一分组就可以把第一个分组从队列中摘除。但插入比删除就稍有点复杂了，因为分组可能失序，因此，要查找队列表，找出应把该项数据插入到哪个位置。在插入开始前，互斥量被锁住，在插入结束后，互斥量又被解锁。文件`rtpqueue.c`包含了有关的代码。

```
/* rtpqueue.c - rtpqdequeue, rtpqinsert */

#include <rtp.h>

/*
 * rtpqdequeue - remove the oldest packet from the RTP packet queue
 */
struct rtpln *
rtpqdequeue (void)
{
    struct rtpln *p, *pln;

    (void) pthread_mutex_lock (&stm.stm_qmutex);
    if ((pln = stm.stm_qhead) != NULL) {
        if ((p = pln->rln_prev) != NULL) {
            stm.stm_qhead = p;
            p->rln_next = NULL;
        } else
            stm.stm_qtail = stm.stm_qhead = NULL;
    }
    (void) pthread_mutex_unlock (&stm.stm_qmutex);
    return pln;
}

/*
 * rtpqinsert - insert a packet into the RTP packet queue
 */
bool
rtpqinsert(struct rtpln *pln)
{
    struct rtpln     *p, *q = NULL;
    bool           head = FALSE;

    (void) pthread_mutex_lock (&stm.stm_qmutex);
    for (p = stm.stm_qtail; p != NULL && p->rln_seq > pln->rln_seq;
         q = p, p = p->rln_next);
    if (q != NULL)
        q->rln_next = pln;
    else
        stm.stm_qtail = pln;
    pln->rln_prev = q;
    if (p != NULL)
```

```
    p->rln_prev = pln;
else
    stm.stm_qhead = pln;
pln->rln_next = p;
head = pln == stm.stm_qhead;
(void) pthread_mutex_unlock (&stm.stm_qmutex);
return head;
}
```

29.8 RTP 输入处理

本例子程序使用一个单独的线程来接收和处理分组，该线程执行 `rtprecv` 过程。`rtprecv` 过程有一个无限循环，在循环中通过反复调用过程 `recv` 来接收和处理分组。每次循环，`rtprecv` 从套接字里提取一个分组，验证首部字段后，将分组排列到队列中。当流开始时，`rtprecv` 创建一个线程以便接收 RTCP 报告。此外，`rtprecv` 还检查同步源冲突（collision），同步源冲突是指传入分组中的同步源标识符与本地使用的标识符一样，每当发生这种情况，`rtprecv` 就随机选择一个新的源标识符。

```
/* rtprecv.c - rtprecv */

#include <rtcp.h>

/*
 * rtprecv - receive and process an RTP packet
 */
void
rtprecv ( )
{
    struct rtpln      *pln;
    struct rtp        *prtp;
    pthread_t         thrid;

    while (TRUE) {
        pln = (struct rtpln *) bpget();
        prtp = &pln->rln_rtp;
        pln->rln_len = recv(sn.sn_rtpfd, prtp, RTP_BPBUFFSZ -
            sizeof(struct rtpln), 0);
        if (pln->rln_len < RTP_MINHDRLEN || prtp->rtp_ver !=
            RTP_CURRVERS) {
            bpfree(pln);
            continue;
        }
        prtp->rtp_seq = ntohs (prtp->rtp_seq);
        prtp->rtp_time = ntohl (prtp->rtp_time);
```

```

        prtp->rtp_ssrc = ntohs (prtp->rtp_ssrc);
        if (sn.sn_ssrc == prtp->rtp_ssrc) {
            rtcpsendbye();
            sn.sn_detected = TRUE;
            bpfree(pln);
            continue;
        }
        if (!sn.sn_detected && prtp->rtp_payload == PCMMU_ID) {
            stm.stm_ssrc = prtp->rtp_ssrc;
            sn.sn_detected = TRUE;
            (void) pthread_create(&thrid, NULL,
                (void * (*) (void *)) rtcpcycle, NULL);
        }
        if (prtp->rtp_ssrc != stm.stm_ssrc || rtpupdate(prtp) ==
            ERROR) {
            bpfree (pln);
            continue;
        }
        pln->rln_seq = ((stm.stm_roll) << 16) | prtp->rtp_seq;
        if (rtpqinsert(pln) || stm.stm_buffering)
            rtpnewdata();
    }
}

```

如果流还正在缓存积累过程中，或者到达的分组是下一个要播放的分组，RTP必须重新计算抖动时延（即必须确定缓存中的最早分组与新分组时间戳值的差是否超过了抖动门限）。第二个条件容易检测——如果传入的分组被排列进表头，就说明时间戳差值超过了抖动门限。而且，只有当新分组到达时，抖动缓存的有效长度才会改变。因此，只要新分组被排列进表头，`rtprecv`便调用`rtpnewdata`，该过程用于处理这两种可能情况。

为了控制抖动缓存的长度，`rtpnewdata`将比较新插入的分组（在队列头）和最早分组（在队列尾）的时间戳的值，如果比较的结果比常量`RTP_JITTHRESH`大，`rtpnewdata`便把`stm.stm_buffering`设置为`FALSE`，表明不再进行缓存了。最后，`rtpnewdata`调用`pthread_cond_signal`来唤醒等待着的应用线程。如果应用线程正因等待数据而被阻塞，该调用便会唤醒应用线程，如果应用已经在执行，该调用不产生效果。

```

/* rtpnewdata.c - rtpnewdata */

#include <rtp.h>

/*
 * rtpnewdata - determine if buffering is complete and signal read cond
 */
void
rtpnewdata (void)

```

```
{  
    mediatime_t begin = 0, end = 0;  
  
    (void) pthread_mutex_lock (&stm.stm_rmutex);  
    if (stm.stm_buffering) {  
        (void) pthread_mutex_lock(&stm.stm_qmutex);  
        if (stm.stm_qhead != NULL) {  
            begin = stm.stm_qhead->rln_rtp.rtp_time;  
            end = stm.stm_qtail->rln_rtp.rtp_time;  
        }  
        (void) pthread_mutex_unlock(&stm.stm_qmutex);  
        if (end - begin >= RTP_JITTHRESH) {  
            stm.stm_clky = begin;  
            (void) gettimeofday (&stm.stm_clkx, NULL);  
            stm.stm_buffering = FALSE;  
        }  
    }  
    (void) pthread_cond_signal (&stm.stm_rcond);  
    (void) pthread_mutex_unlock (&stm.stm_rmutex);  
}
```

29.9 为 RTCP 保存统计信息

RTCP 要周期性地产生接收方报告。为此，它需要有关最新刚刚收到的分组以及抖动的信息。每次有 RTP 分组到达，都要调用过程 rtpupdate，该过程根据协议标准的有关规约，更新接收方报告所需要的信息。下面是 rtpupdate.c 的代码。

```
/* rtpupdate.c - rtpupdate */  
  
# include <rtp.h>  
  
/*-----  
 * rtpupdate - update statistics with each RTP packet received  
 *-----*/  
  
int  
rtpupdate(struct rtp *header)  
{  
    struct timeval now, deltatv;  
    int delta, D;  
  
    (void) pthread_mutex_lock (&stm.stm_smutex);  
    if (rtpupdateseq(header->rtp_seq) == ERROR) {  
        (void) pthread_mutex_unlock (&stm.stm_smutex);  
        return ERROR;  
    }
```

```

    }

    (void) gettimeofday(&now, NULL);
    if (stm.stm_packets++ != 0) {
        timersub(&now, &stm.stm_lastrec, &deltatv);
        delta = tv2lin(deltatv, PCMMU_CLKRT);
        D = delta - (header->rtp_time - stm.stm_lastts);
        D = (D < 0 ? -D : D);
        stm.stm_jitter += ((double) D - stm.stm_jitter) / 16.0;
    }
    stm.stm_inactive = 0;
    stm.stm_lastts = header->rtp_time;
    stm.stm_lastrec = now;
    (void) pthread_mutex_unlock (&stm.stm_smutex);
    return OK;
}

```

29.10 RTP 初始化

我们已经了解了处理传入 RTP 分组的有关过程，现在，我们可以考察一下初始化方面的程序。文件 rtpinit.c 含有初始化代码，它要初始化线程、套接字以及所有的数据结构。

Rtpinit 开始时先把保存会话信息的数据结构清零，接着便创建了一个分组缓存池以及线程所使用的必要的条件变量和互斥量。接着创建了 RTP 和 RTCP 套接字并将其绑定到合适的（组播）地址上。最后，在创建完套接字和所有的互斥量后，rtpinit 启动了第二个和第三个线程，它们一个用于处理传入的 RTP 分组，另一个用于处理传入的 RTCP 分组。值得再次提及的是，用于发送 RTCP 接收方报告的第四个线程，是由 rtrecv 在建立会话时所创建的。

```

/* rtpinit.c - rtpinit */

#include <rtcp.h>

struct session  sn;
struct stream   stm;

/*
 * rtpinit - initialize RTP session and stream structs, join mcast group
 */
void
rtpinit(unsigned int mca, int port)
{
    struct ip_mreq      mreq;
    int                  reuse = TRUE;
    unsigned char        ttl = RTCP_TTL;
    pthread_t            th;

```

```

(void) memset(&sn, 0, sizeof(struct session));
(void) bpinit();
sn.sn_ssrc = random();
sn.sn_rtpfd = socket(PF_INET, SOCK_DGRAM, 0);
sn.sn_rtcpfd = socket(PF_INET, SOCK_DGRAM, 0);
(void) setsockopt(sn.sn_rtpfd, SOL_SOCKET, SO_REUSEADDR, &reuse,
                  sizeof(reuse));
(void) setsockopt(sn.sn_rtcpfd, SOL_SOCKET, SO_REUSEADDR, &reuse,
                  sizeof(reuse));
sn.sn_rtcpto.sin_family = AF_INET;
sn.sn_rtcpto.sin_addr.s_addr = mca;
sn.sn_rtcpto.sin_port = htons(port);
(void) bind(sn.sn_rtpfd, (struct sockaddr *) &sn.sn_rtcpto,
            sizeof(struct sockaddr_in));
sn.sn_rtcpto.sin_port = htons(port + 1);
(void) bind(sn.sn_rtcpfd, (struct sockaddr *) &sn.sn_rtcpto,
            sizeof(struct sockaddr_in));
mreq.imr_multiaddr.s_addr = mca;
mreq.imr_interface.s_addr = htonl(INADDR_ANY);
(void) setsockopt(sn.sn_rtpfd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                  &mreq, sizeof(mreq));
(void) setsockopt(sn.sn_rtcpfd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                  &mreq, sizeof(mreq));
(void) setsockopt(sn.sn_rtcpfd, IPPROTO_IP, IP_MULTICAST_TTL,
                  &ttl, sizeof(ttl));
stm.stm_buffering = TRUE;
stm.stm_probation = RTP_MINSEQUENTIAL;
(void) pthread_mutex_init(&stm.stm_qmutex, NULL);
(void) pthread_mutex_init(&stm.stm_smutex, NULL);
(void) pthread_mutex_init(&stm.stm_rmutex, NULL);
(void) pthread_cond_init(&stm.stm_rcond, NULL);
(void) pthread_create(&th, NULL, (void* (*) (void*))rtprecv, NULL);
(void) pthread_create(&th, NULL, (void* (*) (void*))rtcprecv, NULL);
}

```

主程序提供最后一部分代码。它调用 rtpinit 以初始化 RTP 代码，接着调用 playaudio 播放音频数据。在我们的实现中，playaudio 不会返回——会话结束时，rtpcycle 会调用 exit，从而进程终止。然而，playaudio 后面还是插入了一个 return 语句，这样可以保证程序通过 lint 程序的检查。文件 main.c 包含了有关代码。

```

/* main.c - main */

#include <rtcp.h>

```

```

/*
 * main - initialize and begin to play audio
 */
int
main(int argc, char **argv)
{
    rtpinit(inet_addr(argv[1]), atoi(argv[2]));
    playaudio();
    return 0;
}

```

过程 playaudio.c 从 RTP 读取数据，然后播放对应于“当前”时间的数据。为此，程序需要在本地时钟时间和流的 RTP 媒体时钟所使用的时间之间做出映射，这种映射是由 rtpnewdata 来建立的——当抖动缓存达到门限值时，rtpnewdata 把当前本地时间记录在 stm.stm_clkx 中，把从第一个分组那里得到的媒体时间记录在 stm.stm_clky 中。Playaudio 利用这两个值将当前时钟时间变换为 RTP 分组中的媒体时钟值。这样，当有分组能供播放时，playaudio 就可以确定分组中的数据应在什么时间播放。如果数据对应的时间是过去，playaudio 只是将其跳过，并不进行播放。如果数据对应的时间是将来，playaudio 要计算将来那个应播放的时刻距现在还有多久，然后阻塞在那里延迟播放，一直到应播放的那个时刻。最后，如果 playaudio 发现数据所包含的时间对应于当前时间，playaudio 并不自动开始播放分组中的数据，而是计算出分组数据中刚好对应于当前时刻的那片数据的偏移量，然后从这个位置开始播放。

许多细节使程序复杂化。最单调乏味的细节之一是计算时延。为了理解此中困难，首先要注意到 Linux 是一个分时系统，而不是一个实时操作系统。任何时候，多个线程之间要竞争 CPU，甚至在发生事件后，Linux 也可能要过一小段时间才能将 CPU 交给等待该事件的线程。因此，当有线程请求 N 毫秒时延，操作系统能保证提供一个至少 N 毫秒的时延，但不能保证时延刚好 N 毫秒。而且，即使线程已被调度选中，在真正开始播放前，计算和 I/O 等指令也会进一步引入时延。

对大多数计算来说，稍微有点时延是不要紧的，但对实时音频来说，递交给音频设备的数据发生时延会造成播放的暂停——这样，音频信号的各种人工处理痕迹就会被用户注意到。为了避免这个问题，playaudio 采用了一种具有启发性的方法：在真正需要开始播放前就把线程唤醒。即在计算时延时，playaudio 使用的值比真正的时延值稍短。这点不同使操作系统或设备所引入的额外开销不致妨碍到按时播放。

RTP 的语义进一步使处理步骤复杂了。具体来说，前面已经提到，RTP 分组的到达可能失序，在这种情况下，我们可以考虑一下如果主线程正在睡眠态，分组到达后会发生什么。例如，假设在时间值为 50 的时候，playaudio 检查了队列，发现队列中的第一个分组应在时间值为 60 的时候播放。主线程需要时延的时间应稍小于 10 个时间单位。但在时延过程中的某一时刻，有新分组到达，其所包含的时间表明，该数据应在时间为当前至 60 之间的某一时刻播放，为了适合这种情况，软件必须允许取消时延。本例子程序使用了条件变量来处理这种情况。主线程在条件变量 stm.stm_rcond 上执行按时间控制的等待 (timed wait)。如果时间超时，由操作系统唤醒线程；如果有分组到达，rtpnewdata 调用 pthread_cond_signal 来唤醒线程。文件 playaudio.c 包含了有关代码。

```
/* playaudio.c - playaudio */
```

```
#include <rtp.h>

/*
 * playaudio - dequeue and play packets
 */
void
playaudio(void)
{
    struct rtp      *prtp;
    struct rtpln   *pln;
    struct timespec waketimets;
    struct timeval  now, deltatv, waketimetv;
    mediatime_t     mnow, readfrom, delay;
    int             audiodev, samples, offset, profile = APP_NETWORK;
    audiodev = open (" /dev/audio", O_WRONLY);
    (void) ioctl(audiodev, SNDCTL_DSP_PROFILE, &profile);
    (void) pthread_mutex_lock (&stm.stm_rmutex);
    while (TRUE) {
        if (stm.stm_buffering || (pIn = rtpqdequeue()) == NULL){
            (void) pthread_cond_wait (&stm.stm_rcond,
                                      &stm.stm_rmutex);
            continue;
        }
        prtp = &pln->rln_rtp;
        (void) gettimeofday(&now, NULL);
        timersub(&now, &stm.stm_clkx, &deltatv);
        mnow = tv2lin(deltatv, PCMMU_CLKRT) +stm.stm_clky;
        samples = pln->rln_len - rtphdrlen(prtp);
        if (rtptsigt(mnow, prtp->rtp_time + samples - 1)) {
            bpfree(pln);
            continue;
        }
        readfrom = rtptsmax (prtp->rtp_time, mnow);
        if (rtptsigt (readfrom, mnow + RTP_LEEWAY) ) {
            (void) rtpqinsert(pln);
            delay = readfrom - mnow - RTP_LEEWAY;
            waketimetv.tv_sec = delay / PCMMU_CLKRT;
            waketimetv.tv_usec = (int) (((delay % PCMMU_CLKRT) /
                                         (double) PCMMU_CLKRT) * 1000000);
            timeradd (&now, &waketimetv, &waketimetv);
            TIMEVAL_TO_TIMESPEC (&waketimetv, &waketimets);
            (void) pthread_cond_timedwait (&stm.stm_rcond,
                                           &stm.stm_rmutex, &waketimets);
            continue;
        }
    }
}
```

```

    }
    offset = readfrom - prtp->rtp_time;
    if (prtp->rtp_mark)
        (void) ioctl(audiodev, SNDCTL_DSP_SYNC);
    (void) write(audiodev, rtpdata(prtp) + offset, samples -
                offset);
    bpfree(pln);
}
}

```

29.11 RTCP 的定义

在每个 RTCP 报文的开始，都有一个指明其类型的首部。在我们的例子程序中，结构 rtcp 定义了报文首部的格式。在报文首部后面，可以有两种类型的报文：一种是发送方报告，其格式由结构 sr 定义；一种是接收方报告，其格式由结构 rr 定义。接收方报告由一系列接收报告块所组成，单个接收报告块的格式由结构 rblock 定义。文件 rtcp.h 包含了有关代码。

```

/* rtcp.h - htonl24 */

#include <rtp.h>

#define RTCP_HEADERLEN      4      /* RTCP header length (octets)          */
#define RTCP_BWFRAC         .05     /* fraction of b/w for RTCP           */
#define RTCP_TTL             16     /* TTL for our RTCP packets           */
#define RTCP_MAXPACKETSZ    1024   /* RTCP receive buffer size           */
#define RTCP_SR              200    /* RTCP value for sender report      */
#define RTCP_RR              201    /* RTCP value for receiver report     */
#define RTCP_BYE             203    /* RTCP value for bye message        */
#define htonl24(v)           (((v&0xff) << 16) | (v&0xff00) |
                                         ((v&0xff0000)>>16))

struct rtcp {
    unsigned int rtcp_count:5; /* object count                      */
    unsigned int rtcp_pad:1;   /* padding present flag               */
    unsigned int rtcp_ver:2;   /* version                           */
    unsigned char rtcp_type;  /* message type                      */
    unsigned short rtcp_length; /* message length / 4 - 1           */
    char         rtcp_data[1]; /* message data                      */
};

struct rblock {
    ssrc_t       rb_ssrc;      /* SSRC to which this rblock refers */
    unsigned int rb_fraclost:8; /* fraction lost since prev report */
    int          rb_cumlost:24; /* cumulative packets lost           */
    unsigned int rb_hiseq;     /* extended highest seq received    */
    unsigned int rb_jitter;    /* jitter measure                   */
};

```

```

    unsigned int rb_lastsrts;      /* middle 32 bits of last SR's NTP */ */
    unsigned int rb_delay;        /* delay since last SR */ */
};

struct rr {
    ssrc_t          rr_ssrc;      /* SSRC of sender */ */
    char            rr_rbblock[1]; /* pointer to first report block */ */
};

struct sr {
    ssrc_t          sr_ssrc;      /* SSRC of sender */ */
    unsigned int    sr_intts;     /* NTP timestamp (high 32 bits) */ */
    unsigned int    sr_fracts;    /* NTP timestamp (low 32 bits) */ */
    unsigned int    sr_rtpts;     /* RTP media timestamp */ */
    unsigned int    sr_packets;   /* sender's sent packet count */ */
    unsigned int    sr_octets;    /* sender's sent octet count */ */
    char            sr_rblock[1]; /* pointer to first report block */ */
};

void    rtcpsendbye(void);
void    rtcpheader(struct rtcp *, int, unsigned char, int);
double rtcpinterval(int, int, double, int, int, int *, int);
void    rtcpcycle(void);
void    rtcprecv(void);

```

29.12 接收 RTCP 发送方的报告

前面已经介绍过，我们的RTCP代码由两个单独的线程所组成：一个用于处理到达RTCP分组的接收，另一个用于处理外发代码的生成。`rtcprecv`过程处理传入的发送方报告报文。与发送方报告有关的两则信息对生成外发的接收方报告很有用，一则信息是报告中的NTP时间戳，另一则信息是报文到达时的本地时间。`rtcprecv`从发送方报告中将时间戳值抽取出来，并将该值和由Linux函数`gettimeofday`所返回的本地时间一起记录到流结构中。`rtcprecv`还利用每个传入分组中的同步源标识符将指定的报文发送方与其他发送方分离出来。文件`rtcprecv.c`包含的代码如下。

```

/* rtcprecv.c - rtcprecv */

#include <rtcp.h>

/*
 * rtcprecv - receive and process RTCP sender reports
 */
void
rtcprecv (void)

```

```

{
    char          rtcpbuf[RTCP_MAXPACKETSZ];
    struct rtcp   *prtcp = (struct rtcp *) rtcpbuf;
    struct sr     *psr = (struct sr *) prtcp->rtcp_data;

    while (TRUE) {
        (void) recv(sn.sn_rtcpfd, prtcp, RTCP_MAXPACKETSZ, 0);
        prtcp->rtcp_length = ntohs (prtcp->rtcp_length);
        if (prtcp->rtcp_ver == RTP_CURRVERS &&
            prtcp->rtcp_type == RTCP_SR &&
            ntohs (psr->sr_ssrc) == stm.stm_ssrc) {
            stm.stm_inactive = 0;
            (void) gettimeofday(&stm.stm_lastsr, NULL);
            stm.stm_ntp[0] = ntohs(psr->sr_fracts);
            stm.stm_ntp[1] = ntohs (psr->sr_intts);
        }
    }
}

```

29.13 产生 RTCP 接收方的报告

第二个 RTCP 线程处理接收方报告的生成。由于这段程序很长，我们把它分成两个文件。过程 `rtpcycle` 提供主循环，当有传入会话建立时便开始循环。在每次循环前，程序要时延 `wait` 秒（下面还要讨论关于 `wait` 的计算）才继续进行。在每次循环中，`rtpcycle` 要测试流是否还处于活动状态。如果自上一次循环以来有数据到达，`rtpcycle` 便发送接收方报告。如果自上一次循环以来没有数据到达，`rtpcycle` 使 `stm.stm_inactive` 加一，该值记录连续没有数据到达的循环次数。如果该值小于门限值 `RTP_INACTTHRESH`，`rtpcycle` 什么也不做。如果该值达到门限值，`rtpcycle` 便调用 `rtpsendbye` 产生一个 RTCP bye（再见）报文并且退出。

分组中字段的长度使 `rtpcycle` 复杂了。例如，RTCP 用一个 24 比特整数表示累计的分组丢失数量，该值是由期望到达的分组数减去实际已经到达的分组数得出的。在每次计算该值时，`rtpcycle` 必须调用 `hton24` 将 32 比特整数转换为 24 比特整数，然而，在转换前，`rtpcycle` 必须保证该值没有超出范围。程序首先要调用 `min` 以保证该值比 24 比特能表示的最大正整数小，接着调用 `max` 以保证比 24 比特能表示的最小负整数大。

当需要接收方的报告时，`rtpcycle` 便填写报告块和报文首部，然后将报文传送出去。在发送报告后，`rtpcycle` 把相关数据结构解锁，重新计算时延时间，继续重复进行循环。文件 `rtpcycle.c` 包含了有关代码。

```

/* rtpcycle.c - rtpcycle */

#include <rtcp.h>

/*
 * rtpcycle - generate periodic RTCP receiver reports
 */

```

```
/*
void
rtcpcycle (void)
{
    int          length = RTCP_HEADERLEN + sizeof(ssrc_t) +
                      sizeof (struct rblock);
    char         buf [length];
    struct rtcp *prtcp = (struct rtcp *) buf;
    struct rr   *prr = (struct rr *) prtcp->rtcp_data;
    struct rblock *prb = (struct rblock *) prr->rr_rblock;
    int          exthiseq, cumexp, cyclost, cycexp;
    struct timeval now, delay;
    double       wait = rtcpinterval(1, 0, PCMMU_BW * RTCP_BWFrac,
                                    FALSE, 0, &sn.sn_avgrtcp, TRUE);

    while (TRUE) {
        (void) usleep((unsigned int) (wait * 1000000));
        (void) pthread_mutex_lock(&stm.stm_smutex);
        if (stm.stm_packets == stm.stm_recprior) {
            if (++stm.stm_inactive == RTP_INACTTHRESH) {
                rtcpsendbye();
                exit(0);
            }
            (void) pthread_mutex_unlock(&stm.stm_smutex);
            wait = rtcpinterval(2, 0, PCMMU_BW * RTCP_BWFrac,
                                FALSE, 0, &sn.sn_avgrtcp, FALSE);
            continue;
        }
        prr->rr_ssrc = htonl(sn.sn_ssrc);
        exthiseq = (stm.stm_roll << 16) | stm.stm_hiseq;
        cumexp = exthiseq - stm.stm_firstseq + 1;
        cycexp = cumexp - stm.stm_expprior;
        cyclost = max(cycexp - (stm.stm_packets -
                                  stm.stm_recprior), 0);
        prb->rb_ssrc = htonl(stm.stm_ssrc);
        prb->rb_fraclost = cycexp == 0 ? 0 : (cyclost << 8) / cycexp;
        prb->rb_cumlost = htonl24 (max (min (cumexp -stm.stm_packets,
                                              0xffff), -(1 << 23)));
        prb->rb_hiseq = htonl(exthiseq);
        prb->rb_jitter = htonl((unsigned int) stm.stm_jitter);
        prb->rb_lastsrts = htonl(((stm.stm_ntp[1] & 0x0000ffff) << 16)
                                 | ((stm.stm_ntp[0] & 0xffff0000) >> 16));
        (void) gettimeofday(&now, NULL);
        timersub(&now, &stm.stm_lastsr, &delay);
        prb->rb_delay = stm.stm_lastsr.tv_sec != 0 ? htonl (
            tv2lin(delay, 65536)) : 0;
    }
}
```

```

        rtcpheader(prtcp, 1, RTCP_RR, length);
        (void) sendto(sn.sn_rtcfd, prtcp, length, 0, (struct
            sockaddr *) &sn.sn_rtcp, sizeof(struct sockaddr_in));
        stm.stm_expprior = cumexp;
        stm. stm_recprior = stm.stm_packets;
        (void) pthread_mutex_unlock (&stm.stm_smutex);
        wait = rtcpinterval (2, 1, PCMMU_BW * RTCP_BWFRAC, FALSE,
            length + 28, &sn.sn_avgrtcp, FALSE);
    }
}

```

29.14 创建 RTCP 的头部

过程 `rtcpheader` 将填写头部中的各个字段。

```

/* rtcpheader.c - rtcpheader */

#include <rtcp.h>

/*
 * rtcpheader - fill in an RTCP header in network byte order
 */
void
rtcpheader(struct rtcp *prtcp, int count, unsigned char type, int length)
{
    prtcp->rtcp_ver = RTP_CURVERS;
    prtcp->rtcp_count = count;
    prtcp->rtcp_type = type;
    prtcp->rtcp_length = htons((length / 4) - 1);
    prtcp->rtcp_pad = FALSE;
}

```

29.15 RTCP 时延的计算

RTCP 产生接收方报告的速率取决于参与者的数量（参与者的数量越多，速率越低）。而且，协议标准规定了一个初始速率，在没有其他参与者的信息时，程序按该速率产生接收方报告。过程 `rtcpinterval` 用于计算到下一报告的时间时延。文件 `rtcpinterval.c` 中的代码取自标准。

```

/* rtcpinterval.c - rtcpinterval */
#include <rtcp.h>
/*
 * rtcpinterval - compute seconds until next RTCP cycle (from RFC 1889)
 */

```

```
/*
double
rtcpinterval (int members, int senders, double rtcp_bw, int we_sent,
               int packet_size, int *avg_rtcp_size, int initial)
{
    double const RTCP_MIN_TIME = 5.;
    double const RTCP_SENDER_BW_FRACTION = 0.25;
    double const RTCP_RECV_BW_FRACTION = (1-RTCP_SENDER_BW_FRACTION);
    double const RTCP_SIZE_GAIN = (1./16.);
    double      t;
    double      rtcp_min_time = RTCP_MIN_TIME;
    int         n;

    if (initial) {
        rtcp_min_time /= 2;
        *avg_rtcp_size = 128;
    }
    n = members;
    if (senders > 0 && senders < members * RTCP_SENDER_BW_FRACTION) {
        if (we_sent) {
            rtcp_bw *= RTCP_SENDER_BW_FRACTION;
            n = senders;
        } else {
            rtcp_bw *= RTCP_RECV_BW_FRACTION;
            n -= senders;
        }
    }
    *avg_rtcp_size += (packet_size - *avg_rtcp_size) * RTCP_SIZE_GAIN;
    t = (*avg_rtcp_size) * n / rtcp_bw;
    if (t < rtcp_min_time)
        t = rtcp_min_time;
    return t * (clrand48() + 0.5);
}
```

29.16 RTCP Bye（再见）报文的产生

当要离开 RTP会话时，参与者必须发送一个 RTCP bye 报文。文件 `rtcpsendbye.c` 包含了有关创建和发送该报文的代码。

```
/* rtcpsendbye.c - rtcpsendbye */
#include <rtcp.h>

/*
 * rtcpsendbye - generate an RTCP bye message
 */
```

```

/*
void
rtcpSendBye (void)
{
    int          length = RTCP_HEADERLEN + sizeof(ssrc_t);
    char         buf [length];
    struct rtcp *prtcp = (struct rtcp *) buf;

    * ((ssrc_t *) prtcp->rtcp_data) = htonl(sn.sn_ssrc);
    rtcpheader(prtcp, 1, RTCP_BYE, length);
    (void) sendto(sn.sn_rtcpf, prtcp, length, 0, (struct sockaddr *)
                  &sn.sn_rtcpto, sizeof (struct sockaddr_in));
}

```

29.17 集成实现的大小

前一章我们考察了通用 RTP 库的大小。该库有 6000 多行代码，与此不同的是，本章所给的集成实现只有 700 行代码，大致划分成如下部分：

行	百分数	描述
279	33.2	RTP
169	20.1	RTCP
56	6.7	缓存池实用例程
197	23.4	头文件说明和常量
140	16.6	主程序和应用代码
841		总数（包括主程序和所有的实用例程）

图 29.2 集成实现的大小。该图表明不考虑一般性可以显著减少代码量

如图所示，集成实现只包含 841 行代码，其中包括 140 行的主程序及播放音频的过程。因此，整个应用，包括集成在里面的 RTP，其程序量的大小与第 28 章所描述的库实现相比，不到库实现的 14%。如果忽略主程序，程序量比库实现的 12% 还要小。

29.18 小结

本章考察了一个流式音频应用的实现，该应用能够接收和播放通过 RTP 发送的音频数据。与前一章所描述的通用库实现一样，本章所描述的程序也使用了多个线程。主线程接收和播放音频数据，其他三个线程分别用于处理传入的 RTP 分组、传入的 RTCP 分组以及外发的 RTCP 分组。与库实现不同的是，本程序的限制是一个会话只有一个音频流，而且只使用一种单一的编码方案。最终的程序大小要比库实现小得多。

深入研究

关于 RTP 和 RTCP 分组格式的详细情况可以在 Schulzrinne 等的文章中找到 [RFC 1889]。该文

章还对代码中所使用的很多算法进行了说明，如该标准给出了初始 RTP 序号的选择、发送接收方报告时的时延的计算等。本章中的部分代码直接取自该 RFC。为了与本书的风格相一致，我们去掉了代码中的一些注释，并且稍微重新编排了一下格式，但内容没有改动。

关于 Linux 音频设备（/dev/audio）以及某种计算机所提供的音频编码等信息，可以从联机文档中找到。最后，关于 RTP Audio Tool（RTP 音频工具）的有关信息可以从下面的网站中得到：

<http://www-micc.cs.ucl.ac.uk/multimedia/projects/rat/index.html>

习题

- 29.1 本章的程序假定 Linux 音频设备已经初始化成播放 PCM（(律编码)）音频数据。修改这个例子程序，使其显式地对音频设备进行初始化。
- 29.2 为了避免在回放时出现间断，例子程序提前唤醒了被时延的线程。进行实验，改变 Linux 的负载以及设备缓存的长度，找出负载、缓存长度以及为避免出现间断而需要的提前唤醒时间三者之间的关系。
- 29.3 使用线程出于两种动机：降低代码的复杂性和通过并发提高性能。讨论这两个动机在例子程序中所起的作用。
- 29.4 仔细解剖例子程序，它是受 CPU 限制还是受 I/O 限制？CPU 的时间主要花费在哪？
- 29.5 例子程序允许通过组播传输 RTP 分组，如果底层由单播传输取代，哪些代码能够省去？
- 29.6 修改代码，使用 TCP 而不是 UDP 传递 RTP 报文，主要的困难是什么？主要的优点又是什么？
- 29.7 编写一个程序，它能从 Linux 音频设备提取数据，并能用 RTP 发送出去。传输或接收需要更多的代码吗？为什么？

第30章 Linux服务器中的实用技巧和技术

30.1 引言

本书给出了一些服务器例子的代码，编写这些例子是为了给我们所讨论的概念提供一个简洁的、指导性的说明，这些概念将有助于读者理解基本设计原理。因此，代码省略了许多实际软件所需的细节。

本章将描述一些技术和常规的实践经验，这是专业程序员在构造实际服务器程序时所要遵循的。这里提到的技术将使程序更为强大，且更容易排错。同时也使服务器与启动它的进程相隔绝，并且允许服务器遵循Linux环境^①中安全操作的规定。虽然服务器正确运行并不需要这些技术，但专业的程序员却认为它们非常有用。

30.2 后台操作

与大多数UNIX系统一样，Linux允许进程在前台（foreground）或后台（background）执行。理解两者区别的最简单方法是，想像一下某个用户将一些命令键入到一个命令解释器。通常，每个命令都在前台执行，这意味着命令解释器只创建一个进程执行命令，然后等待命令进程执行完之后，再发出提示并读取另一个命令。但是，如果用户指定某个命令在后台执行，命令解释器便创建一个进程来执行该命令，并在命令解释器给出提示、等待输入下一个命令的同时，允许该新进程继续并发执行。为找出哪一个后台进程已执行完，用户必须检查进程的状态。后台执行为像打印这样的任务提供了方便，因为它允许用户在执行这类命令的同时，继续交互式地执行其他命令。

大多数服务器在后台运行，因为它们要永久地执行。服务器在操作系统启动时开始运行，并且在后台执行，等待请求到达。在Linux中，服务器是在系统启动时创建的。在系统引导过程中，初始化进程执行一系列来自守护进程（daemon）启动目录（/etc/rc.d）下的命令。这个启动目录下的脚本按照指定的顺序启动服务器，这样就可以保证远程文件系统访问这样的系统服务比要使用该服务的其他应用服务先启动。

虽然可以编写启动脚本以便将各个服务器进程放到后台，但大多数实际的服务器会自行快速地、自动地放到后台。要理解速度为何重要，让我们考虑一下执行的顺序。想像启动脚本每执行一步就将启动一个服务器。如果服务器快速移到后台，启动脚本就可继续执行并启动更多服务器。如果某个服务器在移到后台前有时延（如直到执行完初始化代码，才移到后台），运行启动脚本的进程就必须等待该服务器，从而使后续的服务器都被延迟。

构造能自动移到后台的服务器还有助于使系统启动更少出错。如果某个特定的服务器S不自动移到后台，而系统管理员又偶然没有在启动脚本中规定S到后台执行，设想一下，这时会发生什么

^① 尽管本章所讲的技术是用Linux的术语来描述的，但它们也适用于其他UNIX系统。

情况呢？自举进程将执行启动命令的序列，直到它遇到启动S的命令。启动进程使服务器在前台运行，等待它执行完后才能继续前进。很遗憾，由于服务器永远在运行，整个自举进程将阻塞，一直在那里等待一个永不会结束的命令。

30.3 编写在后台运行的服务器

Linux程序员将daemon（守护进程）这个术语用于在后台进行服务的进程。将服务器转换成守护进程很简单。该技术包括让服务器调用fork创建一个新进程，然后让父进程退出。为此，服务器要在启动后，立刻执行如下的代码：

```
i = fork();
if (i<0) {      /* less than zero means error occurred*/
    fprintf(stderr, "error when forking:%s\n",
            strerror(errno));
    exit(1);
}
if (i) {          /* nonzero is parent
    exit(0);      /* normal process exit
}
/* child continues execution here and becomes the server */
```

fork调用可产生三种可能的结果：负值表示发生了差错（如系统没有足够的内存创建新进程）；正值表示调用成功，且本进程是父进程；零值表示调用成功且本进程是新创建的子进程。

在本例的代码中，如果发生了差错，服务器将在标准差错（standard error）输出流中打印报文，并调用exit终止进程且退出码为1。通常约定，一个非零的退出码表示异常进程终止。

如果fork调用成功了，if语句可使用返回值区别父进程和子进程。原来的父进程正常退出，而子进程继续执行并成为服务器。从用户的角度最容易使所发生的情况形象化。想像某个服务器Q含有以上的代码。为运行服务器，用户键入服务器的文件名以构成一个命令。当用户输入命令后，命令似乎立刻就执行完了，而且用户收到了输入下一个命令的提示。但是，如果用户请求获得一个进程清单，它会表明服务器已创建了自己的一个副本，而副本正继续在后台运行。

以上代码还处理了一处小细节：它完全将daemon从其父进程中分离开来。在任何时刻，每个UNIX进程有一个父进程，而且，一些UNIX系统函数采取的动作依赖于父子进程关系。调用fork后，服务器的父进程退出，暂时单独留下了新创建的子进程。一旦进程单独存在，UNIX就指定初始系统进程init（initial system process）成为其父进程。我们称此为：孤单的（orphaned）子进程已被init继承^①。虽然这种继承只是一处小细节，但它意味着服务器可完全退出，因为init将调用Linux系统调用wait终止它的各个子进程。

① 见Linux联机手册页中init进程的描述。

30.4 打开描述符和继承

父进程在创建子进程时，它已打开的文件或套接字描述符的副本都将被子进程继承。因此，服务器进程所继承的描述符集合与它如何启动有关。

由于Linux对文件或其他对象使用了一种引用计数机制，保持描述符打开可能会无意义地使用资源。例如，假定一个父进程打开了一个文件，然后执行了一个服务器，该服务器调用fork并移到后台。即使父进程关闭了它的文件描述符副本，服务器仍会有一个打开的副本，并且文件不会被关闭。产生的后果是，文件只有在服务器退出后，才能从磁盘中删除。

服务器必须关闭它继承的所有文件描述符，从而避免不必要的消耗资源。

30.5 对服务器编程以关闭所继承的描述符

为关闭所有继承的文件描述符，服务器在启动后执行如下的代码：

```
for (i=getdtablesize() -1; i>= 0; --i)
    (void) close(i);
```

由于在各种Linux系统中，一个进程可用的描述符个数是不同的，因而代码中不含固定的常量。它调用函数getdtablesize找出进程描述符表的大小。描述符表是从0开始的有序表。它找到描述符的最大数目后，程序从描述符表的长度减1直到0遍历全表，调用close关闭每个描述符。完成后，服务器就关闭了它继承的所有打开的描述符；在未打开的描述符上调用close将不起作用。

30.6 来自控制 TTY 的信号

Linux中的每个进程继承了一个到终端的连接，该终端已被指定为控制终端（control terminal）。Linux将控制终端称为控制tty（controlling tty）。Linux与控制tty建立联系，允许启动进程的用户能控制该进程（如当通过电话拨号线登录的用户挂机时，给进程发送一个挂起信号）。

与大多数进程不同，一个服务器不应接受由启动该服务器的进程所产生的信号。事实上：

为确保来自用户终端的信号不影响后台运行的服务器，服务器必须使自己与它的控制终端分离。

30.7 对服务器编程以改变它的控制 TTY

使进程与其控制终端分开所需的代码仅包括三行：

```
fd = open("/dev/tty", O_RDWR);
(void) ioctl(fd, TIOCNOTTY, 0);
(void) close(fd);
```

open调用返回控制终端用的描述符，接着，调用ioctl指定进程从控制终端分开。最后，调用close释放文件描述符，使服务器可以重新将描述符用于其他I/O（如用于一个传入连接）。

30.8 转移到一个安全的和已知的目录

服务器进程应总在一个已知的目录中执行。这样做可保证如果服务器由于任何原因异常中止（或者如果系统管理员决定中止错误动作的服务器），管理员都知道产生的核（core）文件的位置。此外，所有服务器应在标准系统目录中执行，而不是它们启动时所处的目录。要理解其中的原因，可以考虑一下如果系统管理员注意到服务器出现故障并重启动时，会发生什么情况。如果管理员正在他或她的宿主（home）目录中运行一个命令解释器，并且在启动服务器前忘记将目录变为系统目录，它就会在管理员的宿主目录中运行。如果让进程在某个目录中运行，任何需要该文件系统卸载（unmounted）的系统管理活动都将被禁止。例如，完成一个日常的文件系统转储（dump）就不可能了。

服务器必须转移到一个已知的目录，在那里它可以一直运行下去而不影响正常的系统管理活动。

30.9 对服务器编程以改变目录

为转换到一个已知的目录，服务器调用系统函数 chdir。例如，需要在根目录执行的服务器将发出如下的调用：

```
(void) chdir("/");
```

由于没有哪一个目录能让所有服务器都最佳地工作，因而选择一个适当的目录可能不容易。例如，服务器要发送电子邮件，可能要将目录改变为系统存储外发邮件（outgoing e-mail）的目录（许多UNIX系统上是/usr/spool/mqueue）。但是，监控空闲终端线路的服务器，会将目录改变为可找到所有终端设备的目录（通常是/dev）。

30.10 Linux umask

在Linux中，每个运行着的进程都有一个umask，它为进程所创建的文件指定了保护模式。umask是一个整数，其中低位9比特提供9比特文件保护模式的屏蔽码。只要Linux创建了一个文件，它就为文件计算一个保护模式，计算方法是，将open调用中所指定的模式与进程的umask的按位补码进行按位“与”操作。例如，假定一个进程的umask为027（8进制）。如果进程尝试用模式0777（任何人可读、可写和可执行）创建一个文件，系统通过计算0750（umask的补码）和模式0777（请求的模式）的按位“与”操作，得到正确的文件保护模式。文件模式的结果是0750（可被文件拥有者和文件组读和执行，只能被文件拥有人写，并且不能被其他用户存取）。

可以将进程的umask看成一个安全网，它将阻止意外地创建具有太多读、写或执行特权的文件。这种保护很重要，因为服务器通常作为超级用户（super user）执行，因此它们创建的文件都由超级用户拥有。要点是：

不管在open调用中如何定义保护模式，系统为进程创建的文件所提供的存取特权，不会多于创建进程的umask所指明的特权。因此，在服务器开始时限制umask，可以防止不正确的模式规划可能导致的问题。

30.11 对服务器编程以设置其 umask

限制文件创建模式所需的代码很简单：

```
(void) umask(027);
```

该调用设置服务器的 umask 为指定的值。在进程发出另一个 umask 调用前，umask 将保持有效。

30.12 进程组

Linux 将每个进程放在进程组（processgroup）中。进程组这种记法使 Linux 可将一组相关的进程看作一个实体。通常，用户认为一个进程组是一个作业（job）。具体来说，如果用户创建了三个通过管道（pipe）互连的进程，命令解释器就将它们放在一个进程组中，以便发给它们的一个终止信号可到达这三个进程。

通常，每个服务器的运行与其他进程相独立。因此，它不应该是任何一个组的一部分，并且它不应该接受发给其父进程的组的信号。总之：

每个进程继承了一个进程组的成员关系。为避免接受对其父进程有意义的信号，服务器必须离开其父进程的进程组。

30.13 对服务器编程以设置其进程组

要将一个服务器进程放入它自己的专有进程组中，所需的代码很简单：

```
(void) setpgrp(0, getpid());
```

getpid 调用返回当前运行进程（即服务器的进程）的进程 id，并且调用 setpgrp 请求系统将指明的进程放进一个新的专有进程组中。

30.14 标准 I/O 描述符

许多库例程希望打开三个标准文件描述符以便用于 I/O：标准输入（0）、标准输出（1）和标准差错（2）。特别是，像 perror（它打印差错报文）这样的标准库例程，会直接把报文写入标准差错描述符而不需加以检查。如果这些描述符有打开的，当服务器调用了一个库例程对其读或写时，这些 I/O 可能会发生在某个终端或某个文件上。为安全起见，程序员通常打开标准描述符，并将其连接到一个无害（harmless）I/O 设备。然后，若服务器中任一例程使用某个标准描述符进行 I/O 时，服务器就不会进行无意识的 I/O 了。总之：

由于许多库例程假定三个标准 I/O 描述符是打开的，实际的服务器通常打开三个描述符，并将其连接到一个无害 I/O 设备上。

30.15 对服务器编程以打开标准描述符

要打开已被关闭的标准描述符，所需的代码由三个系统调用组成：

```
fd = open("/dev/null", O_RDWR);           /* stdin */
(void) dup(fd);                         /* stdout */
(void) dup(fd);                         /* stderr */
```

对open的调用指明了特殊的Linux文件名/dev/null，它对应一台设备。与/dev/null相关联的设备总是在输入上返回一个文件结束条件，并将所有的输出丢弃。因此，读或写到/dev/null是没有反应的；在任何存储设备上也不会有积累数据。

对系统函数dup的调用将复制一个现有的文件描述符。系统通常约定，Linux顺序指派文件描述符，并会从零开始。一旦服务器已打开了/dev/null，并为它获得一个描述符，dup调用将提供标准输出（描述符1）和标准差错（描述符2）的副本。

30.16 服务器互斥

对于大多数服务来说，在任何时刻只有一个主服务器。如果需要并发执行，主服务器应处理所有的并发性问题。如果许多人都有特权，限制服务器的执行可能特别重要。例如，假定系统两个程序员都注意到某个服务器出现故障，并且都同时重启服务器。除非管理员协调了他俩的行动，否则服务器的两个副本都试图运行，而结果可能无法预料。作为一个规定：

一次只应启动执行一个服务器程序的副本；所有的并发性都应由服务器进行处理。

30.17 对服务器编程以避免多个副本

我们说一个服务器的多个副本是互斥的，服务器在开始执行时必须调用一种机制以保证互斥。在Linux中，大多数进程使用一种锁文件(lockfile)进行互斥运行。每个服务器使用一个单独的锁文件。例如，一个提供行打印服务的服务器，可能会使用文件/usr/spool/lpd.lock。而一个白页服务器(white pages server)可能使用文件/usr/spool/wp.lock。当服务器的一个副本开始运行时，它将试图建立该锁文件。如果没有其他进程持有该锁，尝试就会成功；如果服务器的另一个副本已经锁定该文件，尝试就会失败。

实现互斥所需的代码由几个系统调用组成：

```
/* Acquire an exclusive lock, or exit. Assumes          */
/* symbolic constant LOCKF has been defined to be      */
/* the name of the server's lock file. For example,      */
/* #define LOCKF /usr/spool/lpd.lock                   */
/* */

lf = open(LOCKF, O_RDWR | O_CREAT, 0640);
if (lf > 0)           /* error occurred opening file */
    exit (1);
```

```
if (flock(lf, LOCK_EX | LOCK_NB))
    exit(0);           /* could not obtain a lock */
```

正如注释所提示的，该代码例子假定符号常量LOCKF已定义为服务器的锁文件名。调用open获得锁文件的一个描述符，并在必要时创建该文件。注意，文件不含有任何数据；它只要存在就行，以便服务器可持有该锁。flock调用请求互斥使用该锁文件。如果flock成功就返回零，若尝试失败就返回非零。如果服务器不能获得一个互斥锁，它只需简单地退出，因为必然有服务器的另一个副本已在运行。虽然可使用其他机制做到互斥运行（如创建一个具有模式000的加锁文件），但使用flock的主要优点在于它在系统崩溃后可自动释放锁。事实上，进程只在执行时保持flock建立的锁。如果服务器崩溃或系统重启动，锁会被释放。而那些靠创建文件来保证互斥的服务器，要求系统在重启动后（或没有服务器的副本可执行后），于重新运行服务器之前删除已创建的文件。

30.18 记录服务器的进程 ID

在大多数实际环境中，万一服务器出现错误动作或崩溃，系统管理员需要能快速找到它。虽然可列出系统中的所有进程，但在一台负载繁重的机器上，这样做可能要花相当长时间。为避免让系统管理员搜索一个服务器进程，大多数实际的服务器在一个熟知的文件中记录其进程标识符。当管理员们需要查找某个服务器进程时，他们就在该文件中查找服务器的进程标识符。

服务器应使用哪个文件保存其进程标识符呢？最明显的选择是服务器的锁文件。每个服务必须有一个锁文件，而且锁文件通常没有其他有用的内容。

30.19 对服务器编程以记录其进程 ID

一旦服务器打开一个锁文件并获得一个描述符，使用锁文件记录服务器的进程标识符并不复杂。所需的代码如下：

```
char pbuf[10]; /* an array to hold ASCII pid */
/* Assume the lock file has been opened and its */
/* descriptor has been stored in variable lf as */
/* the code in the previous section shows      */

(void) sprintf(pbuf, "%6d0", getpid());
(void) write(lf, pbuf, strlen(pbuf));
```

为使文件易于阅读，代码调用sprintf将进程id转换为可打印字符串，该串中含有等价的十进制值。然后它调用write将字符串写入锁文件。由于文件由可读文本组成，管理员可通过显示该文件找出进程id，并不需要特殊的工具对它进行读或格式化。

30.20 等待一个子进程退出

当Linux进程退出时，系统只有在通知其父进程后才能完成进程的终止。父进程必须在终止完成前调用系统函数wait。同时，终止的进程保持在zombie状态，它有时又称为死(defunct)进程。

30.21 对服务器编程以等待每个子进程退出

我们知道，当服务器创建了进程且父进程退出后，子进程将成为init的子进程。由于init重复调用wait，它所有的子进程可彻底地终止。因此，如果主服务器进程退出了，也不会发生任何问题。

主服务器创建的进程是它的子进程，而不是init的子进程。第11章的代码说明了如何让一个并发服务器等待其子进程终止。它说明了服务器要接受每个终止子进程的信号，并在信号处理程序中调用wait。

30.22 外来信号

即使服务器将自己与其控制终端分开，它仍可能收到信号。特别是，任何系统管理员或有特权的进程可给进程发送信号。通常，管理员发送信号SIGKILL(9)终止服务器。程序员可决定使用一个或多个信号来控制服务器的运行。例如，程序员可让服务器在收到一个HANGUP信号(1)后，重新初始化自己。

30.23 对服务器编程以忽略外来信号

大多数实际服务器都设法忽略除几个用于控制的信号外的所有信号。为此，服务器或者执行系统函数signal调用，或者执行系统函数sigvec调用。例如，为忽略HANGUP信号，服务器调用：

```
(void) signal(SIG_IGN, SIGHUP);
```

与此有关的一个具体事例是，如果进程P指明SIG_IGN忽略SIG_CHID信号，那么，当P的子进程退出时，操作系统不会创建zombie进程。

30.24 使用系统日志设施

30.24.1 产生日志报文

服务器以及某些客户要为系统程序员或系统管理员产生输出。当然，在构造程序过程中的大多数输出是帮助程序员进行代码排错的一些报文。一旦服务器开始实际的服务，输出通常仅限于差错报文，即服务器发现反常情况或意外事件时产生的消息。但是，即使是实际软件，也有可能有规则地产生输出。例如，可编写一个服务器，保存每个连接请求或每个事务处理的日志。为了系统安全的维护，可编写一个服务器，在每次拒绝来自非授权客户的连接请求后，将有关信息记录下来。

许多早期的服务器在一个控制台日志上记录所有输出。此方法源于早期计算机系统，那时，控制台终端是在纸上打印输出的。如果程序把信息写入控制台，它就成为永久系统日志的一部分，以后需要时可以再行查看。在这种系统上，当服务器开始运作时，它打开一个输出描述符用于系统的控制台终端，并在写日志报文时使用该描述符。

30.24.2 间接方式和标准差错的优点

让服务器将日志报文写到控制台终端的主要缺点是缺乏灵活性。当程序员构造并测试服务器代码时，程序员必须走到系统控制台查看日志。此外，系统管理员经常将客户和服务器软件的副本移到新机器上。由于一些机器不使用硬副本（hard-copy）设备作控制台，在服务器软件移到这种机器上之前，必须改变软件，以便使日志报文写入到一个文件中。通常，改变日志报文的目的地意味着要重新编译源代码。

Linux 为每个进程提供了一个用于标准差错输出的文件描述符（描述符 3）。当 Linux 程序员编写服务器软件时，程序员不必知道服务器是将差错报文发给了控制台打印机，还是发给了一个文件。他只需将所有差错报文写到标准差错描述符，并依赖于启动服务器的人来决定将报文输出到哪里。在启动服务器时，可以将标准差错描述符连到了一个文件，也可以将它连到系统控制台。实质上，标准差错这种想法提供了一定程度的间接方式（indirection），它将源程序与运行环境隔离开：源代码引用标准差错描述符，不需要知道它是与文件相关联还是与终端相关联。

考虑一个网点有多台机器，而每台机器上都运行某个服务器的一个副本，在使用标准差错描述符中获得的灵活性就显而易见了。一台机器可能使用一台控制台打印机记录差错报文，而其他机器可能使用文件来记录。系统管理员可在两台机器上运行同一个服务器代码，而不需要重新编译，因为 Linux 允许在启动服务器时进行 I/O 重定向。

30.24.3 I/O 重定向的限制

与将报文写到一个特殊设备或文件的方法相比，使用标准差错描述符使得服务器代码更加灵活了，但它并没有解决所有问题。程序员或系统管理员可能决定让服务器发送一些（或所有）差错报文到用户终端。或者，程序员可能要设法将差错报文转发到另外一台机器上（即将报文发送到一台有打印机的计算机上）。

Linux 标准差错描述符机制不能为处理各种可能的目的地提供足够的灵活性，因为系统不允许输出被重定向到任意一个程序或网络连接上。因而，需要另一种更有力的机制。

30.24.4 客户－服务器的解决方案

系统设计人员已发现，他们可使用客户－服务器模型构造差错日志的机制，这样做比使用 I/O 重定向具有更大的灵活性。实质上，每台参与的计算机运行一个日志服务器，它接受和处理系统日志有关的报文。可以编写日志服务器，将收到的报文写到计算机的控制台终端，或写到文件中，或将它们送到系统管理员的终端，甚至送到另一个机器上的日志服务器。

当一个正运行的程序需要将一个报文写到系统日志时，它就成为了日志服务器的客户。这个程序将其报文发给日志服务器，然后继续执行。日志服务器处理报文，将它写到系统日志中。

如果系统管理员决定改变系统处理差错报文的方式时，管理员只需要改变日志服务器。例如，管理员可配置一个日志服务器，使它在系统控制台上打印所有的日志报文。以后，管理员还可改变日志服务器，使它将所有的日志报文加入到一个文件中。

值得注意的是，日志服务器所提供的这种间接方式，很像前面描述的标准差错描述符。程序只有被编译进代码的关于日志服务器的报文，但并不知道日志服务器如何处理每个报文。此外，日志服务器的地址可用特殊名字 localhost 表示，这允许已编译好的程序在它的机器上找到日志服务器，而不需要知道机器的地址。

30.24.5 syslog 机制

Linux 含有一个日志机制，它使用以上描述的客户 - 服务器方法。该机制就是 *syslog*，它包括日志服务器所用的代码 (*syslogd*)，以及程序用于联系服务器和发送报文的库例程。

syslog 机制提供两个重要特性：(1) 它将报文进行分类；(2) 它使用一个配置文件，允许系统管理员指定服务器应如何处理每个报文类。由于 *syslog* 分开处理各个报文类，系统具有相当大的灵活性。例如，一台机器上的系统管理员，可能选定只要发生严重差错，就发送一个报文给系统管理员，同时又可选定将低优先级的报文（例如，报告性的报文）放在文件中而不必通知任何人。另一台机器上的系统管理员可能还会作出不同的选择。

由于 *syslog* 使用配置文件，它允许系统管理员指定服务器应如何处理各个报文类，从而方便了系统的使用。要改变 *syslog* 处理差错报文的方式，管理员只需要改变配置文件；客户和服务器软件可保持不变。

30.24.6 syslog 的报文类

syslog 按两种方法划分消息类。第一，它将程序分组，每组称为一个设施 (facility)。第二，它将来自各个设施的报文细分为 8 个优先级。

30.24.7 syslog 的设施

图 30.1 中的表格列出了 *syslog* 定义的设施及其含义。

设施名	使用该设施的子系统
LOG_KERN	操作系统核心
LOG_USER	任何用户进程（即正常的应用程序）
LOG_MAIL	电子邮件系统
LOG_DAEMON	在后台运行的系统守护进程
LOG_AUTH	授权和鉴别系统
LOG_LPR	打印机假脱机系统
LOG_RPC	RPC 和 NFS 子系统
LOG_LOCAL0	为本地使用保留；从 LOG_LOCAL1 到 LOG_LOCAL7 也都是保留的

图 30.1 *syslog* 定义的设施类型。每个日志报文必须来源于这些设施中的某一个

如图所示，*syslog* 为每个主要子系统都提供一个设施。例如，有关电子邮件的程序属于 *LOG_MAIL* 设施，而大多数运行在后台的服务器属于 *LOG_DAEMON* 设施。

30.24.8 syslog 的优先级

syslog 定义了 8 个优先级，如图 30.2 所示。优先级的范围从 *LOG_EMERG*（用于大多数严重的紧急情况）到 *LOG_DEBUG*（程序员在需要记录排错信息时使用）。

优先级	描述
LOG_EMERG	极其紧急；该报文应广播给所有用户
LOG_ALERT	某种应立即纠正的情况（如某个崩溃的系统数据库）
LOG_CRIT	像硬件差错这样的关键情况（如磁盘故障）
LOG_ERR	某种要求注意但并不关键的差错
LOG_WARNING	某种警告，它表明可能存在某种差错情况
LOG_NOTICE	某种情况，它可能不是差错，但也许需要注意
LOG_INFO	某种信息报文（如当某个服务器开始执行时发出一个报文）
LOG_DEBUG	程序员用于对程序进行排错的报文

图 30.2 syslog 定义的 8 个优先级。每个日志报文都必须具有以上定义的优先级之一

30.24.9 使用 syslog

在任何时候，当程序使用syslog处理日志报文时，都必须要为报文指定一个设施和一个优先级。为使syslog更易于使用，库例程允许程序员在程序开始时指定一个设施，并让syslog把该设施用于后续的报文。为指定一个初始设施，程序调用过程openlog。openlog带有三个参数，它们分别指明一个标识符字符串、一组处理选项和一个设施说明。syslog在程序写入的所有报文前都加上指明的标识符，以便可在日志中找到它们。通常，程序员选用程序名作为其标识符字符串。openlog初始化日志，并存储标识符字符串和设施以备后用。例如，测试一个专用程序的程序员可调用：

```
openlog("myprog", LOG_PID, LOG_USER);
```

来指明了描述符字符串 myprog、日志选项 LOG_PID 以及设备 LOG_USER。选项 LOG_PID 要求：对每个日志报文，syslog 同时要记录下程序的进程标识符。

当程序需要发送一个日志报文时，它调用过程syslog。syslog将报文发给本地机器中的syslogd服务器。过程syslog带有的参数的个数是可变的。第一个参数指明报文的优先级，第二个参数指明一种类似printf的格式。与在printf中一样，跟在格式后的参数含有格式中所引用的值。最简单的情况下，程序可用常量字符串作报文来调用syslog。例如，要在系统日志上记录排错报文，程序可调用：

```
syslog(LOG_DEBUG, "my program opened its input file");
```

一旦程序不再使用syslog，它可调用过程closelog关闭日志文件（即终止与服务器的联系）。closelog将终止日志文件连接，并释放分配给它的I/O描述符。因此，调用closelog释放日志文件描述符可能是很重要的（如在调用fork前）。

30.24.10 syslog 配置文件举例

syslog机制之所以具有灵活性，主要是因为它使用了配置文件来控制服务器。配置文件使用了模式(pattern)，以便让系统管理员指定如何处理设施和优先级的各种组合。

在大多数系统上，syslog配置文件是/etc/syslog.conf，它是一个含有若干说明的文本文件，每行一条说明。说明是由空格隔开的一对字符串组成。左边字符串指定一个模式，它与设施和优先级的某种组合相匹配。右边字符串指定一个与该模式匹配的报文处理方式(disposition)。例如，有如下说明：

```
lpr.debug    /usr/adm/printer-errs
```

它含有一个模式，该模式与设备为 lpr 和优先级为 debug 的所有报文相匹配。由于处理方式字符串由斜线开始，syslog 就将它解释为一个文件名。因此，如果 syslogd 遇到如上说明，它便将来自 lpr 设施的 debug（排错）报文发送到文件 /usr/adm/printer-errs 中。syslogd 将星号解释为一种处理方式字符串，其含义为“将报文广播给所有用户”。它还将其他不以斜线开始的处理方式字符串解释为应当接收该报文的用户的登录名。

下面将用一个配置文件的例子来阐明这一概念，并说明系统管理员能如何指定日志记录的行为。以下是一个在某个系统上使用的配置文件的样本：

```
*.err;kern.debug;auth.notice          /dev/console
kern.debug;daemon, auth.notice;auth.info;*.err;mail.crit  /usr/adm/messages
lpr.debug                                /usr/adm/printer-errs
mail.debug                               /usr/spool/mqueue/syslog
*.alert;kern.err;daemon.err            operator
*.alert;                                 root
*.emerg                                  *
```

模式匹配是一种强有力的机制，这是因为它允许系统管理员在表示规约时，不必写下设施和优先级的各种组合。例如，以上的配置文件使用了模式 *.err，它的含义是“任何设施上的差错优先级为 err 的所有报文”。

30.25 小结

实际使用当中的服务器使用了一些技术，这些技术使得它们更易于管理和排错、更为强大，而且较少出错。本章讨论了这些技术的出发点，还展现了每种技术所需的代码。我们所讨论的特定技术包括如下概念性的操作：

- 服务器作为后台进程（守护程序 daemon）运行。
- 关闭被继承的文件描述符。
- 将服务器与其控制 TTY 断开。
- 将服务器转移到一个安全的、已知的目录。
- 设置 Linux umask。
- 将服务器放在专用的进程组中。
- 打开三个标准 I/O 描述符。
- 保证互斥运行。
- 记录服务器的进程标识符。
- 等待子进程终止。
- 忽略外来信号。
- 使用 syslog 处理差错报文。

深入研究

Linux 程序员在实现服务器时，他们所遵从的许多规则都来自未写出来的约定和所受的启发；程序员通常是在阅读现有程序时学到这些技术的。Stevens [1998] 讨论了如何编写守护进程，还介绍了一些本章只简要说明的技术。

习题

- 30.1 构造一个过程 `daemonize`，它含有出自本章的技术。
- 30.2 阅读 Linux 联机手册中有关信号的内容。哪些信号不能被忽略？如果服务器收到这些信号中的某一个，情况会如何？
- 30.3 一些程序员让服务器在收到某个特殊信号（如 `SIGHUP`）时从容地终止。这样做使得系统管理员能够通过发送某个设计好的信号来终止服务器。如果服务器已使自己与控制终端分离开，管理员如何发送这个信号？
- 30.4 在上一个问题中，一个无特权的用户，能否通过发送一个指定信号终止服务器？
- 30.5 在我们的例子代码中，先关闭了所有的文件描述符，然后再次打开标准输入、输出和差错。一些服务器保留了这三个描述符的值，即从其父进程继承来的值。这样做的优点和缺点是什么？（提示：考虑有关登录差错信息的记录。）
- 30.6 大多数服务器在运行时具有根特权（root privilege），在 Linux 中，根特权的含义是绝对特权。为什么服务器可选用较低特权运行？如何能做到这一点？
- 30.7 编写一个服务器，它使用 Linux 系统函数 `creat` 提供互斥。（提示：此技术很著名且被广泛使用，特别是在较老版本的 UNIX 程序中。）
- 30.8 比较使用 `creat`（见上一题）的互斥方法、在 `open` 调用中使用 `O_EXCL` 的互斥方法，以及使用 `rename` 的互斥方法。这三种技术中哪种更受欢迎？为什么？
- 30.9 查看某个实际服务器的源代码。它使用了本章中的什么技术？它还用到了其他技术吗？
- 30.10 阅读 Linux 联机手册中的 `syslogd`。系统管理员如何在服务器启动后改变配置文件？你能提出一种替代方案吗？

第31章 客户-服务器系统中的死锁和资源缺乏

31.1 引言

前几章的重点是客户-服务器系统的设计，以及各个客户和服务器程序的结构。我们讨论了客户和服务器支持并发运行的途径，考察了如RPC这样的工具，编程人员可使用这些工具构造客户-服务器软件，还举例加以了说明。

本章将探讨分布式计算的动态行为，重点是客户-服务器系统可能会出错的地方。在前几章中，我们讨论了发现的潜在问题，本章将进一步扩充这些讨论，并将研究可能导致服务延迟或中断的两个条件：死锁和资源缺乏（starvation）。在实际工作环境中是不能允许服务崩溃的，因此，这两种情况对于在实际环境下工作的程序员来说，是特别重要的。

本章除了要考察有二义性的协议规约如何能产生死锁外，还将讨论编程的差错和疏忽如何会使一个误动作的客户造成服务器崩溃，从而使它不能对其他客户提供服务。本章阐述了可用于防止客户-服务器系统死锁的技术，并解释了其后果。

31.2 死锁的定义

在计算机系统中，死锁（deadlock）这一术语用于描述这样一种情况，由于系统中两个或多个部件的集合发生阻塞，并且每个部件都等待集合中其他部件动作，从而使计算无法进行。在典型情况下，每个部件是一个被阻塞的进程，它等待集合中其他进程释放所掌握的资源。

死锁是一种永久故障，不要把它与临时阻塞相混淆。要测试是否发生了死锁很简单：只要看一下某个外部输入是否会使计算进行下去便知道了。例如，考虑三个进程的一个集合。设想两个进程在第三个进程与某个用户交互时阻塞。一旦用户响应，第三个进程就通知其他两个进程，并且处理继续。虽然三个进程在等待某个用户响应时，可能会保持任意长时间的阻塞，但集合并未死锁，因为来自用户的输入会使得处理继续下去。

相反，当集合中每个进程都等待来自集合中其他进程的输入时，这些进程的集合将陷入死锁。由于每个进程被阻塞，没有哪个进程能产生输出。因此，没有进程能够收到输入，也就没有办法能脱离死锁。从上述测试可见，由于集合中没有进程等待外部输入，这种情况就是死锁。

31.3 死锁检测的难度

实时检测死锁^①是困难的，在分布式系统中通常不可能。这有两个原因。首先，为了区别死锁

^① 死锁的同义词包括循环等待（circular wait）、死包围（deadly embrace）和同步锁住（synchronized lock），本章将对它们加以区分。

和临时阻塞，检测机制需要知道每个程序所掌握的资源，以及程序阻塞的原因。在客户—服务器环境中，要获取这种信息就意味着将查询多个操作系统，每个操作系统可能使用了自己独有的一组操作。其次，由于编程人员能创建抽象资源，操作系统就不可能判断哪些程序拥有这些资源——只有创建和使用这些资源的程序自己才知道。

令人惊讶的是，即使提供了系统的每个部件的源代码，判断系统是否会死锁仍同证明一个数学定理一样困难。死锁的潜在性只有在考虑了系统的动态行为时才明显。即死锁依赖于执行的顺序，而且当客户和服务器在不同计算机上运行时，可能出现许多不同的顺序。更重要的是，即使一个分布式系统的每个部件的设计和实现是合理的，死锁仍可能在此系统中发生。结果可阐述如下：

由于检测死锁是非常困难或不可能的，不可能设计出这样一个实用的、能判断若干客户和服务器的集合是否陷入死锁的计算机程序。

31.4 避免死锁

怎样才能确保服务不被瓦解呢？一般来说，答案在于仔细规划设计。设计协议、实现软件或安装和配置客户—服务器系统的每个人，都必须意识到死锁的可能性，必须注意避免创建易受影响的系统。

为避免死锁，我们必须了解它们发生的方式。下一节将描述在客户—服务器系统中可能出现问题的三种方式：在一个客户和一个服务器之间，在多个客户和一个服务器之间，在多个客户和多个服务器之间。

31.5 客户和服务器间的死锁

客户—服务器死锁的最简单形式发生在一个客户和一个服务器之间。如果客户在等待服务器发来报文时阻塞，而服务器为等待客户发来报文阻塞，它们这一对阻塞将陷入永久的死锁。

为防止这种死锁，大多数应用协议的设计都使用请求响应（request-response）范例。也就是，一方（通常是客户）发送请求到响应的另一方。协议必须指定哪一方创建请求和哪一方发送响应。

死锁错误可能通过两种方式引入单个客户和单个服务器的交互中。第一种，若协议设计没有充分地说明同步，这可能导致客户和服务器互操作失败。第二种，若协议设计是以可靠的交付为前提的，而实际上使用了不可靠传输服务，这时将会出错。

要理解对同步缺乏充分的规约会怎样引起一些问题，考虑如下的应用协议：

1. 客户必须首先建立到服务器的连接。
2. 连接建立后，客户或服务器必须立刻发送一个开始报文；另一端等待该报文，并发送对开始报文响应。
3. 开始报文交换完毕后，客户发送请求；服务器为每个请求发送响应。
4. 客户在收到其最后一个请求的响应后，关闭连接。

当设计者试图让实现者自由选择细节时，这种不精确的协议规约可能会出现问题。例如，设计者也许不能决定最好是让客户发送第一个报文，还是让服务器发送第一个报文。因此，为获得最大限度的灵活性，协议故意写得有二义性。遗憾的是，遵从该协议的两种实现都可能死锁：实现客户

端的程序员可能假定服务器会发出开始的报文，而实现服务器端的程序员可能假定客户会发送开始的报文。当这样实现的客户和服务器交互时，他们各自都将阻塞在那里，以便等候另一端发来开始的报文。

为理解不可靠的传输如何引发差错，考虑为可靠传输（如TCP）设计的请求响应协议。协议可指明客户应发送请求，然后等待响应。如果将这种协议用在不可靠传输（如UDP）上，报文就可能丢失。遗憾的是，一个丢失的请求或响应将产生死锁，因客户将等待响应而保持死锁，而服务器为等待下一个请求而保持阻塞。

31.6 在单个交互中避免死锁

有两种方法可避免单个客户和服务器之间的死锁。首先，设计应用协议时应说明同步。一方应负责初始化交互（如发送一个请求），并且交互的顺序不能有二义性。其次，实现必须要么使用可靠传输协议，要么包含一种定时机制，该机制要求发送端等待响应的时间有一个最大限度，超过限度后要重传请求。

虽然一个客户和一个服务器之间的死锁是不希望发生的，但死锁只影响所涉及的一对程序。而一个共享服务器还有另外一个不利之处，这是因为，引起服务器不可用的任何问题都将阻止其他客户获得服务。更重要的是，一个易受这些问题影响的服务器，它容易受到恶意客户的攻击，这种攻击阻止它为其他客户提供服务。

31.7 一组客户和一个服务器之间的资源缺乏

资源缺乏（starvation）这一名称用于描述这样一种情况，一些客户不能获得服务，而其他客户却能。资源缺乏违反了公平原则，即服务器必须为所有客户提供相同服务。

一个循环式的服务器允许交互时间任意长，这本身就是不公平的，因为它使单个客户单独使用服务，而其他客户被排除在外。因此，大多数循环式的服务器不允许单个客户独自长期地使用服务。为确保公平性，循环式服务器可以限制某个给定客户所允许发送请求的数目。例如，一个循环的、面向连接的服务器，可在处理完一个客户请求后关闭连接。或者服务器可以在经过固定长时间后关闭连接。

下面举例说明一个恶意客户如何阻止其他客户使用服务，我们考虑一个客户如何能发现一个循环的、面向连接的服务器。设想协议规定客户应向服务器发送请求，而服务器则作出响应。一个恶意客户为了使其他客户缺乏资源，打开到服务器的一个连接，然后不再发出请求。服务器将阻塞在那里以便等待客户发来请求，但却没有请求到来。而同时，其他客户都不能使用该服务器了。

当然，服务器可通过加入定时机制来阻止这类问题，该机制会在经过一个定长超时后自动关闭连接。当一个客户首先形成连接时，服务器就启动一个计时器，其起始值为零。当请求到达后，服务器就停止计时，并形成响应，再将计时器清零。如果计时器的数值达到T，服务器就关闭连接。

31.8 忙连接和资源缺乏

上一节描述了空闲连接计时器，虽然它处理了客户不发送请求这种情况，仍有可能发生资源缺乏。要知道其原因，我们需要观察两个事实。首先，空闲计时器度量了响应传输和收到下一个请求

之间的时间。其次，一个传输协议使用缓存和流控制。

由于传输协议在连接的两端各放置一个缓存，了解缓冲的细节很重要。在发送端，发送应用程序将输出数据存放在发送缓存中，TCP从中取数据发送；在接收端，TCP将收到的数据存放到接收缓存中，接收应用程序从中提取数据。只要接收端的缓存中还有空间，TCP就继续从发送端的缓存提取数据传送。

一个客户如何能不公平地占用服务器的部分时间呢？客户可通过延迟或阻止传输来做到这一点。具体来说，一个客户可以：

- 指明 TCP 接收缓存大小，但它比所期望收到的数据量要小^①。
- 发送请求，要求服务器传输数据。
- 也可不读取收到的数据而使得接收缓存满，从而阻止传输；或者通过缓慢读取收到的数据而延迟传输。

在这种情况下，发送端 TCP 在接收端 TCP 的缓存满之前发送数据，接收缓存满时，接收方将接收窗口大小变为零。发送方在接收方缓存可重新使用之前不能再发送数据。因此，在客户读取数据前，传输终止了。

在发送端，服务器继续往输出缓存中写数据。由于传输被延迟或停止，输出缓存最终也会满。当服务器试图往已满的缓存中写数据时，服务器进程就会被阻塞。由于服务器不是因等待请求而阻塞，这种情况就不能用一个闲连接计时器来解决。服务器是在发送响应时阻塞的。

31.9 避免阻塞的操作

服务器如何在传输过程中避免阻塞呢？一般来说，有两种解决办法：服务器并发执行；或者服务器避免使用会产生阻塞的调用。在前一种方法中，由于服务器用不同进程处理各个客户，引起延迟的客户不会影响其他客户。在后一种方法中，服务器可为忙连接（busy connection）实现一种定时机制。在调用每个会产生阻塞的操作前（如 send），服务器必须确保调用不会被阻塞。如果系统报告套接字还没准备好接收数据，服务器就设置计时器，然后再尝试。如果套接字在指定的定时内还没准备好，服务器就关闭连接。

31.10 进程、连接和其他限制

虽然并发有助于解决许多死锁问题，但并发也有限制。第 16 章指出，由于无限制的并发会引起问题，并发必须加以管理。具体来说，一个服务器为每个客户创建一个新进程，由于操作系统不允许任意地并发操作，服务器还是容易受到误操作的客户的伤害。最终，一个并发服务器将耗尽系统资源。例如，操作系统限制了进程的数目、活动套接字的数目、可使用的描述符总数和 TCP 可分配的传输控制块（TCB）的数目。另外，每个打开的 TCP 连接要使用缓存空间，因而，每个连接都需要内存。

为使一个服务器免受资源缺乏问题的影响，编程人员必须规划所有资源的使用，包括内存使用、打开的连接和并发的程度。遗憾的是，预期或控制资源的使用是很困难的。由于很少有一个服

^① 有一个 socket 选项允许应用程序定义 TCP 使用的缓存大小。

务器是专为一台计算机构造的，编程人员在编写代码时通常不清楚系统的限制。此外，服务器通常是在复杂的分时计算机系统上运行的，所有进程要共享系统资源。因此，无论在什么时候，一个服务器可使用的资源，与其他应用程序当前所使用的资源有关。

如果不能预期或管理并发性和资源的使用，编程人员至少可以设法让服务器报告问题。例如，服务器可检测每个系统调用的返回值，并使用一个日志报告差错。管理人员可以周期性地检查日志，以判断服务器是否遇到了困难。如果出现差错，管理人员可采取进一步的措施以判断差错原因。虽然这种报告不能阻止问题发生，但他们提供了一种让管理者监控系统行为的机制。

31.11 客户和服务器的循环

死锁和资源缺乏的大多数有害形式可能源于多个服务间的相互依赖性。为理解这一问题，回忆一下，我们说过，一个提供某种服务的服务器可以是另一个服务器的客户。例如，想像一个程序员正构造一个文件服务器。如果文件系统要记录文件最后被改变的时间，服务器在处理 write 请求时就可能需要获得当前时间。

在大多数操作系统中，一个应用程序要调用系统函数获得当前时间。系统函数从硬件时钟里提取当前时间，然后向应用程序返回时间值。然而，在客户-服务器环境中，时间要从某台远程机器中获取，实质上，获得当前时间的函数包含了客户代码。调用函数的应用成了时间服务器的一个客户。客户发送请求，等待包含有时间值的响应，然后将该值返回给调用程序。

在这种环境中，当客户和服务器不慎形成循环时，可能会发生死锁。在上例中，设想程序员被指派修改时间服务器。程序员为了有助于调试错误，可能决定形成一个日志，该日志记录所有发给时间服务器的调用。遗憾的是，如果日志写入到一个文件中，会产生循环依赖：文件服务器调用时间服务器，时间服务器又调用文件服务器以写入一个日志信息。如果两个服务器中有一个不是并发的，就会立刻发生死锁。如果两个服务器都是并发的，文件服务器将再次调用时间服务器，而时间服务器又将调用文件服务器，此循环将会继续，直到资源被耗尽。

上例说明了一种称为活锁（livelock）的问题。类似死锁，活锁产生于对循环的依赖。然而与死锁不同的是，活锁的参与者忙于使用CPU和发送报文。在上例中，如果客户和服务器使用无连接协议通信，就会发生活锁。文件服务器发送报文给一个时间服务器，并等待响应。时间服务器处理报文，它又发送请求给文件服务器。当请求到达后，文件服务器产生第二个报文给时间服务器，这又引起时间服务器产生第三个请求，如此继续。虽然两个服务器都忙于发送和接收报文，但循环却不会被打破。如果其中一个服务器动作较慢，它的接收缓存会变满，会引起后续的一个或多个报文丢失。但是，只要服务器从队列中读取一个报文，另一个报文又会到达，占用前一个报文的位置。由于服务器只忙于处理没完的报文循环，它们已不能完成有用的工作了。

31.12 用文档确认依赖性

为了防止导致死锁或活锁的循环，程序员和管理人员必须避免引入服务器间的依赖循环。由于计算机系统朝着客户-服务器环境发展，理解这种依赖更为困难。例如，虽然UNIX文件系统可配置成用NFS存取远程文件，但用户或程序员不能通过文件名知道文件是远端的。同样，嵌入到系统命令（如查询当前时间）中的客户软件，可能不太显而易见。

为了有助于避免不慎的依赖性，程序员在编写客户-服务器软件时，需要了解每个库例程或操

作系统函数是否要访问远程服务器。为提供必要的信息，一个机构应该保留每个服务器以及它所依赖的其他服务器的详细记录。任何人在创建或安装软件时，应当更新该依赖性清单。

保存依赖信息有两种方法：粗糙和细致。粗糙的避免依赖方法将每个服务作为独立实体对待，并保证在服务间没有循环存在。例如，如果一个远程文件服务依赖时间服务，编写时间服务时就不能使用远程的文件服务。细致的避免依赖方法将每个服务器作为独立实体看待，并确保在服务器间不存在循环。例如，文件服务器 X 可调用时间服务器 Y，且时间服务器 Y 可调用文件服务器 Z（但不能是服务器 X）。

粗糙方法的主要优点是容易记录——一个典型的环境只含有一些服务，并且在这些服务中很少有相互依赖。粗糙方法的主要缺点是不必要地限制了交互。相比而言，细致方法允许最大限度的依赖——只要不在各个服务器间存在循环，就允许服务存在循环。细致方法的主要缺点在于必须保存更多的细节。一旦有服务器加入到集合中，或服务器配置一旦改变，依赖性的记录就必须更改。

31.13 小结

死锁和资源缺乏是客户-服务器环境中的基本问题。死锁指两个或多个系统部件的集合阻塞而相互等待的情况。资源缺乏的概念更通用，它指不能公平访问服务的任何情况：一组客户比其他客户获得更好的访问。

死锁可发生在单个客户和服务器之间。这种死锁通常产生于含糊的协议规约，或者将针对可靠传输的协议用于不可靠的传输。

当多个客户访问单个服务器时，某个客户和服务器间的死锁或活锁可能引起资源缺乏，这意味着其他客户不能获得服务。导致资源缺乏的原因可能是一个客户打开了到某个循环式服务器的连接，但它不发送请求，或者一个客户产生请求但不处理响应。空闲连接计时器可处理前一种情况，但对后一种则不行。

由于一个服务器可以暂时成为一个客户，在两个或多个服务器组成的集合中，如果集合中的每个服务器都试图成为集合中另一个服务器的客户，这个集合就会死锁。为防止这种死锁，必须禁止服务器间的循环依赖关系。

习题

- 31.1 画出一张你所在单位的各个服务之间的依赖性图表。
- 31.2 画出一张你所在单位中各个服务器之间的依赖性图表。
- 31.3 当计算机 B 上的文件系统使用 NFS 存取计算机 A 上的文件时，计算机 A 上的文件系统可以使用 NFS 存取计算机 B 上的文件吗？试解释原因。
- 31.4 在单台计算机上的三个服务器间会发生死锁吗？试解释原因。
- 31.5 对你单位中的服务器做实验，查看它们同时允许多少连接。
- 31.6 检查你的本地操作系统的配置。系统会首先用完 TCB、缓存和套接字吗？
- 31.7 如果让你选择调试一个死锁问题或一个活锁问题，你会选择哪一个？为什么？你将怎么做？
- 31.8 由于服务器接收连接请求的队列充满会发生故障，而网络中断也会发生故障，使用 TCP 的客户能区别这两种故障吗？

-
- 31.9 在 NFS 的一个实现中，完成远程文件安装（mount）的软件阻塞了——NFS 客户软件将阻塞，直到远程系统响应。为避免死锁，程序员使程序在安装另一台机器的文件系统前，先使用 ping（ICMP echo）判断另一台机器是否可用。在什么情况下仍会发生死锁？
- 31.10 网络课上的学生构造能分析网络流量的网络监视设备。两个学生决定使用 X Window 系统在彩色屏幕上显示结果。他俩也可单独运行他们的程序，但只要两个都开始运行，网络就会饱和。试解释其原因。

附录 1 系统调用与套接字使用的库例程

引言

在 Linux 中，通信是以套接字抽象为核心进行的。应用程序使用一组套接字系统调用与操作系统中的 TCP/IP 软件通信。客户应用程序创建套接字，将它连接到远程机器上的服务器，并使用它向远程机器传送数据或从远程机器接收数据。最后，当客户应用使用完套接字后，就把它关闭。服务器创建套接字，将它绑定到本地机上的熟知协议端口，并等待客户与之联系。

在本附录中，我们将逐一描述那些程序员在编写客户或服务器应用程序时所使用的系统调用或库例程。函数是按字母表顺序排列的，每一页描述一个函数。所列出的函数包括：accept、bind、close、connect、fork、gethostbyaddr、gethostbyname、gethostid、gethostname、getpeername、getprotobyname、getservbyname、getsockname、getsockopt、gettimeofday、listen、read、recv、recvfrom、recvmsg、select、send、sendmsg、sendto、sethostid、setsockopt、shutdown、socket 和 write。

其他版本的 UNIX 也有本附录所列出的这些函数。例如，如果套接字标记为非阻塞的，但有关调用却将阻塞时，就会发生错误，早期的 UNIX 版本用符号常量 EWOULDBLOCK 表示该错误，而在 Linux 中，则用符号常量 EAGAIN 表示同样的错误（强调过会儿调用同样的函数可能会成功）。为了获得向后兼容性，Linux 把常量 EWOULDBLOCK 的值定义成与 EAGAIN 相同。

accept 系统调用

用法

```
retcode = accept ( socket, addr, addrlen );
```

说明

服务器调用 `socket` 创建套接字，用 `bind` 指明本地 IP 地址和协议端口号，然后用 `listen` 使套接字处于被动状态（`passive`），并设置连接请求队列的长度。`accept` 从队列中取走下一个连接请求（或一直在那里等待下一个连接请求到达），为请求创建新套接字，并返回新套接字描述符。`accept` 只用于流套接字（如 TCP 套接字）。

参数

参数	类型	含义
<code>socket</code>	<code>int</code>	由 <code>socket</code> 函数创建的套接字描述符。
<code>addr</code>	<code>&sockaddr</code>	地址结构的指针。 <code>accept</code> 在该结构中填入远程机器的 IP 地址和协议端口号。
<code>addrlen</code>	<code>&int</code>	整数指针，初始指明为 <code>sockaddr</code> 参数的大小，当调用返回时，指明为存储在 <code>addr</code> 中的字节数。

返回码

`accept` 成功时返回非负套接字描述符，在发生差错时返回 -1。当发生差错时，全局变量 `errno` 含有如下值之一：

<code>errno</code> 的值	差错原因
<code>EBADF</code>	第一个参数未指明合法的描述符。
<code>ENOTSOCK</code>	第一个参数未指明套接字描述符。
<code>EOPNOTSUPP</code>	套接字类型不是 <code>SOCK_STREAM</code> 。
<code>EFAULT</code>	第二个参数中的指针非法。
<code>EAGAIN</code>	套接字被标记为非阻塞的，且没有正等待的连接（即该调用将阻塞，调用者可以过会儿再试）。
<code>EPERM</code>	防火墙规则禁止连接。
<code>ENOBUFS</code>	没有足够的缓存。
<code>ENOMEM</code>	没有足够的存储器。

bind 系统调用

用法

```
retcode = bind ( socket, localaddr, addrlen );
```

说明

`bind` 为套接字指明本地 IP 地址和协议端口号。`bind` 主要由服务器使用，它需要指明熟知协议端口。

参数

参数	类型	含义
<code>socket</code>	<code>int</code>	由 <code>socket</code> 调用创建的套接字描述符。
<code>localaddr</code>	<code>&sockaddr</code>	地址结构，指明 IP 地址和协议端口号。 <code>addrlen</code> 地址结构的字节数大小。
<code>addrlen</code>	<code>int</code>	以字节为单位的地址结构大小。

第 5 章含有 `sockaddr` 结构的描述。

返回码

`bind` 若成功就返回 0，返回 -1 就表示发生了差错。当差错发生时，全局变量 `errno` 含有代码，它指明差错的原因。可能的差错有：

<code>errno</code> 的值	差错原因
<code>EBADF</code>	参数 <code>socket</code> 未指明合法的描述符。
<code>ENOTSOCK</code>	参数 <code>socket</code> 未指明套接字描述符。
<code>EADDRNOTAVAIL</code>	指明的地址不可用（如 IP 地址与本地接口不匹配）。
<code>EADDRINUSE</code>	指明的地址正在使用（如另一个进程已分配了协议端口）。
<code>EINVAL</code>	套接字已绑定到一个地址上。
<code>EACCES</code>	不允许应用程序使用指明的地址。
<code>EFAULT</code>	参数 <code>localaddr</code> 中的指针无效。
<code>EROFS</code>	套接字索引节点应驻留在只读文件系统上。
<code>ENAMETOOLONG</code>	<code>Localaddr</code> 太长。
<code>ENOENT</code>	文件不存在。
<code>ENOMEM</code>	没有足够的内核存储器。
<code>ENOTDIR</code>	路径前缀的某个部分不是目录。
<code>ELOOP</code>	遇到太多的符号链。

close 系统调用

用法

```
retcode = close ( socket );
```

说明

应用程序使用完一个套接字后调用 `close`。`close` 从容地终止通信，并删除套接字。任何正在套接字上等待被读取的数据都将被丢弃。

实际上，Linux 实现了引用计数机制 (reference count mechanism)，它允许多个进程共享一个套接字。如果 n 个进程共享一个套接字，引用计数将为 n。`close` 每被进程调用一次，就将引用计数减 1。一旦引用计数减到零（即所有进程都已调用了 `close`），套接字将被释放。

参数

参数	类型	含义
socket	int	将被关闭的套接字描述符。

返回码

close 若成功就返回 0, 返回 -1 表示发生了差错。当差错发生时, 全局变量 errno 将含有以下值:

errno 的值	差错原因
EBADF	参数 socket 未指明合法的描述符。

connect 系统调用

用法

```
retcode = connect ( socket, addr, addrlen );
```

说明

connect 允许调用者为先前创建的套接字指明远程端点的地址。如果套接字使用 TCP, connect 就使用三次握手建立连接; 如果套接字使用 UDP, connect 仅指明远程端点, 但不向它传送任何数据报。

参数

参数	类型	含义
socket	int	套接字的描述符。
addr	&sockaddr_in	远程机器端点地址。
addrlen	int	第二个参数的长度。

第 5 章含有 sockaddr_in 结构的说明。

返回码

connect 若成功就返回 0, 返回 -1 表示发生了差错。当差错发生时, 全局变量 errno 含有如下值之一:

errno 的值	差错原因
EBADF	参数 socket 未指明合法的描述符。
ENOTSOCK	参数 socket 未指明套接字描述符。
EAFNOSUPPORT	远程端点指明的地址族不能与这种类型的套接字一起使用。
EADDRNOTAVAIL	指定的地址不可用。
EISCONN	套接字已被连接。
ETIMEDOUT	(只用于 TCP) 协议因未成功建立连接而超时。
ECONNREFUSED	(只用于 TCP) 连接被远程机器拒绝。
ENETUNREACH	(只用于 TCP) 网络当前不可达。

(续表)

errno 的值	差错原因
EADDRINUSE	指明的地址正在使用。
EINPROGRESS	(只用于 TCP) 套接字是非阻塞的，且连接尝试将被阻塞。
EALREADY	(只用于 TCP) 套接字是非阻塞的，且调用将等待前一个连接尝试完成。
EFAULT	套接字结构地址非法。
EACCES	用户试图连接到广播地址，但套接字的广播标记未使能。

fork 系统调用

用法

```
retcode = fork ( );
```

说明

虽然 fork 并不与通信套接字直接相关，但是由于服务器使用它创建并发的进程，因此它很重要。fork 创建一个新进程，执行与原进程相同的代码。两个进程共享在调用 fork 时已打开的所有套接字和文件描述符。两个进程有不同的进程标识符和不同的父进程标识符。

参数

fork 不带任何参数。

返回码

如果成功，fork 就给子进程返回 0，并给原进程返回新建进程的标识符（非零）。它返回 -1 表示发生了差错。当差错发生时，全局变量 errno 中含有如下值之一：

errno 的值	差错原因
EAGAIN	已达到了系统限制的进程总数，或已达到了对每个用户的进程限制。
ENOMEM	系统没有足够的内存用于新进程。

gethostbyaddr 库函数

用法

```
retcode = gethostbyaddr( addr, alen, atype );
```

说明

gethostbyaddr 搜索关于某个给定 IP 地址的主机的信息。

参数

参数	类型	含义
addr	&char	指向数组的指针，该数组含有一个主机地址（如IP地址）。
alen	int	整数，它给出地址长度（IP地址长度是4）。
atype	int	整数，它给出地址类型（IP地址的类型为AF_INET）。

返回码

gethostbyaddr如果成功，返回hostent结构的指针，如果发生差错，则返回0。

hostent结构声明如下：

```
struct hostent{           /* 主机项 */
    char *h_name;          /* 正式主机名 */
    char *h_aliases[];     /* 其他别名列表 */
    int h_addrtype;        /* 主机地址类型 */
    int h_length;           /* 主机地址长度 */
    char **h_addr_list;    /* 主机地址列表 */
};
```

当发生差错时，全局变量h_errno中含有下列值之一：

h_errno的值	差错原因
HOST_NOT_FOUND	不知道所指明的名字。
TRY AGAIN	暂时差错：本地服务器现在不能与授权机构联系。
NO_RECOVERY	发生了无法恢复的差错。
NO_ADDRESS	指明的名字有效，但它无法与某个IP地址对应。
NO_DATA	指明的名字有效，但它无法与某个IP地址对应。

gethostbyname 库调用

用法

```
retcode = gethostbyname(name);
```

说明

gethostbyname将主机名映射为IP地址。

参数

参数	类型	含义
name	&char	含有主机名的字符串的地址。

返回码

`gethostbyname` 如果成功就返回 `hostent` 结构的指针，如发生差错则返回 0。`hostent` 结构声明为：

```
struct hostent{ /* 主机项 */
    char *h_name; /* 正式主机名 */
    char *h_aliases[]; /* 其他别名列表 */
    int h_addrtype; /* 主机地址类型 */
    int h_length; /* 主机地址长度 */
    char **h_addr_list; /* 主机地址列表 */
};
```

当发生差错时，全局变量 `h_errno` 中含有下列值之一：

<code>h_errno</code> 的值	差错原因
<code>HOST_NOT_FOUND</code>	不知道所指明的名字。
<code>TRY AGAIN</code>	暂时差错：本地服务器现在不能与授权机构联系。
<code>NO_RECOVERY</code>	发生了无法恢复的差错。
<code>NO_ADDRESS</code>	指明的名字有效，但它无法与某个 IP 地址对应。
<code>NO_DATA</code>	指明的名字有效，但它无法与某个 IP 地址对应。

gethostid 系统调用

用法

```
hostid = gethostid();
```

说明

应用程序调用 `gethostid` 以获取指派给本地机器的惟一的 32 比特主机标识符。通常，主机标识符是机器的主（primary）IP 地址。

参数

`gethostid` 不带任何参数。

返回码

`gethostid` 返回含有主机标识符的长整数。

gethostname 系统调用

用法

```
retcode = gethostname ( name, namelen );
```

说明

`gethostname` 用文本字符串的形式返回本地机器的主 (primary) 名字。

参数

参数	类型	含义
<code>name</code>	<code>&char</code>	放置名字的字符数组的地址。
<code>namelen</code>	<code>int</code>	名字数组的长度 (至少应为 65)。

返回码

若 `gethostname` 成功则返回 0, 若发生差错则返回 -1。当发生差错时, 全局变量 `errno` 含有以下值:

<code>errno</code> 的值	差错原因
<code>EFAULT</code>	<code>name</code> 或 <code>namelen</code> 参数不正确。
<code>EINVAL</code>	<code>namelen</code> 是负数, 或者在 Linux/i386 上, <code>namelen</code> 比实际长度小。

getpeername 系统调用

用法

```
retcode = getpeername ( socket, remaddr, addrlen );
```

说明

应用程序使用 `getpeername` 获取已建立连接的套接字的远程端点地址。通常, 客户调用 `connect` 时设置了远程端点地址, 所以它知道远程地址。但是, 使用 `accept` 获得连接的服务器, 可能需要查询套接字来找出远程地址。

参数

参数	类型	含义
<code>socket</code>	<code>int</code>	由 <code>socket</code> 函数创建的套接字描述符。
<code>remaddr</code>	<code>&sockaddr</code>	含有对端地址的 <code>sockaddr</code> 结构的指针。
<code>addrlen</code>	<code>&int</code>	整数指针, 调用前, 该整数含有第二个参数的长度, 调用后该整数含有远程端点地址的实际长度。

第 5 章含有 `sockaddr` 结构的说明。

返回码

`getpeername` 如成功则返回 0, 如发生差错则返回 -1。一旦发生差错, 全局变量 `errno` 中含有如下值之一:

errno 的值	差错原因
EBADF	第一个参数未指明合法的描述符。
ENOTSOCK	第一个参数未指明套接字描述符。
ENOTCONN	套接字不是已建连的套接字。
ENOBUFS	系统没有足够的资源完成操作。
EFAULT	remaddr 参数指针无效。

getprotobynam 函数调用

用法

```
retcode = getprotobynam( name );
```

说明

应用程序调用 getprotobynam，以便根据协议名找到该协议的正式整数值。

参数

参数	类型	含义
name	&char	含有协议名的字符串的地址。

返回码

getprotobynam 若成功则返回 protoent 类型的结构指针，若发生差错则返回 0。结构 protoent 的声明如下：

```
struct protoent {           /* 协议的描述项 */
    char *p_name;          /* 协议的正式名 */
    char **p_aliases;       /* 协议的别名列表 */
    int p_proto;            /* 正式协议号 */
};
```

getservbyname 库调用

用法

```
retcode = getservbyname( name, proto );
```

说明

getservbyname 根据给出的服务名，从网络服务库中获取该服务的有关信息。客户和服务器都调用 getservbyname 将服务名映射为协议端口号。

参数

参数	类型	含义
name	&char	含有服务名的字符串的指针。
proto	&char	含有所用协议名的字符串的指针。

返回码

`getservbyname` 若成功则返回 `servent` 结构的指针，若发生差错则返回空指针（0）。`servent` 结构的声明如下：

```
struct servent {           /* 服务项 */
    char *s_name;          /* 正式服务名           */
    char **s_aliases;      /* 其他别名列表         */
    int   s_port;           /* 该服务使用的端口       */
    char *s_proto;          /* 服务所用协议         */
};
```

getsockname 系统调用

用法

```
retcode = getsockname( socket, name, namelen );
```

说明

`getsockname` 获得指明套接字的本地地址。

参数

参数	类型	含义
socket	int	由 <code>socket</code> 函数创建的描述符。
name	&sockaddr	含有 IP 地址和套接字协议端口号的结构的指针。
namelen	&int	结构中的位置数；返回时为结构大小。

返回码

`getsockname` 若成功则返回 0，若发生差错则返回 -1。一旦发生差错，全局变量 `errno` 中含有如下值之一：

<code>errno</code> 的值	差错原因
EBADF	第一个参数未指明合法的描述符。
ENOTSOCK	第一个参数未指明套接字描述符。
ENOBUFS	系统中没有足够的缓存空间可用。
EFAULT	<code>name</code> 或 <code>namelen</code> 的地址不正确。

getsockopt 系统调用

用法

```
retcode = getsockopt ( socket, level, opt, optval, optlen );
```

说明

getsockopt 允许应用程序获得某个套接字的参数（选项）值或该套接字所使用的协议。

参数

参数	类型	含义
socket	int	套接字描述符。
level	int	整数，它标识某个协议级。
opt	int	整数，它标识某个选项。
optval	&char	存放返回值的缓存地址。
optlen	&int	缓存大小；返回时为所发现的值的长度。

适用于所有套接字的套接字级（socket-level）选项包括：

SO_DEBUG	排错信息的状态
SO_REUSEADDR	允许本地地址重用吗？
SO_KEEPALIVE	连接保活（keep-alive）的状态
SO_DONTROUTE	忽略外发报文的选路
SO_LINGER	如果存在数据，延迟关闭吗？
SO_BROADCAST	允许传输广播报文吗？
SO_OOBINLINE	在带内接受带外数据吗？
SO_RCVLOWAT	在套接字层提供给应用程序使用之前，数据应具有大小（Linux 上为 1 字节）
SO_RCVTIMEO	接收超时（Linux 上不可用）
SO_PRIORITY	为所有发送的分组设置优先级
SO_SNDBUF	输出缓存大小
SO_RCVBUF	输入缓存大小
SO_TYPE	套接字的类型
SO_ERROR	获取并清除套接字的上一次差错

返回码

getsockopt 若成功则返回 0，若发生差错则返回 -1。一旦差错发生，全局变量 errno 中含有如下值之一：

errno 的值	差错原因
EBADF	第一个参数未指明合法的描述符。
ENOTSOCK	第一个参数未指明套接字描述符。
ENOPROTOOPT	opt 不正确。
EFAULT	optval 或 optlen 的地址不正确。

gettimeofday 系统调用

用法

```
retcode = gettimeofday( tm, tmzone );
```

说明

gettimeofday 从系统中提取当前时间和日期，以及有关本地时区的信息。

参数

参数	类型	含义
tm	&struct timeval	timeval 结构的地址。
tmzone	&struct timezone	timezone 结构的地址。

gettimeofday 指明的结构声明如下：

```
struct timeval{           /* 存储时间的结构          */
    long tv_sec;          /* 自纪元日期 (1/1/70) 以来的秒数 */
    long tv_usec;          /* 超过 tv_sec 的毫秒数      */
};

struct timezone {          /* timezone 信息结构          */
    int tz_minuteswest;   /* 格林尼治以西的分钟数      */
    int tz_dsttime;        /* 所用校正的类型            */
};
```

返回码

gettimeofday 若成功则返回 0，若发生差错则返回 -1。一旦发生差错，全局变量 errno 中含有如下值：

errno 的值	差错原因
EFAULT	tm 或 tmzone 参数含有不正确的地址。

listen 系统调用

用法

```
retcode = listen ( socket, queuelen );
```

说明

服务器使用 listen 使套接字处于被动状态（即准备接受传入请求）。在服务器处理某个请求时，协议软件应将后续收到的请求排队，listen 也设置排队的连接请求的数目。listen 只用于 TCP 套接字。

参数

参数	类型	含义
socket	int	由 socket 调用创建的套接字描述符。
queueLen	int	传入连接请求的队列大小（通常最大不超过 5）。

返回码

listen 若成功则返回 0，若发生差错则返回 -1。一旦出错，全局变量 errno 含有如下值之一：

errno 的值	差错原因
EBADF	第一个参数未指明合法的描述符。
ENOTSOCK	第一个参数未指明套接字描述符。
EOPNOTSUPP	套接字类型不支持 listen。

read 系统调用

用法

```
retcode=read ( socket, buff, buflen );
```

说明

客户或服务器使用 read 从套接字获取输入。

参数

参数	类型	含义
socket	int	由 socket 函数创建的套接字描述符。
buff	&char	存放输入字符的数组的指针。
buflen	int	整数，它指明 buff 数组中的字节数。

返回码

read 若检测到在套接字上遇到文件结束就返回 0，若它获得输入就返回读取的字节数，若发生差错则返回 -1。一旦出错，全局变量 errno 中含有如下值之一：

errno 的值	差错原因
EBADF	第一个参数未指明合法的描述符。
EFAULT	地址 buff 不合法。
EIO	在读数据时发生 I/O 错误。
EINTR	某个信号中断了操作。
EAGAIN	指明的是非阻塞 I/O，但套接字没有数据。
EINVAL	文件描述符不适于读。
EISDIR	文件描述符指向一个目录。

recv 系统调用

用法

```
retcode = recv ( socket, buffer, length, flags );
```

说明

recv 从套接字获取下一个传入报文。

参数

参数	类型	含义
socket	int	由 socket 函数创建的套接字描述符。
buffer	&char	存放报文的缓存的地址。
length	int	缓存的长度。
flags	int	控制比特，它指明是否接受带外数据和是否预览报文。

返回码

recv 若成功则返回报文中的字节数，若发生差错则返回 -1。一旦出错，全局变量 errno 中含有如下值之一：

errno 的值	差错原因
EBADF	第一个参数未指明合法的描述符。
ENOTSOCK	第一个参数未指明套接字描述符。
EAGAIN	套接字没有数据，但它已被指明为非阻塞 I/O。
EINTR	在读操作可传递数据前到达了信号。
EFAULT	参数 buffer 不正确。
ENOTCONN	套接字还未连接。
EINVAL	传递了非法参数。

recvfrom 系统调用

用法

```
retcode = recvfrom ( socket, buffer, buflen, flags, from, fromlen );
```

说明

recvfrom 从套接字获取下一个传入报文，并记录发送者的地址（允许调用者发送应答）。

参数

参数	类型	含义
socket	int	由 socket 函数创建的套接字描述符。
buffer	&char	存放报文的缓存的地址。

(续表)

参数	类型	含义
buflen	int	缓存的长度。
flags	int	控制比特，它指明是否接受带外数据和是否预览报文。
from	&sockaddr	存放发送方地址结构的地址。
fromlen	&int	缓存的长度，返回时为发送者地址的大小。

第 5 章中含有 sockaddr 结构的说明。

返回码

recvfrom 若成功便回报文中的字节数，若发生差错则返回 -1。一旦出错，全局变量 errno 中含有如下值之一：

errno 的值	差错原因
EBAADF	第一个参数未指明合法的描述符。
ENOTSOCK	第一个参数未指明套接字描述符。
EAGAIN	套接字没有数据，但已被指明为非阻塞 I/O。
EINTR	在读操作可传递数据前到达了信号。
EFAULT	参数 buffer 不正确。
ENOTCONN	套接字还未连接。
EINVAL	传递了非法参数。

recvmsg 系统调用

用法

```
retcode = recvmsg ( socket, msg, flags );
```

说明

recvmsg 返回套接字上到达的下一个报文。它将报文放入一个结构，该结构包括首部和数据。

参数

参数	类型	含义
socket	int	由 socket 函数创建的套接字描述符。
msg	&struct msghdr	报文结构的指针。
flags	int	控制比特，它指明是否接受带外数据和是否预览报文。

报文用 msghdr 结构传递，其格式如下：

```
struct msghdr {
    caddr_t    msg_name;      /* 可选的地址          */
    int        msg_namelen;   /* 地址的大小          */
    struct iovec *msg_iov;    /* 散列 / 紧凑数组    */
    int        msg_iovlen;    /* msg_iov 中的元素 # */
    caddr_t    msg_accrights; /* 发送 / 接受权限    */
};
```

```

    int msg_accrightslen;      /* 特权字段的长度 */ */
}

```

返回码

`recvmsg` 若成功便返回报文中的字节数，若发生差错则返回 -1。一旦出错，全局变量 `errno` 中含有如下值之一：

<code>errno</code> 的值	差错原因
<code>EBADF</code>	第一个参数未指明合法的描述符。
<code>ENOTSOCK</code>	第一个参数未指明套接字描述符。
<code>EAGAIN</code>	套接字没有数据，但已被指明为非阻塞 I/O。
<code>EINTR</code>	在读操作可传递数据前到达了信号。
<code>EFAULT</code>	参数 <code>msg</code> 不正确。
<code>ENOTCONN</code>	套接字还未连接。
<code>EINVAL</code>	传递了非法参数。

select 系统调用

用法

```
retcode = select ( numfds, refds, wrfds, exfds, time );
```

说明

`select` 提供异步 I/O，它允许单进程等待指明文件描述符集合中的任一描述符最先就绪。调用者也可指明等待超时的最大值。

参数

参数	类型	含义
<code>numfds</code>	<code>int</code>	集合中文件描述符的数目。
<code>refds</code>	<code>&fd_set</code>	用作输入的文件描述符的地址。
<code>wrfds</code>	<code>&fd_set</code>	用作输出的文件描述符的地址。
<code>exfds</code>	<code>&fd_set</code>	用作异常处理的文件描述符的地址。
<code>time</code>	<code>&struct timeval</code>	最大等待时间或零。

涉及描述符的参数由整数组成，而整数的第*i* 比特与描述符*i*相对应。宏 `FD_CLR` 和 `FD_SET` 清除或设置各个比特位。Linux 手册中，描述 `gettimeofday` 的页面中有对 `timeval` 结构的描述。

返回码

`select` 若成功则返回准备就绪的文件描述符数，若时间限制已到则返回 0，若发生差错则返回 -1。一旦出错，全局变量 `errno` 中含有如下值之一：

<code>errno</code> 的值	差错原因
<code>EBADF</code>	某个描述符集合指明了非法的描述符。
<code>EINTR</code>	在等待超时或任何被选择的描述符准备就绪以前，到达了信号。

(续表)

errno 的值	差错原因
ENOMEM	不能为内部表分配内存

send 系统调用

用法

```
retcode = send ( socket, msg, msglen, flags );
```

说明

应用程序调用 send 将报文传送到另一台机器。

参数

参数	类型	含义
socket	int	由 socket 函数创建的套接字描述符。
msg	&char	报文的指针。
msglen	int	报文的字节长度。
flags	int	控制比特，它指明是否接受带外数据和是否预览报文。

返回码

send 若成功就返回已发送的字符数，若发生差错则返回 -1。一旦出错，全局变量 errno 中含有如下值之一：

errno 的值	差错原因
EBADF	第一个参数未指明合法的描述符。
ENOTSOCK	第一个参数未指明套接字描述符。
EFAULT	参数 msg 不正确。
EMSGSIZE	报文对套接字而言太大了。
EAGAIN	套接字没有数据，但已被指明为非阻塞 I/O。
ENOBUFS	系统没有足够的资源完成操作。
EINTR	有信号发生。
ENOMEM	没有足够的存储器。
EINVAL	传递了非法参数。
EPIPE	本地端已经关闭了面向连接的套接字。

sendmsg 系统调用

用法

```
retcode = sendmsg ( socket, msg, flags );
```

说明

`sendmsg` 从 `msghdr` 结构中提取报文并发送。

参数

参数	类型	含义
<code>socket</code>	<code>int</code>	由 <code>socket</code> 函数创建的套接字描述符。
<code>msg</code>	<code>&struct msghdr</code>	报文结构的指针。
<code>flags</code>	<code>int</code>	控制比特，它指明是否接受带外数据和是否预览报文。

`msghdr` 结构的说明见 `recvmsg` 的说明页。

返回码

`sendmsg` 若成功便返回已发送的字节数，若发生差错则返回 -1。一旦出错，全局变量 `errno` 中含有如下值之一：

<code>errno</code> 的值	差错原因
<code>EBADF</code>	第一个参数未指明合法的描述符。
<code>ENOTSOCK</code>	第一个参数未指明套接字描述符。
<code>EFAULT</code>	参数 <code>msg</code> 不正确。
<code>EMSGSIZE</code>	报文对套接字而言太大了。
<code>EAGAIN</code>	套接字没有数据，但已被指明为非阻塞 I/O。
<code>ENOBUFS</code>	系统没有足够的资源完成操作。
<code>EINTR</code>	有信号发生。
<code>ENOMEM</code>	没有足够的存储器。
<code>EINVAL</code>	传递了非法参数。
<code>EPIPE</code>	本地端已经关闭了面向连接的套接字。

sendto 系统调用

用法

```
retcode = sendto ( socket, msg, msglen, flags, to, tolen );
```

说明

`sendto` 从一个结构中获取目的地址，然后发送报文。

参数

参数	类型	含义
<code>socket</code>	<code>int</code>	由 <code>socket</code> 函数创建的套接字描述符。
<code>msg</code>	<code>&char</code>	报文的指针。
<code>msglen</code>	<code>int</code>	报文的字节长度。
<code>flags</code>	<code>int</code>	控制比特，它指明是否接受带外数据和是否预览报文。
<code>to</code>	<code>&sockaddr</code>	地址结构的指针。
<code>tolen</code>	<code>&int</code>	地址的字节长度。

第 5 章中含有 sockaddr 结构的说明。

返回码

sendto 若成功就返回已发送的字节数，若发生差错则返回 -1。一旦出错，全局变量 errno 中含有如下值之一：

errno 的值	差错原因
EBADF	第一个参数未指明合法的描述符。
ENOTSOCK	第一个参数未指明套接字描述符。
EFAULT	参数 msg 不正确。
EMSGSIZE	报文对套接字而言太大了。
EAGAIN	套接字没有数据，但已被指明为非阻塞 I/O。
ENOBUFS	系统没有足够的资源完成操作。
EINTR	有信号发生。
ENOMEM	没有足够的存储器。
EINVAL	传递了非法参数。
EPIPE	本地端已经关闭了面向连接的套接字。

sethostid 系统调用

用法

```
(void) sethostid (hostid);
```

说明

系统管理员在系统启动时运行有特权的程序，该程序调用 sethostid 为本地机器指派惟一的 32 比特主机标识符。通常，主机标识符是机器的主（primary）IP 地址。

参数

参数	类型	含义
hostid	int	被保存的作为主机标识符的值。

差错

应用程序必须有根（root）权限，否则 sethostid 不会改变主机的标识符。

setsockopt 系统调用

用法

```
retcode = setsockopt ( socket, level, opt, optval, optlen );
```

说明

`setsockopt` 允许应用程序改变套接字的选项或它所使用的协议。

参数

参数	类型	含义
socket	int	套接字描述符。
level	int	标识某个协议的整数（如 TCP）。
opt	int	标识某个选项的整数。
optval	&char	存放选项值的缓存地址（通常，1 代表允许选项，0 代表禁止选项）。
optlen	int	optval 的长度。

用于所有套接字的套接字级（socket-level）选项包括：

SO_DEBUG	允许/禁止排错信息的状态
SO_REUSEADDR	允许/禁止本地地址重用
SO_KEEPALIVE	允许/禁止连接状态保活（keep-alive）
SO_DONTROUTE	允许/禁止忽略出报文的选路
SO_LINGER	如果存在数据，则延迟关闭
SO_BROADCAST	允许/禁止传输广播报文
SO_OOBINLINE	允许/禁止在带内接受带外数据
SO_SNDBUF	设置输出缓存大小
SO_RCVBUF	设置输入缓存大小
SO_SNDLOWAT	指明数据在传递给运输层之前，需要放在缓存中的最大的字节数
SO_BINDTODEVICE	将套接字与某个设备接口绑定
SO_PRIORITY	指明从套接字发出的分组的优先级

返回码

`setsockopt` 若成功就返回 0，若发生差错则返回 -1。一旦差错发生，全局变量 `errno` 中含有如下值之一：

errno 的值	差错原因
EBADF	第一个参数未指明合法的描述符。
ENOTSOCK	第一个参数未指明差错描述符。
ENOPROTOOPT	选项整数 opt 不正确。
EFAULT	optval 的地址或 optlen 不正确。

shutdown 系统调用

用法

```
retcode = shutdown ( socket, direction );
```

说明

`shutdown` 函数用于全双工的套接字（即已建连的 TCP 套接字），并且用于部分关闭连接。

参数

参数	类型	含义
socket	int	由 <code>socket</code> 调用创建的套接字描述符。
direction	int	<code>shutdown</code> 需要的方向：0 表示终止进一步输入，1 表示终止进一步输出，2 表示终止输入和输出。

返回码

`shutdown` 调用若操作成功则返回 0，若发生差错则返回 -1。一旦出错，全局变量 `errno` 中含有指出差错原因的代码。可能的差错是：

<code>errno</code> 的值	差错原因
EBADF	第一个参数未指明合法的描述符。
ENOTSOCK	第一个参数未指明套接字描述符。
ENOTCONN	指明的套接字当前未建连。

socket 系统调用

用法

```
retcode =socket( family, type, protocol );
```

说明

`socket` 函数创建用于网络通信的套接字，并返回该套接字的整数描述符。

参数

参数	类型	含义
family	int	协议或地址族（对于 TCP/IP 为 PF_INET，也可使用 AF_INET）。
type	int	服务的类型（对于 TCP 为 SOCK_STREAM，对于 UDP SOCK_DGRAM）。
protocol	int	使用的协议号，或用 0 表示给定族和类型的默认协议号。

返回码

`socket` 调用或者返回描述符，或者返回 -1 表示发生了差错。一旦出错，全局变量 `errno` 中含有指出差错原因的代码。可能的差错是：

<code>errno</code> 的值	差错原因
EPROTONOSUPPORT	参数中的错误：申请的服务或指明协议无效。
EMFILE	应用程序的描述符表已满。
ENFILE	内部的系统文件表已满。

(续表)

errno 的值	差错原因
EACCES	拒绝创建套接字。
ENOBUFS	系统没有可用的缓存空间。
ENOMEM	系统没有可用的存储器。
EINVAL	未知或不可用的协议或协议族。

write 系统调用

用法

```
retcode = write ( socket, buf, buflen );
```

说明

write 允许应用程序传递数据给远程的机器。

参数

参数	类型	含义
socket	int	由 socket 的函数创建的套接字描述符。
buf	&char	含有数据的缓存的地址。
buflen	int	buf 中的字节数。

返回码

write 若成功则返回所传送的字节数，若发生差错则返回 -1。一旦出错，全局变量 errno 中含有如下值之一：

errno 的值	差错原因
EBADF	第一个参数未指明合法的描述符。
EPIPE	试图向未连接的流套接字上写。
EFAULT	buf 中的地址不合法。
EINVAL	套接字指针无效。
EIO	发生了 I/O 错误。
EAGAIN	套接字不能无阻塞地接受写入的所有数据，但已指明了非阻塞 I/O。
EINTR	在数据被写入之前，调用被某个信号中断。
ENOSPC	含有文件的设备没有存放数据的空间了。

附录 2 Linux 文件和套接字描述符的操作

引言

在Linux中，所有输入和输出操作都使用一种被称为文件描述符的抽象。程序调用open系统调用获得对某个文件的存取，或者调用socket系统调用获得用于网络通信的描述符。第4章和第5章描述了套接字接口；第24章更详细地描述了Linux描述符和I/O，并且指出，新创建的子进程将继承父进程在创建子进程时已打开的所有文件描述符的副本。最后，第30章探讨了实际服务器如何关闭额外的文件描述符，以及如何打开标准的I/O描述符。

本附录将描述程序如何将标准的I/O描述符用作参数，并将说明父进程在激活子进程前，如何重新安排现有的描述符，使它们与标准I/O描述符相一致。多服务的服务器要调用多个独立的程序处理各个服务，这一技术对它们尤其有用。

把描述符作为隐含参数

当fork函数创建新进程时，新创建的子进程将继承父进程在调用时已打开的所有文件描述符的副本。而且子进程的某个文件或套接字的描述符似乎与父进程的描述符处于同一位置。因此，如果父进程中的描述符5对应一个TCP套接字，新创建的子进程中的描述符5将对应同一个套接字。

在调用execve后，描述符还将保持打开。为执行文件F中的代码而创建一个新进程，进程要调用fork，接着让子进程以文件名F作为参数调用execve。

在概念上，文件描述符构成了新创建进程的隐含参数，当来自某个文件的代码覆盖了正在运行的程序后，这些描述符也构成了该进程的隐含参数。在调用fork前，父进程可选择哪些描述符保持打开状态，哪些需要关闭，而且，还可确切地选择在调用execve后，哪些描述符将保持打开状态。

Linux程序通常使用描述符代替显式参数来控制处理。具体来说，Linux程序在程序开始前期望有三个标准的I/O描述符：标准输入（描述符0）、标准输出（描述符1）和标准错误（描述符2）。程序从描述符0读取输入，输出写入描述符1，并发送差错报文到描述符2。

使用固定的描述符

如果服务器调用一个单独的程序来处理某个请求，它也可将描述符作为隐含参数。例如，某个从程序是面向连接的服务器的一部分，它可这样编写——期望用描述符0对应一个TCP连接。主服务器在描述符0上建立连接，然后使用execve运行从程序。我们也可这样编写主服务器，让它可使用任意描述符，并向从程序传递一个参数，该参数指定连接所对应的描述符。使用固定描述符简化了代码，且没有牺牲任何功能。总之：

进程在调用execve时将描述符用作隐含参数。主服务器通常用单个描述符作为隐含参数，供主服务器所调用的从程序使用。

重新安排描述符的必要性

如果主服务器调用socket系统函数创建用于无连接通信的套接字，或者调用accept获得面向连接通信用的套接字，在这种情况下，服务器不能指明调用所返回的描述符，因为描述符是由操作系统选择的。如果从程序期望描述符0对应通信所用的套接字，主程序不能只在某个任意的描述符上创建套接字，然后运行从程序。它必须在发出调用前重新安排它的各个描述符。

重新安排描述符

进程使用Linux系统调用close和dup2来重新安排其描述符。关闭描述符将切断它与相应的文件或套接字的联系，释放相关资源，并使得它能被重用。例如，如果进程需要使用描述符0，但它又正被使用，进程就将调用close使它成为空闲的。附录1描述了服务器在开始时如何关闭不必要的文件描述符。

- 进程调用函数dup2为某个文件描述符创建另一个副本。该调用有两个参数：

```
(void) dup2 (olddesc, newdesc);
```

此处的olddesc是整数，它识别现有描述符。newdesc也是整数，它识别描述符副本。如果newdesc正被使用，系统就释放它，好像用户在复制olddesc之前就已调用了close一样。完成dup2后，olddesc和newdesc都指向同一个对象（如同一个TCP连接）。

为使套接字从一个描述符“移动”到另一个描述符，程序首先调用dup2，然后调用close释放最初的那个描述符。例如，假设主服务器调用accept获得来的连接，并假设accept选用描述符5用于新连接。主服务器可发出以下两个调用，使socket移到描述符0：

```
(void) dup2 (5, 0);
(void) close (5);
```

close-on-exec

服务器在执行另一个程序前，可关闭不需要的文件描述符。为此，一种方法是服务器为每个不需要的文件描述符显式地调用close。另一种方法是使用系统设施自动关闭描述符。

为使用自动设施，服务器必须在每个希望系统关闭的描述符中设置close-on-exec标志。当服务器调用exec时，系统检查每个描述符，查看是否已设置该标志，如果标志已设置，就自动调用close。表面上自动关闭似乎并未使编程更容易，但必须记住，在大型的、复杂的程序中，在调用exec的程序处，每个描述符的目的可能并不明显。使用close-on-exec允许程序员在创建描述符的程序处，决定描述符是否应该保持打开状态。因此，它是很有用的。

小结

进程在创建新进程或用文件中的代码覆盖正在运行的程序时，使用描述符作为隐含参数。通常，服务器在运行分开编译的程序时，依赖文件描述符传递 TCP 连接或 UDP 套接字。

为使程序更易于编写和维护，程序员通常为每个隐含参数选择固定的描述符。调用者必须在调用 `execve` 前重新安排其描述符，以便使那些用于 I/O 的套接字与被选作隐含参数的描述符相对应。

进程使用 `dup2` 和 `close` 系统调用重新安排其描述符。`dup2` 将现有的描述符复制到一个指定的位置，而 `close` 释放指定的描述符。为移动描述符 A 到描述符 B，进程应首先调用 `dup2(A, B)`，然后调用 `close(A)`。

深入研究

Linux 联机文档含有关于 `dup2` 和 `close` 系统调用的详细信息。

习题

- A2.1 阅读有关 Linux 命令解释器的内容。执行命令之前它指派的是哪一个描述符？
- A2.2 简述命令解释器用于处理文件重定向的算法。说明如何在调用 `execve` 前移动文件描述符。
- A2.3 修改第 15 章中的多服务服务器，让它使用描述符 0 作为隐含参数，并调用 `execve` 执行从程序。
- A2.4 阅读有关程序 `inetd` 的内容。它如何将描述符用作隐含参数？
- A2.5 如果 $n = m$ ，调用 `dup2(n, m)` 会如何动作？编写可用于任意 n 和 m 值的 `dup2` 的一个版本。

参 考 文 献

- ABRAMSON, N. [1970], The ALOHA System - Another Alternative for Computer Communications, *Proceedings of the Fall Joint Computer Conference*.
- ANDREWS, D. W., and G. D. SHULTZ [1982], A Token-Ring Architecture for Local Area Networks: An Update, *Proceedings of Fall 82 COMPCON*, IEEE.
- ATALLAH, M., and D. E. COMER [June 1998], Algorithms for Variable Length Subnet Address Assignment, *IEEE Transactions on Computers*, vol. 47:6, 693-699.
- 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, Upper Saddle River, 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, Upper Saddle River, 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.
- 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. [August 1986], VMTP: A Transport Protocol for the Next Generation of Communication Systems, *Proceedings of ACM SIGCOMM '86*, 406-415.
- 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., and W. FANG [August 1998], Explicit Allocation Of Best-Effort Packet Delivery Service, *IEEE/ACM Transactions On Networking*, 6(4).
- 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., [1999], *Computer Networks And Internets*, 2nd edition, Prentice-Hall, Upper Saddle River, New Jersey.
- COMER, D. E. and J. T. KORB [1983], CSNET Protocol Software: The IP-to-X25 Interface, *Computer Communications Review*, 13(2).
- 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. and D. L. STEVENS [1999], *Internetworking With TCP/IP: Volume II: Design, Implementation, and Internals*, 3rd edition, Prentice-Hall, Upper Saddle River, 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, Upper Saddle River, 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, Upper Saddle River, 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, Upper Saddle River, New Jersey.

- COTTON, I. [1979], Technologies for Local Area Computer Networks, *Proceedings of the Local Area Communications Network Symposium*.
- 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, Upper Saddle River, 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*.
- DIION, 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*.
- 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, Upper Saddle River, 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.
- 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.
- 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*.
- HINDEN, R., J. HAVERTY, and A. SHELTER [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.

- JAIN, R. [1994], *FDDI Handbook; High-Speed Networking Using Fiber and Other Media*, Addison Wesley, Reading, Massachusetts.
- JENNINGS, D. M., L. H. LANDWEBER, and I. H. FUCHS [February 28, 1986], Computer Networking for Scientists and Engineers, *Science* vol 231, 941-950.
- JOUELAS, I., V. HARDMAN, and A. WATSON [November 1996], Lip Synchronisation for use over the Internet: Analysis and implementation, *Proceedings IEEE '96*.
- 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*.
- KAUFMAN, C., PERLMAN, R., and SPECINER, M. [1995], *Network Security: Private Communication in a Public World*, Prentice-Hall, Upper Saddle River, New Jersey.
- KENT, C., and J. MOGUL [August 1987], Fragmentation Considered Harmful, *Proceedings of ACM SIGCOMM '87*.
- LAMPSON, B. W., M. PAUL, and H. J. SIEGERT (EDS.) [1981], *Distributed Systems - Architecture and Implementation (An Advanced Course)*, Springer-Verlag, Berlin.
- LAMPSON, B. W., V. SRINIVASAN, and G. VARGHESE [June 1999], IP Lookups Using Multiway and Multicolumn Search, *IEEE/ACM Transactions on Networking*, vol 7, 324-334.
- 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.
- 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.
- 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.
- MILLS, D., and H-W. BRAUN [August 1987], The NSFNET Backbone Network, *Proceedings of ACM SIGCOMM '87*.
- 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.
- NEWMAN, P., G. MINSHALL, and T. L. LYON [April 1998], IP Switching — ATM Under IP, *IEEE Transactions on Networking*, Vol. 6:2, 117-129.
- 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 Satellite Telecommunications*, Prentice-Hall, Upper Saddle River, New Jersey.
- PERLMAN, R. [2000], *Interconnections: Bridges and Routers*, 2nd edition, 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, [1999], *Computer Networks: A Systems Approach*, 2nd edition, 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., and J. C. HOSKINS [October 1986], Notable Computer Networks, *Communications of the ACM*, 29(10).
- RAMAKRISHNAN, K. and R. JAIN [May 1990], A Binary Feedback Scheme For Congestion Avoidance In Computer Networks, *ACM Transactions on Computer Systems*, 8(2), 158-181.
- 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. [1993], *The Internet Message: Closing The Book with Electronic Mail*, Prentice-Hall, Upper Saddle River, New Jersey.
- 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.
- 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*.
- SOLOMON, J. [1997], *Mobile IP: The Internet Unplugged*, Prentice-Hall, Upper Saddle River, New Jersey.
- SOLOMON, M., L. LANDWEBER, and D. NEUHEGEN [1982], The CSNET Name Server, *Computer Networks* (6), 161-172.
- SRINIVASAN, V., and G. VARGHESE [February 1999], Fast Address Lookups Using Controlled Prefix Expansion, *ACM Transactions on Computer Systems*, vol. 17, 1-40.
- STALLINGS, W. [1997], *Local and Metropolitan Area Networks*, Prentice-Hall, Upper Saddle River, New Jersey.
- STALLINGS, W. [1998], *High-Speed Networks: TCP/IP and ATM Design Principles*, Prentice-Hall, Upper Saddle River, New Jersey.
- STEVENS, W. R. [1998], *UNIX Network Programming*, 2nd edition, Prentice-Hall, Upper Saddle River, 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.
- 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.
- 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 XESIS 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*.

[General Information]

书名=用TCP/IP进行网际互联第三卷：客户——服务器编程与应用：Linux/POSIX套接字版

作者=

页数=431

S S号=10439636

出版日期=